# McMule

*Release v0.5.1*

**The McMule Collaboration**

**Jan 03, 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 Modular structure of the code . . . 21
- 2.2 What happens when running . . . 23

**3 General aspects of using McMule** **25**
- 3.1 Statistics . . . 25
- 3.2 Analysis . . . 26
- 3.3 Manual compilation . . . 27

**4 Technical aspects of McMule** **29**
- 4.1 Phase-space generation . . . 29
- 4.2 Implementation of FKS schemes . . . 32
- 4.3 Calling procedures and function pointers . . . 34
- 4.4 Optional parameters for integrands . . . 34
- 4.5 Random number generation . . . 36
- 4.6 Differential distributions and intermediary state files . . . 37
- 4.7 Basics of containerisation . . . 38

**5 Implementing new processes in McMule** **41**
- 5.1 Creating a new process group . . . 46
- 5.2 Study of $\xi_c$ dependence . . . 48
- 5.3 Example calculations in Mathematica . . . 48
- 5.4 Coding style and best practice . . . 50

**6 The FKS$^2$ scheme** **51**
- 6.1 FKS$^\ell$: extension to N$^\ell$LO . . . 52

**7 Glossary** **53**
- 7.1 Acronyms . . . 53
- 7.2 Technical terms . . . 54

**8 Bibliography** **57**

**9 Particle ID** **59**

**10 Available processes and `which_piece`** **63**

**11 Fortran reference guide** **65**
11.1 User-modifiable parameters . . . 65
11.2 Technical parameters . . . 67
11.3 User-facing functions . . . 67
11.4 The user file . . . 70
11.5 Technical routines . . . 72

**12 pymule user guide** **85**
12.1 Working with files . . . 85
12.2 Working with errors . . . 87
12.3 Plotting . . . 92

**13 pymule reference guide** **97**
13.1 Working with errors . . . 97
13.2 Working with abstract records . . . 103
13.3 Working with vegas records . . . 103
13.4 Working with records of data . . . 106
13.5 Working with $\xi_c$ data . . . 111
13.6 Working with plots . . . 113
13.7 Useful other functions . . . 117

**14 Indices and tables** **119**

**Bibliography** **121**

**Python Module Index** **123**

**Index** **125**

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]

[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

# 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 $\tau \to [\nu\bar{\nu}]e\gamma$. 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 $3\,\sigma$ 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 $E_\gamma \geq 10\,\mathrm{MeV}$ as used in [14, 18].

At first, we need to find out how the process $\tau \to \nu\bar\nu e\gamma$ 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 processes*. The generic process is a prototype for the physical process such as $\ell \to \nu\bar\nu\ell'\gamma$ where the flavour of the lepton $\ell$ is left open. In our case, we need to consider the row for $\mu \to \nu\bar\nu e\gamma$. Since we are only interested in *LO*, the only `which_piece` we need is `m2enng0`. To change from the generic process $\mu \to \nu\bar\nu e\gamma$ to the process we are actually interested in, $\tau \to \nu\bar\nu e\gamma$, we pick the `flavour tau-e` which refers to a $\tau \to e\cdots$ transition. Other options would be `tau-mu` for $\tau \to \mu\cdots$‘ or `mu-e` for $\mu \to e\cdots$.

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

$$\mu^-(p_1) \to e^-(p_2)\,[\bar\nu_e\nu_\mu](p_3,p_4)\gamma(p_5)$$

We can now implement our observables. For this, we need to define a `user.f95` file in the `src` folder. An empty template can be found in the file tools/user-empty.f95. We can use this file to 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 $0 < m_{\gamma e} < 1800\,\mathrm{MeV}$ and $0 \leq E_e \leq 900\,\mathrm{MeV}$.

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 $0 < m_{\gamma e} < 1800\,\mathrm{MeV}$ and $0 \leq E_e \leq 900\,\mathrm{MeV}$.

```
12  real:: &
13     min_val(nrq) = (/    0.,    0. /)
14  real:: &
15     max_val(nrq) = (/ 1800., 900. /)
16  integer :: userdim = 0
```

---

**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*

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

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

> **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!

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
```

We now need to re-compile McMule to ensure that we have the correct version of `user.f95`.

> **Warning:** The `mcmule.mod` header file is autogenerated by GFortran during the compilation 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 $1000 \times 10^3$ points each for pre-conditioning and 50 iterations with $1000 \times 10^3$ points each for the actual numerical evaluation (cf. Section *Statistics* 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. The `flavour` variable is now set to `tau-e` to change from the generic process $\mu \to \nu\bar{\nu}e\gamma$ to the process we are actually interested in, $\tau \to \nu\bar{\nu}e\gamma$. 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.). In a second step, the input `flavour` associates actual numbers to the parameters entering the matrix elements and phase-space generation. 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
tau-e


         *******************************************
         *              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              *
         *                                         *
         *******************************************

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

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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: 1.00000    delcut: 1.00000
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 |
|---|---|---|
| `nenter_ad` | `integer` | calls / iteration during pre-conditioning |
| `itmx_ad` | `integer` | iterations during pre-conditioning |
| `nenter` | `integer` | calls / iteration during main run |
| `itmx` | `integer` | iterations during main run |
| `ran_seed` | `integer` | *random seed* $z_1$ |
| `xinormcut` | `real(prec)` | the $0 < \xi_c \leq 1$ parameter |
| `delcut` | `real(prec)` | the $\delta_{\text{cut}}$ parameter (or at *NNLO* the second $\xi_c$) |
| `which_piece` | `char(10)` | the part of the calculation to perform |
| `flavour` | `char(8)` | the particles involved |
| (opt) | unknown | the user can request further input during `inituser()` |

### 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 `importvegas()`, 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()`.

```
from pymule import *

lifetime = 1/(1000*(6.582119e-25)/(2.903e-13))
# define vegas directory
setup(folder="example1/")
dat = scaleset(
    mergefks(sigma("m2enng0")),
    GF**2*alpha*lifetime
)

dat.keys()
# dict_keys(['time', 'chi2a', 'value', 'Ee', 'minv'])
```

**Warning:** In McMule the numerical value of the Fermi constant $G_F$ and the fine-structure constant $\alpha$ are set to one for predominately historical reasons. This needs to be restored in python, eg. using `scaleset()`

The variable `dat` now contains the runtime (`time`), branching ratio (after multiplication with the lifetime, `value`), and $\chi^2$ 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`

```
from matplotlib.pyplot import *
fig = plt.figure()
errorband(dat['Ee'])
plt.ylabel(r'$\D\mathcal{B}/\D E_e$')
plt.xlabel(r'$E_e\,/\,{\rm MeV}$')
mulify(fig)
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 is 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 $E_\gamma > 10\,\mathrm{MeV}$. 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 $10\,\mathrm{MeV}$. 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 1.3: The measurement function beyond *LO*. The changes w.r.t. to *LO* are highlighted.

```
62  FUNCTION QUANT(q1,q2,q3,q4,q5,q6,q7)
63
64  real (kind=prec), intent(in) :: q1(4),q2(4),q3(4),q4(4), q5(4),q6(4),q7(4)
65  real (kind=prec) :: quant(nr_q)
66  real (kind=prec) :: gs(4), gh(4)
67  !! ==== keep the line below in any case ==== !!
68  call fix_mu
69
70  if (q5(4) > q6(4)) then
71    gh = q5 ; gs = q6
72  else
73    gh = q6 ; gs = q5
74  endif
75
76  if (gh(4) < 10.) pass_cut = .false.
77  if (gs(4) > 10.) pass_cut = .false.
78
79  names(1) = 'minv'
80  quant(1) = sqrt(sq(q2+qh))
81
82  names(2) = 'Ee'
83  quant(2) = q2(4)
84
85  END FUNCTION QUANT
```

### 1.3.2 Running McMule

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

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

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 $\xi_c$ 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 $\xi_c$ 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 1.4: 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
Which flavour combination? [mu-e] tau-e
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 \
      --flavour tau-e \
      --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 1.5 and Listing 1.6

[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 1.5: *menu file* for the present calculation

```
## Generated at 16:00 on February 28 2020 by yannickulrich
# git version: redesign (b558978)

conf example2/m2enng-tau-e.conf

run 19397 1.000000 m2enng0 tau-e 0
run 52088 1.000000 m2enng0 tau-e 0
run 83215 1.000000 m2enng0 tau-e 0
run 93857 1.000000 m2enng0 tau-e 0
run 86361 1.000000 m2enng0 tau-e 0


run 19397 1.000000 m2enngV tau-e 0
run 52088 1.000000 m2enngV tau-e 0
run 83215 1.000000 m2enngV tau-e 0
run 93857 1.000000 m2enngV tau-e 0
run 86361 1.000000 m2enngV tau-e 0


run 19397 0.500000 m2enngC tau-e 0
run 52088 0.500000 m2enngC tau-e 0
run 83215 0.500000 m2enngC tau-e 0
run 93857 0.500000 m2enngC tau-e 0
run 86361 0.500000 m2enngC tau-e 0
run 19397 0.500000 m2enngR tau-e 0
run 52088 0.500000 m2enngR tau-e 0
run 83215 0.500000 m2enngR tau-e 0
run 93857 0.500000 m2enngR tau-e 0
run 86361 0.500000 m2enngR tau-e 0

run 19397 0.250000 m2enngC tau-e 0
run 52088 0.250000 m2enngC tau-e 0
run 83215 0.250000 m2enngC tau-e 0
run 93857 0.250000 m2enngC tau-e 0
run 86361 0.250000 m2enngC tau-e 0
run 19397 0.250000 m2enngR tau-e 0
run 52088 0.250000 m2enngR tau-e 0
run 83215 0.250000 m2enngR tau-e 0
run 93857 0.250000 m2enngR tau-e 0
run 86361 0.250000 m2enngR tau-e 0

run 19397 0.125000 m2enngC tau-e 0
run 52088 0.125000 m2enngC tau-e 0
run 83215 0.125000 m2enngC tau-e 0
run 93857 0.125000 m2enngC tau-e 0
run 86361 0.125000 m2enngC tau-e 0
run 19397 0.125000 m2enngR tau-e 0
run 52088 0.125000 m2enngR tau-e 0
run 83215 0.125000 m2enngR tau-e 0
run 93857 0.125000 m2enngR tau-e 0
run 86361 0.125000 m2enngR tau-e 0
```

Listing 1.6: Configuration file for the present calculation

```
## Generated at 16:00 on February 28 2020 by yannickulrich
# git version: redesign (b558978)

# specify the program to run relative to `pwd`
binary=mcmule

# specify the output folder
folder=example2/

# Specify the variables nenter_ad, itmx_ad, nenter and itmx
# for each piece you want to run.
declare -A STAT=(
  ["m2enng0"]="1000\n10\n1000\n50"
  ["m2enngC"]="1000\n10\n1000\n50"
  ["m2enngR"]="5000\n50\n10000\n100"
  ["m2enngV"]="1000\n10\n1000\n50"
)
```

To start `mcmule`, we now just need to execute the created `example2/submit.sh` after copying the user library `user.so` into the same folder. 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

```
#SBATCH --ntasks=<number of cores>
```

To now run McMule, just execute

```
$ nohup ./example2/submit.sh &
```

The `nohup` is not technically necessary but advisable, especially on remote systems. When running on a SLURM system, the other SLURM parameters `--partition`, `--time`, and `--clusters` need to be adapted as well.

> **Warning:** The *submission script* will call itself on multiple occasions. Therefore, it is not advisable to change its name or the name of the run directory without taking precautions.

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

```
from pymule import *

# 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
```

(continues on next page)

(continued from previous page)

```
setup(folder='example2/out.tar.bz2')

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

# Import NLO corrections from the three pieces
NLO = scaleset(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 = plusnumbers(LO['value'], NLO['value'])

# 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"$\D\mathcal{B}/\D E_e$"
)
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"$\D\mathcal{B}/\D m_{e\gamma}$"
)
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*. `pymule`mergefks()` takes the results of (multiple) `sigma()` invocations, adds results with matching $\xi_c$ 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 1.7) to calculate the full *NLO BR*s from the corrections and the *LO* results

$$\mathcal{B}|_{\text{LO}} = 1.8339(1) \times 10^{-2}$$
$$\mathcal{B}|_{\text{NLO}} = 1.6451(1) \times 10^{-2}$$

which agree with [10, 21], but $\mathcal{B}|_{\text{NLO}}$ 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 $0 \leq m_{e\gamma} \leq 1\,\mathrm{GeV}$. 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 $\tau \to \nu\bar{\nu}e\gamma$. 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

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

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, $E_e \sim 900\,\mathrm{MeV}$, the statistics is out of control.

Figure 1.3: Results of the toy run to compute $E_e$ for $\tau \to \nu\bar{\nu}e\gamma$. 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 $e^+e^- \to \tau^+\tau^-$, we instead use the LO cross section

$$\frac{\mathrm{d}\sigma}{\mathrm{d}(\cos\theta)} \propto \Big(1 + \frac{4m_\tau^2}{s}\Big) + \Big(1 + \frac{4m_\tau^2}{s}\Big)\cos\theta \tag{1.1}$$

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

To accurately simulate this situation, we need to

- choose a random value for $\theta$,
- 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.

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

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

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 $\cos\theta$

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

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

### 1.4.3 Boosting into the lab frame

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

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

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.

```
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) :: ptau, cos_e, cos_g
real (kind=prec) :: p1Lab(4), p2Lab(4), p5Lab(4), p6Lab(4)
real (kind=prec) :: quant(nr_q)
real (kind=prec) :: gs(4), gh(4)
real (kind=prec), parameter :: ez(4) = (/ 0., 0., 1., 0. /)
!! ==== keep the line below in any case ==== !!
call fix_mu
pass_cut = .true.

ptau = sqrt(Etau**2-Mtau**2)
p1Lab = (/ 0., ptau*sqrt(1-cth**2), ptau*cth, Etau /)

p1Lab = boost_back(p1Lab, q1)
p2Lab = boost_back(p1Lab, q2)
p5Lab = boost_back(p1Lab, q5)
p6Lab = boost_back(p1Lab, q6)

if (p5Lab(4) > p6Lab(4)) then
  gh = p5Lab ; gs = p6Lab
else
  gh = p6Lab ; gs = p5Lab
endif

cos_e = cos_th(p2Lab, ez)
cos_g = cos_th(gh   , ez)

if ( (cos_e > 0.95 .or. cos_e < -0.75) .or. (cos_g > 0.95 .or. cos_g < -0.75) ) then
  pass_cut = .false.
  return
endif

if(exclusiveQ == 1) then
  if (gh(4) < 220. .or. gs(4) > 220.) then
    pass_cut = .false.
    return
  endif
else
  if (gh(4) < 220.) then
    pass_cut = .false.
    return
```

(continues on next page)

(continued from previous page)

```
116      endif
117    endif
118
119    names(1) = 'minv'
120    quant(1) = sqrt(sq(q2+gh))
121
122    names(2) = 'Ee'
123    quant(2) = p2Lab(4)
124
125    names(3) = 'cos_e'
126    quant(3) = cos_e
127
128    names(4) = 'cos_g'
129    quant(4) = cos_g
130    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 1.8[2]

Listing 1.8: 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 tau-e 0
run 52506 1.000000 m2enng0 tau-e 0
run 22671 1.000000 m2enng0 tau-e 0

run 53796 1.000000 m2enngV tau-e 0
run 15282 1.000000 m2enngV tau-e 0
run 89444 1.000000 m2enngV tau-e 0

run 98870 0.600000 m2enngC tau-e 0
run 91991 0.600000 m2enngC tau-e 0
run 79769 0.600000 m2enngC tau-e 0

run 21175 0.800000 m2enngC tau-e 0
run 57581 0.800000 m2enngC tau-e 0
run 81929 0.800000 m2enngC tau-e 0

run 70604 0.600000 m2enngR tau-e 0
run 33013 0.600000 m2enngR tau-e 0
run 22530 0.600000 m2enngR tau-e 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 $\xi_c$, 0.6 and 0.8.

(continued from previous page)

```
run 82222 0.800000 m2enngR tau-e 0
run 30935 0.800000 m2enngR tau-e 0
run 40689 0.800000 m2enngR tau-e 0


run 70604 0.600000 m2enngR tau-e 1
run 33013 0.600000 m2enngR tau-e 1
run 22530 0.600000 m2enngR tau-e 1

run 82222 0.800000 m2enngR tau-e 1
run 30935 0.800000 m2enngR tau-e 1
run 40689 0.800000 m2enngR tau-e 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 :numref:fig_Eboost:

Listing 1.9: The analysis pipeline for this calculation

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

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

NLOincl = scaleset(mergefks(
    sigma('m2enngR', obs='i'),
    sigma('m2enngC'),
    anyxi=sigma('m2enngV')
), GF**2*lifetime*alpha**2)
fullNLOincl = addsets([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"$\D\mathcal{B}/\D E_{e}$",
    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 `python`[1]. 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.

## 2.1 Modular structure of the code

McMule consists of several modules with a simple, mostly hierarchic structure. 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.[2] 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 $\xi_1$ 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 `integrands`. Also the variable `ksoft1` is provided that corresponds to the photon momentum without the (vanishing) energy factor $\xi_1$. Routines ending with `FKSS` are routines with two unresolved photons. Correspondingly, a factor $\xi_1\,\xi_2$ is missing in the weight and `xiout1` and `xiout2`, as well as `ksoft1` and `ksoft2`

[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.

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 $\mu$-$e$ and $\mu$-$p$ 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 `ksoft` 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

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

**user:**
For a user of the code who wants to run for an already implemented process, this is the only relevant module. At the beginning of the module, the user has to specify the number of quantities to be computed, `nr_q`, the number of bins in the histogram, `nr_bins`, as well as their lower and upper boundaries, `min_val` and `max_val`. The last three quantities are arrays of length `nr_q`. The quantities themselves, i.e. the measurement function, is to be defined by the user in terms of the momenta of the particles in `quant()`. Cuts can be applied by setting the logical variable `pass_cut` to false[3]. Some auxiliary functions like (pseudo)rapidity, transverse momentum etc. are predefined in `functions`. Each quantity has to be given a name through the array `names`.

Further, `user` contains a subroutine called `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.

**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 (`nr_bins` and `nr_q`, respectively) as well as their bounds (`min_val` and `max_val`) and names, from `user`.

**integrands:**
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 FKS$^2$ 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 $\xi_i$ that were omitted in the phase-space weight have to be included here for the single- and double-subtracted integrands.

**mcmule:**
This is the main program, but actually does little else than read the inputs and call `vegas` with a function provided by `integrands`.

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

`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.

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 $\ell p \to \ell p$ where the flavour of the lepton $\ell$ 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 (6.1) (or correspondingly (6.7) 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).

## 2.2 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*.

1. When started, `mcmule` reads options from `stdin` as specified in Table 1.1.
2. Once McMule knows its configuration it associates the numerical values of the masses, as specified through `flavour`. In particular, we set the generic masses `Mm` and `Me` to `Mtau` and `Mel`. This is done in `init_flavour()`, 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 `integrands`. 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 $\mu(q_1, n_1) \to [\nu(q3)\bar\nu(q4)]e(q_2)\gamma(q_5)$ 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 `quant()`. If one of them passes the cuts, the variable `cuts` 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 `min_val` and `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 encountered 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

# 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 *Statistics*. This is followed by a more in-depth discussion of the analysis strategy in Section *Analysis*.

## 3.1 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 (`nenter`). 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 $i_{\text{ad}} \times n_{\text{ad}}$ (`nenter_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 $n_{\text{ad}}$ 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 $\chi^2/\mathrm{d.o.f.}$ indicates a too small $n$. Try to increase $n_\mathrm{ad}$ 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 $\xi_c$ 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.

## 3.2 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 $\xi_c$. The $\chi^2/\mathrm{d.o.f.}$ 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 (6.11). This includes any partitioning done in Section *Phase-space generation*.
3. (optional) At N$^\ell$LO, perform a fit[1]

$$\sigma^{(\ell)}_{n+j} = c_0^{(j)} + c_1^{(j)} \log\xi_c + c_2^{(j)} \log^2\xi_c + \cdots + c_\ell^{(j)} \log^\ell = \sum_{i=0}^{\ell} c_i^{(j)} \log^i \xi_c \tag{3.1}$$

   This has the advantage that it very clearly quantifies any residual $\xi_c$ dependence. We will come back to this issue in Section *Study of \xi_{c} dependence*.

4. Combine all physical contributions of (6.10) into $\sigma^{(\ell)}(\xi_c)$ which has to be $\xi_c$ independent.
5. Perform detailed checks on $\xi_c$ 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 (3.1) is compatible with a constant, i.e. whether for all $1 \le i \le \ell$

$$\left| \frac{\sum_{j=0}^{\ell} c_i^{(j)}}{\sum_{j=0}^{\ell} \delta c_i^{(j)}} \right| < 1 \tag{3.2}$$

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

   If (3.2) is not satisfied or only very poorly, try to run the Monte Carlo again with an increased $n$.

[1] Note that it is important to perform the fit after combining the phase-space partitionings (cf. Section *Phase-space generation*) but before adding (6.10) as this model is only valid for the terms of (6.11)

[2] Note that the error estimate on the sum of the total coefficients in (3.2) is rather poor and does not include correlations between different $c_i$.

6. Merge the different estimates of (6.10) from the different $\xi_c$ into one final number $\sigma^{(\ell)}$. The $\chi^2/\mathrm{d.o.f.}$ 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 $\xi_c$ 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*).

## 3.3 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
```

### 3.3.1 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 babar-tau-e/m2enng-tau-e.conf

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 4

# 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 FKS^2 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*.

## 4.1 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 4.1.

Listing 4.1: 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
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
```

(continues on next page)

(continued from previous page)

```
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 $\xi_i$ to be variables of the integration. We can fix this by first generating the photon explicitly as

$$k_1 = p_{n+1} = \frac{\sqrt{s}}{2}\xi_1(1, \sqrt{1-y_1^2}\vec{e}_\perp, y_1) \tag{4.1}$$

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

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

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 $\mu \to \nu\bar\nu e\gamma$. 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 $\beta$ (or equivalently small masses), this becomes almost singular as $y \to 1$. If now $y$ is a variable of the integration this can be mediated. An example implementation is shown in Listing 4.2.

Listing 4.2: Example implementation of a so-called *FKS* phase-space where the fifth particle is an *FKS* photon that may becomes soft. Not shown are checks whether $E_i \ge m_i$ .

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

! generate electron q2 and photon q5 s.t. that the
```

(continues on next page)

(continued from previous page)

```
! photon goes into z diractions

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 ommited
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 $\delta$ 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 $\mu$-$e$-scattering

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

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 (4.1). The outgoing electron $p_3$ is more complicated. Writing the $p_4$-phase-space four- instead of three-dimensional

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

we can solve the four-dimensional $\delta$ 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 $\delta$ function

$$\delta(p_4^2-M^2) = \delta\Big(m^2 - M^2 + s(1-\xi) + E_3\sqrt{s}\Big[\xi - 2 - y\xi\beta_3(E_3)\Big]\Big)\,. \tag{4.2}$$

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

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

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 that 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 $\mu$-$e$ scattering for instance is split as follows[1]

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

with $s_{ij} = 2p_i \cdot p_j$ as usual. The integrand of the first $\theta$ function has a final-state *PCS* and hence we use the parametrisation obtained by solving (4.2). The second $\theta$ 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 (4.1). This automatically makes $s_{15} \propto (1 - \beta_{\text{in}} y_1)$ a variable of the integration.

For the double-real corrections of $\mu$-$e$ scattering, we proceed along the same lines except now the argument of the $\delta$ function is more complicated.

## 4.2 Implementation of FKS schemes

Now that we have a phase-space routine that has $\xi_i$ as variables of the integration, we can start implementing the relevant $c$-distributions (6.4)

$$\mathrm{d}\sigma_h^{(1)}(\xi_c) = \mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1} \left(\frac{1}{\xi_1}\right)_c \mathrm{d}\xi_1 \Big(\xi_1^2 \mathcal{M}_{n+1}^{(0)}\Big) \tag{4.3}$$

$$= \mathrm{d}\xi_1 \Bigg(\mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1} \Big(\xi_1^2 \mathcal{M}_{n+1}^{(0)}\Big) - \mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1} \Big(\mathcal{E} \mathcal{M}_n^{(0)}\Big) \theta(\xi_c - \xi_1)\Bigg) \tag{4.4}$$

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 $\mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1}$ 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 $\xi$ and sometimes also on a parameter that encodes the *PCS* such as $y = $ `y2` in (4.1) and Listing 4.2. Events that have values of $\xi$ 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 4.3.

[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 4.3: 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 is 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
```

## 4.3 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', pm2enngav)
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 4.1: 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`.

## 4.4 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 4.2: Frozen Delights!

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

### 4.4.1 $\xi_c$ parameters

For the $\xi_c$ parameters, the user enters a value between zero (exclusive) and one (inclusive). However, the *FKS* procedure requires the bounds of (6.5) 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.

### 4.4.2 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 (4.2)

$$\mathrm{d}\sigma_h^{(1)}(\xi_c) \to \mathrm{d}\xi_1 \times \theta(\xi - \texttt{softcut}) \tag{4.5}$$

$$\times \left( \mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1} \Big( \xi_1^2 \mathcal{M}_{n+1}^{(0)} \Big) - \mathrm{d}\Upsilon_1 \mathrm{d}\Phi_{n,1} \Big( \mathcal{E}\mathcal{M}_n^{(0)} \Big) \theta(\xi_c - \xi_1) \right) \tag{4.6}$$

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

## 4.5 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. *RNG*s 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

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

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 period[2] $p < m$ 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 Theorem[3]. 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 smaller[4].

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.

## 4.6 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 array[5] *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 $\mathrm{d}\Phi \times \mathcal{M} \times w$ is added to the correct entry in `bit_it` where $w = \texttt{wgt}$ is the event weight assigned by `vegas`.

After each iteration of `vegas` we add *quant()* (`quant`$^2$) to an accumulator of the same dimensions called `quantsum` (`quantsumsq`). After $i$ iterations, we can calculate the value and error as

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

where $\Delta$ 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 4.3. The latter includes file version information and the first five characters the source tree's *SHA1* hash, obtained using `make hash`.

The header is followed by records describing the state of the integrator as shown in Table 4.4. 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 4.3: The magic header and version information used by $v_3$. $v_1$ 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 4.4: 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` | $(1-\texttt{it})\chi+\sigma^2/(\delta\sigma)^2$ |
| `0078` | `1A98` | `real(50,17)` | `xi` | the integration grid |
| `1B10` | `000C` | `integer` | `randy` | the current *random number seed* |
| `1B1C` | `0014` | `integer` `integer` `integer` | $n_q$ $n_b$ $n_s$ | number of histograms number of bins len. histogram name |
| `1B30` | $10n_q+8$ $n_sn_q+8$ $10n_q(n_b+2)+8$ | `real(n_q)` `real(n_q)` `character(n_s,n_q)` `real(n_q,n_b+2)` `real(n_q, n_b+2)` | `minv` `maxv` `names` `quantsum` `quantsumsq` | lower bounds upper bounds names of $S$ accu. histograms accu. histograms squared |
| `-014` | `0010` | `real` | `time` | current runtime in seconds |
| `-013` | `0134` | `character(300)` | `msg` | any message |
| `-000` | `EOF` | | | |

## 4.7 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.

### 4.7.1 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 an `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*.

### 4.7.2 Building images

Docker images are built using Dockerfiles, a set of instruction 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 build

```
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.

### 4.7.3 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 5

# 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 $e^-e^- \to e^-e^-$ 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, $ee \to ee$ 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 $\texttt{s}(\texttt{pi},\texttt{pj}) = 2\texttt{pi}\cdot\texttt{pj}$, 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_el.f95` will contain $\mathcal{M}_n^{(0)}$. For example, $\mathcal{M}_n^{(0)}$ is implemented there as shown in Listing 5.1.

Listing 5.1: 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(p4), 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 also contains the soft limits. In our case this is called `ee/ee.f95`.

   The abstract interface `partInterface` (cf. Section *Technical routines*) can take care of the generation of all soft limits for a given `particle` string, as shown in Listing 5.2.[1] See also Section *Calling procedures and function pointers* for more details.

Listing 5.2: 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(p4), 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_el.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_ee2eel.f95`, which defines a function

```
FUNCTION EE2EEl(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(p4), 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 $e\mu \to e\mu$ 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`.

7. 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 5.3.

Listing 5.3: Test routine for $ee \to ee$ 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")
scms = 40000.
musq = me
x = (/0.75,0.5/)
call ps_x2(x,scms,p1,me,p2,me,p3,me,p4,me,weight)
call check("ee2ee" ,ee2ee (p1,p2,p3,p4), 2.273983244890001e4, threshold=2e-8)
call check("ee2eel",ee2eel(p1,p2,p3,p4), 6.964297070440638e7, threshold=2e-8)

scms = 40000.
y = (/0.3,0.6,0.8,0.4,0.9/)
call ps_x3_fks(y,scms,p1,me,p2,me,p3,me,p4,me,p5,weight)
call check("ee2eeg",ee2eeg(p1,p2,p3,p4,p5),7.864297444955537e2, threshold=2e-8)

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
```

8. 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 $2 \to 2$ 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 $\xi_c$ is properly normalised. The user will enter a value from 0 to 1 which needs to be matched to $\xi_c$ as defined in (6.5)

    - 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)
    ```

(continues on next page)

(continued from previous page)

```
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 $e^-e^- \to e^-e^-$ 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 $\xi_c$ independence studies. Tips on how to do this can be found in Section *Study of \xi_{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 $\xi_c$ and *soft cut* analyses.

In the following we comment on a few aspects of this procedure such as the $\xi_c$ study (Section *Study of \xi_{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*).

## 5.1 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 makefile (Listing 5.4), a `mat_el` file (`ee/ee_mat_el.f95`, Listing 5.5) and a module file (`ee/ee.f95`, Listing 5.6). Finally, the name of the *process group* needs to be added to the `GROUPS` and `WGROUPS` variables of the makefile.

Listing 5.4: The bare makefile for the new *process group* `ee`. Large matrix elements that are stored in extra files such as `ee/ee2eeg.f95` or `ee/ee_ee2eel.f95` need to be added to the list of `AUXFILES`

```
group=ee
AUXFILES=ee_ee2eel.f95 ee_ee2eeg.f95

MAIN=$(group)_mat_el.f95 $(group).f95


include ../makefile.conf


all: $(group).a $(group).mod .obj/tree.sha

$(OBJ): ../.obj/functions.mod

.obj/$(group)_mat_el.o .obj/$(group)_mat_el.mod: \
    $(group)_mat_el.f95 $(MOD)
.obj/$(group).o .obj/$(group).mod: \
    $(group).f95 .obj/$(group)_mat_el.mod $(MOD)

$(group).mod: .obj/$(group).mod
    /+cp+/ $< $@

$(group).a:$(OBJ)
    @/+echo AR+/ $@
    @$(AR) $@ $^

clean:
        rm -f .obj/*.o .obj/*.mod .obj/*.gcda .obj/*.gcno *.mod *.a
```

Listing 5.5: 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
  use ee_ee2eeg, only: ee2eeg
  implicit none
```

(continues on next page)

(continued from previous page)

```
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,p1,p2) + Ieik(xieik1,Epart,p1,q1) &
    + Ieik(xieik1,Epart,p1,q2) + Ieik(xieik1,Epart,p2,q1) &
    + Ieik(xieik1,Epart,p2,q2) - Ieik(xieik1,Epart,q1,q2))

END FUNCTION EE2EEF
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!
                      END MODULE EE_MAT_EL
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Listing 5.6: 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

                !!!!!!!!!!!!!!!!!!!!!!!!!!!!
                      END MODULE EE
                !!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

## 5.2 Study of $\xi_c$ dependence

When performing calculations with McMule, we need to check that the dependence of the unphysical $\xi_c$ parameter introduced in the *FKS* scheme (cf. Appendix *The FKS^2 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 $\xi_c$) for production runs.

Because the $\xi_c$ dependence is induced through terms as $\xi_c^{-2\epsilon}/\epsilon$, we know the functional dependence of $\sigma^{(\ell)}_{n+j}$. For example, at *NLO* we have

$$\sigma^{(1)}_n(\xi_c) = a_{0,0} + a_{0,1}\log(\xi_c) \tag{5.1}$$

$$\sigma^{(1)}_{n+1}(\xi_c) = a_{1,0} + a_{1,1}\log(\xi_c) \tag{5.2}$$

where $\xi_c$ independence of $\sigma^{(1)}$ of course requires

$$a_{0,1} + a_{1,1} = 0 \tag{5.3}$$

At *NNLO* we have

$$\sigma^{(2)}_n(\xi_c) = a_{0,0} + a_{0,1}\log(\xi_c) + a_{0,2}\log(\xi_c)^2\,, \tag{5.4}$$

$$\sigma^{(2)}_{n+1}(\xi_c) = a_{1,0} + a_{1,1}\log(\xi_c) + a_{1,2}\log(\xi_c)^2\,, \tag{5.5}$$

$$\sigma^{(2)}_{n+2}(\xi_c) = a_{2,0} + a_{2,1}\log(\xi_c) + a_{2,2}\log(\xi_c)^2\,. \tag{5.6}$$

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 $\sigma^{(\ell)}_{n+j}$. Still, knowing the coefficients can be extremely helpful when debugging the code or to just quantify how well the $\xi_c$ 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, $\sigma^{(2)}_{n+2}(\xi_c)$ is much worse than the one for $\sigma^{(2)}_{n+1}(\xi_c)$, a problem in the double-real corrections can be expected.

## 5.3 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 $\mu\to\nu\bar\nu e\gamma$ can be found in Listing 5.7. 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 5.7: An example on how to calculate the renormalised one-loop matrix element for $\mu\to\nu\bar\nu e$ 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
}]];

M1bare = Simplify[KallenExpand[LoopRefine[LoopRelease[
  Pro2LoopIntegrate[
    Coefficient[M1, μ2, 0]/(16 π^2)
  ]
  + μIntegrate[
    Coefficient[M1, μ2, 1]/(64 π^3),
    1
  ],
  onshell
]]] /. e -> Sqrt[4 πα]];
```

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.

## 5.4 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`.
- Integrands 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 6

# The FKS$^2$ 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

$$\sigma^{(1)} = \sigma^{(1)}_n(\xi_c) + \sigma^{(1)}_{n+1}(\xi_c) \tag{6.1}$$

$$\sigma^{(1)}_n(\xi_c) = \int \mathrm{d}\Phi^{d=4}_n \left(\mathcal{M}^{(1)}_n + \hat{\mathcal{E}}(\xi_c)\,\mathcal{M}^{(0)}_n\right) = \int \mathrm{d}\Phi^{d=4}_n\,\mathcal{M}^{(1)f}_n \tag{6.2}$$

$$\sigma^{(1)}_{n+1}(\xi_c) = \int \mathrm{d}\Phi^{d=4}_{n+1}\left(\frac{1}{\xi_1}\right)_c \left(\xi_1\,\mathcal{M}^{(0)f}_{n+1}\right) \tag{6.3}$$

In (6.3), $\xi_1$ is a variable of the $(n+1)$ parton phase space $\mathrm{d}\Phi^{d=4}_{n+1}$ that corresponds to the (scaled) energy of the emitted photon. For $\xi_1 \to 0$ the real matrix element $\mathcal{M}^{(0)f}_{n+1}$ 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 $\xi_1$ 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

$$\int_0^1 \mathrm{d}\xi_1 \left(\frac{1}{\xi_1}\right)_c f(\xi_1) \equiv \int_0^1 \mathrm{d}\xi_1\,\frac{f(\xi_1) - f(0)\theta(\xi_c-\xi_1)}{\xi_1} \tag{6.4}$$

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 $\xi_1$. This results in the integrated eikonal factor $\hat{\mathcal{E}}(\xi_c)$ times the tree-level matrix element for the $n$ particle process, $\mathcal{M}^{(0)}_n$. The factor $\hat{\mathcal{E}}(\xi_c)$ has an explicit $1/\epsilon$ pole that cancels precisely the corresponding pole in the virtual matrix element $\mathcal{M}^{(1)}_n$. Thus, the combined integrand of (6.2) is free of explicit poles, hence denoted by $\mathcal{M}^{(1)f}_n$, and can be integrated numerically over the $n$ particle phase space $\mathrm{d}\Phi^{d=4}_n$.

The parameter $\xi_c$ that has been introduced to split the real corrections can be chosen arbitrarily as long as

$$0 < \xi_c \le \xi_{\text{max}} = 1 - \frac{\left(\sum_i m_i\right)^2}{s} \tag{6.5}$$

where the sum is over all masses in the final state. The $\xi_c$ dependence has to cancel exactly between (6.2) and (6.3) 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

$$e^{\hat{\mathcal{E}}} \sum_{\ell=0}^{\infty} \mathcal{M}_n^{(\ell)} = \sum_{\ell=0}^{\infty} \mathcal{M}_n^{(\ell)f} = \mathcal{M}_n^{(0)} + \Big(\mathcal{M}_n^{(1)} + \hat{\mathcal{E}}(\xi_c)\,\mathcal{M}_n^{(0)}\Big) + \mathcal{O}(\alpha^2) \tag{6.6}$$

where we exploited the implicit factor $\alpha$ 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

$$\sigma_n^{(2)}(\xi_c) = \int \mathrm{d}\Phi_n^{d=4} \left(\mathcal{M}_n^{(2)} + \hat{\mathcal{E}}(\xi_c)\,\mathcal{M}_n^{(1)} + \frac{1}{2!}\mathcal{M}_n^{(0)}\hat{\mathcal{E}}(\xi_c)^2\right) = \int \mathrm{d}\Phi_n^{d=4}\,\mathcal{M}_n^{(2)f} \tag{6.7}$$

$$\sigma_{n+1}^{(2)}(\xi_c) = \int \mathrm{d}\Phi_{n+1}^{d=4} \left(\frac{1}{\xi_1}\right)_c \Big(\xi_1\,\mathcal{M}_{n+1}^{(1)f}(\xi_c)\Big) \tag{6.8}$$

$$\sigma_{n+2}^{(2)}(\xi_c) = \int \mathrm{d}\Phi_{n+2}^{d=4} \left(\frac{1}{\xi_1}\right)_c \left(\frac{1}{\xi_2}\right)_c \Big(\xi_1\xi_2\,\mathcal{M}_{n+2}^{(0)f}\Big) \tag{6.9}$$

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 $\xi_c$ 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].

## 6.1 FKS$^\ell$: extension to N$^\ell$LO

The pattern that has emerged in the previous cases leads to the following extension to an arbitrary order $\ell$ in perturbation theory:

$$\mathrm{d}\sigma^{(\ell)} = \sum_{j=0}^{\ell} \mathrm{d}\sigma_{n+j}^{(\ell)}(\xi_c) \tag{6.10}$$

$$\mathrm{d}\sigma_{n+j}^{(\ell)}(\xi_c) = \mathrm{d}\Phi_{n+j}^{d=4}\,\frac{1}{j!}\left(\prod_{i=1}^{j}\left(\frac{1}{\xi_i}\right)_c \xi_i\right)\mathcal{M}_{n+j}^{(\ell-j)f}(\xi_c) \tag{6.11}$$

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 (6.6). Furthermore, the phase-space integrations are manifestly finite.

# Chapter 7

# Glossary

## 7.1 Acronyms

**BR**
a branching ratio

**EW**
electroweak

**FKS**
the Frixione-Kunszt-Signer scheme used in McMule. See Section *The FKS^2 scheme*.

**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 sinuglarities, the numerical instabilties 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 $\beta$ (or equivalently small masses), this becomes almost singular as $y \to 1$.

**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-hasing-algorithm-1, used for hashing McMule's source code in autoversioning

**SM**
: the Standard Model of particle physics

**VP**
: vacuum polarisation

## 7.2 Technical terms

**config file**
: a shell file specifying, among other things, the statistics to be used

**containerisation**
: the concept of bundling all dependecies etc. with McMule. See Sections *Basics of containerisation* and *Basics of containerisation*

**container**
: a container that has bundled all dependecies 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. (4.2)

**event**
: the hard event that does not get subtracted in FKS, cf. (4.2)

**full period**
: a surjective *RNG*

**generic pieces**
: 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 processes**
A generic process is a prototype for the physical process such as $\ell p \to \ell p$ where the flavour of the lepton $\ell$ is left open.

**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 $\xi_c$ 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.

**process group**
Processes are grouped into process grous if they share matrix elements such as $\mu \to \nu\bar\nu e$ and $\mu \to \nu\bar\nu e\gamma$ (`mudec`) or $e\mu \to e\mu$ and $\ell p \to \ell p$ (`mue`).

**random seed**
the initial value of the *RNG*. In McMule this may be between 1 and 2<sup>31</sup>-2. See Section *Random number generation* for further details.

**soft cut**
a value of $\xi$ 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*.

## Chapter 8

# Bibliography

# Chapter 9

# Particle ID

The following table lists the `which_pieces` of McMule as well as the corresponding *PID*. For example, when calculating the process $\mu^+ \to e^+ \nu\bar\nu e^+ e^-$, 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 | $\mu^-$ | $\to$ | $e^-$ |
| m2ennF | 1 | | | |
| m2ennFF | 1 | | | |
| m2ennNF | 1 | | | |
| m2ennR | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2ennRF | 1 | | | |
| m2enng0 | 1 | | | |
| m2enngV | 1 | | | |
| m2enngC | 1 | | | |
| m2ennRR | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2enngR | 1 | | | |
| m2ennee0 | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2enneeV | 1 | | | |
| m2enneeC | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2enneeA | 1 | $\mu^-$ | $\to$ | $e^-$ |

| which_piece | P? | $p_1$ | | $p_2$ |
|---|---|---|---|---|
| m2enneeR | 1 | $\mu^+$ | $\to$ | $e^+$ |
| t2mnnee0 | 1 | $\tau^-$ | $\to$ | $\mu^-$ |
| t2mnneeV | 1 | | | |
| t2mnneeC | 1 | $\tau^-$ | $\to$ | $\mu^-$ |
| t2mnneeA | 1 | $\tau^-$ | $\to$ | $\mu^-$ |
| t2mnneeR | 1 | $\tau^+$ | $\to$ | $\mu^+$ |
| m2ej0 | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2ejF | 1 | | | |
| m2ejR | 1 | $\mu^-$ | $\to$ | $e^-$ |
| m2ejg0 | 1 | | | |
| em2em0 | 0 | $e^-$ | | $\mu^-$ |
| em2emV | 0 | | | |
| em2emC | 0 | | | |
| em2emFEE | 0 | | | |
| em2emFEM | 0 | | | |
| em2emFMM | 0 | | | |
| em2emA | 0 | | | |
| em2emFFEEEE | 0 | | | |
| em2emFFMMMM | 0 | | | |
| em2emFFMIXDz | 0 | | | |
| em2emAA | 0 | | | |
| em2emAFEE | 0 | | | |
| em2emAFEM | 0 | | | |
| em2emAFMM | 0 | | | |
| em2emNFEE | 0 | | | |
| em2emNFEM | 0 | | | |
| em2emNFMM | 0 | | | |
| em2emREE | 0 | $e^-$ | | $\mu^-$ |
| em2emREM | 0 | | | |
| em2emRMM | 0 | | | |
| em2emRFEEEE | 0 | | | |
| em2emRFMIXD | 0 | | | |
| em2emRFMMMM | 0 | | | |
| em2emAREE | 0 | | | |
| em2emAREM | 0 | | | |
| em2emARMM | 0 | | | |
| em2emRREEEE | 0 | $e^-$ | | $\mu^-$ |
| em2emRRMIXD | 0 | | | |
| em2emRRMMMM | 0 | | | |
| emZem0X | 0 | $e^-$ | | $\mu^+$ |
| emZemFX | 0 | | | |
| emZemRX | 0 | | | |
| mp2mp0 | 0 | $\mu^-$ | | $p$ |
| mp2mpF | 0 | | | |
| mp2mpA | 0 | | | |
| mp2mpFF | 0 | | | |
| mp2mpAA | 0 | | | |
| mp2mpAF | 0 | | | |
| mp2mpNF | 0 | | | |
| mp2mpR | 0 | $\mu^-$ | | $p$ |

| which_piece | P? | $p_1$ | $p_2$ |
|---|---|---|---|
| mp2mpRF | 0 | | |
| mp2mpAR | 0 | | |
| mp2mpRR | 0 | $\mu^-$ | $p$ |
| ee2mm0 | 2 | $e^-$ | $e^+$ |
| ee2mmF | 0 | | |
| ee2mmFFEEEE | 0 | | |
| ee2mmR | 0 | $e^-$ | $e^+$ |
| ee2mmRFEEEE | 0 | | |
| ee2mmRREEEE | 0 | $e^-$ | $e^+$ |
| ee2mmA | 0 | $e^-$ | $e^+$ |
| ee2mmAA | 2 | | |
| ee2mmNFEE | 2 | | |
| ee2mmAFEE | 0 | | |
| ee2mmAREE | 0 | $e^-$ | $e^+$ |
| eeZmm0 | 2 | $e^-$ | $e^+$ |
| eeZmm0X | 2 | | |
| eeZmmFX | 0 | | |
| eeZmmAX | 0 | | |
| eeZmmRX | 0 | $e^-$ | $e^+$ |
| ee2ee0 | 0 | $e^-$ | $e^-$ |
| ee2eeA | 0 | | |
| ee2eeF | 0 | | |
| ee2eeFF | 0 | | |
| ee2eeAA | 0 | | |
| ee2eeAF | 0 | | |
| ee2eeNF | 0 | | |
| ee2eeR | 0 | $e^-$ | $e^-$ |
| ee2eeRF | 0 | | |
| ee2eeAR | 0 | | |
| ee2eeRR | 0 | $e^-$ | $e^-$ |
| eb2eb0 | 0 | $e^-$ | $e^+$ |
| eb2ebF | 0 | | |
| eb2ebFF | 0 | | |
| eb2ebR | 0 | $e^-$ | $e^+$ |
| eb2ebRF | 0 | | |
| eb2ebRR | 0 | $e^-$ | $e^+$ |
| ee2nn0 | 0 | $e^-$ | $e^+$ |
| ee2nnF | 0 | | |
| ee2nnS | 0 | | |
| ee2nnSS | 0 | | |
| ee2nnCC | 0 | | |
| ee2nnR | 0 | $e^-$ | $e^+$ |
| ee2nnRF | 0 | | |

# Chapter 10

# Available processes and `which_piece`

When running McMule, we recommend using the following `which_piece`

Table 10.1: `which_piece` to use for different physics processes

| Process | Order | $n$-particle | $(n+1)$-particle | $(n+2)$-particle |
|---|---|---|---|---|
| $\mu \to$ | LO | `m2enn0` | | |
| $e\nu\bar{\nu}$ | NLO | `m2ennF` | `m2ennR` | |
| | NNLO | `m2ennFF`, `m2ennNF` | `m2ennRF` | `m2ennRR` |
| $\mu \to$ | LO | `m2enng0` | | |
| $e\nu\bar{\nu}\gamma$ | NLO | `m2enngF` | `m2enngR` | |
| $\mu \to$ | LO | `m2ennee0` | | |
| $e\nu\bar{\nu}ee$ | NLO | `m2enneeV`, `m2enneeC`, `m2enneeA`, | `m2enneeR` | |
| $\mu \to$ | LO | `m2ej0` | | |
| $eJ$ | NLO | `m2ejF` | `m2ejR` | |
| $\mu \to eJ\gamma$ | LO | `m2ejg0` | | |
| $e\mu \to$ | LO | `em2em0` | | |
| $e\mu$ | NLO | `em2emFEE`, `em2emFEM`, `em2emFMM`, `em2emA` | `em2emREE15`, `em2emREE35`, `em2emREM`, `em2emRMM` | |
| | el. NNLO | `em2emFFEEEE`, `em2emAA`, `em2emAFEE`, `em2emNFEE` | `em2emRFEEEE15`, `em2emRFEEEE35`, `em2emRFEEEEco`, `em2emAREE15`, `em2emAREE35` | `em2emRREEEE1516`, `em2emRREEEE3536`, `em2emRREEEEc` |
| | full NNLO | `em2emFFMIXDz`, `em2emFFMMMM`, `em2emAFEM`, `em2emAFMM`, `em2emNFEM`, `em2emNFMM` | `em2emRFMIXD15`, `em2emRFMIXD35`, `em2emRFMIXDco`, `em2emRFMMMM`, `em2emAREM`, `em2emARMM` | `em2emRRMIXD1516`, `em2emRRMIXD3536`, `em2emRRMIXDc`, `em2emRRMMMM` |
| $\mu p \to$ | LO | `mp2mp0` | | |
| $\mu p$ | NLO | `mp2mpF`, `mp2mpA` | `mp2mpR15`, `mp2mpR35` | |
| | NNLO | `mp2mpFF`, `mp2mpAA`, `mp2mpAF`, `mp2mpNF` | `mp2mpRF15`, `mp2mpRF35`, `mp2mpRFco`, `mp2mpAR15`, `mp2mpAR35` | `mp2mpRR1516`, `mp2mpRR3536`, `mp2mpRRco` |
| $ee \to$ | LO | `ee2mm0` | | |
| $\mu\mu$ | NLO | `ee2mmF`, `ee2mmA` | `ee2mmR` | |
| | el. NNLO | `ee2mmFFEEEE`, `ee2mmAA`, `ee2mmAFEE`, `ee2mmNFEE` | `ee2mmRFEEEE`, `ee2mmAREE` | `ee2mmRREEEE`, |
| | LO | `eeZmm0` | | |
| | EW NLO | `eeZmmFX`, `eeZmmAX` | `eeZmmRX` | |
| $e^-e^- \to$ | LO | `ee2ee0` | | |
| $e^-e^-$ | NLO | `ee2eeF`, `ee2eeA` | `ee2eeR125`, `ee2eeR345` | |
| | NNLO | `ee2eeFF`, `ee2eeAF`, `ee2eeAA`, `ee2eeNF` | `ee2eeRF125`, `ee2eeRF345`, `ee2eeAF125`, `ee2eeAR135` | `ee2eeRR15162526`, `ee2eeRR35364546` |
| $e^-e^+ \to$ | LO | `eb2eb0` | | |
| $e^-e^+$ | NLO | `eb2ebF`, `eb2ebA` | `eb2ebR125`, `eb2ebR35`, `eb2ebR45` | |
| | NNLO | `eb2ebFF`, `eb2ebAF`, `eb2ebAA`, `eb2ebNF` | `eb2ebRF125`, `eb2ebRF35`, `eb2ebRF45`, `eb2ebAR125`, `eb2ebAR35`, `eb2ebAR45` | `eb2ebRR15162526`, `eb2ebRR3536`, `eb2ebRR4546` |

We also show a list of all available `which_piece` in Figure 10.1.

Figure 10.1: The `which_piece` implemented in McMule

# Chapter 11

# Fortran reference guide

McMule's Fortran code has hundreds of functions and subroutine and we will not document all of them here. However, we will list the user-facing function that are intended to help construct user files.

## 11.1 User-modifiable parameters

The following parameters may be modified by the user though it might become necessary to completely recompile McMule ones done.

**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

`real(kind=prec) musq`

The renormalisation scale $\mu^2$. This variable *needs* to be set by the user, otherwise McMule will fail.

`integer nel`

Set to 1 if electron *VP* loops are to be included, set to 0 otherwise. More options may be added later

`integer nmu`

Set to 1 if muon *VP* loops are to be included, set to 0 otherwise. More options may be added later

`integer ntau`

Set to 1 if tau *VP* loops are to be included, set to 0 otherwise. More options may be added later

`integer nhad`

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

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

The polarisation of the first polarised particle

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

The polarisation of the second polarised particle

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

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 real(kind=prec) sachs_gmag(q2)`

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 $Q^2$

`subroutine init_flavour(flavour)`

The definitions of the `flavour`. Users may edit this to add new experiments etc.

`real(kind=prec) GF`

The Fermi constant. For predominantly historic reasons, this is set to `1._prec`.

`real(kind=prec) alpha`

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

`real(kind=prec) sw2`

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.

`real(kind=prec) Mel`

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

`real(kind=prec) Mmu`

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

`real(kind=prec) Mtau`

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

`real(kind=prec) Mproton`

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

`real(kind=prec) MZ`

The numerical value of the Z boson mass in MeV

`real(kind=prec) Mm`

The actual value of the `m` particle, usually the muon mass but if `flavour` is eg. `tau-e` the tau mass

`real(kind=prec) Me`

The actual value of the `e` particle, usually the electron mass but if `flavour` is eg. `tau-mu` the muon mass

`real(kind=prec) Mt`

The actual value of the `t` particle, usually the tau mass

`real(kind=prec) scms`

The numerical value of the centre-of-mass energy

`real(kind=prec) lambda`

The dipole coefficient in the Sachs form factors of the proton in MeV:sup:*2*

`real(kind=prec) kappa`

The magnetic moment of the proton in Sachs form factors

## 11.2 Technical parameters

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

`character which_piece(25)`

The piece being integrated, cf. Section *Available processes and which_piece*

`character flavour(15)`

The flavour configuration being used

`real(kind=prec) softcut`

The value of $\xi$ below which the integrand is set to zero without subtraction

`real(kind=prec) colcut`

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

`real(kind=prec) sSwitch`

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

`real(kind=prec) ntsSwitch`

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

## 11.3 User-facing functions

The following function are available for the user to construct observables. Momenta are of the form `(/ px, py, pz, E /)`.

### 11.3.1 Scalar quantities

**function real**(*kind=prec*) *s(p1, p2)*

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 real**(*kind=prec*) *sq(p)*

The Lorentz square $p^2$

**Parameters**

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

**function real**(*kind=prec*) *asymtensor(p1, p2, p3, p4)*

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 real`(*kind=prec) eta(p*)

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 real`(*kind=prec) rap(p*)

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 real`(*kind=prec) pt(p*)

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 real`(*kind=prec) absvec(p*)

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

**Parameters**

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

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

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

**Parameters**

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

**Note:** This may return $100\pi$ if the calculation fails.

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

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 $p_2$

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

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

### 11.3.2 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 $\alpha$ around the current $z$ axis, then by $\beta$ around the current $y$ axis, and the by $\gamma$ 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 $\alpha$
- **b** *[real(kind=prec)]* :: the angle $\beta$
- **c** *[real(kind=prec)]* :: the angle $\gamma$

**Return**
**euler_mat** (4,4) *[real(kind=prec)]* :: the $4 \times 4$ Euler matrix

## 11.4 The user file

### 11.4.1 Mandatory functions

The user must implement the following functions in the user file

`nr_q` *[integer,parameter=* n*]*
The number of distributions the user intends to calculate

`nr_bins` *[integer,parameter=* n*]*
The number of bins in the distributions the user intends to calculate

`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

`userdim` *[integer]*
The number of integrations the user wishes to carry out to account eg. for beam effects

`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.

**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.

`userweight` *[real(kind=prec)]*
The weight the user wishes to attach to a given event

`names` (*nr_q*) *[character(len=namesLen)]*
The names of the distributions the user wishes the calculate

`filenamesuffix` *[character(len=filenamesuffixLen)]*
The observable-specific suffix to the vegas file

`subroutine fix_mu()`
The user needs to choose the renormalisation scale $\mu^2$ by writing to the variable `musq`. This can be done on a per-event basis.

A common example would be

```
SUBROUTINE FIX_MU
musq = Mm**2
END SUBROUTINE FIX_MU
```

**subroutine inituser()**

This is called without arguments once as soon as McMule starts and has read all other configuration, meaning that it can access `which_piece` and `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 *filenamesuffix* variable which is appended to the name of the VEGAS file.

Example for reading a cut:

```
SUBROUTINE INITUSER
integer cut
read*,cut
write(filenamesuffix,'(I2)') cut
END SUBROUTINE INITUSER
```

with a global variable `cut`

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

The *measurement function* the user wishes to calculate. This needs to at least set *pass_cut* but also returns the values of the observables that are to be computed. It usually also calls *fix_mu()* to fix the renormalisation scale though this can be done elsewhere. If the user wishes to consider polarised scattering, `pol1` and `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 ==== !!
call fix_mu
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

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

The user may use this routine in combination with *userweight* to integrate over further parameters, i.e. to calculate

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

with a generalised *measurement function* $f$. A minimal example that does not include extra intgration 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`.

### 11.4.2 Tweaking parameters

In rare cases it may be necessary to tweak some parameters.

**`integer namesLen`**

The maximally allowed length of the histogram `names`.

**`integer filenamesuffixLen`**

The maximally allowed length of the observable name as specified in `filenamesuffix`.

**`integer bin_kind`**

The binning mechanism being used, `0` for $\mathrm{d}\sigma/\mathrm{d}Q$ and `1` for $Q\mathrm{d}\sigma/\mathrm{d}Q$.

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

## 11.5 Technical routines

The following types, variables, and routines are unlikely to be needed by the typical user and are instead aimed at McMule's developers.

### 11.5.1 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 partciles
- **`% 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 *`particle`*s

**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), `t` (only tauonic)

`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]* :: $\xi_c$ (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

### 11.5.2 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

[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 $\ell_1\ell_2 \to \ell_1\ell_2$ 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 $Q_1^3\,Q_2^5$.

### 11.5.3 Package-X function

**Note:** This section needs to be completed, link to issue

`function DiscB()`

`function DiscB_cplx()`

`function ScalarC0IR6()`

`function ScalarC0IR6_cplx()`

`function ScalarC0()`

`function ScalarC0_cplx()`

`function ScalarD0IR16()`

`function ScalarD0IR16_cplx()`

### 11.5.4 VP functions

**Note:** This section needs to be completed, link to issue

### 11.5.5 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 $1 \to 2$ 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 $1 \to 3$ 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 $1 \to 3$ decays, requires $m_4 = 0$. Tuned for $\sphericalangle(p_2, q_4)$ 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 $1 \to 4$ 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 $1 \to 4$ decays, tuned for $\sphericalangle(p_2, q_5)$ and $E_5$, collinear limit is `ra(2) -> 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 $1 \to 4$ decays, requires $m_5 = 0$. Tuned for $\sphericalangle(p_2, q_5)$ 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 $1 \to 5$ 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 $1 \to 5$ decays with FKS-ish tuning. This is designed for the decay $\mu^+ \to e^+ \nu\bar{\nu} e^+ e^-$. `q2` should be the unique particle (electron) and `q3` and `q4` are the identical particles (postirons):

```
                   _/  q
                   /|   2
---<---*~~~~~~~~~~*
       |          |
       ^          ^  q
       | q        |   3
          4
```

The 'spectator' neutrinos are `q5` and `q6`. Start by generating `p2` and `p3` at an angle `* = arccos(y2)`:

```
              ^ p2
             |||
             |||
p3         __|||
 --__     /  |||
     --__/  *|||
         --__|||
             |||
```

Generate `p4` at an angle `* = arccos(y3)` and rotating by an angle `phi` w.r.t. to `p3`:

```
             |||      / p4
          _--|||--_  /
         -_  |||  _-/
p3         --___-- /
 --__        |||__/
     --__    |||*/
         --__|||/
             |||
```

**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 $1 \to 5$ 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 $1 \to 5$ decays, requires $m_6 = 0$. Tuned for $\sphericalangle(p_2, q_6)$ 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 $1 \to 5$ 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 PSD6_FKSS**(*ra*, *q1*, *m1*, *q2*, *m2*, *q3*, *m3*, *q4*, *m4*, *q5*, *q6*, *weight*)

Double-*FKS* phase space routine for $1 \to 5$ 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 $1 \to 6$ 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 $1 \to 6$ 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 $1 \to 6$ 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 $2 \to 2$ 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 $2 \to 3$ 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 $2 \to 3$ 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 $2 \to 4$ 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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not :term`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 $2 \to 4$ 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 $1 \to 5$ decays, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(p_2, q_{5,6})$ and $E_{5,6}$ Partioning 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 $1 \to 5$ decays with FKS-ish tuning. Modification of *PSD6_23_24_34()* with $2 \leftrightarrow 5$. This is designed for the decay $\mu^+ \to e^+\nu\bar{\nu}e^+e^-$.

**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 $1 \to 6$ 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 $2 \to 3$ cross sections, requires $m_5 = 0$. Tuned for *ISR* and not *FSR*. Partioning 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 $2 \to 3$ cross sections, requires $m_5 = 0$. Tuned for $\sphericalangle(q_3, q_5)$ and $E_5$ Partioning 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 $2 \to 3$ cross sections, requires $m_5 = 0$. Tuned for *ISR* and not *FSR*. Partioning 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 $2 \to 3$ cross sections, requires $m_5 = 0$. Tuned for $\sphericalangle(q_3, q_5)$ and $E_5$ Partioning 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 $2 \to 3$ cross sections, requires $m_5 = 0$. Tuned for $\sphericalangle(q_3, q_5)$ and $E_5$ Partioning 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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not :term`FSR` Partioning 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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(q_3, q_{5,6})$ and $E_{5,6}$ Partioning 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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(q_3, q_{5,6})$ and $E_{5,6}$ Partioning of *PSX4_35_36_FKSS()* with $\min\big(s_{15}, s_{16}\big) > \min\big(s_{35}, s_{36}\big)$.

**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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for *ISR* and not :term`FSR` Partioning of *PSX4_FKSS()* with $\min\big(s_{15}, s_{16}, s_{25}, s_{26}\big) < \min\big(s_{35}, s_{36}, s_{54}, s_{46}\big)$.

**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 $2 \to 4$ cross sections, requires $m_5 = m_6 = 0$. Tuned for $\sphericalangle(q_3, q_{5,6})$ and $E_{5,6}$ Partioning of *PSX4_35_36_FKSS()* with $\min\big(s_{45}, s_{46}\big) > \min\big(s_{15}, s_{25}, s_{35}, s_{16}, s_{26}, s_{36}\big)$.

**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 12

# pymule user guide

This section describes all public functions and classes in pymule.

## 12.1 Working with files

`pymule.`**`mergefks`**`(`**sets*, ***kwargs*`)`

performs the FKS merge

**Parameters**

- **`sets`** – random-seed-merged results (usually from `sigma()`)
- **`binwisechi`** – bool, optional, default False; if set to True, also return extra distributions containing the $\chi^2$ of the bin-wise FKS merge. This cannot be used together with `anyxi` and the result should *not* be passed to `scaleset` for obvious reasons.

**Returns**

the FKS-merged final set containing cross sections, distributions, and run-time information. The chi2a return is a list of the following

- the $\chi^2$ of the FKS merge
- a list of $\chi^2$ from previous operations, such as random seed merging or the integration.

---

**Note:** Optional argument `anyxi` (or anything starting with `anyxi`): Sometimes it is necessary to merge $\xi_c$-dependent runs (such as a counter term) and $\xi_c$-independent runs (such as the one-loop term). Do not use this together with **binwisechi**

---

**Example**

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

```
>>> mergefks(sigma("m2enn0"))
```

Load the NLO results

```
>>> mergefks(sigma("m2ennV"), sigma("m2ennR"))
```

Load the NNLO results where `m2ennNF` does not depend on $\xi_c$

```
>>> mergefks(sigma("m2ennFF"), sigma("m2ennRF"), sigma("m2ennRR"),␣
↪anyxi=sigma("m2ennNF"))
```

pymule.**setup**(***kwargs*)

sets the default arguemnts 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 `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. `.*`.
- **filenames** – list, optional; list of files to loads, defaults to all files in **folder** (recurisvely if tar ball) Initialised to `None`, meaning everything.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram `name`. Initialised to to no merging, i.e. `{}`
- **types** – list of callables, optional; functions that convert the groups matched by `r` into python objects. Common examples would be `int` or `float`. Initialised to `[int,float, float]` as per McMule filename convention.
- **sanitycheck** – callable, optional; a function that, given a vegas dict, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.
- **cache** – folder name, optional; if existing folder, use as cache for compressed tarballs

**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*, ***kwargs*)

loads a `which_piece` and statistically combines the random seed.

**Parm piece**

str; `which_piece` to load

**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 `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. `.*`.
- **filenames** – list, optional; list of files to loads, defaults to all files in **folder** (recurisvely if tar ball) Initialised to `None`, meaning everything.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram `name`. Initialised to to no merging, i.e. `{}`
- **types** – list of callables, optional; functions that convert the groups matched by `r` into python objects. Common examples would be `int` or `float`. Initialised to `[int,float, float]` as per McMule filename convention.
- **sanitycheck** – callable, optional; a function that, given a vegas dict, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.
- **cache** – folder name, optional; if existing folder, use as cache for compressed tarballs

**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")
```

Load only observable `O3`

```
>>> sigma("m2enn0", obs="3")
```

## 12.2 Working with errors

`pymule.addplots`(*a*, *b*, *sa=1.0*, *sb=1.0*)

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)`

calculates the $\chi^2$/d.o.f. of numbers

**Parameters**

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

**Returns**

float; the $\chi^2/\text{d.o.f.} = \frac{1}{n}\sum_{n=1}^n(\frac{y_i-\bar y}{\delta y_i})^2$ 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, b)`

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, b, offset=0.0)`

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.integratehistogram(hist)`

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.mergebins(p, n)`

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 `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, quiet=False)`
statistically combines values with uncertainties

**Parameters**

- **values** – Nx2 numpy matrix or list of lists; the values as `[[y1, dy1], [y2, dy2], ...]`
- **quiet** – bool, optional; whether to print or return the $\chi^2$ for the combination

**Returns**
either answer as numpy array `[y, dy]` or tuple of $chi^2$ and answer

**Example**
If `quiet` is not specified this will print the $chi^2$

```
>>> 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.mergeplots(ps, returnchi=False)`
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], ...]), ...]`

- **returnchi** – bool, optional; if `True` returns two plots, the requested combination and the bin-wise $\chi^2$

**Returns**
a Nx3 numpy matrix if `returnchi=False`

**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*`)`

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*, *prec=0*`)`

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, otpional; number of extra signficant figures

**Returns**
str; the formatted string

**Example**
printing $53.2 \pm 0.1$ to one significant figure

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

`pymule.`**`scaleplot`**`(`*a*, *sx*, *sy=None*`)`

rescales a plot such that the integrated plot remains unchanged, i.e. rescale $x \to x/s$ and $y \to y \cdot s$. 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*, *b*)

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])
```

## 12.3 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': nlodata['thetae'],
...     'nnlo':nnlodata['thetae'],
...   },
...   labelx="$\theta_e\,/\,{\rm mrad}$",
...   labelsigma="$\D\sigma/\D\theta_e\ /\ {\rm\upmu b}$",
...   legend={
...     'lo': '$\sigma^{(0)}$',
...     'nlo': '$\sigma^{(1)}$',
...     'nnlo': '$\sigma^{(2)}$'
...   },
...   legendopts={'what': 'u', 'loc': 'lower right'}
... )
```

`pymule.`**`mergefkswithplot`**(*sets*, *scale=1.0*, *showfit=[True, True]*, *xlim=[-7, 0]*)

performs and FKS merge like `mergefks()` but it also produces a $\xi_c$ independence plot

**Note:** In contrast `mergefks()`, here phase-space partioned results need to be merged first. This is done by grouping those into an array first, sorted by number of particles in the final state, i.e. we start with the n-particle corrections.

**Parameters**

- **`sets`** – list of list of random-seed-merged results (usually from `sigma()`), starting with the lowest particle number and going up
- **`scale`** – float, optional; rescale factor for the plot and result
- **`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 final set containing cross sections, distributions, and run-time information. The chi2a return is a list of the following

- the $\chi^2$ of the FKS merge
- a list of $\chi^2$ from previous operations, such as random seed merging or the integration.

**Example**

In the partioned muon-electron scattering case

```
>>> fig, res = mergefkswithplot([
...     [
...         sigma('em2emFEE'), sigma('em2emFMM'), sigma('em2emFEM')
...     ], [
...         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**(*sets*, *n*, *xlim=[-7, 0]*, *scale=1*)

creates a residue plot for a $\xi_c$ fit

**Parameters**

- **sets** – dict or list; random-seed-merged results (usually from `sigma()`) or list thereof
- **n** – int; order of the fit, 1 at NLO, 2 at NNLO
- **xlim** – tuple of floats, optional; upper and lower bounds for $\log\xi_c$
- **scale** – float, optional; rescale factor for the plots

**Returns**

a figure and the fit coefficients as a matrix

# Chapter 13

# pymule reference guide

This section describes all functions and classes in pymule. Most users will not have to view this.

## 13.1 Working with errors

`pymule.errortools.addplots`(*a*, *b*, *sa=1.0*, *sb=1.0*)

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.errortools.chisq`(*values*)

calculates the $\chi^2$/d.o.f. of numbers

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

**Returns**
float; the $\chi^2$/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.errortools.`**`combineNplots`**(*func*, *plots*)
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 $f(p_0, f(p_1, f(p_2, \cdots)))$

`pymule.errortools.`**`combineplots`**(*a*, *b*, *yfunc*, *efunc*, *tol=1e-08*)
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 error `efunc(a, da b, db)`
- **tol** – float, optional; the difference at which values are considered equal

**Returns**
Nx3 numpy matrix; the combined plot `np.array([[x1, yfunc(..), 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, da, b, db: sqrt(da**2 + db**2))
```

Calculate a K factor

```
>>> combineplots(dnlo, lo,
...              lambda a, b: 1 + a/b,
...              lambda a, da, b, db: np.sqrt(db**2 * a**2 / b**4 +
↪da**2 / b**2)
```

pymule.errortools.**dividenumbers**(*a*, *b*)

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*, *b*, *offset=0.0*)

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*)

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 $\int d\sigma/dxdx$ 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.`**`mergebins`**(*p*, *n*)

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 `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.errortools.`**`mergenumbers`**(*values*, *quiet=False*)

statistically combines values with uncertainties

**Parameters**

- **values** – Nx2 numpy matrix or list of lists; the values as `[[y1, dy1], [y2, dy2], ...]`
- **quiet** – bool, optional; whether to print or return the $\chi^2$ for the combination

**Returns**
either answer as numpy array `[y, dy]` or tuple of $chi^2$ and answer

**Example**

If `quiet` is not specified this will print the $chi^2$

```
>>> 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*, *returnchi=False*`)`

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], ...]), ...]`
- **returnchi** – bool, optional; if `True` returns two plots, the requested combination and the bin-wise $\chi^2$

**Returns**

a Nx3 numpy matrix if `returnchi=False`

**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*`)`

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*, *prec=0*`)`

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, otpional; number of extra signficant figures

**Returns**
str; the formatted string

**Example**
printing $53.2 \pm 0.1$ to one significant figure

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

`pymule.errortools.`**`scaleplot`**`(`*a*, *sx*, *sy=None*`)`

rescales a plot such that the integrated plot remains unchanged, i.e. rescale $x \to x/s$ and $y \to y \cdot s$. 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*, *b*`)`

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])
```

## 13.2 Working with abstract records

## 13.3 Working with vegas records

`pymule.vegas.exportvegas(dic, filename='', fp=None)`

saves a vegas file

**Parameters**

- **dic** –

  **a vegas dataset dictionary containing at least**

  – `value`: the best estimate for the cross section and its error as `np.array([y, e])`

  – `chi2a`: the $\chi^2$ estimate of the integrator

  – all histograms as specified by their `name(..)` in user.f95

  **it may also contain the optional keys**

  – `runtime`: the runtime, defaults to `time.clock()`

  – `msg`: any message, defaults to `Warning:  Generated with Python`

  – `SHA`: the first 5 characters of a hash, defaults to `00000`

  – `iteration`: the number of iterations in the file, defaults to 2

- **filename** – file name to open, optional
- **fp** – file pointer to write to, optional
- **returnev** – bool, optional; return the full vegas file or only usable things

---

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

---

**Example**

save a random run to disk

```
>>> dic = {"value": [10, 0.2],
...        "chi2a": 0.2,
...        "Ee": np.array([[1, 5, 0.3], [2, 6, 0.35]])}
>>> exportvegas(dic ,"out.vegas")
```

`pymule.vegas.getplots(s)`

removes all the keys that are not distributions from a vegas dataset

**Parameters**

**s** – a vegas dataset or a list of vegas datasets

**Returns**

a list of plots appearing in all datasets

`pymule.vegas.guess_version(fp, inttype='i')`

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.importvegas`(*filename=''*, *fp=None*, *inttype='i'*, *returnev=False*)

loads a vegas file

**Parameters**

- **filename** – file name to open, optional
- **fp** – file pointer to read, optional
- **inttype** – either `'i'` or `'q'`, optional; the integer type to use for v1 or v2, inferred otherwise
- **returnev** – bool, optional; return the full vegas file or only usable things

**Returns**

**a vegas dataset dictionary containing**

- `time`: the job's run time (since v2)
- `msg`: any message. Usually this contains information on the state of the integrator (since v2)
- `SHA`: the first 5 characters of the source-tree's SHA1 hash at compile time.
- `iteration`: the number of iterations completed in this file
- `value`: the best estimate for the cross section and its error as `np.array([y, e])`
- `chi2a`: the $\chi^2$ estimate of the integrator
- all histograms as specified by their `name(..)` in user.f95

**if returnev is passed, also returns keys**

- `ndo`
- `xi`: the vegas grid
- `randy`: the random number seed

**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

```
>>> importvegas('m2ennRR_mu-e_S0000068031X0.50000D0.50000_ITMX080x150M_
→012.vegas')
{'time': 103523.659092, 'msg': 'Uninterupted integration. Program SHA␣
→isbe42eccf04a8fb0afa5fa2f80be6a492bb2093a4␣
→(git:423084d47038d3dbf51f69459d2e622312eec594)', 'SHA': 'be42e',
→'iteration': 44, 'value': array([-3.65282506e+06,  1.89792449e+02]),
→'chi2a': 0.8714706523473873, 'Ee': array([[          -inf,  0.
```

(continues on next page)

(continued from previous page)

```
↪00000000e+00,  0.00000000e+00],
       [ 1.30000000e-02,  0.00000000e+00,  0.00000000e+00],
       [ 3.90000000e-02,  0.00000000e+00,  0.00000000e+00],
       ...,
       [ 2.59610000e+01,  0.00000000e+00,  0.00000000e+00],
       [ 2.59870000e+01,  0.00000000e+00,  0.00000000e+00],
       [            inf, -3.65294784e+06,  1.79397543e+02]]), 'cthe':␣
↪array([[           -inf,  0.00000000e+00,  0.00000000e+00],
       [-9.99000000e-01, -4.77190754e+06,  4.08877779e+03],
       [-9.97000000e-01, -4.76466432e+06,  4.81770106e+03],
       ...,
       [ 9.97000000e-01,  0.00000000e+00,  0.00000000e+00],
       [ 9.99000000e-01,  0.00000000e+00,  0.00000000e+00],
       [            inf,  0.00000000e+00,  0.00000000e+00]])}
```

`pymule.vegas.`**`read_record`**(*fp*, *typ*)

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 charater (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 `ll`
- body of length `ll`
- 4 byte footer: a repetition of `ll` to make sure the record is properly closed.

Records can contain multiple variables.

`pymule.vegas.`**`write_record`**(*fp*, *typ*, *content*)

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.

## 13.4 Working with records of data

`pymule.loader.addsets(s)`

adds a list of vegas datasets

**Parameters**

`s` –

**a list of vegas datasets dictionaries with the keys**

- time
- value
- chi2a
- all histograms as specified by their `name(..)` in user.f95

**Returns**

the resulting sum. The resulting $\chi^2$ is a list of constituent $\chi^2$.

`pymule.loader.callsanitised(func, **kwargs)`

calls a function with arguments from `kwargs` and those specified in `loadargs`

**Parameters**

- `func` – callable; function to call
- `**kwargs` – arguments overriding `loadargs`

Arguments that don't match **func** are discarded.

`pymule.loader.commit_cache(cachefolder, full_name, fp)`

writes to cache if cachefolder exists

**Parameters**

- `cachefolder` – path to cache folder
- `full_name` – name of file in cache folder
- `fp` – file pointer to read from

`pymule.loader.hash_file(name)`

hashes a files using SHA1

**Parameters**

`name` – file path

**Returns**

hex-digested SHA1 hash of the file

`pymule.loader.importreg(r, folder='.', filenames=None, cachefolder='', merge={}, types=[<class 'int'>, <class 'float'>, <class 'float'>], sanitycheck=<function <lambda>>)`

imports all vegas files matching a regular expression

**Parameters**

- **r** – str; regular expression to match in file names
- **folder** – str, optional; file name, optional; folder or tarball to search for vegas files
- **filenames** – list, optional; list of files to loads, defaults to all files in **folder** (recurisvely if tar ball)
- **cachefolder** – folder name, optional; if existing folder, use as cache for compressed tarballs
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram `name`. defaults to no merging
- **types** – list of callables, optional; functions that convert the groups matched by `r` into python objects. Common examples would be `int` or `float`. Default: `[int,float, float]` as per McMule filename convention
- **sanitycheck** – callable, optional; a function that, given a vegas dict, whether to include the file in the output (return `True`) or to skip (return `False`).

**Returns**

a dictionary of merged vegas datasets, keyed by the groups defined in the regular expression `r` as parsed by `types`

`pymule.loader.mergefks(*sets, **kwargs)`

performs the FKS merge

**Parameters**

- **sets** – random-seed-merged results (usually from `sigma()`)
- **binwisechi** – bool, optional, default False; if set to True, also return extra distributions containing the $\chi^2$ of the bin-wise FKS merge. This cannot be used together with `anyxi` and the result should *not* be passed to `scaleset` for obvious reasons.

**Returns**

the FKS-merged final set containing cross sections, distributions, and run-time information. The chi2a return is a list of the following

- the $\chi^2$ of the FKS merge
- a list of $\chi^2$ from previous operations, such as random seed merging or the integration.

**Note:** Optional argument `anyxi` (or anything starting with `anyxi`): Sometimes it is necessary to merge $\xi_c$-dependent runs (such as a counter term) and $\xi_c$-independent runs (such as the one-loop term). Do not use this together with **binwisechi**

**Example**

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

```
>>> mergefks(sigma("m2enn0"))
```

Load the NLO results

```
>>> mergefks(sigma("m2ennV"), sigma("m2ennR"))
```

Load the NNLO results where m2ennNF does not depend on $\xi_c$

```
>>> mergefks(sigma("m2ennFF"), sigma("m2ennRF"), sigma("m2ennRR"),␣
↪anyxi=sigma("m2ennNF"))
```

pymule.loader.**mergeseeds**(*s*, *key=<function <lambda>>*)

statistically merges the different random seeds of a number of runs, combining cross sections, histograms, and run-time information.

**Parameters**

- **s** – a list of vegas datasets
- **key** – callable; function to define the keys of the resulting dictionary. Usually, this refers to the FKS parameters. In the default notation this is `lambda x: (x[1], x[2])` referring to $\xi_c$ and $\delta$, resp.

**Raises**

KeyError 'time': if merge is unsucessfull because no data is found

**Todo**

make error handling more useful

**Returns**

a merged vegas dataset. The runtime is the sum of individual times. The $\chi^2$ is a list of

- the $\chi^2$ of the cross section combination
- a list of the individual $\chi^2$

pymule.loader.**mergeset**(*s*, *binwisechi=False*)

statistically merges a set of runs, combining cross sections, histograms, and run-time information

**Parameters**

- **s** – a list of vegas datasets
- **binwisechi** – bool, optional; whether to include the bin-wise $\chi^2$ in the result.

**Raises**

KeyError 'time': if merge is unsucessfull because no data is found

**Todo**

make error handling more useful

**Returns**

a merged vegas dataset. The runtime is the sum of individual times. The $\chi^2$ is a list of

- the $\chi^2$ of the cross section combination
- a list of the individual $\chi^2$

pymule.loader.**multiintersect**(*lists*)

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 $l_1, l_2, \ldots, l_n$

**Returns**
the list $l_1 \cap l_2 \cap \cdots \cap l_n$

`pymule.loader.pattern`(*piece='.*'*, *flavour='.*'*, *obs=''*, *folderp='.*'*)
constructs a regular expression to be used in `importreg()` matching the usual McMule file name convention

**Parameters**

- **piece** – str, optional; the `which_piece` to load, defaults to everything
- **flavour** – str, optional; the `flavour` to load, defaults to everything
- **obs** – str, optional; the observable to load (the bit after the `O`), defaults to everything
- **folderp** – str, optional; a regular expression to match directory structures of a tar file, defaults to everything

**Returns**
a regular expression to be used in `importreg()`

`pymule.loader.scaleset`(*s*, *v*)
rescales a vegas dataset

**Parameters**

- **s** –

  **a vegas datasets dictionaries with the keys**

  – time

  – value

  – chi2a

  – all histograms as specified by their `name(..)` in user.f95

- **v** – the value to rescale the y values.

**Returns**
the rescaled dataset

**Note:** This naturally changes the cross section

`pymule.loader.scalesets`(*s*, *v*)
rescales a list of vegas datasets

**Parameters**

- **s** –

  **a list of vegas datasets dictionaries with the keys**

  – time

  – value

  – chi2a

  – all histograms as specified by their `name(..)` in user.f95

- **v** – the value to rescale the y values.

**Returns**
all rescaled datasets

**Note:** This naturally changes the cross section

`pymule.loader.setup(**kwargs)`

sets the default arguemnts 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 `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. `.*`.
- **filenames** – list, optional; list of files to loads, defaults to all files in **folder** (recurisvely if tar ball) Initialised to `None`, meaning everything.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram `name`. Initialised to to no merging, i.e. `{}`
- **types** – list of callables, optional; functions that convert the groups matched by `r` into python objects. Common examples would be `int` or `float`. Initialised to `[int,float, float]` as per McMule filename convention.
- **sanitycheck** – callable, optional; a function that, given a vegas dict, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.
- **cache** – folder name, optional; if existing folder, use as cache for compressed tarballs

**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, **kwargs)`

loads a `which_piece` and statistically combines the random seed.

**Parm piece**

str; `which_piece` to load

**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 `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. `.*`.
- **filenames** – list, optional; list of files to loads, defaults to all files in **folder** (recurisvely if tar ball) Initialised to `None`, meaning everything.
- **merge** – dict, optional: a dict of histograms `{'name': n}` to merge `n` bins in the histogram `name`. Initialised to to no merging, i.e. `{}`
- **types** – list of callables, optional; functions that convert the groups matched by `r` into python objects. Common examples would be `int` or `float`. Initialised to `[int,float, float]` as per McMule filename convention.
- **sanitycheck** – callable, optional; a function that, given a vegas dict, whether to include the file in the output (return `True`) or to skip (return `False`). Initialised to `lambda x : True`, i.e. include everything.
- **cache** – folder name, optional; if existing folder, use as cache for compressed tarballs

**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")
```

Load only observable `O3`

```
>>> sigma("m2enn0", obs="3")
```

## 13.5 Working with $\xi_c$ data

`pymule.xicut.`**`addkeyedsets`**(*sets*)

adds list of keyed sets using `addsets()`

`pymule.xicut.`**`get_errorbands`**(*x*, *coeff*, *covar*, *ndata*, *cf=0.9*)

evaluates the errorbands of the fit obtained by *`get_val()`*

**Parameters**

- **x** – iterable; values of $\xi_c$ 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*, *coeff*)

evaluates the fit obtained by `get_val()`

**Parameters**

- **xs** – iterable; values of $\xi_c$ 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`(*sets*, *scale=1.0*, *showfit=[True, True]*, *xlim=[-7, 0]*)

performs and FKS merge like `mergefks()` but it also produces a $\xi_c$ independence plot

**Note:** In contrast `mergefks()`, here phase-space partioned results need to be merged first. This is done by grouping those into an array first, sorted by number of particles in the final state, i.e. we start with the n-particle corrections.

**Parameters**

- **sets** – list of list of random-seed-merged results (usually from `sigma()`), starting with the lowest particle number and going up
- **scale** – float, optional; rescale factor for the plot and result
- **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 final set containing cross sections, distributions, and run-time information. The chi2a return is a list of the following

- the $\chi^2$ of the FKS merge
- a list of $\chi^2$ from previous operations, such as random seed merging or the integration.

**Example**
In the partioned muon-electron scattering case

```
>>> fig, res = mergefkswithplot([
...     [
...         sigma('em2emFEE'), sigma('em2emFMM'), sigma('em2emFEM')
...     ], [
...         sigma('em2emREE15'), sigma('em2emREE35'),
...         sigma('em2emRMM'),
...         sigma('em2emREM')
...     ]
... ])
```

`pymule.xicut.myfit`(*data*, *n*)

performs a log-polynomial $\sum_{i=0}^n a_i \log(\xi_c)^i$ fit to date

**Parameters**

- **data** – numpy array; The different $\xi_c$ 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`(*sets*, *n*, *xlim=[-7, 0]*, *scale=1*)

creates a residue plot for a $\xi_c$ fit

**Parameters**

- **sets** – dict or list; random-seed-merged results (usually from `sigma()`) or list thereof
- **n** – int; order of the fit, 1 at NLO, 2 at NNLO
- **xlim** – tuple of floats, optional; upper and lower bounds for $\log \xi_c$
- **scale** – float, optional; rescale factor for the plots

**Returns**
a figure and the fit coefficients as a matrix

## 13.6 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': '$\\rm LO$', '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': nlodata['thetae'],
...     'nnlo':nnlodata['thetae'],
...   },
...   labelx="$\theta_e\,/\,{\rm mrad}$",
...   labelsigma="$\D\sigma/\D\theta_e\ /\ {\rm\upmu b}$",
...   legend={
...     'lo': '$\sigma^{(0)}$',
...     'nlo': '$\sigma^{(1)}$',
...     'nnlo': '$\sigma^{(2)}$'
...   },
...   legendopts={'what': 'u', 'loc': 'lower right'}
... )
```

`pymule.plot.`**`setup_pgf`**`()`

setupf_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
- **`midleft`** – Nx3 numpy matrix or list thereof, optional; data plotted in the middle-left axes
- **`colmidleft`** – colour for middle-left data, defaults to colour scheme defined in *`colours`*
- **`labmidleft`** – 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 $\mathrm{d}\sigma/\mathrm{d}\theta_e$

```
>>> 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*, *fg*, *alpha*)

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.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.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.

## 13.7 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.

(**Mo**nte **c**arlo for **Mu**ons and other **le**ptons) 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 13.1 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 13.1: Processes implemented in McMule

| process | order | experiments | comments |
|---|---|---|---|
| $\mu\to\nu\bar\nu e$ | NNLO | MEG I&II | polarised, massified & exact |
| $\mu\to\nu\bar\nu e\gamma$ | NLO | MEG I | polarised |
| $\mu\to\nu\bar\nu eee$ | NLO | Mu3e | polarised |
| $\mu\to\nu\bar\nu e\gamma\gamma$ | LO | MEG | polarised |
| $\tau\to\nu\bar\nu e\gamma$ | NLO | BaBar | cuts in lab frame |
| $\tau\to\nu\bar\nu l\ell\ell$ | NLO | Belle II | |
| | NLO | MUonE | |
| | | NNLO | purely electronic corrections |
| | | | mixed (massified) |
| $\ell p\to\ell p$ | NNLO | P2, MUSE, Prad | only leptonic corrections |
| $e^-e^-\to e^-e^-$ | NNLO | Prad | complete |
| $e^+e^-\to e^+e^-$ | NNLO | | no $n_f$ |
| $e^+e^-\to\gamma\gamma$ | NNLO | PADME | |
| $e^+e^-\to\mu^+\mu^-$ | 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 is used in Section *Getting started*.

# Chapter 14

# 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 $\tau$ 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 \$udocker\$ 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 $\tau$ decays \$\tau \to e\gamma \nu \bar \nu \$ and \$\tau \to \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 $\mathcal B(\tau \to l\gamma \nu \bar \nu ,l=e,\mu )$ 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. "$\mathrm FKS^2$: 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. *\sc 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, 117
pymule.compress, 117
pymule.errortools, 97
pymule.loader, 106
pymule.maths, 117
pymule.mpl_axes_aligner, 117
pymule.mule, 117
pymule.plot, 113
pymule.vegas, 103
pymule.xicut, 111

# Index

## A

addkeyedsets() (*in module pymule.xicut*), 111
addplots() (*in module pymule.errortools*), 97
addsets() (*in module pymule.loader*), 106
alpha_composite() (*in module pymule.colours*), 117

## B

boost_back() (*fortran function*), **69**
boost_rf() (*fortran function*), **69**
BR, **53**

## C

callsanitised() (*in module pymule.loader*), 106
chisq() (*in module pymule.errortools*), 97
combineNplots() (*in module pymule.errortools*), 98
combineplots() (*in module pymule.errortools*), 98
commit_cache() (*in module pymule.loader*), 106
config file, **54**
container, **54**
containerisation, **54**
corner region, **54**
counter-event, **54**

## D

DiscB() (*fortran function*), **75**
DiscB_cplx() (*fortran function*), **75**
dividenumbers() (*in module pymule.errortools*), 99
divideplots() (*in module pymule.errortools*), 99

## E

eik() (*fortran function*), **73**
errorband() (*in module pymule.plot*), 113
euler_mat() (*fortran function*), **69**
event, **54**
EW, **53**
exportvegas() (*in module pymule.vegas*), 103

## F

filenamesuffix (*fortran variable*), **70**
fix_mu() (*fortran subroutine*), **70**
FKS, **53**
format_label_string_with_exponent() (*in module pymule.plot*), 113
FSR, **53**
full period, **54**

## G

generic pieces, **54**
generic processes, **55**
get_errorbands() (*in module pymule.xicut*), 111
get_val() (*in module pymule.xicut*), 112
getplots() (*in module pymule.vegas*), 103
guess_version() (*in module pymule.vegas*), 103

## H

hash_file() (*in module pymule.loader*), 106
HVP, **53**

## I

ieik() (*fortran function*), **73**
importreg() (*in module pymule.loader*), 106
importvegas() (*in module pymule.vegas*), 104
init_flavour() (*fortran subroutine*), **66**
inituser() (*fortran subroutine*), **70**
integratehistogram() (*in module pymule.errortools*), 99
IR, **53**
ISR, **53**

## K

kplot() (*in module pymule.plot*), 114

## L

Li2() (*in module pymule.maths*), 117
Li3() (*in module pymule.maths*), 117
LO, **53**
LP, **53**

## M

make_mlm() (*fortran function*), **73**
max_val (*fortran variable*), **70**
measurement function, **55**
menu file, **55**
mergebins() (*in module pymule.errortools*), 100
mergefks() (*in module pymule.loader*), 107
mergefkswithplot() (*in module pymule.xicut*), 112
mergenumbers() (*in module pymule.errortools*), 100
mergeplots() (*in module pymule.errortools*), 101

mergeseeds() (*in module pymule.loader*), 108
mergeset() (*in module pymule.loader*), 108
min_val (*fortran variable*), **70**
mlm (*fortran type*), **72**
module
    pymule.colours, 117
    pymule.compress, 117
    pymule.errortools, 97
    pymule.loader, 106
    pymule.maths, 117
    pymule.mpl_axes_aligner, 117
    pymule.mule, 117
    pymule.plot, 113
    pymule.vegas, 103
    pymule.xicut, 111
mulify() (*in module pymule.mule*), 117
multiintersect() (*in module pymule.loader*), 108
myfit() (*in module pymule.xicut*), 112

## N

names (*fortran variable*), **70**
NLO, **53**
NLP, **53**
NNLO, **53**
nr_bins (*fortran variable*), **70**
nr_q (*fortran variable*), **70**
NTS, **53**
ntssoft() (*fortran function*), **74**

## O

OS, **54**

## P

part() (*fortran function*), **73**
particle (*fortran type*), **72**
particles (*fortran type*), **72**
partInterface() (*fortran function*), **74**
parts() (*fortran function*), **73**
pass_cut (*fortran variable*), **70**
pattern() (*in module pymule.loader*), 109
PCS, **54**
PID, **54**
plusnumbers() (*in module pymule.errortools*), 101
printnumber() (*in module pymule.errortools*), 101
process group, **55**
PSD3() (*fortran subroutine*), **75**
PSD4() (*fortran subroutine*), **75**
PSD4_FKS() (*fortran subroutine*), **75**
PSD5() (*fortran subroutine*), **76**
PSD5_25() (*fortran subroutine*), **76**
PSD5_FKS() (*fortran subroutine*), **76**
PSD6() (*fortran subroutine*), **76**
PSD6_23_24_34() (*fortran subroutine*), **76**
PSD6_23_24_34_E56() (*fortran subroutine*), **77**
PSD6_25_26_m50_FKS() (*fortran subroutine*), **78**
PSD6_26_2x5() (*fortran subroutine*), **80**
PSD6_FKS() (*fortran subroutine*), **77**
PSD6_FKSS() (*fortran subroutine*), **78**
PSD6_P_25_26_m50_FKS() (*fortran subroutine*), **80**
PSD7() (*fortran subroutine*), **78**
PSD7_27_37_47_2x5_FKS() (*fortran subroutine*), **80**
PSD7_27_37_47_E56_FKS() (*fortran subroutine*), **78**
PSD7_27_37_47_FKS() (*fortran subroutine*), **78**
PSX2() (*fortran subroutine*), **79**
PSX3_35_FKS() (*fortran subroutine*), **79**
PSX3_coP13_35_FKS() (*fortran subroutine*), **81**
PSX3_coP_35_FKS() (*fortran subroutine*), **82**
PSX3_coP_45_FKS() (*fortran subroutine*), **82**
PSX3_FKS() (*fortran subroutine*), **79**
PSX3_P13_35_FKS() (*fortran subroutine*), **81**
PSX3_P_15_25_FKS() (*fortran subroutine*), **81**
PSX3_P_15_FKS() (*fortran subroutine*), **81**
PSX3_P_35_FKS() (*fortran subroutine*), **81**
PSX3_P_45_FKS() (*fortran subroutine*), **82**
PSX4() (*fortran subroutine*), **79**
PSX4_35_36_FKSS() (*fortran subroutine*), **80**
PSX4_coP13_35_36_FKSS() (*fortran subroutine*), **83**
PSX4_coP_35_36_FKSS() (*fortran subroutine*), **83**
PSX4_coP_45_46_FKSS() (*fortran subroutine*), **84**
PSX4_FKSS() (*fortran subroutine*), **79**
PSX4_P13_35_36_FKSS() (*fortran subroutine*), **83**
PSX4_P_15_16_25_26_FKSS() (*fortran subroutine*), **83**
PSX4_P_15_16_FKSS() (*fortran subroutine*), **82**
PSX4_P_35_36_FKSS() (*fortran subroutine*), **82**
PSX4_P_45_46_FKSS() (*fortran subroutine*), **83**
pymule.colours
    module, 117
pymule.compress
    module, 117
pymule.errortools
    module, 97
pymule.loader
    module, 106
pymule.maths
    module, 117
pymule.mpl_axes_aligner
    module, 117
pymule.mule
    module, 117
pymule.plot
    module, 113
pymule.vegas
    module, 103
pymule.xicut
    module, 111

## Q

quant() (*fortran function*), **71**

## R

random seed, **55**
read_record() (*in module pymule.vegas*), 105
real (*fortran type*), **65**
real (*fortran variable*), **65**
real() (*fortran function*), **65–68**
RNG, **54**

## S

ScalarC0() (*fortran function*), **75**
ScalarC0_cplx() (*fortran function*), **75**
ScalarC0IR6() (*fortran function*), **75**
ScalarC0IR6_cplx() (*fortran function*), **75**
ScalarD0IR16() (*fortran function*), **75**
ScalarD0IR16_cplx() (*fortran function*), **75**
scaleplot() (*in module pymule.errortools*), 102
scaleset() (*in module pymule.loader*), 109
scalesets() (*in module pymule.loader*), 109
setup() (*in module pymule.loader*), 110
setup_pgf() (*in module pymule.plot*), 115
SHA1, **54**
sigma() (*in module pymule.loader*), 110
SM, **54**
soft cut, **55**
submission script, **55**

## T

threepanel() (*in module pymule.plot*), 115
timesnumbers() (*in module pymule.errortools*), 102
twopanel() (*in module pymule.plot*), 115

## U

uncompress() (*in module pymule.compress*), 117
userdim (*fortran variable*), **70**
userevent() (*fortran subroutine*), **71**
userweight (*fortran variable*), **70**

## V

VP, **54**

## W

watermark() (*in module pymule.plot*), 116
write_record() (*in module pymule.vegas*), 105

## X

xiresidue() (*in module pymule.xicut*), 113

## Y

yaxes() (*in module pymule.mpl_axes_aligner*), 117