
McMule

Release v0.5.1














The McMule Collaboration

Feb 12, 2024

Contents:

| | | |
|----------|---|-----------|
| 1 | Getting started | 3 |
| 1.1 | Obtaining the code | 3 |
| 1.2 | Simple runs at LO | 4 |
| 1.3 | Running at NLO and beyond | 10 |
| 1.4 | More complicated runs | 15 |
| 2 | Structure of McMule | 21 |
| 2.1 | The user function | 21 |
| 2.2 | Modular structure of libmcmule | 22 |
| 2.3 | What happens when running | 23 |
| 3 | Observable validator | 25 |
| 3.1 | Using the validator | 26 |
| 3.2 | Logging Levels | 28 |
| 3.3 | Measurement function tests in detail | 28 |
| 3.4 | Distance function tests in detail | 37 |
| 4 | General aspects of using McMule | 39 |
| 4.1 | Flavour | 39 |
| 4.2 | Statistics | 40 |
| 4.3 | Analysis | 41 |
| 4.4 | Manual compilation | 42 |
| 5 | The menu file and pymule batch | 45 |
| 5.1 | Menu file syntax | 45 |
| 5.2 | Configuration file | 46 |
| 6 | libmcmule user guide | 47 |
| 6.1 | Parameters, variables, and functions of libmcmule | 47 |
| 6.2 | The user file in Fortran | 57 |
| 6.3 | The user file in C/C++ | 59 |
| 7 | pymule user guide | 61 |
| 7.1 | Working with files | 61 |
| 7.2 | Working with errors | 65 |
| 7.3 | Plotting | 71 |
| 8 | Technical aspects of McMule | 75 |
| 8.1 | Particle string framework | 75 |
| 8.2 | Phase-space generation | 75 |
| 8.3 | Implementation of FKS schemes | 78 |
| 8.4 | Calling procedures and function pointers | 80 |
| 8.5 | Optional parameters for integrands | 80 |

| | | |
|-----------|---|------------|
| 8.6 | Random number generation | 82 |
| 8.7 | Differential distributions and intermediary state files | 83 |
| 8.8 | Basics of containerisation | 84 |
| 9 | Implementing new processes in McMule | 87 |
| 9.1 | Creating a new process group | 92 |
| 9.2 | Study of ξ_c dependence | 94 |
| 9.3 | Example calculations in Mathematica | 95 |
| 9.4 | Coding style and best practice | 96 |
| 10 | The FKS² scheme | 99 |
| 10.1 | FKS ^ℓ : extension to N ^ℓ LO | 100 |
| 11 | Glossary | 101 |
| 11.1 | Acronyms | 101 |
| 11.2 | Technical terms | 102 |
| 12 | Bibliography | 105 |
| 13 | Particle ID | 107 |
| 14 | Available processes and which_piece | 113 |
| 15 | libmcmule reference guide | 115 |
| 15.1 | The particle framework | 115 |
| 15.2 | Matrix element interface | 117 |
| 15.3 | Package-X function | 117 |
| 15.4 | VP functions | 118 |
| 15.5 | Phase spaces | 118 |
| 16 | pymule reference guide | 127 |
| 16.1 | Working with errors | 127 |
| 16.2 | Working with abstract records | 134 |
| 16.3 | Working with vegas records | 137 |
| 16.4 | Working with records of data | 140 |
| 16.5 | Working with ξ_c data | 148 |
| 16.6 | Working with plots | 150 |
| 16.7 | Useful other functions | 154 |
| 16.8 | Validator | 154 |
| 17 | Indices and tables | 169 |
| | Bibliography | 171 |
| | Python Module Index | 173 |
| | Index | 175 |

Yannick Ulrich ¹, Pulak Banerjee ², Antonio Coutinho ³, Tim Engel ⁴, Andrea Gurgone ^{5,6}, Franziska Hagelstein ^{7,8}, Sophie Kollatzsch ^{8,9}, Luca Naterop ^{8,9}, Marco Rocco ⁸, Nicolas Schalch ¹, Vladyslava Sharkovska ^{8,9}, Adrian Signer ^{8,9}, Zach McBrearty ¹⁰

¹ Albert Einstein Center for Fundamental Physics, Universität Bern, CH-3012 Bern, Switzerland

² Department of Physics, Indian Institute of Technology Guwahati, Guwahati-781039, Assam, India

³ IFIC, Universitat de València - CSIC, Parc Científic, Catedrático José Beltrán, 2, E-46980 Paterna, Spain

⁴ Albert-Ludwigs-Universität Freiburg, Physikalisches Institut, Hermann-Herder-Straße 3, D-79104 Freiburg, Germany

⁵ Dipartimento di Fisica, Università di Pavia, I-27100 Pavia, Italy

⁶ INFN, Sezione di Pavia, I-27100 Pavia, Italy

⁷ Institut für Kernphysik & PRISMA⁺ Cluster of Excellence, Johannes Gutenberg Universität Mainz, D-55099 Mainz, Germany

⁸ Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland

⁹ Physik-Institut, Universität Zürich, Winterthurerstrasse 190, CH-8057 Zürich, Switzerland

¹⁰ Institute for Particle Physics Phenomenology, Durham, DH1 3LE, U.K

Chapter 1

Getting started

McMule is written in Fortran 95 with helper and analysis tools written in python. This guide will help you to get started using McMule by describing in detail how to calculate the NLO corrections to $\tau \rightarrow [\nu\bar{\nu}]e\gamma$. Since the neutrinos are not detected, we average over them, indicated by the brackets. Hence, we have to be fully inclusive w.r.t. the neutrinos. Still, the code allows to make any cut on the other final-state particles. As we will see, the *BR* for this process, as measured by BaBar [14, 18] has a discrepancy of more than 3σ from the *SM* value. This will illustrate the importance of fully differential *NLO* corrections in QED.

1.1 Obtaining the code

McMule is distributed multiple ways

- as a precompiled executable for recent-ish GNU Linux distributions. To be precise, your version of glibc needs to be newer than 2.17. The currently supported versions of most popular distributions (CentOS, Debian, Ubuntu, Fedora, RHEL) should be fine.
- as a Docker *image* that can be used on any platform.
- as the source code on [Gitlab](#) that can be compiled by the user. This contains the current release in the default release branch as well as the developer preview (devel).

Here we will focus on the first method as it is by far the easiest. For developers and tinkerers, we refer to Section [Manual compilation](#) on how to compile the code yourself.

First, obtain the McMule distribution from [our website](#) and extract the tarball

```
$ tar xzvf mcmule-release.tar.gz
mcmule-release/mcmule
mcmule-release/mcmule.mod
```

That's it. You can, if you want, install McMule to use it from any directory with the following commands but this is not required

```
$ cp mcmule-release/mcmule /usr/local/bin
$ cp mcmule-release/mcmule.mod /usr/local/include
```

To make use of McMule results, we also require the pymule python package It can be used from the `tools/` folder of the McMule repository but it is recommended that the user installs it

```
$ pip3 install git+https://gitlab.com/mule-tools/pymule.git
```

1.2 Simple runs at LO

1.2.1 Setting McMule up

In this example we want to compute two distributions, the invariant mass of the $e\gamma$ pair, $m_{\gamma e} \equiv \sqrt{(p_e + p_\gamma)^2}$, and the energy of the electron, E_e , in the rest frame of the tau. To avoid an *IR* singularity in the *BR*, we have to require a minimum energy of the photon. We choose this to be $E_\gamma \geq 10$ MeV as used in [14, 18].

At first, we need to find out how the process $\tau \rightarrow \nu\bar{\nu}e\gamma$ is implemented in McMule. For this, we refer to the table in Section *Available processes and which_piece* that specifies the pieces (sometimes called *which_piece*) that is required for a *generic process*. The generic process is a prototype for the physical process such as $\ell \rightarrow \nu\bar{\nu}\ell'\gamma$ where the flavour of the lepton ℓ is left open. In our case, we need to consider the row for $\mu \rightarrow \nu\bar{\nu}e\gamma$. Since we are only interested in *LO*, the only *which_piece* we need is `m2enng0`. To change from the generic process $\mu \rightarrow \nu\bar{\nu}e\gamma$ to the process we are actually interested in, $\tau \rightarrow \nu\bar{\nu}e\gamma$, we pick the *flavour* `tau-e` which refers to a $\tau \rightarrow e\cdots$ transition. Other options would be `tau-mu` for $\tau \rightarrow \mu\cdots$ or `mu-e` for $\mu \rightarrow e\cdots$.

Next, we need to find out which particle ordering is used in McMule for this piece, i.e. which variable will contain eg. the electron momentum. This is called the particle identification or *PID*. We can refer to the table in Section *Particle ID* to find that for the *which_piece* `m2enng0`, we have

$$\mu^-(p_1) \rightarrow e^-(p_2) [\bar{\nu}_e \nu_\mu](p_3, p_4) \gamma(p_5)$$

We can now implement our observables. For this, we need to define a `user.f95` file. An empty template can be found on [our website](#). We can use this file to implement the *measurement function* we want to calculate, i.e. which distributions and cuts we want to apply. We can further add some code that will execute at the beginning of the Monte Carlo run (allowing us eg. to further configure our calculation) and for each event (to simulate beam spread).

We begin by specifying the metadata of our histograms: we want two histograms (`nr_q = 2`) with 90 bins each (`nr_bins = 90`). The ranges should be $0 < m_{\gamma e} < 1800$ MeV and $0 \leq E_e \leq 900$ MeV.

Listing 1.1: The metadata for our calculation with two histograms (`nr_q = 2`) with 90 bins each (`nr_bins = 90`). The ranges should be $0 < m_{\gamma e} < 1800$ MeV and $0 \leq E_e \leq 900$ MeV.

```
10 integer, parameter :: nrq = 2
11 integer, parameter :: nrbins = 90
12 real(kind=prec) :: &
13     min_val(nrq) = (/ 0., 0. /)
14 real(kind=prec) :: &
15     max_val(nrq) = (/ 1800., 900. /)
```

Note: Finding suitable values for the ranges can be tricky beyond *LO* and usually requires a few test runs. Since all histograms have the same number of bins, one is often forced to have empty bins to ensure ‘nice’ bin widths.

We can now define the actual *measurement function* called `quant()`. We need to

- calculate the invariant mass of the $e\gamma$ pair. This is done using the momentum-squaring function `sq()`. The result is store in the first distribution, `quant(1)`.

- store the electron energy in `quant(2)`. Since this is frame-dependent, we need to know that McMule generates the particles in the tau rest frame. However, in general it is better to boost into that frame. Further, McMule stores momenta as $(/px, py, pz, E/)$, meaning the energy is `q2(4)`.
- cut on the photon energy `q5(4)`. The variable `pass_cut` controls the cuts. Initially it is set to `.true.`, to indicate that the event is kept. Applying a cut amounts to setting `pass_cut` to `.false.`.

Listing 1.2: The *measurement function* at LO

```

63 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
64
65 real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
66 real (kind=prec) :: quant(nr_q)
67 !! ==== keep the line below in any case ==== !!
68 call fix_mu
69
70 pass_cut = .true.
71 if(q5(4) < 10._prec) pass_cut = .false.
72
73 names(1) = 'minv'
74 quant(1) = sqrt(sq(q2+q5))
75
76 names(2) = 'Ee'
77 quant(2) = q2(4)
78
79 END FUNCTION QUANT

```

Additionally to the numeric value in `quant(i)` we store a human-readable name in `names`.

Warning: The maximal length of these names is defined in the variable `namesLen` which defaults to 6 characters. Also note that this *measurement function* is not *IR*-safe!

Finally, we need to specify the masses of the particles involved. This is called the *flavour* in McMule. We do this by defining a routine called `inituser()` which calls the McMule function `initflavour()`. Additionally, we also use this for some self-documentation

Listing 1.3: The *measurement function* at LO

```

55 SUBROUTINE INITUSER
56 print*, "Calculating tau->e nu nu gamma at LO"
57 call initflavour("tau-e")
58
59 ! Let the tau be unpolarised
60 poll = (/ 0._prec, 0._prec, 0._prec, 0._prec /)
61 END SUBROUTINE

```

This sets the *flavour* variable is to tau-e to change from the generic process $\mu \rightarrow \nu\bar{\nu}e\gamma$ to the process we are actually interested in, $\tau \rightarrow \nu\bar{\nu}e\gamma$.

We now need to compile our observable into a *shared library* so that McMule can load it. To do this, we run

```
$ gfortran -fPIC --shared -o user.so user.f95
```

This requires the `mcmule.mod` file to either be in the current directory or installed somewhere the compiler can find it.

Otherwise, one needs to add the corresponding flag

```
$ gfortran -I/path/to/the/folder/of/mcmule.mod/ -fPIC --shared -o user.so user.f95
```

Warning: A version of the `mcmule.mod` header file is distributed as part of McMule. If you are using a copy of GFortran prior to version 8, this means you will have to regenerate the header file manually. To do this, you can use the `build-header.sh` script.

1.2.2 Running McMule manually

Now the mule is ready to trot. For quick and dirty runs of McMule, the easiest way is to just execute the `mcmule` binary in the same directory as the `user.so` file and input the configuration by hand. However, since this is not how the code is meant to be used, it will not prompt the user but just expect the correct input.

We now need to choose the statistics we want. For this example, we pick 10 iterations with 1000×10^3 points each for pre-conditioning and 50 iterations with 1000×10^3 points each for the actual numerical evaluation (cf. Section `sec_stat` for some heuristics to determine the statistics needed). We pick a *random seed* between 0 and $2^{31} - 1$ (cf. Section *Random number generation*), say 70998, and for the input variable *which_piece* we enter `m2enng0` as discussed above. This system is used for other processes as well. The input variable *which_piece* determines the generic process and the part of it that is to be computed (i.e. tree level, real, double virtual etc.). This means that we need to input the following (the specifications for the input can be found in Table 1.1):

Warning: When running `mcmule` outside the normal repository, you need to make sure that an `out/` folder exists.

```
$ gfortran -fPIC --shared -o user.so user.f95
$ ./mcmule
1000
10
10000
50
70998
1.0
1.0
m2enng0
```

```
*****
*           C O L L I E R           *
*                                     *
*      Complex One-Loop Library      *
*      In Extended Regularizations    *
*                                     *
*      by A.Denner, S.Dittmaier, L.Hofer *
*                                     *
*           version 1.2.3             *
*                                     *
*****
```

```
- * - * - * - * - * - * -
```

(continues on next page)

(continued from previous page)

```

Version information
Full SHA: 3342511
Git SHA: 1fbc291
Git branch: HEAD
- * - * - * - * - * - * - * -
Calculating tau->e nu nu gamma at L0
- * - * - * - * - * - * - * -

internal avgi, sd:    31902651645147.434      3242845143300.6875
internal avgi, sd:    36962119569527.797      1491060763146.8340
internal avgi, sd:    39908483081760.562      701506532475.22485
internal avgi, sd:    41908326436302.352      183707731215.21738
internal avgi, sd:    41771416194096.336      55441877946.459091
internal avgi, sd:    41871492562379.680      27645368422.638184
internal avgi, sd:    41870973597620.547      21172712863.774796
internal avgi, sd:    41881968277900.094      17287639806.820400
internal avgi, sd:    41894819976244.469      15148087824.181145
internal avgi, sd:    41892443511666.180      13860145189.905710
internal avgi, sd:    41883909931737.320      9081654737.2369480
internal avgi, sd:    41891877400107.203      5996399688.5281315
internal avgi, sd:    41887401454137.172      4967120009.5028763
internal avgi, sd:    41894988589984.109      4318086453.2893734
internal avgi, sd:    41895218930734.938      3855189670.2831044
internal avgi, sd:    41893628691682.039      3569029881.5161963
internal avgi, sd:    41895702521658.094      3301046354.0162683
internal avgi, sd:    41894921420510.164      3068605199.3146548
internal avgi, sd:    41894380982483.836      2884341089.4262500
internal avgi, sd:    41894136940077.953      2719744511.4164872
internal avgi, sd:    41894755045877.328      2575585554.2240872
internal avgi, sd:    41894180414331.000      2448105950.1048141
internal avgi, sd:    41892974586371.242      2335940686.3421283
internal avgi, sd:    41892018243977.422      2237743190.2910728
internal avgi, sd:    41892128399199.852      2151548541.7080536
internal avgi, sd:    41891054946079.172      2079987935.2725692
internal avgi, sd:    41890529496649.336      2009220806.5744867
internal avgi, sd:    41889627128683.867      1952022430.9618585
internal avgi, sd:    41889091169697.750      1901209181.0766854
internal avgi, sd:    41889491086513.711      1851335964.4142988
internal avgi, sd:    41889024177143.492      1804584253.3775585
internal avgi, sd:    41888652800094.414      1763879105.2402925
internal avgi, sd:    41888186242209.695      1734933321.8742726
internal avgi, sd:    41888838662647.031      1702558920.3787835
internal avgi, sd:    41888878166048.805      1664687915.8245957
internal avgi, sd:    41888871786161.102      1628412032.4284451
internal avgi, sd:    41888673286317.961      1598222188.0324447
internal avgi, sd:    41888363240043.000      1570566301.1038322
internal avgi, sd:    41888533287695.047      1541834130.0455377
internal avgi, sd:    41888087919550.688      1514438031.2038479
internal avgi, sd:    41887838382975.297      1487390337.2575607
internal avgi, sd:    41887692329889.953      1465777595.5131192
internal avgi, sd:    41887528786746.531      1450928637.2665195
internal avgi, sd:    41887814931451.086      1432276674.4390638

```

(continues on next page)

(continued from previous page)

```

internal avgi, sd: 41887764015763.508      1409275835.0397925
internal avgi, sd: 41887871329949.469      1388512123.7455208
internal avgi, sd: 41887728279057.234      1369450030.9152539
internal avgi, sd: 41887673022843.117      1350554230.4978600
internal avgi, sd: 41887824787223.562      1332418851.3856776
internal avgi, sd: 41887720562604.266      1315515744.3643637
internal avgi, sd: 41887747627717.641      1297906527.3172038
internal avgi, sd: 41887385610296.359      1280408319.1259799
internal avgi, sd: 41887163475026.672      1262887224.9655898
internal avgi, sd: 41887020587065.422      1248392301.3985555
internal avgi, sd: 41886965905979.375      1236043197.7830524
internal avgi, sd: 41887132288349.984      1225259563.1290646
internal avgi, sd: 41887118281531.000      1211732414.0191844
internal avgi, sd: 41887256099447.883      1199948076.9753296
internal avgi, sd: 41887425753145.656      1188759649.9116085
internal avgi, sd: 41887079359692.539      1176252324.5589268
points: 10* 1M + 50* 10M      random seed: 70998
part: m2enng0      xicut: 1.000000      delcut: 1.000000
points on phase space 321225751 thereof fucked up      0

```

```

result, error: { 4.18871E+13, 1.17625E+09 }; chisq: 1.27

```

```

_ * _ * _ * _ * _ * _ * _ * _

```

McMule begins by printing some auto-versioning information (the *SHA1* hashes of the source code and the git version) as well as some user-defined information from the subroutine *inituser()*. Next, the integration begins. After every iteration, McMule prints the current best estimate and error of the total cross section or decay rate. Before exiting, it will also print again the input used as well as the number of points evaluated and the final result. This run took approximately 15 minutes.

Table 1.1: The options read from stdin by McMule. The calls are multiplied by 1000.

| Variable name | Data type | Comment |
|--------------------|------------|---|
| <i>ncall_ad</i> | integer | calls / iteration during pre-conditioning |
| <i>itmx_ad</i> | integer | iterations during pre-conditioning |
| <i>ncall</i> | integer | calls / iteration during main run |
| <i>itmx</i> | integer | iterations during main run |
| <i>ran_seed</i> | integer | <i>random seed</i> z_1 |
| <i>xinormcut</i> | real(prec) | the $0 < \xi_c \leq 1$ parameter |
| <i>delcut</i> | real(prec) | the δ_{cut} parameter (or at <i>NNLO</i> the second ξ_c) |
| <i>which_piece</i> | char(10) | the part of the calculation to perform |
| (opt) | unknown | the user can request further input during <i>inituser()</i> |

1.2.3 Analysing the output

After running McMule we want to calculate the actual cross section or decay rate and make plots. The McMule output is saved to the `out/` folder as a `.vegas` file that contains the entire state of the integrator (cf. Section *Differential distributions and intermediary state files*). We can open this file in python and make plots.

While it is possible to open just a single file using `pymule.vegas.FileRecord.read()`, this is rarely done as real-world calculations can involve hundreds of `.vegas` files. Instead, we move the `.vegas` file into a new directory, say `example1` and then use `sigma()` and `mergefks()`.

```

1 from pymule import *
2
3 lifetime = 1/(1000*(6.582119e-25)/(2.903e-13))
4 # define vegas directory
5 setup(folder="example1/")
6 dat = mergefks(sigma("m2enng0")) * GF**2*alpha*lifetime
7
8 dat
9 # <vegas-3 Record m2enng0 xsec = 0.0183403(5), 2 histograms>
10 dat.histograms
11 # ['minv', 'Ee']

```

Warning: In McMule the numerical value of the Fermi constant G_F and the fine-structure constant α are set to one for predominately historical reasons. This needs to be restored in python

The variable `dat` now contains the runtime (`time`), branching ratio (after multiplication with the lifetime, `value`), and χ^2 of the integration (`chi2a`) as well as our distributions (`Ee` and `minv`). Numerical values such as cross sections or branching ratios are stored as numpy arrays with errors as `np.array([y, dy])`. Distributions are stored as numpy $N \times 3$ matrices

```

np.array([[x1, y1, dy1],
          [x2, y2, dy2],
          [x3, y3, dy3],
          [x4, y4, dy4],
          [x5, y5, dy5],
          ...
          [xn, yn, dyn]])

```

These can be manipulated eg. using the tools of `pymule` described in Section *pymule user guide*. For now, we will just make a plot of the E_e distribution `Ee`

```

14 fig = plt.figure()
15 errorband(dat['Ee'])
16 plt.ylabel(r'\D\mathcal{B}/\D E_e$')
17 plt.xlabel(r'$E_e\,/\,\{\rm MeV\}$')
18 mulify(fig)
19 fig.savefig("dummy.svg")

```

Figure 1.1: Result of the LO test run for the E_e distribution

1.3 Running at NLO and beyond

A few things change once we go beyond *LO* since we can have extra radiation. To account for this, more *which_piece* need to be ran and then correctly combined. This also increases the number of runs necessary, meaning that the manual approach from above is no longer feasible.

1.3.1 Setting McMule up

Referring back to Section *Available processes and which_piece* we find that we need the pieces *m2ennGF* and *m2ennGR* for virtual and real corrections respectively. The *PID* table of Section *Particle ID* tells us that the real photon can be going to be *q6*.

We first need to decide whether we want to calculate exclusive or inclusive decays. The details here depend on the exact experimental situation which can be tricky to properly implement. Following the BaBar analysis [14, 18] we will consider the exclusive radiative decay, i.e. we request precisely one photon with energy $E_\gamma > 10$ MeV. The function *quant()* will have to take this into account with the additional argument *q6*, the momentum of the second photon.

To ensure *IR* safety, we define the harder and softer photon *gh* and *gs*, respectively, and require that the former (latter) has energy larger (smaller) than 10 MeV. This new version of *quant()* is also suitable for the *LO* calculation and it is generally advisable to use a single *quant()* function for all parts of a computation.

Listing 1.4: The measurement function beyond *LO*. The changes w.r.t. to *LO* are highlighted.

```

64  FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
65
66  real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
67  real (kind=prec) :: quant(nr_q)
68  real (kind=prec) :: gs(4), gh(4)
69  !! ===== keep the line below in any case ===== !!
70  call fix_mu
71
72  if (q5(4) > q6(4)) then
73    gh = q5 ; gs = q6
74  else
75    gh = q6 ; gs = q5
76  endif
77
78  if (gh(4) < 10.) pass_cut = .false.
79  if (gs(4) > 10.) pass_cut = .false.
80
81  names(1) = 'minv'
82  quant(1) = sqrt(sq(q2+gh))
83
84  names(2) = 'Ee'
85  quant(2) = q2(4)
86
87  END FUNCTION QUANT

```

1.3.2 Running McMule

The *FKS* scheme used in McMule introduces an unphysical parameter called ξ_c that can be varied between

$$0 < \xi_c \leq \xi_{\max} = 1 - \frac{(\sum_i m_i)^2}{s}$$

Checking the independence of physical results on the latter serves as a consistency check, both of the implementation of McMule but also of the *IR* safety of the *measurement function*. To do this, it can help to disentangle `m2ennGF` into `m2ennGV` and `m2ennGC` though this is not necessary. Only the latter depends on ξ_c and this part is typically much faster in the numerical evaluation.

A particularly convenient way to run McMule is using *menu files*¹. A *menu file* contains a list of jobs to be computed s.t. the user will only have to vary the *random seed* and ξ_c by hand as the statistical requirements are defined globally in a *config file*. This is completed by a *submission script*, usually called `submit.sh`. The submit script is what will need to be launched which in turn will take care of the starting of different jobs. It can be run on a normal computer or on a SLURM cluster [27]. To prepare the run in this way we can use `pymule`

Listing 1.5: The steps necessary to use `pymule` to prepare running McMule. Note that numbers listed as seeds are random and hence not reproducible.

```
$ pymule create -i
What generic process? [m2enn] m2ennG
How many / which seeds? [5]
Which xi cuts? [[0.5, 0.25, 0.125]]
Where to store data? [m2ennGtau-e] example2
Which pieces? [['0', 'V', 'R']] 0, V, C, R
How much statistics for 0 (pc, pi, c, i)? [(10000, 20, 100000, 100)] 1000,10,1000,50
How much statistics for V (pc, pi, c, i)? [(10000, 20, 100000, 100)] 1000,10,1000,50
How much statistics for C (pc, pi, c, i)? [(10000, 20, 100000, 100)] 1000,10,1000,50
How much statistics for R (pc, pi, c, i)? [(10000, 20, 100000, 100)] 5000,50,10000,100
Building files. To rerun this, execute
pymule create\
  --seeds 70998 66707 69184 75845 63937 \
  -xi 0.5 0.25 0.125 \
  --genprocess m2ennG \
  --output-dir babar-tau-e \
  --prog McMule \
  --stat R,5000,50,10000,100 \
  --stat 0,1000,10,1000,50 \
  --stat V,1000,10,1000,50 \
  --stat C,1000,10,1000,50
Expect 3750 iterations, 20.250000G calls
Created menu, config and submit script in example2
Please change the ntasks and time options accordingly
```

When using the tool, we are asked various questions, most of which have a default answer in square brackets. In the end `pymule` will create a directory that the user decided to call `example2`, where all results will be stored. The *menu* and *config* files generated by `pymule` are shown in Listing 1.6 and Listing 1.7

¹ The name *menu* was originally used by the cryptanalysts at Bletchley Park to describe a particular set of configurations for the ‘computer’ to try

Listing 1.6: *menu file* for the present calculation

```
conf m2enng.toml
mode vegas

run 19397 1.000000 m2enng0 0
run 52088 1.000000 m2enng0 0
run 83215 1.000000 m2enng0 0
run 93857 1.000000 m2enng0 0
run 86361 1.000000 m2enng0 0

run 19397 1.000000 m2enngV 0
run 52088 1.000000 m2enngV 0
run 83215 1.000000 m2enngV 0
run 93857 1.000000 m2enngV 0
run 86361 1.000000 m2enngV 0

run 19397 0.500000 m2enngC 0
run 52088 0.500000 m2enngC 0
run 83215 0.500000 m2enngC 0
run 93857 0.500000 m2enngC 0
run 86361 0.500000 m2enngC 0
run 19397 0.500000 m2enngR 0
run 52088 0.500000 m2enngR 0
run 83215 0.500000 m2enngR 0
run 93857 0.500000 m2enngR 0
run 86361 0.500000 m2enngR 0

run 19397 0.250000 m2enngC 0
run 52088 0.250000 m2enngC 0
run 83215 0.250000 m2enngC 0
run 93857 0.250000 m2enngC 0
run 86361 0.250000 m2enngC 0
run 19397 0.250000 m2enngR 0
run 52088 0.250000 m2enngR 0
run 83215 0.250000 m2enngR 0
run 93857 0.250000 m2enngR 0
run 86361 0.250000 m2enngR 0

run 19397 0.125000 m2enngC 0
run 52088 0.125000 m2enngC 0
run 83215 0.125000 m2enngC 0
run 93857 0.125000 m2enngC 0
run 86361 0.125000 m2enngC 0
run 19397 0.125000 m2enngR 0
run 52088 0.125000 m2enngR 0
run 83215 0.125000 m2enngR 0
run 93857 0.125000 m2enngR 0
run 86361 0.125000 m2enngR 0
```


Listing 1.7: Configuration file for the present calculation

```
[McMule]
# specify the program to run relative to `pwd`
mcmule = "mcmule"
# Specify the variables nenter_ad, itmx_ad, nenter and itmx
# for each piece you want to run.

[McMule.statistics.vegas]
m2enng0 = { ncall_pre = 1000, itmx_pre = 10, ncall = 1000, itmx = 50 }
m2enngV = { ncall_pre = 1000, itmx_pre = 10, ncall = 1000, itmx = 50 }
m2enngC = { ncall_pre = 1000, itmx_pre = 10, ncall = 1000, itmx = 50 }
m2enngR = { ncall_pre = 5000, itmx_pre = 50, ncall = 10000, itmx = 100 }
```

We now need to ensure pymule finds the mcmule executable, either by copying it into the directory pymule just created or by adapting the path in the *config*. After copying the user library *user.so* into the same folder, we can start mcmule by going into the directory *example2* that pymule just created and running

```
cd example2
cp /path/to/mcmule .
cp /path/to/user.so .
pymule batch shell menu-m2enng.menu
```

Note that per default this will spawn at most as many jobs as the computer pymule ran on had CPU cores. If the user wishes a different number of parallel jobs, change the fifth line of *example2/submit.sh* to

```
pymule batch shell -np <number of cores> menu-m2enng.menu
```

When running on a SLURM system, we need to use a SLURM *submission script*, adapted to the needs of the cluster mcmule is running on

```
#!/bin/bash
#SBATCH --partition=<partition>
#SBATCH --time=<time limit>
#SBATCH --ntasks=<number of jobs>
#SBATCH --clusters=<name of the cluster>
#SBATCH --output=slurm-%j.out

pymule batch slurm menu-m2enng.menu
```

1.3.3 Analysing the results

After running the code, we need to combine the various *which_piece* into physical results that we will want to use to create plots. This is the moment where pymule's *mergefks()* shines.

Listing 1.8: An example code to analyse the results for $\tau \rightarrow \nu\bar{\nu}e\gamma$ in pymule. Note that, in the Fortran code $G_F = \alpha = 1$. In pymule they are at their physical values.

```
from pymule import *
```

(continues on next page)

```

# To normalise branching ratios, we need the tau lifetime
lifetime = 1/(1000*(6.582119e-25)/(2.903e-13))

# The folder where McMule has stored the statefiles
setup(folder='example2/out.tar.bz2')

# Import LO data and re-scale to branching ratio
LO = mergefks(sigma('m2enng0')) * GF**2*lifetime*alpha

# Import NLO corrections from the three pieces
NLO = mergefks(
    sigma('m2enngR'),      # real corrections
    sigma('m2enngC'),      # counter term
    anyxi=sigma('m2enngV') # virtual corrections
) * GF**2*lifetime*alpha**2

# The branching ratio at NLO = LO + correction
fullNLO = LO + NLO

# Print results
print("BR_0 = ", printnumber(LO.value))
print("dBR = ", printnumber(NLO.value))

# Produce energy plot
fig1, (ax1, ax2) = kplot(
    {'lo': LO['Ee'], 'nlo': NLO['Ee']},
    labelx=r"$E_e\,/\,\{\rm MeV}\$",
    labelsigma=r"$\mathcal{B}/\mathcal{D} E_e\ $"
)
ax2.set_ylim(-0.2,0.01)

# Produce visible mass plot
fig2, (ax1, ax2) = kplot(
    {'lo': LO['minv'], 'nlo': NLO['minv']},
    labelx=r"$m_{e\gamma}\,/\,\{\rm MeV}\$",
    labelsigma=r"$\mathcal{B}/\mathcal{D} m_{e\gamma}\ $"
)
ax1.set_yscale('log')
ax1.set_xlim(1000,0) ; ax1.set_ylim(5e-9,1e-3)
ax2.set_ylim(-0.2,0.)

```

Once `pymule` is imported and setup, we import the `LO` and `NLO which_piece` and combine them using two central `pymule` commands that we have encountered above: `sigma()` and `mergefks()`. `sigma()` takes the `which_piece` as an argument and imports matching results, already merging different *random seeds*. `mergefks()` takes the results of (multiple) `sigma()` invocations, adds results with matching ξ_c values and combines the result. In the present case, $\sigma_n^{(1)}$ is split into multiple contributions, namely `m2enngV` and `m2enngC`. This is indicated by the `anyxi` argument.

Next, we can use some of `pymule`'s tools (cf. Listing Listing 1.8) to calculate the full `NLO BRs` from the corrections and the `LO` results

$$\mathcal{B}|_{LO} = 1.8339(1) \times 10^{-2}$$

$$\mathcal{B}|_{NLO} = 1.6451(1) \times 10^{-2}$$

which agree with [10, 21], but $\mathcal{B}|_{NLO}$ is in tension with the value $\mathcal{B}|_{\text{exp}} = 1.847(54) \times 10^{-2}$ reported by BaBar [14,

18]. As discussed in [21, 24] it is very likely that this tension would be removed if a full *NLO* result was used to take into account the effects of the stringent experimental cuts to extract the signal. This issue has been explained in detail in [21, 24, 25].

As a last step, we can use the `matplotlib`-backed `kplot()` command to present the results for the distributions (logarithmic for $m_{e\gamma}$ and linear for E_e). The upper panel of Figure 1.2 shows the results for the invariant mass $m_{e\gamma}$ at *LO* (green) and *NLO* (blue) in the range $0 \leq m_{e\gamma} \leq 1$ GeV. Note that this, for the purposes of the demonstration, does not correspond to the boundaries given in the run.

Figure 1.2: Results of the toy run to compute $m_{e\gamma}$ for $\tau \rightarrow \nu\bar{\nu}e\gamma$. Upper panels show the *LO* (green) and *NLO* (blue) results, the lower panels show the *NLO* *K* factor.

The distribution falls sharply for large $m_{e\gamma}$. Consequently, there are only few events generated in the tail and the statistical error becomes large. This can be seen clearly in the lower panel, where the *NLO* *K* factor is shown. It is defined as

$$K^{(1)} = 1 + \frac{d\sigma^{(1)}}{d\sigma^{(0)}}$$

and the band represents the statistical error of the Monte Carlo integration. To obtain a reliable prediction for larger values of $m_{e\gamma}$, i.e. the tail of the distribution, we would have to perform tailored runs. To this end, we should introduce a cut $m_{\text{cut}} \ll m_\tau$ on $m_{e\gamma}$ to eliminate events with larger invariant mass. Due to the adaption in the numerical integration, we then obtain reliable and precise results for values of $m_{e\gamma} \lesssim m_{\text{cut}}$.

Figure 1.3 shows the electron energy distribution, again at *LO* (green) and *NLO* (blue). As for $m_{e\gamma}$ the corrections are negative and amount to roughly 10%. Since this plot is linear, they can be clearly seen by comparing *LO* and *NLO*. In the lower panel once more the *K* factor is depicted. Unsurprisingly, at the very end of the distribution, $E_e \sim 900$ MeV, the statistics is out of control.

Figure 1.3: Results of the toy run to compute E_e for $\tau \rightarrow \nu\bar{\nu}e\gamma$. Upper panels show the *LO* (green) and *NLO* (blue) results, the lower panels show the *NLO* *K* factor.

1.4 More complicated runs

To demonstrate some of McMule capabilities, we tweak the observable a bit. Since the tau is usually produced in $e^+e^- \rightarrow \tau^+\tau^-$, we instead use the *LO* cross section

$$\frac{d\sigma}{d(\cos\theta)} \propto \left(1 + \frac{4m_\tau^2}{s}\right) + \left(1 + \frac{4m_\tau^2}{s}\right) \cos\theta \quad (1.1)$$

with $\sqrt{s} = m_{\Upsilon(4S)} = 10.58$ GeV.

To accurately simulate this situation, we need to

- choose a random value for θ ,
- construct the tau momentum p_1 in the lab frame,
- boost the momenta from McMule into this frame, and
- apply a correction weight from (1.1)

for every event. We require the following cuts in the lab frame

- the produced electron and hard photon have $-0.75 \leq \cos\theta_{i,e^-} \leq +0.95$
- the hard photon energy is bigger than 220 MeV

Further, we want to have a switch for inclusive and exclusive measurements without having to adapt the user file.

1.4.1 Asking for user input

To be able to switch cuts on and off, we need to read input from the user at runtime. This can be done in the `inituser()` routine where input can be read. We can store the result in a global variable (`exclusiveQ`) so we can later use it in `quant()`. Further, we need modify the name of the vegas file by changing `filenamesuffix`. It is also good practice to print the configuration chosen for documentation purposes.

```

58  SUBROUTINE INITUSER
59  call initflavour("tau-e")
60  read*, exclusiveQ
61
62  if(exclusiveQ == 1) then
63    print*, "Calculating tau->e nu nu gamma in ee->tau tau exclusive"
64    filenamesuffix = "e"
65  else
66    print*, "Calculating tau->e nu nu gamma in ee->tau tau inclusive"
67    filenamesuffix = "i"
68  endif
69
70  ! Let the tau be unpolarised
71  pol1 = (/ 0._prec, 0._prec, 0._prec, 0._prec /)
72  END SUBROUTINE

```

Note: When using the *menu file* system, this can only be a single integer. To read multiple bits of information, you need to encode the data somehow.

1.4.2 Generation of the tau momentum

We can use the user integration feature of McMule to generate $\cos\theta$. This allows us to write

$$\sigma \sim \int_0^1 dx_1 \int_0^1 dx_2 \cdots \int_0^1 dx_m \times \int d\Phi |\mathcal{M}_n|^2 f(x_1, x_2, \dots, x_n; p_1, \dots, p_n)$$

with a generalised *measurement function* f . Since (1.1) is sufficiently simple, we will sample $\cos\theta$ with a uniform distribution and apply a correction weight rather trying to sample it directly. We set the variable `userdim` to one to indicate that we want to carry out $m = 1$ extra integrations and define the function `userevent()` that sets global variable `cth` for $\cos\theta$

```

135  SUBROUTINE USEREVENT(X, NDIM)
136  integer :: ndim
137  real(kind=prec) :: x(ndim)
138
139  cth = 2*x(1) - 1
140  userweight = (1+Mm**2/Etau**2) + (1-Mm**2/Etau**2) * cth
141  END SUBROUTINE USEREVENT

```

Warning: This function can be used to change the centre-of-mass energy and masses of the particles. However, one must re-compute the flux factors and ξ_{\max} relations.

1.4.3 Boosting into the lab frame

We begin by writing down the momentum of tau in the lab frame as

$$p_1 = \left(0, |\vec{p}| \sqrt{1 - \cos^2 \theta}, |\vec{p}| \cos \theta, E \right)$$

with $|\vec{p}| = \sqrt{E^2 - m_\tau^2}$. Next, we use the McMule function `boost_back()` to boost the momenta we are given into the lab frame. From there we can continue applying our cuts as before, utilising the McMule function `cos_th` to calculate the angle between the particle and the beam axis.

```

75  FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
76
77  real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
78  real (kind=prec) :: ptau, cos_e, cos_g
79  real (kind=prec) :: p1Lab(4), p2Lab(4), p5Lab(4), p6Lab(4)
80  real (kind=prec) :: quant(nr_q)
81  real (kind=prec) :: gs(4), gh(4)
82  real (kind=prec), parameter :: ez(4) = (/ 0., 0., 1., 0. /)
83  !! ===== keep the line below in any case ===== !!
84  call fix_mu
85  pass_cut = .true.
86
87  ptau = sqrt(Etau**2-Mtau**2)
88  p1Lab = (/ 0., ptau*sqrt(1-cth**2), ptau*cth, Etau /)
89
90  p1Lab = boost_back(p1Lab, q1)
91  p2Lab = boost_back(p1Lab, q2)
92  p5Lab = boost_back(p1Lab, q5)
93  p6Lab = boost_back(p1Lab, q6)
94
95  if (p5Lab(4) > p6Lab(4)) then
96    gh = p5Lab ; gs = p6Lab
97  else
98    gh = p6Lab ; gs = p5Lab
99  endif
100
101  cos_e = cos_th(p2Lab, ez)
102  cos_g = cos_th(gh , ez)
103
104  if ( (cos_e > 0.95 .or. cos_e < -0.75) .or. (cos_g > 0.95 .or. cos_g < -0.75) ) then
105    pass_cut = .false.
106    return
107  endif
108
109  if(exclusiveQ == 1) then
110    if (gh(4) < 220. .or. gs(4) > 220.) then
111      pass_cut = .false.
112      return
113    endif
114  else
115    if (gh(4) < 220.) then
116      pass_cut = .false.
117      return

```

(continues on next page)

(continued from previous page)

```

118     endif
119 endif
120
121 names(1) = 'minv'
122 quant(1) = sqrt(sq(q2+gh))
123
124 names(2) = 'Ee'
125 quant(2) = p2Lab(4)
126
127 names(3) = 'cos_e'
128 quant(3) = cos_e
129
130 names(4) = 'cos_g'
131 quant(4) = cos_g
132 END FUNCTION QUANT

```

1.4.4 Running and analysis

At this point we can run McMule and proceed with the analysis as before. We need to do two runs, one for the exclusive and one for the inclusive. However, only the real corrections differ, therefore we only need 24 runs and not 36. The last argument of the run command in the *menu file* will be passed as the observable we have defined in *inituser()*. We need to pass 1 (exclusive) or 0 (inclusive) as shown in Listing 1.9²

Listing 1.9: The *menu file* for the present calculation

```

image registry.gitlab.com/mule-tools/mcmule:redesign example3/user.f95

conf example3/m2enng-tau-e.conf

run 75217 1.000000 m2enng0 0
run 52506 1.000000 m2enng0 0
run 22671 1.000000 m2enng0 0

run 53796 1.000000 m2enngV 0
run 15282 1.000000 m2enngV 0
run 89444 1.000000 m2enngV 0

run 98870 0.600000 m2enngC 0
run 91991 0.600000 m2enngC 0
run 79769 0.600000 m2enngC 0

run 21175 0.800000 m2enngC 0
run 57581 0.800000 m2enngC 0
run 81929 0.800000 m2enngC 0

run 70604 0.600000 m2enngR 0
run 33013 0.600000 m2enngR 0
run 22530 0.600000 m2enngR 0

```

(continues on next page)

² Looking at the results from our previous run, we can deduce that $\xi_c \sim 0.7$ is the optimal place for running since $\sigma_n(\xi_c = 0.7) \sim \sigma_{n+1}(\xi_c = 0.7)$ which reduces cancellation between the different pieces. This optimisation is not strictly necessary and we still run for two values of ξ_c , 0.6 and 0.8.

(continued from previous page)

```

run 82222 0.800000 m2enngR 0
run 30935 0.800000 m2enngR 0
run 40689 0.800000 m2enngR 0

run 70604 0.600000 m2enngR 1
run 33013 0.600000 m2enngR 1
run 22530 0.600000 m2enngR 1

run 82222 0.800000 m2enngR 1
run 30935 0.800000 m2enngR 1
run 40689 0.800000 m2enngR 1

```

We can now run McMule. When analysing the output we need to take care to not mix the different observables which we do by passing the optional argument `obs` to `sigma()`. The resulting plot is shown in [Figure 1.4](#)

Listing 1.10: The analysis pipeline for this calculation

```

# Loading the LO is the same as before
setup(folder='example3/out.tar.bz2')
LO = mergefks(sigma('m2enng0')) * GF**2*lifetime*alpha

# Import the excl. NLO by specifying the observable
# for the real corrections
NLOexcl = mergefks(
    sigma('m2enngR', obs='e'),
    sigma('m2enngC'),
    anyxi=sigma('m2enngV')
) * GF**2*lifetime*alpha**2
fullNLOexcl = LO + NLOexcl

NLOincl = mergefks(
    sigma('m2enngR', obs='i'),
    sigma('m2enngC'),
    anyxi=sigma('m2enngV')
) * GF**2*lifetime*alpha**2
fullNLOincl = LO + NLOincl

print("BR_0 = ", printnumber(LO.value))
print("BRexcl = ", printnumber(fullNLOexcl.value))
print("BRincl = ", printnumber(fullNLOincl.value))

fig3, (ax1, ax2) = kplot(
    {
        'lo': scaleplot(LO['Ee'], 1e3),
        'nlo': scaleplot(NLOexcl['Ee'], 1e3),
        'nlo2': scaleplot(NLOincl['Ee'], 1e3)
    },
    labelx=r"$E_e\,/\,\{\rm GeV}\$",
    labelsigma=r"$\mathcal{B}/\mathcal{D} E_{e}\$",
    legend={

```

(continues on next page)

(continued from previous page)

```
'lo': '$\\rm LO$',  
'nlo': '$\\rm NLO\\ exclusive$',  
'nlo2': '$\\rm NLO\\ inclusive$'  
},  
legendopts={'what': 'l', 'loc': 'lower left'}  
)  
ax2.set_ylim(-0.12,0.02)
```

Figure 1.4: Results of the toy run to compute E_e in the labframe.

Chapter 2

Structure of McMule

McMule is written in Fortran 95 with helper and analysis tools written in python¹. The code is written with two kinds of applications in mind. First, several processes are implemented, some at *NLO*, some at *NNLO*. For these, the user can define an arbitrary (infrared safe), fully differential observable and compute cross sections and distributions. Second, the program is set up such that additional processes can be implemented by supplying the relevant matrix elements.

The main part of McMule is the `libmcmule` library which consists of several modules with a simple, mostly hierarchic structure. The `mcmule` executable loads this library which in turn loads the *measurement function* defined by the user.

2.1 The user function

For a user of the code who wants to run for an already implemented process, this is the only relevant part. The user function is defined in a (usually dynamically loaded) library and defines the *measurement function*.

At the beginning of the module, the user has to specify the number of quantities to be computed, `user/nq`, the number of bins in the histogram, `user/nbins`, as well as their lower and upper boundaries, `user/min_val` and `user/max_val`. The last three quantities are arrays of length `user/nq`. The quantities themselves, i.e. the measurement function, is to be defined by the user in terms of the momenta of the particles in `user/quant()`. Cuts can be applied by setting the logical variable `mcmule/pass_cut` to false². Some auxiliary functions like (pseudo)rapidity, transverse momentum etc. are predefined in `mcmule` as documented in Section *libmcmule user guide*. Each quantity has to be given a name through the array `mcmule/names`.

Further, `user` contains a subroutine called `quant/inituser()`. This allows the user to read additional input at runtime, for example which of multiple cuts should be calculated. It also allows the user to print some information on the configuration implemented. Needless to say that it is good idea to do this for documentation purposes.

Once written (or more likely adapted from a `template`), the user function is compiled so it can be loaded by `libmcmule`

```
$ gfortran -fPIC --shared -o user.so user.f95
```

It is possible to write a user function in languages other than Fortran and a `mcmule.h` header file is provided for C and C++. However, many of the quality-of-life functions of `mcmule` are only available in Fortran (cf. Section *Symbols exposed to C/C++*)

¹ Additionally to the `python` tool a `Mathematica` tool is available.

² For quad precision `prec=16` and the compiler flag `-fdefault-real-16` is required.

2.2 Modular structure of libmcmule

The relation between the most important Fortran modules is depicted in [Figure 2.1](#). A solid arrow indicates “using” the full module, whereas a dashed arrow is indicative of partial use. In what follows we give a brief description of the various modules and mention some variables that play a prominent role in the interplay between the modules.

Figure 2.1: The structure of McMule

global_def:

This module simply provides some parameters such as fermion masses that are needed throughout the code. It also defines `real(kind=prec)` as a generic type for the precision used.³ Currently, this simply corresponds to double precision.

functions:

This module is a library of basic functions that are needed at various points in the code. This includes dot products, eikonal factors, the integrated eikonal, and an interface for scalar integral functions among others.

collier:

This is an external module [[3](#), [4](#), [5](#), [6](#)]. It will be linked to McMule during compilation and provides the numerical evaluations of the scalar and in some cases tensor integral functions in `functions`.

phase_space:

The routines for generating phase-space points and their weights are collected in this module. Phase-space routines ending with `FKS` are prepared for the *FKS* subtraction procedure with a single unresolved photon. In the weight of such routines a factor ξ_1 is omitted to allow the implementation of the distributions in the *FKS* method. This corresponds to a global variable `xiout1`. This factor has to be included in the integrand of the module `integrand`s. Also the variable `qsoft` is provided that corresponds to the photon momentum without the (vanishing) energy factor ξ_1 . Routines ending with `FKSS` are routines with two unresolved photons. Correspondingly, a factor $\xi_1 \xi_2$ is missing in the weight and `xioutA` and `xioutB`, as well as `qsoftA` and `qsoftB` are provided. To ensure numerical stability it is often required to tune the phase-space routine to a particular kinematic situation.

openloops:

This is the external OpenLoops library [[1](#), [2](#)] that we use for some real-virtual matrix elements. It is pulled as a `git` submodule and linked to McMule during compilation.

olinterface:

This connects `openloops` to the rest of McMule by initialising OpenLoops for the process under consideration and converting to and from the OpenLoops conventions which are slightly different than the ones used by McMule.

{pg}_mat_el:

Matrix elements are grouped into *process groups* such as muon decay (`mudec`) or μ - e and μ - p scattering (`mue`). Each *process group* contains a `mat_el` module that provides all matrix elements for its group. Simple matrix elements are coded directly in this module. More complicated results are imported from sub-modules not shown in [Figure 2.1](#). A matrix element starting with `P` contains a polarised initial state. A matrix element ending in `av` is averaged over a neutrino pair in the final state.

{pg}:

In this module the soft limits of all applicable matrix elements of a *process group* are provided to allow for the soft subtractions required in the *FKS* scheme. These limits are simply the eikonal factor evaluated with `qsoft` from `phase_space` times the reduced matrix element, provided through `mat_el`.

This module also functions as the interface of the *process group*, exposing all necessary functions that are imported by

³ Technically, `pass_cut` is a list of length `nr_q`, allowing to decide whether to cut for each histogram separately.

mat_el,

which collects all matrix elements as well as their particle labelling or *PID*.

user_dummy:

This interfaces to the (usually dynamically loaded) *measurement function* defined by the user.

vegas:

As the name suggests this module contains the adaptive Monte Carlo routine vegas [15]. The binning routine *bin_it* is also in this module, hence the need for the binning metadata, i.e. the number of bins and histograms as well as their bounds (*min_val* and *max_val*) and names, from *user_dummy*.

integrand:

In this module the functions that are to be integrated by vegas are coded. There are three types of integrands: non-subtracted, single-subtracted, and double-subtracted integrands, corresponding to the three parts of the FKS² scheme [8, 25]. The matrix elements to be evaluated and the phase-space routines used are set using function pointers through a subroutine *initpiece*. The factors ξ_i that were omitted in the phase-space weight have to be included here for the single- and double-subtracted integrands.

mcmule_header:

This is the main part of the library that initialised and runs McMule with configurations passed by the main executable. It also defines the Module *mcmule* which contains *runmcmule()*.

test:

For developing purposes, a separate main program exists that is used to validate the code after each change. Reference values for matrix elements and results of short integrations are stored here and compared against.

mcmule:

This is the main executable, but actually does little else than read the inputs and call *mcmule_header*.

The library of matrix elements deserves a few comments. As matrix elements quickly become very large, we store them separately from the main code. This makes it also easy to extend the program by minimising the code that needs to be changed.

We group matrix elements into *process groups*, *generic processes*, and *generic pieces* as indicated in Appendix *Available processes and which_piece*. The generic process is a prototype for the physical process such as $\ell p \rightarrow \ell p$ where the flavour of the lepton ℓ is left open. The generic piece describes a part of the calculation such as the real or virtual corrections, i.e. the different pieces of (10.1) (or correspondingly (10.7) at *NNLO*), that themselves may be further subdivided as is convenient. In particular, in some cases a generic piece is split into various partitions (cf. Section *Phase-space generation* for details on why that is important).

2.3 What happens when running

In the following we discuss what happens behind the scene when asking McMule to perform the calculation of *m2enng0* in Section *Simple runs at LO*.

0. When started, the first thing *mcmule* does is load the *measurement function* as from a *shared library* using the *load_quant()* routine.
1. When started, *mcmule* reads options from *stdin* as specified in Table 1.1. It then passes these values to the *runmcmule()* function of *mcmule*.
2. Once McMule knows its configuration it associates the numerical values of the masses, as specified through flavour as set by the user. In particular, we set the generic masses *Mm* and *Me* to *Mtau* and *Me1*. This is done in *initflavour()*, defined in *global_def*. For other processes this might also involve setting e.g. centre-of-mass energies *scms* to default values.
3. Next, the function to be integrated by vegas is determined. This is a function stored in *integrand*. There are basically three types of integrands: a standard, non-subtracted integrand, *sigma_0*, a single-subtracted integrand

needed beyond *LO*, `sigma_1`, and a double-subtracted integrand needed beyond *NLO*, `sigma_2`. Which integrand is needed and what matrix elements and phase-space it depends on is determined by calling the function `init_piece` which uses the variable `which_piece` to point function pointers at the necessary procedures. For our *LO* case, `init_piece` sets the integrand to `sigma_0` and fixes the dimension of the integration to `ndim = 8`.

4. The matrix element pointer is assigned to the matrix element that needs to be called, `Pm2enngAV(q1, n1, q2, q3, q4, q5)`. The name of the function suggests we compute $\mu(q_1, n_1) \rightarrow [\nu(q_3)\bar{\nu}(q_4)]e(q_2)\gamma(q_5)$ with the polarisation vector `n1` of the initial lepton. Even though we average over the neutrinos, their momenta are still given for completeness.
5. The interplay between the function `sigma_0(x, wgt, ndim)` and `vegas` is as usual, through an array of random numbers `x` of length `ndim` that corresponds to the dimension of the integration. In addition there is the `vegas` weight of the event, `wgt` due to the Jacobian introduced by the importance sampling. The function `sigma_0` simply evaluates the complete weight `wg` of a particular event by combining `wgt` with the matrix element supplemented by symmetry, flux, and phase-space factors.
 1. In a first step a phase-space routine of `phase_space` is called. For our *LO* calculation, `init_piece` pointed a pointer to the phase-space routine `psd5_25()`, a phase-space routine optimised for radiative lepton decays (cf. Section *Phase-space generation*). This will be called as a first step in the integrand to generate the momenta with correct masses as well as the phase-space weight `weight`.
 2. With these momenta the observables to be computed are evaluated with a call to `user/quant()`. If one of them passes the cuts, the variable `pass_cut` is set to true.
 3. This triggers the computation of the matrix element and the assembly of the full weight.
 4. In a last step, the routine `bin_it`, stored in `vegas`, is called to put the weight into the correct bins of the various distributions. If the variable under- or overshoots the bounds specified by `user/min_val` and `user/max_val`, the event is placed into dedicated, infinitely big under- and overflow bins.

These steps are done for all events and those after pre-conditioning are used to obtain the final distributions.

6. After preconditioning the state of the integrator is reset, as is usual.
7. During the main run, the code generates a statefile from which the full state of the integrator can be reconstructed should the integration be interrupted (cf. Section *Differential distributions and intermediary state files* for details). This makes the statefile ideal to also store results in a compact format.
8. The value and error estimate of the integration is printed to `stdout`.

To analyse these results, we provide a python tool `pymule`, additionally to the main code for McMule. `pymule` uses `numpy` [26] for data storage and `matplotlib` for plotting [13]. While `pymule` works with any python interpreter, IPython [22] is recommended. We will encounter `pymule` in Section *Analysing the results* when we discuss how to use it to analyse results. A full list of functions provided can be found in Appendix *pymule user guide*.

Chapter 3

Observable validator

The validator is designed to take in a *userfile* (such as those made in *Getting started*) and a *which_piece* or *generic process* (see section *Available processes and which_piece*) and perform various checks of the *measurement function* to see if the observables it implements are *physical*.

The *measurement function* acts as a *perfect* observer of the system. This can have some draw backs: for example two indistinguishable particles can be distinguished by the *measurement function* and can lead to results that are physically impossible to obtain in real life.

The design of the validator is such that installing McMule in full on the computer running the tests is not required. Instead, pymule provides the mocks of the functions made available by `mcmule.mod` (cf. Section *libmcmule user guide*). Should the user function require more McMule internals, a full installation is required and needs to be manually initialised.

The checks results are sorted into three different categories: passed, warnings, and failed. Failing any test means that the *measurement function* is most certainly unphysical and should be fixed (if you believe a test is failed incorrectly, please open an *issue* with the full logs). Warnings are merely recommendations on how the user function might be modified to make it more universally applicable.

Tests that may result in an error include:

- Soft photon limit: taking a soft photon's energy to zero should have the same result as the photon not existing
- Swap symmetry: swapping two indistinguishable particles should not change the observables
- Soft-limit swap-symmetry check: a combination of swapping indistinguishable photons, one of which is soft
- Out-of-bounds check: all events fit inside the histogram bounds

Tests that may result in a warning include:

- Rotation check: it should not matter if the system decays in the Z axis or any other direction
- Lorentz boost check: ideally a frame should be chosen and boosted into inside the *measurement function*
- Flip check: symmetrically flipping the system in any axis should have no impact on the observables

Further usage and interpretation of each test can be found below.

3.1 Using the validator

When using the validator, we can either compile the user file ourselves

```
$ gfortran -I/path/to/mcmule.mod --shared -o user.so -fPIC user.f95 -fdefault-real-8
$ pymule validator <arguments> user.so
```

or use pymule to compile

```
$ pymule validator <arguments> user.f95
```

A full list of arguments is given in below.

A *measurement function* is always tested against a *which_piece* or collection of pieces. To test eg. `m2enng0` we type

```
$ pymule validator -l INFO -w m2enng0 user.so
```

where we specify the logging to be *INFO* so we see all the tests.

The output here can be split into 3 sections

1. Loading the required libraries

At the logging level of *INFO* we see each of the shared libraries which must be loaded for McMule to work correctly, including the *userfile*.

```
INFO:pymule.validator.importer:Loading shared libraries
INFO:pymule.validator.importer:Loading library: libcollier.so
INFO:pymule.validator.importer:Loading library: libolcommon.so
INFO:pymule.validator.importer:Loading library: libtrred.so
INFO:pymule.validator.importer:Loading library: liboneloop.so
INFO:pymule.validator.importer:Loading library: libcuttools.so
INFO:pymule.validator.importer:Loading library: libopenloops.so
INFO:pymule.validator.importer:Loading library: libqedc19.so
INFO:pymule.validator.importer:Loading library: libmcmule.so
INFO:pymule.validator.importer:Loading library: ./user.so
```

2. Loading the *measurement function* and variables

Next the functions and variables required to test the *userfile* are found and loaded, including running `inituser()` if it can be found (in this case simply printing This is an empty user file).

```
INFO:pymule.validator.importer:Function 'QUANT' found with name: __user_MOD_quant
INFO:pymule.validator.importer:Function 'inituser' found with name: __user_MOD_inituser
INFO:pymule.validator.importer:Variable 'filenamesuffixLen' found with name: __user_MOD_
↪filenamesuffixlen
This is a empty user file
INFO:pymule.validator.importer:Variable 'nrq' found with name: __user_MOD_nrq
INFO:pymule.validator.importer:Variable 'nrbins' found with name: __user_MOD_nrbins
```

3. Testing of the *measurement function*

Then the actually testing of the *measurement function* can begin. The validator will automatically find and run the appropriate tests for a each *which_piece*, in this case:

```
INFO:pymule.validator.validator:Testing whichpiece: m2enng0 with 6 tests
INFO:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... passed
```

(continues on next page)

(continued from previous page)

```
INFO:pymule.validator.validator:Test 2: RotationValidator(0.08644597231507764, 0, 0)...
↳passed
INFO:pymule.validator.validator:Test 3: RotationValidator(5.104560253717271, 3.
↳868168781137662, 5.213091066969582)... passed
INFO:pymule.validator.validator:Test 4: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 5: FlipValidator(y)... passed
INFO:pymule.validator.validator:Test 6: FlipValidator(z)... passed
INFO:pymule.validator.validator:Overall pass rate: 6 / 6, 100.0%
```

For this *userfile* all the appropriate tests ran correctly and passed. Different *which_pieces* may run different tests with different results.

If a *Non-fatal* test fails then, instead of reporting at the *INFO* level, it reports at *WARNING* level.

```
WARNING:pymule.validator.validator:Test 4: FlipValidator(x)... failed,
...
WARNING:pymule.validator.validator:Overall pass rate: 5 / 6, 83.33%
```

If a *Fatal* test fails it reports at the *ERROR* level

```
ERROR:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... failed,
...
ERROR:pymule.validator.validator:Overall pass rate: 5 / 6 with 1 fatal failure! 83.33%
```

Similarly a *generic piece* can have all the *which_pieces* under it be checked in one go.

For example testing all orders of the *generic piece* ee2ee is as simple as typing

```
$ pymule validate -g ee2ee -l INFO user.so
...
INFO:pymule.validator.validator:Testing at order LO:
INFO:pymule.validator.validator:Testing whichpiece: ee2ee0 with 8 tests
...
INFO:pymule.validator.validator:Testing at order NLO:
INFO:pymule.validator.validator:Testing whichpiece: ee2eeF with 8 tests
INFO:pymule.validator.validator:Testing whichpiece: ee2eeA with 8 tests
...
INFO:pymule.validator.validator:Testing at order NNLO:
INFO:pymule.validator.validator:Testing whichpiece: ee2eeFF with 8 tests
...
```

If we only want to test a single order (such as *NLO*) we could write

```
$ pymule validate -g ee2ee -o NLO -l INFO user.so
...
INFO:pymule.validator.validator:Testing at order NLO:
INFO:pymule.validator.validator:Testing whichpiece: ee2eeF with 8 tests
INFO:pymule.validator.validator:Testing whichpiece: ee2eeA with 8 tests
...
```

3.2 Logging Levels

Table 3.1: Various logging levels for the validator

| Integer | Name | Effect |
|---------|----------|--|
| -1 | CRITICAL | No output of anykind from the validator, mainly used for internal code testing |
| 0 | ERROR | Only <i>Fatal</i> failures output, suppresses graphing |
| 1 | WARNING | <i>Fatal</i> and <i>Non-fatal</i> failures are output. Allows graphing |
| 2 | INFO | All tests being run are output, even if they pass. Allows graphing |
| 3 | DEBUG | All output possible, including forcing graphs to be generated if applicable |

3.3 Measurement function tests in detail

Each test checks some aspect of the *measurement function*. This section will go through example cases of where each test may fail and how to prevent that failure.

3.3.1 Out-of-bounds validator

The out-of-bounds validator generates a fixed number of Monte Carlo events with uniform weight and calculates the binning of the events that pass the cuts. If any event ends up outside the histogram bounds, this test fails. As this test is fairly slow, it needs to be activated by specifying the number of points to be used. After all pieces have been considered, the validator will print a list of recommended selection cuts, eg.

| Observable | Selection | min_val | act. min | act. max | max_val |
|------------|-----------|-----------|-----------|----------|---------|
| p+ | 51: 80 | 0.45 | 0.450094 | 0.522873 | 1 |
| p- | 51: 80 | 0.45 | 0.450053 | 0.522873 | 1 |
| thav | 1:600 | 1 | 1.00002 | 2.14154 | 2.14159 |
| dphi | 1:600 | -0 | 0 | 0.149923 | 0.15 |
| xi | 1:394 | -0 | 0 | 0.164099 | 0.25 |
| mxx | 252:185 | 299.167 | 387.413 | 422.634 | 700 |
| cth+ | 126:471 | -0.646667 | -0.605093 | 0.593247 | 0.64 |
| cth- | 156:446 | -0.646667 | -0.540288 | 0.540256 | 0.64 |
| p+c | inf:-inf | 0.448 | inf | -inf | 1 |
| p-c | inf:-inf | 0.448 | inf | -inf | 1 |

The columns of this table indicate

- observable: the name of the observable
- selection: the recommended selection that guarantees the histogram bounds are as tight as possible without decreasing the bin width
- min_val and max_val: the user-provided *min_val* and *max_val*
- actual minimum and maximum: the actual bounds of the histogram as found using Monte Carlo

As McMule requires a constant number of histogram bins, one might artificially broaden the *min_val* – *max_val* range. This makes the out-of-bounds validator less powerful. It is hence recommended to include a selection hint as part of the observable specification. For example


```

integer, parameter :: nrq = 10
integer, parameter :: nrbins = 600
real(kind=prec), parameter :: &
  min_val(nrq) = (/ 0.40, 0.40, 1.0, 0.0, 0.00, 200., -1.0, -1.0, 0.40, 0.40 /)
real(kind=prec), parameter :: &
  max_val(nrq) = (/ 1.00, 1.00, pi-1, 0.15, 0.25, 700., 1.0, 1.0, 1.00, 1.00 /)

integer :: selection_hint(2*nrq) = (/ &
  51, 51, 1, 1, 1, 120, 107, 107, 49, 49, &
  -1, -1, -1, -1, -1, -1, 492, 492, -1, -1 /)

```

The `selection_hint` is an array with $2N$ entries listing first the indices of the first bins one wishes to use and then the indices of the last bins where `-1` defaults to the last bin. In the above example, the first histogram is defined to for the range `[0.40, 1.00]`. However, the cuts will mean that the first 50 bins `[0.40, 0.45]` are empty. Providing the `selection_hint` ensures `pymule` knows about this.

3.3.2 Swap symmetry validator

The swap symmetry validator checks that two indistinguishable particles can be swapped without the observables changing. In this example we look at the `ee2ee0` *which_piece* which has 2 distinguishable electrons in and 2 indistinguishable electrons out. Taking `S` to be our *measurement function* we want to check

$$S(e_1, e_2) = S(e_2, e_1),$$

i.e. that the *measurement function* is the same after swapping the indistinguishable electrons.

An example *measurement function* which fails this test is given below, along with the validator being called on the code,

Listing 3.1: An example *measurement function* which is not swap symmetry safe

```

58 ! This should fail swap symmetry !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60   ! q1 -> e_in_1, q2 -> e_in_2 | q3 -> e_out_1, q4 -> e_out_2
61   real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62   real (kind=prec) :: q1_rf(4), q2_rf(4), q3_rf(4), q4_rf(4)
63   real (kind=prec) :: quant(nr_q)
64   !! ===== keep the line below in any case ===== !!
65   call fix_mu
66
67   q1_rf = BOOST_RF(q1, q1)
68   q2_rf = BOOST_rf(q1, q2)
69   q3_rf = BOOST_rf(q1, q3)
70   q4_rf = BOOST_rf(q1, q4)
71
72   names(1) = 'Ee1'
73   quant(1) = q3_rf(4)
74
75   names(2) = 'Ee2'
76   quant(2) = q4_rf(4)
77
78 END FUNCTION QUANT

```

```

$ pymule validate -w ee2ee0 -l INFO FailsSwapSymmetry.so
...
INFO:pymule.validator.validator:Testing whichpiece: ee2ee0 with 8 tests
ERROR:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... failed fatally,
INFO:pymule.validator.validator:Test 2: LorentzBoostValidator(0)... passed
INFO:pymule.validator.validator:Test 3: LorentzBoostValidator(1)... passed
INFO:pymule.validator.validator:Test 4: RotationValidator(4.224052347507555, 0, 0)...
↳passed
INFO:pymule.validator.validator:Test 5: RotationValidator(1.7262535645602621, 3.
↳5353046863936015, 5.066130454491858)... passed
INFO:pymule.validator.validator:Test 6: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 7: FlipValidator(y)... passed
INFO:pymule.validator.validator:Test 8: FlipValidator(z)... passed
ERROR:pymule.validator.validator:Overall pass rate: 7 / 8 with 1 fatal failure! 87.5%

```

The *measurement function* correctly chooses a frame (the restframe of q1) so passes the LorentzBoostValidators (Tests 2 and 3). However it directly looks at the energies of the two electrons and assigns them to electron 1 and electron 2. This is something which could not be done in the real world since they are indistinguishable. The proper way of doing this would be looking at the higher energy electron and lower energy electron:

Listing 3.2: An example *measurement function* which is swap symmetry safe

```

58 ! This should fail swap symmetry !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60   ! q1 -> e_in_1, q2 -> e_in_2 | q3 -> e_out_1, q4 -> e_out_2
61   real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62   real (kind=prec) :: q1_rf(4), q2_rf(4), q3_rf(4), q4_rf(4), qHigh(4), qLow(4)
63   real (kind=prec) :: quant(nr_q)
64   !! ==== keep the line below in any case ==== !!
65   call fix_mu
66
67   q1_rf = BOOST_RF(q1, q1)
68   q2_rf = BOOST_RF(q1, q2)
69   q3_rf = BOOST_RF(q1, q3)
70   q4_rf = BOOST_RF(q1, q4)
71
72   if (q3_rf(4) > q4_rf(4)) then
73     qHigh = q3_rf; qLow = q4_rf
74   else
75     qHigh = q4_rf; qLow = q3_rf
76   endif
77
78   names(1) = 'High'
79   quant(1) = qHigh(4)
80
81   names(2) = 'Low'
82   quant(2) = qLow(4)
83
84 END FUNCTION QUANT

```

```

$ pymule validate -w ee2ee0 -l INFO PassesSwapSymmetry.so
...

```

(continues on next page)

(continued from previous page)

```

INFO:pymule.validator.validator:Testing whichpiece: ee2ee0 with 8 tests
INFO:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... passed
INFO:pymule.validator.validator:Test 2: LorentzBoostValidator(0)... passed
INFO:pymule.validator.validator:Test 3: LorentzBoostValidator(1)... passed
INFO:pymule.validator.validator:Test 4: RotationValidator(2.052666305609015, 0, 0)...
↳passed
INFO:pymule.validator.validator:Test 5: RotationValidator(1.1268264348854389, 5.
↳94661892034897, 1.2312470761319962)... passed
INFO:pymule.validator.validator:Test 6: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 7: FlipValidator(y)... passed
INFO:pymule.validator.validator:Test 8: FlipValidator(z)... passed
INFO:pymule.validator.validator:Overall pass rate: 8 / 8, 100.0%

```

Looking at the higher and lower energy electrons (instead of electron 1 and electron 2) means that swapping them has no effect on the observable, so the swap symmetry validator passes.

3.3.3 Lorentz boost validator

In the previous section the *measurement function* very deliberately chose to boost into the rest frame of the first incoming electron. This means that no matter how the system was boosted before this point, a frame is chosen and boosted into. This overwrites any previous boosts and gives a unique frame.

Note: LorentzBoostValidator(0) refers to testing the system by boosting into the rest frame of the zeroth particle (i.e. q1 in the *measurement function*)

Listing 3.3: An example *measurement function* which is not lorentz boost safe

```

58 ! This should fail Lorentz Boost !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60   ! q1 -> e_in_1, q2 -> e_in_2 | q3 -> e_out_1, q4 -> e_out_2
61   real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62   real (kind=prec) :: qHigh(4), qLow(4)
63   real (kind=prec) :: quant(nr_q)
64   !! ==== keep the line below in any case ==== !!
65   call fix_mu
66
67   if (q3(4) > q4(4)) then
68     qHigh = q3; qLow = q4
69   else
70     qHigh = q4; qLow = q3
71   endif
72
73   names(1) = 'High'
74   quant(1) = qHigh(4)
75
76   names(2) = 'Low'
77   quant(2) = qLow(4)
78
79 END FUNCTION QUANT

```

```

$ pymule validate -w ee2ee0 -l INFO FailsLorentzBoost.so
...
INFO:pymule.validator.validator:Testing whichpiece: ee2ee0 with 8 tests
INFO:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... passed
WARNING:pymule.validator.validator:Test 2: LorentzBoostValidator(0)... failed, assuming
↳the center of mass frame is allowed but not advisable
WARNING:pymule.validator.validator:Test 3: LorentzBoostValidator(1)... failed, assuming
↳the center of mass frame is allowed but not advisable
INFO:pymule.validator.validator:Test 4: RotationValidator(4.354392616889299, 0, 0)...
↳passed
INFO:pymule.validator.validator:Test 5: RotationValidator(4.167936819187568, 5.
↳129621429511378, 1.185060876837443)... passed
INFO:pymule.validator.validator:Test 6: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 7: FlipValidator(y)... passed
INFO:pymule.validator.validator:Test 8: FlipValidator(z)... passed
WARNING:pymule.validator.validator:Overall pass rate: 6 / 8, 75.0%

```

Choosing a frame to boost into (like in the swap symmetry example) means that the test no longer fails.

Listing 3.4: An example *measurement function* which is Lorentz boost safe

```

58  ! This should fail swap symmetry !
59  FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60      ! q1 -> e_in_1, q2 -> e_in_2 | q3 -> e_out_1, q4 -> e_out_2
61      real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62      real (kind=prec) :: q1_rf(4), q2_rf(4), q3_rf(4), q4_rf(4), qHigh(4), qLow(4)
63      real (kind=prec) :: quant(nr_q)
64      !! ==== keep the line below in any case ==== !!
65      call fix_mu
66
67      q1_rf = BOOST_RF(q1, q1)
68      q2_rf = BOOST_RF(q1, q2)
69      q3_rf = BOOST_RF(q1, q3)
70      q4_rf = BOOST_RF(q1, q4)
71
72      if (q3_rf(4) > q4_rf(4)) then
73          qHigh = q3_rf; qLow = q4_rf
74      else
75          qHigh = q4_rf; qLow = q3_rf
76      endif
77
78      names(1) = 'High'
79      quant(1) = qHigh(4)
80
81      names(2) = 'Low'
82      quant(2) = qLow(4)
83
84      END FUNCTION QUANT

```

```

$ pymule validate -w ee2ee0 -l INFO FailsLorentzBoost.so
...
INFO:pymule.validator.validator:Testing whichpiece: ee2ee0 with 8 tests

```

(continues on next page)

(continued from previous page)

```

INFO:pymule.validator.validator:Test 1: SwapSymmetryValidator(2, 3)... passed
INFO:pymule.validator.validator:Test 2: LorentzBoostValidator(0)... passed
INFO:pymule.validator.validator:Test 3: LorentzBoostValidator(1)... passed
INFO:pymule.validator.validator:Test 4: RotationValidator(0.20776147014583715, 0, 0)...
↳passed
INFO:pymule.validator.validator:Test 5: RotationValidator(0.21958279258314367, 1.
↳470413694055576, 1.8065232010826624)... passed
INFO:pymule.validator.validator:Test 6: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 7: FlipValidator(y)... passed
INFO:pymule.validator.validator:Test 8: FlipValidator(z)... passed
INFO:pymule.validator.validator:Overall pass rate: 8 / 8, 100.0%

```

3.3.4 Soft limit validator and soft-swap validator

A soft photon is one whose energy can be taken to zero. For example in the *which_piece* $m2enngR$ ($\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu \gamma \gamma$) there is one soft photon (represented by the R) and one always real photon (represented by g).

Mathematically, in the soft limit validator we are testing,

$$\lim_{\gamma \rightarrow 0} S(q, \gamma) = S(q),$$

and in the case of the soft-swap validator we are testing,

$$\lim_{\gamma_1 \rightarrow 0} S(q, \gamma_1, \gamma_2) = \lim_{\gamma_2 \rightarrow 0} S(q, \gamma_2, \gamma_1) = S(q, \gamma_2)$$

where S is our *measurement function*.

An example of failing this test is looking directly at number of photons (or experimentally number of photons detected)

If we say a photon exists if it has any energy at all then we could write

Listing 3.5: An example *measurement function* which is not soft limit safe

```

58 ! This should fail soft limit !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60   ! m(q1) -> e(q2) n(q3) n(q4) g(q5) R/0(q6)
61   real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62   real (kind=prec) :: quant(nr_q)
63   integer :: numphotons
64   !! ==== keep the line below in any case ==== !!
65   call fix_mu
66
67   numphotons = 0
68
69   if (q5(4) > 0) then
70     numphotons = numphotons + 1
71   endif
72   if (q6(4) > 0) then
73     numphotons = numphotons + 1
74   endif
75
76   names(1) = 'numpho'

```

(continues on next page)

(continued from previous page)

```

77     quant(1) = numphotons
78
79     names(2) = 'const'
80     quant(2) = 5
81
82 END FUNCTION QUANT

```

```

$ pymule validate -w m2enngR -l INFO FailsSoftLimit.so
...
INFO:pymule.validator.validator:Testing whichpiece: m2enngR with 9 tests
# Graph appears!
ERROR:pymule.validator.validator:Test 1: SoftLimitValidator(0)... failed fatally,
INFO:pymule.validator.validator:Test 2: SwapSymmetryValidator(2, 3)... passed
INFO:pymule.validator.validator:Test 3: SwapSymmetryValidator(4, 5)... passed
# Another graph appears!
ERROR:pymule.validator.validator:Test 4: SoftSwapValidator(0, 4, 5)... failed fatally,
...
ERROR:pymule.validator.validator:Overall pass rate: 7 / 9 with 2 fatal failures! 77.78%

```

The failure occurs because at the photon energy proportion $\xi = 0$, the soft photon has 0 energy so only one photon exists so `numphotons = 1`. At *any* higher energy this photon is counted so `numphotons = 2`. This discrepancy means the limit above doesn't hold, so the test fails. Notice that there is some additional functionality when a soft limit test (or soft-swap test) fails. The graph that is generated shows which of the observables caused the test to fail. In this case `numpho` is shown but not `const`. We define the $Z = \left\| \frac{S(\xi) - S(0)}{tol} \right\|$ where tol is a relative tolerance to the value $S(0)$ (usually $S(0) \times 10^{-3}$) plus an absolute tolerance which is used for $S(0) = 0$ (usually also 10^{-3}).

Figure 3.1: There is a line at $Z = 1$ indicating that a value must (consistently) fall below that line in order to be considered smooth and going to the expected limit. Note that `numpho` is shown but not `const`.

A *measurement function* more inline with physical reality is one where the photon is not counted if it is below some threshold, e.g. the detection limit of the detector.

Listing 3.6: An example *measurement function* which is soft limit safe

```

58 ! This should pass soft limit !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60     ! m(q1) -> e(q2) n(q3) n(q4) g(q5) R/0(q6)
61     real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62     real (kind=prec) :: quant(nr_q)
63     integer :: numphotons
64     !! ===== keep the line below in any case ===== !!
65     call fix_mu
66
67     numphotons = 0
68
69     if (q5(4) > 1e-3) then ! NOTE: threshold for counting the photon
70         numphotons = numphotons + 1
71     endif
72     if (q6(4) > 1e-3) then ! NOTE: threshold for counting the photon
73         numphotons = numphotons + 1
74     endif
75

```

(continues on next page)

(continued from previous page)

```

76 names(1) = 'numpho'
77 quant(1) = numphotons
78
79 names(2) = 'const'
80 quant(2) = 5
81
82 END FUNCTION QUANT

```

```

$ pymule validate -w m2enngR -l INFO PassesSoftLimit.so
...
INFO:pymule.validator.validator:Testing whichpiece: m2enngR with 9 tests
INFO:pymule.validator.validator:Test 1: SoftLimitValidator(0)... passed
INFO:pymule.validator.validator:Test 2: SwapSymmetryValidator(2, 3)... passed
INFO:pymule.validator.validator:Test 3: SwapSymmetryValidator(4, 5)... passed
INFO:pymule.validator.validator:Test 4: SoftSwapValidator(0, 4, 5)... passed
...

```

If we still wish to see the graph (for example if it shouldn't have passed but did) we can send the DEBUG flag to the logging and force the graph to be displayed.

```

$ pymule validate -w m2enngR -l DEBUG PassesSoftLimit.so
...
DEBUG:matplotlib ...
...
# Graph appears!
INFO:pymule.validator.validator:Test 1: SoftLimitValidator(0)... passed
...

```

There is unfortunately quite a bit of matplotlib spam (In the works to be fixed!) but the following graph is produced.

Figure 3.2: We can see that at around $\xi = 10^{-5}$ the graph starts to dip down. Numerically numpho has gone to 1 but the Richardson extrapolation which is used reach the limit faster for other observables lags slightly. After the value has been below the $Z = 1$ threshold for three iterations it is said to converge. Note also that the observable const is also in the legend of the graph.

Note: DEBUG will force all observables onto the graph regardless of if they pass or not. In this case all values of const are at $Z = 0$ which is not displayed on the log-log plot.

In the m2enngR *which_piece* the energy to be passed around is $E \approx 100$ MeV, meaning the cut off for the photon counting happens at $E\xi = 10^{-3}$ MeV, $\xi \approx 10^{-5}$ as seen in the graph.

3.3.5 Rotation validator and Flip validator

These two tests are bundled together as it is difficult to only fail one and not both. When calling the validator two rotation tests are performed. The first involves only rotating around the Z-axis. This will be passed if, for example, the observable only looks at the momentum perpendicular to the Z-axis. But the test would be failed if the observable looks for example only at the momentum in the X-axis. The second rotation test is a rotation of an arbitrary set of angles, designed to test complete rotational invariance.

The flip tests are designed to check the symmetry of flipping the system in the X, Y, and Z axes (individually).

All these tests are *non-fatal*, meaning that failing them is not necessarily bad as McMule will not be performing arbitrary transformations of the phase space. Failing them however is still to be discouraged, if only for completeness.

An example failing the complete rotation test, but not the perpendicular rotation test is

Listing 3.7: An example *measurement function* which is not rotation safe

```

58 ! This should fail total rotation but pass Z rotation !
59 FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
60   ! m(q1) 2 e(q2) n(q3) n(q4) !
61   real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
62   real (kind=prec) :: qHigh(4), qLow(4)
63   real (kind=prec) :: quant(nr_q)
64   !! ==== keep the line below in any case ==== !!
65   call fix_mu
66
67   names(1) = 'ElPerp'
68   quant(1) = sqrt(q2(1)**2 + q2(2)**2)
69
70   names(2) = 'ElLong'
71   quant(2) = q2(3)
72
73 END FUNCTION QUANT
74
75
76 SUBROUTINE USEREVENT(X, NDIM)
77   integer :: ndim
78   real(kind=prec) :: x(ndim)
79   userweight = 1.
80 END SUBROUTINE USEREVENT

```

```

$ pymule validate -w m2enngR -l INFO FailsRotation.so
...
INFO:pymule.validator.validator:Test 5: RotationValidator(1.7834175407965058, 0, 0)...
↳passed
WARNING:pymule.validator.validator:Test 6: RotationValidator(4.772496460705239, 3.
↳8977085904094153, 2.9542522534242135)... failed,
INFO:pymule.validator.validator:Test 7: FlipValidator(x)... passed
INFO:pymule.validator.validator:Test 8: FlipValidator(y)... passed
WARNING:pymule.validator.validator:Test 9: FlipValidator(z)... failed,
WARNING:pymule.validator.validator:Overall pass rate: 7 / 9, 77.78%

```

As we can see, the rotation around Z (Test 5) passed as expected and the arbitrary rotation (Test 6) failed as expected. The flip validator across the Z axis also fails. This is because when looking at the longitudinal momentum the Z momentum is used and will flip sign, thus changing the observable and failing the test. An easy solution to this is to

replace line 71 with

```
quant(2) = abs(q2(3))
```

thus becoming invariant under flipping across Z .

3.3.6 Minimum and Maximum values, and bin-width testing

Currently the minimum (*min_val*) and maximum values (*max_val*) for the histogram as well as the bin widths (determined by number of bins *nrbins*) are decided by the user before knowing the shape of the data output in the process. This can lead to a time consuming trial-and-error of runs until the correct histogram shape and size is established.

It is intended in the validator to have an additional test which generates many random results and sees if the parameters for the histogram are suitable. This will still (potentially) be a trial-and-error process but will be much quicker than setting up and running the entirety of McMule.

3.4 Distance function tests in detail

A *distance function* may optionally be included in the *userfile*. This function is used to determine the distance between two points in the momentum phasespace. The simplest example of this would perhaps be a euclidean distance between the spatial components of all the momenta. This *distance function* must obey certain criteria however.

3.4.1 Triangle validator

A test of the triangle inequality such that:

$$ab + c,$$

where a is the direct distance between two events, and b and c are distances between a third event to the first two.

3.4.2 Event Swap validator

The distance from event 1 (e_1) to event 2 (e_2) should be equal to the distance from event 2 to event 1. If D is our distance function then:

$$D(e_1, e_2) = D(e_2, e_1),$$

in a similar way to the particle swapping in *Swap symmetry validator*

3.4.3 Same event validator

If an event is the same as (indistinguishable to) another event then the distance between these two events should be zero.

Chapter 4

General aspects of using McMule

In this section, we will collect a few general points of interest regarding McMule. In particular, we will discuss heuristics on how much statistics is necessary for different contributions in Section `sec_stat`. This is followed by a more in-depth discussion of the analysis strategy in Section *Analysis*.

4.1 Flavour

McMule implements *generic processes* that are connected to physical processes by setting the masses of the particles. For example, `m2enng0` can refer to $\mu \rightarrow e\nu\bar{\nu}\gamma$, $\tau \rightarrow e\nu\bar{\nu}\gamma$, or $\tau \rightarrow \mu\nu\bar{\nu}\gamma$, depending on the values of `Mm` and `Me`. These parameters are controlled by `flavour` as set during `initflavour()`. Further, `initflavour()` also sets the centre-of-mass energy `scms` for $2 \rightarrow n$ processes. We provide flavours for all relevant combinations as well as a few presets for `scms` as shown in Table 4.1. For processes that support it, appending a + or - can switch the charge of one of the involved flavours.

Table 4.1: The different flavour in McMule

| Flavour | Mapping | | | Default centre-of-mass energy |
|----------|---------|--------|-------|---|
| | t | m | e | |
| mu-e | | μ | e | no default |
| tau-mu | | τ | μ | no default |
| tau-e | | τ | e | no default |
| tau-mue | τ | μ | e | no default |
| tau-emu | τ | e | μ | no default |
| mu-0 | | μ | 0 | no default |
| tau-0 | | τ | 0 | no default |
| e-p | | p | e | no default |
| mu-p | | p | μ | no default |
| muone | | μ | e | $m_e^2 + m_\mu^2 + 2m_e \times 150 \text{ GeV}$ |
| mesa | | p | e | $m_e^2 + m_p^2 + 2m_p \times 155 \text{ MeV}$ |
| belleTau | | τ | e | $M_{\Upsilon(4S)}^2$ |

4.2 Statistics

McMule is a Monte Carlo program. This means it samples the integrand at N (pseudo-)random points to get an estimate for the integral. However, because it uses the adaptive Monte Carlo integration routine `vegas` [15], we split $N = i \times n$ into i iterations (`itmx`), each with n points (`nca11`). After each iteration, `vegas` changes the way it will sample the next iteration based on the results of the previous one. Hence, the performance of the integration is a subtle interplay between i and n – it is not sufficient any more to consider their product N .

Further, we always perform the integration in two steps: a pre-conditioning with $i_{\text{ad}} \times n_{\text{ad}}$ (`nca11_ad` and `itmx_ad`, respectively), that is used to optimise the integration strategy and after which the result is discarded, and a main integration that benefits from the integrator’s understanding of the integrand.

Of course there are no one-size-fits-all rules of how to choose the i and n for pre-conditioning and main run. However, the following heuristics have proven helpful:

- n is always much larger than i . For very simple integrands, $n = \mathcal{O}(10 \cdot 10^3)$ and $i = \mathcal{O}(10)$.
- Increasing n reduces errors that can be thought of as systematic because it allows the integrator to ‘discover’ new features of the integrand. Increasing i on the other hand will rarely have that effect and only improves the statistical error. This is especially true for distributions
- There is no real limit on n , except that it has to fit into the datatype used – integrations with $n = \mathcal{O}(2^{31} - 1)$ are not too uncommon – while i is rarely (much) larger than 100.
- For very stringent cuts it can happen that that typical values of n_{ad} are too small for any point to pass the cuts. In this case `vegas` will return NaN, indicating that no events were found. Barring mistakes in the definition of the cuts, a pre-pre-conditioning with extremely large n but $i = 1 - 2$ can be helpful.
- n also needs to be large enough for `vegas` to reliably find all features of the integrand. It is rarely obvious that it did, though sometimes it becomes clear when increasing n or looking at intermediary results as a function of the already-completed iterations.
- The main run should always have larger i and n than the pre-conditioning. Judging how much more is a delicate game though $i/i_{\text{ad}} = \mathcal{O}(5)$ and $n/n_{\text{ad}} = \mathcal{O}(10 - 50)$ have been proven helpful.
- If, once the integration is completed, the result is unsatisfactory, take into account the following strategies
 - A large $\chi^2/\text{d.o.f.}$ indicates a too small n . Try to increase n_{ad} and, to a perhaps lesser extent, n .
 - Increase i . Often it is a good idea to consciously set i to a value so large that the integrator will never reach it and to keep looking at ‘intermediary’ results.
 - If the error is small enough for the application but the result seems incorrect (for example because the ξ_c dependence does not vanish), massively increase n .
- Real corrections need much more statistics in both i and n ($\mathcal{O}(10)$ times more for n , $\mathcal{O}(2)$ for i) than the corresponding *LO* calculations because of the higher-dimensional phase-space.
- Virtual corrections have the same number of dimensions as the *LO* calculation and can go by with only a modest increase to account for the added functional complexity.
- `vegas` tends to underestimate the numerical error.

These guidelines are often helpful but should not be considered infallible as they are just that – guidelines.

McMule is not parallelised; however, because Monte Carlo integrations require a *random seed* anyway, it is possible to calculate multiple estimates of the same integral using different *random seeds* z_1 and combining the results obtained this way. This also allows to for a better, more reliable understanding of the error estimate.

4.3 Analysis

Once the Monte Carlo has run, an offline analysis of the results is required. This entails loading, averaging, and combining the data. This is automatised in `pymule` but the basic steps are

0. Load the data into a suitable analysis framework such as `python`.
1. Combine the different *random seeds* into one result per contribution and ξ_c . The $\chi^2/\text{d.o.f.}$ of this merging must be small. Otherwise, try to increase the statistics or choose of different phase-space parametrisation.
2. Add all contributions that combine into one of the physical contributions (10.11). This includes any partitioning done in Section *Phase-space generation*.
3. (optional) At $N^\ell\text{LO}$, perform a fit¹

$$\sigma_{n+j}^{(\ell)} = c_0^{(j)} + c_1^{(j)} \log \xi_c + c_2^{(j)} \log^2 \xi_c + \cdots + c_\ell^{(j)} \log^\ell \xi_c = \sum_{i=0}^{\ell} c_i^{(j)} \log^i \xi_c \quad (4.1)$$

This has the advantage that it very clearly quantifies any residual ξ_c dependence. We will come back to this issue in Section *Study of ξ_c dependence*.

4. Combine all physical contributions of (10.10) into $\sigma^{(\ell)}(\xi_c)$ which has to be ξ_c independent.
5. Perform detailed checks on ξ_c independence. This is especially important on the first time a particular configuration is run. Beyond *NLO*, it is also extremely helpful to check whether the sum of the fits (4.1) is compatible with a constant, i.e. whether for all $1 \leq i \leq \ell$

$$\left| \frac{\sum_{j=0}^{\ell} c_i^{(j)}}{\sum_{j=0}^{\ell} \delta c_i^{(j)}} \right| < 1 \quad (4.2)$$

where $\delta c_i^{(j)}$ is the error estimate on the coefficient $c_i^{(j)}$.² `pymule`'s `mergefkswithplot()` can be helpful here.

If (4.2) is not satisfied or only very poorly, try to run the Monte Carlo again with an increased n .

6. Merge the different estimates of (10.10) from the different ξ_c into one final number $\sigma^{(\ell)}$. The $\chi^2/\text{d.o.f.}$ of this merging must be small.
7. Repeat the above for any distributions produced, though often bin-wise fitting as in Point 3 is rarely necessary or helpful.

If a total cross section is ξ_c independent but the distributions (or a cross section obtained after applying cuts) are not, this is a hint that the distribution (or the applied cuts) is not *IR* safe.

These steps have been almost completely automatised in `pymule` and `Mathematica`. Though all steps of this pipeline could be easily implemented in any other language by following the specification of the file format below (Section *Differential distributions and intermediary state files*).

¹ Note that it is important to perform the fit after combining the phase-space partitionings (cf. Section *Phase-space generation*) but before adding (10.10) as this model is only valid for the terms of (10.11)

² Note that the error estimate on the sum of the total coefficients in (4.2) is rather poor and does not include correlations between different c_i .

4.4 Manual compilation

You might need to compile McMule manually if you are not using a sufficiently recent Linux distribution or want to work it on yourself. In this case, you first need to obtain a copy of the McMule source code. We recommend the following approach

```
$ git clone --recursive https://gitlab.com/mule-tools/mcmule
```

To build McMule, you will need

- Python 3.8 or newer
- Meson 0.64.0 or newer
- ninja 1.8.2 or newer
- GFortran 4.8 or newer

Now you need to configure and build McMule using meson and ninja

```
$ meson setup build  
$ ninja -C build
```

Note that this will distribute the build on as many CPUs as your machine has which can cause memory issues. If you do not want to do that, add `-j <number of jobs>` flag to the ninja command. Despite the parallelisation, a full build of McMule is can take up to 1h, depending on your machine. If you only need to compile some parts of McMule (such as Bhabha scattering), you can control which *process groups* are build

```
$ meson setup build -Dgroups=mue,ee
```

If you need debug symbols, you can disable optimisation

```
$ meson setup build --buildtype=debug
```

Alternatively, we provide a Docker *container* [17] for easy deployment and legacy results (cf. Section *Basics of containerisation*). In multi-user environments, *udocker* [12] can be used instead. In either case, a pre-compiled copy of the code can be obtained by calling

```
$ docker pull registry.gitlab.com/mule-tools/mcmule # requires Docker to be installed  
$ udocker pull registry.gitlab.com/mule-tools/mcmule # requires uDocker to be installed
```

4.4.1 Running in a container

Linux *containers* are an emergent new technology in the software engineering world. The main idea behind such *containerisation* is to bundle all dependencies with a software when shipping. This allows the software to be executed regardless of the Linux distribution running on the host system without having to install any software beyond the containerising tool. This is possible without any measurable loss in performance. For these reasons, containerising McMule allows the code to be easily deployed on any modern computer, including systems running macOS or Windows (albeit with a loss of performance), and all results to be perfectly reproducible.

A popular *containerisation* tool is Docker [17]. Unfortunately, Docker requires some processes to be executed in privileged mode which is rarely available in the multi-user environments usually found on computing infrastructures. This led to the creation of *udocker* [12] which circumvents these problems.

udocker can be installed by calling

Warning: It might be advisable to point the variable `UDOCKER_DIR` to a folder on a drive without quota first as *udocker* requires sizeable disk space

```
$ curl https://raw.githubusercontent.com/indigo-dc/udocker/master/udocker.py > udocker
$ chmod u+rx ./udocker
$ ./udocker install
```

Once Docker or *udocker* has been installed, McMule can be downloaded by simply calling

```
$ docker pull yulrich/mcmule # requires Docker to be installed
$ udocker pull yulrich/mcmule # requires udocker to be installed
```

This automatically fetches the latest public release of McMule deemed stable by the McMule collaboration. We will discuss some technical details behind *containerisation* in Section *Basics of containerisation*.

McMule can be run containerised on a specified `user.f95` which is compiled automatically into `mcmule`. This is possible both directly or using *menu files* as discussed above. To run McMule directly on a specified `user.f95`, simply call

```
$ ./tools/run-docker.sh -i yulrich/mcmule:latest -u path/to/user.f95 -r
```

This requests the same input already discussed in Table 1.1. To run a *containerised menu file*, add an `image` command before the first `conf` command in the *menu file*

```
image yulrich/mcmule:latest path/to/user.f95
conf m2enng.toml

run 70998 0.500000 m2enngR tau-e 0
...
```

Note that only one `image` command per *menu file* is allowed. After this, the *menu file* can be executed normally though the drive where Docker or *udocker* is installed needs to be shared between all nodes working on the job. It is recommended that all legacy results use be produced with *udocker* or Docker.

Chapter 5

The menu file and pymule batch

To avoid having to run dozens of jobs manually, McMule uses *menu files*. These files list all McMule jobs that need to be executed which is done by pymule. Currently pymule supports two *executors*: `shell` for running on a single machine and `slurm` for running on a SLURM cluster. To run a *menu file*, one either runs

```
pymule batch shell -np <number of cores> menu-m2enng.menu # using shell executor
pymule batch slurm menu-m2enng.menu # using slurm executor as part of a SLURM script
```

5.1 Menu file syntax

Menu files use a *DSL* to minimise effort. Statements are separated by line breaks similar to python or bash. Blocks are indented, again similar to python. Variables are indicated with dollar signs, eg. `$x`

conf `</path/to/config.toml>`

loads the variables of a *config file* to use in the following runs. *config file* can be (partially) overridden with a second `conf` call

image `<image name>`

loads the docker *image* to run in

mode `<mode>`

sets the running mode of McMule. Currently only `vegas` is supported

user `<path/to/user.so>`

sets the measurement function to be executed

run `<seed> <xic> <piece> [optional arguments..]`

executes `mcmule` with the currently active variables for the specified *which_piece*.

let `<var> = <val>`

sets the variable `var` to the value `val` in all following commands. Note that `var` does not contain a dollar sign here

for `<var> in <a> <c>...`

executes the body of this block with `$var = a`, `$var = b`, `$var = c`.

if `<$var> <op> <value>`

executes the body of this block only if the expression evaluates to true. valid operators are `==`, `<=`, `<`, `>`, `>=`, and `!=`.

An example can be found in Listing

```
let a = 3
conf ../manual/example2/m2enng.toml
mode vegas

run $a 0.3 m2enng0 0

for b in 0.1 0.2 0.3 0.4
  run $a $b m2enngR 0
  if $b == 0.3
    run $a $b m2enngR 1
```

5.2 Configuration file

The *config file* is written using *TOML* and can be used to set arbitrary variables. However, most commonly used to set statistics and the name of the `mcmule` executable.

Chapter 6

libmcmule user guide

McMule's Fortran code has hundreds of functions and subroutine and we will not document all of them here. However, we will list those exposed by the `mcmule` module.

McMule uses a custom type for real numbers that usually defaults to double precision and is used in the following

type real (kind=prec) [*fixed*]

The real number type used in McMule. This cannot be changed at runtime by the user but should be used for all interactions with the code. It usually refers to double precision

Momenta in McMule are of the form (`/ px, py, pz, E /`) though it is often advisable to instead use the functions defined in Section *Scalar quantities*.

6.1 Parameters, variables, and functions of libmcmule

The main module of McMule, `mcmule`, exposes the following parameters, variables, and functions to the user

Some parameters are denoted as *fixed* (cannot be changed without recompiling McMule) or *protected* (can be changed only through the use of functions)

6.1.1 Variables to be set by the user file

pass_cut (*nr_q*) [*logical*]

This controls whether the event is acceptable. If at least one entry of this array is `.true.` the event will be calculated and added to the cross section. If individual elements are `.false.`, this event will *not* be added to the corresponding histogram. `mcmule` will ensure that the size of this array is correct. Exposed as `mcmule_pass_cut` in C.

names (*nr_q*) [*character(len=namesLen)*]

The names of the distributions the user wishes the calculate. `mcmule` will ensure that the size of this array is correct Exposed as `mcmule_names` in C.

Note: Even though it is possible to calculate multiple closely related cuts simultaneously, this can harm the speed of convergence as the VEGAS algorithm optimises for the cross section and not for the distributions.

musq [*real(kind=prec)*]

The renormalisation scale μ^2 . This variable *needs* to be set by the user, otherwise McMule will fail. Exposed as `mcmule_musq` in C.

pol1 (4) [*real(kind=prec)*]

The polarisation of the first polarised particle. Exposed as `mcmule_pol1` in C.

pol2 (4) [*real(kind=prec)*]

The polarisation of the second polarised particle Exposed as `mcmule_pol2` in C.

filenamesuffix [*character(len=filenamesuffixLen)*]

The observable-specific suffix to the vegas file Exposed as `mcmule_filenamesuffix` in C.

userweight [*real(kind=prec)*]

The weight the user wishes to attach to a given event, may be set by `user.userevent()`. Exposed as `mcmule_userweight` in C.

6.1.2 Couplings and masses

Couplings are (usually) hard-coded in McMule and cannot be changed. Masses can be changed using the `initflavour()` routine.

GF [*real(kind=prec),fixed*]

The Fermi constant. For predominantly historic reasons, this is set to `1._prec`. Exposed as `mcmule_Gf` in C.

alpha [*real(kind=prec),fixed*]

The fine-structure constant in the *OS* scheme. For predominantly historic reasons, this is set to `1._prec`. Exposed as `mcmule_alpha` in C.

sw2 [*real(kind=prec)*]

The weak mixing angle $\sin(\theta_W)^2$. This can be changed by the user at runtime to modify the *EW* scheme that is used. Exposed as `mcmule_sw2` in C.

Me1 [*real(kind=prec),fixed*]

The numerical value of the electron mass in MeV, irregardless of the *flavour*

Mmu [*real(kind=prec),fixed*]

The numerical value of the muon mass in MeV, irregardless of the *flavour*

Mtau [*real(kind=prec),fixed*]

The numerical value of the tau mass in MeV, irregardless of the *flavour*

Mproton [*real(kind=prec),fixed*]

The numerical value of the proton mass in MeV, irregardless of the *flavour*

MZ [*real(kind=prec),fixed*]

The numerical value of the Z boson mass in MeV

Mm [*real(kind=prec),protected*]

The actual value of the *m* particle, usually the muon mass but if *flavour* is eg. tau-e the tau mass. Exposed as `mcmule_Mm` in C.

Me [*real(kind=prec),protected*]

The actual value of the *e* particle, usually the electron mass but if *flavour* is eg. tau-mu the muon mass. Exposed as `mcmule_Me` in C.

Mt [*real(kind=prec),protected*]

The actual value of the t particle, usually the tau mass Exposed as *mcmule_Mtau* in C.

scms [*real(kind=prec),protected*]

The numerical value of the centre-of-mass energy Exposed as *mcmule_scms* in C.

subroutine initflavour(*flavour, scms*)

This sets the values of the masses and centre-of-mass energy as per Section *Flavour*.

Parameters

- **flavour** [*character(len=*)*] :: the flavour to use, will be written into *flavour*
- **scms** [*real(kind=prec),optional*] :: the Mandelstam $s = (p_1 + p_2)^2$ to use.

Warning: Calling this function at run-time is possible (eg. during *userevent()*) but requires updating of masses (cf. [Issue#57](#))

Mj [*real(kind=prec)*]

The mass of a New Physics particle. Exposed as *mcmule_Mj* in C.

Warning: Calling this function at run-time is possible (eg. during *userevent()*) but requires updating of masses (cf. [Issue#57](#))

CLj, CRj

The values of the left- and right-handed $J\bar{\psi}\psi$ coupling Exposed as *mcmule_CLj* and *mcmule_CRj* in C.

the_proton_ff [*procedure,pointer*]

A function pointer to the proton form factor. This needs to be of the form

```
subroutine proton_formfactor(Q2, Gel, Gmag)
real(kind=prec), intent(in) :: Q2
real(kind=prec), intent(out) :: Gel, Gmag
...
end subroutine proton_formfactor
```

where Q2 is the momentum transfer $-(p_e - p_p)^2$ and Gel and Gmag are the Sachs form factors.

lambda [*real(kind=prec)*]

When using a dipole or monopole proton form factor, this is the width of the dipole in MeV².

kappa [*real(kind=prec)*]

When using a dipole or monopole proton form factor, this the proton's magnetic dipole moment

6.1.3 Scalar quantities

function `s(p1, p2) [real(kind=prec)]`

The scalar product $2p_1 \cdot p_2$

Parameters

- **p1** (4) [real(kind=prec)] :: the first momentum p_1
- **p2** (4) [real(kind=prec)] :: the second momentum p_2

function `sq(p) [real(kind=prec)]`

The Lorentz square p^2

Parameters

p (4) [real(kind=prec)] :: the momentum p

function `asymtensor(p1, p2, p3, p4) [real(kind=prec)]`

The total asymmetric tensor $\varepsilon_{\mu\nu\rho\sigma} p_1^\mu p_2^\nu p_3^\rho p_4^\sigma$

Parameters

- **p1** (4) [real(kind=prec)] :: the momentum p_1
- **p2** (4) [real(kind=prec)] :: the momentum p_2
- **p3** (4) [real(kind=prec)] :: the momentum p_3
- **p4** (4) [real(kind=prec)] :: the momentum p_4

function `cos_th(p1, p2) [real(kind=prec)]`

The cosine of the angle between the two momenta p_1 and p_2

$$\cos \theta_{12} = \frac{\vec{p}_1 \cdot \vec{p}_2}{|\vec{p}_1| |\vec{p}_2|}$$

Parameters

- **p1** (4) [real(kind=prec)] :: the first momentum p_1
- **p2** (4) [real(kind=prec)] :: the second momentum p_2

Note: This will return 0 if the computation fails

function `eta(p) [real(kind=prec)]`

The pseudorapidity w.r.t. the z axis

$$\eta = \frac{1}{2} \log \frac{|\vec{p}| + p_z}{|\vec{p}| - p_z}$$

Parameters

p (4) [real(kind=prec)] :: the momentum p

function `rap(p) [real(kind=prec)]`

The rapidity w.r.t. the z axis

$$y = \frac{1}{2} \log \frac{E + p_z}{E - p_z}$$

Parameters

p (4) [real(kind=prec)] :: the momentum p

function `pt(p)` [*real(kind=prec)*]

The transverse momentum w.r.t. the z axis

$$p_T = \sqrt{p_x^2 + p_z^2}$$

Parameters

`p` (4) [*real(kind=prec)*] :: the momentum p

function `absvec(p)` [*real(kind=prec)*]

The length of the three-vector part $|\vec{p}$

Parameters

`p` (4) [*real(kind=prec)*] :: the momentum p

function `phi(p)` [*real(kind=prec)*]

The azimuthal angle of p , $-\pi < \phi < \pi$

Parameters

`p` (4) [*real(kind=prec)*] :: the momentum p

Note: This may return 100π if the calculation fails.

function `rij(p1, p2)` [*real(kind=prec)*]

The jet distance R_{12} between the two momenta p_1 and p_2 , normalised by $D_{\text{res}} = 0.7$

$$R_{12} = \frac{\Delta y_{12}^2 + \Delta \phi_{12}^2}{D_{\text{res}}^2}$$

Parameters

- `p1` (4) [*real(kind=prec)*] :: the first momentum p_1
- `p2` (4) [*real(kind=prec)*] :: the second momentum p_2

6.1.4 Transformations

function `boost_back(rec, mo)`

boosts the momentum `mo` from the frame where `rec` is at rest to the frame where `rec` is specified, i.e.

```
boost_back(rec, (/ 0., 0., 0., sqrt(sq(rec)) /)) = rec
```

This function can be viewed as the inversion of `boost_rf()`.

Parameters

- `rec` (4) [*real(kind=prec)*] :: the system to boost into
- `mo` (4) [*real(kind=prec)*] :: the momentum to boost

Return

`boost_back` (4) [*real(kind=prec)*] :: the boosted momentum

function `boost_rf(rec, mo)`

boosts `mo` to (non-unique) rest frame of `rec`, i.e.

```
boost_rf(rec, rec) = (/ 0., 0., 0., sqrt(sq(rec)) /)
```

This function can be viewed as the inversion of `boost_back()`.

Parameters

- **rec** (4) [*real(kind=prec)*] :: the system to boost into
- **mo** (4) [*real(kind=prec)*] :: the momentum to boost

Return

boost_back (4) [*real(kind=prec)*] :: the boosted momentum

function euler_mat(*a, b, c*)

gives the Euler rotation matrix formed by rotation by α around the current z axis, then by β around the current y axis, and the by γ around the current z axis.

$$\begin{pmatrix} c_\alpha c_\beta c_\gamma - s_\alpha s_\gamma & -c_\alpha c_\beta s_\gamma - c_\gamma s_\alpha & c_\alpha s_\beta & 0 \\ c_\beta c_\gamma s_\alpha + c_\alpha s_\gamma & c_\alpha c_\gamma - c_\beta s_\alpha s_\gamma & s_\alpha s_\beta & 0 \\ -c_\gamma s_\beta & s_\beta s_\gamma & c_\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parameters

- **a** [*real(kind=prec)*] :: the angle α
- **b** [*real(kind=prec)*] :: the angle β
- **c** [*real(kind=prec)*] :: the angle γ

Return

euler_mat (4,4) [*real(kind=prec)*] :: the 4×4 Euler matrix

6.1.5 Main routine

If the user wishes to call McMule from their own executable rather than using the default `mcmule`, they may use

subroutine runmcmule(*ncall_ad, itmx_ad, ncall, itmx, initial_ran_seed, xi1, xi2, piece, flav, tfilename*)

This function initialises McMule and performs the integration. Exposed as `mcmule_runmcmule()` in C.

Parameters

- **ncall_ad** [*integer*] :: number of calls per iteration during pre-conditioning
- **itmxl_ad** [*integer*] :: number of iterations during pre-conditioning. No pre-conditioning is performed if this value is zero
- **ncall** [*integer*] :: number of calls per iteration during the main run
- **itmxl** [*integer*] :: number of iterations during the main run (cf. Section `sec_stat`).
- **initial_ran_seed** [*integer*] :: the initial random seed (cf. Section *Random number generation*).
- **xi1** [*real(kind=prec)*] :: the ξ values, either used for subtraction or the integrated eikonal. It is recommended that they are equal.
- **piece** [*character(len=*)*] :: the `mcmule/which_piece` to calculate
- **flav** [*character(len=*)*] :: the `mcmule/flavour` to use
- **tfilename** [*character(len=*),optional*] :: the filename to use for the output, autogenerated if not present

subroutine load_quant(*filename*)

Loads the observable defined in *filename* using `dl_open`

Parameters

filename [*character(len=*)*] :: the library to open

subroutine set_observable(*number_hist*, *number_bins*, *lower_bounds*, *upper_bounds*, *measurement_function*, *user_integration*, *user_initialisation*, *user_integration_dimension*, *names_length*, *filenamesuffix_length*)

Set the observable to an already loaded function.

Parameters

- **number_hist** [*integer*] :: The number of distributions to calculate, similar to `user/nq`
- **number_bins** [*integer*] :: The number of bins in the distributions to calculate, similar to `user/nbins`
- **lower_bounds** (*number_hist*) [*real(kind=prec),target*] :: The lower bounds of the distributions, similar to `user/min_val`
- **upper_bounds** (*number_hist*) [*real(kind=prec),target*] :: The upper bounds of the distributions, similar to `user/max_val`
- **measurement_function** [*quantfunc*] :: a pointer to the measurement function like `user/quant()`
- **user_integration** [*usereventfunc,optional*] :: a pointer to the user event function like `user/userevent()`
- **user_initialisation** [*inituserfunc,optional*] :: a pointer to the user initialisation
- **user_integration_dimension** [*integer,optional*] :: the number of integrations the user wishes to carry out to account eg. for beam effects, similar to `user/userdim`
- **names_length** [*integer*] :: the maximally allowed length of the histogram `mcmule/names`, similar to `user/namesLen`
- **filenamesuffix_length** [*integer,optional*] :: the maximally allowed length of the observable name as specified in `mcmule/filenamesuffix`, similar to `user/filenamesuffixLen`

6.1.6 Technical parameters

Warning: The following parameters are exposed for the user to view for finer control over the *measurement function*. Extreme care must be taken when using these.

nel [*integer*]

Set to 1 if electron *VP* loops are to be included, set to 0 otherwise. More options may be added later. Exposed as `mcmule_Nel` in C.

nmu [*integer*]

Set to 1 if muon *VP* loops are to be included, set to 0 otherwise. More options may be added later. Exposed as `mcmule_Nmu` in C.

ntau [*integer*]

Set to 1 if tau *VP* loops are to be included, set to 0 otherwise. More options may be added later. Exposed as `mcmule_Ntau` in C.

nhad [*integer*]

Set to 1 if *HVP* loops are to be included, set to 0 otherwise. More options may be added later. Exposed as *mcmule_Nhad* in C.

which_piece [*character(len=25),protected*]

The piece being integrated, cf. Section *Available processes and which_piece* and *which_piece*. This variable is read-only and is set by McMule.

flavour [*character(len=15),protected*]

The flavour configuration being used. This can only be changed using *initflavour()*

The following parameters should not be modified by the user unless especially advised to do so

softcut [*real(kind=prec),advanced*]

The value of ξ below which the integrand is set to zero without subtraction

collcut [*real(kind=prec),advanced*]

The value of $\cos \theta$ below which the integrand is set to zero

sSwitch [*real(kind=prec),advanced*]

The value of ξ below which the matrix element is approximated at *LP*. This is only available for some matrix elements

ntsSwitch [*real(kind=prec),advanced*]

The value of ξ below which the matrix element is approximated at *NLP*. This is only available for some matrix elements

pcSwitch [*real(kind=prec),advanced*]

The value of $1 - y$ below which the matrix element is approximated. This is only available for some matrix elements

xicut1 [*real(kind=prec),protected*]

The ξ_c value below which to subtract the first photon

xicut2 [*real(kind=prec),protected*]

The ξ_c value below which to subtract the second photon

xieik1 [*real(kind=prec),protected*]

The argument to the first integrated eikonal

xicut2 [*real(kind=prec),protected*]

The argument to the second integrated eikonal

zero [*real(kind=prec),fixed*]

Real numbers below this value may be truncated

pi [*real(kind=prec),fixed*]**xiout** [*protected*]

The current values of ξ for single-real corrections.

xioutA, xioutB

The current values of ξ_i for double-real corrections.

yout [*protected*]

The current values of y for single-real corrections.

youtA, youtB

The current values of y_i for double-real corrections.

qsoft [*protected*]

The momentum of the soft photon *without* ξ for single-real corrections.

qsoftA, qsoftB

The momenta of the soft photons *without* ξ_i for double-real corrections.

nr_q [*protected*]

This variable is obtained by reading user.nq. Exposed as *mcmule_nr_q* in C;

function sachs_gel(*q2*) [*real(kind=prec)*]

The electric Sachs form factor of the proton. In the dipole approximation this is

$$G_e(Q^2) = \frac{1}{(1 + Q^2/\Lambda)^2}$$

Parameters

q2 [*real(kind=prec)*] :: the value of Q^2

function sachs_gmag(*q2*) [*real(kind=prec)*]

The magnetic Sachs form factor of the proton. In the dipole approximation this is

$$G_m(Q^2) = \frac{\kappa}{(1 + Q^2/\Lambda)^2}$$

Parameters

q2 [*real(kind=prec)*] :: the value of Q^2

6.1.7 Symbols exposed to C/C++

McMule is inherently a Fortran code. However, it can be used with a user file written in C/C++ even though many of the quality-of-life functions defined in Sections *Scalar quantities* and *Transformations* are not available.

type real

The real number type used in McMule

bool *mcmule_pass_cut

Preprocessor alias for *pass_cut*.

char *mcmule_names

Preprocessor alias for *names*. It is recommended to use the *MCMULE_SET_NAME* macro to interact with this variable.

real mcmule_musq

Preprocessor alias for *musq*.

real *mcmule_pol1

Preprocessor alias, points to the first element of *pol1*.

real *mcmule_pol2

Preprocessor alias, points to the first element of *pol2*.

char *mcmule_filenamesuffix

Preprocessor alias, points to the first element of *filenamesuffix*.

real ***mcmule_userweight**
Preprocessor alias, points to the user weight

real **mcmule_Gf**
Preprocessor alias for *GF*.

real **mcmule_alpha**
Preprocessor alias for *alpha*.

real **mcmule_sw2**
Preprocessor alias for *sw2*.

real **mcmule_Mm**
Preprocessor alias for *Mm*.

real **mcmule_Me**
Preprocessor alias for *Me*.

real **mcmule_Mtau**
Preprocessor alias for *Mt*.

real **mcmule_scms**
Preprocessor alias for *scms*.

real **mcmule_Mj**
Preprocessor alias for *Mj*.

real **mcmule_CLj**
Preprocessor alias for *CLj*.

real **mcmule_CRj**
Preprocessor alias for *CRj*.

proton_ff **mcmule_protonff**
Preprocessor alias for *the_proton_ff*.

real **mcmule_protonff_lambda**
Preprocessor alias for *lambda*.

real **mcmule_protonff_kappa**
Preprocessor alias for *kappa*.

int **mcmule_Nel**
Preprocessor alias for *Nel*.

int **mcmule_Nmu**
Preprocessor alias for *Nmu*.

int **mcmule_Ntau**
Preprocessor alias for *Ntau*.

int **mcmule_Nhad**
Preprocessor alias for *Nhad*.

int **mcmule_nr_q**
Preprocessor alias for *nr_q*.

MCMULE_SET_NAME(*q*, *name*)

Sets the *q*-th observable to the name *name*

void **mcmule_initflavour**(char *flavour, double *scms)

Interface fo f:subr:*initflavour*.

Parameters

- **flavour** (char*) – the flavour combination to use
- **scms** (double*) – a pointer to the centre-of-mass energy. Passing a null pointer will is only allowed for decay processes or when using a flavour preset.

void **mcmule_runmcmule**(int ncall_ad, int itmx_ad, int ncall, int itmx, int initial_ran_seed, *real* xicut1, *real* xicut2, char *piece, char *flav, char *filename);

Interface for *runmcmule()*.

Parameters

- **tfilename** (char*) – set to NULL to use default

void **mcmule_set_observable**(int number_hist, int number_bins, *real* *lower_bounds, *real* *upper_bounds, quantfunc measurement_function, useeventfunc user_integration, inituserfunc user_initialisation, int user_integration_dimension, int names_length, int filenamesuffix_length)

Interface for *set_observable()*.

6.2 The user file in Fortran

A Fortran user file must be in the form of a module `user` that defines the following symbols.

Warning: None of the values may be parameters in Fortran as they are not exposed to libmcmule

namesLen [*integer, default = 6*]

The maximally allowed length of the histogram *mcmule/names*.

filenamesuffixLen [*integer, default = 10*]

The maximally allowed length of the observable name as specified in *mcmule/filenamesuffix*.

nq [*integer, default=1*]

The number of distributions the user intends to calculate.

nbins [*integer, default=100*]

The number of bins in the distributions the user intends to calculate

userdim [*integer, default=0*]

The number of integrations the user wishes to carry out to account eg. for beam effects

bin_kind [*integer, default=0*]

The binning mechanism being used, 0 for $d\sigma/dQ$ and 1 for $Qd\sigma/dQ$.

Warning: Note that the latter is not properly tested and should only be used with great care

min_val (*nr_q*) [*real(kind=prec)*]

The lower bounds of the distributions

max_val (*nr_q*) [*real(kind=prec)*]

The upper bounds of the distributions

subroutine **inituser**()

If defined, this is called without arguments once as soon as McMule starts and has read all other configuration, meaning that it can access *mcmule/which_piece*. It needs to at least set *mcmule/flavour* through *mcmule/initflavour()*. It may be used to read any further information (like cut configuration etc). The user does not have to print hashes – this is already taken care of – but is very much invited to include information of what it is they are doing.

If the user is using the cut channel of the menu, they may need to set the *mcmule/filenameprefix* variable which is appended to the name of the VEGAS file.

Example for reading a cut and setting the flavour:

```
SUBROUTINE INITUSER
integer cut
call initflavour("mu-e", 1000.)
read*, cut
write(filenameprefix, '(I2)') cut
END SUBROUTINE INITUSER
```

with a global variable *cut*

subroutine **userevent**(*x*, *ndim*)

If defined, the user may use this routine in combination with *mcmule/userweight* to integrate over further parameters, i.e. to calculate

$$\sigma \sim \int_0^1 dx_1 \int_0^1 dx_2 \cdots \int_0^1 dx_m \times \int d\Phi |\mathcal{M}_n|^2 f(x_1, x_2, \dots, x_n; p_1, \dots, p_n)$$

with a generalised *measurement function* *f*. A minimal example that does not include extra integration is

```
SUBROUTINE USEREVENT(X, NDIM)
integer :: ndim
real(kind=prec) :: x(ndim)
userweight = 1.
END SUBROUTINE USEREVENT
```

Parameters

- **x** (*ndim*) [*real(kind=prec)*] :: the values of the integration
- **ndim** [*integer*] :: the dimension of **x**, should equal *userdim*.

function **quant**(*q1*, *q2*, *q3*, *q4*, *q5*, *q6*, *q7*)

The *measurement function* the user wishes to calculate. This needs to at least set *mcmule/pass_cut* but also returns the values of the observables that are to be computed. It also fixes the renormalisation scale *musq* though this could be done elsewhere. If the user wishes to consider polarised scattering, *mcmule/pol1* and *mcmule/pol2* need to be set.

A minimal example that accepts every event and does not calculate a distribution would be

```

FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4),q5(4),q6(4),q7(4)
real (kind=prec) :: quant(nr_q)
  !! ==== keep the line below in any case ==== !!
  musq = me**2
  pol1 = 0.
  pass_cut = .true.
END FUNCTION QUANT

```

Parameters

qi (4) [*real(kind=prec)*] :: the momenta

Return

quant (*nr_q*) [*real(kind=prec)*] :: the observables that are to be histogrammed

6.3 The user file in C/C++

A C/C++ user file must define the following symbols. The defaults are identical to Fortran.

int mcmule_namelen

The maximally allowed length of the histogram *mcmule_names*.

int mcmule_filename_suffix_length

The maximally allowed length of the observable name as specified in *mcmule_filenamesuffix*.

int mcmule_number_hist

The number of distributions the user intends to calculate.

int mcmule_number_bins

The number of bins in the distributions the user intends to calculate

int mcmule_user_integration_dimension

The number of integrations the user wishes to carry out to account eg. for beam effects

int mcmule_bin_kind

The binning mechanism being used, 0 for $d\sigma/dQ$ and 1 for $Qd\sigma/dQ$.

Warning: Note that the latter is not properly tested and should only be used with great care

real **mcmule_lower_bounds**[nr_q]

The lower bounds of the distributions

real **mcmule_upper_bounds**[nr_q]

The upper bounds of the distributions

void mcmule_user_initialisation()

If defined, this is called without arguments once as soon as McMule starts and has read all other configuration, meaning that it can access *mcmule/which_piece* and *mcmule/flavour*. It may be used to read any further information (like cut configuration etc). The user does not have to print hashes – this is already taken care of – but is very much invited to include information of what it is they are doing.

If the user is using the cut channel of the menu, they may need to set the *mcmule_filenamesuffix* variable which is appended to the name of the VEGAS file.

void `mcmule_user_integration`(*real* *x, int ndim)

If defined, the user may use this routine in combination with `mcmule_userweight` to integrate over further parameters.

Parameters

- **x** (*real**) – a pointer to the first element of the values of the integration
- **ndim** (int) – the dimension of x, should equal `mcmule_user_integration_dimension`.

void `mcmule_measurement_function`(double **res, *real* *q1, *real* *q2, *real* *q3, *real* *q4, *real* *q5, *real* *q6, *real* *q7)

The *measurement function* the user wishes to calculate. This needs to at least set `mcmule_pass_cut` but also returns the values of the observables that are to be computed. It also fixes the renormalisation scale `mcmule_musq` though this could be done elsewhere. If the user wishes to consider polarised scattering, `mcmule_pol1` and `mcmule_pol2` need to be set. This may use the macro `MCMULE_SET_NAME`

Parameters

- **res** (*real****) – a double pointer to the result.
- **qi** (*real**) – pointers to the first component of the momenta

```
void mcmule_measurement_function(double**res, double*p1, double*p2, double*p3,
↪double*p4, double*p5, double*p6, double*p7) {
    mcmule_musq = 1.;

    res[0][0] = p3[2] / p3[3]; // Set the second quant to p3z/E3
    res[0][1] = p4[2] / p4[3]; // Set the second quant to p4z/E4

    mcmule_pass_cut[0] = 1; // Accept event
    mcmule_pass_cut[1] = 1; // Accept event

    MCMULE_SET_NAME(0, "p3z/E3")
    MCMULE_SET_NAME(1, "p4z/E4")
}
```


Chapter 7

pymule user guide

This section describes all public functions and classes in pymule.

7.1 Working with files

class `pymule.Chi2`

An object to represent a χ^2 value.

This can be created using `Chi2.from_single()` or `Chi2.from_merge()`. This class supports addition and float.

Attribute type

an enum to indicate the type of χ^2 this is

Attribute value

the value of χ^2 for single and mergers

Attribute children

the children `Chi2` objects for sums and mergers

class `Type`(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The χ^2 type enum

Attribute NONE

empty entry

Attribute SINGLE

a single entry

Attribute SUM

a sum of entries

Attribute MERGE

a statistical merge of entries

classmethod `from_merge`(*chi: float*, **s: Chi2*) → *Chi2*

combines multiple χ^2 into a single one

Parameters

- **chi** – the value of χ^2 of the merge, must be positive
- **s** – a sequence of `Chi2` objects

Returns

a new Chi2 object

Return type

Chi2

Raises

- **ValueError** – the parameter `chi` is not positive or no Chi2 is provided in `s`
- **TypeError** – the parameter `chi` is not a number or the objects provided are not Chi2

Example

Provide a merged set

```
>>> c1 = Chi2.from_single(1.02)
>>> c2 = Chi2.from_single(1.41)
>>> c3 = Chi2.from_single(2.02)
>>> tot = Chi2.from_merge(1.3, c1, c2, c3)
```

classmethod `from_single(v: float) → Chi2`

creates a single Chi2 object

Parameters

`v` – the value of χ^2 , must be positive

Returns

a new Chi2 object

Return type

Chi2

Raises

- **ValueError** – the parameter `v` is not positive
- **TypeError** – the parameter `v` is not a number

Example

Create a $\chi^2 = 1.02$

```
>>> Chi2.from_single(1.02)
<Chi^2: single 1.02>
```

class `pymule.FileRecord`

A subclass of `Record` to track McMule output files.

Additionally to the attributes of `Record` this contains all relevant properties of a McMule run

Attribute version

the file version

Attribute inttype

the integer type used in McMule in the format for `struct`

Attribute sha

the sha hash provided by the file

Attribute sampler

the sampler used, currently either `vegas` or `foam`

Attribute iteration

the current iteration of McMule

Attribute randy

the current random seed

Attribute msg

the integration message

ExampleLoad a record stored at `/path/to/record.mcmule` and print its value

```
>>> rec = FileRecord.read("/path/to/record.mcmule")
>>> print(rec.value)
[1.00646123e-04 1.06589201e-08]
```

classmethod `read(filename: str = "", fp: IO[bytes] | None = None) → FileRecord`

opens a McMule file, infers the correct file version, and creates a subclass of `FileRecord`**Parameters**

- **filename** – file name to open, optional
- **fp** – file pointer to read, optional

Returns

a recording corresponding to the file

Return typesubclass of `FileRecord`

Note: Either **filename** xor **fp** need to be specified

Note: If less than two iterations have been completed, no histograms will be returned

Example

Load a file for the muon decay

```
>>> FileRecord.read('m2ennRR_mu-e_S0000068031X0.50000D0.50000_
↳ITMX080x150M_012.vegas')
<Record m2ennRR_mu-e_S0000068031X0.50000D0.50000_ITMX080x150M_012.
↳vegas xsec = 0.00010065(1), 12 histograms>
```

`pymule.mergefks(*sets: XiRecord, **kwargs: XiRecord) → Record`

performs the FKS merge

Parameters**sets** – random-seed-merged results of type `XiRecord` (usually from `sigma()`)**Returns**

the FKS-merged final set containing cross sections, distributions, and run-time information.

Return type`Record`

Note: Optional argument `anyxi` (or anything starting with `anyxi`): Sometimes it is necessary to merge ξ_c -dependent runs (such as a counter term) and ξ_c -independent runs (such as the one-loop term).

Example

Load the LO results for the muon decay using `sigma()`

```
>>> mergefks(sigma("m2enn0"))
>>> <Record m2enn0 xsec = 2211500(2), 1 histograms>
```

Load the NLO results

```
>>> mergefks(sigma("m2ennV"), sigma("m2ennR"))
<Record xsec = -4143205.8581450013, 1 histograms>
```

Load the NNLO results where `m2ennNF` does not depend on ξ_c

```
>>> mergefks(
...     sigma("m2ennFF"), sigma("m2ennRF"), sigma("m2ennRR"),
...     anyxi=sigma("m2ennNF")
... )
<Record xsec = -131.13(9), 1 histograms>
```

`pymule.setup(folder: str | None = None, **kwargs: Any) → None`

sets the default arguments for `sigma()`.

Parameters

- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files Initialised to current directory (`.`).
- **flavour** – str, optional; the flavour to load, defaults to everything Initialised to everything, i.e. `.*`.
- **obs** – str, optional; the observable to load (the bit after the 0), defaults to everything Initialised to everything, i.e. `''`.
- **folderp** – str, optional; a regular expression to match directory structures of a tar file, defaults to everything Initialised to everything, i.e. `.*`.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram name. Initialised to no merging, i.e. `{}`
- **sanitycheck** – callable, optional; a function that, given a `FileRecord`, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.

Example

Setup some folders, ensure that `/tmp/mcmule` exists

```
>>> setup(folder="path/to/data.tar.bz2", cachefolder="/tmp/mcmule")
```

Example

Restrict observable

```
>>> setup(obs="3")
```

Example

Drop runs with a $\chi^2 > 10$

```
>>> setup(sanitycheck=lambda x : x['chi2a'] < 10)
```

`pymule.sigma(piece: str, **kwargs: Any) → XiRecord`

loads a `which_piece` and returns a `XiRecord`

Parameters

- **piece** – str; `which_piece` to load
- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files Initialised to current directory (.).
- **flavour** – str, optional; the flavour to load, defaults to everything Initialised to everything, i.e. `.*`.
- **obs** – str, optional; the observable to load (the bit after the O), defaults to everything. Initialised to everything, i.e. `' '`.
- **folderp** – str, optional; a regular expression to match directory structures of a tar file, defaults to everything. Initialised to everything, i.e. `.*`.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram name. Initialised to to no merging, i.e. `{}`
- **sanitycheck** – callable, optional; a function that, given a `FileRecord`, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.

Returns

a dict with the tuples of FKS parameters as keys and vegas datasets as values.

Note: Use `setup()` to set the defaults. Arguments provided here override the defaults

Example

Load the leading order muon decay

```
>>> sigma("m2enn0")
>>> <XiRecord set m2enn0, xic = [1.0], 1 histograms>
```

Load only observable 03

```
>>> sigma("m2enn0", obs="3")
>>> <XiRecord set m2enn0, xic = [1.0], 1 histograms>
```

7.2 Working with errors

`pymule.addplots(a: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, b: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, sa: float = 1.0, sb: float = 1.0) → <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`

adds or subtracts two plots

Parameters

- **a** – Nx3 numpy matrix; the first plot
- **b** – Nx3 numpy matrix; the second plot

- **sa** – float, optional; the coefficient of the first plot
- **sb** – float, optional; the coefficient of the second plot

Returns

a Nx3 numpy matrix with $s_a \cdot a + s_b \cdot b$

Note: a and b must share x values, otherwise entries are dropped

Example

subtract two plots a and b

```
>>> addplots(a, b, sb=-1)
```

Example

Given the LO plots thetaLO and the NLO corrections thetadNLO, we calculate the *K* factor as either

```
>>> thetaNLO = addplots(thetaLO, thetadNLO)
```

```
pymule.chisq(values: ~typing.List[~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → float
```

calculates the χ^2 /d.o.f. of numbers

Parameters

value – Nx2 numpy matrix or list of lists; the values as [[y1, dy1], [y2, dy2], ...]

Returns

float; the χ^2 /d.o.f. = $\frac{1}{n} \sum_{n=1}^n (\frac{y_i - \bar{y}}{\delta y_i})^2$ with the average value \bar{y}

Example

a good example

```
>>> chisq([[20.0, 0.8],
...       [21.6, 0.9],
...       [18.7, 1.2]])
1.3348808062205872
```

and a bad example

```
>>> chisq([[16.2, 0.8],
...       [22.9, 0.9],
...       [8.81, 1.2]])
30.173852184366673
```

```
pymule.dividenumbers(a: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, b: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → __getitem__()' id='140279735182480'>
```

divides numbers

Parameters

- **a** – list of floats; the numerator with error [a, da]

- **b** – list of floats; the denominator with error [**b**, **db**]

Returns

the result of the division a/b [**y**, **dy**]

Example

Divide $(2.3 \pm 0.1)/(45 \pm 0.01)$

```
>>> dividenumbers([2.3, 0.1], [45., 0.01])
array([0.05111111, 0.00222225])
```

```
pymule.divideplots(a: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, b:
<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, offset:
float = 0.0) → <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>
```

divides two plots

Parameters

- **a** – $N \times 3$ numpy matrix; the numerator plot
- **b** – $N \times 3$ numpy matrix; the denominator plot
- **offset** – float, optional; shifts the result

Returns

a $N \times 3$ numpy matrix with $a/b + offset$

Note: **a** and **b** must share **x** values, otherwise entries are dropped

Example

Given the LO plots **thetaLO** and the NLO corrections **thetadNLO**, we calculate the *K* factor as either

```
>>> thetaNLO = addplots(thetaLO, thetadNLO)
>>> thetaK = divideplots(thetaNLO, thetaLO)
>>> thetaK = divideplots(thetadNLO, thetaLO, offset=+1.)
```

```
pymule.integratehistogram(hist: <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>) → float
```

integrates a histogram

Parameters

hist – $N \times 3$ numpy matrix; the histogram to integrate $d\sigma/dx$ as `np.array([[x1, y1, e1], [x2, y2, e2], ...])`

Returns

float; the integrated histogram $\int d\sigma/dx dx$ without error estimate

Example

Integrate a histogram

```
>>> hist
array([[
    -inf, 0.00000000e+00, 0.00000000e+00],
    [5.00000000e-02, 4.77330751e+01, 2.26798977e-01],
    [1.50000000e-01, 7.40641192e+01, 2.36498021e-01],
    ...,
    ...])
```

(continues on next page)

(continued from previous page)

```

[8.85000000e+00, 1.67513948e+00, 1.16218116e-01],
[8.95000000e+00, 0.00000000e+00, 0.00000000e+00],
[
    inf, 0.00000000e+00, 0.00000000e+00]]
>>> integratehistogram(hist)
4188.519369660588

```

`pymule.mergebins(p: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, n: int) → <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`

merges n adjacent bins into one larger bin, reducing the uncertainty.

Parameters

- **p** – $N \times 3$ numpy matrix; the plot
- **n** – int; how many bins to merge

Returns

a $(N/n) \times 3$ numpy matrix

Note: This process loses $\text{len}(p)\%n$ bins at the end of the histogram

Example

merge five bins

```

>>> len(p)
200
>>> len(mergebins(p, 5))
40

```

Bins may be lost

```

>>> len(p)
203
>>> len(mergebins(p, 5))
40

```

`pymule.mergenumbers(values: ~typing.List[~typing.List[float]] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → __getitem__()' id='140279735182480'>`

statistically combines values with uncertainties

Parameters

values – $N \times 2$ numpy matrix or list of lists; the values as $[[y_1, dy_1], [y_2, dy_2], \dots]$

Returns

either answer as numpy array $[y, dy]$ or tuple of χ^2 and answer

Example

If quiet is not specified this will print the χ^2

```

>>> mergenumbers([[20.0, 0.8],
...               [21.6, 0.9],
...               [18.7, 1.2]])

```

(continues on next page)

(continued from previous page)

```
1.3348808062205872
array([20.30718232,  0.53517179])
```

Otherwise, it will return it

```
>>> mergenumbers([[20.0, 0.8],
...               [21.6, 0.9],
...               [18.7, 1.2]], quiet=True)
(1.3348808062205872, array([20.30718232,  0.53517179]))
```

```
pymule.mergeplots(ps: ~typing.List[<MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>] | <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>) → <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>
```

statistically combines a list of plots

Parameters

ps – list of Nx3 numpy matrices; the plots to combine as `[np.array([[x1, y1, e1], [x2, y2, e2], ...]), ...]`

Returns

a Nx3 numpy matrix

Example

Load a number of vegas files and merge them

```
>>> data = [
...     importvegas(i)['thetae']
...     for i in glob.glob('out/em2em0*')
... ]
>>> mergeplots(data)
```

```
pymule.plusnumbers(*args: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>) → __getitem__() id='140279735182480'>
```

adds numbers and errors

Parameters

yi – list of floats; a number with error `[yi, dyi]`

Returns

the result of the addition `[y, dy]`

Example

Adding $(10 \pm 1) + (20 \pm 0.5) + (-5 \pm 2)$

```
>>> plusnumbers([10, 1], [20, 0.5], [-5, 2])
array([25.          ,  2.29128785])
```

```
pymule.printnumber(x: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>, prec: int = 0) → str
```

returns a string representation of a number with uncertainties to one significant digit

Parameters

- **x** – a list with two floats; the number as `[x, dx]`
- **prec** – int, optional; number of extra significant figures

Returns

str; the formatted string

Example

printing 53.2 ± 0.1 to one significant figure

```
>>> printnumber([53.2, 0.1])
"53.2(1)"
```

```
pymule.scaleplot(a: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, sx:
float, sy: float | None = None) → <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>
```

rescales a plot such that the integrated plot remains unchanged, i.e. rescale $x \rightarrow x/s$ and $y \rightarrow y \cdot s$. This is useful to, for example, change units.

Parameters

- **a** – Nx3 numpy matrix; the plot
- **sx** – float; the inverse scale factor for the x direction
- **sy** – float, optional; if present, sy will be used for the y direction instead of sx

Returns

a Nx3 numpy matrix

Example

rescaling units from rad to mrad

```
>>> scaleplot(data, 1e-3)
```

```
pymule.timesnumbers(a: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>, b: ~typing.List[float] | <MagicMock
name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → __getitem__()'
id='140279735182480'>
```

multiplies numbers

Parameters

- **a** – list of floats; the first factor with error [a, da]
- **b** – list of floats; the second factor with error [b, db]

Returns

the result of the multiplication a*b [y, dy]

Example

Divide $(0.5 \pm 0.02) * (45 \pm 0.01)$

```
>>> timesnumbers([0.5,0.02], [45, 0.1])
array([22.5      ,  0.90138782])
```

7.3 Plotting

`pymule.errorband(p, ax=None, col='default', underflow=False, overflow=False, linestyle='solid')`

plots an errorband of a compatible histogram

Parameters

- **p** – Nx3 numpy matrix; the histogram to plot as `np.array([[x1, y1, e1], [x2, y2, e2], ...])`
- **ax** – axes, optional: the axes object to use, defaults to `gca()` which may create a new axes.
- **col** – the colour to be used for the plot. Per default matplotlib decides using the order specified in `colours`
- **underflow** – bool, optional; whether to plot the underflow bin. Either logical or number indicating the how much bigger it shall be
- **overflow** – bool, optional; whether to plot the overflow bin. Either logical or number indicating the how much bigger it shall be
- **linestyle** – str, optional; which line style to use

Returns

the artis of the main line but not the one of the errorbars

Example

Make a simple plot

```
>>> errorband(dat)
```

Make a plot in red with dashed lines

```
>>> errorband(dat, 'red', 'dashed')
```

`pymule.kplot(sigma, labelx='x_e', labelsigma=None, labelknlo='$\delta K^{(1)}$', labelknnlo='$\delta K^{(2)}$', legend={'lo': '$\rm LO$', 'nlo': '$\rm NLO$', 'nnlo': '$\rm NNLO$'}, legendopts={'loc': 'upper right', 'what': 'l'}, linestyle2=':', show=[0, -1], showk=[1, 2], nomule=False)`

produces a K factor plot in line with McMule's design, i.e. a two-panel plot showing in the upper panel the cross sections and in the lower panel the K factor defined as

$$K^{(i)} = d\sigma^{(i)} / d\sigma^{(i-1)}$$

Parameters

- **sigma** – dict; the data to plot, given as a dict with keys `lo`, `nlo`, and possibly `nnlo`. Only pass the corrections, not the full distribution
- **labelx** – str, optional; label for the x axis (supports LaTeX maths)
- **labelsigma** – str, optional; label for the upper y axis (supports LaTeX maths)
- **labelknlo** – str, optional; the labels for the NLO K factor
- **labelknnlo** – str, optional; the labels for the NNLO K factor
- **show** – list, optional; a list which cross sections to show, 0 indicates the LO cross section, 1 the NLO etc. -1 indicates the last given cross section
- **showk** – list, optional; a list which K factors to show, 0 indicates the LO cross section, 1 the NLO etc. -1 indicates the last given cross section

- **legend** – dict, optional; a dict with the legend for `lo`, `nlo`, `nnlo`. The keys `nlo2` and `nnlo2` are optional and will be drawn dashed in the lower panel.
- **legendopts** – dict, optional; a kwargs dict of options to be passed to `legend(...)` as well as the `what` key indicating whether the legend such be placed in the lower panel (`l`, default), upper panel (`u`), or as a `figlegend` (`fig`). Notable is the `loc`-key that places the legend inside the object specified by `what`. Possible values are (cf. `legend`)
 - upper right
 - upper left
 - lower left
 - lower right
 - right
 - center left
 - center right
 - lower center
 - upper center
 - center
- **nomule** – bool, optional; if set to `True`, no mule will be printed

Returns

the figure as well as all axis created

Example

An NNLO K factor plot

```
>>> fig, (ax1, ax2, ax3) = kplot(
...     {
...     'lo': lodata['thetae'],
...     'nlo': nloata['thetae'],
...     'nnlo': nnloata['thetae'],
...     },
...     labelx="$\theta_e, \wedge, \{\rm mrad}\$",
...     labelsigma="$\D\sigma/\D\theta_e \wedge \{\rm \upmu b}\$",
...     legend={
...     'lo': '$\sigma^{(0)}$',
...     'nlo': '$\sigma^{(1)}$',
...     'nnlo': '$\sigma^{(2)}$'
...     },
...     legendopts={'what': 'u', 'loc': 'lower right'}
... )
```

`pymule.mergefkswithplot(*psets: XiRecord, showfit: tuple[bool, bool] = (True, True), xlim: tuple[float, float] = (-7.0, 0.0)) → tuple[Any, Record]`

performs and FKS merge like `mergefks()` but it also produces a ξ_c independence plot.

Note: In contrast `mergefks()`, here phase-space partitioned results need to be summed first.

Parameters

- **sets** – a sequence of *XiRecord* (or sums thereof, usually from *sigma()*), starting with the lowest particle number and going up
- **showfit** – [bool, bool], optional; whether to show the fit lines in the overview plot (first element) and the zoomed in plot (second element)
- **xlim** – tuple of floats, optional; upper and lower bounds for $\log \xi_c$

Returns

a figure and the FKS-merged *Record* containing cross sections, distributions, and run-time information.

Example

In the partitioned muon-electron scattering case

```
>>> fig, res = mergefkswithplot(
...     alpha*(sigma('em2emFEE')+sigma('em2emFMM')+sigma('em2emFEM')),
...     alpha*(
...         sigma('em2emREE15') + sigma('em2emREE35') +
...         sigma('em2emRMM') + sigma('em2emREM')
...     )
... )
```

`pymule.mulify(fig, delx=0, dely=0, col='lightgray', realpha=True)`

adds the McMule logo to a figure

Parameters

- **fig** – figure to add the logo
- **delx** – float, optional; shift the logo in x direction
- **dely** – float, optional; shift the logo in x direction
- **col** – colour specifier, optional; colour to use for the logo
- **realpha** – bool, optional; whether to re-run the alpha channel

`pymule.watermark(fig, txt='PRELIMINARY', fontsize=60, rotation=20)`

watermarks a figure

Parameters

- **fig** – the figure to watermark
- **txt** – str, optional; the watermark text to use
- **fontsize** – int, optional; the fontsize of the watermark
- **rotation** – int, optional; the angle of the watermark in deg

Example

Watermark a figure as preliminary

```
>>> fig = figure()
>>> ...
>>> watermark(fig)
```

Watermark a figure as incomplete

```
>>> fig = figure()
>>> ...
>>> watermark(fig, "INCOMPLETE")
```

`pymule.xiresidue`(*pset*: `XiRecord`, *n*: `int`, *xlim*: `tuple[float, float] = (-7.0, 0.0)`) → `tuple[~typing.Any, <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>]`

creates a residue plot for a ξ_c fit

Parameters

- **sets** – a `XiRecord`
- **n** – `int`; order of the fit, 1 at NLO, 2 at NNLO
- **xlim** – `tuple` of floats, optional; upper and lower bounds for $\log \xi_c$

Returns

a figure and the fit coefficients as a matrix

Chapter 8

Technical aspects of McMule

In this section, we will review the very technical details of the implementation. This is meant for those readers, who wish to truly understand the nuts and bolts holding the code together. We begin by discussing the phase-space generation and potential pitfalls in Section *Phase-space generation*. Next, in Section *Implementation of FKS schemes*, we discuss how the *FKS* scheme [8, 23, 25, 28, 29]. This is meant for those readers, who wish to truly understand the nuts and bolts holding the code together. We begin by discussing the phase-space generation and potential pitfalls in Section *Phase-space generation*. Next, in Section *Implementation of FKS schemes*, we discuss how the *FKS* scheme [8, 23, 25, 28, 29] (cf. Appendix *The FKS² scheme* for a review) is implemented in Fortran code. This is followed by a brief review of the random number generator used in McMule in Section *Random number generation*. Finally, we give an account of how the statefiles work and how they are used to store distributions in Section *Differential distributions and intermediary state files*.

8.1 Particle string framework

Note: This section needs to be completed, link to [issue](#)

8.2 Phase-space generation

We use the vegas algorithm for numerical integration [15]. As vegas only works on the hypercube, we need a routine that maps $[0, 1]^{3n-4}$ to the momenta of an n -particle final state, including the corresponding Jacobian. The simplest way to do this uses iterative two-particle phase-spaces and boosting the generated momenta all back into the frame under consideration. An example of how this is done is shown in [Listing 8.1](#).

Listing 8.1: Example implementation of iterative phase-space. Not shown are the checks to make sure that all particles have at least enough energy for their mass, i.e. that $E_i \geq m_i$.

```
! use a random number to decide how much energy should
! go into the first particle
minv3 = ra(1)*energy

! use two random numbers to generate the momenta of
! particles 1 and the remainder in the CMS frame
```

(continues on next page)

(continued from previous page)

```

call pair_dec(ra(2:3),energy,q2,m2,qq3,minv3)

! adjust the Jacobian
weight = minv3*energy/pi
weight = weight*0.125*sq_lambda(energy**2,m2,minv3)/energy**2/pi

! use a random number to decide how much energy should
! go into the second particle
minv4 = ra(4)*energy
! use two random numbers to generate the momenta of
! particles 2 and the remainder in their rest frame
call pair_dec(ra(5:6),minv3,q3,m3,qq4,minv4)

! adjust the Jacobian
weight = weight*minv4*energy/pi
weight = weight*0.125*sq_lambda(minv3**2,m3,minv4)/minv3**2/pi

! repeat this process until all particles are generated

! boost all generated particles back into the CMS frame

q4 = boost_back(qq4, q4)
q5 = boost_back(qq4, q5)

q3 = boost_back(qq3, q3)
q4 = boost_back(qq3, q4)
q5 = boost_back(qq3, q5)

```

As soon as we start using *FKS*, we cannot use this simplistic approach any longer. The *c*-distributions of *FKS* require the photon energies ξ_i to be variables of the integration. We can fix this by first generating the photon explicitly as

$$k_1 = p_{n+1} = \frac{\sqrt{s}}{2} \xi_1 (1, \sqrt{1 - y_1^2} \vec{e}_\perp, y_1) \quad (8.1)$$

where \vec{e}_\perp is a $(d - 2)$ dimensional unit vector and the ranges of y_1 (the cosine of the angle) and ξ_1 (the scaled energy) are $-1 \leq y_1 \leq 1$ and $0 \leq \xi_1 \leq \xi_{\max}$, respectively. The upper bound ξ_{\max} depends on the masses of the outgoing particles. Following [28] we find

$$\xi_{\max} = 1 - \frac{\left(\sum_i m_i\right)^2}{s}$$

Finally, the remaining particles are generated iteratively again. This can always be done and is guaranteed to work.

For processes with one or more *PCS*s this approach is suboptimal. The numerical integration can be improved by orders of magnitude by aligning the pseudo-singular contribution to one of the variables of the integration, as this allows *vegas* to optimise the integration procedure accordingly. As an example, consider once again $\mu \rightarrow \nu \bar{\nu} e \gamma$. The *PCS* comes from

$$\mathcal{M}_{n+1}^{(\ell)} \propto \frac{1}{q \cdot k} = \frac{1}{\xi^2} \frac{1}{1 - y\beta}$$

where y is the angle between photon (k) and electron (q). For large velocities β (or equivalently small masses), this becomes almost singular as $y \rightarrow 1$. If now y is a variable of the integration this can be mediated. An example implementation is shown in Listing 8.2.

Listing 8.2: Example implementation of a so-called *FKS* phase-space where the fifth particle is an *FKS* photon that may become soft. Not shown are checks whether $E_i \geq m_i$.

```

xi5 = ra(1)
y2 = 2*ra(2) - 1.

! generate electron q2 and photon q5 s.t. that the
! photon goes into z directions

eme = energy*ra(3)
pme = sqrt(eme**2-m2**2)
q2 = (/ 0., pme*sqrt(1. - y2**2), pme*y2, eme /)
q5 = (/ 0., 0., 1., 1. /)
q5 = 0.5*energy*xi5*q5

! generate euler angles and rotate all momenta

euler_mat = get_euler_mat(ra(4:6))

q2 = matmul(euler_mat,q2)
q5 = matmul(euler_mat,q5)

qq34 = q1-q2-q5
minv34 = sqrt(sq(qq34))

! The event weight, note that a factor xi5**2 has been omitted
weight = energy**3*pme/(4.*(2.*pi)**4)

! generate remaining neutrino momenta

call pair_dec(ra(7:8),minv34,q3,m3,q4,m4,enough_energy)
weight = weight*0.125*sq_lambda(minv34**2,m3,m4)/minv34**2/pi

q3 = boost_back(qq34, q3)
q4 = boost_back(qq34, q4)

```

The approach outlined above is very easy to do in the case of the muon decay as the neutrinos can absorb any timelike four-momentum. This is because the δ function of the phase-space was solved through the neutrino's `pair_dec`. However, for scattering processes where all final state leptons could be measured, this fails. Writing a routine for μ - e -scattering

$$e(p_1) + \mu(p_2) \rightarrow e(p_3) + \mu(p_4) + \gamma(p_5)$$

that optimises on the incoming electron is rather trivial because its direction stays fixed s.t. the photon just needs to be generated according to (8.1). The outgoing electron p_3 is more complicated. Writing the p_4 -phase-space four- instead of three-dimensional

$$d\Phi_5 = \delta^{(4)}(p_1 + p_2 - p_3 - p_4 - p_5) \delta(p_4^2 - M^2) \Theta(E_4) \frac{d^4\vec{p}_4}{(2\pi)^4} \frac{d^3\vec{p}_3}{(2\pi)^3 2E_3} \frac{d^3\vec{p}_5}{(2\pi)^3 2E_5}$$

we can solve the four-dimensional δ function for p_4 and proceed for the generation p_3 and p_5 almost as for the muon decay above. Doing this we obtain for the final δ function

$$\delta(p_4^2 - M^2) = \delta\left(m^2 - M^2 + s(1 - \xi) + E_3\sqrt{s}\left[\xi - 2 - y\xi\beta_3(E_3)\right]\right). \quad (8.2)$$

When solving this for E_3 , we need to take care to avoid extraneous solutions of this radical equation [11]. We have now obtained our phase-space parametrisation, albeit with one caveat: for anti-collinear photons, i.e. $-1 < y < 0$ with energies

$$\xi_1 = 1 - \frac{m}{\sqrt{s}} + \frac{M^2}{\sqrt{s}(m - \sqrt{s})} < \xi < \xi_{\max} = 1 - \frac{(m + M)^2}{s}$$

there are still two solutions. One of these corresponds to very low-energy electron that are almost produced at rest. This is rather fortunate as most experiments will have an electron detection threshold higher than this. Otherwise, phase-spaces optimised this way also define a *which_piece* for this *corner region*.

There is one last subtlety when it comes to these type of phase-space optimisations. Optimising the phase-space for emission from one leg often has adverse effects on terms with dominant emission from another leg. In other words, the numerical integration works best if there is only one *PCS* on which the phase-space is tuned. As most processes have more than one *PCS* we need to resort to something that was already discussed in the original *FKS* paper [29]. Scattering processes that involve multiple massless particles have overlapping singular regions. The *FKS* scheme now mandates that the phase-space is partitioned in such a way as to isolate at most one singularity per region with each region having its own phase-space parametrisation. Similarly we have to split the phase-space to contain at most one *PCS* as well as the soft singularity. In McMule μ - e scattering for instance is split as follows¹

$$1 = \theta(s_{15} > s_{35}) + \theta(s_{15} < s_{35})$$

with $s_{ij} = 2p_i \cdot p_j$ as usual. The integrand of the first θ function has a final-state *PCS* and hence we use the parametrisation obtained by solving (8.2). The second θ function, on the other hand, has an initial-state *PCS* which can be treated by just directly parametrising the photon in the centre-of-mass frame as per (8.1). This automatically makes $s_{15} \propto (1 - \beta_{\text{in}} y_1)$ a variable of the integration.

For the double-real corrections of μ - e scattering, we proceed along the same lines except now the argument of the δ function is more complicated.

8.3 Implementation of FKS schemes

Now that we have a phase-space routine that has ξ_i as variables of the integration, we can start implementing the relevant c -distributions (10.4)

$$d\sigma_h^{(1)}(\xi_c) = d\Upsilon_1 d\Phi_{n,1} \left(\frac{1}{\xi_1} \right)_c d\xi_1 \left(\xi_1^2 \mathcal{M}_{n+1}^{(0)} \right) \quad (8.3)$$

$$= d\xi_1 \left(d\Upsilon_1 d\Phi_{n,1} \left(\xi_1^2 \mathcal{M}_{n+1}^{(0)} \right) - d\Upsilon_1 d\Phi_{n,1} \left(\mathcal{E} \mathcal{M}_n^{(0)} \right) \theta(\xi_c - \xi_1) \right) \quad (8.4)$$

We refer to the first term as the *event* and the second as the *counter-event*.

Note that, due to the presence of $\delta(\xi_1)$ in the counter-event (that is implemented through the eikonal factor \mathcal{E}) the momenta generated by the phase-space $d\Upsilon_1 d\Phi_{n,1}$ are different. Thus, it is possible that the momenta of the event pass the cuts or on-shell conditions, while those of the counter event fail, or vice versa. This subtlety is extremely important to properly implement the *FKS* scheme and many problems fundamentally trace back to this.

Finally, we should note that, in order to increase numerical stability, we introduce cuts on ξ and sometimes also on a parameter that encodes the *PCS* such as $y = y_2$ in (8.1) and Listing 8.2. Events that have values of ξ smaller than this *soft cut* are discarded immediately and no subtraction is considered. The dependence on this slicing parameter is not expected to drop out completely and hence, the *soft cut* has to be chosen small enough to not influence the result.

An example implementation can be found in Listing 8.3.

¹ When implementing this, care must be taken to ensure that the split is also well defined if the photon is soft, i.e. if $\xi = 0$.

Listing 8.3: An example implementation of the *FKS* scheme in Fortran. Not shown are various checks performed, the binning as well as initialisation blocks.

```

FUNCTION SIGMA_1(x, wgt, ndim)

! The first random number x(1) is xi.
arr = x

! Generate momenta for the event using the function pointer ps
call gen_mom_fks(ps, x, masses(1:nparticle), vecs, weight)

! Whether unphysical or not, take the value of xi
xifix = xiout

! Check if the event is physical ...
if(weight > zero) then
  ! and whether it passes the cuts
  var = quant(vecs(:,1), vecs(:,2), vecs(:,3), vecs(:,4), ...)
  cuts = any(pass_cut)
  if(cuts) then
    ! Calculate the  $\xi^{*2} * M_{n+1}^0$  using the pointer matel
    mat = matel(vecs(:,1), vecs(:,2), vecs(:,3), vecs(:,4), ...)
    mat = xifix*weight*mat
    sigma_1 = mat
  end if
end if

! Check whether soft subtraction is required
if(xifix < xicut1) then
  ! Implement the delta function and regenerate events
  arr(1) = 0._prec
  call gen_mom_fks(ps, arr, masses(1:nparticle), vecs, weight)
  ! Check whether to include the counter event
  if(weight > zero) then
    var = quant(vecs(:,1), vecs(:,2), vecs(:,3), vecs(:,4), ...)
    cuts = any(pass_cut)
    if(cuts) then
      mat = matel_s(vecs(:,1), vecs(:,2), vecs(:,3), vecs(:,4), ...)
      mat = weight*mat/xifix
      sigma_1 = sigma_1 - mat
    endif
  endif
endif
END FUNCTION SIGMA_1

```

8.4 Calling procedures and function pointers

McMule uses function pointers to keep track of which functions to call for the integrand, phase-space routine, and matrix element(s). These pointers are assigned during `init_piece()` and then called throughout `integrands` and `phase_space`. The pointers for the phase-space generator and integrand are just assigned using the `=>` operator, i.e.

```
ps => psx2 ; fxn => sigma_0
```

The relevant abstract interface for the integrand `fxn` is

```
abstract interface
  function integrand(x,wgt,ndim)
    import prec
    integer :: ndim
    real(kind=prec) :: x(ndim),wgt
    real(kind=prec) :: integrand
  end function integrand
end interface
```

Doing the same for the matrix elements is not possible as they do not have a consistent interface. Instead, we are using a C function `set_func` that is implemented in a separate file to assign the functions, ignoring the interface

```
call set_func('00000000', pm2engav)
call set_func('00000001', pm2ennav)
call set_func('11111111', m2enn_part)
```

The first argument corresponds to the type of functions that is being set.

Table 8.1: Arguments for `set_func`

| Bitmask | Name | Description |
|----------|----------|-------------------------------|
| 00000000 | matel0 | hard matrix element |
| 00000001 | matel1 | reduced matrix element |
| 00000010 | matel2 | doubly reduced matrix element |
| 11111111 | partfunc | particle string function |
| 10000001 | matel_s | single soft limit |
| 10000010 | matel_hs | hard-soft limit |
| 10000100 | matel_sh | soft-hard limit |
| 10000110 | matel_ss | double soft limit |

If the soft limits are not assigned, they are auto-generated using the `partfunc`.

8.5 Optional parameters for integrands

The integration is configured during the `initpiece()` routine. Additionally to identifying what is to be integrated (cf. Section *Calling procedures and function pointers*), one also configures other parameters such as the dimensionality or the masses involved.

Table 8.2: Frozen Delights!

| Variable | Type | Description | Required |
|------------------------|----------------------|---|---|
| <code>nparticle</code> | integer | the number of total particles (initial & final) | yes |
| <code>ndim</code> | integer | the dimensionality of the phase space. usually this is $3n_f - 4$, for calculations with extra integrations, these are included | yes |
| <code>masses</code> | <code>real(:)</code> | the masses of all particles | yes |
| <code>xicut1</code> | real | the value of ξ_c for the first subtraction | for real corrections |
| <code>xicut2</code> | real | the value of ξ_c for the second subtraction | for double-real corrections |
| <code>xieik1</code> | real | the value of ξ_c for the first eikonal | for virtual or real-virtual corrections |
| <code>xieik2</code> | real | the value of ξ_c for the second eikonal | for double-virtual corrections |
| <code>polarised</code> | integer | the number of polarised particles | no, defaults to 0 |
| <code>symfac</code> | real | the symmetry factor for indistinguishable final states | no, defaults to 1 |
| <code>softcut</code> | real | the soft cut parameter | no, but recommended, defaults to 0 |
| <code>collcut</code> | real | the collinear cut parameter | no, but recommended, defaults to 0 |
| <code>ntsSwitch</code> | real | the <i>NTS</i> switching point | only for <i>NTS</i> matrix elements |

8.5.1 ξ_c parameters

For the ξ_c parameters, the user enters a value between zero (exclusive) and one (inclusive). However, the *FKS* procedure requires the bounds of (10.5) and the parameters hence need to be rescaled accordingly. In principle the user may enter two different values (`xinormcut = xinormcut1` and `xinormcut2`) though this is rarely called for.

8.5.2 Soft and collinear cut parameter

To improve numerical stability, we set events that have a value of $\xi(y)$ lower than `softcut` (`collcut`) to zero.

Warning: This introduces a systematic error that needs to be studied. For small values, the improvement in stability is generally worth a small error that is anyway drowned out by the statistical error

This means that we are changing the integration (8.3)

$$d\sigma_h^{(1)}(\xi_c) \rightarrow d\xi_1 \times \theta(\xi - \text{softcut}) \tag{8.5}$$

$$\times \left(d\Upsilon_1 d\Phi_{n,1}(\xi_1^2 \mathcal{M}_{n+1}^{(0)}) - d\Upsilon_1 d\Phi_{n,1}(\mathcal{E} \mathcal{M}_n^{(0)}) \theta(\xi_c - \xi_1) \right) \tag{8.6}$$

and similarly with `collcut`. We have found that values of `softcut = 1e-10` and `collcut = 1.e-11` give reliable results.

8.6 Random number generation

A Monte Carlo integrator relies on a (pseudo) *random number generator* (*RNG* or PRNG) to work. The pseudo-random numbers need to be of high enough quality, i.e. have no discernible pattern and a long period, to consider each point of the integration independent but the *RNG* needs to be simple enough to be called many billion times without being a significant source of runtime. *RNGs* used in Monte Carlo applications are generally poor in quality and often predictable s.t. they could not be used for cryptographic applications.

A commonly used trade-off between unpredictability and simplicity, both in speed and implementation, is the Park-Miller *RNG*, also known as `minstd`[19]. As a linear congruential generator, its $(k + 1)$ th output x_{k+1} can be found as

$$z_{k+1} = a \cdot z_k \bmod m = a^{k+1} z_1 \bmod m \quad \text{and} \quad x_k = z_k / m \in (0, 1)$$

where m is a large, preferably prime, number and $2 < a < m - 1$ an integer. The initial value z_1 is called the *random seed* and is chosen integer between 1 and $m - 1$. It can easily be seen that any such *RNG* has a fixed period² $p < m$ s.t. $z_{k+p} = z_k$ because any z_{k+1} only depends on z_k and there are finitely many possible z_k . We call the *RNG* attached to (m, a) to be of *full period* if $p = m - 1$, i.e. all integers between 1 and $m - 1$ appear in the sequence z_k .

Assuming $z_1 = 1$ then the existence of p s.t. $z_{p+1} = 1$ is guaranteed by Fermat's Theorem³. This means that the *RNG* is of full period iff a is a primitive root modulo m , i.e.

$$\forall g \text{ co-prime to } m \quad \exists k \in \mathbb{Z} \quad \text{s.t.} \quad a^k \equiv g \pmod{m}$$

Park and Miller suggest to use the Mersenne prime $m = 2^{31} - 1$, noting that there are 534,600,000 primitive roots of which 7 is the smallest. Because $7^b \bmod m$ is also a primitive root as long as b is co-prime to $(m - 1)$, [19] settled on $b = 5$, i.e. $a = 16807$ as a good choice for the multiplier that, per construction, has full period and passes certain tests of randomness.

The points generated by any such *RNG* will fall into $\sqrt[n]{n! \cdot m}$ hyperplanes if scattered in an n dimensional space [16]. However, for bad choices of the multiplier a the number of planes can be a lot smaller⁴.

Presently, the period length of $p = m - 1 = 2^{31} - 2$ is believed to be sufficient though detailed studies quantifying this would be welcome.

² Note that, because of the simple recursion the *RNG* will not repeat any number until the full period is complete

³ If p is prime, for any integer a , $a^p - a$ is a multiple of p .

⁴ An infamous example is `randu` that used $a = 2^{16} + 3$ and $m = 2^{31}$ that in three dimension produces only 15 planes instead of the maximum 2344.

8.7 Differential distributions and intermediary state files

Distributions are always calculated as histograms by binning each event according to its value for the observable S . This is done by having an $(n_b \times n_q)$ -dimensional array⁵ `quant()` where n_q is the number of histograms to be calculated (`nr_q`) and n_b is the number of bins used (`nr_bins`). The weight of each event $d\Phi \times \mathcal{M} \times w$ is added to the correct entry in `bit_it` where $w = \text{wgt}$ is the event weight assigned by vegas.

After each iteration of vegas we add `quant()` (`quant2`) to an accumulator of the same dimensions called `quantsum` (`quantsumsq`). After i iterations, we can calculate the value and error as

$$\frac{d\sigma}{dS} \approx \frac{\text{quantsum}}{\Delta \times i} \quad \text{and} \quad \delta\left(\frac{d\sigma}{dS}\right) \approx \frac{1}{\Delta} \sqrt{\frac{\text{quantsumsq} - \text{quantsum}^2/i}{i(i-1)}}$$

where Δ is the bin-size.

Related to this discussion is the concept of intermediary state files. Their purpose is to record the complete state of the integrator after every iteration in order to recover should the program crash – or more likely be interrupted by a batch system. McMule uses a custom file format `.vegas` for this purpose which uses Fortran’s record-based (instead of stream- or byte-based) format. This means that each entry starts with 32bit unsigned integer, i.e. 4 byte, indicating the record’s size and ends with the same 32bit integer. As this is automatically done for each record, it minimises the amount of metadata that have to be written.

The current version (`v3`) must begin with the magic header and version self-identification shown in [Table 8.3](#). The latter includes file version information and the first five characters the source tree’s `SHAI` hash, obtained using `make hash`.

The header is followed by records describing the state of the integrator as shown in [Table 8.4](#). Additionally to information required to continue integration such as the current value and grid information, this file also has 300 bytes for a message. This is usually set by the routine to store information on the fate of the integration such as whether it was so-far uninterrupted or whether there is reason to believe it to be inconsistent.

The latter point is particularly important. While McMule cannot read intermediary files from a different version of the file format, it will continue any integration for which it can read the state file. This also includes cases where the source tree has been changed. In this case McMule prints a warning but continues the integration deriving potentially inconsistent results.

Table 8.3: The magic header and version information used by `v3`. v_1 indicates the current version number and v_2 whether long integers are used (L) or not (N). s_1 - s_5 indicate the first five characters of the SHA1 hash produced by the source code at compile time (`make hash`).

| | | | | | | | | | | | | | | | | |
|---------------|----|----|----|----|----|-------|-------|-------|-------|-------|-------|-------|----|----|----|----|
| offset | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| hex | 09 | 00 | 00 | 00 | 20 | 4D | 63 | 4D | 75 | 6C | 65 | 20 | 20 | 09 | 00 | 00 |
| ASCII | \t | | | | ‘ | M | c | M | u | l | e | ‘ | ‘ | \t | | |
| offset | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F |
| hex | 00 | 0A | 00 | 00 | 00 | 76 | xx | xx | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 0A |
| ASCII | | \n | | | v | v_1 | v_2 | ‘ | ‘ | ‘ | ‘ | ‘ | ‘ | ‘ | ‘ | \n |
| offset | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
| hex | 00 | 00 | 00 | 05 | 00 | 00 | 00 | xx | xx | xx | xx | xx | 05 | 00 | 00 | 00 |
| ASCII | | | | | | | | s_1 | s_2 | s_3 | s_4 | s_5 | | | | |

⁵ To be precise, the actual dimensions are $(n_b + 2) \times n_q$ to accommodate under- and overflow bins

Table 8.4: The body of a .vegas file storing all important information. Each horizontal line indicates as dressed record. In the offset and length columns, all integers are in hexadecimal notation. Negative numbers count from the end of file (EOF).

| Off | Len | Type | Var. | Comment |
|------|----------------------|-------------------------|---------------|---|
| 0030 | 000C | integer | it | the current iteration |
| 003C | 000C | integer | ndo | subdiv. on an axis |
| 0048 | 0010 | real | si | $\sigma/(\delta\sigma)^2$ |
| 0058 | 0010 | real | swgt | $1/(\delta\sigma)^2$ |
| 0068 | 0010 | real | schi | $(1 - it)\chi + \sigma^2/(\delta\sigma)^2$ |
| 0078 | 1A98 | real(50, 17) | xi | the integration grid |
| 1B10 | 000C | integer | randy | the current <i>random number seed</i> |
| 1B1C | 0014 | integer integer integer | $n_q n_b n_s$ | number of histograms number of bins len. histogram name |
| 1B30 | $10n_q + 8$ | real(n_q) | minv | lower bounds |
| | $n_s n_q + 8$ | character(n_s, n_q) | maxv | upper bounds |
| | $10n_q(n_b + 2) + 8$ | real(n_q, n_b+2) | names | names of S accu. histograms |
| | | real(n_q, n_b+2) | quantsum | accu. histograms squared |
| | | | quantsumsq | |
| -014 | 0010 | real | time | current runtime in seconds |
| -013 | 0134 | character(300) | msg | any message |
| -000 | EOF | | | |

8.8 Basics of containerisation

McMule is Docker-compatible. Production runs should be performed with Docker [17], or its user-space complement *udocker* [12], to facilitate reproducibility and data retention. On Linux, Docker uses *chroot* to simulate an operating system with McMule installed. In our case, the underlying system is Alpine Linux, a Linux distribution that is approximately 5MB in size.

8.8.1 Terminology

To understand Docker, we need to introduce some terms

- An image is a representation of the system's 'hard disk'. One host system can have multiple images. In (u)Docker, the images can be listed with `docker image ls` (*udocker images*).
- Images can be have names, called tags, otherwise Docker assigns a name as the SHA256 hash.
- Because keeping multiple full file systems is rather wasteful, images are split into layers that can be shared among images. In *uDocker*, these are tar files containing the changes made to the file system.
- To execute an image, a *container* needs to be generated. Essentially, this involves uncompressing all layers into a directory and *chrooting* into said directory.

It is important to note, that *containers* are ephemeral, i.e. changes made to the *container* are not stored unless explicitly requested. This is usually not required anyway.

For external interfacing, folders of the host system are mounted into the *container*.

8.8.2 Building images

Docker images are built using Dockerfiles, a set of instructions on how to create the image from external information and a base image. To speed up building of the image, McMule uses a custom base image called `mcmule-pre` that is constructed as follows

```
FROM alpine:3.11

LABEL maintainer="yannick.ulrich@psi.ch"
LABEL version="1.0"
LABEL description="The base image for the full McMule suite"

# Install a bunch of things
RUN apk add py3-numpy py3-scipy ipython py3-pip git tar gfortran gcc make curl musl-dev
RUN echo "http://dl-8.alpinelinux.org/alpine/edge/community" >> /etc/apk/repositories && \
↪ \
    apk add py3-matplotlib && \
    sed -i '$ d' /etc/apk/repositories
```

On top of this, McMule is built

```
FROM yulrich/mcmule-pre:1.0.0

LABEL maintainer="yannick.ulrich@psi.ch"
LABEL version="1.0"
LABEL description="The full McMule suite"
RUN pip3 install git+gitlab.com/mule-tools/pymule.git
COPY . /monte-carlo
WORKDIR /monte-carlo
RUN ./configure
RUN make
```

To build this image, run

```
mcmule$ docker build -t $mytagname . # Using Docker
mcmule$ udocker build -t=$mytagname . # Using udocker
```

The CI system uses `udocker` to perform builds after each push. Note that using `udocker` for building requires a patched version of the code that is available from the McMule collaboration.

8.8.3 Creating containers and running

In Docker, *containers* are usually created and run in one command

```
$ docker run --rm $imagename $cmd
```

The flag `-rm` makes sure the *container* is deleted after it is completed. If the command is a shell (usually `ash`), the flag `-i` also needs to be provided.

For `udocker`, creation and running can be done in two steps

```
$ udocker create $imagename
# this prints the container id
$ udocker run $containerid $cmd
```

(continues on next page)

(continued from previous page)

```
# work in container  
$ udocker rm $containerid
```

or in one step

```
$ udocker run --rm $imagename $cmd
```

Running *containers* can be listed with `udocker ps` and `docker ps`. For further details, the reader is pointed to the manuals of Docker and *udocker*.

Chapter 9

Implementing new processes in McMule

In this section we will discuss how new processes can be added to McMule. Not all of the points below might be applicable to any particular process. Further, all points are merely guidelines that could be deviated from if necessary as long as proper precautions are taken.

As an example, we will discuss how Møller scattering $e^-e^- \rightarrow e^-e^-$ could be implemented.

1. A new *process group* may need to be created if the process does not fit any of the presently implemented groups. This requires a new folder with a makefile as well as modifications to the main makefile as discussed in Section *Creating a new process group*.

In our case, $ee \rightarrow ee$ does not fit any of the groups, so we create a new group that we shall call *ee*.

2. Calculate the tree-level matrix elements needed at *LO* and *NLO*: $\mathcal{M}_n^{(0)}$ and $\mathcal{M}_{n+1}^{(0)}$. This is relatively straightforward and – crucially – unambiguous as both are finite in $d = 4$. We will come back to an example calculation in Section *Example calculations in Mathematica*.
3. A generic matrix element file is needed to store ‘simple’ matrix elements as well as importing more complicated matrix elements. Usually, this file should not contain matrix elements that are longer than a few dozen or so lines. In most cases, this applies to $\mathcal{M}_n^{(0)}$.

After each matrix element, the *PID* needs to be denoted in a comment. Further, all required masses as well as the centre-of-mass energy, called *scms* to avoid collisions with the function $\mathbf{s}(\mathbf{p}_i, \mathbf{p}_j) = 2\mathbf{p}_i \cdot \mathbf{p}_j$, need to be calculated in the matrix element to be as localised as possible.

In the case of Møller scattering, a file `ee/ee_mat_e1.f95` will contain $\mathcal{M}_n^{(0)}$. For example, $\mathcal{M}_n^{(0)}$ is implemented there as shown in Listing 9.1.

Listing 9.1: An example implementation of $\mathcal{M}_n^{(0)}$ for Møller scattering. Note that the electron mass and the centre-of-mass energy are calculated locally. A global factor of $8e^4 = 128\pi^2\alpha^2$ is included at the end.

```
FUNCTION EE2EE(p1, p2, p3, p4)
  !! e-(p1) e-(p2) -> e-(p3) e-(p4)
  !! for massive (and massless) electrons
  implicit none
  real(kind=prec), intent(in) :: p1(4), p2(4), p3(4), p4(4), ee2ee
  real(kind=prec) :: den1, den2, t, scms, m2
  t = sq(p1-p3) ; scms = sq(p1+p2) ; m2 = sq(p1)
  den1 = sq(p1-p3) ; den2 = sq(p1-p4)
```

(continues on next page)

(continued from previous page)

```

ee2ee=(8**m2**2 - 8*m2*scms + 2*s**2 + 2*scms*t + t**2)/den1**2
ee2ee=ee2ee+2*(12*m2**2 - 8*m2*scms + scms**2) / den1 / den2
ee2ee=ee2ee+(24*m2**2 + scms**2 + t**2 - 8*m2*(s + t))/den2**2

```

```
ee2ee = ee2ee * 128*pi**2*alpha**2
```

```
END FUNCTION
```

4. Further, we need an interface file that implements the abstract interface *partInterface* (cf. Section *Particle string framework*). In our case this is called `ee/ee.f95`, as shown in Listing 9.2.

In most cases, the *particle* string framework can automatically define all soft limits and *NTS* matrix elements. In rare cases, this needs to be done manually in `ee/ee.f95`.¹

Listing 9.2: An example implementation of the soft limits for Møller scattering in the particle framework.

```

FUNCTION EE2EE_part(p1, p2, p3, p4)
  !! e-(p1) e-(p2) -> e-(p3) e-(p4)
  !! for massive (and massless) electrons
  implicit none
  real(kind=prec) :: p1(4), p2(4), p3(4), p4(4)
  type(particles) :: ee2ee_part

  ee2ee_part = parts((/part(p1, 1, 1), part(p2, 1, 1), part(p3, 1, -1), part(p4, 1, -1)/))
END FUNCTION

```

5. Because $\mathcal{M}_{n+1}^{(0)}$ is border-line large, we will assume that it will be stored in an extra file, `ee/ee2eeg.f95`. The required functions are to be imported in `ee/ee_mat_e1.f95`.
6. Calculate the one-loop virtual matrix element $\mathcal{M}_n^{(1)}$, renormalised in the *OS* scheme. Of course, this could be done in any regularisation scheme. However, results in McMule shall be in the FDH (or equivalently the FDF) scheme. Divergent matrix elements in McMule are implemented as c_{-1} , c_0 , and c_1

$$\mathcal{M}_n^{(1)} = \frac{(4\pi)^\epsilon}{\Gamma(1-\epsilon)} \left(\frac{c_{-1}}{\epsilon} + c_0 + c_1\epsilon + \mathcal{O}(\epsilon^2) \right).$$

For c_{-1} and c_0 this is equivalent to the conventions employed by Package-X [20] up to a factor $1/16\pi^2$. While not strictly necessary, it is generally advisable to also include c_{-1} in the Fortran code.

For *NLO* calculations, c_1 does not enter. However, we wish to include Møller scattering up to *NNLO* and hence will need it sooner rather than later anyway.

In our case, we will create a file `ee/ee_ee2ee1.f95`, which defines a function

```

FUNCTION EE2EE1(p1, p2, p3, p4, sing, lin)
  !! e-(p1) e-(p2) -> e-(p3) e-(p4)
  !! for massive electrons
  implicit none
  real(kind=prec), intent(in) :: p1(4), p2(4), p3(4), p4(4)

```

(continues on next page)

¹ Further coding may be required if the user needs to isolate different gauge-invariant contributions to the process. For example, for $e\mu \rightarrow e\mu$ scattering, the function `em2em_ee_part = parts((/part(p1, 1, 1), part(p2, 1, 1), part(p3, 1, -1), part(p4, 1, -1)/), "e")` can be used to generate all soft limits due to emissions from the electron line only. Similarly, the function `em2em_mm_part` can be used for all soft limits from the muon line only. However, the function `em2em_em_part = parts((/part(p1, 1, 1), part(p2, 1, 1), part(p3, 1, -1), part(p4, 1, -1)/), "x")` generates only a subset of all soft contributions due to emissions from both lepton lines. Thus, for this mixed case, the required soft limits are hard-coded in `mue/mue.f95`

(continued from previous page)

```

real(kind=prec) :: ee2eel
real(kind=prec), intent(out), optional :: sing, lin
...
END FUNCTION

```

The function shall return c_0 in ee2eel and, if present c_{-1} and c_1 in sing and lin.

- At this stage, a new subroutine in the program test with reference values for all three matrix elements should be written to test the Fortran implementation. This is done by generating a few points using an appropriate phase-space routine and comparing to as many digits as possible using the routine check.

In our case, we would construct a subroutine TESTEEMATEL as shown in Listing 9.3.

Listing 9.3: Test routine for $ee \rightarrow ee$ matrix elements and integrands.
The reference values for the integration are yet to be determined.

```

SUBROUTINE TESTEEMATEL
implicit none
real (kind=prec) :: x(2),y(5)
real (kind=prec) :: single, finite, lin
real (kind=prec) :: weight
integer ido

call blockstart("ee matrix elements")
call initflavour("mu-e", mys=40000._prec)
scms = 40000.
musq = mm**2
x = (/0.75,0.5/)
call psx2(x,q1,me,q2,me,q3,me,q4,me,weight)
call check("ee2ee:",ee2ee (q1,q2,q3,q4),5684.9581122306845)
mat = ee2eel(q1,q2,q3,q4)
call check("ee2eel fin:",mat,221816.84997000592)

call initflavour("mu-e", mys=40000.)
y = (/0.3,0.6,0.8,0.4,0.9/)
call psx3_fks(y,q1,me,q2,me,q3,me,q4,me,q5,weight)
call check("ee2eeg",ee2eeg(q1,q2,q3,q4,q5),434.98382653487744)

call blockend(3)
END SUBROUTINE

SUBROUTINE TESTMEEVEGAS
xinormcut1 = 0.2
xinormcut2 = 0.3

call blockstart("Moller VEGAS test")

call test_INT('ee2ee0', sigma_0, 2,10,10, NaN)
call test_INT('ee2eeF', sigma_0, 2,10,10, NaN)
call test_INT('ee2eeR', sigma_1, 5,10,10, NaN)
call blockend(3)
END SUBROUTINE

```

- In addition, McMule provides built-in routines for testing the convergence of real-emission matrix elements to

the corresponding soft limits, for ever smaller photon energies.

In our case, we would construct a subroutine

```
SUBROUTINE TESTEESOFTN1
implicit none
real(kind=prec) y0(5)

call blockstart("e-e \xi->0")
call initflavour("muone")
xinormcut1 = 0.3
y0 = (/0.01,0.6,0.8,0.999,0.01/)

call test_softlimit(y0, ["ee2eeR"])
END SUBROUTINE
```

where `test_soft_limit` compares the real matrix element (`ee2eeR`) with its soft limit implemented in `ee/ee.f95`. The comparison starts at the energy defined by the phase-space point `y0` and proceeds with ever smaller photon energies. A flavour (`muone`) as well as `xinormcut1` are required in order to complete the phase-space generation.

9. Define a default observable in user for this process. This observable must be defined for any *which_piece* that might have been defined and test all relevant features of the implementation such as polarisation if applicable.
10. Add the matrix elements to the integrands defined in `integrands.f95`. This is done by adding a new case corresponding to the new *which_piece* in the `initpiece()`.
 - for a IR-finite, non-radiative piece (i.e. *LO* but also *VP*), add

```
case('eb2eb0')
call set_func('00000000', eb2eb)
ps => psx2 ; fxn => sigma_0
nparticle = 4 ; ndim = 2
masses(1:4) = (/ Me, Me, Me, Me /)
```

which adds a *which_piece* `ee2ee0` that is calculated using the matrix element `ee2ee`. The phase space is generated with `psx4()` and integrated using `sigma_0()` (no subtraction). The process involves 4 particles and, since it is a $2 \rightarrow 2$ process, the integration is two-dimensional. The masses of the involved particles are all *Me*.

- for pieces with an IR cancellation between real and virtual corrections, i.e. calculations involving photon loops, we need to specify `xieik1` (at one-loop) and/or `xieik2` (at two-loop)

```
case('eb2ebF')
call set_func('00000000', eb2ebf)
ps => psx2 ; fxn => sigma_0
nparticle = 4 ; ndim = 2
masses(1:4) = (/ Me, Me, Me, Me /)
xieik1 = xinormcut*(1.-(2*me)**2/scms)
```

One needs to take care that ξ_c is properly normalised. The user will enter a value from 0 to 1 which needs to be matched to ξ_c as defined in (10.5)

- for real corrections we need to use a subtracting integrand, i.e. `sigma_1()` for single-real and `sigma_2()` for double-real corrections.

```

case('eb2ebR')
  call set_func('00000000', eb2ebg)
  call set_func('00000001', eb2eb)
  call set_func('11111111', eb2eb_part)
  ps => psx3_fks ; fxn => sigma_1
  nparticle = 5 ; ndim = 5
  masses(1:5) = (/ Me, Me, Me, Me, 0._prec /)
  xicut1 = xinormcut*(1.-(2*me)**2/scms)

```

Additionally to the real matrix element `eb2ebg`, we also specified the reduced matrix element `eb2eb` and the *particle* string function `eb2eb_part`. Note further changes to the number of particles and phase space dimension to accommodate the extra photon.

Additionally to these required parameters, there are number of optional parameters such as `symmfac` (which is set to 2 for $e^-e^- \rightarrow e^-e^-$ because of the two indistinguishable final state particles), `polarised` (whether to consider the process polarised), and `softCut` and `collCut`. For a full list of parameters, see Section *Optional parameters for integrands*.

Once *integrands* are implemented, a second test routine should be written that runs short integrations against a reference value. Because `test_INT` uses a fixed *random seed*, this is expected to be possible very precisely. Unfortunately, COLLIER [6] might produce slightly different results on different machines. Hence, integrands involving complicated loop functions are only required to agree up to $\mathcal{O}(10^{-8})$.

11. After some short test runs, it should be clear whether new phase-space routines are required. Add those, if need be, to `phase_space` as described in Section *Phase-space generation*.
12. Per default the stringent *soft cut*, that may be required to stabilise the numerical integration (cf. Section *Implementation of FKS schemes*), is set to zero. Study what the smallest value is that still permits integration.
13. Perform very precise ξ_c independence studies. Tips on how to do this can be found in Section *Study of ξ_c dependence*.

At this stage, the *NLO* calculation is complete and may, after proper integration into McMule and adherence to coding style has been confirmed, be added to the list of McMule processes in a new release. Should *NNLO* precision be required, the following steps should be taken

14. Calculate the real-virtual and double-real matrix elements $\mathcal{M}_{n+1}^{(1)}$ and $\mathcal{M}_{n+2}^{(0)}$ and add them to the test routines as well as integrands.
15. Prepare the n -particle contribution $\sigma_n^{(2)}$. In a pinch, massified results can be used also for $\hat{\mathcal{E}}(\xi_c)\mathcal{M}_n^{(1)}$ though of course one should default to the fully massive results.
16. Study whether the pre-defined phase-space routines are sufficient. Even if it was possible to use an old phase-space at *NLO*, this might no longer work at *NNLO* due to the added complexity. Adapt and partition further if necessary, adding more test integrations in the process.
17. Perform yet more detailed ξ_c and *soft cut* analyses.

In the following we comment on a few aspects of this procedure such as the ξ_c study (Section *Study of ξ_c dependence*), the calculation of matrix elements (Section *Example calculations in Mathematica*), and a brief style guide for McMule code (Section *Coding style and best practice*).

9.1 Creating a new process group

Adding Møller scattering to McMule, the example discussed above, requires the addition of a new *process group* ee. For this we create a new folder in McMule called ee containing a meson.build (Listing 9.4), a mat_el file (ee/ee_mat_el.f95, Listing 9.5) and a module file (ee/ee.f95, Listing 9.6). Finally, the name of the *process group* needs to be added to the GROUPS and WGROUPS variables of the makefile.

Listing 9.4: The bare meson.build for the new *process group* ee. Large matrix elements are stored in extra files such as ee/ee2eeg.f95 or ee/ee_ee2eel.f95.

```
# add files that require compiler optimisation here
src_opt = []

# add all other files here
src = [
  'ee_ee2eel.f95',
  'ee_ee2eeg.f95',
  'ee_mat_el.f95',
  'ee.f95'
]

lib_opt = static_library(
  'ee_opt', src_opt, dependencies : mcmule_pre_dep,
  fortran_args : [opt_level1]
)
dep_opt = declare_dependency(
  link_with : lib_opt,
  include_directories : lib_opt.private_dir_include()
)
groups_lib += static_library(
  'ee', [src, lib_opt.extract_all_objects(recursive : true)],
  dependencies : [mcmule_pre_dep, dep_opt]
)
groups_incl += lib_opt.private_dir_include()

src_test += '../ee/ee_test.f95'

foreach i : src + src_opt
  src_groups += 'ee/' + i
endforeach
```

Listing 9.5: The file ee/ee_mat_el.f95 imports the complicated matrix elements ee2eel and ee2eegl, defines the simple matrix element ee2ee as per 16, and provides an interface for the $\mathcal{M}_n^{(1)f}$ that is called from *integrands*.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      MODULE EE_MAT_EL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

use functions
use ee_ee2eel, only: ee2eel
```

(continues on next page)

(continued from previous page)

```

use ee_ee2eeg, only: ee2eeg
implicit none

contains

FUNCTION EE2EE(p1,p2,q1,q2)
  !! e-(p1) e-(p2) -> e-(q1) e-(q2)
  !! for massive electrons
  ...
END FUNCTION EE2EE

FUNCTION EE2EEF(p1,p2,q1,q2)
  !! e-(p1) e-(p2) -> e-(q1) e-(q2)
  !! massive electrons
real(kind=prec) :: p1(4),p2(4),q1(4),q2(4)
real(kind=prec) :: ee2eef, mat0, Epart

Epart = sqrt(sq(p1+p2))
mat0 = ee2ee(p1,p2,q1,q2)

ee2eef = ee2eel(p1,p2,q1,q2) + alpha / (2 * pi) * mat0 * &
  Ieik(xieik1, Epart, parts((/ part(p1, 1, 1), part(p2, 1, 1), &
    part(q1, 1, -1), part(q2, 1, -1) /)))

END FUNCTION EE2EEF
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                END MODULE EE_MAT_EL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

Listing 9.6: The module file ee/ee.f95 which imports all matrix elements of ee_mat_el and defines the soft limits.

```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                MODULE EE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

use functions
use phase_space, only: ksoft, ksoftA, ksoftB
use ee_mat_el
implicit none
contains

FUNCTION EE2EE_part(p1, p2, p3, p4)
  !! e-(p1) e-(p2) --> e-(p3) e-(p4)
  !! both massive and massless electrons
real (kind=prec) :: p1(4),p2(4),p3(4),p4(4)
type(particles) :: ee2ee_part
ee2ee_part = parts((/part(p1, 1, 1), part(p2, 1, 1), part(p3, 1, -1), part(p4, 1, -1)/
←))
END FUNCTION EE2EE_part

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

(continues on next page)

END MODULE EE

!!

9.2 Study of ξ_c dependence

When performing calculations with McMule, we need to check that the dependence of the unphysical ξ_c parameter introduced in the *FKS* scheme (cf. Appendix *The FKS² scheme*) actually drops out at *NLO* and *NNLO*. In principle it is sufficient to do this once during the development phase. However, we consider it good practice to also do this (albeit with a reduced range of ξ_c) for production runs.

Because the ξ_c dependence is induced through terms as $\xi_c^{-2\epsilon}/\epsilon$, we know the functional dependence of $\sigma_{n+j}^{(\ell)}$. For example, at *NLO* we have

$$\sigma_n^{(1)}(\xi_c) = a_{0,0} + a_{0,1} \log(\xi_c) \quad (9.1)$$

$$\sigma_{n+1}^{(1)}(\xi_c) = a_{1,0} + a_{1,1} \log(\xi_c) \quad (9.2)$$

where ξ_c independence of $\sigma^{(1)}$ of course requires

$$a_{0,1} + a_{1,1} = 0 \quad (9.3)$$

At *NNLO* we have

$$\sigma_n^{(2)}(\xi_c) = a_{0,0} + a_{0,1} \log(\xi_c) + a_{0,2} \log(\xi_c)^2, \quad (9.4)$$

$$\sigma_{n+1}^{(2)}(\xi_c) = a_{1,0} + a_{1,1} \log(\xi_c) + a_{1,2} \log(\xi_c)^2, \quad (9.5)$$

$$\sigma_{n+2}^{(2)}(\xi_c) = a_{2,0} + a_{2,1} \log(\xi_c) + a_{2,2} \log(\xi_c)^2. \quad (9.6)$$

We require

$$a_{0,i} + a_{1,i} + a_{2,i} = 0$$

for $i = 1, 2$. However, the *IR* structure allows for an even stronger statement for the $a_{j,2}$ terms

$$a_{0,2} = a_{2,2} = -\frac{a_{1,2}}{2}.$$

Of course we cannot directly calculate any of the $a_{1,i}$ or $a_{2,i}$ because we use numerical integration to obtain the $\sigma_{n+j}^{(\ell)}$. Still, knowing the coefficients can be extremely helpful when debugging the code or to just quantify how well the ξ_c dependence vanishes. Hence, we use a fitting routine to fit the Monte Carlo results *after* any phase-space partitioning has been undone. Sometimes non of this is sufficient to pin-point the source of a problem to any one integrand. However, if the goodness of, for example, $\sigma_{n+2}^{(2)}(\xi_c)$ is much worse than the one for $\sigma_{n+1}^{(2)}(\xi_c)$, a problem in the double-real corrections can be expected.

9.3 Example calculations in Mathematica

A thorough understanding of one-loop matrix elements is crucial for any higher-order calculation. In McMule, one-loop matrix elements either enter as the virtual contribution to *NLO* corrections or the real-virtual contribution in *NNLO* calculations. In any case, a fast numerical routine is required that computes the matrix element.

We perform all one-loop calculations in FDF as this is arguably the simplest scheme available. For theoretical background, we refer to [25] and references therein.

We use Qgraf for the diagram generation. Using the in-house Mathematica package qgraf we convert Qgraf's output for manipulation with Package-X [20]. This package is available on request through the McMule collaboration

<https://gitlab.com/mule-tools/qgraf>

An example calculation for the one-loop calculation of $\mu \rightarrow \nu \bar{\nu} e \gamma$ can be found in Listing 9.7. Of course this example can be made more efficient by, for example, feeding the minimal amount of algebra to the loop integration routine.

When using qgraf for fdf some attention needs to be paid when considering diagrams with closed fermion loops. By default, qgraf.wl evaluates these traces in d dimensions. RunQgraf has an option to keep this from happening.

Listing 9.7: An example on how to calculate the renormalised one-loop matrix element for $\mu \rightarrow \nu \bar{\nu} e$ in fdf.

```
<<qgraf.wl
onshell = {
  p.p -> M^2, q.q -> m^2, p.q -> s/2
};

A0 = (4GF/Sqrt[2]) "diag1"/.RunQgraf[{"mum"}, {"nu", "elm"}, 0] //. {
  line[_ , x_] -> x, p1->p, q1->p-q, q2->q, _δZ | δm -> 0
};
A1 = pref /. RunQgraf[{"mum"}, {"nu", "elm"}, 1] //. {
  line[_ , x_] -> x, p1->p, q1->p-q, q2->q, _δZ | δm -> 0
};

M0=Block[{Dim=4}, Simplify[Contract[
  1/2 Z2[m] Z2[M] FermionSpinSum[
    A0 /. DiracPL -> (Dirac1 - Z5 γ5)/2,
    A0 /. DiracPL -> (Dirac1 + Z5 γ5)/2
  ]
]] /. onshell] /. {
  Z2[M_] -> 1 + (α/(4π)) (-3/(2ε)-5/2 + 3/2 Log[M^2/Mu^2]),
  Z5 -> 1 - (α/(4π))
};

M1=Block[{Dim=4}, Simplify[Contract[
  1/2 FermionSpinSum[
    A1/.γ.k1 -> γ. 4[k1]+I γ5 μ,
    A0
  ]
]] /. onshell /. {
  μ^n_ /; EvenQ[n] -> μ2^(n/2), μ -> 0
} /. {
  4[k1]. 4[k1] -> k1.k1 + μ2, 4[k1] -> k1
}}];
```

(continues on next page)

(continued from previous page)

```

M1bare = Simplify[KallenExpand[LoopRefine[LoopRelease[
  Pro2LoopIntegrate[
    Coefficient[M1,  $\mu$ 2, 0]/(16  $\pi$ ^2)
  ]
  +  $\mu$ Integrate[
    Coefficient[M1,  $\mu$ 2, 1]/(64  $\pi$ ^3),
    1
  ],
  onshell
]]] /. e -> Sqrt[4  $\pi$  $\alpha$ ]];

```

There is a subtlety here that only arise for complicated matrix elements. Because the function Package-X uses for box integrals, `ScalarD0IR6()`, is so complicated, no native Fortran implementation exists in McMule. Instead, we are defaulting to COLLIER [6] and should directly evaluate the finite part of the PVD function above. The same holds true for the more complicated triangle functions. In fact, only the simple `DiscB()` and `ScalarC0IR6()` are natively implemented without need for external libraries. For any other functions, a judgement call is necessary of whether one should `LoopRefine` the finite part in the first place. In general, if an integral can be written through logarithms and dilogs of simple arguments (resulting in real answers) or `DiscB()` and `ScalarC0IR6()`, it makes sense to do so. Otherwise, it is often easier to directly link to COLLIER.

9.4 Coding style and best practice

A large-scale code base like McMule cannot live without some basic agreements regarding coding style and operational best practice. These range from a (recommended but not enforced) style guide over the management of the git repository to how to best run McMule in development scenarios. All aspects have been discussed within the McMule collaboration.

Fortran code in McMule is (mostly) written in accordance with the following style guide. If new code is added, compliance would be appreciated but deviation is allowed if necessary. If in doubt, contact any member of the McMule collaboration.

- Indentation width is two spaces. In Vim this could be implemented by adding the following to `.vimrc`

```

autocmd FileType fortran set tabstop=8 softtabstop=0 expandtab shiftwidth=2 smarttab

```

- Function and subroutine names are in all-upper case.
- A function body is not indented beyond its definition.
- When specifying floating point literals specify the precision when possible, i.e. `1._prec`.
- Integrand should have `ndim` specified.
- Internal functions should be used where available.
- Masses and other kinematic parameters must be calculated in the matrix elements as local variables; using the global parameters `Mm` and `Me` is strictly forbidden.
- These rules also hold for matrix elements.

For python code, i.e. `pymule` as well as the analysis code, PEP8 compliance is strongly encouraged with the exception of E231 (Missing whitespace after `,`, `;`, and `:`), E731 (Do not assign a lambda expression, use a `def`) as well, in justified cases, i.e. if required by the visual layout, E272 (Multiple spaces before keyword), and E131 (Continuation line unaligned for hanging indent).

McMule uses a git repositories for version management. Development usually happens on feature branches that are merged into the `devel` branch semi-frequently by the McMule collaboration after sufficient vetting was performed.

Finally, once a project has been finished, the `devel` branch gets merged into the `release` branch that is to be used by McMule's users.

In general, developers are encouraged to not commit wrong or unvetted code though this can obviously not be completely avoided in practice. To avoid uncontrollable growth of the git repository, large files movements are strongly discouraged. This also means that matrix elements should not be completely overhauled barring unanimous agreement. Instead, developers are encouraged to add a new matrix element file and link to that instead.

Even when running McMule for development purposes the usage of *menu files* is strongly encouraged because the code will do its utmost to automatically document the run by storing the git version as well as any modification thereof. This allows for easy and unique reconstruction of what was running. For production runs this is not optional; these must be conducted with *menu files* after which the run folder must be stored with an analysis script and all data on the AFS as well as the user file library to ensure data retention.

Chapter 10

The FKS² scheme

In the following we very briefly review the *FKS* [28, 29] and *FKS*² schemes [8] though this is not meant as an introduction into these schemes. For this see [8, 23, 25]. Here, we just give a schematic overview with the basic information required to understand the structure of the code.

The core idea of this method is to render the phase-space integration of a real matrix element finite by subtracting all possible soft limits. The subtracted pieces are partially integrated over the phase space and combined with the virtual matrix elements to form finite integrands.

The *NLO* corrections $\sigma^{(1)}$ to a cross section are split into a n particle and $(n + 1)$ particle contribution and are written as

$$\sigma^{(1)} = \sigma_n^{(1)}(\xi_c) + \sigma_{n+1}^{(1)}(\xi_c) \quad (10.1)$$

$$\sigma_n^{(1)}(\xi_c) = \int d\Phi_n^{d=4} \left(\mathcal{M}_n^{(1)} + \hat{\mathcal{E}}(\xi_c) \mathcal{M}_n^{(0)} \right) = \int d\Phi_n^{d=4} \mathcal{M}_n^{(1)f} \quad (10.2)$$

$$\sigma_{n+1}^{(1)}(\xi_c) = \int d\Phi_{n+1}^{d=4} \left(\frac{1}{\xi_1} \right)_c (\xi_1 \mathcal{M}_{n+1}^{(0)f}) \quad (10.3)$$

In (10.3), ξ_1 is a variable of the $(n + 1)$ parton phase space $d\Phi_{n+1}^{d=4}$ that corresponds to the (scaled) energy of the emitted photon. For $\xi_1 \rightarrow 0$ the real matrix element $\mathcal{M}_{n+1}^{(0)f}$ develops a singularity. The superscripts (0) and f indicate that the matrix element is computed at tree level and is finite, i.e. free of explicit infrared poles $1/\epsilon$. In order to avoid an implicit infrared pole upon integration, the ξ_1 integration is modified by the factor $\xi_1(1/\xi_1)_c$, where the distribution $(1/\xi_1)_c$ acts on a test function f as

$$\int_0^1 d\xi_1 \left(\frac{1}{\xi_1} \right)_c f(\xi_1) \equiv \int_0^1 d\xi_1 \frac{f(\xi_1) - f(0)\theta(\xi_c - \xi_1)}{\xi_1} \quad (10.4)$$

Thus, for $\xi_1 < \xi_c$, the integrand is modified through the subtraction of the soft limit. This renders the integration finite. However, it also modifies the result. The missing piece of the real corrections can be trivially integrated over ξ_1 . This results in the integrated eikonal factor $\hat{\mathcal{E}}(\xi_c)$ times the tree-level matrix element for the n particle process, $\mathcal{M}_n^{(0)}$. The factor $\hat{\mathcal{E}}(\xi_c)$ has an explicit $1/\epsilon$ pole that cancels precisely the corresponding pole in the virtual matrix element $\mathcal{M}_n^{(1)}$. Thus, the combined integrand of (10.2) is free of explicit poles, hence denoted by $\mathcal{M}_n^{(1)f}$, and can be integrated numerically over the n particle phase space $d\Phi_n^{d=4}$.

The parameter ξ_c that has been introduced to split the real corrections can be chosen arbitrarily as long as

$$0 < \xi_c \leq \xi_{\max} = 1 - \frac{(\sum_i m_i)^2}{s} \quad (10.5)$$

where the sum is over all masses in the final state. The ξ_c dependence has to cancel exactly between (10.2) and (10.3) since at no point any approximation was made in the integration. Checking this independence is a very useful tool to test the implementation of the method, as well as its numerical stability.

The finite matrix element $\mathcal{M}_n^{(1)f}$ is simply the first-order expansion of the general YFS exponentiation formula for soft singularities

$$e^{\hat{\mathcal{E}}} \sum_{\ell=0}^{\infty} \mathcal{M}_n^{(\ell)} = \sum_{\ell=0}^{\infty} \mathcal{M}_n^{(\ell)f} = \mathcal{M}_n^{(0)} + \left(\mathcal{M}_n^{(1)} + \hat{\mathcal{E}}(\xi_c) \mathcal{M}_n^{(0)} \right) + \mathcal{O}(\alpha^2) \quad (10.6)$$

where we exploited the implicit factor α in $\hat{\mathcal{E}}$.

For QED with massive fermions this scheme can be extended to *NNLO* and, in fact beyond. The *NNLO* corrections are split into three parts

$$\sigma_n^{(2)}(\xi_c) = \int d\Phi_n^{d=4} \left(\mathcal{M}_n^{(2)} + \hat{\mathcal{E}}(\xi_c) \mathcal{M}_n^{(1)} + \frac{1}{2!} \mathcal{M}_n^{(0)} \hat{\mathcal{E}}(\xi_c)^2 \right) = \int d\Phi_n^{d=4} \mathcal{M}_n^{(2)f} \quad (10.7)$$

$$\sigma_{n+1}^{(2)}(\xi_c) = \int d\Phi_{n+1}^{d=4} \left(\frac{1}{\xi_1} \right)_c \left(\xi_1 \mathcal{M}_{n+1}^{(1)f}(\xi_c) \right) \quad (10.8)$$

$$\sigma_{n+2}^{(2)}(\xi_c) = \int d\Phi_{n+2}^{d=4} \left(\frac{1}{\xi_1} \right)_c \left(\frac{1}{\xi_2} \right)_c \left(\xi_1 \xi_2 \mathcal{M}_{n+2}^{(0)f} \right) \quad (10.9)$$

Thus we have to evaluate n parton contributions, single-subtracted $(n+1)$ parton contributions, and double-subtracted $(n+2)$ parton contributions. This structure will be mirrored in the Fortran code. The ξ_c dependence cancels, once all three contributions are taken into account. For this subtraction method we need the matrix elements with massive fermions. If the two-loop amplitudes are available only for massless fermions, it is possible to use massification [7].

10.1 FKS^ℓ: extension to N^ℓLO

The pattern that has emerged in the previous cases leads to the following extension to an arbitrary order ℓ in perturbation theory:

$$d\sigma^{(\ell)} = \sum_{j=0}^{\ell} d\sigma_{n+j}^{(\ell)}(\xi_c) \quad (10.10)$$

$$d\sigma_{n+j}^{(\ell)}(\xi_c) = d\Phi_{n+j}^{d=4} \frac{1}{j!} \left(\prod_{i=1}^j \left(\frac{1}{\xi_i} \right)_c \xi_i \right) \mathcal{M}_{n+j}^{(\ell-j)f}(\xi_c) \quad (10.11)$$

The eikonal subtracted matrix elements

$$\mathcal{M}_m^{(\ell)f} = \sum_{j=0}^{\ell} \frac{\hat{\mathcal{E}}^j}{j!} \mathcal{M}_m^{(\ell-j)}$$

(with the special case $\mathcal{M}_m^{(0)f} = \mathcal{M}_m^{(0)}$ included) are free from $1/\epsilon$ poles, as indicated in (10.6). Furthermore, the phase-space integrations are manifestly finite.

Chapter 11

Glossary

11.1 Acronyms

| | |
|-------------|---|
| BR | a branching ratio |
| DSL | a domain specific language |
| EW | electroweak |
| FKS | the Frixione-Kunszt-Signer scheme used in McMule. See Section <i>The FKS² scheme</i> . |
| FSR | final state radiation |
| IR | infra-red |
| HVP | hadronic vacuum polarisation |
| ISR | initial state radiation |
| LO | leading order |
| LP | leading power |
| NLO | next-to-leading order |
| NLP | next-to-leading power |
| NNLO | next-to-next-to-leading order |
| NTS | next-to-soft |

OS

on-shell renormalisation scheme in which the masses correspond to the poles of the propagators and $\alpha = \alpha(q^2 = 0)$ in the Thomson limit

PCS

pseudo-collinear singularities, the numerical instabilities in

$$\mathcal{M}_{n+1}^{(\ell)} \propto \frac{1}{q \cdot k} = \frac{1}{\xi^2} \frac{1}{1 - y\beta}$$

where y is the angle between photon (k) and electron (q). For large velocities β (or equivalently small masses), this becomes almost singular as $y \rightarrow 1$.

PID

particle identification, the ordering of particles in the code

RNG

random number generator, used to generate pseudo-random numbers for the Monte Carlo generation. See Section *Random number generation*

SHA1

secure-hashing-algorithm-1, used for hashing McMule's source code in autoversioning

SM

the Standard Model of particle physics

TOML

the name of the markup language used for *config files*. See <https://toml.io/> for more information

VP

vacuum polarisation

11.2 Technical terms

config file

a shell file specifying, among other things, the statistics to be used

containerisation

the concept of bundling all dependencies etc. with McMule. See Sections *Basics of containerisation* and *Basics of containerisation*

container

a container that has bundled all dependencies etc. with McMule. See Sections *Basics of containerisation* and *Basics of containerisation*

corner region

a region of phase space where the mapping defined in Section *Phase-space generation* is not unique. The corner region refers to the smaller part of this double mapping.

counter-event

the soft event that gets subtracted in FKS, cf. (8.3)

distance function

A function which determines the distance between two points in the momentum phasespace.

event

the hard event that does not get subtracted in FKS, cf. (8.3)

executor

a pymule plugin that executes a menu file

fatal

A test which is not allowed to fail, such as the Soft photon limit or Swap symmetry tests.

full period

a surjective *RNG*

generic piece

a generic piece describes a part of the calculation such as the real or virtual corrections that themselves may be further subdivided as is convenient.

generic process

A generic process is a prototype for the physical process such as $\ell p \rightarrow \ell p$ where the flavour of the lepton ℓ is left open.

image

a disk image that has bundled all dependencies etc. with McMule. See Sections *Basics of containerisation* and *Basics of containerisation*

menu file

A menu file contains a list of jobs to be computed s.t. the user will only have to vary the random seed and ξ_c by hand as the statistical requirements are defined globally in a *config file*.

measurement function

A function that takes as arguments the four-momenta of all particles involved in the reaction and returns the experimentally measured quantity.

non-fatal

A test which is allowed to fail, but ideally shouldn't. Includes most lorentz transformations.

process group

Processes are grouped into process groups if they share matrix elements such as $\mu \rightarrow \nu\bar{\nu}e$ and $\mu \rightarrow \nu\bar{\nu}e\gamma$ (*mudec*) or $e\mu \rightarrow e\mu$ and $\ell p \rightarrow \ell p$ (*mue*).

random seed

the initial value of the *RNG*. In McMule this may be between 1 and $2^{31} - 2$. See Section *Random number generation* for further details.

shared library

a library that can be loaded dynamically. On Linux system the customary filename suffix is *.so*. McMule can be used as a shared library and also load a shared library for the user file.

soft cut

a value of ξ below which no subtraction takes place and the integrand is set to zero

submission script

a script that is provided by pymule to run a *menu file*.

userfile

the **.f95* or **.c* file which defines the QUANT function or *measurement function* along with histogram variables.

which_piece

A which piece is a single subdivision of a *generic piece* designed to build up to a full calculation

Chapter 12

Bibliography

Chapter 13

Particle ID

The following table lists the `which_pieces` of McMule as well as the corresponding *PID*. For example, when calculating the process $\mu^+ \rightarrow e^+ \nu \bar{\nu} e^+ e^-$, the measurement function may receive up to seven arguments that can be mapped to particles as follows:

```

FUNCTION QUANT(Q1,Q2,Q3,Q4,Q5,Q6,Q7)
real(kind=prec) :: q1(4) ! incoming muon+
real(kind=prec) :: q2(4) ! outgoing electron+
real(kind=prec) :: q3(4) ! outgoing neutrino, averaged over
real(kind=prec) :: q4(4) ! outgoing neutrino, averaged over
real(kind=prec) :: q5(4) ! outgoing electron-
real(kind=prec) :: q6(4) ! outgoing electron+
real(kind=prec) :: q7(4) ! outgoing optional photon

pol1 = (/ 0., 0., -0.85, 0. /) ! set incoming muon polarisation
...
END FUNCTION

```

Additionally to the particle mapping, we see that neutrinos are averaged over as indicated by $[\bar{\nu}_\mu \nu_e]$. We can further tell that the first initial state particle is polarised since the P-column lists a 1.

| which_piece | P? | p_1 | p_2 |
|-------------|----|---------|-------------------|
| m2enn0 | 1 | μ^- | $\rightarrow e^-$ |
| m2ennF | 1 | | |
| m2ennR | 1 | μ^- | $\rightarrow e^-$ |
| m2ennFFz | 1 | μ^- | $\rightarrow e^-$ |
| m2ennFF | 1 | | |
| m2ennLL | 1 | | |
| m2ennRF | 1 | μ^- | $\rightarrow e^-$ |
| m2ennRR | 1 | μ^- | $\rightarrow e^-$ |
| m2ennNF | 1 | μ^- | $\rightarrow e^-$ |
| m2ej0 | 1 | μ^- | $\rightarrow e^-$ |
| m2ejF | 1 | | |
| m2ejR | 1 | μ^- | $\rightarrow e^-$ |
| m2ejg0 | 1 | | |
| m2enng0 | 1 | μ^- | $\rightarrow e^-$ |
| m2enngV | 1 | | |

| which_piece | P? | p_1 | p_2 |
|---------------|----|----------|---------------------|
| m2enngC | 1 | | |
| m2enngR | 1 | μ^- | $\rightarrow e^-$ |
| m2ennee0 | 1 | μ^- | $\rightarrow e^-$ |
| m2enneeV | 1 | | |
| m2enneeA | 1 | | |
| m2enneeC | 1 | | |
| m2enneeR | 1 | μ^+ | $\rightarrow e^+$ |
| t2mnnee0 | 1 | τ^- | $\rightarrow \mu^-$ |
| t2mnneeV | 1 | | |
| t2mnneeA | 1 | | |
| t2mnneeC | 1 | | |
| t2mnneeR | 1 | τ^+ | $\rightarrow \mu^+$ |
| em2em0 | 0 | e^- | μ^- |
| em2emV | 0 | | |
| em2emC | 0 | | |
| em2emFEE | 0 | | |
| em2emFEM | 0 | | |
| em2emFMM | 0 | | |
| em2emREE | 0 | e^- | μ^- |
| em2emREE15 | 0 | | |
| em2emREE35 | 0 | | |
| em2emREEco | 0 | | |
| em2emREM | 0 | | |
| em2emREM15 | 0 | | |
| em2emREM35 | 0 | | |
| em2emREMco | 0 | | |
| em2emRMM | 0 | | |
| em2emFFEEEE | 0 | e^- | μ^- |
| em2emFFEEEEz | 0 | | |
| em2emFFMMMM | 0 | | |
| em2emFFMIXDz | 0 | | |
| em2emFF31z | 0 | | |
| em2emFF22z | 0 | | |
| em2emFF13z | 0 | | |
| em2emFFz | 0 | | |
| em2emRFEEEE | 0 | e^- | μ^- |
| em2emRFEEEE15 | 0 | | |
| em2emRFEEEE35 | 0 | | |
| em2emRFEEEEco | 0 | | |
| em2emRFMMMM | 0 | | |
| em2emRFMIXD | 0 | | |
| em2emRFMIXD15 | 0 | | |
| em2emRF3115 | 0 | | |
| em2emRF2215 | 0 | | |
| em2emRF1315 | 0 | | |
| em2emRFMIXD35 | 0 | | |
| em2emRF3135 | 0 | | |
| em2emRF2235 | 0 | | |
| em2emRF1335 | 0 | | |
| em2emRFMIXDco | 0 | | |

| which_piece | P? | p_1 | p_2 |
|-----------------|----|---------|---------|
| em2emRF | 0 | | |
| em2emRF15 | 0 | | |
| em2emRF35 | 0 | | |
| em2emRFco | 0 | | |
| em2emRREEEE | 0 | e^- | μ^- |
| em2emRREEEE1516 | 0 | | |
| em2emRREEEE3536 | 0 | | |
| em2emRREEEEc | 0 | | |
| em2emRRMMM | 0 | | |
| em2emRRMIXD | 0 | | |
| em2emRRMIXD1516 | 0 | | |
| em2emRR311516 | 0 | | |
| em2emRR221516 | 0 | | |
| em2emRR131516 | 0 | | |
| em2emRRMIXD3536 | 0 | | |
| em2emRR313536 | 0 | | |
| em2emRR223536 | 0 | | |
| em2emRR133536 | 0 | | |
| em2emRRMIXDc | 0 | | |
| emZemOX | 0 | e^- | μ^+ |
| emZemFX | 0 | | |
| emZemRX | 0 | | |
| em2emA | 0 | e^- | μ^- |
| em2emAA | 0 | | |
| em2emAFEE | 0 | | |
| em2emAFMM | 0 | | |
| em2emAFEM | 0 | | |
| em2emAF | 0 | | |
| em2emAREE | 0 | e^- | μ^- |
| em2emAREE15 | 0 | | |
| em2emAREE35 | 0 | | |
| em2emARMM | 0 | | |
| em2emAREM | 0 | | |
| em2emAR | 0 | | |
| em2emAR15 | 0 | | |
| em2emAR35 | 0 | | |
| em2emNFEE | 0 | e^- | μ^- |
| em2emNFMM | 0 | | |
| em2emNFEM | 0 | | |
| em2emNF | 0 | | |
| em2emNFEMCT | 0 | | |
| mp2mp0 | 0 | μ^- | p |
| mp2mp0nuc | 0 | | |
| mp2mpF | 0 | | |
| mp2mpFMP | 0 | | |
| mp2mpR | 0 | μ^- | p |
| mp2mpRMP | 0 | μ^- | p |
| mp2mpR15 | 0 | μ^- | p |
| mp2mpR35 | 0 | | |
| mp2mpRco | 0 | | |

| which_piece | P? | p_1 | p_2 |
|-------------|----|---------|-------|
| mp2mpRMP15 | 0 | μ^- | p |
| mp2mpRMP35 | 0 | | |
| mp2mpRMPco | 0 | | |
| mp2mpFF | 0 | μ^- | p |
| mp2mpRF | 0 | μ^- | p |
| mp2mpRF15 | 0 | | |
| mp2mpRF35 | 0 | | |
| mp2mpRFco | 0 | | |
| mp2mpRR | 0 | μ^- | p |
| mp2mpRR1516 | 0 | | |
| mp2mpRR3536 | 0 | | |
| mp2mpRRc | 0 | | |
| mp2mpA | 0 | μ^- | p |
| mp2mpAA | 0 | | |
| mp2mpAF | 0 | | |
| mp2mpAR | 0 | μ^- | p |
| mp2mpAR15 | 0 | | |
| mp2mpAR35 | 0 | | |
| mp2mpNF | 0 | μ^- | p |
| ee2mm0 | 2 | e^- | e^+ |
| eeZmm0 | 2 | | |
| eeZmm0X | 2 | | |
| ee2mmF | 0 | | |
| eeZmmFX | 0 | | |
| ee2mmR | 0 | e^- | e^+ |
| eeZmmRX | 0 | | |
| ee2mmFFEEEE | 0 | e^- | e^+ |
| ee2mmRFEEEE | 0 | e^- | e^+ |
| ee2mmRREEEE | 0 | e^- | e^+ |
| ee2mmA | 0 | e^- | e^+ |
| eeZmmAX | 0 | | |
| ee2mmAA | 2 | | |
| ee2mmAFEE | 0 | | |
| ee2mmAREE | 0 | e^- | e^+ |
| ee2mmNFEE | 2 | e^- | e^+ |
| ee2nn0 | 0 | e^- | e^+ |
| ee2nnF | 0 | | |
| ee2nnR | 0 | e^- | e^+ |
| ee2nnS | 0 | e^- | e^+ |
| ee2nnSS | 0 | | |
| ee2nnCC | 0 | | |
| ee2nnRF | 0 | e^- | e^+ |
| ee2nnRR | 0 | e^- | e^+ |
| ee2ee0 | 0 | e^- | e^- |
| ee2eeA | 0 | | |
| ee2eeF | 0 | | |
| ee2eeR | 0 | e^- | e^- |
| ee2eeR125 | 0 | | |
| ee2eeR345 | 0 | | |
| ee2eeR35 | 0 | | |

| which_piece | P? | p_1 | p_2 |
|------------------|----|-------|-------|
| ee2eeR45 | 0 | | |
| ee2eeR345co | 0 | | |
| ee2eeR35co | 0 | | |
| ee2eeR45co | 0 | | |
| ee2eeRF | 0 | | |
| ee2eeRF125 | 0 | | |
| ee2eeRF345 | 0 | | |
| ee2eeRF35 | 0 | | |
| ee2eeRF45 | 0 | | |
| ee2eeRF345co | 0 | | |
| ee2eeRF35co | 0 | | |
| ee2eeRF45co | 0 | | |
| ee2eeRR | 0 | e^- | e^- |
| ee2eeRR15162526 | 0 | | |
| ee2eeRR35364546 | 0 | | |
| ee2eeRR35364546c | 0 | | |
| ee2eeFF | 0 | e^- | e^- |
| ee2eeAA | 0 | | |
| ee2eeAF | 0 | | |
| ee2eeAR | 0 | e^- | e^- |
| ee2eeAR125 | 0 | | |
| ee2eeAR345 | 0 | | |
| ee2eeAR345co | 0 | | |
| ee2eeNF | 0 | e^- | e^- |
| eb2eb0 | 0 | e^- | e^+ |
| eb2ebA | 0 | | |
| eb2ebF | 0 | | |
| eb2ebR | 0 | e^- | e^+ |
| eb2ebR125 | 0 | | |
| eb2ebR35 | 0 | | |
| eb2ebR45 | 0 | | |
| eb2ebR35co | 0 | | |
| eb2ebR45co | 0 | | |
| eb2ebFF | 0 | e^- | e^+ |
| eb2ebRF | 0 | e^- | e^+ |
| eb2ebRF125 | 0 | | |
| eb2ebRF35 | 0 | | |
| eb2ebRF45 | 0 | | |
| eb2ebRF35co | 0 | | |
| eb2ebRF45co | 0 | | |

Chapter 14

Available processes and `which_piece`

When running McMule, we recommend using the following `which_piece`

| Process | Order | (n)-particle | (n + 1)-particle | (n + 2)-particle |
|--|-------|--|---|---|
| $\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu$ | LO | m2enn0 | | |
| | NLO | m2ennF | m2ennR | |
| $\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu \gamma$ | NNL | m2ennFF, m2ennNF | m2ennRF | m2ennRR |
| | LO | m2eng0 | | |
| $\mu^- \rightarrow e^- \bar{\nu}_e \nu_\mu \gamma$ | NLO | m2engC, m2engV | m2engR | |
| | LO | m2ennee0 | | |
| $\mu^- \rightarrow e^- \bar{\nu}_e \nu_\tau e$ | NLO | m2enneeV, m2enneeA | m2enneeC, m2enneeR | |
| | LO | m2ej0 | | |
| $e^- j$ | NLO | m2ejF | m2ejR | |
| | LO | m2ejg0 | | |
| $e^- j \gamma$ | LO | m2ejg0 | | |
| | NLO | m2ejgF | m2ejgR | |
| $e^- \mu^- \rightarrow e^- \mu^-$ | LO | em2em0 | | |
| | NLO | em2emFEE, em2emFEM, em2emFMM, em2emA | em2emREE15, em2emREM, em2emRMM | em2emREE35 |
| | el. | em2emFFEEEE, em2emAA | em2emRFEEEE15, em2emRFEEEE35, em2emRFEEEEco, em2emAREE15, em2emAREE35 | em2emRREEEEE1516, em2emRREEEEE3536, em2emRREEEEc |
| | NNL | em2emAFEE, em2emNFEE | | |
| | full | em2emFFMIXDz, em2emFFMMMM, em2emAFEM, em2emNFEM, em2emNFMM | em2emRFMIXD15, em2emRFMIXD35, em2emRFMIXDco, em2emAREM, em2emARMM | em2emRRMIXD1516, em2emRRMIXD3536, em2emRRMIXDc, em2emRRMMMM |
| | NNL | | | |
| $\mu^- p \rightarrow \mu^- p$ | LO | mp2mp0 | | |
| | NLO | mp2mpF, mp2mpA | mp2mpR15, mp2mpR35 | |
| | NNL | mp2mpFF, mp2mpAF, mp2mpNF | mp2mpRF15, mp2mpRFco, mp2mpAR15, mp2mpAR35 | mp2mpRR1516, mp2mpRR3536 |
| | NNL | mp2mpAA, mp2mpAF | | |
| $e^- e^+ \rightarrow \mu^- \mu^+$ | LO | ee2mm0 | | |
| | NLO | ee2mmF, ee2mmA | ee2mmR | |
| | el. | ee2mmFFEEEE, ee2mmAA | ee2mmRFEEEE, ee2mmAREE | ee2mmRREEEEE |
| | NNL | ee2mmAFEE, ee2mmNFEE | | |
| | EW | eeZmm0 | | |
| | LO | eeZmm0 | | |
| | EW | eeZmmFX, eeZmmAX | eeZmmRX | |
| | NLO | | | |
| $e^- e^- \rightarrow e^- e^-$ | LO | ee2ee0 | | |
| | NLO | ee2eeF, ee2eeA | ee2eeR125, ee2eeR345 | |
| | NNL | ee2eeFF, ee2eeAA, ee2eeNF | ee2eeRF125, ee2eeRF345 | ee2eeRR15162526, ee2eeRR35364546 |
| | NNL | ee2eeAF | | |
| $e^- e^+ \rightarrow e^- e^+$ | LO | eb2eb0 | | |
| | NLO | eb2ebF, eb2ebA | eb2ebR125, eb2ebR45 | eb2ebR35 |
| | NNL | eb2ebFF | eb2ebRF125, eb2ebRF45 | eb2ebRR15162526, eb2ebRR3536, eb2ebRR4546 |
| | NNL | eb2ebAF | | |

Chapter 15

libmcmule reference guide

The following types, variables, and routines are unlikely to be needed by the typical user and are instead aimed at McMule's developers.

15.1 The particle framework

integer maxparticles

The maximal number of particles allowed

type mlm

Type fields

- **% momentum** (4) [*real(kind=prec)*] :: the momentum

type particle

Type fields

- **% momentum** (4) [*real(kind=prec)*] :: the momentum
- **% effcharge** [*integer*] :: the effective charge, corresponding to the +charge for incoming and -charge for outgoing particles.
- **% charge** [*integer*] :: the actual charge
- **% incoming** [*logical*] :: `.true.` for incoming particles
- **% lepcharge** [*integer*] :: the lepton family (1 for electrons, 2 for muons, 3 for taus), defaults to zero

type particles

Type fields

- **% vec** (maxparticles) [*type(particle)*] :: the constituent particles
- **% n** [*integer*] :: the number of particles actually used
- **% combo** [*character(len=1)*] :: the flavour combination used, allowed values are * (any combination), x (only mixed), e (only electronic), m (only muonic), t (only tauonic)

function make_mlm(*qq*)

Construct a *mlm*, i.e. a massless momentum

Parameters

qq (4) [*real(kind=prec),in*] :: the momentum

function part(*qq, charge, inc* [, *lepcharge*])

Construct a *particle*.

Parameters

- **qq** (4) [*real(kind=prec),in*] :: the momentum
- **charge** [*integer,in*] :: the charge of the particle
- **inc** [*integer,in*] :: +1 for incoming, -1 for outgoing

Options

lepcharge [*integer,1*] :: the lepton family number

function parts(*ps* [, *combo*])

Construct *particles* from a list of *particles*

Parameters

ps (*) [*type(particle),in*] :: a list of *particle*

Options

combo [*character(len=1)*] :: the flavour combination used, allowed values are * (any combination), x (only mixed), e (only electronic), m (only muonic), τ (only taonic)

function eik()

An interface to construct the eikonal factor. *eik* can be called with

- (**kg**, **pp**), using the type *particles*. The optional flavour combination **combo** restricts the emission to the desired set of fermion lines. If **combo** is set to **x**, all contributions but the self-eikonal are included.
- ({**q1**, **k1**}, **kg**, {**q2**, **k2**}), with an explicit call to the momenta of the {massive, massless} emitter, before (1) and after (2) the emission.

Parameters

- **pp** [*type(particles),in*] :: the fermions involved in the photon emission
- **qi** (4) [*real(kind=prec),in*] :: the momenta of the massive emitter
- **ki** [*type(mlm),in*] :: the momenta of the massless emitter
- **kg** [*type(mlm),in*] :: the momentum of the photon

Return

eik :: the eikonal factor

function ieik()

An interface to construct the integrated eikonal factor [28]. *ieik* can be called with

- (**xicut**, **epcmf**, **pp** [, **pole**]), using the type *particles*. The optional flavour combination **combo** restricts the emission to the desired set of fermion lines. If **combo** is set to **x**, all contributions but the self-eikonal are included.
- (**xicut**, **epcmf**, **q1**, **q2** [, **pole**]), with an explicit call to the momenta of the massive emitter, before (1) and after (2) the emission.

Parameters

- **xicut** [*real(kind=prec),in*] :: ξ_c (cf. Section *Running at NLO and beyond*)
- **epcmf** [*real(kind=prec),in*] :: square root of *scms*
- **pp** [*type(particles),in*] :: the fermions involved in the photon emission
- **qi** (4) [*real(kind=prec),in*] :: the momenta of the massive emitter

Options

pole [*real(kind=prec),out*] :: the singular part of the integrated eikonal, as a coefficient of $1/\epsilon$

Return

ieik :: the finite part of the integrated eikonal factor

function ntssoft(*pp, kk, pole*)

The (universal) soft contribution to the LBK theorem at 1 loop [9], i.e. the *NTS* soft function. The optional flavour combination for the *particles* *pp* restricts the emission to the desired set of fermion lines.¹

Parameters

- **pp** [*type(particles),in*] :: the fermions involved in the photon emission
- **kk** (4) [*real(kind=prec),in*] :: the momentum of the photon

Options

pole [*real(kind=prec),out*] :: the singular part of the *NTS* soft function, as a coefficient of $1/\epsilon$

Return

ieik :: the finite part of the *NTS* soft function

15.2 Matrix element interface

function partInterface(*q1, q2, q3, q4, q5, q6, q7*)

an abstract interface to construct *particles* for a given process.

Parameters

qi (4) [*real(kind=prec)*] :: the momenta

Return

partInterface [*particles*] :: the constructed particle string

15.3 Package-X function

Note: This section needs to be completed, link to [issue](#)

function DiscB()

function DiscB_cplx()

function ScalarC0IR6()

¹ The user is allowed to further split mixed contributions at NNLO, i.e. contributions with emissions connecting different fermion lines. This is achieved via the optional parameter *mx* of the auxiliary function *combonts*. The latter sets the desired flavour combination for *ntssoft*, and *mx=1* allows to choose among different mixed contributions. For example, for $\ell_1 \ell_2 \rightarrow \ell_1 \ell_2$ scattering, if a formal charge $Q_{1(2)}$ is assigned for each photon emission from $\ell_{1(2)}$, *ntssoft* will be able to distinguish among the contributions labelled by $Q_1^5 Q_2^3$, $Q_1^4 Q_2^4$ and $Q_1^3 Q_2^5$.

```
function ScalarC0IR6_cplx()
function ScalarC0()
function ScalarC0_cplx()
function ScalarD0IR16()
function ScalarD0IR16_cplx()
```

15.4 VP functions

Note: This section needs to be completed, link to [issue](#)

15.5 Phase spaces

McMule has implemented a number of phase routines that map from the hypercube to the physical momenta. Here is a list of currently used ones

subroutine PSD3(*ra, q1, m1, q2, m2, q3, m3, weight*)

Generic phase space routine for $1 \rightarrow 2$ decays

Parameters

- **ra** (2) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD4(*ra, q1, m1, q2, m2, q3, m3, q4, m4, weight*)

Generic phase space routine for $1 \rightarrow 3$ decays

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD4_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, weight*)

FKS phase space routine for $1 \rightarrow 3$ decays, requires $m_4 = 0$. Tuned for $\langle(p_2, q_4)$ and E_4

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD5(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, weight*)

Generic phase space routine for $1 \rightarrow 4$ decays

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD5_25(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, weight*)

Phase space routine for $1 \rightarrow 4$ decays, tuned for $\langle(p_2, q_5)$ and E_5 , collinear limit is $\text{ra}(2) \rightarrow 0$

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD5_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $1 \rightarrow 4$ decays, requires $m_5 = 0$. Tuned for $\langle(p_2, q_5)$ and E_5

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, weight*)

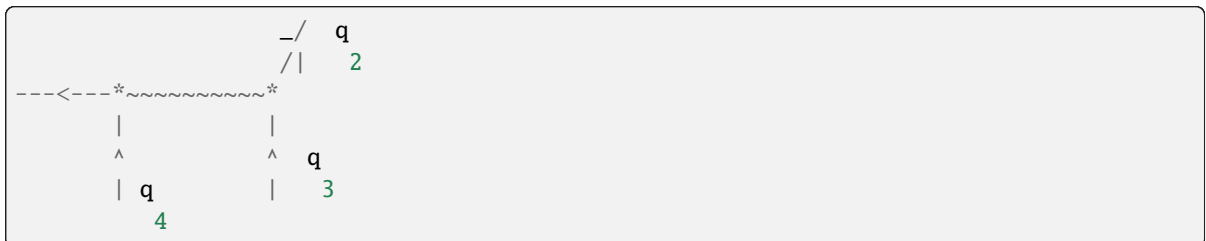
Generic phase space routine for $1 \rightarrow 5$ decays

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_23_24_34(*ra, q1, m1, q2, m2, q5, m5, q6, m6, q3, m3, q4, m4, weight*)

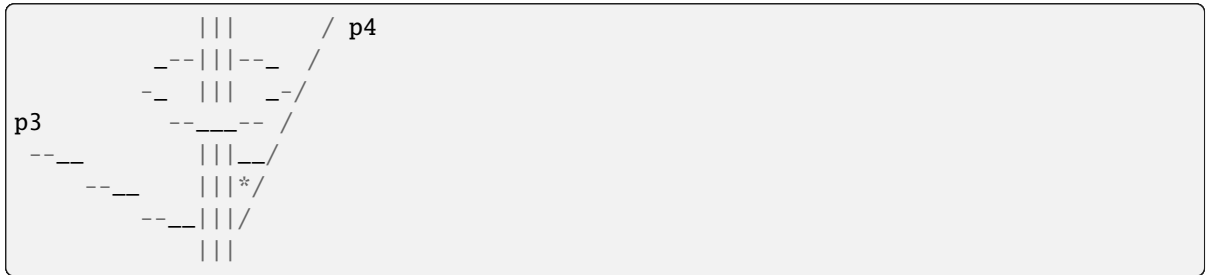
Phase space routine for $1 \rightarrow 5$ decays with FKS-ish tuning. This is designed for the decay $\mu^+ \rightarrow e^+ \nu \bar{\nu} e^+ e^-$. q_2 should be the unique particle (electron) and q_3 and q_4 are the identical particles (positrons):



The ‘spectator’ neutrinos are q_5 and q_6 . Start by generating p_2 and p_3 at an angle $* = \arccos(y_2)$:



Generate p_4 at an angle $\phi = \arccos(y_3)$ and rotating by an angle ϕ w.r.t. to p_3 :



Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_23_24_34_E56(*ra, q1, m1, q2, m2, q5, m5, q6, m6, q3, m3, q4, m4, weight*)

Phase space routine for $1 \rightarrow 5$ decays with FKS-ish tuning, similar to *PSD6_23_24_34()* but with special tuning on the $E_5 + E_6$ tail.

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, weight*)

FKS phase space routine for $1 \rightarrow 5$ decays, requires $m_6 = 0$. Tuned for $\langle(p_2, q_6)$ and E_6

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_25_26_m50_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, weight*)

FKS phase space routine for $1 \rightarrow 5$ decays, requires $m_5 = m_6 = 0$. Tuned for $\langle(p_2, q_{5,6})$ and $E_{5,6}$

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $1 \rightarrow 5$ decays, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(p_2, q_{5,6})$ and $E_{5,6}$

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD7(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, q7, m7, weight*)

Generic phase space routine for $1 \rightarrow 6$ decays

Parameters

- **ra** (14) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD7_27_37_47_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, q7, weight*)

FKS phase space routine for $1 \rightarrow 6$ decays, tuned for $\sphericalangle(p_{2,3,4}, q_7)$ and E_7

Parameters

- **ra** (14) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD7_27_37_47_E56_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, q7, weight*)

FKS phase space routine for $1 \rightarrow 6$ decays, tuned for $\sphericalangle(p_{2,3,4}, q_7)$ and E_7 and tuned for the $E_5 + E_6$ tail

Parameters

- **ra** (14) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX2(*ra, q1, m1, q2, m2, q3, m3, q4, m4, weight*)

Generic phase space routine for $2 \rightarrow 2$ cross sections

Parameters

- **ra** (2) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta

- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for 2 → 3 cross sections, requires $m_5 = 0$. Tuned for *ISR* and not *FSR*

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_35_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight* [, *sol*])

FKS phase space routine for 2 → 3 cross sections, requires $m_5 = 0$. Tuned for $\sphericalangle(q_3, q_5)$ and E_5

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

Options

sol [*integer,in*] :: which solution to pick

subroutine PSX4(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, weight*)

Generic phase space routine for 2 → 4 cross sections

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for 2 → 4 cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not *FSR*

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_35_36_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight* [, *sol*])

Double-*FKS* phase space routine for 2 → 4 cross sections, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(q_3, q_{5,6})$ and $E_{5,6}$

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers

- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

Options

sol [*integer,in*] :: which solution to pick

subroutine PSD6_P_25_26_m50_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, weight*)

FKS phase space routine for $1 \rightarrow 5$ decays, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(p_2, q_{5,6})$ and $E_{5,6}$ Partitioning of *PSD6_25_26_m50_FKS()* with $s_{26} < s_{25}$.

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD6_26_2x5(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, weight*)

Phase space routine for $1 \rightarrow 5$ decays with FKS-ish tuning. Modification of *PSD6_23_24_34()* with $2 \leftrightarrow 5$. This is designed for the decay $\mu^+ \rightarrow e^+ \nu \bar{\nu} e^+ e^-$.

Parameters

- **ra** (11) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSD7_27_37_47_2x5_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, m5, q6, m6, q7, weight*)

FKS phase space routine for $1 \rightarrow 6$ decays, tuned for $\sphericalangle(p_{2,3,4}, q_7)$ and E_7 Modification of *PSD7_27_37_47_FKS()* with $2 \leftrightarrow 5$.

Parameters

- **ra** (14) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_P_15_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $2 \rightarrow 3$ cross sections, requires $m_5 = 0$. Tuned for *ISR* and not *FSR*. Partitioning of *PSX3_FKS()* with $s_{15} < s_{35}$.

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_P13_35_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $2 \rightarrow 3$ cross sections, requires $m_5 = 0$. Tuned for $\triangleleft(q_3, q_5)$ and E_5 Partitioning of *PSX3_35_FKS()* with $s_{15} > s_{35}$.

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_coP13_35_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

The corner piece to *PSX3_P13_35_FKS()*

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_P_15_25_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $2 \rightarrow 3$ cross sections, requires $m_5 = 0$. Tuned for *ISR* and not *FSR*. Partitioning of *PSX3_FKS()* with $\min(s_{15}, s_{25}) < \min(s_{35}, s_{45})$.

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_P_35_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $2 \rightarrow 3$ cross sections, requires $m_5 = 0$. Tuned for $\triangleleft(q_3, q_5)$ and E_5 Partitioning of *PSX3_35_FKS()* with $s_{35} < \min(s_{15}, s_{25}, s_{45})$.

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_P_45_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

FKS phase space routine for $2 \rightarrow 3$ cross sections, requires $m_5 = 0$. Tuned for $\triangleleft(q_3, q_5)$ and E_5 Partitioning of *PSX3_35_FKS()* with $s_{45} < \min(s_{15}, s_{25}, s_{35})$ and $3 \leftrightarrow 4$

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses

- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_coP_35_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

The corner piece to [PSX3_P_35_FKS\(\)](#)

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX3_coP_45_FKS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, weight*)

The corner piece to [PSX3_P_45_FKS\(\)](#)

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_P_15_16_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $2 \rightarrow 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not :term`FSR` Partitioning of [PSX4_FKSS\(\)](#) with $\min(s_{15}, s_{16}) < \min(s_{35}, s_{36})$.

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_P_35_36_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $2 \rightarrow 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\langle(q_3, q_{5,6})$ and $E_{5,6}$ Partitioning of [PSX4_35_36_FKSS\(\)](#) with $\min(s_{15}, s_{36}) < \min(s_{15}, s_{25}, s_{45}, s_{16}, s_{26}, s_{46})$.

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_coP_35_36_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

The corner piece to [PSX4_P_35_36_FKSS\(\)](#)

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_P13_35_36_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $2 \rightarrow 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\triangleleft(q_3, q_{5,6})$ and $E_{5,6}$ Partitioning of *PSX4_35_36_FKSS*() with $\min(s_{15}, s_{16}) > \min(s_{35}, s_{36})$.

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_coP13_35_36_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

The corner piece to *PSX4_P13_35_36_FKSS*()

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_P_15_16_25_26_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $2 \rightarrow 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not :term`FSR` Partitioning of *PSX4_FKSS*() with $\min(s_{15}, s_{16}, s_{25}, s_{26}) < \min(s_{35}, s_{36}, s_{54}, s_{46})$.

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_P_45_46_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

Double-*FKS* phase space routine for $2 \rightarrow 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\triangleleft(q_3, q_{5,6})$ and $E_{5,6}$ Partitioning of *PSX4_35_36_FKSS*() with $\min(s_{45}, s_{46}) > \min(s_{15}, s_{25}, s_{35}, s_{16}, s_{26}, s_{36})$.

Parameters

- **ra** (8) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

subroutine PSX4_coP_45_46_FKSS(*ra, q1, m1, q2, m2, q3, m3, q4, m4, q5, q6, weight*)

The corner piece to *PSX4_P_45_46_FKSS*()

Parameters

- **ra** (5) [*real(kind=prec),in*] :: the random numbers
- **qi** (4) [*real(kind=prec),out*] :: the momenta
- **mi** [*real(kind=prec),in*] :: the masses
- **weight** [*real(kind=prec),out*] :: the Jacobian

Chapter 16

pymule reference guide

This section describes all functions and classes in pymule. Most users will not have to view this.

16.1 Working with errors

```
pymule.errortools.addplots(a: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, b: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, sa: float = 1.0, sb: float = 1.0) → <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

adds or subtracts two plots

Parameters

- **a** – Nx3 numpy matrix; the first plot
- **b** – Nx3 numpy matrix; the second plot
- **sa** – float, optional; the coefficient of the first plot
- **sb** – float, optional; the coefficient of the second plot

Returns

a Nx3 numpy matrix with $s_a \cdot a + s_b \cdot b$

Note: a and b must share x values, otherwise entries are dropped

Example

subtract two plots a and b

```
>>> addplots(a, b, sb=-1)
```

Example

Given the LO plots `thetaLO` and the NLO corrections `thetadNLO`, we calculate the K factor as either

```
>>> thetaNLO = addplots(thetaLO, thetadNLO)
```

```
pymule.errorertools.chisq(values: ~typing.List[~typing.List[float] | <MagicMock
                        name='mock.typing.NDArray.__getitem__()' id='140279735182480'>] |
                        <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>])
                        → float
```

calculates the χ^2 /d.o.f. of numbers

Parameters

value – Nx2 numpy matrix or list of lists; the values as `[[y1, dy1], [y2, dy2], ...]`

Returns

float; the χ^2 /d.o.f. = $\frac{1}{n} \sum_{n=1}^n \left(\frac{y_i - \bar{y}}{\delta y_i}\right)^2$ with the average value \bar{y}

Example

a good example

```
>>> chisq([[20.0, 0.8],
...       [21.6, 0.9],
...       [18.7, 1.2]])
1.3348808062205872
```

and a bad example

```
>>> chisq([[16.2, 0.8],
...       [22.9, 0.9],
...       [8.81, 1.2]])
30.173852184366673
```

```
pymule.errorertools.combineNplots(func: ~typing.Callable[<MagicMock
                        name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
                        <MagicMock name='mock.typing.NDArray.__getitem__()'
                        id='140279735182480'>], <MagicMock
                        name='mock.typing.NDArray.__getitem__()' id='140279735182480'>],
                        plots: ~typing.List[<MagicMock
                        name='mock.typing.NDArray.__getitem__()' id='140279735182480'>] |
                        <MagicMock name='mock.typing.NDArray.__getitem__()'
                        id='140279735182480'>]) → <MagicMock
                        name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

combines a list of plots using a function

Parameters

- **func** – callable with two arguments; the function to combine the plots
- **plots** – list of Nx3 numpy matrices

Returns

a Nx3 numpy matrix $f(p_0, f(p_1, f(p_2, \dots)))$

```
pymule.errortools.combineplots(a: <MagicMock name='mock.typing.NDArray.__getitem__()'
                               id='140279735182480'>, b: <MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
                               yfunc: ~typing.Callable[[<MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
                               <MagicMock name='mock.typing.NDArray.__getitem__()'
                               id='140279735182480'>], <MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>],
                               efunc: ~typing.Callable[[<MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
                               <MagicMock name='mock.typing.NDArray.__getitem__()'
                               id='140279735182480'>, <MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
                               <MagicMock name='mock.typing.NDArray.__getitem__()'
                               id='140279735182480'>], <MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>],
                               float = 1e-08) → <MagicMock
                               name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

combines two plots using functions for the value and the error

Parameters

- **a** – Nx3 numpy matrix; the first plot
- **b** – Nx3 numpy matrix; the second plot
- **yfunc** – callable; a function to calculate the value `yfunc(a, b)`
- **efunc** – callable; a function to calculate the squared error `efunc(a, da**2 b, db**2)`
- **tol** – float, optional; the difference at which values are considered equal

Returns

Nx3 numpy matrix; the combined plot `np.array([[x1, yfunc(..), sqrt(efunc(..))], ..])`

Note: a and b must share x values, up to the tolerance `tol`, otherwise values may be dropped

Example

Add two plots A and B

```
>>> combineplots(A, B,
...             lambda a, b: a+b,
...             lambda a, da2, b, db2: da2 + db2))
```

Calculate a K factor

```
>>> combineplots(dnlo, lo,
...             lambda a, b: 1 + a/b,
...             lambda a, da2, b, db2: db2 * a**2 / b**4 + da2 / b**2)
```

```
pymule.errortools.dividenumbers(a: ~typing.List[float] | <MagicMock
                                name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, b:
                                ~typing.List[float] | <MagicMock
                                name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) →
                                __getitem__() id='140279735182480'>
```

divides numbers

Parameters

- **a** – list of floats; the numerator with error [a, da]
- **b** – list of floats; the denominator with error [b, db]

Returns

the result of the division a/b [y, dy]

Example

Divide $(2.3 \pm 0.1)/(45 \pm 0.01)$

```
>>> dividenumbers([2.3, 0.1], [45., 0.01])
array([0.05111111, 0.00222225])
```

```
pymule.errortools.divideplots(a: <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>, b: <MagicMock
name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, offset:
float = 0.0) → <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>
```

divides two plots

Parameters

- **a** – Nx3 numpy matrix; the numerator plot
- **b** – Nx3 numpy matrix; the denominator plot
- **offset** – float, optional; shifts the result

Returns

a Nx3 numpy matrix with $a/b + offset$

Note: a and b must share x values, otherwise entries are dropped

Example

Given the LO plots `thetaLO` and the NLO corrections `thetadNLO`, we calculate the K factor as either

```
>>> thetaNLO = addplots(thetaLO, thetadNLO)
>>> thetaK = divideplots(thetaNLO, thetaLO)
>>> thetaK = divideplots(thetadNLO, thetaLO, offset=+1.)
```

```
pymule.errortools.integratehistogram(hist: <MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>) → float
```

integrates a histogram

Parameters

hist – Nx3 numpy matrix; the histogram to integrate $d\sigma/dx$ as `np.array([[x1, y1, e1], [x2, y2, e2], ...])`

Returns

float; the integrated histogram $\int d\sigma/dx dx$ without error estimate

Example

Integrate a histogram

```
>>> hist
array([[          -inf,  0.00000000e+00,  0.00000000e+00],
       [5.00000000e-02,  4.77330751e+01,  2.26798977e-01],
       [1.50000000e-01,  7.40641192e+01,  2.36498021e-01],
       ...,
       [8.85000000e+00,  1.67513948e+00,  1.16218116e-01],
       [8.95000000e+00,  0.00000000e+00,  0.00000000e+00],
       [          inf,  0.00000000e+00,  0.00000000e+00]])
>>> integratehistogram(hist)
4188.519369660588
```

```
pymule.errortools.mergetbins(p: <MagicMock name='mock.typing.NDArray.__getitem__()'
                             id='140279735182480'>, n: int) → <MagicMock
                             name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

merges n adjacent bins into one larger bin, reducing the uncertainty.

Parameters

- **p** – Nx3 numpy matrix; the plot
- **n** – int; how many bins to merge

Returns

a (N/n)x3 numpy matrix

Note: This process loses $\lfloor \text{len}(p) / n \rfloor$ bins at the end of the histogram

Example

merge five bins

```
>>> len(p)
200
>>> len(mergetbins(p, 5))
40
```

Bins may be lost

```
>>> len(p)
203
>>> len(mergetbins(p, 5))
40
```

```
pymule.errortools.mergenumbers(values: ~typing.List[~typing.List[float]] | <MagicMock
                                name='mock.typing.NDArray.__getitem__()' id='140279735182480'> |
                                <MagicMock name='mock.typing.NDArray.__getitem__()'
                                id='140279735182480'>) → __getitem__()' id='140279735182480'>
```

statistically combines values with uncertainties

Parameters

values – Nx2 numpy matrix or list of lists; the values as $[[y_1, dy_1], [y_2, dy_2], \dots]$

Returns

either answer as numpy array [y, dy] or tuple of χ^2 and answer

Example

If quiet is not specified this will print the χ^2

```
>>> mergenumbers([[20.0, 0.8],
...               [21.6, 0.9],
...               [18.7, 1.2]])
1.3348808062205872
array([20.30718232,  0.53517179])
```

Otherwise, it will return it

```
>>> mergenumbers([[20.0, 0.8],
...               [21.6, 0.9],
...               [18.7, 1.2]], quiet=True)
(1.3348808062205872, array([20.30718232,  0.53517179]))
```

```
pymule.errortools.mergeplots(ps: ~typing.List[<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

statistically combines a list of plots

Parameters

ps – list of Nx3 numpy matrices; the plots to combine as [np.array([[x1, y1, e1], [x2, y2, e2], ...]), ...]

Returns

a Nx3 numpy matrix

Example

Load a number of vegas files and merge them

```
>>> data = [
...     importvegas(i)['thetae']
...     for i in glob.glob('out/em2em0*')
... ]
>>> mergeplots(data)
```

```
pymule.errortools.plusnumbers(*args: ~typing.List[float] | <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → __getitem__()' id='140279735182480'>
```

adds numbers and errors

Parameters

yi – list of floats; a number with error [yi, dyi]

Returns

the result of the addition [y, dy]

Example

Adding $(10 \pm 1) + (20 \pm 0.5) + (-5 \pm 2)$


```
>>> plusnumbers([10, 1], [20, 0.5], [-5, 2])
array([25.          ,  2.29128785])
```

`pymule.errortools.printnumber`(*x*: *~typing.List[float]* | *<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>*, *prec*: *int = 0*) → *str*

returns a string representation of a number with uncertainties to one significant digit

Parameters

- **x** – a list with two floats; the number as [**x**, **dx**]
- **prec** – int, optional; number of extra significant figures

Returns

str; the formatted string

Example

printing 53.2 ± 0.1 to one significant figure

```
>>> printnumber([53.2, 0.1])
"53.2(1)"
```

`pymule.errortools.scaleplot`(*a*: *<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>*, *sx*: *float*, *sy*: *float | None = None*) → *<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>*

rescales a plot such that the integrated plot remains unchanged, i.e. rescale $x \rightarrow x/s$ and $y \rightarrow y \cdot s$. This is useful to, for example, change units.

Parameters

- **a** – Nx3 numpy matrix; the plot
- **sx** – float; the inverse scale factor for the x direction
- **sy** – float, optional; if present, sy will be used for the y direction instead of sx

Returns

a Nx3 numpy matrix

Example

rescaling units from rad to mrad

```
>>> scaleplot(data, 1e-3)
```

`pymule.errortools.timesnumbers`(*a*: *~typing.List[float]* | *<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>*, *b*: *~typing.List[float]* | *<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>*) → *__getitem__()' id='140279735182480'>*

multiplies numbers

Parameters

- **a** – list of floats; the first factor with error [**a**, **da**]
- **b** – list of floats; the second factor with error [**b**, **db**]

Returns

the result of the multiplication $a*b$ [**y**, **dy**]

Example

Divide $(0.5 \pm 0.02) * (45 \pm 0.01)$

```
>>> timesnumbers([0.5,0.02], [45, 0.1])
array([22.5      ,  0.90138782])
```

16.2 Working with abstract records

class pymule.record.Chi2

An object to represent a χ^2 value.

This can be created using `Chi2.from_single()` or `Chi2.from_merge()`. This class supports addition and float.

Attribute type

an enum to indicate the type of χ^2 this is

Attribute value

the value of χ^2 for single and mergers

Attribute children

the children `Chi2` objects for sums and mergers

class Type(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

The χ^2 type enum

Attribute NONE

empty entry

Attribute SINGLE

a single entry

Attribute SUM

a sum of entries

Attribute MERGE

a statistical merge of entries

classmethod from_merge(chi: float, *s: Chi2) → Chi2

combines multiple χ^2 into a single one

Parameters

- **chi** – the value of χ^2 of the merge, must be positive
- **s** – a sequence of `Chi2` objects

Returns

a new `Chi2` object

Return type

`Chi2`

Raises

- **ValueError** – the parameter `chi` is not positive or no `Chi2` is be provided in `s`
- **TypeError** – the parameter `chi` is not a number or the objects provided are not `Chi2`

Example

Provide a merged set

```
>>> c1 = Chi2.from_single(1.02)
>>> c2 = Chi2.from_single(1.41)
>>> c3 = Chi2.from_single(2.02)
>>> tot = Chi2.from_merge(1.3, c1, c2, c3)
```

classmethod `from_single(v: float) → Chi2`

creates a single *Chi2* object

Parameters

v – the value of χ^2 , must be positive

Returns

a new *Chi2* object

Return type

Chi2

Raises

- **ValueError** – the parameter **v** is not positive
- **TypeError** – the parameter **v** is not a number

Example

Create a $\chi^2 = 1.02$

```
>>> Chi2.from_single(1.02)
<Chi^2: single 1.02>
```

class `pymule.record.Record`

A McMule record, either corresponding to a single file or multiple.

This should usually not be instantiated directly but using either *VegasRecord* or *XiRecord*. This supports addition, subtraction, and scalar multiplication.

Properties can be accessed as attributes or as keys. Histograms are accessed as keys.

Example

Assuming that `r0`, `rV`, and `rR` contains records of leading order, virtual, and real corrections (as obtained using `pymule.vegas.FileRecord.read()` or `mergefks()`)

```
>>> total = r0 + alpha * rV + alpha * rR
```

Add the VP contributions from `rA` to this by

```
>>> total += alpha * rA
```

Print the cross section and values

```
>>> print(total.value)
[1.00646123e-04 1.06589201e-08]
```

```
>>> total.histograms
dict_keys(['thetae', 'thetam'])
```

```
>>> total.chi2a
<Chi^2: single 1.0530713790136834>
```

```
>>> total['thetae']
array([[ -inf,  0.00000000e+00,  0.00000000e+00],
       [ 1.75000000e-03, -1.92701245e-03,  1.27052491e-05],
       [ 5.25000000e-03,  1.57686769e-03,  2.08601107e-05],
       [ 8.75000000e-03,  3.10660221e-03,  2.45339189e-05],
       [ 1.22500000e-02,  3.90154737e-03,  2.68166764e-05],
       [ 1.57500000e-02,  5.04369602e-03,  2.64601744e-05],
       [ 1.92500000e-02,  7.87371009e-03,  2.37634926e-05],
       [ 2.27500000e-02,  8.54989271e-03,  1.14554010e-05],
       [ 2.62500000e-02,  4.47774416e-04,  1.48451350e-06],
       [          inf,  7.29295875e-08,  1.24592720e-09]])
```

property chi2a: *Chi2*

the χ^2 estimate of the integrator or combination using *Chi2*

property histograms: *list[str]*

a list of histograms included in this record

classmethod merge(*s: *Record*) → *Record*

statistically merges a set of runs, combining cross sections, histograms, and run-time information.

Parameters

s – a sequence of *Record* objects (or subclasses like *VegasRecord*)

Returns

a new record containing cross sections, distributions, and run-time information.

Return type

Record

Raises

- **ValueError** – no record is provided
- **TypeError** – not all records are *Record* objects

Note: The result only contains histograms that are in all records. Entries that appear less often are dropped

Example

Assuming that r1, r2, and r3 are statistically independent records that should be merged

```
>>> r = Record.merge(r1, r2, r3)
```

alternatively

```
>>> records = [r1, r2, r3]
>>> r = Record.merge(*records)
```

property time: *float*

the run time required by the record in seconds, either for a single job or combinations

property value: *__getitem__()* 'id='140279735182480'>

the best estimate for the cross section and its error as `np.array([y, e])`

16.3 Working with vegas records

class `pymule.vegas.BareRecordV4`

A subclass of `FileRecord` for v4 bare files.

class `pymule.vegas.FileRecord`

A subclass of `Record` to track McMule output files.

Additionally to the attributes of `Record` this contains all relevant properties of a McMule run

Attribute version

the file version

Attribute inttype

the integer type used in McMule in the format for `struct`

Attribute sha

the sha hash provided by the file

Attribute sampler

the sampler used, currently either `vegas` or `foam`

Attribute iteration

the current iteration of McMule

Attribute randy

the current random seed

Attribute msg

the integration message

Example

Load a record stored at `/path/to/record.mcmule` and print its value

```
>>> rec = FileRecord.read("/path/to/record.mcmule")
>>> print(rec.value)
[1.00646123e-04 1.06589201e-08]
```

classmethod `read(filename: str = "", fp: IO[bytes] | None = None) → FileRecord`

opens a McMule file, infers the correct file version, and creates a subclass of `FileRecord`

Parameters

- **filename** – file name to open, optional
- **fp** – file pointer to read, optional

Returns

a recording corresponding to the file

Return type

subclass of `FileRecord`

Note: Either **filename** xor **fp** need to be specified

Note: If less than two iterations have been completed, no histograms will be returned

Example

Load a file for the muon decay

```
>>> FileRecord.read('m2ennRR_mu-e_S0000068031X0.50000D0.50000_
↳ITMX080x150M_O12.vegas')
<Record m2ennRR_mu-e_S0000068031X0.50000D0.50000_ITMX080x150M_O12.
↳vegas xsec = 0.00010065(1), 12 histograms>
```

class pymule.vegas.FoamRecordV4

A subclass of *FileRecord* to track Foam files. Additionally to the attributes of *Record* and *FileRecord* this contains relevant information for foam

Attribute active

mask of active cells

Attribute parents

list of parents of cells or -1 for the root cell

Attribute offset

offset of the bottom right corner of each cell

Attribute length

size of each cell

Attribute probability

probability that a cell is selected for sampling

Attribute nbins

number of bins in used in cell division

write(filename: str = "", fp: IO[bytes] | None = None) → None

writes this record to a file or file pointer

Parameters

- **filename** – file name to open, optional
- **fp** – file pointer to write to, optional

Note: Either **filename** xor **fp** need to be specified

Example

save a random run to disk

```
>>> rec.write("out.vegas")
```

class pymule.vegas.VegasRecord

A subclass of *FileRecord* to track Vegas files. Additionally to the attributes of *Record* and *FileRecord* this contains relevant information for vegas

Attribute ndo

number of subdivisions on an axis

Attribute xi

the vegas grid, location of the i -th division on the j -th axis, normalised to lie between 0 and 1.

class pymule.vegas.VegasRecordV1

A subclass of *VegasRecord* for v1 Vegas files

class pymule.vegas.VegasRecordV2

A subclass of *VegasRecord* for v2 Vegas files

class pymule.vegas.VegasRecordV3

A subclass of *VegasRecord* for v3 Vegas files

class pymule.vegas.VegasRecordV4

A subclass of *VegasRecord* for v4 Vegas files. Since this is the most recent version it supports writing as well as reading

write(filename: str = "", fp: IO[bytes] | None = None) → None

writes this record to a file or file pointer

Parameters

- **filename** – file name to open, optional
- **fp** – file pointer to write to, optional

Note: Either **filename** xor **fp** need to be specified

Example

save a random run to disk

```
>>> rec.write("out.vegas")
```

pymule.vegas.**guess_version**(fp: IO[bytes], inttype: str = 'i') → tuple[int, str]

infers version of the vegas file using either the version string (since v3) or the file length (v1 and v2).

Parameters

- **fp** – file pointer
- **inttype** – either 'i' or 'q', optional; the integer type to use for v1 or v2, inferred otherwise

Returns

tuple of version number and integer type

pymule.vegas.**read_record**(fp: IO[bytes], typ: str) → int | float | bytes | List[int | float]

reads a single FORTRAN record

Parameters

- **fp** – file pointer
- **typ** – the type to read, everything that **struct** understands. Examples are
 - **i**: 32 bit signed integer (standard integer in Fortran)
 - **I**: 32 bit unsigned integer
 - **q**: 64 bit signed integer (**integer*8** in Fortran)
 - **c**: 8 bit character (character in Fortran)
 - **d**: **64 bit double precision (real(kind=prec) in Fortran,**
with default prec)

Records are data structures that are build as follows:

- 4 byte header: length of the record as a 32 bit unsigned integer, called l1
- body of length l1
- 4 byte footer: a repetition of l1 to make sure the record is properly closed.

Records can contain multiple variables.

`pymule.vegas.write_record(fp: IO[bytes], typ: str, content: int | float | bytes | List[int | float] | str) → None`
writes a single FORTRAN record

Parameters

- **fp** – file pointer
- **typ** – the type to read, everything that `struct` understands. Examples are
 - `i`: 32 bit signed integer (standard integer in Fortran)
 - `I`: 32 bit unsigned integer
 - `q`: 64 bit signed integer (`integer*8` in Fortran)
 - `c`: 8 bit charater (character in Fortran)
 - **d**: **64 bit double precision (`real(kind=prec)` in Fortran, with default `prec`)**
- **content** – scalar, list, `str` or `bytes`; the data to write

Records are data structures that are build as follows:

- 4 byte header: length of the record as a 32 bit unsigned integer, called l1
- body of length l1
- 4 byte footer: a repetition of l1 to make sure the record is properly closed.

Records can contain multiple variables.

16.4 Working with records of data

`class pymule.loader.FolderLoader(path: str)`

A *Loader* than can act on folders using `os.listdir`

`classmethod can_open(path: str) → bool`

checks if this loader can open a given path

Parameters

path – the path to check

Returns

a bool to indicate whether this loader can open the path

`list()` → list[str]

lists all files known to this loader

Returns

an iterable of all files known

open1(*file: str*) → *FileRecord*

opens a file

Parameters

file – the file path to be opened

Returns

the loaded record

Return type

FileRecord

class pymule.loader.**GitLabLoader**(*path: str*)

A *Loader* than can read tar balls from GitLab

classmethod **can_open**(*path: str*) → bool

checks if this loader can open a given path

Parameters

path – the path to check

Returns

a bool to indicate whether this loader can open the path

list() → list[str]

lists all files known to this loader

Returns

an iterable of all files known

open1(*file: str*) → *FileRecord*

opens a file

Parameters

file – the file path to be opened

Returns

the loaded record

Return type

FileRecord

class pymule.loader.**Loader**(*path: str*)

An abstract class to load vegas files from various sources.

Implementations need to provide *Loader.can_open()*, *Loader.list()*, and *Loader.open1()*

abstract classmethod **can_open**(*path: str*) → bool

checks if this loader can open a given path

Parameters

path – the path to check

Returns

a bool to indicate whether this loader can open the path

static chain(*loaders: list[Loader], pattern: Pattern[str]*) → Generator[tuple[Any, *FileRecord*], None, None]

chains a list of loaders together using their *Loader.open()*

Parameters

- **loaders** – a list of loaders to chain

- **pattern** – the pattern to match in each loader

Returns

a generator of *FileRecord* objects that match the pattern

Example

```
>>> for rec in Loader.chain([11, 12, 13], pattern):  
...     print(rec)
```

abstract list() → Generator[str, None, None] | List[str]

lists all files known to this loader

Returns

an iterable of all files known

open(pattern: Pattern[str]) → Generator[tuple[Any, *FileRecord*], None, None]

checks all files known to this loader and opens all that match the provided pattern

Parameters

pattern – a regular expression to match the filename

Returns

a generator of *FileRecord* objects that match the pattern

Example

```
>>> for rec in loader.open(pattern):  
...     print(rec)
```

abstract open1(file: str) → *FileRecord*

opens a file

Parameters

file – the file path to be opened

Returns

the loaded record

Return type

FileRecord

class pymule.loader.Options

Options used for loading and merging records

Attribute merge

a dict of histograms {'name': n} to merge n bins in the histogram name. defaults to no merging

Attribute sanitycheck

callable; a function that, given a *FileRecord*, whether to include the file in the output (return True) or to skip (return False).

Attribute flavour

str; the flavour to load, defaults to everything Initialised to everything, i.e. .*.

Attribute obs

str; the observable to load (the bit after the 0), defaults to everything Initialised to everything, i.e. ''.

Attribute folderp

str; a regular expression to match directory structures of a tar file, defaults to everything Initialised to everything, i.e. .*.

Attribute KNOWN_LOADERS

a list of *Loader* subclasses to try

add_folder(*folder: str*) → None

adds the first *Loader* that can open a folder or file

Parameters

folder – the path to folder or file try add

Raises

NotImplemented – no loader can open this object

reset() → None

resets this loader to the default state

class pymule.loader.**TarLoader**(*path: str*)

A *Loader* than can read tar balls, potentially compressed using BZip2 or GZip

classmethod **can_open**(*path: str*) → bool

checks if this loader can open a given path

Parameters

path – the path to check

Returns

a bool to indicate whether this loader can open the path

list() → list[str]

lists all files known to this loader

Returns

an iterable of all files known

open1(*file: str*) → *FileRecord*

opens a file

Parameters

file – the file path to be opened

Returns

the loaded record

Return type

FileRecord

class pymule.loader.**XiRecord**

A collection of McMule records, keyed by seed or ξ_c

This should be created using *XiRecord.load()* or *sigma()* which loads all matching records into this object. Default options can be changed using the options attribute. Records can then be merged or modified. This supports addition, subtraction, and scalar multiplication.

Specific values of ξ_c can be accessed as keys.

Attribute options

An instance of *Options* with the defaults to use. This can be modified

Example

Load all records for piece em2emFEE in folder out

```
>>> XiRecord.options.folder = "out/"
>>> rf = XiRecord.load('em2emFMM')
>>> rf
<XiRecord set em2emFMM, xic = [0.3, 0.5, 1.0, 0.1], 12 histograms>
>>> rr = XiRecord.load('em2emRMM', obs="0")
>>> tot = rf + rr
>>> tot
<XiRecord set em2emFMM + em2emRMM, xic = [0.3, 0.5, 1.0, 0.1], 12_
↳ histograms>
>>> tot.xic
{(0.1, 0.1), (0.3, 0.3), (0.5, 0.5), (1.0, 1.0)}
>>> tot[1.,1.]
<Record em2emFMM xsec = -0.000232279(3), 12 histograms>
```

as_record() → *Record*

returns the record if there is only one ξ_c record

returns

the record

rtype

Record

raises IndexError

data is not FKS merged

Example

Open a leading order calculation

```
>>> XiRecord.load('em2em0')
<XiRecord set em2em0, xic = [1.0], 12 histograms>
>>> XiRecord.load('em2em0').as_record()
<Record em2em0 xsec = 0.0058559441(2), 12 histograms>
```

property histograms: **set[str]**

the histograms present in in all records. Entries that appear less often are dropped

classmethod load(*piece: str, folder: str | None = None, **kwargs: Any*) → *XiRecord*

imports all matching records files

Parameters

- **piece** – str; which_piece to load
- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files
- **merge** – a dict of histograms {'name': n} to merge n bins in the histogram name. defaults to no merging
- **sanitycheck** – callable; a function that, given a *FileRecord*, whether to include the file in the output (return True) or to skip (return False).
- **flavour** – str; the flavour to load, defaults to everything Initialised to everything, i.e. `.*`.

- **obs** – str; the observable to load (the bit after the 0), defaults to everything Initialised to everything, i.e. ''.
- **folderp** – str; a regular expression to match directory structures of a tar file, defaults to everything Initialised to everything, i.e. .*.

Returns

an in-memory representation of the records

Return type

XiRecord

Note: Use `setup()` or modify `XiRecord.options` to set the defaults. Arguments provided here override the defaults

Example

```
>>> XiRecord.load('em2emFMM')
<XiRecord set em2emFMM, xic = [0.3, 0.5, 1.0, 0.1], 12 histograms>
>>> XiRecord.load('em2emRMM', obs="0")
<XiRecord set em2emRMM, xic = [0.3, 0.5, 1.0, 0.1], 12 histograms>
```

property merged_seeds: dict[tuple[float, float], *Record*]

the random-seed-merged *Record* for each ξ_c combination

Example

```
>>> rr
<XiRecord set em2emRMM, xic = [0.3, 0.5, 1.0, 0.1], 12 histograms>
>>> rr.merged_seeds
{(0.3, 0.3): <Record em2emRMM xsec = 0.00037124(3), 12 histograms>,
 (0.5, 0.5): <Record em2emRMM xsec = 0.00002654(2), 12 histograms>,
 (1.0, 1.0): <Record em2emRMM xsec = -0.0004411(3), 12 histograms>,
 (0.1, 0.1): <Record em2emRMM xsec = 0.00111258(3), 12 histograms>}
```

property seeds: set[int]

a set of seeds in this record

Example

```
>>> rr.seeds
{96012, 96018, 96019}
```

property time: float

the run time required by the record in seconds, either for a single job or combinations

property xic: set[tuple[float, float]]

a set of pairs of ξ_c in this record

Example

```
>>> rr.xic
{(0.1, 0.1), (0.3, 0.3), (0.5, 0.5), (1.0, 1.0)}
```

`pymule.loader.mergefks(*sets: XiRecord, **kwargs: XiRecord) → Record`

performs the FKS merge

Parameters

sets – random-seed-merged results of type *XiRecord* (usually from *sigma()*)

Returns

the FKS-merged final set containing cross sections, distributions, and run-time information.

Return type

Record

Note: Optional argument `anyxi` (or anything starting with `anyxi`): Sometimes it is necessary to merge ξ_c -dependent runs (such as a counter term) and ξ_c -independent runs (such as the one-loop term).

Example

Load the LO results for the muon decay using *sigma()*

```
>>> mergefks(sigma("m2enn0"))
>>> <Record m2enn0 xsec = 2211500(2), 1 histograms>
```

Load the NLO results

```
>>> mergefks(sigma("m2ennV"), sigma("m2ennR"))
<Record xsec = -4143205.8581450013, 1 histograms>
```

Load the NNLO results where `m2ennNF` does not depend on ξ_c

```
>>> mergefks(
...     sigma("m2ennFF"), sigma("m2ennRF"), sigma("m2ennRR"),
...     anyxi=sigma("m2ennNF")
... )
<Record xsec = -131.13(9), 1 histograms>
```

`pymule.loader.multiintersect(lists: list[list[T] | set[T]]) → set[T]`

finds elements that are common to all lists. This is used to find a list of FKS parameters of a given run.

Parameters

list – list of lists l_1, l_2, \dots, l_n

Returns

the list $l_1 \cap l_2 \cap \dots \cap l_n$

`pymule.loader.setup(folder: str | None = None, **kwargs: Any) → None`

sets the default arguments for *sigma()*.

Parameters

- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files Initialised to current directory (`.`).
- **flavour** – str, optional; the flavour to load, defaults to everything Initialised to everything, i.e. `.*`.
- **obs** – str, optional; the observable to load (the bit after the 0), defaults to everything Initialised to everything, i.e. `'`.

- **folderp** – str, optional; a regular expression to match directory structures of a tar file, defaults to everything Initialised to everything, i.e. `.*`.
- **merge** – dict, optional: a dict of histograms `{ 'name' : n }` to merge `n` bins in the histogram name. Initialised to to no merging, i.e. `{}`
- **sanitycheck** – callable, optional; a function that, given a *FileRecord*, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.

Example

Setup some folders, ensure that `/tmp/mcmule` exists

```
>>> setup(folder="path/to/data.tar.bz2", cachefolder="/tmp/mcmule")
```

Example

Restrict observable

```
>>> setup(obs="3")
```

Example

Drop runs with a $\chi^2 > 10$

```
>>> setup(sanitycheck=lambda x : x['chi2a'] < 10)
```

`pymule.loader.sigma`(*piece: str, **kwargs: Any*) → *XiRecord*

loads a `which_piece` and returns a *XiRecord*

Parameters

- **piece** – str; `which_piece` to load
- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files Initialised to current directory (`.`).
- **flavour** – str, optional; the flavour to load, defaults to everything Initialised to everything, i.e. `.*`.
- **obs** – str, optional; the observable to load (the bit after the 0), defaults to everything. Initialised to everything, i.e. `' '`.
- **folderp** – str, optional; a regular expression to match directory structures of a tar file, defaults to everything. Initialised to everything, i.e. `.*`.
- **merge** – dict, optional: a dict of histograms `{ 'name' : n }` to merge `n` bins in the histogram name. Initialised to to no merging, i.e. `{}`
- **sanitycheck** – callable, optional; a function that, given a *FileRecord*, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.

Returns

a dict with the tuples of FKS parameters as keys and vegas datasets as values.

Note: Use `setup()` to set the defaults. Arguments provided here override the defaults

Example

Load the leading order muon decay

```
>>> sigma("m2enn0")
>>> <XiRecord set m2enn0, xic = [1.0], 1 histograms>
```

Load only observable 03

```
>>> sigma("m2enn0", obs="3")
>>> <XiRecord set m2enn0, xic = [1.0], 1 histograms>
```

16.5 Working with ξ_c data

```
pymule.xicut.get_errorbands(x: <MagicMock name='mock.typing.NDArray.__getitem__().__or__()'
                             id='140279712303568'>, coeff: <MagicMock
                             name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, covar:
                             <MagicMock name='mock.typing.NDArray.__getitem__()'
                             id='140279735182480'>, ndata: int, cf: float = 0.9) → <MagicMock
                             name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

evaluates the errorbands of the fit obtained by `get_val()`

Parameters

- **x** – iterable; values of ξ_c to evaluate
- **coeff** – list; coefficient list [a_0 , a_1 , ..., a_n]
- **covar** – list; the covariance matrix
- **ndata** – int; the number of data points used in the fit, required for the t value estimation
- **cl** – float, optional; the confidence level used

Returns

list; the values and errors of the fit at the presented values as [x , y , $y-dy$, $y+dy$]

```
pymule.xicut.get_val(xs: ~typing.Iterable[float], coeff: <MagicMock
                    name='mock.typing.NDArray.__getitem__().__ror__()' id='140279714938768'>) →
                    list[float]
```

evaluates the fit obtained by `get_val()`

Parameters

- **xs** – iterable; values of ξ_c to evaluate
- **coeff** – list; coefficient list [a_0 , a_1 , ..., a_n]

Returns

list; the values of the fit at the presented values.

```
pymule.xicut.mergefkswithplot(*psets: XiRecord, showfit: tuple[bool, bool] = (True, True), xlim: tuple[float,
                                                                    float] = (-7.0, 0.0)) → tuple[Any, Record]
```

performs and FKS merge like `mergefks()` but it also produces a ξ_c independence plot.

Note: In contrast `mergefks()`, here phase-space partitioned results need to be summed first.

Parameters

- **sets** – a sequence of *XiRecord* (or sums thereof, usually from *sigma()*), starting with the lowest particle number and going up
- **showfit** – [bool, bool], optional; whether to show the fit lines in the overview plot (first element) and the zoomed in plot (second element)
- **xlim** – tuple of floats, optional; upper and lower bounds for $\log \xi_c$

Returns

a figure and the FKS-merged *Record* containing cross sections, distributions, and run-time information.

Example

In the partitioned muon-electron scattering case

```
>>> fig, res = mergefkswithplot(
...     alpha*(sigma('em2emFEE')+sigma('em2emFMM')+sigma('em2emFEM')),
...     alpha*(
...         sigma('em2emREE15') + sigma('em2emREE35') +
...         sigma('em2emRMM') + sigma('em2emREM')
...     )
... )
```

```
pymule.xicut.myfit(data: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
n: int) → tuple[<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>]
```

performs a log-polynomial $\sum_{i=0}^n a_i \log(\xi_c)^i$ fit

Parameters

- **data** – numpy array; The different ξ_c values in the format `np.array([[xi1, y1, e1], [xi2, y2, e2], ...])`.
- **n** – the degree of the polynomial

Result

the coefficients and covariant matrix

```
pymule.xicut.xiresidue(pset: XiRecord, n: int, xlim: tuple[float, float] = (-7.0, 0.0)) → tuple[~typing.Any, <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>]
```

creates a residue plot for a ξ_c fit

Parameters

- **sets** – a *XiRecord*
- **n** – int; order of the fit, 1 at NLO, 2 at NNLO
- **xlim** – tuple of floats, optional; upper and lower bounds for $\log \xi_c$

Returns

a figure and the fit coefficients as a matrix

16.6 Working with plots

`pymule.plot.errorband(p, ax=None, col='default', underflow=False, overflow=False, linestyle='solid')`

plots an errorband of a compatible histogram

Parameters

- **p** – Nx3 numpy matrix; the histogram to plot as `np.array([[x1, y1, e1], [x2, y2, e2], ...])`
- **ax** – axes, optional: the axes object to use, defaults to `gca()` which may create a new axes.
- **col** – the colour to be used for the plot. Per default matplotlib decides using the order specified in [colours](#)
- **underflow** – bool, optional; whether to plot the underflow bin. Either logical or number indicating the how much bigger it shall be
- **overflow** – bool, optional; whether to plot the overflow bin. Either logical or number indicating the how much bigger it shall be
- **linestyle** – str, optional; which line style to use

Returns

the artis of the main line but not the one of the errorbars

Example

Make a simple plot

```
>>> errorband(dat)
```

Make a plot in red with dashed lines

```
>>> errorband(dat, 'red', 'dashed')
```

`pymule.plot.format_label_string_with_exponent(ax, axis='both')`

Format the label string with the exponent from the ScalarFormatter

```
pymule.plot.kplot(sigma, labelx='$x_e$', labelsigma=None, labelknlo='$\delta K^{(1)}$', labelknnlo='$\delta K^{(2)}$', legend={'lo': '$\nu$rm LO$', 'nlo': '$\nu$rm NLO$', 'nnlo': '$\nu$rm NNLO$'}, legendopts={'loc': 'upper right', 'what': 'T'}, linestyle2=':', show=[0, -1], showk=[1, 2], nomule=False)
```

produces a K factor plot in line with McMule’s design, i.e. a two-panel plot showing in the upper panel the cross sections and in the lower panel the K factor defined as

$$K^{(i)} = d\sigma^{(i)} / d\sigma^{(i-1)}$$

Parameters

- **sigma** – dict; the data to plot, given as a dict with keys `lo`, `nlo`, and possibly `nnlo`. Only pass the corrections, not the full distribution
- **labelx** – str, optional; label for the x axis (supports LaTeX maths)
- **labelsigma** – str, optional; label for the upper y axis (supports LaTeX maths)
- **labelknlo** – str, optional; the labels for the NLO K factor
- **labelknnlo** – str, optional; the labels for the NNLO K factor

- **show** – list, optional; a list which cross sections to show, 0 indicates the LO cross section, 1 the NLO etc. -1 indicates the last given cross section
- **showk** – list, optional; a list which K factors to show, 0 indicates the LO cross section, 1 the NLO etc. -1 indicates the last given cross section
- **legend** – dict, optional; a dict with the legend for `lo`, `nlo`, `nnlo`. The keys `nlo2` and `nnlo2` are optional and will be drawn dashed in the lower panel.
- **legendopts** – dict, optional; a kwargs dict of options to be passed to `legend(...)` as well as the `what` key indicating whether the legend such be placed in the lower panel (1, default), upper panel (u), or as a `figlegend` (fig). Notable is the `loc`-key that places the legend inside the object specified by `what`. Possible values are (cf. `legend`)
 - upper right
 - upper left
 - lower left
 - lower right
 - right
 - center left
 - center right
 - lower center
 - upper center
 - center
- **nomule** – bool, optional; if set to True, no mule will be printed

Returns

the figure as well as all axis created

Example

An NNLO K factor plot

```
>>> fig, (ax1, ax2, ax3) = kplot(
...     {
...     'lo': lodata['thetae'],
...     'nlo': nldata['thetae'],
...     'nnlo': nnldata['thetae'],
...     },
...     labelx="$\theta_e, \backslash, \{\rm mrad}\$",
...     labelsigma="$\D\sigma/\D\theta_e \backslash \{\rm \upmu b}\$",
...     legend={
...     'lo': '$\sigma^{\{0\}}$',
...     'nlo': '$\sigma^{\{1\}}$',
...     'nnlo': '$\sigma^{\{2\}}$'
...     },
...     legendopts={'what': 'u', 'loc': 'lower right'}
... )
```

`pymule.plot.setup_pgf()`

`setup_pgf()` ensures that Matplotlib exports PGF compatible plots.

```
pymule.plot.threepanel(labelx="", upleft=[], labupleft="", colupleft=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], middleleft=[], labmiddleleft="", colmiddleleft=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], downleft=[], labdownleft="", coldownleft=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'])
```

creates three panel plot, accommodating at most three axes (upper, middle, lower). The x axis is naturally shared.

Parameters

- **labelx** – str, optional; label for the x axis
- **upleft** – Nx3 numpy matrix or list thereof, optional; data plotted in the upper-left axes
- **colupleft** – colour for upper-left data, defaults to colour scheme defined in [colours](#)
- **labupleft** – str, optional; the label for the upper-left data
- **middleleft** – Nx3 numpy matrix or list thereof, optional; data plotted in the middle-left axes
- **colmiddleleft** – colour for middle-left data, defaults to colour scheme defined in [colours](#)
- **labmiddleleft** – str, optional; the label for the middle-left data
- **downleft** – Nx3 numpy matrix or list thereof, optional; data plotted in the lower-left axes
- **coldownleft** – colour for lower-left data, defaults to colour scheme defined in [colours](#)
- **labdownleft** – str, optional; the label for the lower-left data

Returns

the figure and a list of all axes created

```
pymule.plot.twopanel(labelx="", upleft=[], labupleft="", colupleft=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], downleft=[], labdownleft="", coldownleft=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], upright=[], labupright="", colupright=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], downright=[], labdownright="", coldownright=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'], upalign=[], downalign=[])
```

creates two panel plot, accommodating at most four axes (upper left, upper right, lower left, and lower right). The x axis is naturally shared.

Parameters

- **labelx** – str, optional; label for the x axis
- **upleft** – Nx3 numpy matrix or list thereof, optional; data plotted in the upper-left axes
- **colupleft** – colour for upper-left data, defaults to colour scheme defined in [colours](#)
- **labupleft** – str, optional; the label for the upper-left data
- **upright** – Nx3 numpy matrix or list thereof, optional; data plotted in the upper-right axes
- **colupright** – colour for upper-right data, defaults to colour scheme defined in [colours](#)
- **labupright** – str, optional; the label for the upper-right data
- **downleft** – Nx3 numpy matrix or list thereof, optional; data plotted in the lower-left axes
- **coldownleft** – colour for lower-left data, defaults to colour scheme defined in [colours](#)
- **labdownleft** – str, optional; the label for the lower-left data
- **downright** – Nx3 numpy matrix or list thereof, optional; data plotted in the lower-right axes
- **coldownright** – colour for lower-right data, defaults to colour scheme defined in [colours](#)
- **labdownright** – str, optional; the label for the lower-right data

- **upalign** – list of two values, optional; align the first and second values of the left and right y axes in the upper panel
- **downalign** – list of two values, optional; align the first and second values of the left and right y axes in the lower panel

Returns

the figure and a list of all axes created

Example

make a comparison plot between `dat` and `dat_ref` as a $d\sigma/d\theta_e$

```
>>> fig, (ax1, ax2)=twopanel(
...     r'$\theta_e\,/\,\{\rm mrad}\$',
...     upleft=[dat, dat_ref],
...     downleft=divideplots(dat, dat_ref),
...     labupleft=r"$D\sigma\,/\,D\theta_e\,/\,\upmu\{\rm b}\$",
...     labdownleft=r'$\rm rel. difference\$',
... )
```

```
pymule.plot.watermark(fig, txt='PRELIMINARY', fontsize=60, rotation=20)
```

watermarks a figure

Parameters

- **fig** – the figure to watermark
- **txt** – str, optional; the watermark text to use
- **fontsize** – int, optional; the fontsize of the watermark
- **rotation** – int, optional; the angle of the watermark in deg

Example

Watermark a figure as preliminary

```
>>> fig = figure()
>>> ...
>>> watermark(fig)
```

Watermark a figure as incomplete

```
>>> fig = figure()
>>> ...
>>> watermark(fig, "INCOMPLETE")
```

```
pymule.colours.alpha_composite(bg: str, fg: str, alpha: float) → str
```

calculates the result of alpha-compositing two colours

Parameters

- **bf** – colour specifier for the background
- **fg** – colour specifier for the foreground
- **alpha** – float; alpha value

Result

the resulting colour

`pymule.mule.mulfy(fig, delx=0, dely=0, col='lightgray', realpha=True)`

adds the McMule logo to a figure

Parameters

- **fig** – figure to add the logo
- **delx** – float, optional; shift the logo in x direction
- **dely** – float, optional; shift the logo in x direction
- **col** – colour specifier, optional; colour to use for the logo
- **realpha** – bool, optional; whether to re-run the alpha channel

`pymule.mpl_axes_aligner.yaxes(ax1, ax2, y1=1, y2=None)`

`yaxes(ax1, ax2, y=1)` changes the limits of `ax1` and `ax2` to align the values of `y` on both axis.

`yaxes(ax1, y1, ax2, y2)` changes the limits of the axis `ax1` and `ax2` such that the value for `y1` on `ax1` is aligned to the value of `y2` on `ax2`.

16.7 Useful other functions

`pymule.compress.uncompress(b)`

`uncompress("string")` recovers an expression from a compressed string representation generated by Mathematica's `Compress`. Only lists, numbers, and strings are supported. Lists can be nested.

`pymule.maths.Li2(x)`

`Li2(x)` returns `PolyLog[2, x]` for `x` as a number, a list, or an `np.ndarray`.

`pymule.maths.Li3(x)`

`Li3(x)` returns `PolyLog[3, x]` for `x` as a number, a list, or an `np.ndarray`.

16.8 Validator

The validator (cf. Section *Observable validator*) allows the checking of user files. It is implemented as part of `pymule`.

16.8.1 Phase space generation

exception `pymule.validator.psgen.NotEnoughEnergyError(min: float, ms: list[float])`

An error that indicates that not enough energy is available to generate this process. This may depend on the kinematic point.

```
pymule.validator.psgen.psgen(ra: <MagicMock name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, Ms: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>) → <MagicMock
name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

General purpose momentum generator

Parameters

- **ra** – array of random numbers to generate the momentum from. see note in `psgen_X_FKS()` for details

- **Ms** – array of masses including 1 initial state masses and (≥ 2) final state masses

```
pymule.validator.psgen.psgen_FKS(ra: <MagicMock name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, Ms: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, NumPhotons: int) → <MagicMock
name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

General purpose FKS-photon momentum generator

Parameters

- **ra** – array of random numbers to generate the momentum from. see note in [psgen_X_FKS\(\)](#) for details
- **Ms** – array of masses including 1 initial state mass and (≥ 2) final state masses, NOT including FKS photons

```
pymule.validator.psgen.psgen_X(ra: <MagicMock name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, Ms: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, scms: float) → <MagicMock
name='mock.typing.NDArray.__getitem__()' id='140279735182480'>
```

General purpose cross-section momentum generator

Parameters

- **ra** – array of random numbers to generate the momentum from. see note in [psgen_X_FKS](#) for details
- **Ms** – array of masses including 2 initial state masses and (≥ 2) final state masses
- **scms** – center of momentum energy for the system, typically s in the literature.

```
pymule.validator.psgen.psgen_X_FKS(ra: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, Ms: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, NumPhotons: int, scms: float) →
<MagicMock name='mock.typing.NDArray.__getitem__()'
id='140279735182480'>
```

General purpose cross section and FKS-photon momentum generator

Parameters

- **ra** – array of random numbers to generate the momentum from.
- **Ms** – array of masses including 2 initial state masses and (≥ 2) final state masses, NOT including FKS-photons
- **scms** – centre of mass energy for the system

Note: Letting E_γ be a photon energy, n_γ be number of photons, E_{inv} be decay energy, and n_{masses} be number of masses, the random numbers are structured as:

$$[E_\gamma] \otimes n_\gamma, [\theta_\gamma \phi_\gamma] \otimes n_\gamma, [E_{inv} \theta_{masses} \phi_{masses}] * (n_{masses} - 4), [\theta_{masses} \phi_{masses}].$$

The - 4 comes from there being 2 masses in and 2 masses out in the X case, for non-cross section cases it is - 3 instead.

16.8.2 SO loader

class pymule.validator.importer.DistanceFunction

A *distance function* base class. This can be called which passes the momenta of two events and returns the distance.

```
classmethod build(func: ~typing.Callable[["<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,"<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>"], float]) → DistanceFunction
```

Used as a decorator to change a normal function into a “distance function”

class pymule.validator.importer.FortranDistanceFunction(fort_load: FortranLoader)

Parameters

fort_load – An instance of FortranLoader which is shared between Measurement and Distance

Attribute distance_names

List of names with which the distance function may be found

class pymule.validator.importer.FortranLoader(userfile: str)

A class which acts as a wrapper to the *measurement function* which is defined in the *userfile*.

Parameters

userfile – name of the file to be loaded, including extension default: user.so

Attribute inituser_names

List of names with which the *inituser()* function may be found. This function may be unavailable

Attribute filenamesuffixLen_names

List of names with which the *filenamesuffixLen* variable may be found

Attribute nrq_names

List of names with which the *nr_q* variable may be found

Attribute nrbins_names

List of names with which the *nr_bins* variable may be found

class pymule.validator.importer.FortranMeasurementFunction(fort_load: FortranLoader)

A *measurement function* implemented as a shared library for use in McMule. Calling this will evaluate *quant* and return the histogram values, *:f:var:pass_cut()*, and *names*

Parameters

fort_load – An instance of FortranLoader which is shared between Measurement and Distance

Attribute quant_names

List of names with which the measurement function may be found

process_names() → None

takes the binary names from the *userfile* and converts them into a string, assuming UTF-8 encoding.

Returns

a string of human-readable names.

class pymule.validator.importer.MeasurementFunction

A *measurement function* base class. This can be called which passes the momenta to the function and returns the histogram values, *pass_cut*, and *names*

pymule.validator.importer.names_t

alias of Any

16.8.3 Validation logic

```
class pymule.validator.validator.BaseValidator(npoints: int = 0, order_max: int = 100, t: int = 2,
                                              smoothness_rtol: float = 0.001, smoothness_ratol: float
                                              = 1.0, limit_rtol: float = 0.001, limit_atol: float =
                                              0.001, allowed_to_fail: bool = False, fail_msg: str =
                                              "")
```

Base class for the testing of a MeasurementFunction (or function with the same signature)

Attribute npoints

the number of Monte Carlo points to use. If zero, no MC tests are carried out

Attribute order_max

the maximum order to use in Richardson extrapolation

Attribute t

amount to reduce ξ_0 by each time (ξ_0/t^n) in the Richardson extrapolation

Attribute smoothness_rtol

used to determine smoothness, if two consecutive results differ by $r[1] / r[0] > \text{smoothness_rtol}$ then the function is not smooth

Attribute smoothness_atol

similar to `smoothness_rtol` but provides an absolute tolerance relative to the largest result found, i.e. $(r[1] - r[0]) > \text{ratol} * r[\text{max}]$ is not smooth

Attribute limit_rtol

allowed relative tolerance between the limit at 0 and the final result or allowed relative tolerance between the modified and unmodified results

Attribute limit_atol

allowed absolute tolerance between limit at 0 and final result or allowed absolute tolerance between the modified and unmodified results

Attribute allow_to_fail

failing these test is not fatal (eg. *RotationValidator*)

```
graph(xi: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, results:
      <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, limit_res:
      <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>, names:
      ~typing.Any, rtol: float | None = 0.001, atol: float | None = 0.001) → None
```

Parameters

- **xi** – x-axis, usually energy proportion of the soft photon
- **results** – y-axis, list of results corresponding to each point
- **limit_res** – horizontal line, result when $\xi \rightarrow 0$
- **names** – legend, names which are used to identify each line produced by results

```
modmomenta(momenta: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>)
            → None
```

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

```
modpoint(points: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
          **kwargs: ~typing.Any) → None
```

Parameters

points – numpy array of random numbers to be modified in place

```
class pymule.validator.validator.BoundsValidator(npoints: int, meas_func: MeasurementFunction,
                                               dist_func: DistanceFunction | None = None,
                                               **kwargs: Any)
```

Tests the histogram bounds by Monte Carlo sampling the phase space

```
class pymule.validator.validator.EventSwappingValidator(dist_func: DistanceFunction, meas_func:
                                                       MeasurementFunction | None = None,
                                                       **kwargs: Any)
```

Validates the symmetry relation $d(e_1, e_2) = d(e_2, e_1)$ for a *distance function*

```
class pymule.validator.validator.FlipValidator(axis: int, meas_func: MeasurementFunction, dist_func:
                                               DistanceFunction | None = None, **kwargs: Any)
```

Test spatial-symmetry by flipping across an axis. This test may fail

Parameters

axis – int, axis to be flipped across 0 for *x*, 1 for *y*, 2 for *z*

```
modmomenta(momenta: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>)
→ None
```

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

```
class pymule.validator.validator.LorentzBoostValidator(frame_momentum: <MagicMock
name='mock.typing.NDArray.__getitem__().__ror__()'
id='140279714938768'>, meas_func: ~pymule.validator.importer.MeasurementFunction,
dist_func: ~pymule.validator.importer.DistanceFunction |
None = None, **kwargs: ~typing.Any)
```

Test the boost-invariance, usually only used for cross-sections

This test may fail

Parameters

- **int** (*frame_momentum*,) – boost into the restframe of this particle using `boost_RF()`
- **vector** (*frame_momentum*, *numpy*) – the 4-momentum to boost the frame into using `boost_back()`

```
modmomenta(momenta: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>)
→ None
```

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

```
class pymule.validator.validator.MeasurementValidator
```

A mixin class for Validators that can test both *measurement function* and distance functions. It implements the test framework for discrete and continuous tests by creating a base point and modifying it.

Parameters

- **meas_func** – the *measurement function* callable
- **dist_func** – the *distance function* callable
- **limit_rtol** – relative tolerance

- **limit_atol** – absolute tolerance
- **softPhotonIndex** – the index or list of indices that correspond to soft photons
- **order_max** – the maximum order to use in Richardson extrapolation
- **t** – amount to reduce ξ_0 by each time (ξ_0/t^n) in the Richardson extrapolation :param t:

```
compare(base_result: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
base_pass_cut: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
mod_result: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
mod_pass_cut: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>,
rtol: float, atol: float | list[float]) → tuple[list[bool], list[bool]]
```

Compares a base point with a modified point.

Parameters

- **base_result** – result from the *measurement function* from the momenta
- **base_pass_cut** – *pass_cut* from the *measurement function* from the momenta
- **mod_result** – result from the *measurement function* evaluated with modified momenta from calling *BaseValidator.modpoint()* on the random numbers, and *BaseValidator.modmomenta()* on the momenta
- **mod_pass_cut** – *pass_cut* from the modified point
- **rtol** – relative tolerance between the numbers being compared
- **atol** – absolute tolerance between the numbers being compared, either a single number for all, or an array for individualised comparisons

Returns

a tuple of 2 boolean lists containing if each number was close or not

```
class pymule.validator.validator.Piece(name: str, p_in: list[~pymule.utils.pid.Particle], p_out:
list[~pymule.utils.pid.Particle], masses: <MagicMock name='mock.typing.NDArray.__getitem__().__ror__()' id='140279714938768'>,
scms: float | None, randlen: int, psgenfunc: ~typing.Callable[[...], <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>], numphotons: int | None, soft_photons: int)
```

A *which_piece* that can be used for validation by *BaseValidator*. Normally this is constructed with *Piece.from_whichpiece()* or *Piece.from_process()*

Attribute name

the name of the *which_piece*

Attribute p_in

a list of incoming Particle

Attribute p_out

a list of outgoing Particle

Attribute masses

a list of masses

Attribute scms

the centre-of-mass energy *s*

Attribute randlen

the number of random numbers required

Attribute psgenfunc

the phase space routine used by this

Attribute numphotons

the number of photons in the process, not counting the first two particles

Attribute soft_photons

the number of photons that actually can become soft

Can be used to call the phase space generator

static convert_masses(masses: list[str]) → list[float]

Parameters

masses – list of strings which correspond to physical masses, eg mm for the muon mass $m_\mu = 105$ MeV

Note: gets masses from pymule/constants.py

classmethod from_process(PIDTable_: PIDTable, process: dict[str, dict[str, list[str]]], order: str | None = None, scms: float | None = None) → Generator[Piece, None, None]

Yields all *Piece* required for a specific generic process up to a given order or all orders

Parameters

- **PIDTable** – the PIDTable
- **process** – a dictionary encoding the process, cf. *load_processes_from_json()*
- **order** – str, optional;

Returns

a generator of *Piece*

classmethod from_whichpiece(PIDTable_: PIDTable, whichpiece: str, scms: float | None = None) → *Piece*

Generates a *Piece* from a *which_piece* and the PIDTable

Parameters

- **PIDTable** – the PIDTable
- **whichpiece** – the *which_piece*, eg. em2em0

Returns

a *Piece*

Warning: The number of photons numphotons is only the number of photons that can be generated using FKS. For processes such as ee2gg, these will not be listed

static get_soft_from_sigma(fxn: str) → int

Parameters

fxn – sigma_n for $n = 0..9$, trailing characters after n are ignored

Returns

number of soft photons

Warning: currently doesn't work for sigma_YFS (i.e. arbitrary photons)

static pick_psgen_func(*p_in*: List[Particle], *p_out*: List[Particle]) → Tuple[Callable[[...], Any], int | None]

return the appropriate psgen function based on particles in and out

Parameters

- **p_in** – particles into the system
- **p_out** – particles out of the system

Returns

a tuple of the phase space generator and the number of FKS photons (None if no FKS exist)

Note: there can be photons present which cannot be generated by FKS (e.g. ee2gg or eg2eg) this is handled by returning number of FKS photons not number of photons total

class pymule.validator.validator.RotationValidator(*alpha*: float, *beta*: float, *gamma*: float, *meas_func*: MeasurementFunction, *dist_func*: DistanceFunction | None = None, ***kwargs*: Any)

Test the rotational-invariance of the *measurement function* and *distance function*, e.g. rotating all the momenta by $\pi/3$ around the *x*-axis shouldn't change either result

This test may fail

Parameters

- **alpha** – Euler angle for rotation
- **beta** – Euler angle for rotation
- **gamma** – Euler angle for rotation

modmomenta(*momenta*: <MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>) → None

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

class pymule.validator.validator.SameZeroValidator(*dist_func*: DistanceFunction, *meas_func*: MeasurementFunction | None = None, ***kwargs*: Any)

Validates the relation $d(e_1, e_1) = 0$ for a *distance function*

class pymule.validator.validator.SoftLimitValidator(*softPhotonIndex*: int | list[int] | tuple[int], *meas_func*: MeasurementFunction, *dist_func*: DistanceFunction | None = None, ***kwargs*: Any)

Test whether $\lim_{\gamma \rightarrow 0} S(q, g) = S(q)$

Parameters

softPhotonIndex – either the index of the random number associated with the photon energy or a list of indices which should go to 0 with the same ratio

modpoint (*points*: `<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`, *t*: `int = 2`, *zero*: `bool = False`) \rightarrow None

Parameters

points – numpy array of random numbers to be modified in place

class `pymule.validator.validator.SoftSwapValidator` (*softPhotonIndex*: `int | list[int] | tuple[int]`, *swapAIndex*: `int`, *swapBIndex*: `int`, *meas_func*: `MeasurementFunction`, *dist_func*: `DistanceFunction | None = None`, ***kwargs*: `Any`)

Combination of the *SoftLimitValidator* and the *SoftSwapValidator* symmetry test, i.e.

$$\lim_{\gamma_1 \rightarrow 0} S(q, \gamma_1, \gamma_2) = S(q, \gamma_2) = \lim_{\gamma_2 \rightarrow 0} S(q, \gamma_2, \gamma_1) = S(q, \gamma_1)$$

Parameters

- **softPhotonIndex** – index of the random number associated with the photon energy
- **swapAIndex** – index of the first particle to be swapped
- **swapBIndex** – index of the second particle to be swapped

modmomenta (*momenta*: `<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`) \rightarrow None

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

modpoint (*points*: `<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`, *t*: `int = 2`, *zero*: `bool = False`) \rightarrow None

Parameters

points – numpy array of random numbers to be modified in place

class `pymule.validator.validator.SwapSymmetryValidator` (*swapAIndex*: `int`, *swapBIndex*: `int`, *meas_func*: `MeasurementFunction`, *dist_func*: `DistanceFunction | None = None`, ***kwargs*: `Any`)

Test whether $S(p_1, p_2, q) = S(p_2, p_1, q)$ for 2 identical particles p_1, p_2

Parameters

- **swapAIndex** – index of the first particle to be swapped
- **swapBIndex** – index of the second particle to be swapped

modmomenta (*momenta*: `<MagicMock name='mock.typing.NDArray.__getitem__()' id='140279735182480'>`) \rightarrow None

Parameters

momenta – a nx4 numpy matrix of 4-momenta which are transformed in place

class `pymule.validator.validator.TriangleValidator` (*dist_func*: `DistanceFunction`, *meas_func*: `MeasurementFunction | None = None`, ***kwargs*: `Any`)

Validates the triangle inequality $d(e_1, e_2) \geq d(e_1, e_3) + d(e_3, e_2)$ for a *distance function*

```
class pymule.validator.validator.ValidatorBuilder(meas_func: MeasurementFunction | None,
                                                dist_func: DistanceFunction | None = None,
                                                npoints: int = 0, order_max: int = 100, t: int = 2,
                                                smoothness_rtol: float = 0.001, smoothness_ratol:
                                                float = 1.0, limit_rtol: float = 0.001, limit_atol:
                                                float = 0.001, allowed_to_fail: bool = False,
                                                fail_msg: str = "")
```

Builder class to build tests from *BaseValidator* with a common set of initialisations

Attribute npoints

the number of Monte Carlo points to use. If zero, no MC tests are carried out

Attribute order_max

the maximum order to use in Richardson extrapolation

Attribute t

amount to reduce ξ_0 by each time (ξ_0/t^n) in the Richardson extrapolation

Attribute smoothness_rtol

used to determine smoothness, if two consecutive results differ by $r[1] / r[0] > \text{smoothness_rtol}$ then the function is not smooth

Attribute smoothness_atol

similar to `smoothness_rtol` but provides an absolute tolerance relative to the largest result found, i.e. $(r[1] - r[0]) > \text{ratol} * r[\text{max}]$ is not smooth

Attribute limit_rtol

allowed relative tolerance between the limit at 0 and the final result or allowed relative tolerance between the modified and unmodified results

Attribute limit_atol

allowed absolute tolerance between limit at 0 and final result or allowed absolute tolerance between the modified and unmodified results

Attribute allow_to_fail

failing these test is not fatal (eg. *RotationValidator*)

Attribute fail_msg

the message being printed if the test fails

Example

```
>>> # Looking at the m2ennR process where 1 photon is generated
>>> piece = Piece.from_whichpiece(table, "m2ennR")
>>> builder = ValidatorBuilder(
    meas_func=FortranMeasurementFunction(),
)
>>> # check if swapping the neutrinos causes a change
>>> swap_sym = builder.build_swap_symmetry(3, 4)
>>> swap_sym.run_test(piece)
```

build_all(*piece*: *Piece*) → Generator[*BaseValidator*, None, None]

Generates all tests that can be performed for this *Piece* with the given options

Returns

an iterator of *BaseValidator*

build_boost(*frame_momentum*: <MagicMock name='mock.typing.NDArray.__getitem__().__ror__()' id='140279714938768'>) → *LorentzBoostValidator*

build an instance of *LorentzBoostValidator* with parameters from the builder

Parameters

- **int** (*frame_momentum*,) – boost into the restframe of this particle using `boost_RF()`
- **vector** (*frame_momentum*, *numpy*) – the 4-momentum to boost the frame into using `boost_back()`

Returns

instance of *LorentzBoostValidator*

build_bounds() → *BoundsValidator*

build an instance of *BoundsValidator* with parameters from the builder

Parameters

npoints – int, number of Monte Carlo points to use

Returns

instance of *BoundsValidator*

build_eventswap() → *EventSwappingValidator*

build an instance of *EventSwappingValidator* with parameters from the builder

Returns

instance of *EventSwappingValidator*

build_flip(axis: int) → *FlipValidator*

build an instance of *FlipValidator* with parameters from the builder

Parameters

axis – int, axis to be flipped across 0 for *x*, 1 for *y*, 2 for *z*

Returns

instance of *FlipValidator*

build_rotation(alpha: float, beta: float, gamma: float) → *RotationValidator*

build an instance of *RotationValidator* with parameters from the builder

Parameters

- **alpha** – Euler angle for rotation
- **beta** – Euler angle for rotation
- **gamma** – Euler angle for rotation

Returns

instance of *RotationValidator*

build_samezero() → *SameZeroValidator*

build an instance of *SameZeroValidator* with parameters from the builder

Returns

instance of *SameZeroValidator*

build_soft_limit(softPhotonIndex: int | list[int] | tuple[int]) → *SoftLimitValidator*

build an instance of *SoftLimitValidator* with parameters from the builder

Parameters

softPhotonIndex – index of the random number associated with the photon energy

Returnsinstance of *SoftLimitValidator*

build_soft_swap(*softPhotonIndex*: int | tuple[int] | list[int], *swapAIndex*: int, *swapBIndex*: int) → *SoftSwapValidator*

build an instance of *SoftSwapValidator* with parameters from the builder**Parameters**

- **softPhotonIndex** – index of the random number associated with the photon energy
- **swapAIndex** – index of the first particle to be swapped
- **swapBIndex** – index of the second particle to be swapped

Returnsinstance of *SoftSwapValidator*

build_swap_symmetry(*swapAIndex*: int, *swapBIndex*: int) → *SwapSymmetryValidator*

build an instance of *SwapSymmetryValidator* with parameters from the builder**Parameters**

- **swapAIndex** – index of the first particle to be swapped
- **swapBIndex** – index of the second particle to be swapped

Returnsinstance of *SwapSymmetryValidator*

build_triangle() → *TriangleValidator*

build an instance of *TriangleValidator* with parameters from the builder**Returns**instance of *TriangleValidator*

`pymule.validator.validator.convert_to_PID(p_in: list[Particle], p_out: list[Particle])` → str
convert lists containing the incoming and outgoing :class:'Particle's to the generic group they belong to

Example

```
>>> p_in = [Muon(-1)]
>>> p_out = [Electron(-1), Neutrino(""), Neutrino("")]
>>> convert_to_PID(p_in, p_out)
"m2enn"
```

`pymule.validator.validator.do_test_from_whichpiece(PIDTable_: PIDTable, whichpiece: str, measurement_func: MeasurementFunction, distance_function: DistanceFunction | None = None, scms: float | None = None, **kwargs: Any)` → dict[*BaseValidator*, tuple[int, int, int]]

Performs all tests for one *which_piece*. See *ValidatorBuilder* for more options**Parameters**

- **PIDTable** – instance of *PIDTable* in which *whichpiece* can be found
- **measurement_func** – instance of *MeasurementFunction* or similar to be tested
- **whichpiece** – the string *which_piece* to generate tests for

Example

```

>>> PIDTable_ = PIDTable()
>>> M = MeasurementFunction()
>>> do_test_from_whichpiece(PIDTable_, M, "m2enn0")
...

```

`pymule.validator.validator.do_tests_from_process(PIDTable_: PIDTable, generic_process: str, measurement_function: MeasurementFunction, distance_function: DistanceFunction | None = None, order: str | None = None, scms: float | None = None, **kwargs: Any) → dict[BaseValidator, tuple[int, int]]`

Perform tests for all whichpieces under a *generic process* at a certain order (or all orders). See *ValidatorBuilder* for more options

Parameters

- **PIDTable** – instance of `PIDTable` in which the `generic_process` can be found
- **MeasurementFunction** – instance of `:class:'MeasurementFunction'` to be tested
- **generic_process** – the *generic process* to be tested with
- **order** – str: LO, NLO, NNLO (sometimes split into `el. NNLO` and `full NNLO`) None: Test all orders available

Example

```

>>> do_tests_from_process(..., generic_process="m2enn", order="NLO",...)
INFO:Testing whichpiece: 'm2ennV'...
INFO:Testing whichpiece: 'm2ennC'...
INFO:Testing whichpiece: 'm2ennR'...

```

Example

```

>>> do_tests_from_process(..., generic_process="m2enn", order=None,...)
INFO:Testing at order LO:
INFO:Testing whichpiece: 'm2enn0'...
INFO:Testing at order NLO:
INFO:Testing whichpiece: 'm2ennF'...
INFO:Testing whichpiece: 'm2ennR'...
INFO:Testing at order NNLO:
...

```

`pymule.validator.validator.load_processes_from_json(filename: str, path: Path) → dict[str, dict[str, dict[str, list[str]]]]`

Parameters

- **filename** – .json file to be loaded
- **path** – path to the file

Returns

dict-object containing the .json data

(Monte carlo for **Muons** and other **leptons**) is a generic framework for higher-order QED calculations of scattering and decay processes involving leptons. It is written in Fortran 95 with two types of users in mind. First, several processes are implemented, some at *NLO*, some at *NNLO*. For these processes, the user can define an arbitrary (infrared safe), fully differential observable and compute cross sections and distributions. McMule's processes, present and, future, are

listed in Table 16.1 together with the relevant experiments for which the cuts are implemented. Second, the program is set up s.t. additional processes can be implemented by supplying the relevant matrix elements.

Table 16.1: Processes implemented in McMule

| process | order | experiments | comments |
|---|-------|----------------|-------------------------------|
| $\mu \rightarrow \nu \bar{\nu} e$ | NNLO | MEG I&II | polarised, massified & exact |
| $\mu \rightarrow \nu \bar{\nu} e \gamma$ | NLO | MEG I | polarised |
| $\mu \rightarrow \nu \bar{\nu} e e e$ | NLO | Mu3e | polarised |
| $\mu \rightarrow \nu \bar{\nu} e \gamma \gamma$ | LO | MEG | polarised |
| $\tau \rightarrow \nu \bar{\nu} e \gamma$ | NLO | BaBar | cuts in lab frame |
| $\tau \rightarrow \nu \bar{\nu} l l l$ | NLO | Belle II | |
| | NLO | MUonE | |
| | NNLO | | purely electronic corrections |
| | | | mixed (massified) |
| $\ell p \rightarrow \ell p$ | NNLO | P2, MUSE, Prad | only leptonic corrections |
| $e^- e^- \rightarrow e^- e^-$ | NNLO | Prad | complete |
| $e^+ e^- \rightarrow e^+ e^-$ | NNLO | | no n_f |
| $e^+ e^- \rightarrow \gamma \gamma$ | NNLO | PADME | |
| $e^+ e^- \rightarrow \mu^+ \mu^-$ | NNLO | Belle | massified |

The public version of the code can be found at

<https://gitlab.com/mule-tools/mcmule>

To obtain a copy of the code, `git` is recommended

```
$ git clone --recursive https://gitlab.com/mule-tools/mcmule
```

Alternatively, we provide a Docker *container* for easy deployment and legacy results (cf. Section *Basics of containerisation*). In multi-user environments, **udocker** can be used instead. In either case, a pre-compiled copy of the code can be obtained by calling

```
$ docker pull yulrich/mcmule # requires Docker to be installed
$ udocker pull yulrich/mcmule # requires udocker to be installed
```

We provide instructions on how it is used in Section *Getting started*.

Chapter 17

Indices and tables

- `genindex`
- `modindex`
- `search`

Bibliography

- [1] Federico Buccioni, Jean-Nicolas Lang, Jonas M. Lindert, Philipp Maierhöfer, Stefano Pozzorini, Hantian Zhang, and Max F. Zoller. OpenLoops 2. *Eur. Phys. J. C*, 79(10):866, 2019. arXiv:1907.13071, doi:10.1140/epjc/s10052-019-7306-2.
- [2] Federico Buccioni, Stefano Pozzorini, and Max Zoller. On-the-fly reduction of open loops. *Eur. Phys. J. C*, 78(1):70, 2018. arXiv:1710.11452, doi:10.1140/epjc/s10052-018-5562-1.
- [3] A. Denner and S. Dittmaier. Scalar one-loop 4-point integrals. *Nucl. Phys.*, B844:199–242, 2011. arXiv:1005.2076, doi:10.1016/j.nuclphysb.2010.11.002.
- [4] Ansgar Denner and S. Dittmaier. Reduction of one loop tensor five point integrals. *Nucl. Phys.*, B658:175–202, 2003. arXiv:hep-ph/0212259, doi:10.1016/S0550-3213(03)00184-6.
- [5] Ansgar Denner and S. Dittmaier. Reduction schemes for one-loop tensor integrals. *Nucl. Phys.*, B734:62–115, 2006. arXiv:hep-ph/0509141, doi:10.1016/j.nuclphysb.2005.11.007.
- [6] Ansgar Denner, Stefan Dittmaier, and Lars Hofer. Collier: a fortran-based Complex One-Loop Library in Extended Regularizations. *Comput. Phys. Commun.*, 212:220–238, 2017. arXiv:1604.06792, doi:10.1016/j.cpc.2016.10.013.
- [7] T. Engel, C. Gnendiger, A. Signer, and Y. Ulrich. Small-mass effects in heavy-to-light form factors. *JHEP*, 02:118, 2018. arXiv:1811.06461, doi:10.1007/JHEP02(2019)118.
- [8] T. Engel, A. Signer, and Y. Ulrich. A subtraction scheme for massive QED. *JHEP*, 01:085, 2020. arXiv:1909.10244, doi:10.1007/JHEP01(2020)085.
- [9] Tim Engel, Adrian Signer, and Yannick Ulrich. Universal structure of radiative QED amplitudes at one loop. *JHEP*, 04:097, 2022. arXiv:2112.07570, doi:10.1007/JHEP04(2022)097.
- [10] M. Fael, L. Mercolli, and M. Passera. Radiative μ and τ leptonic decays at NLO. *JHEP*, 07:153, 2015. arXiv:1506.03416, doi:10.1007/JHEP07(2015)153.
- [11] Eleftherios Gkioulekas. Using restrictions to accept or reject solutions of radical equations. *Int. J. of Mathematical Education in Science and Technology*, 49(8):1278–1292, 2018. doi:10.1080/0020739X.2018.1458341.
- [12] Jorge Gomes, Emanuele Bagnaschi, Isabel Campos, Mario David, Luís Alves, João Martins, João Pina, Alvaro López-García, and Pablo Orviz. Enabling rootless Linux Containers in multi-user environments: the \$sudocker\$ tool. *Comput. Phys. Commun.*, 232:84–97, 2018. arXiv:1711.01758, doi:10.1016/j.cpc.2018.05.021.
- [13] John D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.
- [14] J. P. Lees and others. Measurement of the branching fractions of the radiative leptonic τ decays $\tau \rightarrow e \gamma \nu \bar{\nu}$ and $\tau \rightarrow \mu \gamma \nu \bar{\nu}$ at textsc BaBar. *Phys. Rev.*, D91:051103, 2015. arXiv:1502.01784, doi:10.1103/PhysRevD.91.051103.
- [15] G. Peter Lepage. VEGAS: an adaptive multidimensional integration program. Technical Report, "LNS Cornell", Mar 1980.

- [16] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, 61(1):25–28, 1968. URL: <https://www.pnas.org/content/61/1/25>, arXiv:<https://www.pnas.org/content/61/1/25.full.pdf>, doi:10.1073/pnas.61.1.25.
- [17] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, March 2014.
- [18] B. Oberhof. *Measurement of $B(\tau \rightarrow \Gamma \nu \bar{\nu} \nu, l=e, \mu)$ at BaBar*. PhD thesis, University of Pisa, Italy, 2015.
- [19] S. K. Park and K. W. Miller. Random Number Generators: Good Ones Are Hard to Find. *Commun. ACM*, 31(10):1192–1201, October 1988. URL: <http://doi.acm.org/10.1145/63039.63042>, doi:10.1145/63039.63042.
- [20] Hiren H. Patel. Package-X: A Mathematica package for the analytic calculation of one-loop integrals. *Comput. Phys. Commun.*, 197:276–290, 2015. arXiv:1503.01469, doi:10.1016/j.cpc.2015.08.017.
- [21] G. M. Pruna, A. Signer, and Y. Ulrich. Fully differential NLO predictions for the radiative decay of muons and taus. *Phys. Lett.*, B772:452–458, 2017. arXiv:1705.03782, doi:10.1016/j.physletb.2017.07.008.
- [22] Fernando Pérez and Brian E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, 9(3):21–29, 2007. doi:10.1109/MCSE.2007.53.
- [23] Y. Ulrich. " FKS^2 : extending the FKS scheme to double soft correction". Technical Report, "Paul Scherrer Institute", "2019".
- [24] Y. Ulrich. Fully differential NLO predictions for rare and radiative lepton decays. *PoS, NuFact2017*:124, 2018. arXiv:1712.05633, doi:10.22323/1.295.0124.
- [25] Yannick Ulrich. *McMule: QED Corrections for Low-Energy Experiments*. PhD thesis, University of Zurich, 2020. arXiv:2008.09383.
- [26] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.
- [27] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "slurm: simple linux utility for resource management". In *Job Scheduling Strategies for Parallel Processing*, 44–60. Berlin, Heidelberg, "2003". Springer Berlin Heidelberg.
- [28] Rikkert Frederix, Stefano Frixione, Fabio Maltoni, and Tim Stelzer. Automation of next-to-leading order computations in QCD: the FKS subtraction. *Journal of High Energy Physics*, Oct 2009. arXiv:0908.4272v2.
- [29] S. Frixione, Z. Kunszt, and A. Signer. Three-jet cross sections to next-to-leading order. *Nuclear Physics B*, 467(3):399–442, May 1996. arXiv:hep-ph/9512328v1.

Python Module Index

p

- `pymule.colours`, 153
- `pymule.compress`, 154
- `pymule.errortools`, 127
- `pymule.loader`, 140
- `pymule.maths`, 154
- `pymule.mpl_axes_aligner`, 154
- `pymule.mule`, 153
- `pymule.plot`, 150
- `pymule.record`, 134
- `pymule.validator.importer`, 156
- `pymule.validator.psgen`, 154
- `pymule.validator.validator`, 157
- `pymule.vegas`, 137
- `pymule.xicut`, 148

Index

A

absvec() (fortran function), **51**
add_folder() (pymule.loader.Options method), 143
addplots() (in module pymule.errortools), 127
alpha (fortran variable), **48**
alpha_composite() (in module pymule.colours), 153
as_record() (pymule.loader.XiRecord method), 144
asymtensor() (fortran function), **50**

B

BareRecordV4 (class in pymule.vegas), 137
BaseValidator (class in pymule.validator.validator), 157
bin_kind (fortran variable), **57**
boost_back() (fortran function), **51**
boost_rf() (fortran function), **51**
BoundsValidator (class in pymule.validator.validator), 158
BR, **101**
build() (pymule.validator.importer.DistanceFunction class method), 156
build_all() (pymule.validator.validator.ValidatorBuilder method), 163
build_boost() (pymule.validator.validator.ValidatorBuilder method), 163
build_bounds() (pymule.validator.validator.ValidatorBuilder method), 164
build_eventswap() (pymule.validator.validator.ValidatorBuilder method), 164
build_flip() (pymule.validator.validator.ValidatorBuilder method), 164
build_rotation() (pymule.validator.validator.ValidatorBuilder method), 164
build_samezero() (pymule.validator.validator.ValidatorBuilder method), 164
build_soft_limit() (pymule.validator.validator.ValidatorBuilder method), 164
build_soft_swap() (pymule.validator.validator.ValidatorBuilder method), 165

build_swap_symmetry() (pymule.validator.validator.ValidatorBuilder method), 165
build_triangle() (pymule.validator.validator.ValidatorBuilder method), 165

C

can_open() (pymule.loader.FolderLoader class method), 140
can_open() (pymule.loader.GitLabLoader class method), 141
can_open() (pymule.loader.Loader class method), 141
can_open() (pymule.loader.TarLoader class method), 143
chain() (pymule.loader.Loader static method), 141
Chi2 (class in pymule.record), 134
Chi2.Type (class in pymule.record), 134
chi2a (pymule.record.Record property), 136
chisq() (in module pymule.errortools), 127
collcut (fortran variable), **54**
combineNplots() (in module pymule.errortools), 128
combineplots() (in module pymule.errortools), 128
compare() (pymule.validator.validator.MeasurementValidator method), 159
config file, **102**
container, **102**
containerisation, **102**
convert_masses() (pymule.validator.validator.Piece static method), 160
convert_to_PID() (in module pymule.validator.validator), 165
corner region, **102**
cos_th() (fortran function), **50**
counter-event, **102**

D

DiscB() (fortran function), **117**
DiscB_cplx() (fortran function), **117**
distance function, **102**
DistanceFunction (class in pymule.validator.importer), 156
dividenumbers() (in module pymule.errortools), 129
divideplots() (in module pymule.errortools), 130

- do_test_from_whichpiece() (in module *py-mule.validator.validator*), 165
- do_tests_from_process() (in module *py-mule.validator.validator*), 166
- DSL, 101
- ## E
- eik() (fortran function), 116
- errorband() (in module *pymule.plot*), 150
- eta() (fortran function), 50
- euler_mat() (fortran function), 52
- event, 102
- EventSwappingValidator (class in *py-mule.validator.validator*), 158
- EW, 101
- executor, 102
- ## F
- fatal, 103
- filenamesuffix (fortran variable), 48
- filenamesuffixLen (fortran variable), 57
- FileRecord (class in *pymule.vegas*), 137
- FKS, 101
- flavour (fortran variable), 54
- FlipValidator (class in *pymule.validator.validator*), 158
- FoamRecordV4 (class in *pymule.vegas*), 138
- FolderLoader (class in *pymule.loader*), 140
- format_label_string_with_exponent() (in module *pymule.plot*), 150
- FortranDistanceFunction (class in *py-mule.validator.importer*), 156
- FortranLoader (class in *pymule.validator.importer*), 156
- FortranMeasurementFunction (class in *py-mule.validator.importer*), 156
- from_merge() (*pymule.record.Chi2* class method), 134
- from_process() (*pymule.validator.validator.Piece* class method), 160
- from_single() (*pymule.record.Chi2* class method), 135
- from_whichpiece() (*pymule.validator.validator.Piece* class method), 160
- FSR, 101
- full period, 103
- ## G
- generic piece, 103
- generic process, 103
- get_errorbands() (in module *pymule.xicut*), 148
- get_soft_from_sigma() (py-mule.validator.validator.Piece static method), 160
- get_val() (in module *pymule.xicut*), 148
- GF (fortran variable), 48
- GitLabLoader (class in *pymule.loader*), 141
- graph() (*pymule.validator.validator.BaseValidator* method), 157
- guess_version() (in module *pymule.vegas*), 139
- ## H
- histograms (*pymule.loader.XiRecord* property), 144
- histograms (*pymule.record.Record* property), 136
- HVP, 101
- ## I
- ieik() (fortran function), 116
- image, 103
- initflavour() (fortran subroutine), 49
- inituser() (fortran subroutine), 58
- integratehistogram() (in module *pymule.errortools*), 130
- IR, 101
- ISR, 101
- ## K
- kappa (fortran variable), 49
- kplot() (in module *pymule.plot*), 150
- ## L
- lambda (fortran variable), 49
- Li2() (in module *pymule.maths*), 154
- Li3() (in module *pymule.maths*), 154
- list() (*pymule.loader.FolderLoader* method), 140
- list() (*pymule.loader.GitLabLoader* method), 141
- list() (*pymule.loader.Loader* method), 142
- list() (*pymule.loader.TarLoader* method), 143
- LO, 101
- load() (*pymule.loader.XiRecord* class method), 144
- load_processes_from_json() (in module *py-mule.validator.validator*), 166
- load_quant() (fortran subroutine), 52
- Loader (class in *pymule.loader*), 141
- LorentzBoostValidator (class in *py-mule.validator.validator*), 158
- LP, 101
- ## M
- make_mlm() (fortran function), 115
- max_val (fortran variable), 58
- mcmule_alpha (C member), 56
- mcmule_bin_kind (C member), 59
- mcmule_CLj (C member), 56
- mcmule_CRj (C member), 56
- mcmule_filename_suffix_length (C member), 59
- mcmule_filenamesuffix (C member), 55
- mcmule_Gf (C member), 56
- mcmule_initflavour (C function), 57

- mcmule_lower_bounds (C member), 59
 mcmule_Me (C member), 56
 mcmule_measurement_function (C function), 60
 mcmule_Mj (C member), 56
 mcmule_Mm (C member), 56
 mcmule_Mtau (C member), 56
 mcmule_musq (C member), 55
 mcmule_namelen (C member), 59
 mcmule_names (C member), 55
 mcmule_Nel (C member), 56
 mcmule_Nhad (C member), 56
 mcmule_Nmu (C member), 56
 mcmule_nr_q (C member), 56
 mcmule_Ntau (C member), 56
 mcmule_number_bins (C member), 59
 mcmule_number_hist (C member), 59
 mcmule_pass_cut (C member), 55
 mcmule_pol1 (C member), 55
 mcmule_pol2 (C member), 55
 mcmule_protonff (C member), 56
 mcmule_protonff_kappa (C member), 56
 mcmule_protonff_lambda (C member), 56
 mcmule_runmcmule (C function), 57
 mcmule_scms (C member), 56
 MCMULE_SET_NAME (C macro), 56
 mcmule_set_observable (C function), 57
 mcmule_sw2 (C member), 56
 mcmule_upper_bounds (C member), 59
 mcmule_user_initialisation (C function), 59
 mcmule_user_integration (C function), 60
 mcmule_user_integration_dimension (C member), 59
 mcmule_userweight (C member), 55
 Me (fortran variable), 48
 measurement function, 103
 MeasurementFunction (class in py-mule.validator.importer), 156
 MeasurementValidator (class in py-mule.validator.validator), 158
 Mel (fortran variable), 48
 menu file, 103
 merge() (pymule.record.Record class method), 136
 mergebins() (in module pymule.errortools), 131
 merged_seeds (pymule.loader.XiRecord property), 145
 mergefks() (in module pymule.loader), 145
 mergefkswithplot() (in module pymule.xicut), 148
 mergenumbers() (in module pymule.errortools), 131
 mergeplots() (in module pymule.errortools), 132
 min_val (fortran variable), 57
 Mj (fortran variable), 49
 mlm (fortran type), 115
 Mm (fortran variable), 48
 Mmu (fortran variable), 48
 modmomenta() (pymule.validator.validator.BaseValidator method), 157
 modmomenta() (pymule.validator.validator.FlipValidator method), 158
 modmomenta() (pymule.validator.validator.LorentzBoostValidator method), 158
 modmomenta() (pymule.validator.validator.RotationValidator method), 161
 modmomenta() (pymule.validator.validator.SoftSwapValidator method), 162
 modmomenta() (pymule.validator.validator.SwapSymmetryValidator method), 162
 modpoint() (pymule.validator.validator.BaseValidator method), 157
 modpoint() (pymule.validator.validator.SoftLimitValidator method), 161
 modpoint() (pymule.validator.validator.SoftSwapValidator method), 162
 module
 pymule.colours, 153
 pymule.compress, 154
 pymule.errortools, 127
 pymule.loader, 140
 pymule.maths, 154
 pymule.mpl_axes_aligner, 154
 pymule.mule, 153
 pymule.plot, 150
 pymule.record, 134
 pymule.validator.importer, 156
 pymule.validator.psgen, 154
 pymule.validator.validator, 157
 pymule.vegas, 137
 pymule.xicut, 148
 Mproton (fortran variable), 48
 Mt (fortran variable), 48
 Mtau (fortran variable), 48
 mulify() (in module pymule.mule), 153
 multiintersect() (in module pymule.loader), 146
 musq (fortran variable), 47
 myfit() (in module pymule.xicut), 149
 MZ (fortran variable), 48

N
 names (fortran variable), 47
 names_t (in module pymule.validator.importer), 156
 namesLen (fortran variable), 57
 nbins (fortran variable), 57
 nel (fortran variable), 53
 nhad (fortran variable), 53
 NLO, 101
 NLP, 101
 nmu (fortran variable), 53
 NNLO, 101
 non-fatal, 103

NotEnoughEnergyError, 154

nq (fortran variable), 57

nr_q (fortran variable), 55

ntau (fortran variable), 53

NTS, 101

ntssoft() (fortran function), 117

ntsSwitch (fortran variable), 54

O

open() (pymule.loader.Loader method), 142

open1() (pymule.loader.FolderLoader method), 140

open1() (pymule.loader.GitLabLoader method), 141

open1() (pymule.loader.Loader method), 142

open1() (pymule.loader.TarLoader method), 143

Options (class in pymule.loader), 142

OS, 102

P

part() (fortran function), 116

particle (fortran type), 115

particles (fortran type), 115

partInterface() (fortran function), 117

parts() (fortran function), 116

pass_cut (fortran variable), 47

PCS, 102

pcSwitch (fortran variable), 54

phi() (fortran function), 51

pi (fortran variable), 54

pick_psgen_func() (pymule.validator.validator.Piece static method), 161

PID, 102

Piece (class in pymule.validator.validator), 159

plusnumbers() (in module pymule.errortools), 132

pol1 (fortran variable), 48

pol2 (fortran variable), 48

printnumber() (in module pymule.errortools), 133

process group, 103

process_names() (pymule.validator.importer.FortranMeasurementFunction method), 156

PSD3() (fortran subroutine), 118

PSD4() (fortran subroutine), 118

PSD4_FKS() (fortran subroutine), 118

PSD5() (fortran subroutine), 118

PSD5_25() (fortran subroutine), 119

PSD5_FKS() (fortran subroutine), 119

PSD6() (fortran subroutine), 119

PSD6_23_24_34() (fortran subroutine), 119

PSD6_23_24_34_E56() (fortran subroutine), 120

PSD6_25_26_m50_FKS() (fortran subroutine), 120

PSD6_26_2x5() (fortran subroutine), 123

PSD6_FKS() (fortran subroutine), 120

PSD6_FKSS() (fortran subroutine), 121

PSD6_P_25_26_m50_FKS() (fortran subroutine), 123

PSD7() (fortran subroutine), 121

PSD7_27_37_47_2x5_FKS() (fortran subroutine), 123

PSD7_27_37_47_E56_FKS() (fortran subroutine), 121

PSD7_27_37_47_FKS() (fortran subroutine), 121

psgen() (in module pymule.validator.psgen), 154

psgen_FKS() (in module pymule.validator.psgen), 155

psgen_X() (in module pymule.validator.psgen), 155

psgen_X_FKS() (in module pymule.validator.psgen), 155

PSX2() (fortran subroutine), 121

PSX3_35_FKS() (fortran subroutine), 122

PSX3_coP13_35_FKS() (fortran subroutine), 124

PSX3_coP_35_FKS() (fortran subroutine), 125

PSX3_coP_45_FKS() (fortran subroutine), 125

PSX3_FKS() (fortran subroutine), 122

PSX3_P13_35_FKS() (fortran subroutine), 123

PSX3_P_15_25_FKS() (fortran subroutine), 124

PSX3_P_15_FKS() (fortran subroutine), 123

PSX3_P_35_FKS() (fortran subroutine), 124

PSX3_P_45_FKS() (fortran subroutine), 124

PSX4() (fortran subroutine), 122

PSX4_35_36_FKSS() (fortran subroutine), 122

PSX4_coP13_35_36_FKSS() (fortran subroutine), 126

PSX4_coP_35_36_FKSS() (fortran subroutine), 125

PSX4_coP_45_46_FKSS() (fortran subroutine), 126

PSX4_FKSS() (fortran subroutine), 122

PSX4_P13_35_36_FKSS() (fortran subroutine), 125

PSX4_P_15_16_25_26_FKSS() (fortran subroutine), 126

PSX4_P_15_16_FKSS() (fortran subroutine), 125

PSX4_P_35_36_FKSS() (fortran subroutine), 125

PSX4_P_45_46_FKSS() (fortran subroutine), 126

pt() (fortran function), 50

pymule.colours

module, 153

pymule.compress

module, 154

pymule.errortools

module, 127

pymule.loader

module, 140

pymule.maths

module, 154

pymule.mpl_axes_aligner

module, 154

pymule.mule

module, 153

pymule.plot

module, 150

pymule.record

module, 134

pymule.validator.importer

module, 156

pymule.validator.psgen

module, 154

pymule.validator.validator
 module, 157
 pymule.vegas
 module, 137
 pymule.xicut
 module, 148

Q

qsoft (fortran variable), 55
 quant() (fortran function), 58

R

random seed, 103
 rap() (fortran function), 50
 read() (pymule.vegas.FileRecord class method), 137
 read_record() (in module pymule.vegas), 139
 real (C type), 55
 real (fortran type), 47
 Record (class in pymule.record), 135
 reset() (pymule.loader.Options method), 143
 rij() (fortran function), 51
 RNG, 102
 RotationValidator (class in py-
 mule.validator.validator), 161
 runmcmule() (fortran subroutine), 52

S

s() (fortran function), 50
 sachs_gel() (fortran function), 55
 sachs_gmag() (fortran function), 55
 SameZeroValidator (class in py-
 mule.validator.validator), 161
 ScalarC0() (fortran function), 118
 ScalarC0_cplx() (fortran function), 118
 ScalarC0IR6() (fortran function), 117
 ScalarC0IR6_cplx() (fortran function), 117
 ScalarD0IR16() (fortran function), 118
 ScalarD0IR16_cplx() (fortran function), 118
 scaleplot() (in module pymule.errortools), 133
 scms (fortran variable), 49
 seeds (pymule.loader.XiRecord property), 145
 set_observable() (fortran subroutine), 53
 setup() (in module pymule.loader), 146
 setup_pgfc() (in module pymule.plot), 151
 SHA1, 102
 shared library, 103
 sigma() (in module pymule.loader), 147
 SM, 102
 soft cut, 103
 softcut (fortran variable), 54
 SoftLimitValidator (class in py-
 mule.validator.validator), 161
 SoftSwapValidator (class in py-
 mule.validator.validator), 162

sq() (fortran function), 50
 sSwitch (fortran variable), 54
 submission script, 103
 sw2 (fortran variable), 48
 SwapSymmetryValidator (class in py-
 mule.validator.validator), 162

T

TarLoader (class in pymule.loader), 143
 the_proton_ff (fortran variable), 49
 threepanel() (in module pymule.plot), 151
 time (pymule.loader.XiRecord property), 145
 time (pymule.record.Record property), 136
 timesnumbers() (in module pymule.errortools), 133
 TOML, 102
 TriangleValidator (class in py-
 mule.validator.validator), 162
 twopanel() (in module pymule.plot), 152

U

uncompress() (in module pymule.compress), 154
 userdim (fortran variable), 57
 userevent() (fortran subroutine), 58
 userfile, 103
 userweight (fortran variable), 48

V

ValidatorBuilder (class in py-
 mule.validator.validator), 162
 value (pymule.record.Record property), 136
 VegasRecord (class in pymule.vegas), 138
 VegasRecordV1 (class in pymule.vegas), 138
 VegasRecordV2 (class in pymule.vegas), 139
 VegasRecordV3 (class in pymule.vegas), 139
 VegasRecordV4 (class in pymule.vegas), 139
 VP, 102

W

watermark() (in module pymule.plot), 153
 which_piece, 103
 which_piece (fortran variable), 54
 write() (pymule.vegas.FoamRecordV4 method), 138
 write() (pymule.vegas.VegasRecordV4 method), 139
 write_record() (in module pymule.vegas), 140

X

xic (pymule.loader.XiRecord property), 145
 xicut1 (fortran variable), 54
 xicut2 (fortran variable), 54
 xieik1 (fortran variable), 54
 xiout (fortran variable), 54
 XiRecord (class in pymule.loader), 143
 xiresidue() (in module pymule.xicut), 149

Y

`yaxes()` (in module `pymule.mpl_axes_aligner`), 154

`yout` (fortran variable), 54

Z

`zero` (fortran variable), 54