WHIZARD 3.0

A generic
Monte-Carlo integration and event generation package
for multi-particle processes

MANUAL [1]

Wolfgang Kilian, Thorsten Ohl, Jürgen Reuter, with contributions from
Fabian Bach, Simon Brass, Pia Bredt, Bijan Chokoufé Nejad, Christian Fleper,
Vincent Rothe, Sebastian Schmidt, Marco Sekulla, Christian Speckner, So
Young Shim, Florian Staub, Pascal Stienemeier, Christian Weiss

Universität Siegen, Emmy-Noether-Campus, Walter-Flex-Str. 3, D–57068 Siegen, Germany
Universität Würzburg, Emil-Hilb-Weg 22, D–97074 Würzburg, Germany
Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, D–22603 Hamburg, Germany
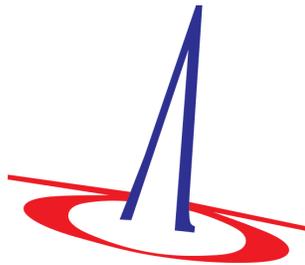
ABSTRACT

WHIZARD is a program system designed for the efficient calculation of multi-particle scattering cross sections and simulated event samples. The generated events can be written to file in various formats (including HepMC, LHEF, STDHEP, LCIO, and ASCII) or analyzed directly on the parton or hadron level using a built-in LaTeX-compatible graphics package.

Complete tree-level matrix elements are generated automatically for arbitrary partonic multi-particle processes by calling the built-in matrix-element generator O'Mega. Beyond hard matrix elements, WHIZARD can generate (cascade) decays with complete spin correlations. Various models beyond the SM are implemented, in particular, the MSSM is supported with an interface to the SUSY Les Houches Accord input format. Matrix elements obtained by alternative methods (e.g., including loop corrections) may be interfaced as well.

The program uses an adaptive multi-channel method for phase space integration, which allows to calculate numerically stable signal and background cross sections and generate unweighted event samples with reasonable efficiency for processes with up to eight and more final-state particles. Polarization is treated exactly for both the initial and final states. Quark or lepton flavors can be summed over automatically where needed.

For hadron collider physics, we ship the package with the most recent PDF sets from the MSTW/MMHT and CTEQ/CT10/CJ12/CJ15/CT14 collaborations. Furthermore, an interface to the LHAPDF library is provided.

For Linear Collider physics, beamstrahlung (CIRCE1, CIRCE2), Compton and ISR spectra are included for electrons and photons, including the most recent ILC and CLIC collider designs. Alternatively, beam-crossing events can be read directly from file.

For parton showering and matching/merging with hard matrix elements , fragmenting and hadronizing the final state, a first version of two different parton shower algorithms are included in the WHIZARD package. This also includes infrastructure for the MLM matching and merging algorithm. For hadronization and hadronic decays, PYTHIA and HERWIG interfaces are provided which follow the Les Houches Accord. In addition, the last and final version of (Fortran) PYTHIA is included in the package.

The WHIZARD distribution is available at

$$\text{https://whizard.hepforge.org}$$

where also the svn repository is located.

Contents

Chapter 1

Introduction

1.1   Disclaimer

This is a preliminary version of the WHIZARD manual. Many parts are still missing or incomplete, and some parts will be rewritten and improved soon. To find updated versions of the manual, visit the **WHIZARD** *website*

<div align="center">

https://whizard.hepforge.org

</div>

or consult the current version in the **svn** *repository on* **https://whizard.hepforge.org** *directly. Note, that the most recent version of the manual might contain information about features of the current* **svn** *version, which are not contained in the last official release version!*

For information that is not (yet) written in the manual, please consult the examples in the **WHIZARD** *distribution. You will find these in the subdirectory* **share/examples** *of the main directory where* **WHIZARD** *is installed. More information about the examples can be found on the* **WHIZARD** *Wiki page*

<div align="center">

https://whizard.hepforge.org/trac/wiki.

</div>
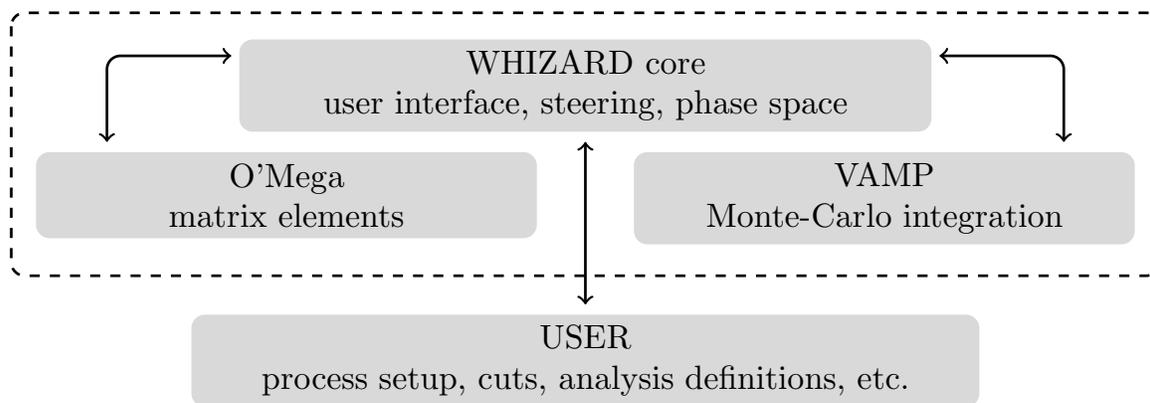
Figure 1.1: *General structure of the* `WHIZARD` *package.*

1.2 Overview

`WHIZARD` is a multi-purpose event generator that covers all parts of event generation (unweighted and weighted), either through intrinsic components or interfaces to external packages. Realistic collider environments are covered through sophisticated descriptions for beam structures at hadron colliders, lepton colliders, lepton-hadron colliders, both circular and linear machines. Other options include scattering processes e.g. for dark matter annihilation or particle decays. `WHIZARD` contains its in-house generator for (tree-level) high-multiplicity matrix elements, `O'Mega` that supports the whole Standard Model (SM) of particle physics and basically all possibile extensions of it. QCD parton shower describe high-multiplicity partonic jet events that can be matched with matrix elements. At the moment, only hadron collider parton distribution functions (PDFs) and hadronization are handled by packages not written by the main authors.

This manual is organized mainly along the lines of the way how to run `WHIZARD`: this is done through a command language, `SINDARIN` (Scripting INtegration, Data Analysis, Results display and INterfaces.) Though this seems a complication at first glance, the user is rewarded with a large possibility, flexibility and versatility on how to steer `WHIZARD`.

After some general remarks in the follow-up sections, in Chap. 2 we describe how to get the program, the package structure, the prerequisites, possible external extensions of the program and the basics of the installation (both as superuser and locally). Also, a first technical overview how to work with `WHIZARD` on single computer, batch clusters and farms are given. Furthermore, some rare uncommon possible build problems are discussed, and a tour through options for debugging, testing and validation is being made.

A first dive into the running of the program is made in Chap. 3. This is following by an extensive, but rather technical introduction into the steering language `SINDARIN` in Chap. 4. Here, the basic elements of the language like commands, statements, control structures, expressions and variables as well as the form of warnings and error messages are explained in detail.

Chap. 5 contains the application of the `SINDARIN` command language to the main tasks in running `WHIZARD` in a physics framework: the defintion of particles, subevents, cuts, and event selections. The specification of a particular physics models is discussed, while the next sections

are devoted to the setup and compilation of code for particular processes, the specification of beams, beam structure and polarization. The next step is the integration, controlling the integration, phase space, generator cuts, scales and weights, proceeding further to event generation and decays. At the end of this chapter, WHIZARD's internal data analysis methods and graphical visualization options are documented.

The following chapters are dedicated to the physics implemented in WHIZARD: methods for hard matrix interactions in Chap. 9. Then, in Chap. 10, implemented methods for adaptive multi-channel integration, particularly the integrator VAMP are explained, together with the algorithms for the generation of the phase-space in WHIZARD. Finally, an overview is given over the physics models implemented in WHIZARD and its matrix element generator O'Mega, together with possibilities for their extension. After that, the next chapter discusses parton showering, matching and hadronization as well as options for event normalizations and supported event formats. Also weighted event generation is explained along the lines with options for negative weights.

Chap. 12 is a stand-alone documentation of GAMELAN, the interal graphics support for the visualization of data and analysis. The next chapter, Chap. 14 details user interfaces: how to use more options of the WHIZARD command on the command line, how to use WHIZARD interactively, and how to include WHIZARD as a library into the user's own program.

Then, an extensive list of examples in Chap. 15 documenting physics examples from the LEP, SLC, HERA, Tevatron, and LHC colliders to future linear and circular colliders. This chapter is a particular good reference for the beginning, as the whole chain from choosing a model, setting up processes, the beam structure, the integration, and finally simulation and (graphical) analysis are explained in detail.

More technical details about efficiency, tuning and advance usage of WHIZARD are collected in Chap. 16. Then, Chap. 17 shows how to set up your own new physics model with the help of external programs like SARAH or FeynRules program or the Universal Feynrules Output, UFO, and include it into the WHIZARD event generator.

In the appendices, we e.g. give an exhaustive reference list of SINDARIN commands and built-in variables.

Please report any inconsistencies, bugs, problems or simply pose open questions to our contact whizard@desy.de.

There is now also a support page on Launchpad, which offers support that is easily visible for the whole user community: https://launchpad.net/whizard.

1.3 Historical remarks

This section gives a historical overview over the development of WHIZARD and can be easily skipped in a (first) reading of the manual. WHIZARD has been developed in a first place as a tool for the physics at the then planned linear electron-positron collider TESLA around 1999. The intention was to have a tool at hand to describe electroweak physics of multiple weak bosons and the Higgs boson as precise as possible with full matrix elements. Hence, the acronym: WHiZard, which stood for **W**, **H**iggs, **Z**, **a**nd **r**espective **d**ecays.

Several components of the WHIZARD package that are also available as independent sub-packages have been published already before the first versions of the WHIZARD generator itself: the multi-channel adaptive Monte-Carlo integration package VAMP has been released mid 1998 [5]. The dedicated packages for the simulation of linear lepton collider beamstrahlung and the option for a photon collider on Compton backscattering (CIRCE1/2) date back even to mid 1996 [6]. Also parts of the code for WHIZARD's internal graphical analysis (the gamelan module) came into existence already around 1998.

After first inofficial versions, the official version 1 of WHIZARD was release in the year 2000. The development, improvement and incorporation of new features continued for roughly a decade. Major milestones in the development were the full support of all kinds of beyond the Standard Model (BSM) models including spin 3/2 and spin 2 particles and the inclusion of the MSSM, the NMSSM, Little Higgs models and models for anomalous couplings as well as extra-dimensional models from version 1.90 on. In the beginning, several methods for matrix elements have been used, until the in-house matrix element generator O'Mega became available from version 1.20 on. It was included as a part of the WHIZARD package from version 1.90 on. The support for full color amplitudes came with version 1.50, but in a full-fledged version from 2.0 on. Version 1.40 brought the necessary setups for all kinds of collider environments, i.e. asymmetric beams, decay processes, and intrinsic p_T in structure functions.

Version 2.0 was released in April 2010 as an almost complete rewriting of the original code. It brought the construction of an internal density-matrix formalism which allowed the use of factorized production and (cascade) decay processes including complete color and spin correlations. Another big new feature was the command-line language SINDARIN for steering all parts of the program. Also, many performance improvement have taken place in the new release series, like OpenMP parallelization, speed gain in matrix element generation etc. Version 2.2 came out in May 2014 as a major refactoring of the program internals but keeping (almost everywhere) the same user interface. New features are inclusive processes, reweighting, and more interfaces for QCD environments (BLHA/HOPPET).

The following tables shows some of the major steps (physics implementation and/or technical improvements) in the development of WHIZARD(we break the table into logical and temporal blocks of WHIZARD development).

`WHIZARD 1`, first line of development, ca. 1998-2010:

0.99	08/1999	Beta version
1.00	12/2000	First public version
1.10	03/2001	Libraries; `PYTHIA6` interface
1.11	04/2001	PDF support; anomalous couplings
1.20	02/2002	`O'Mega` matrix elements; `CIRCE` support
1.22	03/2002	QED ISR; beam remnants, phase space improvements
1.25	05/2003	MSSM; weighted events; user-code plug-in
1.28	04/2004	Improved phase space; SLHA interface; signal catching
1.30	09/2004	Major technical overhaul
1.40	12/2004	Asymmetric beams; decays; p_T in structure functions
1.50	02/2006	QCD support in `O'Mega` (color flows); LHA format
1.51	06/2006	Hgg, $H\gamma\gamma$; Spin $3/2 + 2$; BSM models
1.90	11/2007	`O'Mega` included; LHAPDF support; Z'; WW scattering
1.92	03/2008	LHE format; UED; parton shower beta version
1.93	04/2009	NMSSM; SLHA2 accord; improved color/flavor sums
1.95	02/2010	MLM matching; development stop in version 1
1.97	05/2011	Manual for version 1 completed.

`WHIZARD 2.0-2.2`: first major refactoring and early new release, ca. 2007-2015:

2.0.0	04/2010	Major refactoring: automake setup; dynamic libraries improved speed; cascades; OpenMP; `SINDARIN` steering language
2.0.3	07/2010	QCD ISR+FSR shower; polarized beams
2.0.5	05/2011	Builtin PDFs; static builds; relocation scripts
2.0.6	12/2011	Anomalous top couplings; unit tests
2.1.0	06/2012	Analytic ISR+FSR parton shower; anomalous Higgs couplings
2.2.0	05/2014	Major technical refactoring: abstract object-orientation; THDM; reweighting; LHE v2/3; BLHA; HOPPET interface; inclusive processes
2.2.1	05/2014	CJ12 PDFs; FastJet interface
2.2.2	07/2014	LHAPDF6 support; correlated LC beams; GuineaPig interface
2.2.3	11/2014	O'Mega virtual machine; lepton collider top pair threshold; Higgs singlet extension
2.2.4	02/2015	LCIO support; progress on NLO; many technical bug fixes
2.2.7	08/2015	progress on POWHEG; fixed-order NLO events; revalidation of ILC event chain
2.2.8	11/2015	support for quadruple precision; StdHEP included; SM dim 6 operators supported

`WHIZARD 2.3-2.8`, completion of refactoring, continuous development, ca. 2015-2020:

2.3.0	07/2016	NLO: resonance mappings for FKS subtraction;
		more advanced cascade syntax;
		GUI (α version); UFO support (α version); ILC v1.9x-v2.x final validation
2.3.1	08/2016	Complex mass scheme
2.4.0	11/2016	Refactoring of NLO setup
2.4.1	03/2017	α version of new VEGAS implementation
2.5.0	05/2017	Full UFO support (SM-like models)
2.6.0	09/2017	MPI parallel integration and event generation; resonance histories
		for showers; RECOLA support
2.6.1	11/2017	EPA/ISR transverse distributions, handling of shower resonances;
		more efficient (alternative) phase space generation
2.6.2	12/2017	Hee coupling, improved resonance matching
2.6.3	02/2018	Partial NLO refactoring for quantum numbers,
		unified RECOLA 1/2 interface.
2.6.4	08/2018	Gridpack functionality; Bug fixes: color flows, HSExt model, MPI setup
2.7.0	01/2019	PYTHIA8 interface, process setup refactoring, RAMBO PS option;
		gfortran 5.0+ necessary
2.8.0	08/2019	(Almost) complete UFO support, general Lorentz structures, n-point vertices
2.8.1	09/2019	HepMC3, NLO QCD pp (almost) complete, b/c jet selection, photon isolation
2.8.2	10/2019	Support for OCaml $\geq$ 4.06.0, UFO Spin-2 support, LCIO alternative weights
2.8.3	07/2020	UFO Majorana feature complete, many e^+e^- related improvements
2.8.4	07/2020	Bug fix for UFO Majorana models
2.8.5	09/2020	Bug fix for polarizations in $H \rightarrow \tau\tau$

`WHIZARD 3.0` and onwards, the NLO series:

3.0.0	04/2021	NLO QCD automation & UFO Majorana support released
3.0.1	07/2021	MPI load balancer, rescan of ILC mass production samples
3.0.2	11/2021	NLO EW for pp processes, sums/products in `SINDARIN`
3.0.3	04/2022	NLO EW/QCD mixed processes, NLL electron PDFs

For a detailed overview over the historical development of the code confer the `ChangeLog` file and the commit messages in our revision control system repository.

1.4   About examples in this manual

Although WHIZARD has been designed as a Monte Carlo event generator for LHC physics, several elementary steps and aspects of its usage throughout the manual will be demonstrated with the famous textbook example of $e^+e^- \to \mu^+\mu^-$. This is the same process, the textbook by Peskin/Schroeder [58] uses as a prime example to teach the basics of quantum field theory. We use this example not because it is very special for WHIZARD or at the time being a relevant physics case, but simply because it is the easiest fundamental field theoretic process without the complications of structured beams (which can nevertheless be switched on like for ISR and beamstrahlung!), the need for jet definitions/algorithms and flavor sums; furthermore, it easily accomplishes a demonstration of polarized beams. After the basics of WHIZARD usage have been explained, we move on to actual physics cases from LHC (or Tevatron).

Chapter 2

Installation

2.1 Package Structure

`WHIZARD` is a software package that consists of a main executable program (which is called `whizard`), libraries, auxiliary executable programs, and machine-independent data files. The whole package can be installed by the system administrator, by default, on a central location in the file system (`/usr/local` with its proper subdirectories). Alternatively, it is possible to install it in a user's home directory, without administrator privileges, or at any other location.

A `WHIZARD` run requires a workspace, i.e., a writable directory where it can put generated code and data. There are no constraints on the location of this directory, but we recommend to use a separate directory for each `WHIZARD` project, or even for each `WHIZARD` run.

Since `WHIZARD` generates the matrix elements for scattering and decay processes in form of `Fortran` code that is automatically compiled and dynamically linked into the running program, it requires a working `Fortran` compiler not just for the installation, but also at runtime.

The previous major version `WHIZARD1` did put more constraints on the setup. In a nutshell, not just the matrix element code was compiled at runtime, but other parts of the program as well, so the whole package was interleaved and had to be installed in user space. The workflow was controlled by `make` and PERL scripts. These constraints are gone in the present version in favor of a clean separation of installation and runtime workspace.

2.2 Prerequisites

2.2.1 No Binary Distribution

`WHIZARD` is currently not distributed as a binary package, nor is it available as a debian or RPM package. This might change in the future. However, compiling from source is very simple (see below). Since the package needs a compiler also at runtime, it would not work without some development tools installed on the machine, anyway.

Note, however, that we support an install script, that downloads all necessary prerequisites, and does the configuration and compilation described below automatically. This is called the "instant WHIZARD" and is accessible through the WHIZARD webpage from version

2.1.1 on: https://whizard.hepforge.org/versions/install/install-whizard-2.X.X.sh.
Download this shell script, make it executable by

```
chmod +x install-whizard-2.X.X.sh
```

and execute it. Note that this also involves compilation of the required `Fortran` compiler
which takes 1-3 hours depending on your system. `Darwin` operating systems (a.k.a. as `Mac
OS X`) have a very similar general system for all sorts of software, called `MacPorts` (http:
//www.macports.org). This offers to install `WHIZARD` as one of its software ports, and is very
similar to "instant WHIZARD" described above.

2.2.2   Tarball Distribution

This is the recommended way of obtaining `WHIZARD`. You may download the current stable
distribution from the `WHIZARD` webpage, hosted at the HepForge webpage

    https://whizard.hepforge.org

The distribution is a single file, say `whizard-3.0.3.tgz` for version 3.0.3.
    You need the additional prerequisites:

- GNU `tar` (or `gunzip` and `tar`) for unpacking the tarball.

- The `make` utility. Other standard Unix utilities (`sed`, `grep`, etc.) are usually installed by
  default.

- A modern `Fortran` compiler (see Sec. 2.2.6 for details).

- The `OCaml` system. `OCaml` is a functional and object-oriented language. Version 4.02.3 or
  newer is required to compile all components of `WHIZARD`. The package is freely available
  either as a debian/RPM package on your system (it might be necessary to install it from
  the usual repositories), or you can obtain it directly from

      http://caml.inria.fr

  and install it yourself. If desired, the package can be installed in user space without
  administrator privileges[1].

The following optional external packages are not required, but used for certain purposes. Make
sure to check whether you will need any of them, before you install `WHIZARD`.

- LaTeX and `MetaPost` for data visualization. Both are part of the TeX program family. These
  programs are not absolutely necessary, but `WHIZARD` will lack the tools for visualization
  without them.

---

[1]Unfortunately, the version of the `OCaml` compiler from 3.12.0 broke backwards compatibility. Therefore,
versions of `O'Mega`/`WHIZARD` up to 2.0.2 only compile with older versions (3.11.x works). This has been fixed in
versions 2.0.3 and later. See also Sec. 3.4.1. `WHIZARD` versions up to 2.7.1 were still backwards compatible with
`OCaml` 3.12.0

- The `LHAPDF` structure-function library. See Sec. 2.2.7.

- The `HOPPET` structure-function matching tool. See Sec. 2.2.8.

- The `HepMC` event-format package. See Sec. 2.2.9.

- The `FastJet` jet-algorithm package. See Sec. 2.2.12.

- The `LCIO` event-format package. See Sec. 2.2.14.

Until version v2.2.7 of `WHIZARD`, the event-format package `STDHEP` used to be available as an external package. As their distribution is frozen with the final version v5.06.01, and it used to be notoriously difficult to compile and link `STDHEP` into `WHIZARD`, it was decided to include `STDHEP` into `WHIZARD`. This is the case from version v2.2.8 of `WHIZARD` on. Linking against an external version of `STDHEP` is precluded from there on. Nevertheless, we list some explanations in Sec. 2.2.13, particularly on the need to install the `libtirpc` headers for the legacy support of this event format. Once these prerequisites are met, you may unpack the package in a directory of your choice

```
some-directory> tar xzf whizard-3.0.3.tgz
```

and proceed.[2]

For using external physics models that are directly supported by `WHIZARD` and `O'Mega`, the user can use tools like `SARAH` or `FeynRules`. There installation and linking to `WHIZARD` will be explained in Chap. 17. Besides this, also new models can be conveniently included via `UFO` files, which will be explained as well in that chapter.

The directory will then contain a subdirectory `whizard-3.0.3` where the complete source tree is located. To update later to a new version, repeat these steps. Each new version will unpack in a separate directory with the appropriate name.

2.2.3 SVN Repository Version

If you want to install the latest development version, you have to check it out from the `WHIZARD` SVN repository. Note that since a couple of years our development is now via a Git revision control system hosted at the University of Siegen, cf. the next subsection.

In addition to the prerequisites listed in the previous section, you need:

- The `subversion` package (`svn`), the tool for dealing with SVN repositories.

- The `autoconf` package, part of the `autotools` development system. `automake` is needed with version `1.12.2` or newer.

- The `noweb` package, a light-weight tool for literate programming. This package is nowadays often part of Linux distributions[3]. You can obtain the source code from[4]

---

[2]Without GNU `tar`, this would read `gunzip -c whizard-3.0.3.tgz | tar xz -`

[3]In Ubuntu from version 10.04 on, and in Debian since squeeze. For `Mac OS X`, `noweb` is available via the `MacPorts` system.

[4]Please, do not use any of the binary builds from this webpage. Probably all of them are quite old and broken.

http://www.cs.tufts.edu/~nr/noweb/

To start, go to a directory of your choice and execute

```
your-src-directory> svn checkout
svn+ssh://vcs@phab.hepforge.org/source/whizardsvn/trunk \;\; .
```

Note that for the time being after the HepForge system modernization early September 2018, a HepForge account with a local ssl key is necessary to checkout the subversion repository. This is enforced by the phabricator framework of HepForge, and will hopefully be relaxed in the future. The SVN source tree will appear in the current directory. To update later, you just have to execute

```
your-src-directory> svn update
```

within that directory.

After checking out the sources, you first have to create `configure.ac` by executing the shell script `build_master.sh`. In order to build the `configure` script, the `autotools` package `autoreconf` has to be run. On some `Unix` systems the `RPC` headers needed for the legacy support of the `STDHEP` event format are provided by the `TIRPC` library (cf. Sec. 2.2.13). To easily check for them, `configure.ac` processed by `autoreconf` makes use of the `pkg-config` tool which needs to be installed for the developer version. So now, run[5]

```
your-src-directory> autoreconf
```

This will generate a `configure` script.

2.2.4   Public Git Repository Version

Since a couple of years, development of WHIZARD is done by means of a Git revision system, hosted at the University of Siegen. There is a public mirror of that Git repository available at

https://gitlab.tp.nt.uni-siegen.de/whizard/public

Cloning via HTTPS brings the user to the same change as the SVN checkout from HepForge described in the previous subsection:

git clone https://gitlab.tp.nt.uni-siegen.de/whizard/public.git

The next steps are the same as described in the previous subsection.

2.2.5   Nightly development snapshots

Nightly development snapshots that are pre-packaged in the same way as an official distribution are available from

https://whizard.tp.nt.uni-siegen.de/

Building WHIZARD works the way as described in Sec. 2.2.2.

---

[5]At least, version 2.65 of the `autoconf` package is required.

2.2.6 Fortran Compilers

`WHIZARD` is written in modern `Fortran`. To be precise, it uses a subset of the `Fortran2003` standard. At the time of this writing, this subset is supported by, at least, the following compilers:

- `gfortran` (GNU, Open Source). You will need version 5.1.0 or higher[6]. We recommend to use at least version 5.4 or higher, as especially the the early version of the `gfortran` experience some bugs. `gfortran` 6.5.0 has a severe regression and cannot be used.

- `nagfor` (NAG). You will need version 6.2 or higher.

- `ifort` (Intel). You will need version 19.0.2 or higher

2.2.7 LHAPDF

For computing scattering processes at hadron colliders such as the LHC, `WHIZARD` has a small set of standard structure-function parameterizations built in, cf. Sec. 5.5.4. For many applications, this will be sufficient, and you can skip this section.

However, if you need structure-function parameterizations that are not in the default set (e.g. PDF error sets), you can use the `LHAPDF` structure-function library, which is an external package. It has to be linked during `WHIZARD` installation. For use with `WHIZARD`, version 5.3.0 or higher of the library is required[7]. The `LHAPDF` package has undergone a major rewriting from `Fortran` version 5 to `C++` version 6. While still maintaining the interface for the `LHAPDF` version 5 series, from version 2.2.2 of `WHIZARD` on, the new release series of `LHAPDF`, version 6.0 and higher, is also supported.

If `LHAPDF` is not yet installed on your system, you can download it from

  https://lhapdf.hepforge.org

for the most recent LHAPDF version 6 and newer, or

  https://lhapdf.hepforge.org/lhapdf5

for version 5 and older, and install it. The website contains comprehensive documentation on the configuring and installation procedure. Make sure that you have downloaded and installed not just the package, but also the data sets. Note that `LHAPDF` version 5 needs both a `Fortran` and a `C++` compiler.

During `WHIZARD` configuration, `WHIZARD` looks for the script `lhapdf` (which is present in `LHAPDF` series 6) first, and then for `lhapdf-config` (which is present since `LHAPDF` version 4.1.0): if those are in an executable path (or only the latter for `LHAPDF` version 5), the environment variables for `LHAPDF` are automatically recognized by `WHIZARD`, as well as the version number. This should look like this in the `configure` output (for `LHAPDF` version 6 or newer),

---

[6]Note that `WHIZARD` versions 2.0.0 until 2.3.1 compiled with `gfortran` 4.7.4, but the object-oriented refactoring of the `WHIZARD` code from 2.4.0 on until version 2.6.5 made a switch to `gfortran` 4.8.4 or higher necessary. In the same way, since version 2.7.0, `gfortran` 5.1.0 or newer is needed

[7]Note that PDF sets which contain photons as partons are only supported with `WHIZARD` for `LHAPDF` version 5.7.1 or higher

```
configure: ----------------------------------------------------------------
configure: --- LHAPDF ---
configure:
checking for lhapdf... /usr/local/bin/lhapdf
checking for lhapdf-config... /usr/local/bin/lhapdf-config
checking the LHAPDF version... 6.2.1
checking the major version... 6
checking the LHAPDF pdfsets path... /usr/local/share/LHAPDF
checking the standard PDF sets...  all standard PDF sets installed
checking if LHAPDF is functional... yes
checking LHAPDF... yes
configure: ----------------------------------------------------------------
```

while for LHAPDF version 5 and older it looks like this:

```
configure: ----------------------------------------------------------------
configure: --- LHAPDF ---
configure:
checking for lhapdf... no
checking for lhapdf-config... /usr/local/bin/lhapdf-config
checking the LHAPDF version... 5.9.1
checking the major version... 5
checking the LHAPDF pdfsets path... /usr/local/share/lhapdf/PDFsets
checking the standard PDF sets...  all standard PDF sets installed
checking for getxminm in -lLHAPDF... yes
checking for has_photon in -lLHAPDF... yes
configure: ----------------------------------------------------------------
```

If you want to use a different LHAPDF (e.g. because the one installed on your system by default is an older one), the preferred way to do so is to put the lhapdf (and/or lhapdf-config) scripts in an executable path that is checked before the system paths, e.g. <home>/bin.

For the old series, LHAPDF version 5, a possible error could arise if LHAPDF had been compiled with a different Fortran compiler than WHIZARD, and if the run-time library of that Fortran compiler had not been included in the WHIZARD configure process. The output then looks like this:

```
configure: ----------------------------------------------------------------
configure: --- LHAPDF ---
configure:
checking for lhapdf... no
checking for lhapdf-config... /usr/local/bin/lhapdf-config
checking the LHAPDF version... 5.9.1
checking the major version... 5
checking the LHAPDF pdfsets path... /usr/local/share/lhapdf/PDFsets
checking for standard PDF sets... all standard PDF sets installed
checking for getxminm in -lLHAPDF... no
checking for has_photon in -lLHAPDF... no
configure: ----------------------------------------------------------------
```

So, the WHIZARD configure found the LHAPDF distribution, but could not link because it could not resolve the symbols inside the library. In case of failure, for more details confer the config.log.

If LHAPDF is installed in a non-default directory where WHIZARD would not find it, set the environment variable LHAPDF_DIR to the correct installation path when configuring WHIZARD.

The check for the standard PDF sets are those sets that are used in the default WHIZARD self tests in the case LHAPDF is enabled and correctly linked. If some of them are missing, then this test will result in a failure. They are the CT10 set for LHAPDF version 6 (for version 5, cteq61.LHpdf, cteq6ll.LHpdf, cteq5l.LHgrid, and GSG961.LHgrid are demanded). If you want to use LHAPDF inside WHIZARD please install them such that WHIZARD could perform all its sanity checks with them. The last check is for the has_photon flag, which tests whether photon PDFs are available in the found LHAPDF installation.

2.2.8 HOPPET

HOPPET (not Hobbit) is a tool for the QCD DGLAP evolution of PDFs for hadron colliders. It provides possibilities for matching algorithms for 4- and 5-flavor schemes, that are important for precision simulations of *b*-parton initiated processes at hadron colliders. If you are not interested in those features, you can skip this section. Note that this feature is not enabled by default (unlike e.g. LHAPDF), but has to be explicitly during the configuration (see below):

```
your-build-directory> your-src-directory/configure --enable-hoppet
```

If you configure messages like the following:

```
configure: ----------------------------------------------------------------
configure: --- HOPPET ---
configure:
checking for hoppet-config... /usr/local/bin/hoppet-config
checking for hoppetAssign in -lhoppet_v1... yes
checking the HOPPET version... 1.2.0
configure: ----------------------------------------------------------------
```

then you know that HOPPET has been found and was correctly linked. If that is not the case, you have to specify the location of the HOPPET library, e.g. by adding

```
HOPPET=<hoppet\_directory>/lib
```

to the configure options above. For more details, please confer the HOPPET manual.

2.2.9 HepMC

With version 2.8.1, WHIZARD supports both the "classical" version 2 as well as the newly designed version 3 (release 2019). The configure step can successfully recognize the two different versions, the user do not have to specify which version is installed.

HepMC is a C++ class library for handling collider scattering events. In particular, it provides a portable format for event files. If you want to use this format, you should link WHIZARD with HepMC, otherwise you can skip this section.

If it is not already installed on your system, you may obtain `HepMC` from one of these two webpages:

http://hepmc.web.cern.ch/hepmc/

or

http://hepmc.web.cern.ch/hepmc/

If the `HepMC` library is linked with the installation, `WHIZARD` is able to read and write files in the `HepMC` format.

Detailed information on the installation and usage can be found on the `HepMC` homepage. We give here only some brief details relevant for the usage with `WHIZARD`: For the compilation of HepMC one needs a `C++` compiler. Then the procedure is the same as for the `WHIZARD` package, namely configure HepMC:

```
configure --with-momentum=GEV --with-length=MM --prefix=<install dir>
```

Note that the particle momentum and decay length flags are mandatory, and we highly recommend to set them to the values `GEV` and `MM`, respectively. After configuration, do `make`, an optional `make check` (which might sometimes fail for non-standard values of momentum and length), and finally `make install`.

The latest version of `HepMC` (2.6.10) as well as the new relase series use `cmake` for their build process. For more information, confer the `HepMC` webpage.

A `WHIZARD` configuration for `HepMC` looks like this:

```
configure: ----------------------------------------------------------------
configure: --- HepMC ---
configure:
checking for HepMC-config... no
checking HepMC3 or newer... no
configure: HepMC3 not found, incompatible, or HepMC-config not found
configure: looking for HepMC2 instead ...
checking the HepMC version... 2.06.10
checking for GenEvent class in -lHepMC... yes
configure: ----------------------------------------------------------------
```

If `HepMC` is installed in a non-default directory where `WHIZARD` would not find it, set the environment variable `HEPMC_DIR` to the correct installation path when configuring `WHIZARD`. Furthermore, the environment variable `CXXFLAGS` allows you to set specific `C/C++` preprocessor flags, e.g. non-standard include paths for header files.

A typical configuration of `HepMC3` will look like this:

```
configure: ----------------------------------------------------------------
configure: --- ROOT ---
configure:
checking for root-config... /usr/local/bin/root-config
checking for root... /usr/local/bin/root
checking for rootcint... /usr/local/bin/rootcint
```

```
checking for dlopen in -ldl... (cached) yes
configure: ---------------------------------------------------------------
configure: --- HepMC ---
configure:
checking for HepMC3-config... /usr/local/bin/HepMC3-config
checking if HepMC3 is built with ROOT interface... yes
checking if HepMC3 is functional... yes
checking for HepMC3... yes
checking the HepMC3 version... 3.02.01
configure: ---------------------------------------------------------------
```

As can be seen, `WHIZARD` will check for the `ROOT` environment as well as whether `HepMC3` has been built with support for the `ROOT` and `RootTree` writer classes. This is an easy option to use `WHIZARD` to write out `ROOT` events. For more information see Sec. 13.1.

2.2.10 PYTHIA6

The `WHIZARD` package ships with the final version of the old `PYTHIA6` release series, v6.427. This is no longer maintained, but many analyses are still set up for this shower and hadronization tool, so `WHIZARD` offers the possibility of backwards compatibility here.

> configure: ———————————————————————— configure: — SHOWERS PYTHIA6 PYTHIA8 MPI — configure: checking whether we want to enable PYTHIA6... yes checking for PYTHIA6... (enabled) checking for PYTHIA6 eh settings... (disabled)

`WHIZARD` automatically compiles `PYTHIA6`, it has not to be specifically enabled by the user.

In order to properly use `PYTHIA6` for high-energy electron-hadron collisions which allow much further forward regions to be explored as old experiments like HERA, there is a special switch to enable those specific settings for *eh*-colliders:

> `--enable-pythia6_ep`

Those settings have been provided by [59].

2.2.11 PYTHIA8

`PYTHIA8` is a `C++` class library for handling hadronization, showering and underlying event. If you want to use this feature (once it is fully supported in `WHIZARD`), you should link `WHIZARD` with `PYTHIA8`, otherwise you can skip this section.

If it is not already installed on your system, you may obtain `PYTHIA8` from

> http://home.thep.lu.se/~torbjorn/Pythia.html

If the `PYTHIA8` library is linked with the installation, `WHIZARD` will be able to use its hadronization and showering, once this is fully supported within `WHIZARD`.

To link a `PYTHIA8` installation to `WHIZARD`, you should specify the flag

```
      --enable-pythia8
```

to `configure`. If PYTHIA8 is installed in a non-default directory where WHIZARD would not find it, specify also

```
      --with-pythia8=<your-pythia8-installation-path>
```

A successful WHIZARD configuration should produce a screen output similar to this:

```
configure: ----------------------------------------------------------------
configure: --- SHOWERS PYTHIA6 PYTHIA8 MPI ---
configure:
[....]
checking for pythia8-config... /usr/local/bin/pythia8-config
checking if PYTHIA8 is functional... yes
checking PYTHIA8... yes
configure: WARNING: PYTHIA8 configure is for testing purposes at the moment.
configure: ----------------------------------------------------------------
```

2.2.12   FastJet

FastJet is a `C++` class library for handling jet clustering. If you want to use this feature, you should link WHIZARD with FastJet, otherwise you can skip this section.

If it is not already installed on your system, you may obtain FastJet from

> http://fastjet.fr

If the FastJet library is linked with the installation, WHIZARD is able to call the jet algorithms provided by this program for the purposes of applying cuts and analysis.

To link a FastJet installation to WHIZARD, you should specify the flag

```
      --enable-fastjet
```

to `configure`. If FastJet is installed in a non-default directory where WHIZARD would not find it, specify also

```
      --with-fastjet=<your-fastjet-installation-path>
```

A successful WHIZARD configuration should produce a screen output similar to this:

```
configure: ----------------------------------------------------------------
configure: --- FASTJET ---
configure:
checking for fastjet-config... /usr/local/bin/fastjet-config
checking if FastJet is functional... yes
checking FastJet... yes
checking the FastJet version... 3.3.4
configure: ----------------------------------------------------------------
```

Note that when compiling on Darwin/macOS it might be necessary to set the option `--disable-auto-ptr` when compiling with `clang++`.

2.2.13 STDHEP

STDHEP is a library for handling collider scattering events [60]. In particular, it provides a portable format for event files. Until version 2.2.7 of WHIZARD, STDHEP that was maintained by Fermilab, could be linked as an externally compiled library. As the STDHEP package is frozen in its final release v5.06.1 and no longer maintained, it has from version 2.2.8 been included WHIZARD. This eases many things, as it was notoriously difficult to compile and link STDHEP in a way compatible with WHIZARD. Not the full package has been included, but only the libraries for file I/O (mcfio, the library for the XDR conversion), while the various translation tools for PYTHIA, HERWIG, etc. have been abandoned. Note that STDHEP has largely been replaced in the hadron collider community by the HepMC format, and in the lepton collider community by LCIO. WHIZARD might serve as a conversion tools for all these formats, but other tools also exist, of course. Note that the mcfio framework makes use of the RPC headers. These come – provided by SunOS/Oracle America, Inc. – together with the system headers, but on some Unix systems (e.g. ArchLinux, Fedora) have been replaced by the libtirpc headers . The configure script searches for these headers so these have to be installed mandatorily.

If the STDHEP library is linked with the installation, WHIZARD is able to write files in the STDHEP format, the corresponding configure output notifies you that STDHEP is always included:

```
configure: ----------------------------------------------------------------
configure: --- STDHEP ---
configure:
checking for pkg-config... /opt/local/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking for libtirpc... no
configure: for StdHEP legacy code: using SunRPC headers and library
configure: StdHEP v5.06.01 is included internally
configure: ----------------------------------------------------------------
```

2.2.14 LCIO

LCIO is a C++ class library for handling collider scattering events. In particular, it provides a portable format for event files. If you want to use this format, you should link WHIZARD with LCIO, otherwise you can skip this section.

If it is not already installed on your system, you may obtain LCIO from:

> http://lcio.desy.de

If the LCIO library is linked with the installation, WHIZARD is able to read and write files in the LCIO format.

Detailed information on the installation and usage can be found on the LCIO homepage. We give here only some brief details relevant for the usage with WHIZARD: For the compilation of LCIO one needs a C++ compiler. LCIO is based on cmake. For the corresponding options please confer the LCIO manual.

A WHIZARD configuration for LCIO looks like this:

```
configure: ----------------------------------------------------------------
```

```
configure: --- LCIO ---
configure:
checking the LCIO version... 2.12.1
checking for LCEventImpl class in -llcio... yes
configure: ----------------------------------------------------------------
```

If `LCIO` is installed in a non-default directory where `WHIZARD` would not find it, set the environment variable `LCIO` or `LCIO_DIR` to the correct installation path when configuring `WHIZARD`. The first one is the variable exported by the `setup.sh` script while the second one is analogous to the environment variables of other external packages. `LCIO` takes precedence over `LCIO_DIR`. Furthermore, the environment variable `CXXFLAGS` allows you to set specific `C/C++` preprocessor flags, e.g. non-standard include paths for header files.

2.3   Installation

Once you have unpacked the source (either the tarball or the SVN version), you are ready to compile it. There are several options.

2.3.1   Central Installation

This is the default and recommended way, but it requires adminstrator privileges. Make sure that all prerequisites are met (Sec. 2.2).

1. Create a fresh directory for the `WHIZARD` build. It is recommended to keep this separate from the source directory.

2. Go to that directory and execute

   ```
   your-build-directory> your-src-directory/configure
   ```

   This will analyze your system and prepare the compilation of `WHIZARD` in the build directory. Make sure to set the proper options to `configure`, see Sec. 2.3.3 below.

3. Call `make` to compile and link `WHIZARD`:

   ```
   your-build-directory> make
   ```

4. If you want to make sure that everything works, run

   ```
   your-build-directory> make check
   ```

   This will take some more time.

5. Become superuser and say

   ```
   your-build-directory> make install
   ```

`WHIZARD` should now installed in the default locations, and the executable should be available in the standard path. Try to call `whizard --help` in order to check this.

2.3.2 Installation in User Space

You may lack administrator privileges on your system. In that case, you can still install and run `WHIZARD`. Make sure that all prerequisites are met (Sec. 2.2).

1. Create a fresh directory for the `WHIZARD` build. It is recommended to keep this separate from the source directory.

2. Reserve a directory in user space for the `WHIZARD` installation. It should be empty, or yet non-existent.

3. Go to that directory and execute

   ```
   your-build-directory> your-src-directory/configure
                              --prefix=your-install-directory
   ```

   This will analyze your system and prepare the compilation of `WHIZARD` in the build directory. Make sure to set the proper additional options to `configure`, see Sec. 2.3.3 below.

4. Call `make` to compile and link `WHIZARD`:

   ```
   your-build-directory> make
   ```

5. If you want to make sure that everything works, run

   ```
   your-build-directory> make check
   ```

   This will take some more time.

6. Install:

   ```
   your-build-directory> make install
   ```

`WHIZARD` should now be installed in the installation directory of your choice. If the installation is not in your standard search paths, you have to account for this by extending the paths appropriately, see Sec. 3.3.1.

2.3.3 Configure Options

The configure script accepts environment variables and flags. They can be given as arguments to the `configure` program in arbitrary order. You may run `configure --help` for a listing; only the last part of this long listing is specific for the `WHIZARD` system. Here is an example:

```
configure  FC=gfortran-5.4  FCFLAGS="-g -O3"  --enable-fc-openmp
```

The most important options are

- `FC` (variable): The `Fortran` compiler. This is necessary if you need a compiler different from the standard compiler on the system, e.g., if the latter is too old.

- **FCFLAGS** (variable): The flags to be given to the **Fortran** compiler. The main use is to control the level of optimization.

- **--prefix=⟨*directory-name*⟩**: Specify a non-default directory for installation.

- **--enable-fc-openmp**: Enable parallel executing via OpenMP on a multi-processor/multi-core machine. This works only if OpenMP is supported by the compiler (e.g., **gfortran**). When running **WHIZARD**, the number of processors that are actually requested can be controlled by the user. Without this option, **WHIZARD** will run in serial mode on a single core. See Sec. 5.4.3 for further details.

- **--enable-fc-mpi**: Enable parallel executing via MPI on a single machine using several cores or several machines. This works only if a MPI library is installed (e.g. **OpenMPI**) and **FC=mpifort CC=mpicc CXX=mpic++** is set. Without this option, **WHIZARD** will run in serial mode on a single core. The flag can be combined with **--enable-fc-openmp**. See Sec. 3.3.2 for further details.

- **LHADPF_DIR** (variable): The location of the optional **LHAPDF** package, if non-default.

- **LOOPTOOLS_DIR** (variable): The location of the optional **LOOPTOOLS** package, if non-default.

- **OPENLOOPS_DIR** (variable): The location of the optional **OpenLoops** package, if non-default.

- **GOSAM_DIR** (variable): The location of the optional **Gosam** package, if non-default.

- **HOPPET_DIR** (variable): The location of the optional **HOPPET** package, if non-default.

- **HEPMC_DIR** (variable): The location of the optional **HepMC** package, if non-default.

- **LCIO/LCIO_DIR** (variable): The location of the optional **LCIO** package, if non-default.

Other flags that might help to work around possible problems are the flags for the C and **C++** compilers as well as the **Fortran77** compiler, or the linker flags and additional libraries for the linking process.

- **CC** (variable): **C** compiler command

- **F77** (variable): **Fortran77** compiler command

- **CXX** (variable): **C++** compiler command

- **CPP** (variable): **C** preprocessor

- **CXXCPP** (variable): **C++** preprocessor

- **CFLAGS** (variable): **C** compiler flags

- **FFLAGS** (variable): **Fortran77** compiler flags

- CXXFLAGS (variable): C++ compiler flags

- LIBS (variable): libraries to be passed to the linker as -l*library*

- LDFLAGS (variable): non-standard linker flags

For other options (like e.g. --with-precision=... etc.) please see the configure --help option.

2.3.4 Details on the Configure Process

The configure process checks for the build and host system type; only if this is not detected automatically, the user would have to specify this by himself. After that system-dependent files are searched for, LaTeX and Acroread for documentation and plots, the Fortran compiler is checked, and finally the OCaml compiler. The next step is the checks for external programs like LHAPDF and HepMC. Finally, all the Makefiles are being built.

The compilation is done by invoking make and finally make install. You could also do a make check in order to test whether the compilation has produced sane files on your system. This is highly recommended.

Be aware that there be problems for the installation if the install path or a user's home directory is part of an AFS file system. Several times problems were encountered connected with conflicts with permissions inside the OS permission environment variables and the AFS permission flags which triggered errors during the make install procedure. Also please avoid using make -j options of parallel execution of Makefile directives as AFS filesystems might not be fast enough to cope with this.

For specific problems that might have been encountered in rare circumstances for some FORTRAN compilers confer the webpage https://whizard.hepforge.org/compilers.html.

Note that the PYTHIA bundle for showering and hadronization (and some other external legacy code pieces) do still contain good old Fortran77 code. These parts should better be compiled with the very same Fortran2003 compiler as the WHIZARD core. There is, however, one subtlety: when the configure flag FC gets a full system path as argument, libtool is not able to recognize this as a valid (GNU) Fortran77 compiler. It then searches automatically for binaries like f77, g77 etc. or a standard system compiler. This might result in a compilation failure of the Fortran77 code. A viable solution is to define an executable link and use this (not the full path!) as FC flag.

It is possible to compile WHIZARD without the OCaml parts of O'Mega, namely by using the --disable-omega option of the configure. This will result in a built of WHIZARD with the O'Mega Fortran library, but without the binaries for the matrix element generation. All selftests (cf. 2.3.7) requiring O'Mega matrix elements are thereby switched off. Note that you can install such a built (e.g. on a batch system without OCaml installation), but the try to build a distribution (all make distxxx targets) will fail.

2.3.5   Building on Darwin/macOS

The easiest way to build `WHIZARD` on Darwin/macOS is to install the complete GNU compiler suite (`gcc/g++/gfortran`). This can be done with one of the code repositories like `MacPorts`, `HomeBrew` or `Fink`. In order to include `ROOT` which natively should be built using the intrinsic `clang/clang++` for the graphics support, there is also the possibility to build external tools like `HepMC3`, `PYTHIA8`, `FastJet`, and `LCIO` with `clang++`, and set in the configure option for `WHIZARD` `C` and `C++` compiler accordingly:

> ../configure CC=clang CXX=clang++ [...]

Note that `FastJet` might need to be configured with the `--disable-auto-ptr` option when compiling with `clang++` and strict `C++17` standard.

Since Darwin v10.11, the security measures of the new Darwin systems do not allow e.g. environment variables passed to subprocesses. This does not change anything for the installed WHIZARD, but the testsuite (make check) will not work before make install has been executed. make distcheck will not work on El Capitan. There is also the option to disable the System Integrity Protocol (SIP) of modern OSX by booting in Recovery Mode, open a terminal and type `csrutil disable`. However, we do not recommend to do so.

2.3.6   Building on Windows

For Windows, from `Windows 10` onwards, there is the possibility to install and use an underlying Linux operating system, e.g. `Ubuntu`. Installation and usage of `WHIZARD` works then the same way as described above.

2.3.7   `WHIZARD` self tests/checks

`WHIZARD` has a number of self-consistency checks and tests which assure that most of its features are running in the intended way. The standard procedure to invoke these self tests is to perform a `make check` from the `build` directory. If `src` and `build` directories are the same, all relevant files for these self-tests reside in the `tests` subdirectory of the main `WHIZARD` directory. In that case, one could in principle just call the scripts individually from the command line. Note, that if `src` and `build` directory are different as recommended, then the input files will have been installed in `prefix/share/whizard/test`, while the corresponding test shell scripts remain in the `srcdir/test` directory. As the main shell script `run_whizard.sh` has been built in the `build` directory, one now has to copy the files over by and set the correct paths by hand, if one wishes to run the test scripts individually. `make check` still correctly performs all `WHIZARD` self-consistency tests. The tests itself fall into two categories, unit self test that individually test the modular structure of `WHIZARD`, and tests that are run by `SINDARIN` files. In future releases of `WHIZARD`, these two categories of tests will be better separated than in the 2.2.1 release.

There are additional, quite extensiv numerical tests for validation and backwards compatibility checks for SM and MSSM processes. As a standard, these extended self tests are not invoked. However, they can be enabled by executing the corresponding specific `make check` operations in the subdirectories for these extensive tests.

As the new `WHIZARD` testsuite does very thorough and scrupulous tests of the whole `WHIZARD` structure, it is always possible that some tests are failing due to some weird circumstances or because of numerical fluctuations. In such a case do not panic, contact the developers (`whizard@desy.de`) and provide them with the logfiles of the failing test as well as the setup of your configuration.

Chapter 3

Working with `WHIZARD`

`WHIZARD` can run as a stand-alone program. You (the user) can steer `WHIZARD` either interactively or by a script file. We will first describe the latter method, since it will be the most common way to interact with the `WHIZARD` system.

3.1   Hello World

The legacy version series 1 of the program relied on a bunch of input files that the user had to provide in some obfuscated format. This approach is sufficient for straightforward applications. However, once you get experienced with a program, you start thinking about uses that the program's authors did not foresee. In case of a Monte Carlo package, typical abuses are parameter scans, complex patterns of cuts and reweighting factors, or data analysis without recourse to external packages. This requires more flexibility.

Instead of transferring control over data input to some generic scripting language like `PERL` or `Python` (or even `C++`), which come with their own peculiarities and learning curves, we decided to unify data input and scripting in a dedicated steering language that is particularly adapted to the needs of Monte-Carlo integration, simulation, and simple analysis of the results. Thus we discovered what everybody knew anyway: that W(h)izards communicate in `SINDARIN`, Scripting INtegration, Data Analysis, Results display and INterfaces.

`SINDARIN` is a DSL – a domain-specific scripting language – that is designed for the single purpose of steering and talking to `WHIZARD`. Now since `SINDARIN` is a programming language, we honor the old tradition of starting with the famous Hello World program. In `SINDARIN` this reads simply

```
printf "Hello World!"
```

Open your favorite editor, type this text, and save it into a file named `hello.sin`.

Now we assume that you – or your kind system administrator – has installed `WHIZARD` in your executable path. Then you should open a command shell and execute (we will come to the meaning of the `-r` option later.)

```
/home/user$ whizard -r hello.sin
```

```
| Writing log to 'whizard.log'
|===============================================================================|
|                                                                               |
|    WW            WW  WW   WW  WW  WWWWWW      WW       WWWWW    WWWW            |
|     WW    WW     WW  WW   WW  WW    WW        WWWW     WW WW   WW WW           |
|     WW  WW WW  WW    WWWWWWW  WW    WW       WW WW     WWWWW    WW  WW         |
|      WWWW    WWWW    WW  WW   WW    WW      WWWWWWWW   WW WW    WW  WW         |
|       WW      WW     WW  WW   WW  WWWWWW  WW      WW  WW    WW  WWWW           |
|                                                                               |
|                                                                               |
|                                       W                                       |
|                                      sW                                       |
|                                      WW                                       |
|                                     sWW                                       |
|                                     WWW                                       |
|                                    wWWW                                       |
|                                   wWWWW                                       |
|                                   WW WW                                       |
|                                   WW WW                                       |
|                                  wWW WW                                       |
|                                 wWW  WW                                       |
|                                 WW   WW                                       |
|                                 WW   WW                                       |
|                                 WW   WW                                       |
|                                 WW   WW                                       |
|                                 WW   WW                                       |
|                                 WW   WW                                       |
|                wwwwww           WW   WW                                       |
|              WWWWWww            WW   WW                                       |
|             WWWWWwwwww   WW     WW                                           |
|              wWWWwwwwwWW        WW                                           |
|             wWWWWWWWWWWwWWW      WW                                          |
|              wWWWWW      wW    WWWWWWW                                       |
|              WWWW      wW     WW  wWWWWWWWWwww                               |
|               WWWW            wWWWWWWWWwwww                                  |
|                WWWW               WWWW    WWw                               |
|                 WWWWww             WWWW                                     |
|                  WWWwwww            WWWW                                    |
|                   wWWWWwww       wWWWWW                                     |
|                      Wwwwwwwww WWW                                          |
|                                                                               |
|                                                                               |
|                                                                               |
|  by:    Wolfgang Kilian, Thorsten Ohl, Juergen Reuter                        |
|         with contributions from Christian Speckner                           |
|         Contact: <whizard@desy.de>                                           |
|                                                                               |
|  if you use WHIZARD please cite:                                             |
|        W. Kilian, T. Ohl, J. Reuter,  Eur.Phys.J.C71 (2011) 1742             |
|                                       [arXiv: 0708.4233 [hep-ph]]            |
|        M. Moretti, T. Ohl, J. Reuter, arXiv: hep-ph/0102195                  |
|                                                                               |
|===============================================================================|
|                              WHIZARD 3.0.3                                    |
|===============================================================================|
| Reading model file '/usr/local/share/whizard/models/SM.mdl'
| Preloaded model: SM
| Process library 'default_lib': initialized
| Preloaded library: default_lib
| Reading commands from file 'hello.sin'
Hello World!
| WHIZARD run finished.
|===============================================================================|
```

Figure 3.1: *Output of the* `"Hello world!"` *SINDARIN script.*

and if everything works well, you get the output (the complete output including the WHIZARD
banner is shown in Fig. 3.1)

```
| Writing log to 'whizard.log'
```

[... here a banner is displayed]

```
|==============================================================================|
|                                 WHIZARD 3.0.3                                 |
|==============================================================================|
| Reading model file '/usr/local/share/whizard/models/SM.mdl'
| Preloaded model: SM
! Process library 'default_lib': initialized
! Preloaded library: default_lib
| Reading commands from file 'hello.sin'
Hello World!
| WHIZARD run finished.
|==============================================================================|
```

If this has just worked for you, you can be confident that you have a working WHIZARD installation,
and you have been able to successfully run the program.

3.2   A Simple Calculation

You may object that WHIZARD is not exactly designed for printing out plain text. So let us
demonstrate a more useful example.

Looking at the Hello World output, we first observe that the program writes a log file named
(by default) whizard.log. This file receives all screen output, except for the output of external
programs that are called by WHIZARD. You don't have to cache WHIZARD's screen output yourself.

After the welcome banner, WHIZARD tells you that it reads a physics *model*, and that it
initializes and preloads a *process library*. The process library is initially empty. It is ready for
receiving definitions of elementary high-energy physics processes (scattering or decay) that you
provide. The processes are set in the context of a definite model of high-energy physics. By
default this is the Standard Model, dubbed SM.

Here is the SINDARIN code for defining a SM physics process, computing its cross section,
and generating a simulated event sample in Les Houches event format:

```
process ee = e1, E1 => e2, E2
sqrts = 360 GeV
n_events = 10
sample_format = lhef
simulate (ee)
```

As before, you save this text in a file (named, e.g., ee.sin) which is run by

```
/home/user$ whizard -r ee.sin
```

(We will come to the meaning of the -r option later.) This produces a lot of output which looks similar to this:

```
| Writing log to 'whizard.log'
[... banner ...]
 |=============================================================================|
 |                              WHIZARD 3.0.3
 |=============================================================================|
 | Reading model file '/usr/local/share/whizard/models/SM.mdl'
 | Preloaded model: SM
 | Process library 'default_lib': initialized
 | Preloaded library: default_lib
 | Reading commands from file 'ee.sin'
 | Process library 'default_lib': recorded process 'ee'
 sqrts =  3.600000000000E+02
 n_events = 10


 | Starting simulation for process 'ee'
 | Simulate: process 'ee' needs integration
 | Integrate: current process library needs compilation
 | Process library 'default_lib': compiling ...
 | Process library 'default_lib': writing makefile
 | Process library 'default_lib': removing old files
 rm -f default_lib.la
 rm -f default_lib.lo default_lib_driver.mod opr_ee_i1.mod ee_i1.lo
 rm -f ee_i1.f90
 | Process library 'default_lib': writing driver
 | Process library 'default_lib': creating source code
 rm -f ee_i1.f90
 rm -f opr_ee_i1.mod
 rm -f ee_i1.lo
 /usr/local/bin/omega_SM.opt -o ee_i1.f90 -target:whizard
  -target:parameter_module parameters_SM -target:module opr_ee_i1
  -target:md5sum '70DB728462039A6DC1564328E2F3C3A5' -fusion:progress
  -scatter 'e- e+ -> mu- mu+'
 [1/1] e- e+ -> mu- mu+ ... allowed. [time: 0.00 secs, total: 0.00 secs, remaining: 0.00 secs]
 all processes done. [total time: 0.00 secs]
 SUMMARY: 6 fusions, 2 propagators, 2 diagrams
 | Process library 'default_lib': compiling sources
[.....]


 | Process library 'default_lib': loading
 | Process library 'default_lib': ... success.
 | Integrate: compilation done
 | RNG: Initializing TAO random-number generator
 | RNG: Setting seed for random-number generator to 9616
 | Initializing integration for process ee:
 | ------------------------------------------------------------------------
 | Process [scattering]: 'ee'
 |   Library name  = 'default_lib'
```

```
|   Process index = 1
|   Process components:
|     1: 'ee_i1':   e-, e+ => mu-, mu+ [omega]
| ------------------------------------------------------------------------
| Beam structure: [any particles]
| Beam data (collision):
|   e-  (mass = 5.1099700E-04 GeV)
|   e+  (mass = 5.1099700E-04 GeV)
|   sqrts = 3.600000000000E+02 GeV
| Phase space: generating configuration ...
| Phase space: ... success.
| Phase space: writing configuration file 'ee_i1.phs'
| Phase space: 2 channels, 2 dimensions
| Phase space: found 2 channels, collected in 2 groves.
| Phase space: Using 2 equivalences between channels.
| Phase space: wood
Warning: No cuts have been defined.


| Starting integration for process 'ee'
| Integrate: iterations not specified, using default
| Integrate: iterations = 3:1000:"gw", 3:10000:""
| Integrator: 2 chains, 2 channels, 2 dimensions
| Integrator: Using VAMP channel equivalences
| Integrator: 1000 initial calls, 20 bins, stratified = T
| Integrator: VAMP
|=============================================================================|
| It       Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
   1         784  8.3282892E+02  1.68E+00    0.20    0.06*  39.99
   2         784  8.3118961E+02  1.23E+00    0.15    0.04*  76.34
   3         784  8.3278951E+02  1.36E+00    0.16    0.05   54.45
|-----------------------------------------------------------------------------|
   3        2352  8.3211789E+02  8.01E-01    0.10    0.05   54.45   0.50    3
|-----------------------------------------------------------------------------|
   4        9936  8.3331732E+02  1.22E-01    0.01    0.01*  54.51
   5        9936  8.3341072E+02  1.24E-01    0.01    0.01   54.52
   6        9936  8.3331151E+02  1.23E-01    0.01    0.01*  54.51
|-----------------------------------------------------------------------------|
   6       29808  8.3334611E+02  7.10E-02    0.01    0.01   54.51   0.20    3
|=============================================================================|


[.....]
| Simulate: integration done
| Simulate: using integration grids from file 'ee_m1.vg'
| RNG: Initializing TAO random-number generator
| RNG: Setting seed for random-number generator to 9617
| Simulation: requested number of events = 10
|           corr. to luminosity [fb-1] =   1.2000E-02
| Events: writing to LHEF file 'ee.lhe'
| Events: writing to raw file 'ee.evx'
```

```
| Events: generating 10 unweighted, unpolarized events ...
| Events: event normalization mode '1'
|         ... event sample complete.
| Events: closing LHEF file 'ee.lhe'
| Events: closing raw file 'ee.evx'
| There were no errors and    1 warning(s).
| WHIZARD run finished.
|=============================================================================|
```

The final result is the desired event file, ee.lhe.

Let us discuss the output quickly to walk you through the procedures of a WHIZARD run: after the logfile message and the banner, the reading of the physics model and the initialization of a process library, the recorded process with tag 'ee' is recorded. Next, user-defined parameters like the center-of-mass energy and the number of demanded (unweighted) events are displayed. As a next step, WHIZARD is starting the simulation of the process with tag 'ee'. It recognizes that there has not yet been an integration over phase space (done by an optional integrate command, cf. Sec. 5.7.1), and consequently starts the integration. It then acknowledges, that the process code for the process 'ee' needs to be compiled first (done by an optional compile command, cf. Sec. 5.4.5). So, WHIZARD compiles the process library, writes the makefile for its steering, and as a safeguard against garbage removes possibly existing files. Then, the source code for the library and its processes are generated: for the process code, the default method – the matrix element generator O'Mega is called (cf. Sec. 9.3); and the sources are being compiled.

The next steps are the loading of the process library, and WHIZARD reports the completion of the integration. For the Monte-Carlo integration, a random number generator is initialized. Here, it is the default generator, TAO (for more details, cf. Sec. 6.2, while the random seed is set to a value initialized by the system clock, as no seed has been provided in the SINDARIN input file.

Now, the integration for the process 'ee' is initialized, and information about the process (its name, the name of its process library, its index inside the library, and the process components out of which it consists, cf. Sec. 5.4.4) are displayed. Then, the beam structure is shown, which in that case are symmetric partonic electron and positron beams with the center-of-mass energy provided by the user (360 GeV). The next step is the generation of the phase space, for which the default phase space method wood (for more details cf. Sec. 8.3) is selected. The integration is performed, and the result with absolute and relative error, unweighting efficiency, accuracy, χ^2 quality is shown.

The final step is the event generation (cf. Chap. 11). The integration grids are now being used, again the random number generator is initialized. Finally, event generation of ten unweighted events starts (WHIZARD let us know to which integrated luminosity that would correspond), and events are written both in an internal (binary) event format as well as in the demanded LHE format. This concludes the WHIZARD run.

After a more comprehensive introduction into the SINDARIN steering language in the next chapter, Chap. 4, we will discuss all the details of the different steps of this introductory example.

3.3 WHIZARD in a Computing Environment

3.3.1 Working on a Single Computer

After installation, `WHIZARD` is ready for use. There is a slight complication if `WHIZARD` has been installed in a location that is not in your standard search paths.

In that case, to successfully run `WHIZARD`, you may either

- manually add `your-install-directory/bin` to your execution PATH and `your-install-directory/lib` to your library search path (LD_LIBRARY_PATH), or

- whenever you start a project, execute

  ```
  your-workspace> . your-install-directory/bin/whizard-setup.sh
  ```

  which will enable the paths in your current environment, or

- source `whizard-setup.sh` script in your shell startup file.

In either case, try to call `whizard --help` in order to check whether this is done correctly.

For a new `WHIZARD` project, you should set up a new (empty) directory. Depending on the complexity of your task, you may want to set up separate directories for each subproblem that you want to tackle, or even for each separate run. The location of the directories is arbitrary.

To run, `WHIZARD` needs only a single input file, a `SINDARIN` command script with extension `.sin` (by convention). Running `WHIZARD` is as simple as

```
your-workspace> whizard your-input.sin
```

No other configuration files are needed. The total number of auxiliary and output files generated in a single run may get quite large, however, and they may clutter your workspace. This is the reason behind keeping subdirectories on a per-run basis.

Basic usage of `WHIZARD` is explained in Chapter 3, for more details, consult the following chapters. In Sec. 14.1 we give an account of the command-line options that `WHIZARD` accepts.

3.3.2 Working Parallel on Several Computers

For integration (only VAMP2), `WHIZARD` supports parallel execution via MPI by communicating between parallel tasks on a single machine or distributed over several machines.

During integration the calculation of channels is distributed along several workers where a master worker collects the results and adapts weights and grids. In wortwhile cases (e.g. high number of calls in one channel), the calculation of a single grid is additionally distributed. For that, we provide two different parallelization methods, which can be steered by `$vamp_parallel_method`, implementing the dualistic parallelization approach between channels and single grids. The `simple` method provides a locally-fixed assignment approach without the need of intermediate communication between the MPI workers. Whereas the `load` method provides a global queue with a master worker acting as a (communication) governor, therefore,

excluding itself as potential "computing" worker. The governor receives and distributes work requests from all other workers, and, finally, receives their results. The methods differ from each other only in the way how they distribute excessive workers, in the case, where there are more workers than channels. Here, the `load` method implements a balancing condition based on the channel weights in contrast to the simplistic ansatz.

Both methods use a full non-blocking communication approach in order to collect the integration results of each channel after each iteration. After finishing the computation of a channel, the associated slave worker spawns a callback mechansim leading to the initialization of a sending process to the master. The master worker organizes, depending on the parallelization method, the correct closing of the sending process for a given channel by a matching receiving process. The callback approach allows us to concurrently communicate and produce integration results providing an increased parallelization portion, i.e. better HPC performance and utilization.

The `load` method comes with a drawback that it does not work with less than three workers. Hence, we recommend (e.g. for debugging purpose of the parallel setup) to use the `simple` method, and to use the `load` method only for direct production runs.

In order to use these advancements, WHIZARD requires an installed MPI-3.1 capable library (e.g. OpenMPI) and configuration and compilation with the appropriate flags, cf. Sec. 2.3.

MPI support is only active when the integration method is set to VAMP2. Additionally, to preserve the numerical properties of a single task run, it is recommended to use the RNGstream as random number generator.

```
$integration_method = 'vamp2'
$rng_method = 'rng_stream'
$vamp_parallel_method = 'simple' !! or 'load'
```

WHIZARD has then to be called by mpirun

```
your-workspace> mpirun -f hostfile -np 4 --output-filename mpi.log whizard your-input.sin
```

where the number of parallel tasks can be set by `-np` and a hostfile can be given by `--hostfile`. It is recommended to use `--output-filename` which lets mpirun redirect the standard (error) output to a file, for each worker separately.

Notes on Parallelization with MPI

The parallelization of WHIZARD requires that all instances of the parallel run be able to write and read all files produced by WHIZARD in a network file system as the current implementation does not handle parallel I/O. Usually, high-performance clusters have support for at least one network filesystem.

Furthermore, not all functions of WHIZARD are currently supported or are only supported in a limited way in parallel mode. Currently the `?rebuild_<flags>` for the phase space and the matrix element library are not yet available, as well as the calculation of matrix elements with resonance histories.

Some features that have been missing in the very first implementation of the parallelized integration have now been made available, like the support of run IDs and the parallelization of the event generation.

A final remark on the stability of the numerical results in terms of the number of workers involved. Under certain circumstances, results between different numbers of workers but using otherwise an identical `SINDARIN` file can lead to slightly numerically different (but statistically compatible) results for integration or event generation This is related to the execution of the computational operations in MPI, which we use to reduce results from all workers. If the order of the numbers in the arithmetical operations changes, for example, by different setups of the workers, then the numerical results change slightly, which in turn is amplified under the influence of the adaptation. Nevertheless, the results are all statistically consistent.

3.3.3 Stopping and Resuming WHIZARD Jobs

On a Unix-like system, it is possible to prematurely stop running jobs by a `kill(1)` command, or by entering `Ctrl-C` on the terminal.

If the system supports this, `WHIZARD` traps these signals. It also traps some signals that a batch operating system might issue, e.g., for exceeding a predefined execution time limit. `WHIZARD` tries to complete the calculation of the current event and gracefully close open files. Then, the program terminates with a message and a nonzero return code. Usually, this should not take more than a fraction of a second.

If, for any reason, the program does not respond to an interrupt, it is always possible to kill it by `kill -9`. A convenient method, on a terminal, would be to suspend it first by `Ctrl-Z` and then to kill the suspended process.

The program is usually able to recover after being stopped. Simply run the job again from start, with the same input, all output files generated so far left untouched. The results obtained so far will be quickly recovered or gathered from files written in the previous run, and the actual time-consuming calculation is resumed near the point where it was interrupted.[1] If the interruption happened during an integration step, it is resumed after the last complete iteration. If it was during event generation, the previous events are taken from file and event generation is continued.

The same mechanism allows for efficiently redoing a calculation with similar, somewhat modified input. For instance, you might want to add a further observable to event analysis, or write the events in a different format. The time for rerunning the program is determined just by the time it takes to read the existing integration or event files, and the additional calculation is done on the recovered information.

By managing various checksums on its input and output files, `WHIZARD` detects changes that affect further calculations, so it does a real recalculation only where it is actually needed. This applies to all steps that are potentially time-consuming: matrix-element code generation, compilation, phase-space setup, integration, and event generation. If desired, you can set command-line options or `SINDARIN` parameters that explicitly discard previously generated information.

---

[1]This holds for simple workflow. In case of scans and repeated integrations of the same process, there may be name clashes on the written files which prevent resuming. A future `WHIZARD` version will address this problem.

3.3.4   Files and Directories: default and customization

`WHIZARD` jobs take a small set of files as input. In many cases, this is just a single `SINDARIN` script provided by the user. When running, `WHIZARD` can produce a set of auxiliary and output files:

1. **Job.** Files pertaining to the `WHIZARD` job as a whole. This is the default log file `whizard.log`.

2. **Process compilation.** Files that originate from generating and compiling process code. If the default `O'Mega` generator is used, these files include `Fortran` source code as well as compiled libraries that are dynamically linked to the running executable. The file names are derived from either the process-library name or the individual process names, as defined in the `SINDARIN` input. The default library name is `default_lib`.

3. **Integration.** Files that are created by integration, i.e., when calculating the total cross section for a scattering process using the Monte-Carlo algorithm. The file names are derived from the process name.

4. **Simulation.** Files that are created during simulation, i.e., generating event samples for a process or a set of processes. By default, the file names are derived from the name of the first process. Event-file formats are distinguished by appropriate file name extensions.

5. **Result Analysis.** Files that are created by the internal analysis tools and written by the command `write_analysis` (or `compile_analysis`). The default base name is `whizard_analysis`.

A complex workflow with several processes, parameter sets, or runs, can easily lead to in file-name clashes or a messy working directory. Furthermore, running a batch job on a dedicated computing environment often requires transferring data from a user directory to the server and back.

Custom directory and file names can be used to organize things and facilitate dealing with the environment, along with the available batch-system tools for coordinating file transfer.

1. **Job.**

   - The `-L` option on the command line defines a custom base name for the log file.
   - The `-J` option on the command line defines a job ID. For instance, this may be set to the job ID assigned by the batch system. Within the `SINDARIN` script, the job ID is available as the string variable `$job_id` and can be used for constructing custom job-specific file and directory names, as described below.

2. **Process compilation.**

   - The user can require the program to put all files created during the compilation step including the library to be linked, in a subdirectory of the working directory. To enable this, set the string variable `$compile_workspace` within the `SINDARIN` script.

3. **Integration.**

   - The value of the string variable `$run_id`, if set, is appended to the base name of all files created by integration, separated by dots. If the `SINDARIN` script scans over parameters, varying the run ID avoids repeatedly overwriting files with identical name during the scan.

   - The user can require the program to put the important files created during the integration step – the phase-space configuration file and the `VAMP` grid files – in a subdirectory of the working directory. To enable this, set the string variable `$integrate_workspace` within the `SINDARIN` script. (`$compile_workspace` and `$integrate_workspace` may be set to the same value.)

   Log files produced during the integration step are put in the working directory.

4. **Simulation.**

   - The value of the string variable `$run_id`, if set, identifies the specific integration run that is used for the event sample. It is also inserted into default event-sample file names.

   - The variable `$sample`, if set, defines an arbitrary base name for the files related to the event sample.

   Files resulting from simulation are put in the working directory.

5. **Result Analysis.**

   - The variable `$out_file`, if set, defines an arbitrary base name for the analysis data and auxiliary files.

   Files resulting from result analysis are put in the working directory.

3.3.5 Batch jobs on a different machine

It is possible to separate the tasks of process-code compilation, integration, and simulation, and execute them on different machines. To make use of this feature, the local and remote machines including all installed libraries that are relevant for `WHIZARD`, must be binary-compatible.

1. Process-code compilation may be done once on a local machine, while the time-consuming tasks of integration and event generation for specific parameter sets are delegated to a remote machine, e.g., a batch cluster. To enable this, prepare a `SINDARIN` script that just produces process code (i.e., terminates with a `compile` command) for the local machine. You may define `$compile_workspace` such that all generated code conveniently ends up in a single subdirectory.

   To start the batch job, transfer the workspace subdirectory to the remote machine and start `WHIZARD` there. The `SINDARIN` script on the remote machine must include the local script

unchanged in all parts that are relevant for process definition. The program will recognize the contents of the workspace, skip compilation and instead link the process library immediately. To proceed further, the script should define the run-specific parameters and contain the appropriate commands for integration and simulation.

2. Analogously, you may execute both process-code compilation and integration locally, but generate event samples on a remote machine. To this end, prepare a `SINDARIN` script that produces process code and computes integrals (i.e., terminates with an `integrate` command) for the local machine. You may define `$compile_workspace` and `$integrate_workspace` (which may coincide) such that all generated code, phase-space and integration grid data conveniently end up in subdirectories.

   To start the batch job, transfer the workspace(s) to the remote machine and start `WHIZARD` there. The `SINDARIN` script on the remote machine must include the local script unchanged in all parts that are relevant for process definition and integration. The program will recognize the contents of the workspace, skip compilation and integration and instead load the process library and integration results immediately. To proceed further, the script should define the sample-specific parameters and contain the appropriate commands for simulation.

To simplify transferring whole directories, `WHIZARD` supports the `--pack` and `--unpack` options. You may specify any number of these options for a `WHIZARD` run. (The feature relies on the GNU version of the `tar` utility.)

For instance,

```
whizard script1.sin --pack my_ws
```

runs `WHIZARD` with the `SINDARIN` script `script1.sin` as input, where within the script you have defined

```
$compile_workspace = "my_ws"
```

as the target directory for process-compilation files. After completion, the program will tar and gzip the target directory as `my_ws.tgz`. You should copy this file to the remote machine as one of the job's input files.

On the remote machine, you can then run the program with

```
whizard script2.sin --unpack my_ws.tgz
```

where `script2.sin` should include `script1.sin`, and add integration or simulation commands. The contents of `ws.tgz` will thus be unpacked and reused on the remote machine, instead of generating new process code.

3.3.6   Static Linkage

In its default running mode, `WHIZARD` compiles process-specific matrix element code on the fly and dynamically links the resulting library. On the computing server, this requires availability of the appropriate `Fortran` compiler, as well as the `OCaml` compiler suite, and the dynamical linking feature.

Since this may be unavailable or undesired, there is a possibility to distribute `WHIZARD` as a statically linked executable that contains a pre-compiled library of processes. This removes the need for the `Fortran` compiler, the `OCaml` system, and extra dynamic linking. Any external libraries that are accessed (the `Fortran` runtime environment, and possibly some dynamically linked external libraries and/or the `C++` runtime library, must still be available on the target system, binary-compatible. Otherwise, there is no need for transferring the complete `WHIZARD` installation or process-code compilation data.

Generating, compiling and linking matrix element code is done in advance on a machine that can access the required tools and produces compatible libraries. This procedure is accomplished by `SINDARIN` commands, explained below in Sec. 5.4.7.

3.4   Troubleshooting

In this section, we list known issues or problems and give advice on what can be done in case
something does not work as intended.

3.4.1   Possible (uncommon) build problems

OCaml versions and O'Mega builds

For the matrix element generator O'Mega of WHIZARD the functional programming language
OCaml is used. Unfortunately, the versions of the OCaml compiler from 3.12.0 on broke backwards
compatibility. Therefore, versions of O'Mega/WHIZARD up to v2.0.2 only compile with older
versions (3.04 to 3.11 works). This has been fixed in all WHIZARD versions from 2.0.3 on.

Identical Build and Source directories

There is a problem that only occurred with version 2.0.0 and has been corected for all follow-up
versions. It can only appear if you compile the WHIZARD sources in the source directory. Then
an error like this may occur:

```
...
libtool: compile:  gfortran -I../misc -I../vamp -g -O2 -c processes.f90 -fPIC -o
          .libs/processes.o
libtool: compile:  gfortran -I../misc -I../vamp -g -O2 -c processes.f90 -o
          processes.o >/dev/null 2>&1
make[2]: *** No rule to make target 'limits.lo', needed by 'decays.lo'.  Stop.
...
make: *** [all-recursive] Error 1
```

In this case, please unpack a fresh copy of WHIZARD and configure it in a separate directory (not
necessarily a subdirectory). Then the compilation will go through:

```
$ zcat whizard-3.0.3.tar.gz | tar xf -
$ cd whizard-3.0.3
$ mkdir _build
$ cd _build
$ ../configure FC=gfortran
$ make
```

The developers use this setup to be able to test different compilers. Therefore building in the
same directory is not as thoroughly tested. This behavior has been patched from version 2.0.1
on. But note that in general it is always adviced to keep build and source directory apart from
each other.

3.4.2   What happens if WHIZARD throws an error?

Particle name special characters in process declarations

Trying to use a process declaration like

```
process foo = e-, e+ => mu-, mu+
```

will lead to a SINDARIN syntax error:

```
process foo = e-, e+ => mu-, mu+
              ^^
| Expected syntax: SEQUENCE    <cmd_process> = process <process_id> '=' <process_p
| Found token: KEYWORD:    '-'
********************************************************************************
********************************************************************************
*** FATAL ERROR:  Syntax error (at or before the location indicated above)
********************************************************************************
********************************************************************************
```

WHIZARD tries to interpret the minus and plus signs as operators (KEYWORD: '-'), so you have to quote the particle names: `process foo = "e-", "e+" => "mu-", "mu+"`.

Missing collider energy

This happens if you forgot to set the collider energy in the integration of a scattering process:

```
********************************************************************************
********************************************************************************
*** FATAL ERROR:  Colliding beams: sqrts is zero (please set sqrts)
********************************************************************************
********************************************************************************
```

This will solve your problem:

```
sqrts = <your_energy>
```

Missing process declaration

If you try to integrate or simulate a process that has not declared before (and is also not available in a library that might be loaded), WHIZARD will complain:

```
********************************************************************************
********************************************************************************
*** FATAL ERROR: Process library doesn't contain process 'f00'
********************************************************************************
********************************************************************************
```

Note that this could sometimes be a simple typo, e.g. in that case an `integrate (f00)` instead of `integrate (foo)`

Ambiguous initial state without beam declaration

When the user declares a process with a flavor sum in the initial state, e.g.

```
process qqaa = u:d, U:D => A, A
sqrts = <your_energy>
integrate (qqaa)
```

then a fatal error will be issued:

```
*******************************************************************************
*******************************************************************************
*** FATAL ERROR: Setting up process 'qqaa':
***              -------------------------------------------
***              Inconsistent initial state. This happens if either
***              several processes with non-matching initial states
***              have been added, or for a single process with an
***              initial state flavor sum. In that case, please set beams
***              explicitly [singling out a flavor / structure function.]
*******************************************************************************
*******************************************************************************
```

What now? Either a structure function providing a tensor structure in flavors has to be provided like

```
beams = p, pbar => pdf_builtin
```

or, if the partonic process was intended, a specific flavor has to be singled out,

```
beams = u, U
```

which would take only the up-quarks. Note that a sum over process components with varying initial states is not possible.

Invalid or unsupported beam structure

An error message like

```
*******************************************************************************
*******************************************************************************
*** FATAL ERROR: Beam structure: [.......] not supported
*******************************************************************************
*******************************************************************************
```

This happens if you try to use a beam structure with is either not supported by WHIZARD (meaning that there is no phase-space parameterization for Monte-Carlo integration available in order to allow an efficient sampling), or you have chosen a combination of beam structure functions that do not make sense physically. Here is an example for the latter (lepton collider ISR applied to protons, then proton PDFs):

```
beams = p, p => isr => pdf_builtin
```

Mismatch in beams

Sometimes you get a rather long error output statement followed by a fatal error:

```
 Evaluator product
 First interaction
 Interaction: 6
 Virtual:
 Particle 1
   [momentum undefined]
[.......]
 State matrix:  norm =  1.000000000000E+00
 [f(2212)]
   [f(11)]
     [f(92) c(1 )]
       [f(-6) c(-1 )] => ME(1) = ( 0.000000000000E+00, 0.000000000000E+00)
[.......]
 *********************************************************************************
 *********************************************************************************
 *** FATAL ERROR: Product of density matrices is empty
 ***              ------------------------------------------
 ***              This happens when two density matrices are convoluted
 ***              but the processes they belong to (e.g., production
 ***              and decay) do not match. This could happen if the
 ***              beam specification does not match the hard
 ***              process. Or it may indicate a WHIZARD bug.
 *********************************************************************************
 *********************************************************************************
```

As `WHIZARD` indicates, this could have happened because the hard process setup did not match the specification of the beams as in:

```
process neutral_current_DIS = e1, u => e1, u
beams_momentum = 27.5 GeV, 920 GeV
beams = p, e => pdf_builtin, none
integrate (neutral_current_DIS)
```

In that case, the order of the beam particles simply was wrong, exchange proton and electron (together with the structure functions) into `beams = e, p => none, pdf_builtin`, and `WHIZARD` will be happy.

Unstable heavy beam particles

If you try to use unstable particles as beams that can potentially decay into the final state particles, you might encounter the following error message:

```
 *********************************************************************************
 *********************************************************************************
 *** FATAL ERROR:  Phase space: Initial beam particle can decay
 *********************************************************************************
 *********************************************************************************
```

This happens basically only for processes in testing/validation (like $t\bar{t} \to b\bar{b}$). In principle, it could also happen in a real physics setup, e.g. when simulating electron pairs at a muon collider:

```
process mmee = "mu-", "mu+" => "e-", "e+"
```

However, WHIZARD at the moment does not allow a muon width, and so WHIZARD is not able to decay a muon in a scattering process. A possibile decay of the beam particle into (part of) the final state might lead to instabilities in the phase space setup. Hence, WHIZARD do not let you perform such an integration right away. When you nevertheless encounter such a rare occasion in your setup, there is a possibility to convert this fatal error into a simple warning by setting the flag:

```
?fatal_beam_decay = false
```

Impossible beam polarization

If you specify a beam polarization that cannot correspond to any physically allowed spin density matrix, e.g.,

```
beams = e1, E1
beams_pol_density = @(-1), @(1:1:.5, -1, 1:-1)
```

WHIZARD will throw a fatal error like this:

```
 Trace of matrix square =    1.4444444444444444
 Polarization: spin density matrix
   spin type    = 2
   multiplicity = 2
   massive      = F
   chirality    = 0
   pol.degree   = 1.0000000
   pure state   = F
   @(+1: +1: ( 3.333333333333E-01, 0.000000000000E+00))
   @(-1: -1: ( 6.666666666667E-01, 0.000000000000E+00))
   @(-1: +1: ( 6.666666666667E-01, 0.000000000000E+00))
********************************************************************************
********************************************************************************
*** FATAL ERROR: Spin density matrix: not permissible as density matrix
********************************************************************************
********************************************************************************
```

Beams with crossing angle

Specifying a crossing angle (e.g. at a linear lepton collider) without explicitly setting the beam momenta,

```
sqrts = 1 TeV
beams = e1, E1
beams_theta = 0, 10 degree
```

triggers a fatal:

```
********************************************************************************
********************************************************************************
*** FATAL ERROR: Beam structure: angle theta/phi specified but momentum/a p undefined
********************************************************************************
********************************************************************************
```

In that case the single beam momenta have to be explicitly set:

```
beams = e1, E1
beams\_momentum = 500 GeV, 500 GeV
beams\_theta = 0, 10 degree
```

Phase-space generation failed

Sometimes an error might be issued that WHIZARD could not generate a valid phase-space parameterization:

```
| Phase space: ... failed.  Increasing phs_off_shell ...
| Phase space: ... failed.  Increasing phs_off_shell ...
| Phase space: ... failed.  Increasing phs_off_shell ...
| Phase space: ... failed.  Increasing phs_off_shell ...
********************************************************************************
********************************************************************************
*** FATAL ERROR: Phase-space: generation failed
********************************************************************************
********************************************************************************
```

You see that WHIZARD tried to increase the number of off-shell lines that are taken into account for the phase-space setup. The second most important parameter for the phase-space setup, **phs_t_channel**, however, is not increased automatically. Its default value is 6, so e.g. for the process $e^+e^- \to 8\gamma$ you will run into the problem above. Setting

```
phs_off_shell = <n>-1
```

where `<n>` is the number of final-state particles will solve the problem.

Non-converging process integration

There could be several reasons for this to happen. The most prominent one is that no cuts have been specified for the process (WHIZARD2 does not apply default cuts), and there are singular regions in the phase space over which the integration stumbles. If cuts have been specified, it could be that they are not sufficient. E.g. in $pp \to jj$ a distance cut between the two jets prevents singular collinear splitting in their generation, but if no p_T cut have been set, there is still singular collinear splitting from the beams.

Why is there no event file?

If no event file has been generated, WHIZARD stumled over some error and should have told you, or, you simply forgot to set a `simulate` command for your process. In case there was a `simulate` command but the process under consideration is not possible (e.g. a typo, `e1, E1 => e2, E3` instead of `e1, E1 => e3, E3`), then you get an error like that:

```
******************************************************************************
*** ERROR: Simulate: no process has a valid matrix element.
******************************************************************************
```

Why is the event file empty?

In order to get events, you need to set either a desired number of events:

```
n_events = <integer>
```

or you have to specify a certain integrated luminosity (the default unit being inverse femtobarn:

```
luminosity = <real> / 1 fbarn
```

In case you set both, WHIZARD will take the one that leads to the higher number of events.

Parton showering fails

For BSM models containing massive stable or long-lived particles parton showering with PYTHIA6 fails:

```
     Advisory warning type 3 given after       0 PYEXEC calls:
     (PYRESD:) Failed to decay particle  1000022 with mass   15.000
******************************************************************************
******************************************************************************
*** FATAL ERROR: Simulation: failed to generate valid event after 10000 tries
******************************************************************************
******************************************************************************
```

The solution to that problem is discussed in Sec. 10.7.3.

3.4.3   Debugging, testing, and validation

Catching/tracking arithmetic exceptions

Catching arithmetic exceptions is not automatically supported by Fortran compilers. In general, flags that cause the compiler to keep track of arithmetic exceptions are diminishing the maximally possible performance, and hence they should not be used in production runs. Hence, we refrained from making these flags a default. They can be added using the `FCFLAGS = <flags>` settings during configuration. For the NAG Fortran compiler we use the flags `-C=all -nan -gline` for debugging purposes. For the gfortran compilers, the flags `-ffpe-trap=invalid,zero,overflow`

are the corresponding debugging flags. For tests, debugging or first sanity checks on your setup, you might want to make use of these flags in order to track possible numerical exceptions in the produced code. Some compilers started to include `IEEE` exception handling support (`Fortran 2008` status), but we do not use these implementations in the `WHIZARD` code (yet).

Chapter 4

Steering WHIZARD: `SINDARIN` Overview

4.1 The command language for WHIZARD

A conventional physics application program gets its data from a set of input files. Alternatively, it is called as a library, so the user has to write his own code to interface it, or it combines these two approaches. `WHIZARD` 1 was built in this way: there were some input files which were written by the user, and it could be called both stand-alone or as an external library.

`WHIZARD` 2 is also a stand-alone program. It comes with its own full-fledged script language, called `SINDARIN`. All interaction between the user and the program is done in `SINDARIN` expressions, commands, and scripts. Two main reasons led us to this choice:

- In any nontrivial physics study, cuts and (parton- or hadron-level) analysis are of central importance. The task of specifying appropriate kinematics and particle selection for a given process is well defined, but it is impossible to cover all possiblities in a simple format like the cut files of `WHIZARD` 1.

  The usual way of dealing with this problem is to write analysis driver code (often in `C++`, using external libraries for Lorentz algebra etc. However, the overhead of writing correct `C++` or `Fortran` greatly blows up problems that could be formulated in a few lines of text.

- While many problems lead to a repetitive workflow (process definition, integration, simulation), there are more involved tasks that involve parameter scans, comparisons of different processes, conditional execution, or writing output in widely different formats. This is easily done by a steering script, which should be formulated in a complete language.

The `SINDARIN` language is built specifically around event analysis, suitably extended to support steering, including data types, loops, conditionals, and I/O.

It would have been possible to use an established general-purpose language for these tasks. For instance, `OCaml` which is a functional language would be a suitable candidate, and the matrix-element generator `O'Mega` is written in that language. Another candidate would be a popular scripting language such as PYTHON.

We started to support interfaces for commonly used languages: prime examples for `C`, `C++`, and PYTHON are found in the `share/interfaces` subdirectory. However, introducing a special-purpose language has the three distinct advantages: First, it is compiled and executed by the very `Fortran` code that handles data and thus accesses it without interfaces. Second, it can be designed with a syntax especially suited to the task of event handling and Monte-Carlo steering, and third, the user is not forced to learn all those features of a generic language that are of no relevance to the application he/she is interested in.

4.2   SINDARIN scripts

A `SINDARIN` script tells the `WHIZARD` program what it has to do. Typically, the script is contained in a file which you (the user) create. The file name is arbitrary; by convention, it has the extension '`.sin`'. `WHIZARD` takes the file name as its argument on the command line and executes the contained script:

```
/home/user$ whizard script.sin
```

Alternatively, you can call `WHIZARD` interactively and execute statements line by line; we describe this below in Sec.14.2.

A `SINDARIN` script is a sequence of *statements*, similar to the statements in any imperative language such as `Fortran` or `C`. Examples of statements are commands like `integrate`, variable declarations like `logical ?flag` or assigments like `mH = 130 GeV`.

The script is free-form, i.e., indentation, extra whitespace and newlines are syntactically insignificant. In contrast to most languages, there is no statement separator. Statements simply follow each other, just separated by whitespace.

```
statement1 statement2
statement3
              statement4
```

Nevertheless, for clarity we recommend to write one statement per line where possible, and to use proper indentation for longer statements, nested and bracketed expressions.

A command may consist of a *keyword*, a list of *arguments* in parantheses (. . .), and an *option* script which itself is a sequence of statements.

```
command
command_with_args (arg1, arg2)
command_with_option { option }
command_with_options (arg) {
  option_statement1
  option_statement2
}
```

As a rule, parentheses `()` enclose arguments and expressions, as you would expect. Arguments enclosed in square brackets `[]` also exist. They have a special meaning, they denote subevents (collections of momenta) in event analysis. Braces `{}` enclose blocks of `SINDARIN` code. In particular, the option script associated with a command is a block of code that may contain

local parameter settings, for instance. Braces always indicate a scoping unit, so parameters will be restored their previous values when the execution of that command is completed.

The script can contain comments. Comments are initiated by either a # or a ! character and extend to the end of the current line.

```
statement
# This is a comment
statement  ! This is also a comment
```

4.3   Errors

Before turning to proper SINDARIN syntax, let us consider error messages. SINDARIN distinguishes syntax errors and runtime errors.

Syntax errors are recognized when the script is read and compiled, before any part is executed. Look at this example:

```
process foo = u, ubar => d, dbar
md = 10
integrade (foo)
```

WHIZARD will fail with the error message

```
sqrts = 1 TeV
integrade (foo)
          ^^
| Expected syntax: SEQUENCE    <cmd_num> = <var_name> '=' <expr>
| Found token: KEYWORD:    '('
********************************************************************************
********************************************************************************
*** FATAL ERROR:  Syntax error (at or before the location indicated above)
********************************************************************************
********************************************************************************
WHIZARD run aborted.
```

which tells you that you have misspelled the command integrate, so the compiler tried to interpret it as a variable.

Runtime errors are categorized by their severity. A warning is simply printed:

```
Warning: No cuts have been defined.
```

This indicates a condition that is suspicious, but may actually be intended by the user.

When an error is encountered, it is printed with more emphasis

```
********************************************************************************
*** ERROR: Variable 'md' set without declaration
********************************************************************************
```

and the program tries to continue. However, this usually indicates that there is something wrong. (The d quark is defined massless, so md is not a model parameter.) WHIZARD counts errors and warnings and tells you at the end

```
| There were  1 error(s) and no warnings.
```

just in case you missed the message.

Other errors are considered fatal, and execution stops at this point.

```
*****************************************************************************
*****************************************************************************
*** FATAL ERROR:  Colliding beams: sqrts is zero (please set sqrts)
*****************************************************************************
*****************************************************************************
```

Here, `WHIZARD` was unable to do anything sensible. But at least (in this case) it told the user what to do to resolve the problem.

4.4   Statements

`SINDARIN` statements are executed one by one. For an overview, we list the most common statements in the order in which they typically appear in a `SINDARIN` script, and quote the basic syntax and simple examples. This should give an impression on the `WHIZARD`'s capabilities and on the user interface. The list is not complete. Note that there are no mandatory commands (although an empty `SINDARIN` script is not really useful). The details and options are explained in later sections.

4.4.1   Process Configuration

model

> `model = ⟨model-name⟩`

This assignment sets or resets the current physics model. The Standard Model is already preloaded, so the `model` assignment applies to non-default models. Obviously, the model must be known to `WHIZARD`. Example:

> `model = MSSM`

See Sec. 5.3.

alias

> `alias ⟨alias-name⟩ = ⟨alias-definition⟩`

Particles are specified by their names. For most particles, there are various equivalent names. Names containing special characters such as a `+` sign have to be quoted. The `alias` assignment defines an alias for a list of particles. This is useful for setting up processes with sums over flavors, cut expressions, and more. The alias name is then used like a simple particle name. Example:

> `alias jet = u:d:s:U:D:S:g`

See Sec. 5.2.1.

process

```
process ⟨tag⟩ = ⟨incoming⟩ => ⟨outgoing⟩
```

Define a process. You give the process a name ⟨*tag*⟩ by which it is identified later, and specify the incoming and outgoing particles, and possibly options. You can define an arbitrary number of processes as long as they are distinguished by their names. Example:

```
process w_plus_jets = g, g => "W+", jet, jet
```

See Sec. 5.4.

sqrts

```
sqrts = ⟨energy-value⟩
```

Define the center-of-mass energy for collision processes. The default setup will assume head-on central collisions of two beams. Example:

```
sqrts = 500 GeV
```

See Sec. 5.5.1.

beams

```
beams = ⟨beam-particles⟩
beams = ⟨beam-particles⟩ => ⟨structure-function-setup⟩
```

Declare beam particles and properties. The current value of `sqrts` is used, unless specified otherwise. Example:

```
beams = u:d:s, U:D:S => lhapdf
```

With options, the assignment allows for defining beam structure in some detail. This includes beamstrahlung and ISR for lepton colliders, precise structure function definition for hadron colliders, asymmetric beams, beam polarization, and more. See Sec. 5.5.

4.4.2 Parameters

Parameter settings

```
⟨parameter⟩ = ⟨value⟩
⟨type⟩ ⟨user-parameter⟩
⟨type⟩ ⟨user-parameter⟩ = ⟨value⟩
```

Specify a value for a parameter. There are predefined parameters that affect the behavior of a command, model-specific parameters (masses, couplings), and user-defined parameters. The latter have to be declared with a type, which may be `int` (integer), `real`, `complex`, `logical`, `string`, or `alias`. Logical parameter names begin with a question mark, string parameter names with a dollar sign. Examples:

```
mb = 4.2 GeV
?rebuild_grids = true
real mass_sum = mZ + mW
string $message = "This is a string"
```

The value need not be a literal, it can be an arbitrary expression of the correct type. See Sec. 4.7.

read_slha

> read_slha (⟨*filename*⟩)

This is useful only for supersymmetric models: read a parameter file in the SUSY Les Houches Accord format. The file defines parameter values and, optionally, decay widths, so this command removes the need for writing assignments for each of them.

```
read_slha ("sps1a.slha")
```

See Sec. 10.2.

show

> show (⟨*data-objects*⟩)

Print the current value of some data object. This includes not just variables, but also models, libraries, cuts, etc. This is rather a debugging aid, so don't expect the output to be concise in the latter cases. Example:

```
show (mH, wH)
```

See Sec. 5.10.

printf

> printf ⟨*format-string*⟩ (⟨*data-objects*⟩)

Pretty-print the data objects according to the given format string. If there are no data objects, just print the format string. This command is borrowed from the C programming language; it is actually an interface to the system's printf(3) function. The conversion specifiers are restricted to d,i,e,f,g,s, corresponding to the output of integer, real, and string variables. Example:

```
printf "The Higgs mass is %f GeV" (mH)
```

See Sec. 5.10.

4.4.3   Integration

cuts

```
cuts = ⟨logical-cut-expression⟩
```

The cut expression is a logical macro expression that is evaluated for each phase space point during integration and event generation. You may construct expressions out of various observables that are computed for the (partonic) particle content of the current event. If the expression evaluates to `true`, the matrix element is calculated and the event is used. If it evaluates to `false`, the matrix element is set zero and the event is discarded. Note that for collisions the expression is evaluated in the lab frame, while for decays it is evaluated in the rest frame of the decaying particle. In case you want to impose cuts on a factorized process, i.e. a combination of a production process and one or more decay processes, you have to use the `selection` keyword instead.

Example for the keyword `cuts`:

```
cuts = all Pt > 20 GeV [jet]
   and  all mZ - 10 GeV < M < mZ + 10 GeV [lepton, lepton]
   and  no  abs (Eta) < 2 [jet]
```

See Sec. 5.2.5.

integrate

```
integrate (⟨process-tags⟩)
```

Compute the total cross section for a process. The command takes into account the definition of the process, the beam setup, cuts, and parameters as defined in the script. Parameters may also be specified as options to the command.

Integration is necessary for each process for which you want to know total or differential cross sections, or event samples. Apart from computing a value, it sets up and adapts phase space and integration grids that are used in event generation. If you just need an event sample, you can omit an explicit `integrate` command; the `simulate` command will call it automatically. Example:

```
integrate (w_plus_jets, z_plus_jets)
```

See Sec. 5.7.1.

?phs_only/n_calls_test

```
integrate (⟨process-tag⟩) { ?phs_only = true n_calls_test = 1000 }
```

These are just optional settings for the `integrate` command discussed just a second ago. The `?phs_only = true` (note that variables starting with a question mark are logicals) option tells `WHIZARD` to prepare a process for integration, but instead of performing the integration, just to generate a phase space parameterization. `n_calls_test = <num>` evaluates the sampling function for random integration channels and random momenta. `VAMP` integration grids are neither generated nor used, so the channel selection corresponds to the first integration pass,

before any grids or channel weights are adapted. The number of sampling points is given by `<num>`. The output contains information about the timing, number of sampling points that passed the kinematics selection, and the number of matrix-element values that were actually evaluated. This command is useful mainly for debugging and diagnostics. Example:

```
integrate (some_large_process) { ?phs_only = true  n_calls_test = 1000 }
```

(Note that there used to be a separate command `matrix_element_test` until version 2.1.1 of WHIZARD which has been discarded in order to simplify the SINDARIN syntax.)

4.4.4   Events

histogram

> histogram ⟨*tag*⟩ (⟨*lower-bound*⟩, ⟨*upper-bound*⟩)
> histogram ⟨*tag*⟩ (⟨*lower-bound*⟩, ⟨*upper-bound*⟩, ⟨*step*⟩)

Declare a histogram for event analysis. The histogram is filled by an analysis expression, which is evaluated once for each event during a subsequent simulation step. Example:

```
histogram pt_distribution (0, 150 GeV, 10 GeV)
```

See Sec. 5.9.3.

plot

> plot ⟨*tag*⟩

Declare a plot for displaying data points. The plot may be filled by an analysis expression that is evaluated for each event; this would result in a scatter plot. More likely, you will use this feature for displaying data such as the energy dependence of a cross section. Example:

```
plot total_cross_section
```

See Sec. 5.9.4.

selection

> selection = ⟨*selection-expression*⟩

The selection expression is a logical macro expression that is evaluated once for each event. It is applied to the event record, after all decays have been executed (if any). It is therefore intended e.g. for modelling detector acceptance cuts etc. For unfactorized processes the usage of `cuts` or `selection` leads to the same results. Events for which the selection expression evaluates to false are dropped; they are neither analyzed nor written to any user-defined output file. However, the dropped events are written to WHIZARD's native event file. For unfactorized processes it is therefore preferable to implement all cuts using the `cuts` keyword for the integration, see `cuts` above. Example:

```
selection = all Pt > 50 GeV [lepton]
```

The syntax is generically the same as for the `cuts expression`, see Sec. 5.2.5. For more information see also Sec. 5.9.

analysis

> `analysis = ⟨`*analysis-expression*`⟩`

The analysis expression is a logical macro expression that is evaluated once for each event that passes the integration and selection cuts in a subsequent simulation step. The expression has type logical in analogy with the cut expression; however, its main use will be in side effects caused by embedded `record` expressions. The `record` expression books a value, calculated from observables evaluated for the current event, in one of the predefined histograms or plots. Example:

```
analysis = record pt_distribution (eval Pt [photon])
      and  record mval (eval M [lepton, lepton])
```

See Sec. 5.9.

unstable

> `unstable ⟨`*particle*`⟩ (⟨`*decay-channels*`⟩)`

Specify that a particle can decay, if it occurs in the final state of a subsequent simulation step. (In the integration step, all final-state particles are considered stable.) The decay channels are processes which should have been declared before by a `process` command (alternatively, there are options that `WHIZARD` takes care of this automatically; cf. Sec. 5.8.2). They may be integrated explicitly, otherwise the `unstable` command will take care of the integration before particle decays are generated. Example:

```
unstable Z (z_ee, z_jj)
```

Note that the decay is an on-shell approximation. Alternatively, `WHIZARD` is capable of generating the final state(s) directly, automatically including the particle as an internal resonance together with irreducible background. Depending on the physical problem and on the complexity of the matrix-element calculation, either option may be more appropriate.

See Sec. 5.8.2.

n_events

> `n_events = ⟨`*integer*`⟩`

Specify the number of events that a subsequent simulation step should produce. By default, simulated events are unweighted. (Unweighting is done by a rejection operation on weighted events, so the usual caveats on event unweighting by a numerical Monte-Carlo generator do apply.) Example:

```
n_events = 20000
```

See Sec. 5.8.1.

simulate

> `simulate (⟨process-tags⟩)`

Generate an event sample. The command allows for analyzing the generated events by the `analysis` expression. Furthermore, events can be written to file in various formats. Optionally, the partonic events can be showered and hadronized, partly using included external (`PYTHIA`) or truly external programs called by `WHIZARD`. Example:

> `simulate (w_plus_jets) { sample_format = lhef }`

See Sec. 5.8.1 and Chapter 11.

graph

> `graph (⟨tag⟩) = ⟨histograms-and-plots⟩`

Combine existing histograms and plots into a common graph. Also useful for pretty-printing single histograms or plots. Example:

```
graph comparison {
  $title = "$p_T$ distribution for two different values of $m_h$"
} = hist1 & hist2
```

See Sec. 12.4.

write_analysis

> `write_analysis (⟨analysis-objects⟩)`

Writes out data tables for the specified analysis objects (plots, graphs, histograms). If the argument is empty or absent, write all analysis objects currently available. The tables are available for feeding external programs. Example:

> `write_analysis`

See Sec. 5.9.

compile_analysis

> `compile_analysis (⟨analysis-objects⟩)`

Analogous to `write_analysis`, but the generated data tables are processed by LaTeX and `gamelan`, which produces Postscript and PDF versions of the displayed data. Example:

> `compile_analysis`

See Sec. 5.9.

4.5 Control Structures

Like any complete programming language, `SINDARIN` provides means for branching and looping the program flow.

4.5.1 Conditionals

if

```
if ⟨logical_expression⟩ then ⟨statements⟩
elsif ⟨logical_expression⟩ then ⟨statements⟩
else ⟨statements⟩
endif
```

Execute statements conditionally, depending on the value of a logical expression. There may be none or multiple `elsif` branches, and the `else` branch is also optional. Example:

```
if (sqrts > 2 * mtop) then
  integrate (top_pair_production)
else
  printf "Top pair production is not possible"
endif
```

The current `SINDARIN` implementation puts some restriction on the statements that can appear in a conditional. For instance, process definitions must be done unconditionally.

4.5.2 Loops

scan

```
scan ⟨variable⟩ = (⟨value-list⟩) { ⟨statements⟩ }
```

Execute the statements repeatedly, once for each value of the scan variable. The statements are executed in a local context, analogous to the option statement list for commands. The value list is a comma-separated list of expressions, where each item evaluates to the value that is assigned to ⟨*variable*⟩ for this iteration.

The type of the variable is not restricted to numeric, scans can be done for various object types. For instance, here is a scan over strings:

```
scan string $str = ("%.3g", "%.4g", "%.5g") { printf $str (mW) }
```

The output:

```
[user variable] $str = "%.3g"
80.4
[user variable] $str = "%.4g"
80.42
[user variable] $str = "%.5g"
80.419
```

For a numeric scan variable in particular, there are iterators that implement the usual functionality of `for` loops. If the scan variable is of type integer, an iterator may take one of the forms

⟨*start-value*⟩ => ⟨*end-value*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /+ ⟨*add-step*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /- ⟨*subtract-step*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /* ⟨*multiplicator*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ // ⟨*divisor*⟩

The iterator can be put in place of an expression in the ⟨*value-list*⟩. Here is an example:

```
scan int i = (1, (3 => 5), (10 => 20 /+ 4))
```

which results in the output

```
[user variable] i =             1
[user variable] i =             3
[user variable] i =             4
[user variable] i =             5
[user variable] i =            10
[user variable] i =            14
[user variable] i =            18
```

[Note that the ⟨*statements*⟩ part of the scan construct may be empty or absent.]

For real scan variables, there are even more possibilities for iterators:

⟨*start-value*⟩ => ⟨*end-value*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /+ ⟨*add-step*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /- ⟨*subtract-step*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /* ⟨*multiplicator*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ // ⟨*divisor*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /+/ ⟨*n-points-linear*⟩
⟨*start-value*⟩ => ⟨*end-value*⟩ /*/ ⟨*n-points-logarithmic*⟩

The first variant is equivalent to /+ 1. The /+ and /- operators are intended to add or subtract the given step once for each iteration. Since in floating-point arithmetic this would be plagued by rounding ambiguities, the actual implementation first determines the (integer) number of iterations from the provided step value, then recomputes the step so that the iterations are evenly spaced with the first and last value included.

The /* and // operators are analogous. Here, the initial value is intended to be multiplied by the step value once for each iteration. After determining the integer number of iterations, the actual scan values will be evenly spaced on a logarithmic scale.

Finally, the /+/ and /*/ operators allow to specify the number of iterations (not counting the initial value) directly. The ⟨*start-value*⟩ and ⟨*end-value*⟩ are always included, and the intermediate values will be evenly spaced on a linear (/+/) or logarithmic (/*/) scale.

Example:

```
scan real mh = (130 GeV,
          (140 GeV => 160 GeV /+ 5 GeV),
          180 GeV,
          (200 GeV => 1 TeV /*/ 10))
   {  integrate (higgs_decay) }
```

4.5.3 Including Files

include

> include (⟨*file-name*⟩)

Include a `SINDARIN` script from the specified file. The contents must be complete commands; they are compiled and executed as if they were part of the current script. Example:

> include ("default_cuts.sin")

4.6 Expressions

`SINDARIN` expressions are classified by their types. The type of an expression is verified when the script is compiled, before it is executed. This provides some safety against simple coding errors.

Within expressions, grouping is done using ordinary brackets `()`. For subevent expressions, use square brackets `[]`.

4.6.1 Numeric

The language supports the classical numeric types

- `int` for integer: machine-default, usually 32 bit;

- `real`, usually *double precision* or 64 bit;

- `complex`, consisting of real and imaginary part equivalent to a `real` each.

`SINDARIN` supports arithmetic expressions similar to conventional languages. In arithmetic expressions, the three numeric types can be mixed as appropriate. The computation essentially follows the rules for mixed arithmetic in `Fortran`. The arithmetic operators are `+`, `-`, `*`, `/`, `^`. Standard functions such as `sin`, `sqrt`, etc. are available. See Sec. 5.1.1 to Sec. 5.1.3.

Numeric values can be associated with units. Units evaluate to numerical factors, and their use is optional, but they can be useful in the physics context for which `WHIZARD` is designed. Note that the default energy/mass unit is `GeV`, and the default unit for cross sections is `fbarn`.

4.6.2 Logical and String

The language also has the following standard types:

- `logical` (a.k.a. boolean). Logical variable names have a `?` (question mark) as prefix.

- `string` (arbitrary length). String variable names have a `$` (dollar) sign as prefix.

There are comparisons, logical operations, string concatenation, and a mechanism for formatting objects as strings for output.

4.6.3   Special

Furthermore, `SINDARIN` deals with a bunch of data types tailored specifically for Monte Carlo applications:

- `alias` objects denote a set of particle species.

- `subevt` objects denote a collection of particle momenta within an event. They have their uses in cut and analysis expressions.

- `process` object are generated by a `process` statement. There are no expressions involving processes, but they are referred to by `integrate` and `simulate` commands.

- `model`: There is always a current object of type and name `model`. Several models can be used concurrently by appropriately defining processes, but this happens behind the scenes.

- `beams`: Similarly, the current implementation allows only for a single object of this type at a given time, which is assigned by a `beams =` statement and used by `integrate`.

In the current implementation, `SINDARIN` has no container data types derived from basic types, such as lists, arrays, or hashes, and there are no user-defined data types. (The `subevt` type is a container for particles in the context of events, but there is no type for an individual particle: this is represented as a one-particle `subevt`). There are also containers for inclusive processes which are however simply handled as an expansion into several components of a master process tag.

4.7   Variables

`SINDARIN` supports global variables, variables local to a scoping unit (the option body of a command, the body of a `scan` loop), and variables local to an expression.

Some variables are predefined by the system (*intrinsic variables*). They are further separated into *independent* variables that can be reset by the user, and *derived* or locked variables that are automatically computed by the program, but not directly user-modifiable. On top of that, the user is free to introduce his own variables (*user variables*).

The names of numerical variables consist of alphanumeric characters and underscores. The first character must not be a digit. Logical variable names are furthermore prefixed by a `?` (question mark) sign, while string variable names begin with a `$` (dollar) sign.

Character case does matter. In this manual we follow the convention that variable names consist of lower-case letters, digits, and underscores only, but you may also use upper-case letters if you wish.

Physics models contain their own, specific set of numeric variables (masses, couplings). They are attached to the model where they are defined, so they appear and disappear with the model that is currently loaded. In particular, if two different models contain a variable with the same name, these two variables are nevertheless distinct: setting one doesn't affect the other. This feature might be called, in computer-science jargon, a *mixin*.

User variables – global or local – are declared by their type when they are introduced, and acquire an initial value upon declaration. Examples:

```
int i = 3
real my_cut_value = 10 GeV
complex c = 3 - 4 * I
logical ?top_decay_allowed = mH > 2 * mtop
string $hello = "Hello world!"
alias q = d:u:s:c
```

An existing user variable can be assigned a new value without a declaration:

```
i = i + 1
```

and it may also be redeclared if the new declaration specifies the same type, this is equivalent to assigning a new value.

Variables local to an expression are introduced by the `let ... in` contruct. Example:

```
real a = let int n = 2 in
            x^n + y^n
```

The explicit `int` declaration is necessary only if the variable `n` has not been declared before. An intrinsic variable must not be declared: `let mtop = 175.3 GeV in ...`

`let` constructs can be concatenated if several local variables need to be assigned: `let a = 3 in let b = 4 in expression`.

Variables of type `subevt` can only be defined in `let` constructs.

Exclusively in the context of particle selections (event analysis), there are *observables* as special numeric objects. They are used like numeric variables, but they are never declared or assigned. They get their value assigned dynamically, computed from the particle momentum configuration. Hence, they may be understood as (intrinsic and predefined) macros. By convention, observable names begin with a capital letter.

Further macros are

- `cuts` and `analysis`. They are of type logical, and can be assigned an expression by the user. They are evaluated once for each event.

- `scale`, `factorization_scale` and `renormalization_scale` are real numeric macros which define the energy scale(s) of an event. The latter two override the former. If no scale is defined, the partonic energy is used as the process scale.

- `weight` is a real numeric macro. If it is assigned an expression, the expression is evaluated for each valid phase-space point, and the result multiplies the matrix element.

Chapter 5

SINDARIN in Details

5.1 Data and expressions

5.1.1 Real-valued objects

Real literals have their usual form, mantissa and, optionally, exponent:

```
0.  3.14  -.5  2.345e-3  .890E-023
```

Internally, real values are treated as double precision. The values are read by the `Fortran` library, so details depend on its implementation.

A special feature of SINDARIN is that numerics (real and integer) can be immediately followed by a physical unit. The supported units are presently hard-coded, they are

```
meV  eV  keV  MeV  GeV  TeV
  nbarn  pbarn  fbarn  abarn
      rad  mrad  degree
              %
```

If a number is followed by a unit, it is automatically normalized to the corresponding default unit: `14.TeV` is transformed into the real number `14000`. Default units are `GeV`, `fbarn`, and `rad`. The `%` sign after a number has the effect that the number is multiplied by 0.01. Note that no checks for consistency of units are done, so you can add `1 meV + 3 abarn` if you absolutely wish to. Omitting units is always allowed, in that case, the default unit is assumed.

Units are not treated as variables. In particular, you can't write `theta / degree`, the correct form is `theta / 1 degree`.

There is a single predefined real constant, namely π which is referred to by the keyword `pi`. In addition, there is a single predefined complex constant, which is the complex unit i, being referred to by the keyword `I`.

The arithmetic operators are

$$+ - * / \char`\^$$

with their obvious meaning and the usual precedence rules.

SINDARIN supports a bunch of standard numerical functions, mostly equivalent to their Fortran counterparts:

$$\texttt{abs} \quad \texttt{conjg} \quad \texttt{sgn} \quad \texttt{mod} \quad \texttt{modulo}$$
$$\texttt{sqrt} \quad \texttt{exp} \quad \texttt{log} \quad \texttt{log10}$$
$$\texttt{sin} \quad \texttt{cos} \quad \texttt{tan} \quad \texttt{asin} \quad \texttt{acos} \quad \texttt{atan}$$
$$\texttt{sinh} \quad \texttt{cosh} \quad \texttt{tanh}$$

(Unlike Fortran, the sgn function takes only one argument and returns 1., or −1.) The function argument is enclosed in brackets: sqrt (2.), tan (11.5 degree).

There are two functions with two real arguments:

$$\texttt{max} \quad \texttt{min}$$

Example: `real lighter_mass = min (mZ, mH)`

The following functions of a real convert to integer:

$$\texttt{int} \quad \texttt{nint} \quad \texttt{floor} \quad \texttt{ceiling}$$

and this converts to complex type:

$$\texttt{complex}$$

Real values can be compared by the following operators, the result is a logical value:

$$\texttt{==} \quad \texttt{<>}$$
$$\texttt{>} \quad \texttt{<} \quad \texttt{>=} \quad \texttt{<=}$$

In SINDARIN, it is possible to have more than two operands in a logical expressions. The comparisons are done from left to right. Hence,

$$\texttt{115 GeV < mH < 180 GeV}$$

is valid SINDARIN code and evaluates to true if the Higgs mass is in the given range.

Tests for equality and inequality with machine-precision real numbers are notoriously unreliable and should be avoided altogether. To deal with this problem, SINDARIN has the possibility to make the comparison operators "fuzzy" which should be read as "equal (unequal) up to an absolute tolerance", where the tolerance is given by the real-valued intrinsic variable tolerance. This variable is initially zero, but can be set to any value (for instance, tolerance = 1.e-13 by the user. Note that for non-zero tolerance, operators like == and <> or < and > are not mutually exclusive[1].

---

[1]In older versions of WHIZARD, until v2.1.1, there used to be separate comparators for the comparisons up to a tolerance, namely ==~ and <>~. These have been discarded from v2.2.0 on in order to simplify the syntax.

5.1.2   Integer-valued objects

Integer literals are obvious:

<div align="center">

1 -98765 0123

</div>

Integers are always signed. Their range is the default-integer range as determined by the `Fortran` compiler.

Like real values, integer values can be followed by a physical unit: `1 TeV`, `30 degree`. This actually transforms the integer into a real.

Standard arithmetics is supported:

<div align="center">

+ - * / ^

</div>

It is important to note that there is no fraction datatype, and pure integer arithmetics does not convert to real. Hence `3/4` evaluates to `0`, but `3 GeV / 4 GeV` evaluates to `0.75`.

Since all arithmetics is handled by the underlying `Fortran` library, integer overflow is not detected. If in doubt, do real arithmetics.

Integer functions are more restricted than real functions. We support the following:

<div align="center">

abs sgn mod modulo
max min

</div>

and the conversion functions

<div align="center">

real complex

</div>

Comparisons of integers among themselves and with reals are possible using the same set of comparison operators as for real values. This includes the operators with a finite tolerance.

5.1.3   Complex-valued objects

Complex variables and values are currently not yet used by the physics models implemented in `WHIZARD`. There complex input coupling constants are always split into their real and imaginary parts (or modulus and phase). They are exclusively available for arithmetic calculations.

There is no form for complex literals. Complex values must be created via an arithmetic expression,

<div align="center">

complex c = 1 + 2 * I

</div>

where the imaginary unit `I` is predefined as a constant.

The standard arithmetic operations are supported (also mixed with real and integer). Support for functions is currently still incomplete, among the supported functions there are `sqrt`, `log`, `exp`.

5.1.4   Logical-valued objects

There are two predefined logical constants, `true` and `false`. Logicals are *not* equivalent to integers (like in C) or to strings (like in PERL), but they make up a type of their own. Only in `printf` output, they are treated as strings, that is, they require the `%s` conversion specifier.

The names of logical variables begin with a question mark `?`. Here is the declaration of a logical user variable:

```
logical ?higgs_decays_into_tt = mH > 2 * mtop
```

Logical expressions use the standard boolean operations

$$\text{or} \quad \text{and} \quad \text{not}$$

The results of comparisons (see above) are logicals.

There is also a special logical operator with lower priority, concatenation by a semicolon:

$$lexpr1 \; ; \; lexpr2$$

This evaluates *lexpr1* and throws its result away, then evaluates *lexpr2* and returns that result. This feature is to used with logical expressions that have a side effect, namely the `record` function within analysis expressions.

The primary use for intrinsic logicals are flags that change the behavior of commands. For instance, `?unweighted = true` and `?unweighted = false` switch the unweighting of simulated event samples on and off.

5.1.5   String-valued objects and string operations

String literals are enclosed in double quotes: `"This is a string."` The empty string is `""`. String variables begin with the dollar sign: `$`. There is only one string operation, concatenation

```
string $foo = "abc" & "def"
```

However, it is possible to transform variables and values to a string using the `sprintf` function. This function is an interface to the system's C function `sprintf` with some restrictions and modifications. The allowed conversion specifiers are

$$
\begin{array}{c}
\text{\%d} \quad \text{\%i (integer)} \\
\text{\%e} \quad \text{\%f} \quad \text{\%g} \quad \text{\%E} \quad \text{\%F} \quad \text{\%G (real)} \\
\text{\%s (string and logical)}
\end{array}
$$

The conversions can use flag parameter, field width, and precision, but length modifiers are not supported since they have no meaning for the application. (See also Sec. 5.10.)

The `sprintf` function has the syntax

$$\texttt{sprintf} \; \textit{format-string} \; (\textit{arg-list})$$

This is an expression that evaluates to a string. The format string contains the mentioned conversion specifiers. The argument list is optional. The arguments are separated by commas. Allowed arguments are integer, real, logical, and string variables, and numeric expressions. Logical and string expressions can also be printed, but they have to be dressed as *anonymous variables*. A logical anonymous variable has the form `?(`*logical_expr*`)` (example: `?(mH > 115 GeV)`). A string anonymous variable has the form `$(`*string-expr*`)`.

Example:

```
string $unit = "GeV"
string $str = sprintf "mW = %f %s" (mW, $unit)
```

The related `printf` command with the same syntax prints the formatted string to standard output[2].

5.2 Particles and (sub)events

5.2.1 Particle aliases

A particle species is denoted by its name as a string: `"W+"`. Alternatively, it can be addressed by an `alias`. For instance, the W^+ boson has the alias `Wp`. Aliases are used like variables in a context where a particle species is expected, and the user can specify his/her own aliases.

An alias may either denote a single particle species or a class of particles species. A colon `:` concatenates particle names and aliases to yield multi-species aliases:

```
alias quark = u:d:s
alias wboson = "W+":"W-"
```

Such aliases are used for defining processes with summation over flavors, and for defining classes of particles for analysis.

Each model files define both names and (single-particle) aliases for all particles it contains. Furthermore, it defines the class aliases `colored` and `charged` which are particularly useful for event analysis.

5.2.2 Subevents

Subevents are sets of particles, extracted from an event. The sets are unordered by default, but may be ordered by appropriate functions. Obviously, subevents are meaningful only in a context where an event is available. The possible context may be the specification of a cut, weight, scale, or analysis expression.

To construct a simple subevent, we put a particle alias or an expression of type particle alias into square brackets:

---

[2]In older versions of WHIZARD, until v2.1.1, there also used to be a `sprintd` function and a `printd` command for default formats without a format string. They have been discarded in order to simplify the syntax from version v2.2.0 on.

```
["W+"]   [u:d:s]   [colored]
```

These subevents evaluate to the set of all W^+ bosons (to be precise, their four-momenta), all u, d, or s quarks, and all colored particles, respectively.

A subevent can contain pseudoparticles, i.e., particle combinations. That is, the four-momenta of distinct particles are combined (added conmponent-wise), and the results become subevent elements just like ordinary particles.

The (pseudo)particles in a subevent are non-overlapping. That is, for any of the particles in the original event, there is at most one (pseudo)particle in the subevent in which it is contained.

Sometimes, variables (actually, named constants) of type subevent are useful. Subevent variables are declared by the `subevt` keyword, and their names carry the prefix `@`. Subevent variables exist only within the scope of a `cuts` (or `scale`, `analysis`, etc.) macro, which is evaluated in the presence of an actual event. In the macro body, they are assigned via the `let` construct:

```
cuts =
  let subevt @jets = select if Pt > 10 GeV [colored]
  in
  all Theta > 10 degree [@jets, @jets]
```

In this expression, we first define `@jets` to stand for the set of all colored partons with $p_T > 10$ GeV. This abbreviation is then used in a logical expression, which evaluates to true if all relative angles between distinct jets are greater than 10 degree.

We note that the example also introduces pairs of subevents: the square bracket with two entries evaluates to the list of all possible pairs which do not overlap. The objects within square brackets can be either subevents or alias expressions. The latter are transformed into subevents before they are used.

As a special case, the original event is always available as the predefined subevent `@evt`.

5.2.3 Subevent functions

There are several functions that take a subevent (or an alias) as an argument and return a new subevent. Here we describe them:

collect

```
collect [particles]
collect if condition [particles]
collect if condition [particles, ref_particles]
```

First version: collect all particle momenta in the argument and combine them to a single four-momentum. The *particles* argument may either be a `subevt` expression or an `alias` expression. The result is a one-entry `subevt`. In the second form, only those particles are collected which satisfy the *condition*, a logical expression. Example: `collect if Pt > 10 GeV [colored]`

The third version is useful if you want to put binary observables (i.e., observables constructed from two different particles) in the condition. The *ref_particles* provide the second argument for

binary observables in the *condition*. A particle is taken into account if the condition is true with respect to all reference particles that do not overlap with this particle. Example: `collect if Theta > 5 degree [photon, charged]`: combine all photons that are separated by 5 degrees from all charged particles.

cluster

```
cluster [particles]
cluster if condition [particles]
```

First version: collect all particle momenta in the argument and cluster them to a set of jets. The *particles* argument may either be a `subevt` expression or an `alias` expression. The result is a one-entry `subevt`. In the second form, only those particles are clustered which satisfy the *condition*, a logical expression. Example: `cluster if Pt > 10 GeV [colored]`

This command is available from `WHIZARD` version 2.2.1 on, and only if the `FastJet` package has been installed and linked with `WHIZARD` (cf. Sec.2.2.12); in a future version of `WHIZARD` it is foreseen to have also an intrinsic clustering package inside `WHIZARD` which will be able to support some of the clustering algorithms below. To use it in an analysis, you have to set the variable `jet_algorithm` to one of the predefined jet-algorithm values (integer constants):

```
kt_algorithm
cambridge_algorithm
antikt_algorithm
genkt_algorithm
cambridge_for_passive_algorithm
genkt_for_passive_algorithm
ee_kt_algorithm
ee_genkt_algorithm
plugin_algorithm
```

and the variable `jet_r` to the desired R parameter value, as appropriate for the analysis and the jet algorithm. Example:

```
jet_algorithm = antikt_algorithm
jet_r = 0.7
cuts = all Pt > 15 GeV [cluster if Pt > 5 GeV [colored]]
```

select_b_jet, select_non_b_jet, select_c_jet, select_light_jet

This command is available from `WHIZARD` version 2.8.1 on, and it only generates anything non-trivial if the `FastJet` package has been installed and linked with `WHIZARD` (cf. Sec.2.2.12). It only returns sensible results when it is applied to subevents after the `cluster` command (cf. the paragraph before). It is similar to the `select` command, and accepts a logical expression as a possible condition. The four commands `select_b_jet`, `select_non_b_jet`, `select_c_jet`, and `select_light_jet` select b jets, non-b jets (anything lighter than bs), c jets (neither b nor light) and light jets (anything besides b and c), respectively. An example looks like this:

```
alias lightjet = u:U:d:D:s:S:c:C:gl
alias jet = b:B:lightjet
process eebbjj = e1, E1 => b, B, lightjet, lightjet
jet_algorithm = antikt_algorithm
jet_r = 0.5
cuts = let subevt @clustered_jets = cluster [jet] in
       let subevt @bjets = select_b_jet [@clustered_jets] in
       ....
```

photon_isolation

This command is available from `WHIZARD` version 2.8.1 on. It provides isolation of photons from hadronic (and possibly electromagnetic) activity in the event to define a (especially) NLO cross section that is completely perturbative. The isolation criterion according to Frixione, cf. [61], removes the non-perturbative contribution from the photon fragmentation function. This command can in principle be applied to elementary hard process partons (and leptons), but generates something sensible only if the `FastJet` package has been installed and linked with `WHIZARD` (cf. Sec.2.2.12). There are three parameters which allow to tune the isolation, `photon_iso_r0`, which is the radius R_γ^0 of the isolation cone, `photon_iso_eps`, which is the fraction ϵ_γ of the photon (transverse) energy that enters the isolation criterion, and the exponent of the isolation cone, `photon_iso_n`, n^γ. For more information cf. [61]. The command allows also a conditional cut on the photon which is applied before the isolation takes place. The first argument are the photons in the event, the second the particles from which they should be isolated. If also the electromagnetic activity is to be isolated, photons need to be isolated from themselves and must be included in the second argument. This is mandatory if leptons appear in the second argument. Two examples look like this:

```
alias jet = u:U:d:D:s:S:c:C:gl
process eeaajj = e1, E1 => A, A, jet, jet
jet_algorithm = antikt_algorithm
jet_r = 0.5
cuts = photon_isolation if Pt > 10 GeV [A, jet]
       ....
cuts = let subevt @jets = cluster [jet] in
       photon_isolation if Pt > 10 GeV [A, @jets]
       ....
process eeajmm = e1, E1 => A, jet, e2, E2
cuts = let subevt @jets = cluster [jet] in
       let subevt @iso = join [@jets, A:e2:E2] in
       photon_isolation [A, @iso]
```

photon_recombination

```
photon_recombination [particles]
photon_recombination if condition [particles]
```

This function, which maps a subevent into another subevent, is used for electroweak (and mixed coupling) higher order calculations. It takes the selection of photons in `particles` (for

the moment, `WHIZARD` restricts this to one explicit photon in the final state) and recombines it with the closest non-photon particle from `particles` in R-distance, if the R-distance is smaller than the parameter set by `photon_rec_r0`. Otherwise the `particles` subevent is left unchanged so that it may contain possibly non-recombined photons. The logical variable `?keep_flavors_when_recombining` determines whether `WHIZARD` keeps the flavor of the particle with which the photon is recombined into the pseudoparticle, the default being `true`. An example for photon recombination is shown here:

```
alias lep = e1:e2:e3:E1:E2:E3
process eevv = e1, E1 => A, lep, lep, lep, lep
photon_rec_r0 = 0.15
cuts = let subevt @reco =
          photon_recombination if abs (Eta) < 2.5 [A:lep] in
          ....
```

combine

```
combine [particles_1, particles_2]
combine if condition [particles_1, particles_2]
```

Make a new subevent of composite particles. The composites are generated by combining all particles from subevent *particles_1* with all particles from subevent *particles_2* in all possible combinations. Overlapping combinations are excluded, however: if a (composite) particle in the first argument has a constituent in common with a composite particle in the second argument, the combination is dropped. In particular, this applies if the particles are identical.

If a *condition* is provided, the combination is done only when the logical expression, applied to the particle pair in question, returns true. For instance, here we reconstruct intermediate W^- bosons:

```
let @W_candidates = combine if 70 GeV < M < 80 GeV ["mu-", "numubar"]
in ...
```

Note that the combination may fail, so the resulting subevent could be empty.

operator +

If there is no condition, the + operator provides a convenient shorthand for the `combine` command. In particular, it can be used if there are several particles to combine. Example:

```
cuts = any 170 GeV < M < 180 GeV [b + lepton + invisible]
```

select

```
select if condition [particles]
select if condition [particles, ref_particles]
```

One argument: select all particles in the argument that satisfy the *condition* and drop the rest. Two arguments: the *ref_particles* provide a second argument for binary observables. Select particles if the condition is satisfied for all reference particles.

extract

```
extract [particles]
extract index index-value [particles]
```

Return a single-particle subevent. In the first version, it contains the first particle in the
subevent *particles*. In the second version, the particle with index *index-value* is returned, where
index-value is an integer expression. If its value is negative, the index is counted from the end
of the subevent.

The order of particles in an event or subevent is not always well-defined, so you may wish to
sort the subevent before applying the *extract* function to it.

sort

```
sort [particles]
sort by observable [particles]
sort by observable [particles, ref_particle]
```

Sort the subevent according to some criterion. If no criterion is supplied (first version), the
subevent is sorted by increasing PDG code (first particles, then antiparticles). In the second
version, the *observable* is a real expression which is evaluated for each particle of the subevent
in turn. The subevent is sorted by increasing value of this expression, for instance:

```
let @sorted_evt = sort by Pt [@evt]
in ...
```

In the third version, a reference particle is provided as second argument, so the sorting can be
done for binary observables. It doesn't make much sense to have several reference particles at
once, so the sort function uses only the first entry in the subevent *ref-particle*, if it has more
than one.

join

```
join [particles, new_particles]
join if condition [particles, new_particles]
```

This commands appends the particles in subevent *new_particles* to the subevent *particles*, i.e., it
joins the two particle sets. To be precise, a (pseudo)particle from *new_particles* is only appended
if it does not overlap with any of the (pseudo)particles present in *particles*, so the function will
not produce overlapping entries.

In the second version, each particle from *new_particles* is also checked with all particles
in the first set whether *condition* is fulfilled. If yes, and there is no overlap, it is appended,
otherwise it is dropped.

operator &

Subevents can also be concatenated by the operator `&`. This effectively applies `join` to all operands in turn. Example:

```
let @visible =
      select if Pt > 10 GeV and E > 5 GeV [photon]
    & select if Pt > 20 GeV and E > 10 GeV [colored]
    & select if Pt > 10 GeV [lepton]
  in ...
```

5.2.4 Calculating observables

Observables (invariant mass `M`, energy `E`, ...) are used in expressions just like ordinary numeric variables. By convention, their names start with a capital letter. They are computed using a particle momentum (unary observables), or two particle momenta (binary observables) or all momenta of the particles (n-ary/subeventary observables) which are taken from a subsequent subevent argument.

We can extract the value of an observable for an event and make it available for computing the `scale` value, or for histogramming etc.:

eval

```
eval expr [particles]
eval expr [particles_1, particles_2]
```

The function `eval` takes an expression involving observables and evaluates it for the first momentum (or momentum pair) of the subevent (or subevent pair) in square brackets that follows the expression. For example,

```
eval Pt [colored]
```

evaluates to the transverse momentum of the first colored particle,

```
eval M [@jets, @jets]
```

evaluates to the invariant mass of the first distinct pair of jets (assuming that `@jets` has been defined in a `let` construct), and

```
eval E - M [combine [e1, N1]]
```

evaluates to the difference of energy and mass of the combination of the first electron-neutrino pair in the event.

The last example illustrates why observables are treated like variables, even though they are functions of particles: the `eval` construct with the particle reference in square brackets after the expression allows to compute derived observables – observables which are functions of new observables – without the need for hard-coding them as new functions.

For subeventary observables, e.g. `Ht`, the momenta of all particles in the subevent are taken to evaluate the observables, e.g.

```
eval Ht/2 [t:T:Z:jet]
```

takes the (half of) the transverse mass of all tops, Zs and jets in the final state.

sum

This `SINDARIN` statement works similar to the `eval` statement above, with the syntax

```
sum <expr> [<subevt>]
```

It sums the `<expr>` over all elements of the subevents `<subevt>`, e.g.

```
sum sqrt(Pt^2 + M^2)/2 [t:T:H:Z]
```

would calculate the transverse mass (square root of the sum of squared transverse momentum and squared mass) of all tops, Higgs and Z bosons in the final state.

prod

Identical to `sum`, but takes the product, not the sum of the expression `<expr>` evaluated over the full subevent. Syntax:

```
prod <expr> [<subevt>]
```

5.2.5   Cuts and event selection

Instead of a numeric value, we can use observables to compute a logical value.

all

```
all logical_expr [particles]
all logical_expr [particles_1, particles_2]
```

The `all` construct expects a logical expression and one or two subevent arguments in square brackets.

```
all Pt > 10 GeV [charged]
all 80 GeV < M < 100 GeV [lepton, antilepton]
```

In the second example, `lepton` and `antilepton` should be aliases defined in a `let` construct. (Recall that aliases are promoted to subevents if they occur within square brackets.)

This construction defines a cut. The result value is `true` if the logical expression evaluates to `true` for all particles in the subevent in square brackets. In the two-argument case it must be `true` for all non-overlapping combinations of particles in the two subevents. If one of the arguments is the empty subevent, the result is also `true`.

any

```
any logical_expr [particles]
any logical_expr [particles_1, particles_2]
```

The `any` construct is true if the logical expression is true for at least one particle or non-overlapping particle combination:

```
any E > 100 GeV [photon]
```

This defines a trigger or selection condition. If a subevent argument is empty, it evaluates to `false`

no

```
no logical_expr [particles]
no logical_expr [particles_1, particles_2]
```

The `no` construct is true if the logical expression is true for no single one particle or non-overlapping particle combination:

```
no 5 degree < Theta < 175 degree ["e-":"e+"]
```

This defines a veto condition. If a subevent argument is empty, it evaluates to `true`. It is equivalent to `not any...`, but included for notational convenience.

5.2.6 More particle functions

count

```
count [particles]
count [particles_1, particles_2]
count if logical-expr [particles]
count if logical-expr [particles, ref_particles]
```

This counts the number of events in a subevent, the result is of type `int`. If there is a conditional expression, it counts the number of `particle` in the subevent that pass the test. If there are two arguments, it counts the number of non-overlapping particle pairs (that pass the test, if any).

Predefined observables

The following real-valued observables are available in `SINDARIN` for use in `eval`, `all`, `any`, `no`, and `count` constructs. The argument is always the subevent or alias enclosed in square brackets.

- `M2`

  - One argument: Invariant mass squared of the (composite) particle in the argument.

  – Two arguments: Invariant mass squared of the sum of the two momenta.

- `M`

  – Signed square root of `M2`: positive if `M2` > 0, negative if `M2` < 0.

- `E`

  – One argument: Energy of the (composite) particle in the argument.
  – Two arguments: Sum of the energies of the two momenta.

- `Px, Py, Pz`

  – Like `E`, but returning the spatial momentum components.

- `P`

  – Like `E`, returning the absolute value of the spatial momentum.

- `Pt, Pl`

  – Like `E`, returning the transversal and longitudinal momentum, respectively.

- `Theta`

  – One argument: Absolute polar angle in the lab frame
  – Two arguments: Angular distance of two particles in the lab frame.

- `Theta_star` Only with two arguments, gives the relative polar angle of the two momenta in the rest system of the momentum sum (i.e. mother particle).

- `Phi`

  – One argument: Absolute azimuthal angle in the lab frame
  – Two arguments: Azimuthal distance of two particles in the lab frame

- `Rap, Eta`

  – One argument: rapidity / pseudorapidity
  – Two arguments: rapidity / pseudorapidity difference

- `Dist`

  – Two arguments: Distance on the η-ϕ cylinder, i.e., $\sqrt{\Delta\eta^2 + \Delta\phi^2}$

- `kT`

  – Two arguments: k_T jet clustering variable: $2\min(E_{j1}^2, E_{j2}^2)/Q^2 \times (1 - \cos\theta_{j1,j2})$. At the moment, $Q^2 = 1$ GeV^2.

There are also integer-valued observables:

- PDG

  – One argument: PDG code of the particle. For a composite particle, the code is undefined (value 0). For flavor sums in the `cuts` statement, this observable always returns the same flavor, i.e. the first one from the flavor list. It is thus only sensible to use it in an `analysis` or `selection` statement when simulating events.

- Ncol

  – One argument: Number of open color lines. Only count color lines, not anticolor lines. This is defined only if the global flag `?colorize_subevt` is true.

- Nacl

  – One argument: Number of open anticolor lines. Only count anticolor lines, not color lines. This is defined only if the global flag `?colorize_subevt` is true.

5.3   Physics Models

A physics model is a combination of particles, numerical parameters (masses, couplings, widths), and Feynman rules. Many physics analyses are done in the context of the Standard Model (SM). The SM is also the default model for WHIZARD. Alternatively, you can choose a subset of the SM (QED or QCD), variants of the SM (e.g., with or without nontrivial CKM matrix), or various extensions of the SM. The complete list is displayed in Table 10.1.

   The model definitions are contained in text files with filename extension `.mdl`, e.g., `SM.mdl`, which are located in the `share/models` subdirectory of the WHIZARD installation. These files are easily readable, so if you need details of a model implementation, inspect their contents. The model file contains the complete particle and parameter definitions as well as their default values. It also contains a list of vertices. This is used only for phase-space setup; the vertices used for generating amplitudes and the corresponding Feynman rules are stored in different files within the O'Mega source tree.

   In a SINDARIN script, a model is a special object of type `model`. There is always a *current* model. Initially, this is the SM, so on startup WHIZARD reads the `SM.mdl` model file and assigns its content to the current model object. (You can change the default model by the `--model` option on the command line. Also the preloading of a model can be switched off with the `--no-model` option) Once the model has been loaded, you can define processes for the model, and you have all independent model parameters at your disposal. As noted before, these are intrinsic parameters which need not be declared when you assign them a value, for instance:

```
mW = 80.33 GeV
wH = 243.1 MeV
```

Other parameters are *derived*. They can be used in expressions like any other parameter, they are also intrinsic, but they cannot be modified directly at all. For instance, the electromagnetic coupling `ee` is a derived parameter. If you change either `GF` (the Fermi constant), `mW` (the W mass), or `mZ` (the Z mass), this parameter will reflect the change, but setting it directly is an error. In other words, the SM is defined within `WHIZARD` in the G_F-m_W-m_Z scheme. (While this scheme is unusual for loop calculations, it is natural for a tree-level event generator where the Z and W poles have to be at their experimentally determined location[3].)

The model also defines the particle names and aliases that you can use for defining processes, cuts, or analyses.

If you would like to generate a SUSY process instead, for instance, you can assign a different model (cf. Table 10.1) to the current model object:

```
model = MSSM
```

This assignment has the consequence that the list of SM parameters and particles is replaced by the corresponding MSSM list (which is much longer). The MSSM contains essentially all SM parameters by the same name, but in fact they are different parameters. This is revealed when you say

```
model = SM
mb = 5.0 GeV
model = MSSM
show (mb)
```

After the model is reassigned, you will see the MSSM value of m_b which still has its default value, not the one you have given. However, if you revert to the SM later,

```
model = SM
show (mb)
```

you will see that your modification of the SM's m_b value has been remembered. If you want both mass values to agree, you have to set them separately in the context of their respective model. Although this might seem cumbersome at first, it is nevertheless a sensible procedure since the parameters defined by the user might anyhow not be defined or available for all chosen models.

When using two different models which need an SLHA input file, these *have* to be provided for both models.

Within a given scope, there is only one current model. The current model can be reset permanently as above. It can also be temporarily be reset in a local scope, i.e., the option body of a command or the body of a `scan` loop. It is thus possible to use several models within the same script. For instance, you may define a SUSY signal process and a pure-SM background process. Each process depends only on the respective model's parameter set, and a change to a parameter in one of the models affects only the corresponding process.

---

[3]In future versions of `WHIZARD` it is foreseen to implement other electroweak schemes.

5.4 Processes

The purpose of WHIZARD is the integration and simulation of high-energy physics processes: scatterings and decays. Hence, process objects play the central role in SINDARIN scripts.

A SINDARIN script may contain an arbitrary number of process definitions. The initial states need not agree, and the processes may belong to different physics models.

5.4.1 Process definition

A process object is defined in a straightforward notation. The definition syntax is straightforward:

```
process process-id = incoming-particles => outgoing-particles
```

Here are typical examples:

```
process w_pair_production = e1, E1 => "W+", "W-"
process zdecay = Z => u, ubar
```

Throughout the program, the process will be identified by its *process-id*, so this is the name of the process object. This identifier is arbitrary, chosen by the user. It follows the rules for variable names, so it consists of alphanumeric characters and underscores, where the first character is not numeric. As a special rule, it must not contain upper-case characters. The reason is that this name is used for identifying the process not just within the script, but also within the Fortran code that the matrix-element generator produces for this process.

After the equals sign, there follow the lists of incoming and outgoing particles. The number of incoming particles is either one or two: scattering processes and decay processes. The number of outgoing particles should be two or larger (as $2 \to 1$ processes are proportional to a δ function they can only be sensibly integrated when using a structure function like a hadron collider PDF or a beamstrahlung spectrum.). There is no hard upper limit; the complexity of processes that WHIZARD can handle depends only on the practical computing limitations (CPU time and memory). Roughly speaking, one can assume that processes up to $2 \to 6$ particles are safe, $2 \to 8$ processes are feasible given sufficient time for reaching a stable integration, while more complicated processes are largely unexplored.

We emphasize that in the default setup, the matrix element of a physics process is computed exactly in leading-order perturbation theory, i.e., at tree level. There is no restriction of intermediate states, the result always contains the complete set of Feynman graphs that connect the initial with the final state. If the result would actually be expanded in Feynman graphs (which is not done by the O'Mega matrix element generator that WHIZARD uses), the number of graphs can easily reach several thousands, depending on the complexity of the process and on the physics model.

More details about the different methods for quantum field-theoretical matrix elements can be found in Chap. 9. In the following, we will discuss particle names, options for processes like restrictions on intermediate states, parallelization, flavor sums and process components for inclusive event samples (process containers).

5.4.2   Particle names

The particle names are taken from the particle definition in the current model file. Looking at the SM, for instance, the electron entry in **share/models/SM.mdl** reads

```
particle E_LEPTON 11
  spin 1/2  charge  -1   isospin -1/2
  name "e-" e1 electron e
  anti "e+" E1 positron
  tex_name "e^-"
  tex_anti "e^+"
  mass me
```

This tells that you can identify an electron either as `"e-"`, `e1`, `electron`, or simply `e`. The first version is used for output, but needs to be quoted, because otherwise SINDARIN would interpret the minus sign as an operator. (Technically, unquoted particle identifiers are aliases, while the quoted versions – you can say either `e1` or `"e1"` – are names. On input, this makes no difference.) The alternative version `e1` follows a convention, inherited from `CompHEP` [64], that particles are indicated by lower case, antiparticles by upper case, and for leptons, the generation index is appended: `e2` is the muon, `e3` the tau. These alternative names need not be quoted because they contain no special characters.

In Table 5.1, we list the recommended names as well as mass and width parameters for all SM particles. For other models, you may look up the names in the corresponding model file.

Where no mass or width parameters are listed in the table, the particle is assumed to be massless or stable, respectively. This is obvious for particles such as the photon. For neutrinos, the mass is meaningless to particle physics collider experiments, so it is zero. For quarks, the u or d quark mass is unobservable directly, so we also set it zero. For the heavier quarks, the mass may play a role, so it is kept. (The s quark is borderline; one may argue that its mass is also unobservable directly.) On the other hand, the electron mass is relevant, e.g., in photon radiation without cuts, so it is not zero by default.

It pays off to set particle masses to zero, if the approximation is justified, since fewer helicity states will contribute to the matrix element. Switching off one of the helicity states of an external fermion speeds up the calculation by a factor of two. Therefore, script files will usually contain the assignments

```
me = 0  mmu = 0  ms = 0  mc = 0
```

unless they deal with processes where this simplification is phenomenologically unacceptable. Often m_τ and m_b can also be neglected, but this excludes processes where the Higgs couplings of τ or b are relevant.

Setting fermion masses to zero enables, furthermore, the possibility to define multi-flavor aliases

```
alias q = d:u:s:c
alias Q = D:U:S:C
```

and handle processes such as

	Particle	Output name	Alternative names	Mass	Width
Leptons	e^-	e-	e1 electron	me	
	e^+	e+	E1 positron	me	
	μ^-	mu-	e2 muon	mmu	
	μ^+	mu+	E2	mmu	
	τ^-	tau-	e3 tauon	mtau	
	τ^+	tau+	E3	mtau	
Neutrinos	ν_e	nue	n1		
	$\bar{\nu}_e$	nuebar	N1		
	ν_μ	numu	n2		
	$\bar{\nu}_\mu$	numubar	N2		
	ν_τ	nutau	n3		
	$\bar{\nu}_\tau$	nutaubar	N3		
Quarks	d	d	down		
	$\bar{d}$	dbar	D		
	u	u	up		
	$\bar{u}$	ubar	U		
	s	s	strange	ms	
	$\bar{s}$	sbar	S	ms	
	c	c	charm	mc	
	$\bar{c}$	cbar	C	mc	
	b	b	bottom	mb	
	$\bar{b}$	bbar	B	mb	
	t	t	top	mtop	wtop
	$\bar{t}$	tbar	T	mtop	wtop
Vector bosons	g	gl	g G gluon		
	γ	A	gamma photon		
	Z	Z		mZ	wZ
	W^+	W+	Wp	mW	wW
	W^-	W-	Wm	mW	wW
Scalar bosons	H	H	h Higgs	mH	wH

Table 5.1: *Names that can be used for SM particles. Also shown are the intrinsic variables that can be used to set mass and width, if applicable.*

```
process two_jets_at_ilc = e1, E1 => q, Q
process w_pairs_at_lhc = q, Q => Wp, Wm
```

where a sum over all allowed flavor combination is automatically included. For technical reasons, such flavor sums are possible only for massless particles (or more general for mass-degenerate particles). If you want to generate inclusive processes with sums over particles of different masses (e.g. summing over W/Z in the final state etc.), confer below the section about process components, Sec. 5.4.4.

Assignments of masses, widths and other parameters are actually in effect when a process is integrated, not when it is defined. So, these assignments may come before or after the process definition, with no significant difference. However, since flavor summation requires masses to be zero, the assignments may be put before the alias definition which is used in the process.

The muon, tau, and the heavier quarks are actually unstable. However, the width is set to zero because their decay is a macroscopic effect and, except for the muon, affected by hadron physics, so it is not described by WHIZARD. (In the current WHIZARD setup, all decays occur at the production vertex. A future version may describe hadronic physics and/or macroscopic particle propagation, and this restriction may be eventually removed.)

5.4.3 Options for processes

The process definition may contain an optional argument:

```
process process-id = incoming-particles => outgoing-particles {options...}
```

The *options* are a SINDARIN script that is executed in a context local to the process command. The assignments it contains apply only to the process that is defined. In the following, we describe the set of potentially useful options (which all can be also set globally):

Model reassignment

It is possible to locally reassign the model via a model = statment, permitting the definition of process using a model other than the globally selected model. The process will retain this association during integration and event generation.

Restrictions on matrix elements

Another useful option is the setting

```
$restrictions = string
```

This option allows to select particular classes of Feynman graphs for the process when using the O'Mega matrix element generator. The $restrictions string specifies e.g. propagators that the graph must contain. Here is an example:

```
process zh_invis = e1, E1 => n1:n2:n3, N1:N2:N3, H { $restrictions = "1+2 ~ Z" }
```

The complete process $e^-e^+ \to \nu\bar{\nu}H$, summed over all neutrino generations, contains both ZH pair production (Higgs-strahlung) and $W^+W^- \to H$ fusion. The restrictions string selects the Higgs-strahlung graph where the initial electrons combine to a Z boson. Here, the particles in the process are consecutively numbered, starting with the initial particles. An alternative for the same selection would be `$restrictions = "3+4 ~ Z"`. Restrictions can be combined using `&&`, for instance

```
$restrictions = "1+2 ~ Z && 3 + 4 ~ Z"
```

which is redundant here, however.

The restriction keeps the full energy dependence in the intermediate propagator, so the Breit-Wigner shape can be observed in distributions. This breaks gauge invariance, in particular if the intermediate state is off shell, so you should use the feature only if you know the implications. For more details, cf. the Chap. 9 and the `O'Mega` manual.

Other restrictions that can be combined with the restrictions above on intermediate propagators allow to exclude certain particles from intermediate propagators, or to exclude certain vertices from the matrix elements. For example,

```
process eemm = e1, E1 => e2, E2  { $restrictions = "!A" }
```

would exclude all photon propagators from the matrix element and leaves only the Z exchange here. In the same way, `$restrictions = "!gl"` would exclude all gluon exchange. This exclusion of internal propagators works also for lists of particles, like

```
$restrictions = "!Z:H"
```

excludes all Z and H propagators from the matrix elements.

Besides excluding certain particles as internal lines, it is also possible to exclude certain vertices using the restriction command

```
process eeww = e1, E1 => Wp, Wm  { $restrictions = "^[W+,W-,Z]" }
```

This would generate the matrix element for the production of two W bosons at LEP without the non-Abelian vertex W^+W^-Z. Again, these restrictions are able to work on lists, so

```
$restrictions = "^[W+,W-,A:Z]"
```

would exclude all triple gauge boson vertices from the above process and leave only the t-channel neutrino exchange.

It is also possible to exlude vertices by their coupling constants, e.g. the photon exchange in the process $e^+e^- \to \mu^+\mu^-$ can also be removed by the following restriction:

```
$restrictions = "^qlep"
```

Here, `qlep` is the `Fortran` variable for the coupling constant of the electron-positron-photon vertex.

The Tab. 5.2 gives a list of options that can be applied to the `O'Mega` matrix elements.

`3+4~Z`	external particles 3 and 4 must come from intermediate Z
`&&`	logical "and", e.g. in `3+5~t && 4+6~tbar`
`!A`	exclude all γ propagators
`!e+:nue`	exclude a list of propagators, here γ, ν_e
`^qlep:gnclep`	exclude all vertices with `qlep,gnclep` coupling constants
`^[A:Z,W+,W-]`	exclude all vertices W^+W^-Z, $W^+W^-\gamma$
`^c1:c2:c3[H,H,H]`	exclude all triple Higgs couplings with c_i constants

Table 5.2: *List of possible restrictions that can be applied to* `O'Mega` *matrix elements.*

Other options

There are some further options that the `O'Mega` matrix-element generator can take. If desired, any string of options that is contained in this variable

    $omega_flags = *string*

will be copied verbatim to the `O'Mega` call, after all other options.

One important application is the scheme of treating the width of unstable particles in the t-channel. This is modified by the `model:` class of `O'Mega` options.

It is well known that for some processes, e.g., single W production from photon-W fusion, gauge invariance puts constraints on the treatment of the unstable-particle width. By default, `O'Mega` puts a nonzero width in the s channel only. This correctly represents the resummed Dyson series for the propagator, but it violates QED gauge invariance, although the effect is only visible if the cuts permit the photon to be almost on-shell.

An alternative is

    $omega_flags = "-model:fudged_width" ,

which puts zero width in the matrix element, so that gauge cancellations hold, and reinstates the s-channel width in the appropriate places by an overall factor that multiplies the whole matrix element. Note that the fudged width option only applies to charged unstable particles, such as the W boson or top quark. Another possibility is

    $omega_flags = "-model:constant_width" ,

which puts the width both in the s- and in the t-channel like diagrams. A third option is provided by the running width scheme

    $omega_flags = "-model:running_width" ,

which applies the width only for s-channel like diagrams and multiplies it by a factor of p^2/M^2. The additional p^2-dependent factor mimicks the momentum dependence of the imaginary part of a vacuum polarization for a particle decaying into massles decay products. It is noted that none of the above options preserves gauge invariance.

For a gauge preserving approach (at least at tree level), `O'Mega` provides the complex-mass scheme

```
$omega_flags = "-model:cms_width .
```

However, in this case, one also has to modify the model in usage. For example, the parameter setting for the Standard Model can be changed by,

```
model = SM (Complex_Mass_Scheme) .
```

Multithreaded calculation of helicity sums via OpenMP

On multicore and / or multiprocessor systems, it is possible to speed up the calculation by using multiple threads to perform the helicity sum in the matrix element calculation. As the processing time used by WHIZARD is not used up solely in the matrix element, the speedup thus achieved varies greatly depending on the process under consideration; while simple processes without flavor sums do not profit significantly from this parallelization, the computation time for processes involving flavor sums with four or more particles in the final state is typically reduced by a factor between two and three when utilizing four parallel threads.

The parallization is implemented using OpenMP and requires WHIZARD to be compiled with an OpenMP aware compiler and the appropiate compiler flags This is done in the configuration step, cf. Sec. 2.3.

As with all OpenMP programs, the default number of threads used at runtime is up to the compiler runtime support and typically set to the number of independent hardware threads (cores / processors / hyperthreads) available in the system. This default can be adjusted by setting the OMP_NUM_THREADS environment variable prior to calling WHIZARD. Alternatively, the available number of threads can be reset anytime by the SINDARIN parameter openmp_num_threads. Note however that the total number of threads that can be sensibly used is limited by the number of nonvanishing helicity combinations.

5.4.4 Process components

It was mentioned above that processes with flavor sums (in the initial or final state or both) have to be mass-degenerate (in most cases massless) in all particles that are summed over at a certain position. This condition is necessary in order to use the same phase-space parameterization and integration for the flavor-summed process. However, in many applications the user wants to handle inclusive process definitions, e.g. by defining inclusive decays, inclusive SUSY samples at hadron colliders (gluino pairs, squark pairs, gluino-squark associated production), or maybe lepton-inclusive samples where the tau and muon mass should be kept at different values. In WHIZARD from version v2.2.0 on, there is the possibility to define such inclusive process containers. The infrastructure for this feature is realized via so-called process components: processes are allowed to contain several process components. Those components need not be provided by the same matrix element generator, e.g. internal matrix elements, O'Mega matrix elements, external matrix element (e.g. from a one-loop program, OLP) can be mixed. The very same infrastructure can also be used for next-to-leading order (NLO) calculations, containing the born with real emission, possible subtraction terms to make the several components infrared- and collinear finite, as well as the virtual corrections.

Here, we want to discuss the use for inclusive particle samples. There are several options, the simplest of which to add up different final states by just using the + operator in SINDARIN, e.g.:

```
process multi_comp = e1, E1 => (e2, E2) + (e3, E3) + (A, A)
```

The brackets are not only used for a better grouping of the expressions, they are not mandatory for WHIZARD to interpret the sum correctly. When integrating, WHIZARD tells you that this a process with three different components:

```
| Initializing integration for process multi_comp_1_p1:
| ---------------------------------------------------------------------------
| Process [scattering]: 'multi_comp'
|   Library name  = 'default_lib'
|   Process index = 1
|   Process components:
|     1: 'multi_comp_i1':   e-, e+ => m-, m+ [omega]
|     2: 'multi_comp_i2':   e-, e+ => t-, t+ [omega]
|     3: 'multi_comp_i3':   e-, e+ => A, A [omega]
| ---------------------------------------------------------------------------
```

A different phase-space setup is used for each different component. The integration for each different component is performed separately, and displayed on screen. At the end, a sum of all components is shown. All files that depend on the components are being attached an _i<n> where <n> is the number of the process component that appears in the list above: the Fortran code for the matrix element, the .phs file for the phase space parameterization, and the grid files for the VAMP Monte-Carlo integration (or any other integration method). However, there will be only one event file for the inclusive process, into which a mixture of events according to the size of the individual process component cross section enter.

More options are to specify additive lists of particles. WHIZARD then expands the final states according to tensor product algebra:

```
process multi_tensor = e1, E1 => e2 + e3 + A, E2 + E3 + A
```

This gives the same three process components as above, but WHIZARD recognized that e.g. $e^- e^+ \to \mu^- \gamma$ is a vanishing process, hence the numbering is different:

```
| Process component 'multi_tensor_i2': matrix element vanishes
| Process component 'multi_tensor_i3': matrix element vanishes
| Process component 'multi_tensor_i4': matrix element vanishes
| Process component 'multi_tensor_i6': matrix element vanishes
| Process component 'multi_tensor_i7': matrix element vanishes
| Process component 'multi_tensor_i8': matrix element vanishes
| ---------------------------------------------------------------------------
| Process [scattering]: 'multi_tensor'
|   Library name  = 'default_lib'
|   Process index = 1
|   Process components:
|     1: 'multi_tensor_i1':   e-, e+ => m-, m+ [omega]
|     5: 'multi_tensor_i5':   e-, e+ => t-, t+ [omega]
|     9: 'multi_tensor_i9':   e-, e+ => A, A [omega]
| ---------------------------------------------------------------------------
```

Identical copies of the same process that would be created by expanding the tensor product of final states are eliminated and appear only once in the final sum of process components.

Naturally, inclusive process definitions are also available for decays:

```
process multi_dec = Wp => E2 + E3, n2 + n3
```

This yields:

```
| Process component 'multi_dec_i2': matrix element vanishes
| Process component 'multi_dec_i3': matrix element vanishes
| ------------------------------------------------------------------------
| Process [decay]: 'multi_dec'
|   Library name  = 'default_lib'
|   Process index = 2
|   Process components:
|     1: 'multi_dec_i1':   W+ => mu+, numu [omega]
|     4: 'multi_dec_i4':   W+ => tau+, nutau [omega]
| ------------------------------------------------------------------------
```

5.4.5   Compilation

Once processes have been set up, to make them available for integration they have to be compiled. More precisely, the matrix-element generator `O'Mega` (and it works similarly if a different matrix element method is chosen) is called to generate matrix element code, the compiler is called to transform this `Fortran` code into object files, and the linker is called to collect this in a dynamically loadable library. Finally, this library is linked to the program. From version v2.2.0 of `WHIZARD` this is no longer done by system calls of the OS but steered via process library Makefiles. Hence, the user can execute and manipulate those Makefiles in order to manually intervene in the particular steps, if he/she wants to do so.

All this is done automatically when an `integrate`, `unstable`, or `simulate` command is encountered for the first time. You may also force compilation explicitly by the command

```
compile
```

which performs all steps as listed above, including loading the generated library.

The `Fortran` part of the compilation will be done using the `Fortran` compiler specified by the string variable `$fc` and the compiler flags specified as `$fcflags`. The default settings are those that have been used for compiling `WHIZARD` itself during installation. For library compatibility, you should stick to the compiler. The flags may be set differently. They are applied in the compilation and loading steps, and they are processed by `libtool`, so `libtool`-specific flags can also be given.

`WHIZARD` has some precautions against unnecessary repetitions. Hence, when a `compile` command is executed (explicitly, or implicitly by the first integration), the program checks first whether the library is already loaded, and whether source code already exists for the requested processes. If yes, this code is used and no calls to `O'Mega` (or another matrix element method) or to the compiler are issued. Otherwise, it will detect any modification to the process configuration and regenerate the matrix element or recompile accordingly. Thus, a `SINDARIN` script can be

executed repeatedly without rebuilding everything from scratch, and you can safely add more processes to a script in a subsequent run without having to worry about the processes that have already been treated.

This default behavior can be changed. By setting

```
?rebuild_library = true
```

code will be re-generated and re-compiled even if WHIZARD would think that this is unncessary. The same effect is achieved by calling WHIZARD with a command-line switch,

```
/home/user$ whizard --rebuild_library
```

There are further `rebuild` switches which are described below. If everything is to be rebuilt, you can set a master switch `?rebuild` or the command line option `--rebuild`. The latter can be abbreviated as a short command-line option:

```
/home/user$ whizard -r
```

Setting this switch is always a good idea when starting a new project, just in case some old files clutter the working directory. When re-running the same script, possibly modified, the `-r` switch should be omitted, so the existing files can be reused.

5.4.6   Process libraries

Processes are collected in *libraries*. A script may use more than one library, although for most applications a single library will probably be sufficient.

The default library is `default_lib`. If you do not specify anything else, the processes you compile will be collected by a driver file `default_lib.f90` which is compiled together with the process code and combined as a libtool archive `default_lib.la`, which is dynamically linked to the running WHIZARD process.

Once in a while, you work on several projects at once, and you didn't care about opening a new working directory for each. If the `-r` option is given, a new run will erase the existing library, which may contain processes needed for the other project. You could omit `-r`, so all processes will be collected in the same library (this does not hurt), but you may wish to cleanly separate the projects. In that case, you should open a separate library for each project.

Again, there are two possibilities. You may start the script with the specification

```
library = "my_lhc_proc"
```

to open a library `my_lhc_proc` in place of the default library. Repeating the command with different arguments, you may introduce several libraries in the script. The active library is always the one specified last. It is possible to issue this command locally, so a particular process goes into its own library.

Alternatively, you may call WHIZARD with the option

```
/home/user$ whizard --library=my_lhc_proc
```

If several libraries are open simultaneously, the `compile` command will compile all libraries that the script has referenced so far. If this is not intended, you may give the command an argument,

```
compile ("my_lhc_proc", "my_other_proc")
```

to compile only a specific subset.

The command

```
show (library)
```

will display the contents of the actually loaded library together with a status code which indicates the status of the library and the processes within.

5.4.7 Stand-alone WHIZARD with precompiled processes

Once you have set up a process library, it is straightforward to make a special stand-alone WHIZARD executable which will have this library preloaded on startup. This is a matter of convenience, and it is also useful if you need a statically linked executable for reasons of profiling, batch processing, etc.

For this task, there is a variant of the `compile` command:

```
compile as "my_whizard" ()
```

which produces an executable `my_whizard`. You can omit the library argument if you simply want to include everything. (Note that this command will *not* load a library into the current process, it is intended for creating a separate program that will be started independently.)

As an example, the script

```
process proc1 = e1, E1 => e1, E1
process proc2 = e1, E1 => e2, E2
process proc3 = e1, E1 => e3, E3
compile as "whizard-leptons" ()
```

will make a new executable program `whizard-leptons`. This program behaves completely identical to vanilla WHIZARD, except for the fact that the processes `proc1`, `proc2`, and `proc3` are available without configuring them or loading any library.

5.5 Beams

Before processes can be integrated and simulated, the program has to know about the collider properties. They can be specified by the `beams` statement.

In the command script, it is irrelevant whether a `beams` statement comes before or after process specification. The `integrate` or `simulate` commands will use the `beams` statement that was issued last.

5.5.1   Beam setup

If the beams have no special properties, and the colliding particles are the incoming particles in the process themselves, there is no need for a `beams` statement at all. You only *must* specify the center-of-momentum energy of the collider by setting the value of $\sqrt{s}$, for instance

```
sqrts = 14 TeV
```

The `beams` statement comes into play if

- the beams have nontrivial structure, e.g., parton structure in hadron collision or photon radiation in lepton collision, or

- the beams have non-standard properties: polarization, asymmetry, crossing angle.

Note that some of the abovementioned beam properties had not yet been reimplemented in the `WHIZARD2` release series. From version v2.2.0 on all options of the legacy series `WHIZARD1` are available again. From version v2.1 to version v2.2 of `WHIZARD` there has also been a change in possible options to the `beams` statement: in the early versions of `WHIZARD2` (v2.0/v2.1), local options could be specified within the beam settings, e.g. `beams = p, p  sqrts = 14 TeV => pdf_builtin`. These possibility has been abandoned from version v2.2 on, and the `beams` command does not allow for *any* optional arguments any more.

Hence, beam parameters can – with the exception of the specification of structure functions – be specified only globally:

```
sqrts = 14 TeV
beams = p, p => lhapdf
```

It does not make any difference whether the value of `sqrts` is set before or after the `beams` statement, the last value found before an `integrate` or `simulate` is the relevant one. This in particularly allows to specify the beam structure, and then after that perform a loop or scan over beam energies, beam parameters, or structure function settings.

The `beams` statement also applies to particle decay processes, where there is only a single beam. Here, it is usually redundant because no structure functions are possible, and the energy is fixed to the decaying particle's mass. However, it is needed for computing polarized decay, e.g.

```
beams = Z
beams_pol_density = @(0)
```

where for a boson at rest, the polarization axis is defined to be the z axis.

Beam polarization is described in detail below in Sec. 5.6.

Note also that future versions of `WHIZARD` might give support for single-beam events, where structure functions for single particles indeed do make sense.

In the following sections we list the available options for structure functions or spectra inside `WHIZARD` and explain their usage. More about the physics of the implemented structure functions can be found in Chap. 9.

5.5.2 Asymmetric beams and Crossing angles

WHIZARD not only allows symmetric beam collisions, but basically arbitrary collider setups. In the case there are two different beam energies, the command

```
beams_momentum = <beam_mom1>, <beam_mom2>
```

allows to specify the momentum (or as well energies for massless particles) for the beams. Note that for scattering processes both values for the beams must be present. So the following to setups for 14 TeV LHC proton-proton collisions are equivalent:

```
beams = p, p => pdf_builtin
sqrts = 14 TeV
```

and

```
beams = p, p => pdf_builtin
beams_momentum = 7 TeV, 7 TeV
```

Asymmetric setups can be set by using different values for the two beam momenta, e.g. in a HERA setup:

```
beams = e, p => none, pdf_builtin beams_momentum = 27.5 GeV, 920 GeV
```

or for the BELLE experiment at the KEKB accelerator:

```
beams = e1, E1 beams_momentum = 8 GeV, 3.5 GeV
```

WHIZARD lets you know about the beam structure and calculates for you that the center of mass energy corresponds to 10.58 GeV:

```
| Beam structure: e-, e+
|   momentum = 8.000000000000E+00, 3.500000000000E+00
| Beam data (collision):
|   e-  (mass = 5.1099700E-04 GeV)
|   e+  (mass = 5.1099700E-04 GeV)
|   sqrts = 1.058300530253E+01 GeV
| Beam structure: lab and c.m. frame differ
```

It is also possible to specify beams for decaying particles, where `beams_momentum` then only has a single argument, e.g.:

```
process zee = Z => "e-", "e+"
beams = Z
beams_momentum = 500 GeV
simulate (zee) { n_events = 100 }
```

This would correspond to a beam of Z bosons with a momentum of 500 GeV. Note, however, that WHIZARD will always do the integration of the particle width in the particle's rest frame, while the moving beam is then only taken into account for the frame of reference for the simulation.

Further options then simply having different beam energies describe a non-vanishing between the two incoming beams. Such concepts are quite common e.g. for linear colliders to improve the beam properties in the collimation region at the beam interaction points. Such crossing angles can be specified in the beam setup, too, using the beams_theta command:

```
beams = e1, E1
beams_momentum = 500 GeV, 500 GeV
beams_theta = 0, 10 degree
```

It is important that when a crossing angle is being specified, and the collision system consequently never is the center-of-momentum system, the beam momenta have to explicitly set. Besides a planar crossing angle, one is even able to rotate an azimuthal distance:

```
beams = e1, E1
beams_momentum = 500 GeV, 500 GeV
beams_theta = 0, 10 degree
beams_phi = 0, 45 degree
```

5.5.3   LHAPDF

For incoming hadron beams, the beams statement specifies which structure functions are used. The simplest example is the study of parton-parton scattering processes at a hadron-hadron collider such as LHC or Tevatron. The LHAPDF structure function set is selected by a syntax similar to the process setup, namely the example already shown above:

```
beams = p, p => lhapdf
```

Note that there are slight differences in using the LHAPDF release series 6 and the older Fortran LHAPDF release series 5, at least concerning the naming conventions for the PDF sets [4]. The above beams statement selects a default LHAPDF structure-function set for both proton beams (which is the CT10 central set for LHAPDF 6, and cteq6ll.LHpdf central set for LHAPDF5). The structure function will apply for all quarks, antiquarks, and the gluon as far as supported by the particular LHAPDF set. Choosing a different set is done by adding the filename as a local option to the lhapdf keyword:

```
beams = p, p => lhapdf
$lhapdf_file = "MSTW2008lo68cl"
```

for the actual LHAPDF 6 series, and

```
beams = p, p => lhapdf
$lhapdf_file = "MSTW2008lo68cl.LHgrid"
```

---

[4]Until WHIZARD version 2.2.1 including, only the LHAPDF series 5 was supported, while from version 2.2.2 on also the LHAPDF release series 6 has been supported.

for `LHAPDF5`.Similarly, a member within the set is selected by the numeric variable `lhapdf_member` (for both release series of `LHAPDF`).

In some cases, different structure functions have to be chosen for the two beams. For instance, we may look at *ep* collisions:

```
beams = "e-", p => none, lhapdf
```

Here, there is a list of two independent structure functions (each with its own option set, if applicable) which applies to the two beams.

Another mixed case is *pγ* collisions, where the photon is to be resolved as a hadron. The simple assignment

```
beams = p, gamma => lhapdf, lhapdf_photon
```

will be understood as follows: `WHIZARD` selects the appropriate default structure functions (here we are using `LHAPDF` 5 as an example as the support of photon and pion PDFs in `LHAPDF` 6 has been dropped), `cteq6ll.LHpdf` for the proton and `GSG960.LHgrid` for the photon. The photon case has an additional integer-valued parameter `lhapdf_photon_scheme`. (There are also pion structure functions available.) For modifying the default, you have to specify separate structure functions

```
beams = p, gamma => lhapdf, lhapdf_photon
$lhapdf_file = ...
$lhapdf_photon_file = ...
```

Finally, the scattering of elementary photons on partons is described by

```
beams = p, gamma => lhapdf, none
```

Note that for `LHAPDF` version 5.7.1 or higher and for PDF sets which support it, photons can be used as partons.

There is one more option for the `LHAPDF` PDFs, namely to specify the path where the `LHAPDF` PDF sets reside: this is done with the string variable `$lhapdf_dir = "<path-to-lhapdf>"`. Usually, it is not necessary to set this because `WHIZARD` detects this path via the `lhapdf-config` script during configuration, but in the case paths have been moved, or special files/special locations are to be used, the user can specify this location explicitly.

5.5.4 Built-in PDFs

In addition to the possibility of linking against `LHAPDF`, `WHIZARD` comes with a couple of built-in PDFs which are selected via the `pdf_builtin` keyword

```
beams = p, p => pdf_builtin
```

The default PDF set is CTEQ6L, but other choices are also available by setting the string variable `$pdf_builtin_set` to an appropiate value. E.g, modifying the above setup to

```
beams = p, p => pdf_builtin
$pdf_builtin_set = "mrst2004qedp"
```

Tag	Name	Notes	References
cteq6l	CTEQ6L	—	[65]
cteq6l1	CTEQ6L1	—	[65]
cteq6d	CTEQ6D	—	[65]
cteq6m	CTEQ6M	—	[65]
mrst2004qedp	MRST2004QED (proton)	includes photon	[66]
mrst2004qedn	MRST2004QED (neutron)	includes photon	[66]
mstw2008lo	MSTW2008LO	—	[67]
mstw2008nlo	MSTW2008NLO	—	[67]
mstw2008nnlo	MSTW2008NNLO	—	[67]
ct10	CT10	—	[68]
CJ12_max	CJ12_max	—	[69]
CJ12_mid	CJ12_mid	—	[69]
CJ12_min	CJ12_min	—	[69]
CJ15LO	CJ15LO	—	[70]
CJ15NLO	CJ15NLO	—	[70]
mmht2014lo	MMHT2014LO	—	[71]
mmht2014nlo	MMHT2014NLO	—	[71]
mmht2014nnlo	MMHT2014NNLO	—	[71]
CT14LL	CT14LLO	—	[72]
CT14L	CT14LO	—	[72]
CT14N	CT1414NLO	—	[72]
CT14NN	CT14NNLO	—	[72]

Table 5.3: *All PDF sets available as builtin sets. The two MRST2004QED sets also contain a photon.*

would select the proton PDF from the MRST2004QED set. A list of all currently available PDFs can be found in Table 5.3.

The two MRST2004QED sets also contain the photon as a parton, which can be used in the same way as for `LHAPDF` from v5.7.1 on. Note, however, that there is no builtin PDF that contains a photon structure function. There is a `beams` structure function specifier `pdf_builtin_photon`, but at the moment this throws an error. It just has been implemented for the case that in future versions of `WHIZARD` a photon structure function might be included.

Note that in general only the data sets for the central values of the different PDFs ship with `WHIZARD`. Using the error sets is possible, i.e. it is supported in the syntax of the code, but you have to download the corresponding data sets from the web pages of the PDF fitting collaborations.

5.5.5 HOPPET b parton matching

When the `HOPPET` tool [73] for hadron-collider PDF structure functions and their manipulations are correctly linked to `WHIZARD`, it can be used for advanced calculations and simulations of hadron collider physics. Its main usage inside `WHIZARD` is for matching schemes between 4-flavor and 5-flavor schemes in b-parton initiated processes at hadron colliders. Note that in versions 2.2.0 and 2.2.1 it only worked together with `LHAPDF` version 5, while with the `LHAPDF` version 6 interface from version 2.2.2 on it can be used also with the modern version of PDFs from `LHAPDF`. Furthermore, from version 2.2.2, the `HOPPET` b parton matching also works for the builtin PDFs.

It depends on the corresponding process and the energy scales involved whether it is a better description to use the $g \to b\bar{b}$ splitting from the DGLAP evolution inside the PDF and just take the b parton content of a PDF, e.g. in BSM Higgs production for large $\tan\beta$: $pp \to H$ with a partonic subprocess $b\bar{b} \to H$, or directly take the gluon PDFs and use $pp \to b\bar{b}H$ with a partonic subprocess $gg \to b\bar{b}H$. Elaborate schemes for a proper matching between the two prescriptions have been developed and have been incorporated into the `HOPPET` interface.

Another prime example for using these matching schemes is single top production at hadron colliders. Let us consider the following setup:

```
process proc1 = b, u => t, d
process proc2 = u, b => t, d
process proc3 = g, u => t, d, B          { $restrictions = "2+4 ~ W+" }
process proc4 = u, g => t, d, B          { $restrictions = "1+4 ~ W+" }

beams = p,p => pdf_builtin
sqrts = 14 TeV
?hoppet_b_matching = true

$sample = "single_top_matched"
luminosity = 1 / 1 fbarn
simulate (proc1, proc2, proc3, proc4)
```

The first two processes are single top production from b PDFs, the last two processes contain an explicit $g \to b\bar{b}$ splitting (the restriction, cf. Sec. 5.4.3 has been placed in order to single out the

single top production signal process). PDFs are then chosen from the default builtin PDF (which is `CTEQ6L`), and the `HOPPET` matching routines are switched on by the flag `?hoppet_b_matching`.

5.5.6   Lepton Collider ISR structure functions

Initial state QED radiation off leptons is an important feature at all kinds of lepton colliders: the radiative return to the Z resonance by ISR radiation was in fact the largest higher-order effect for the SLC and LEP I colliders. The soft-collinear and soft photon radiation can indeed be resummed/exponentiated to all orders in perturbation theory [7], while higher orders in hard-collinear photons have to be explicitly calculated order by order [8,9]. WHIZARD has an intrinsic implementation of the lepton ISR structure function that includes all orders of soft and soft-collinear photons as well as up to the third order in hard-collinear photons. It can be switched on by the following statement:

```
beams = e1, E1 => isr
```

As the ISR structure function is a single-beam structure function, this expression is synonymous for

```
beams = e1, E1 => isr, isr
```

The ISR structure function can again be applied to only one of the two beams, e.g. in a HERA-like setup:

```
beams = e1, p => isr, pdf_builtin
```

Their are several options for the lepton-collider ISR structure function that are summarized in the following:

Parameter	Default	Meaning
`isr_alpha`	0/intrinsic	value of α_{QED} for ISR
`isr_order`	3	max. order of hard-collinear photon emission
`isr_mass`	0/intrinsic	mass of the radiating lepton
`isr_q_max`	$0/\sqrt{s}$	upper cutoff for ISR
`?isr_recoil`	false	flag to switch on recoil/p_T (*deprecated*)
`?isr_keep_energy`	false	recoil flag: conserve energy in splitting (*deprecated*)

The maximal order of the hard-collinear photon emission taken into account by WHIZARD is set by the integer variable `isr_order`; the default is the maximally available order of three. With the variable `isr_alpha`, the value of the QED coupling constant α_{QED} used in the ISR structure function can be set. The default is taken from the active physics model. The mass of the radiating lepton (in most cases the electron) is set by `isr_mass`; again the default is taken from the active physics model. Furthermore, the upper integration border for the ISR structure function which acts roughly as an upper hardness cutoff for the emitted photons, can be set through `isr_q_max`; if not set, the collider energy (possibly after beamstrahlung, cf. Sec. 5.5.7) $\sqrt{s}$ (or $\sqrt{\hat{s}}$) is taken. Note that WHIZARD accounts for the exclusive effects of ISR radiation

at the moment by a single (hard, resolved) photon in the event; a more realistic treatment of exclusive ISR photons in simulation is foreseen for a future version.

While the ISR structure function is evaluated in the collinear limit, it is possible to generate transverse momentum for both the radiated photons and the recoiling partonic system. We recommend to stick to the collinear approximation for the integration step. Integration cuts should be set up such that they do not significantly depend on photon transverse momentum. In a subsequent simulation step, it is possible to transform the events with collinear ISR radiation into more realistic events with non-collinear radiation. To this end, `WHIZARD` provides a separate ISR photon handler which can be activated in the simulation step. The algorithm operates on the partonic event: it takes the radiated photons and the partons entering the hard process, and applies a p_T distribution to those particles and their interaction products, i.e., all outgoing particles. Cuts that depend on photon p_T may be applied to the modified events. For details on the ISR photon handler, cf. Sec. 10.4.

The flag `?isr_recoil` switches on p_T recoil of the emitting lepton against photon radiation during integration; per default it is off. The flag `?isr_keep_energy` controls the mode of on-shell projection for the splitting process with p_T. Note that this feature is kept for backwards compatibility, but should not be used for new simulations. The reason is as follows: For a fraction of events, p_T will become significant, and (i) energy/momentum non-conservation, applied to both beams separately, can lead to unexpected and unphysical effects, and (ii) the modified momenta enter the hard process, so the collinear approximation used in the ISR structure function computation does not hold.

5.5.7 Lepton Collider Beamstrahlung

At linear lepton colliders, the macroscopic electromagnetic interaction of the bunches leads to a distortion of the spectrum of the bunches that is important for an exact simulation of the beam spectrum. There are several methods to account for these effects. The most important tool to simulate classical beam-beam interactions in lepton-collider physics is `GuineaPig++` [10,11,12]. A direct interface between this tool `GuineaPig++` and `WHIZARD` had existed as an inofficial add-on to the legacy branch `WHIZARD1`, but is no longer applicable in `WHIZARD2`. A `WHIZARD`-internal interface is foreseen for the very near future, most probably within this v2.2 release. Other options are to use parameterizations of the beam spectrum that have been included in the package `CIRCE1` [6] which has been interfaced to `WHIZARD` since version v1.20 and been included in the `WHIZARD2` release series. Another option is to generate a beam spectrum externally and then read it in as an ASCII data file, cf. Sec. 5.5.8. More about this can be found in a dedicated section on lepton collider spectra, Sec. 10.3.

In this section, we discuss the usage of beamstrahlung spectra by means of the `CIRCE1` package. The beamstrahlung spectra are true spectra, so they have to be applied to pairs of beams, and an application to only one beam is meaningless. They are switched on by this `beams` statement including structure functions:

```
beams = e1, E1 => circe1
```

It is important to note that the parameterization of the beamstrahlung spectra within `CIRCE1` contain also processes where $e \to \gamma$ conversions have been taking place, i.e. also hard processes

with one (or two) initial photons can be simulated with beamstrahlung switched on. In that case, the explicit photon flags, `?circe1_photon1` and `?circe1_photon2`, for the two beams have to be properly set, e.g. (ordering in the final state does not play a role):

```
process proc1 = A, e1 => A, e1
sqrts = 500 GeV
beams = e1, E1 => circe1
?circe1_photon1 = true
integrate (proc1)

process proc2 = e1, A => A, e1
sqrts = 1000 GeV
beams = e1, A => circe1
?circe1_photon2 = true
```

or

```
process proc1 = A, A => Wp, Wm
sqrts = 200 GeV
beams = e1, E1 => circe1
?circe1_photon1 = true
?circe1_photon2 = true
?circe1_generate = false
```

In all cases (one or both beams with photon conversion) the beam spectrum applies to both beams simultaneously.

In the last example ($\gamma\gamma \to W^+W^-$) the default `CIRCE1` generator mode was turned off by unsetting `?circe1_generate`. In the other examples this flag is set, by default. For standard use cases, `CIRCE1` implements a beam-event generator inside the `WHIZARD` generator, which provides beam-event samples with correctly distributed probability. For electrons, the beamstrahlung spectrum sharply peaks near maximum energy. This distribution is most efficiently handled by the generator mode. By contrast, in the $\gamma\gamma$ mode, the beam-event c.m. energy is concentrated at low values. For final states with low invariant mass, which are typically produced by beamstrahlung photons, the generator mode is appropriate. However, the W^+W^- system requires substantial energy, and such events will be very rare in the beam-event sample. Switching off the `CIRCE1` generator mode solves this problem.

This is an overview over all options and flags for the `CIRCE1` setup for lepton collider beamstrahlung:

Parameter	Default	Meaning
`?circe1_photon1`	`false`	$e \to \gamma$ conversion for beam 1
`?circe1_photon2`	`false`	$e \to \gamma$ conversion for beam 2
`circe1_sqrts`	$\sqrt{s}$	collider energy for the beam spectrum
`?circe1_generate`	`true`	flag for the `CIRCE1` generator mode
`?circe1_map`	`true`	flag to apply special phase-space mapping
`circe1_mapping_slope`	`2.`	value of PS mapping exponent
`circe1_eps`	`1E-5`	parameter for mapping of spectrum peak position
`circe1_ver`	`0`	internal version of `CIRCE1` package
`circe1_rev`	`0`/most recent	internal revision of `CIRCE1`
`$circe1_acc`	`SBAND`	accelerator type
`circe1_chat`	`0`	chattiness/verbosity of `CIRCE1`

The collider energy relevant for the beamstrahlung spectrum is set by `circe1_sqrts`. As a default, this is always the value of `sqrts` set in the `SINDARIN` script. However, sometimes these values do not match, e.g. the user wants to simulate $t\bar{t}h$ at `sqrts = 550 GeV`, but the only available beam spectrum is for 500 GeV. In that case, `circe1_sqrts = 500 GeV` has to be set to use the closest possible available beam spectrum.

As mentioned in the discussion of the examples above, in `CIRCE1` there are two options to use the beam spectra for beamstrahlung: intrinsic semi-analytic approximation formulae for the spectra, or a Monte-Carlo sampling of the sampling. The second possibility always give a better description of the spectra, and is the default for `WHIZARD`. It can, however, be switched off by setting the flag `?circe1_generate` to `false`.

As the beamstrahlung spectra are sharply peaked at the collider energy, but still having long tails, a mapping of the spectra for an efficient phase-space sampling is almost mandatory. This is the default in `WHIZARD`, which can be changed by the flag `?circe1_map`. Also, the default exponent for the mapping can be changed from its default value `2.` with the variable `circe1_mapping_slope`. It is important to efficiently sample the peak position of the spectrum; the effective ratio of the peak to the whole sampling interval can be set by the parameter `circe1_eps`. The integer parameter `circe1_chat` sets the chattiness or verbosity of the `CIRCE1` package, i.e. how many messages and warnings from the beamstrahlung generation/sampling will be issued.

The actual internal version and revision of the `CIRCE1` package are set by the two integer parameters `circe1_ver` and `circe1_rev`. The default is in any case always the newest version and revision, while older versions are still kept for backwards compatibility and regression testing.

Finally, the geometry and design of the accelerator type is set with the string variable `$circe1_acc`: it contains the possible options for the old `"SBAND"` and `"XBAND"` setups, as well as the `"TESLA"` and JLC/NLC SLAC design `"JLCNLC"`. The setups for the most important energies of the ILC as they are summarized in the ILC TDR [13,14,15,16] are available as `ILC`. Beam spectra for the CLIC [18,19,20] linear collider are much more demanding to correctly simulate (due to the drive beam concept; only the low-energy modes where the drive beam is off can be simulated with the same setup as the abovementioned machines). Their setup will be

supported soon in one of the upcoming `WHIZARD` versions within the `CIRCE2` package.

An example of how to generate beamstrahlung spectra with the help of the package `CIRCE2` (that is also a part of `WHIZARD`) is this:

```
process eemm = e1, E1 => e2, E2
sqrts = 500 GeV
beams = e1, E1 => circe2
$circe2_file = "ilc500.circe"
$circe2_design = "ILC"
?circe_polarized = false
```

Here, the ILC design is used for a beamstrahlung spectrum at 500 GeV nominal energy, with polarization averaged (hence, the setting of polarization to `false`). A list of all available options can be found in Sec. 5.5.13.

More technical details about the simulation of beamstrahlung spectra see the documented source code of the `CIRCE1` package, as well as Chap. 9. In the next section, we discuss how to read in beam spectra from external files.

5.5.8   Beam events

As mentioned in the previous section, beamstrahlung is one of the crucial ingredients for a realistic simulation of linear lepton colliders. One option is to take a pre-generated beam spectrum for such a machine, and make it available for simulation within `WHIZARD` as an external ASCII data file. Such files basically contain only pairs of energy fractions of the nominal collider energy $\sqrt{s}$ (x values). In `WHIZARD` they can be used in simulation with the following `beams` statement:

```
beams = e1, E1 => beam_events
$beam_events_file = "<beam_spectrum_file>"
```

Note that beam spectra must always be pair spectra, i.e. they are automatically applied to both beam simultaneously. Beam spectra via external files are expected to reside in the current working directory. Alternatively, `WHIZARD` searches for them in the install directory of `WHIZARD` in `share/beam-sim`. There you can find an example file, `uniform_spread_2.5%.dat` for such a beam spectrum. The only possible parameter that can be set is the flag `?beam_events_warn_eof` whose default is `true`. This triggers the issuing of a warning when the end of file of an external beam spectrum file is reached. In such a case, `WHIZARD` starts to reuse the same file again from the beginning. If the available data points in the beam events file are not big enough, this could result in an insufficient sampling of the beam spectrum.

5.5.9   Gaussian beam-energy spread

Real beams have a small energy spread. If beamstrahlung is small, the spread may be approximately described as Gaussian. As a replacement for the full simulation that underlies `CIRCE2` spectra, it is possible to impose a Gaussian distributed beam energy, separately for each beam.

```
beams = e1, E1 => gaussian
gaussian_spread1 = 0.1\%
gaussian_spread2 = 0.2\%
```

(Note that the % sign means multiplication by 0.01, as it should.) The spread values are defined as the σ value of the Gaussian distribution, i.e., $2/3$ of the events are within $\pm 1\sigma$ for each beam, respectively.

5.5.10 Equivalent photon approximation

The equivalent photon approximation (EPA) uses an on-shell approximation for the $e \to e\gamma$ collinear splitting to allow the simulation of photon-induced backgrounds in lepton collider physics. The original concept is that of the Weizsäcker-Williams approximation [21,22,23]. This is a single-beam structure function that can be applied to both beams, or also to one beam only. Usually, there are some simplifications being made in the derivation. The formula which is implemented here and seems to be the best for the QCD background for low-p_T hadrons, corresponds to Eq. (6.17) of Ref. [23]. As this reference already found, this leads to an "overshooting" of accuracy, and especially in the high-x (high-energy) region to wrong results. This formula corresponds to

$$f(x) = \frac{\alpha}{\pi} \frac{1}{x} \left[\left(\bar{x} + \frac{x^2}{2} \right) \log \frac{Q_{\max}^2}{Q_{\min}^2} - \left(1 - \frac{x}{2} \right)^2 \log \frac{x^2 + \frac{Q_{\max}^2}{E^2}}{x^2 + \frac{Q_{\min}^2}{E^2}} - \frac{m_e^2 x^2}{Q_{\min}^2} \left(1 - \frac{Q_{\min}^2}{Q_{\max}^2} \right) \right] \qquad . \quad (5.1)$$

Here, x is the ratio of the photon energy (called frequency ω in [23] over the original electron (or positron) beam energy E. The energy of the electron (or positron) after the splitting is given by $\bar{x} = 1 - x$.

The simplified version is the one that corresponds to many publications about the EPA during SLC and LEP times, and corresponds to the q^2 integration of Eq. (6.16e) in [23], where q^2 is the virtuality or momentum transfer of the photon in the EPA:

$$f(x) = \frac{\alpha}{\pi} \frac{1}{x} \left[\left(\bar{x} + \frac{x^2}{2} \right) \log \frac{Q_{\max}^2}{Q_{\min}^2} - \frac{m_e^2 x^2}{Q_{\min}^2} \left(1 - \frac{Q_{\min}^2}{Q_{\max}^2} \right) \right] \qquad . \quad (5.2)$$

While Eq. (5.1) is supposed to be the better choice for simulating hadronic background like low-p_T hadrons and should be applied for the low-x region of the EPA, Eq. (5.2) seems better suited for high-x simulations like the photoproduction of BSM resonances etc. Note that the first term in Eqs. (5.1) and (5.2) is the standard Altarelli-Parisi QED splitting function of electron, $P_{e \to e\gamma}(x) \propto 1 + (1-x)^2$, while the last term in both equations is the default power correction.

The two parameters $Q_{\max}^2$ and $Q_{\min}^2$ are the integration boundaries of the photon virtuality integration. Usually, they are given by the kinematic limits:

$$Q_{\min}^2 = \frac{m_e^2 x^2}{\bar{x}} \qquad\qquad Q_{\max}^2 = 4E^2 \bar{x} = s\bar{x} \qquad . \quad (5.3)$$

For low-p_T hadron simulations, it is not a good idea to take the kinematic limit as an upper limit, but one should cut the simulation off at a hadronic scale like e.g. a multiple of the ρ mass.

The user can switch between the two different options using the setting

```
$epa_mode = "default"
```

or

```
$epa_mode = "Budnev_617"
```

for Eq. (5.1), while Eq. (5.2) can be chosen with

```
$epa_mode = "Budnev_616e"
```

Note that a thorough study for high-energy e^+e^- colliders regarding the suitability of different EPA options is still lacking.

For testing purposes also three more variants or simplifications of Eq. (5.2) are implemented: the first, steered by `$epa_mode = log_power` uses simply $Q^2_{\max} = s$. This is also the case for the two other method. But the switch `$epa_mode = log_simple` uses just `epa_mass` (cf. below) as $Q^2_{\min}$. The final simplification is to drop the power correction, which can be chosen with `$epa_mode = log`. This corresponds to the simple formula:

$$f(x) = \frac{\alpha}{2\pi} \frac{1}{x} \, \log \frac{s}{m^2} \qquad . \tag{5.4}$$

Examples for the application of the EPA in `WHIZARD` are:

```
beams = e1, E1 => epa
```

or for a single beam:

```
beams = e1, p => epa, pdf_builtin
```

The last process allows the reaction of (quasi-) on-shell photons with protons.

In the following, we collect the parameters and flags that can be adjusted when using the EPA inside `WHIZARD`:

Parameter	Default	Meaning
`epa_alpha`	0/intrinsic	value of α_{QED} for EPA
`epa_x_min`	0.	soft photon cutoff in x (mandatory)
`epa_q_min`	0.	minimal γ momentum transfer
`epa_mass`	0/intrinsic	mass of the radiating fermion (mandatory)
`epa_q_max`	$0/\sqrt{s}$	upper cutoff for EPA
`?epa_recoil`	`false`	flag to switch on recoil/p_T
`?epa_keep_energy`	`false`	recoil flag to conserve energy in splitting

The adjustable parameters are partially similar to the parameters in the QED initial-state radiation (ISR), cf. Sec. 5.5.6: the parameter `epa_alpha` sets the value of the electromagnetic coupling constant, α_{QED} used in the EPA structure function. If not set, this is taken from the value inside the active physics model. The same is true for the mass of the particle that radiates the photon of the hard interaction, which can be reset by the user with the variable `epa_mass`. There are two dimensionful scale parameters, the minimal momentum transfer to the

photon, `epa_q_min`, which must not be zero, and the upper momentum-transfer cutoff for the EPA structure function, `epa_q_max`. The default for the latter value is the collider energy, $\sqrt{s}$, or the energy reduced by another structure function like e.g. beamstrahlung, $\sqrt{\hat{s}}$. Furthermore, there is a soft-photon regulator for the splitting function in x space, `epa_x_min`, which also has to be explicitly set different from zero. Hence, a minimal viable scenario that will be accepted by `WHIZARD` looks like this:

```
beams = e1, E1 => epa
epa_q_min = 5 GeV
epa_x_min = 0.01
```

Finally, like the ISR case in Sec. 5.5.6, there is a flag to consider the recoil of the photon against the radiating electron by setting `?epa_recoil` to `true` (default: `false`).

Though in principle processes like $e^+e^- \to e^+e^-\gamma\gamma$ where the two photons have been created almost collinearly and then initiate a hard process could be described by exact matrix elements and exact kinematics. However, the numerical stability in the very far collinear kinematics is rather challenging, such that the use of the EPA is very often an acceptable trade-off between quality of the description on the one hand and numerical stability and speed on the other hand.

In the case, the EPA is set after a second structure function like a hadron collider PDF, there is a flavor summation over the quark constituents inside the proton, which are then the radiating fermions for the EPA. Here, the masses of all fermions have to be identical.

More about the physics of the equivalent photon approximation can be found in Chap. 9.

5.5.11 Effective W approximation

An approach similar to the equivalent photon approximation (EPA) discussed in the previous section Sec. 5.5.10, is the usage of a collinear splitting function for the radiation of massive electroweak vector bosons W/Z, the effective W approximation (EWA). It has been developed for the description of high-energy weak vector-boson fusion and scattering processes at hadron colliders, particularly the Superconducting Super-Collider (SSC). This was at a time when the simulation of $2 \to 4$ processes war still very challenging and $2 \to 6$ processes almost impossible, such that this approximation was the only viable solution for the simulation of processes like $pp \to jjVV$ and subsequent decays of the bosons $V \equiv W, Z$.

Unlike the EPA, the EWA is much more involved as the structure functions do depend on the isospin of the radiating fermions, and are also different for transversal and longitudinal polarizations. Also, a truely collinear kinematics is never possible due to the finite W and Z boson masses, which start becoming more and more negligible for energies larger than the nominal LHC energy of 14 TeV.

Though in principle all processes for which the EWA might be applicable are technically feasible in `WHIZARD` to be generated also via full matrix elements, the EWA has been implemented in `WHIZARD` for testing purposes, backwards compatibility and comparison with older simulations. Like the EPA, it is a single-beam structure function that can be applied to one or both beams. We only give an example for both beams here, this is for a 3 TeV CLIC collider:

```
sqrts = 3 TeV
beams = e1, E1 => ewa
```

And this is for LHC or a higher-energy follow-up collider (which also shows the concatenation of the single-beam structure functions, applied to both beams consecutively, cf. Sec. 5.5.14:

```
sqrts = 14 TeV
beams = p, p => pdf_builtin => ewa
```

Again, we list all the options, parameters and flags that can be adapted for the EWA:

Parameter	Default	Meaning
`ewa_x_min`	`0.`	soft W/Z cutoff in x (mandatory)
`ewa_mass`	0/intrinsic	mass of the radiating fermion
`ewa_pt_max`	$0/\sqrt{\hat{s}}$	upper cutoff for EWA
`?ewa_recoil`	`false`	recoil switch
`?ewa_keep_energy`	`false`	energy conservation for recoil in splitting

First of all, all coupling constants are taken from the active physics model as they have to be consistent with electroweak gauge invariance. Like for EPA, there is a soft x cutoff for the $f \to fV$ splitting, `ewa_x_min`, that has to be set different from zero by the user. Again, the mass of the radiating fermion can be set explicitly by the user; and, also again, the masses for the flavor sum of quarks after a PDF as radiators of the electroweak bosons have to be identical. Also for the EWA, there is an upper cutoff for the p_T of the electroweak boson, that can be set via `eta_pt_max`. Indeed, the transversal W/Z structure function is logarithmically divergent in that variable. If it is not set by the user, it is estimated from $\sqrt{s}$ and the splitting kinematics.

For the EWA, there is a flag to switch on a recoil for the electroweak boson against the radiating fermion, `?ewa_recoil`. Note that this is an experimental feature that is not completely tested. In any case, the non-collinear kinematics violates 4-four momentum conservation, so there are two choices: either to conserve the energy (`?ewa_keep_energy = true`) or to conserve 3-momentum (`?ewa_keep_energy = false`). Momentum conservation for the kinematics is the default. This is due to the fact that for energy conservation, there will be a net total momentum in the event including the beam remnants (ISR/EPA/EWA radiated particles) that leeds to unexpected or unphysical features in the energy distributions of the beam remnants recoiling against the rest of the event.

More details about the physics can be found in Chap. 9.

5.5.12   Energy scans using structure functions

In WHIZARD, there is an implementation of a pair spectrum, `energy_scan`, that allows to scan the energy dependence of a cross section without actually scanning over the collider energies. Instead, only a single integration at the upper end of the scan interval over the process with an additional pair spectrum structure function performed. The structure function is chosen in such a way, that the distribution of x values of the energy scan pair spectrum translates in a plot over the energy of the final state in an energy scan from 0 to `sqrts` for the process under consideration.

The simplest example is the $1/s$ fall-off with the Z resonance in $e^+e^- \to \mu^+\mu^-$, where the syntax is very easy:

```
process eemm = e1, E1 => e2, E2
sqrts = 500 GeV
cuts = sqrts_hat > 50
beams = e1, E1 => energy_scan
integrate (eemm)
```

The value of `sqrts = 500 GeV` gives the upper limit for the scan, while the cut effectively lets the scan start at 50 GeV. There are no adjustable parameters for this structure function. How to plot the invariant mass distribution of the final-state muon pair to show the energy scan over the cross section, will be explained in Sec. 5.9.

More details can be found in Chap. 9.

5.5.13 Photon collider spectra

One option that has been discussed as an alternative possibility for a high-energy linear lepton collider is to convert the electron and positron beam via Compton backscattering off intense laser beams into photon beams [24,25,26]. Naturally, due to the production of the photon beams and the inherent electron spectrum, the photon beams have a characteristic spectrum. The simulation of such spectra is possible within WHIZARD by means of the subpackage CIRCE2, which have been mentioned already in Sec. 5.5.7. It allows to give a much more elaborate description of a linear lepton collider environment than CIRCE1 (which, however, is not in all cases necessary, as the ILC beamspectra for electron/positrons can be perfectly well described with CIRCE1).

Here is a typical photon collider setup where we take a photon-initiated process:

```
process aaww = A, A => Wp, Wm

beams = A, A => circe2
$circe2_file = "teslagg_500_polavg.circe"
$circe2_design = "TESLA/GG"
?circe2_polarized = false
```

Here, the photons are the initial states initiating the hard scattering. The structure function is `circe2` which always is a pair spectrum. The list of available options are:

Parameter	Default	Meaning
`?circe2_polarized`	`true`	spectrum respects polarization info
`$circe2_file`	–	name of beam spectrum data file
`$circe2_design`	`"*"`	collider design

The only logical flag `?circe2_polarized` let WHIZARD know whether it should keep polarization information in the beam spectra or average over polarizations. Naturally, because of the Compton backscattering generation of the photons, photon spectra are always polarized. The collider design can be specified by the string variable `$circe2_design`, where the default setting `"*"` corresponds to the default of CIRCE2 (which is the TESLA 500 GeV machine as discussed

in the TESLA Technical Design Report [27,28]). Note that up to now there have not been any setups for a photon collider option for the modern linear collider concepts like ILC and CLIC. The string variable `$circe2_file` then allows to give the name of the file containing the actual beam spectrum; all files that ship with WHIZARD are stored in the directory `circe2/share/data`.

More details about the subpackage CIRCE2 and the physics it covers, can be found in its own manual and the chapter Chap. 9.

5.5.14   Concatenation of several structure functions

As has been shown already in Sec. 5.5.10 and Sec. 5.5.11, it is possible within WHIZARD to concatenate more than one structure function, irrespective of the fact, whether the structure functions are single-beam structure functions or pair spectra. One important thing is whether there is a phase-space mapping for these structure functions. Also, there are some combinations which do not make sense from the physics point of view, for example using lepton-collider ISR for protons, and then afterwards switching on PDFs. Such combinations will be vetoed by WHIZARD, and you will find an error message like (cf. also Sec. 4.3):

```
*******************************************************************************
*******************************************************************************
*** FATAL ERROR: Beam structure: [....] not supported
*******************************************************************************
*******************************************************************************
```

Common examples for the concatenation of structure functions are linear collider applications, where beamstrahlung (macroscopic electromagnetic beam-beam interactions) and electron QED initial-state radiation are both switched on:

```
beams = e1, E1 => circe1 => isr
```

Another possibility is the simulation of photon-induced backgrounds at ILC or CLIC, using beamstrahlung and equivalent photon approximation (EPA):

```
beams = e1, E1 => circe1 => epa
```

or with beam events from a data file:

```
beams = e1, E1 => beam_events => isr
```

In hadron collider physics, parton distribution functions (PDFs) are basically always switched on, while afterwards the user could specify to use the effective W approximation (EWA) to simulate high-energy vector boson scattering:

```
sqrts = 100 TeV
beams = p, p => pdf_builtin => ewa
```

Note that this last case involves a flavor sum over the five active quark (and anti-quark) species u, d, c, s, b in the proton, all of which act as radiators for the electroweak vector bosons in the EWA.

This would be an example with three structure functions:

```
beams = e1, E1 => circe1 => isr => epa
```

5.6 Polarization

5.6.1 Initial state polarization

`WHIZARD` supports polarizing the inital state fully or partially by assigning a nontrivial density matrix in helicity space. Initial state polarization requires a beam setup and is initialized by means of the `beams_pol_density` statement[5]:

```
beams_pol_density = @([<spin entries>]), @([<spin entries>])
```

The command `beams_pol_fraction` gives the degree of polarization of the two beams:

```
beams_pol_fraction = <degree beam 1>, <degree beam 2>
```

Both commands in the form written above apply to scattering processes, where the polarization of both beams must be specified. The `beams_pol_density` and `beams_pol_fraction` are possible with a single beam declaration if a decay process is considered, but only then.

While the syntax for the command `beams_pol_fraction` is pretty obvious, the syntax for the actual specification of the beam polarization is more intricate. We start with the polarization fraction: for each beam there is a real number between zero (unpolarized) and one (complete polarization) that can be specified either as a floating point number like `0.4` or with a percentage: `40 %`. Note that the actual arithmetics is sometimes counterintuitive: 80 % left-handed electron polarization means that 80 % of the electron beam are polarized, 20 % are unpolarized, i.e. 20 % have half left- and half right-handed polarization each. Hence, 90 % of the electron beam is left-handed, 10 % is right-handed.

How does the specification of the polarization work? If there are no entries at all in the polarization constructor, `@()`, the beam is unpolarized, and the spin density matrix is proportional to the unit/identity matrix. Placing entries into the `@()` constructor follows the concept of sparse matrices, i.e. the entries that have been specified will be present, while the rest remains zero. Single numbers do specify entries for that particular helicity on the main diagonal of the spin density matrix, e.g. for an electron `@(-1)` means (100%) left-handed polarization. Different entries are separated by commas: `@(1,-1)` sets the two diagonal entries at positions $(1, 1)$ and $(-1, -1)$ in the density matrix both equal to one. Two remarks are in order already here. First, note that you do not have to worry about the correct normalization of the spin density matrix, `WHIZARD` is taking care of this automatically. Second, in the screen output for the beam data, only those entries of the spin density matrix that have been specified by the user, will be displayed. If a `beams_pol_fraction` statement appears, other components will be non-zero, but might not be shown. E.g. ILC-like, 80 % polarization of the electrons, 30 % positron polarization will be specified like this for left-handed electrons and right-handed positrons:

```
beams = e1, E1
beams_pol_density = @(-1), @(+1)
beams_pol_fraction = 80%, 30%
```

The screen output will be like this:

---

[5]Note that the syntax for the specification of beam polarization has changed from version v2.1 to v2.2 and is incompatible between the two release series. The old syntax `beam_polarization` with its different polarization constructors has been discarded in favor of a unified syntax.

```
| --------------------------------------------------------------------
| Beam structure: e-, e+
|   polarization (beam 1):
|      @(-1: -1: ( 1.000000000000E+00, 0.000000000000E+00))
|   polarization (beam 2):
|      @(+1: +1: ( 1.000000000000E+00, 0.000000000000E+00))
|   polarization degree = 0.8000000, 0.3000000
| Beam data (collision):
|   e-   (mass = 0.0000000E+00 GeV)  polarized
|   e+   (mass = 0.0000000E+00 GeV)  polarized
```

But because of the fraction of unpolarized electrons and positrons, the spin density matrices for electrons and positrons are:

$$\rho(e^-) = \mathrm{diag}\,(0.10, 0.90) \qquad \rho(e^+) = \mathrm{diag}\,(0.65, 0.35) \quad ,$$

respectively. So, in general, only the entries due to the polarized fraction will be displayed on screen. We will come back to more examples below.

Again, the setting of a single entry, e.g. $@(\pm m)$, which always sets the diagonal component $(\pm m, \pm m)$ of the spin density matrix equal to one. Here m can have the following values for the different spins (in parentheses are entries that exist only for massive particles):

Spin j	Particle type	possible m values
0	Scalar boson	0
1/2	Spinor	+1, -1
1	(Massive) Vector boson	+1, (0), -1
3/2	(Massive) Vectorspinor	+2, (+1), (-1), -2
2	(Massive) Tensor	+2, (+1), (0), (-1), -2

Off-diagonal entries that are equal to one (up to the normalization) of the spin-density matrix can be specified simply by the position, namely: $@(m{:}m',\ m'')$. This would result in a spin density matrix with diagonal entry 1 for the position (m'', m''), and an entry of 1 for the off-diagonal position (m, m').

Furthermore, entries in the density matrix different from 1 with a numerical value `<val>` can be specified, separated by another colon: $@(m{:}m'{:}$`<val>`$)$. Here, it does not matter whether m and m' are different or not. For $m = m'$ also diagonal spin density matrix entries different from one can be specified. Note that because spin density matrices have to be Hermitian, only the entry (m, m') has to be set, while the complex conjugate entry at the transposed position (m', m) is set automatically by WHIZARD.

We will give some general density matrices now, and after that a few more definite examples. In the general setups below, we always give the expression for the spin density matrix only for one single beam.

- **Unpolarized:**

$$\mathtt{beams\_pol\_density = @()}$$

This has the same effect as not specifying any polarization at all and is the only constructor available for scalars and fermions declared as left- or right-handed (like the neutrino). Density matrix:

$$\rho = \frac{1}{|m|}\mathbb{I}$$

($|m|$: particle multiplicity which is 2 for massless, $2j + 1$ for massive particles).

- **Circular polarization:**

  beams_pol_density = @($\pm j$)   beams_pol_fraction = f

  A fraction f (parameter range $f \in [0 \ ; \ 1]$) of the particles are in the maximum / minimum helicity eigenstate $\pm j$, the remainder is unpolarized. For spin $\frac{1}{2}$ and massless particles of spin > 0, only the maximal / minimal entries of the density matrix are populated, and the density matrix looks like this:

  $$\rho = \text{diag}\left(\frac{1 \pm f}{2} \ , \ 0 \ , \ \ldots \ , \ 0 \ , \ \frac{1 \mp f}{2}\right)$$

- **Longitudinal polarization (massive):**

  beams_pol_density = @(0)   beams_pol_fraction = f

  We consider massive particles with maximal spin component j, a fraction f of which having longitudinal polarization, the remainder is unpolarized. Longitudinal polarization is (obviously) only available for massive bosons of spin > 0. Again, the parameter range for the fraction is: $f \in [0 \ ; \ 1]$. The density matrix has the form:

  $$\rho = \text{diag}\left(\frac{1 - f}{|m|} \ , \ \ldots \ , \ \frac{1 - f}{|m|} \ , \ \frac{1 + f\,(|m| - 1)}{|m|} \ , \ \frac{1 - f}{|m|} \ , \ \ldots \ , \ \frac{1 - f}{|m|}\right)$$

  ($|m| = 2j + 1$: particle multiplicity)

- **Transverse polarization (along an axis):**

  beams_pol_density = @(j, -j, j:-j:exp(-I*phi))   beams_pol_fraction = f

  This so called transverse polarization is a polarization along an arbitrary direction in the $x - y$ plane, with $\phi = 0$ being the positive x direction and $\phi = 90°$ the positive y direction. Note that the value of phi has either to be set inside the beam polarization expression explicitly or by a statement real phi = *val* degree before. A fraction f of the particles are polarized, the remainder is unpolarized. Note that, although this yields a valid density matrix for all particles with multiplicity > 1 (in which the only the highest and lowest helicity states are populated), it is meaningful only for spin $\frac{1}{2}$ particles and

massless bosons of spin > 0. The range of the parameters are: $f \in [0 \; ; \; 1]$ and $\phi \in \mathbb{R}$. This yields a density matrix:

$$\rho = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \frac{f}{2}\,e^{-i\phi} \\ 0 & 0 & \ddots & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 0 & 0 \\ \frac{f}{2}\,e^{i\phi} & \cdots & \cdots & 0 & 1 \end{pmatrix}$$

(for antiparticles, the matrix is conjugated).

- **Polarization along arbitrary axis** (θ, ϕ)**:**

```
beams_pol_density = @(j:j:1-cos(theta), j:-j:sin(theta)*exp(-I*phi),
            -j:-j:1+cos(theta))          beams_pol_fraction = f
```

This example describes polarization along an arbitrary axis in polar coordinates (polar axis in positive z direction, polar angle θ, azimuthal angle ϕ). A fraction f of the particles are polarized, the remainder is unpolarized. Note that, although axis polarization defines a valid density matrix for all particles with multiplicity > 1, it is meaningful only for particles with spin $\frac{1}{2}$. Valid ranges for the parameters are $f \in [0 \; ; \; 1]$, $\theta \in \mathbb{R}$, $\phi \in \mathbb{R}$. The density matrix then has the form:

$$\rho = \frac{1}{2} \cdot \begin{pmatrix} 1 - f\cos\theta & 0 & \cdots & \cdots & f\sin\theta\,e^{-i\phi} \\ 0 & 0 & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 0 & 0 \\ f\sin\theta\,e^{i\phi} & \cdots & \cdots & 0 & 1 + f\cos\theta \end{pmatrix}$$

- **Diagonal density matrix:**

```
beams_pol_density = @(j:j:h_j, j-1:j-1:h_{j-1}, ..., -j:-j:h_{-j})
```

This defines an arbitrary diagonal density matrix with entries $\rho_{j,j}$, $\ldots$, $\rho_{-j,-j}$.

- **Arbitrary density matrix:**

```
beams_pol_density = @({m : m' : x_{m,m'}}):
```

Here, $\{m : m' : x_{m,m'}\}$ denotes a selection of entries at various positions somewhere in the spin density matrix. WHIZARD will check whether this is a valid spin density matrix, but it does e.g. not have to correspond to a pure state.

The beam polarization statements can be used both globally directly with the `beams` specification, or locally inside the `integrate` or `simulate` command. Some more specific examples are in order to show how initial state polarization works:

- ```
  beams = A, A
  beams_pol_density = @(+1),  @(1, -1, 1:-1:-I)
  ```

  This declares the initial state to be composed of two incoming photons, where the first photon is right-handed, and the second photon has transverse polarization in y direction.

- ```
  beams = A, A
  beams_pol_density = @(+1),  @(1, -1, 1:-1:-1)
  ```

  Same as before, but this time the second photon has transverse polarization in x direction.

- ```
  beams = "W+"
  beams_pol\_density = @(0)
  ```

  This example sets up the decay of a longitudinal vector boson.

- ```
  beams = E1, e1
  scan int hel_ep = (-1, 1) {
     scan int hel_em = (-1, 1) {
        beams_pol_density = @(hel_ep), @(hel_em)
        integrate (eeww)
     }
  }
  integrate (eeww)
  ```

  This example loops over the different positron and electron helicity combinations and calculates the respective integrals. The `beams_pol_density` statement is local to the scan loop(s) and, therefore, the last `integrate` calculates the unpolarized integral.

Although beam polarization should be straightforward to use, some pitfalls exist for the unwary:

- Once `beams_pol_density` is set globally, it persists and is applied every time `beams` is executed (unless it is reset). In particular, this means that code like

  ```
  process wwaa = Wp, Wm => A, A
  process zee = Z => e1, E1

  sqrts = 200 GeV
  beams_pol_density = @(1, -1, 1:-1:-1), @()
  beams = Wp, Wm
  integrate (wwaa)
  beams = Z
  integrate (zee)
  beams_pol_density = @(0)
  ```

  will throw an error, because `WHIZARD` complains that the spin density matrix has the wrong dimensionality for the second (the decay) process. This kind of trap can be avoided be using `beams_pol_density` only locally in `integrate` or `simulate` statements.

- On-the-fly integrations executed by `simulate` use the beam setup found at the point of execution. This implies that any polarization settings you have previously done affect the result of the integration.

- The `unstable` command also requires integrals of the selected decay processes, and will compute them on-the-fly if they are unavailable. Here, a polarized integral is not meaningful at all. Therefore, this command ignores the current `beam` setting and issues a warning if a previous polarized integral is available; this will be discarded.

5.6.2   Final state polarization

Final state polarization is available in WHIZARD in the sense that the polarization of real final state particles can be retained when generating simulated events. In order for the polarization of a particle to be retained, it must be declared as polarized via the `polarized` statement

```
polarized particle [, particle, ...]
```

The effect of `polarized` can be reversed with the `unpolarized` statement which has the same syntax. For example,

```
polarized "W+", "W-", Z
```

will cause the polarization of all final state W and Z bosons to be retained, while

```
unpolarized "W+", "W-", Z
```

will reverse the effect and cause the polarization to be summed over again. Note that `polarized` and `unpolarized` are global statements which cannot be used locally as command arguments and if you use them e.g. in a loop, the effects will persist beyond the loop body. Also, a particle cannot be `polarized` and `unstable` at the same time (this restriction might be loosened in future versions of WHIZARD).

After toggling the polarization flag, the generation of polarized events can be requested by using the `?polarized_events` option of the `simulate` command, e.g.

```
simulate (eeww) { ?polarized_events = true }
```

When `simulate` is run in this mode, helicity information for final state particles that have been toggled as `polarized` is written to the event file(s) (provided that polarization is supported by the selected event file format(s)) and can also be accessed in the analysis by means of the `Hel` observable. For example, an analysis definition like

```
analysis =
  if (all Hel == -1 ["W+"] and all Hel == -1 ["W-"] ) then
    record cta_nn (eval cos (Theta) ["W+"]) endif;
  if (all Hel == -1 ["W+"] and all Hel ==  0 ["W-"] )
    then record cta_nl (eval cos (Theta) ["W+"]) endif
```

can be used to histogram the angular distribution for the production of polarized W pairs (obviously, the example would have to be extended to cover all possible helicity combinations). Note, however, that helicity information is not available in the integration step; therefore, it is not possible to use `Hel` as a cut observable.

While final state polarization is straightforward to use, there is a caveat when used in combination with flavor products. If a particle in a flavor product is defined as `polarized`, then all particles "originating" from the product will act as if they had been declared as `polarized` — their polarization will be recorded in the generated events. E.g., the example

```
process test = u:d, ubar:dbar => d:u, dbar:ubar, u, ubar

! insert compilation, cuts and integration here

polarized d, dbar
simulate (test) {?polarized_events = true}
```

will generate events including helicity information for all final state d and $\overline{d}$ quarks, but only for part of the final state u and $\overline{u}$ quarks. In this case, if you had wanted to keep the helicity information also for all u and $\overline{u}$, you would have had to explicitly include them into the `polarized` statement.

5.7 Cross sections

Integrating matrix elements over phase space is the core of `WHIZARD`'s activities. For any process where we want the cross section, distributions, or event samples, the cross section has to be determined first. This is done by a doubly adaptive multi-channel Monte-Carlo integration. The integration, in turn, requires a *phase-space setup*, i.e., a collection of phase-space *channels*, which are mappings of the unit hypercube onto the complete space of multi-particle kinematics. This phase-space information is encoded in the file *xxx*.`phs`, where *xxx* is the process tag. `WHIZARD` generates the phase-space file on the fly and can reuse it in later integrations.

For each phase-space channel, the unit hypercube is binned in each dimension. The bin boundaries are allowed to move during a sequence of iterations, each with a fixed number of sampled phase-space points, so they adapt to the actual phase-space density as far as possible. In addition to this *intrinsic* adaptation, the relative channel weights are also allowed to vary.

All these steps are done automatically when the `integrate` command is executed. At the end of the iterative adaptation procedure, the program has obtained an estimate for the integral of the matrix element over phase space, together with an error estimate, and a set of integration *grids* which contains all information on channel weights and bin boundaries. This information is stored in a file *xxx*.`vg`, where *xxx* is the process tag, and is used for event generation by the `simulate` command.

5.7.1 Integration

Since everything can be handled automatically using default parameters, it often suffices to write the command

```
        integrate (proc1)
```

for integrating the process with name tag `proc1`, and similarly

```
        integrate (proc1, proc2, proc3)
```

for integrating several processes consecutively. Options to the integrate command are specified, if not globally, by a local option string

```
        integrate (proc1, proc2, proc3) { mH = 200 GeV }
```

(It is possible to place a `beams` statement inside the option string, if desired.)

If the process is configured but not compiled, compilation will be done automatically. If it is not available at all, integration will fail.

The integration method can be specified by the string variable

```
        $integration_method = "<method>"
```

The default method is called `"vamp"` and uses the `VAMP` algorithm and code. (At the moment, there is only a single simplistic alternative, using the midpoint rule or rectangle method for integration, `"midpoint"`. This is mainly for testing purposes. In future versions of `WHIZARD`, more methods like e.g. Gauss integration will be made available). `VAMP`, however, is clearly the main integration method. It is done in several *passes* (usually two), and each pass consists of several *iterations*. An iteration consists of a definite number of *calls* to the matrix-element function.

For each iteration, `WHIZARD` computes an estimate of the integral and an estimate of the error, based on the binned sums of matrix element values and squares. It also computes an estimate of the rejection efficiency for generating unweighted events, i.e., the ratio of the average sampling function value over the maximum value of this function.

After each iteration, both the integration grids (the binnings) and the relative weights of the integration channels can be adapted to minimize the variance estimate of the integral. After each pass of several iterations, `WHIZARD` computes an average of the iterations within the pass, the corresponding error estimate, and a χ^2 value. The integral, error, efficiency and χ^2 value computed for the most recent integration pass, together with the most recent integration grid, are used for any subsequent calculation that involves this process, in particular for event generation.

In the default setup, during the first pass(es) both grid binnings and channel weights are adapted. In the final (usually second) pass, only binnings are further adapted. Roughly speaking, the final pass is the actual calculation, while the previous pass(es) are used for "warming up" the integration grids, without using the numerical results. Below, in the section about the specification of the iterations, Sec. 5.7.3, we will explain how it is possible to change the behavior of adapting grids and weights.

Here is an example of the integration output, which illustrates these properties. The `SINDARIN` script describes the process $e^+e^- \to q\bar{q}q\bar{q}$ with q being any light quark, i.e., W^+W^- and ZZ production and hadronic decay together will any irreducible background. We cut on p_T and energy of jets, and on the invariant mass of jet pairs. Here is the script:

```
    alias q = d:u:s:c
    alias Q = D:U:S:C
    process proc_4f = e1, E1 => q, Q, q, Q

    ms = 0  mc = 0
    sqrts = 500 GeV
    cuts = all (Pt > 10 GeV and E > 10 GeV) [q:Q]
       and all M > 10 GeV [q:Q, q:Q]

    integrate (proc_4f)
```

After the run is finished, the integration output looks like

```
    | Process library 'default_lib': loading
    | Process library 'default_lib': ... success.
    | Integrate: compilation done
    | RNG: Initializing TAO random-number generator
    | RNG: Setting seed for random-number generator to 12511
    | Initializing integration for process proc_4f:
    | ------------------------------------------------------------------------
    | Process [scattering]: 'proc_4f'
    |   Library name  = 'default_lib'
    |   Process index = 1
    |   Process components:
    |     1: 'proc_4f_i1':   e-, e+ => d:u:s:c, dbar:ubar:sbar:cbar,
    |                                  d:u:s:c, dbar:ubar:sbar:cbar [omega]
    | ------------------------------------------------------------------------
    | Beam structure: [any particles]
    | Beam data (collision):
    |   e-  (mass = 5.1099700E-04 GeV)
    |   e+  (mass = 5.1099700E-04 GeV)
    |   sqrts = 5.000000000000E+02 GeV
    | Phase space: generating configuration ...
    | Phase space: ... success.
    | Phase space: writing configuration file 'proc_4f_i1.phs'
    | Phase space: 123 channels, 8 dimensions
    | Phase space: found 123 channels, collected in 15 groves.
    | Phase space: Using 195 equivalences between channels.
    | Phase space: wood
    | Applying user-defined cuts.
    | OpenMP: Using 8 threads
    | Starting integration for process 'proc_4f'
    | Integrate: iterations not specified, using default
    | Integrate: iterations = 10:10000:"gw", 5:20000:""
    | Integrator: 15 chains, 123 channels, 8 dimensions
    | Integrator: Using VAMP channel equivalences
    | Integrator: 10000 initial calls, 20 bins, stratified = T
    | Integrator: VAMP
    |=============================================================================|
    | It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
    |=============================================================================|
       1       9963  2.3797857E+03  3.37E+02    14.15   14.13*   4.02
```

```
   2       9887   2.8307603E+03  9.58E+01   3.39   3.37*   4.31
   3       9815   3.0132091E+03  5.10E+01   1.69   1.68*   8.37
   4       9754   2.9314937E+03  3.64E+01   1.24   1.23*  10.65
   5       9704   2.9088284E+03  3.40E+01   1.17   1.15*  12.99
   6       9639   2.9725788E+03  3.53E+01   1.19   1.17   15.34
   7       9583   2.9812484E+03  3.10E+01   1.04   1.02*  17.97
   8       9521   2.9295139E+03  2.88E+01   0.98   0.96*  22.27
   9       9435   2.9749262E+03  2.94E+01   0.99   0.96   20.25
  10       9376   2.9563369E+03  3.01E+01   1.02   0.99   21.10
|-------------------------------------------------------------------------------|
  10      96677   2.9525019E+03  1.16E+01   0.39   1.22   21.10    1.15   10
|-------------------------------------------------------------------------------|
  11      19945   2.9599072E+03  2.13E+01   0.72   1.02   15.03
  12      19945   2.9367733E+03  1.99E+01   0.68   0.96*  12.68
  13      19945   2.9487747E+03  2.03E+01   0.69   0.97   11.63
  14      19945   2.9777794E+03  2.03E+01   0.68   0.96*  11.19
  15      19945   2.9246612E+03  1.95E+01   0.67   0.94*  10.34
|-------------------------------------------------------------------------------|
  15      99725   2.9488622E+03  9.04E+00   0.31   0.97   10.34    1.05    5
|===============================================================================|
| Time estimate for generating 10000 events: 0d:00h:00m:51s
| Creating integration history display proc_4f-history.ps and proc_4f-history.pdf
```

Each row shows the index of a single iteration, the number of matrix element calls for that iteration, and the integral and error estimate. Note that the number of calls displayed are the real calls to the matrix elements after all cuts and possible rejections. The error should be viewed as the 1σ uncertainty, computed on a statistical basis. The next two columns display the error in percent, and the *accuracy* which is the same error normalized by $\sqrt{n_{\mathrm{calls}}}$. The accuracy value has the property that it is independent of n_{calls}, it describes the quality of adaptation of the current grids. Good-quality grids have a number of order one, the smaller the better. The next column is the estimate for the rejection efficiency in percent. Here, the value should be as high as possible, with $100\,\%$ being the possible maximum.

In the example, the grids are adapted over ten iterations, after which the accuracy and efficiency have saturated at about 1.0 and $10\,\%$, respectively. The asterisk in the accuracy column marks those iterations where an improvement over the previous iteration is seen. The average over these iterations exhibits an accuracy of 1.22, corresponding to $0.39\,\%$ error, and a χ^2 value of 1.15, which is just right: apparently, the phase-space for this process and set of cuts is well-behaved. The subsequent five iterations are used for obtaining the final integral, which has an accuracy below one (error $0.3\,\%$), while the efficiency settles at about $10\,\%$. In this example, the final χ^2 value happens to be quite small, i.e., the individual results are closer together than the error estimates would suggest. One should nevertheless not scale down the error, but rather scale it up if the χ^2 result happens to be much larger than unity: this often indicates sub-optimally adapted grids, which insufficiently map some corner of phase space.

One should note that all values are subject to statistical fluctuations, since the number of calls within each iterations is finite. Typically, fluctuations in the efficiency estimate are considerably larger than fluctuations in the error/accuracy estimate. Two subsequent runs of the same script should yield statistically independent results which may differ in all quantities,

Integration Results Display

Process: `proc_4f-history`
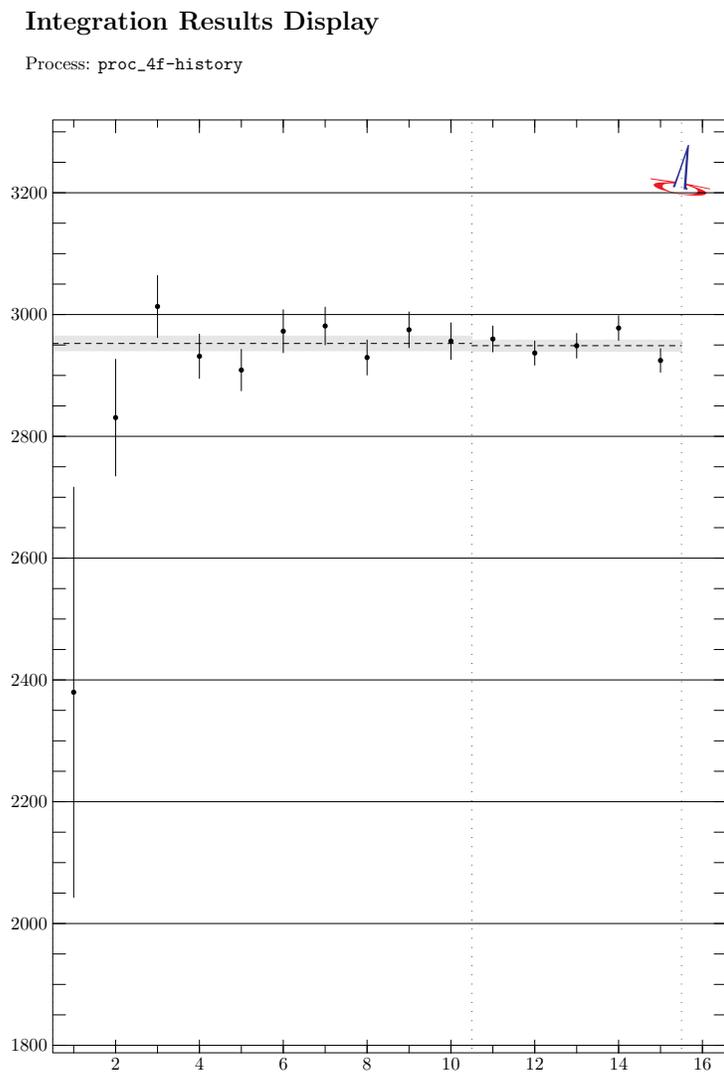
Figure 5.1:   *Graphical output of the convergence of the adaptation during the integration of a* `WHIZARD` *process.*

within the error estimates, since the seed of the random-number generator will differ by default.

It is possible to get exactly reproducible results by setting the random-number seed explicitly, e.g.,

```
seed = 12345
```

at any point in the SINDARIN script. `seed` is a predefined intrinsic variable. The value can be any 32bit integer. Two runs with different seeds can be safely taken as statistically independent. In the example above, no seed has been set, and the seed has therefore been determined internally by WHIZARD from the system clock.

The concluding line with the time estimate applies to a subsequent simulation step with unweighted events, which is not actually requested in the current example. It is based on the timing and efficiency estimate of the most recent iteration.

As a default, a graphical output of the integration history will be produced (if both LATEX and `MetaPost` have been available during configuration). Fig. 5.1 shows how this looks like, and demonstrates how a proper convergence of the integral during the adaptation looks like. The generation of these graphical history files can be switched off using the command `?vis_history = false`.

5.7.2   Integration run IDs

A single SINDARIN script may contain multiple calls to the `integrate` command with different parameters. By default, files generated for the same process in a subsequent integration will overwrite the previous ones. This is undesirable when the script is re-run: all results that have been overwritten have to be recreated.

To avoid this, the user may identify a specific run by a string-valued ID, e.g.

```
integrate (foo) { $run_id = "first" }
```

This ID will become part of the file name for all files that are created specifically for this run. Often it is useful to create a run ID from a numerical value using `sprintf`, e.g., in this scan:

```
scan real mh = (100 => 200 /+ 10) {
  $run_id = sprintf "%e" (mh)
  integrate (h_production)
}
```

With unique run IDs, a subsequent run of the same SINDARIN script will be able to reuse all previous results, even if there is more than a single integration per process.

5.7.3   Controlling iterations

WHIZARD has some predefined numbers of iterations and calls for the first and second integration pass, respectively, which depend on the number of initial and final-state particles. They are guesses for values that yield good-quality grids and error values in standard situations, where no exceptionally strong peaks or loose cuts are present in the integrand. Actually, the large number of warmup iterations in the previous example indicates some safety margin in that respect.

It is possible, and often advisable, to adjust the iteration and call numbers to the particular situation. One may reduce the default numbers to short-cut the integration, if either less accuracy is needed, or CPU time is to be saved. Otherwise, if convergence is bad, the number of iterations or calls might be increased.

To set iterations manually, there is the `iterations` command:

```
iterations = 5:50000, 3:100000
```

This is a comma-separated list. Each pair of values corresponds to an integration pass. The value before the colon is the number of iterations for this pass, the other number is the number of calls per iteration.

While the default number of passes is two (one for warmup, one for the final result), you may specify a single pass

```
iterations = 5:100000
```

where the relative channel weights will *not* be adjusted (because this is the final pass). This is appropriate for well-behaved integrands where weight adaptation is not necessary.

You can also define more than two passes. That might be useful when reusing a previous grid file with insufficient quality: specify the previous passes as-is, so the previous results will be read in, and then a new pass for further adaptation.

In the final pass, the default behavior is to not adapt grids and weights anymore. Otherwise, different iterations would be correlated, and a final reliable error estimate would not be possible. For all but the final passes, the user can decide whether to adapt grids and weights by attaching a string specifier to the number of iterations: `"g"` does adapt grids, but not weights, `"w"` the other way round. `"gw"` or `"wg"` does adapt both. By the setting `""`, all adaptations are switched off. An example looks like this:

```
iterations = 2:10000:"gw", 3:5000
```

Since it is often not known beforehand how many iterations the grid adaptation will need, it is generally a good idea to give the first pass a large number of iterations. However, in many cases these turn out to be not necessary. To shortcut iterations, you can set any of

```
accuracy_goal
error_goal
relative_error_goal
```

to a positive value. If this is done, `WHIZARD` will skip warmup iterations once all of the specified goals are reached by the current iteration. The final iterations (without weight adaptation) are always performed.

5.7.4 Phase space

Before `integrate` can start its work, it must have a phase-space configuration for the process at hand. The method for the phase-space parameterization is determined by the string variable `$phs_method`. At the moment there are only two options, `"single"`, for testing purposes, that is mainly used internally, and `WHIZARD`'s traditional method, `"wood"`. This parameterization

is particularly adapted and fine-tuned for electroweak processes and might not be the ideal for for pure jet cross sections. In future versions of WHIZARD, more options for phase-space parameterizations will be made available, e.g. the RAMBO algorithm and its massive cousin, and phase-space parameterizations that take care of the dipole-like emission structure in collinear QCD (or QED) splittings. For the standard method, the phase-space parameterization is laid out in an ASCII file *<process-name>*_i*<comp>*.phs. Here, *<process-name>* is the process name chosen by the user while *<comp>* is the number of the process component of the corresponding process. This immediately shows that different components of processes are getting different phase space setups. This is necessary for inclusive processes, e.g. the sum of $pp \rightarrow Z + nj$ and $pp \rightarrow W + nj$, or in future versions of WHIZARD for NLO processes, where one component is the interference between the virtual and the Born matrix element, and another one is the subtraction terms. Normally, you do not have to deal with this file, since WHIZARD will generate one automatically if it does not find one. (WHIZARD is careful to check for consistency of process definition and parameters before using an existing file.)

Experts might find it useful to generate a phase-space file and inspect and/or modify it before proceeding further. To this end, there is the parameter ?phs_only. If you set this true, WHIZARD skips the actual integration after the phase-space file has been generated. There is also a parameter ?vis_channels which can be set independently; if this is true, WHIZARD will generate a graphical visualization of the phase-space parameterizations encoded in the phase-space file. This file has to be taken with a grain of salt because phase space channels are represented by sample Feynman diagrams for the corresponding channel. This does however *not* mean that in the matrix element other Feynman diagrams are missing (the default matrix element method, O'Mega, is not using Feynman-diagrammatic amplitudes at all).

Things might go wrong with the default phase-space generation, or manual intervention might be necessary to improve later performance. There are a few parameters that control the algorithm of phase-space generation. To understand their meaning, you should realize that phase-space parameterizations are modeled after (dominant) Feynman graphs for the current process.

The main phase space setup *wood*

For the main phase-space parameterization of WHIZARD, which is called "wood", there are many different parameters and flags that allow to tune and customize the phase-space setup for every certain process:

The parameter phs_off_shell controls the number of off-shell lines in those graphs, not counting s-channel resonances and logarithmically enhanced s- and t-channel lines. The default value is 2. Setting it to zero will drop everything that is not resonant or logarithmically enhanced. Increasing it will include more subdominant graphs. (WHIZARD increases the value automatically if the default value does not work.)

There is a similar parameter phs_t_channel which controls multiperipheral graphs in the parameterizations. The default value is 6, so graphs with up to 6 t/u-channel lines are considered. In particular cases, such as $e^+e^- \rightarrow n\gamma$, all graphs are multiperipheral, and for $n > 7$ WHIZARD would find no parameterizations in the default setup. Increasing the value of phs_t_channel

solves this problem. (This is presently not done automatically.)

There are two numerical parameters that describe whether particles are treated like massless particles in particular situations. The value of `phs_threshold_s` has the default value 50 GeV. Hence, W and Z are considered massive, while b quarks are considered massless. This categorization is used for deciding whether radiation of b quarks can lead to (nearly) singular behavior, i.e., logarithmic enhancement, in the infrared and collinear regions. If yes, logarithmic mappings are applied to phase space. Analogously, `phs_threshold_t` decides about potential t-channel singularities. Here, the default value is 100 GeV, so amplitudes with W and Z in the t-channel are considered as logarithmically enhanced. For a high-energy hadron collider of 40 or 100 TeV energy, also W and Z in s-channel like situations might be necessary to be considered massless.

Such logarithmic mappings need a dimensionful scale as parameter. There are three such scales, all with default value 10 GeV: `phs_e_scale` (energy), `phs_m_scale` (invariant mass), and `phs_q_scale` (momentum transfer). If cuts and/or masses are such that energies, invariant masses of particle pairs, and momentum transfer values below 10 GeV are excluded or suppressed, the values can be kept. In special cases they should be changed: for instance, if you want to describe $\gamma^* \to \mu^+\mu^-$ splitting well down to the muon mass, no cuts, you may set `phs_m_scale = mmu`. The convergence of the Monte-Carlo integration result will be considerably faster.

There are more flags. These and more details about the phase space parameterization will be described in Sec. 8.3.

5.7.5   Cuts

`WHIZARD` 2 does not apply default cuts to the integrand. Therefore, processes with massless particles in the initial, intermediate, or final states may not have a finite cross section. This fact will manifest itself in an integration that does not converge, or is unstable, or does not yield a reasonable error or reweighting efficiency even for very large numbers of iterations or calls per iterations. When doing any calculation, you should verify first that the result that you are going to compute is finite on physical grounds. If not, you have to apply cuts that make it finite.

A set of cuts is defined by the `cuts` statement. Here is an example

```
cuts = all Pt > 20 GeV [colored]
```

This implies that events are kept only (for integration and simulation) if the transverse momenta of all colored particles are above 20 GeV.

Technically, `cuts` is a special object, which is unique within a given scope, and is defined by the logical expression on the right-hand side of the assignment. It may be defined in global scope, so it is applied to all subsequent processes. It may be redefined by another `cuts` statement. This overrides the first cuts setting: the `cuts` statement is not cumulative. Multiple cuts should be specified by the logical operators of `SINDARIN`, for instance

```
cuts = all Pt > 20 GeV [colored]
  and all E > 5 GeV [photon]
```

Cuts may also be defined local to an `integrate` command, i.e., in the options in braces. They will apply only to the processes being integrated, overriding any global cuts.

The right-hand side expression in the `cuts` statement is evaluated at the point where it is used by an `integrate` command (which could be an implicit one called by `simulate`). Hence, if the logical expression contains parameters, such as

```
mH = 120 GeV
cuts = all M > mH [b, bbar]
mH = 150 GeV
integrate (myproc)
```

the Higgs mass value that is inserted is the value in place when `integrate` is evaluated, 150 GeV in this example. This same value will also be used when the process is called by a subsequent `simulate`; it is `integrate` which compiles the cut expression and stores it among the process data. This behavior allows for scanning over parameters without redefining the cuts every time.

The cut expression can make use of all variables and constructs that are defined at the point where it is evaluated. In particular, it can make use of the particle content and kinematics of the hard process, as in the example above. In addition to the predefined variables and those defined by the user, there are the following variables which depend on the hard process:

integer:	`n_in`, `n_out`, `n_tot`
real:	`sqrts`, `sqrts_hat`

Example:

```
cuts = sqrts_hat > 150 GeV
```

The constants `n_in` etc. are sometimes useful if a generic set of cuts is defined, which applies to various processes simultaneously.

The user is encouraged to define his/her own set of cuts, if possible in a process-independent manner, even if it is not required. The `include` command allows for storing a set of cuts in a separate SINDARIN script which may be read in anywhere. As an example, the system directories contain a file `default_cuts.sin` which may be invoked by

```
include ("default_cuts.sin")
```

5.7.6   QCD scale and coupling

WHIZARD treats all physical parameters of a model, the coefficients in the Lagrangian, as constants. As a leading-order program, WHIZARD does not make use of running parameters as they are described by renormalization theory. For electroweak interactions where the perturbative expansion is sufficiently well behaved, this is a consistent approach.

As far as QCD is concerned, this approach does not yield numerically reliable results, even on the validity scale of the tree approximation. In WHIZARD2, it is therefore possible to replace the fixed value of α_s (which is accessible as the intrinsic model variable `alphas`), by a function of an energy scale μ.

This is controlled by the parameter `?alphas_is_fixed`, which is `true` by default. Setting it to `false` enables running α_s. The user has then to decide how α_s is calculated.

One option is to set `?alphas_from_lhapdf` (default `false`). This is recommended if the LHAPDF library is used for including structure functions, but it may also be set if LHAPDF is not invoked. WHIZARD will then use the α_s formula and value that matches the active LHAPDF structure function set and member.

In the very same way, the α_s running from the PDFs implemented intrinsically in WHIZARD can be taken by setting `?alphas_from_pdf_builtin` to `true`. This is the same running then the one from LHAPDF, if the intrinsic PDF coincides with a PDF chosen from LHAPDF.

If this is not appropriate, there are again two possibilities. If `?alphas_from_mz` is `true`, the user input value `alphas` is interpreted as the running value $\alpha_s(m_Z)$, and for the particular event, the coupling is evolved to the appropriate scale μ. The formula is controlled by the further parameters `alphas_order` (default 0, meaning leading-log; maximum 2) and `alphas_nf` (default 5).

Otherwise there is the option to set `?alphas_from_lambda_qcd = true` in order to evaluate α_s from the scale Λ_{QCD}, represented by the intrinsic variable `lambda_qcd`. The reference value for the QCD scale is $\Lambda\_QCD = 200$ MeV. `alphas_order` and `alphas_nf` apply analogously.

Note that for using one of the running options for α_s, always `?alphas_is_fixed = false` has to be invoked.

In any case, if α_s is not fixed, each event has to be assigned an energy scale. By default, this is $\sqrt{\hat{s}}$, the partonic invariant mass of the event. This can be replaced by a user-defined scale, the special object `scale`. This is assigned and used just like the `cuts` object. The right-hand side is a real-valued expression. Here is an example:

```
scale = eval Pt [sort by -Pt [colored]]
```

This selects the p_T value of the first entry in the list of colored particles sorted by decreasing p_T, i.e., the p_T of the hardest jet.

The `scale` definition is used not just for running α_s (if enabled), but it is also the factorization scale for the LHAPDF structure functions.

These two values can be set differently by specifying `factorization_scale` for the scale at which the PDFs are evaluated. Analogously, there is a variable `renormalization_scale` that sets the scale value for the running α_s. Whenever any of these two values is set, it supersedes the `scale` value.

Just like the `cuts` expression, the expressions for `scale`, `factorization_scale` and also `renormalization_scale` are evaluated at the point where it is read by an explicit or implicit `integrate` command.

5.7.7 Reweighting factor

It is possible to reweight the integrand by a user-defined function of the event kinematics. This is done by specifying a `weight` expression. Syntax and usage is exactly analogous to the `scale` expression. Example:

```
weight = eval (1 + cos (Theta) ^ 2) [lepton]
```

We should note that the phase-space setup is not aware of this reweighting, so in complicated cases you should not expect adaptation to achieve as accurate results as for plain cross sections.

Needless to say, the default `weight` is unity.

5.8   Events

After the cross section integral of a scattering process is known (or the partial-width integral of a decay process), `WHIZARD` can generate event samples. There are two limiting cases or modes of event generation:

1. For a physics simulation, one needs *unweighted* events, so the probability of a process and a kinematical configuration in the event sample is given by its squared matrix element.

2. Monte-Carlo integration yields *weighted* events, where the probability (without any grid adaptation) is uniformly distributed over phase space, while the weight of the event is given by its squared matrix element.

The choice of parameterizations and the iterative adaptation of the integration grids gradually shift the generation mode from option 2 to option 1, which obviously is preferred since it simulates the actual outcome of an experiment. Unfortunately, this adaptation is perfect only in trivial cases, such that the Monte-Carlo integration yields non-uniform probability still with weighted events. Unweighted events are obtained by rejection, i.e., accepting an event with a probability equal to its own weight divided by the maximal possible weight. Furthermore, the maximal weight is never precisely known, so this probability can only be estimated.

The default generation mode of `WHIZARD` is unweighted. This is controlled by the parameter `?unweighted` with default value `true`. Unweighted events are easy to interpret and can be directly compared with experiment, if properly interfaced with detector simulation and analysis.

However, when applying rejection to generate unweighted events, the generator discards information, and for a single event it needs, on the average, $1/\epsilon$ calls, where the efficiency ϵ is the ratio of the average weight over the maximal weight. If `?unweighted` is `false`, all events are kept and assigned their respective weights in histograms or event files.

5.8.1   Simulation

The `simulate` command generates an event sample. The number of events can be set either by specifying the integer variable `n_events`, or by the real variable `luminosity`. (This holds for unweighted events. If weighted events are requested, the luminosity value is ignored.) The luminosity is measured in femtobarns, but other units can be used, too. Since the cross sections for the processes are known at that point, the number of events is determined as the luminosity multiplied by the cross section.

As usual, both parameters can be set either as global or as local parameters:

```
n_events = 10000
simulate (proc1)
simulate (proc2, proc3) { luminosity = 100 / 1 pbarn }
```

In the second example, both `n_events` and `luminosity` are set. In that case, `WHIZARD` chooses whatever produces the larger number of events.

If more than one process is specified in the argument of `simulate`, events are distributed among the processes with fractions proportional to their cross section values. The processes are mixed randomly, as it would be the case for real data.

The raw event sample is written to a file which is named after the first process in the argument of `simulate`. If the process name is `proc1`, the file will be named `proc1.evx`. You can choose another basename by the string variable `$sample`. For instance,

```
simulate (proc1) { n_events = 4000  $sample = "my_events" }
```

will produce an event file `my_events.evx` which contains 4000 events.

This event file is in a machine-dependent binary format, so it is not of immediate use. Its principal purpose is to serve as a cache: if you re-run the same script, before starting simulation, it will look for an existing event file that matches the input. If nothing has changed, it will find the file previously generated and read in the events, instead of generating them. Thus you can modify the analysis or any further steps without repeating the time-consuming task of generating a large event sample. If you change the number of events to generate, the program will make use of the existing event sample and generate further events only when it is used up. If necessary, you can suppress the writing/reading of the raw event file by the parameters `?write_raw` and `?read_raw`.

If you try to reuse an event file that has been written by a previous version of `WHIZARD`, you may run into an incompatibility, which will be detected as an error. If this happens, you may enforce a compatibility mode (also for writing) by setting `$event_file_version` to the appropriate version string, e.g., `"2.0"`. Be aware that this may break some more recent features in the event analysis.

Generating an event sample can serve several purposes. First of all, it can be analyzed directly, by `WHIZARD`'s built-in capabilities, to produce tables, histograms, or calculate inclusive observables. The basic analysis features of `WHIZARD` are described below in Sec. 5.9. It can be written to an external file in a standard format that a human or an external program can understand. In Chap. 11, you will find a more thorough discussion of event generation with `WHIZARD`, which also covers in detail the available event-file formats. Finally, `WHIZARD` can rescan an existing event sample. The event sample may either be the result of a previous `simulate` run or, under certain conditions, an external event sample produced by another generator or reconstructed from data.

```
rescan "my_events" (proc1) { $pdf_builtin_set = "MSTW2008LO" }
```

The rescanning may apply different parameters and recalculate the matrix element, it may apply a different event selection, it may reweight the events by a different PDF set (as above). The modified event sample can again be analyzed or written to file. For more details, cf. Sec. 11.7.

5.8.2 Decays

Normally, the events generated by the `simulate` command will be identical in structure to the events that the `integrate` command generates. This implies that for a process such as

$pp \rightarrow W^+W^-$, the final-state particles are on-shell and stable, so they appear explicitly in the generated event files. If events are desired where the decay products of the W bosons appear, one has to generate another process, e.g., $pp \rightarrow u\bar{d}\bar{u}d$. In this case, the intermediate vector bosons, if reconstructed, are off-shell as dictated by physics, and the process contains all intermediate states that are possible. In this example, the matrix element contains also ZZ, photon, and non-resonant intermediate states. (This can be restricted via the `$restrictions` option, cf. 5.4.3.

Another approach is to factorize the process in production (of W bosons) and decays ($W \rightarrow q\bar{q}$). This is actually the traditional approach, since it is much less computing-intensive. The factorization neglects all off-shell effects and irreducible background diagrams that do not have the decaying particles as an intermediate resonance. While `WHIZARD` is able to deal with multi-particle processes without factorization, the needed computing resources rapidly increase with the number of external particles. Particularly, it is the phase space integration that becomes the true bottleneck for a high multiplicity of final state particles.

In order to use the factorized approach, one has to specify particles as `unstable`. (Also, the `?allow_decays` switch must be `true`; this is however its default value.) We give an example for a $pp \rightarrow Wj$ final state:

```
process wj  = u, gl => d, Wp
process wen = Wp => E1, n1

integrate (wen)

sqrts = 7 TeV
beams = p, p => pdf_builtin
unstable Wp (wen)
simulate (wj) { n_events = 1 }
```

This defines a $2 \rightarrow 2$ hard scattering process of $W + j$ production at the 7 TeV LHC 2011 run. The W^+ is marked as unstable, with its decay process being $W^+ \rightarrow e^+\nu_e$. In the `simulate` command both processes, the production process `wj` and the decay process `wen` will be integrated, while the W decays become effective only in the final event sample. This event sample will contain final states with multiplicity 3, namely $e^+\nu_e d$. Note that here only one decay process is given, hence the branching ratio for the decay will be taken to be 100% by `WHIZARD`.

A natural restriction of the factorized approach is the implied narrow-width approximation. Theoretically, this restriction is necessary since whenever the width plays an important role, the usage of the factorized approach will not be fully justified. In particular, all involved matrix elements must be evaluated on-shell, or otherwise gauge-invariance issues could spoil the calculation. (There are plans for a future `WHIZARD` version to also include Breit-Wigner or Gaussian distributions when using the factorized approach.)

Decays can be concatenated, e.g. for top pair production and decay, $e^+e^- \rightarrow t\bar{t}$ with decay $t \rightarrow W^+b$, and subsequent leptonic decay of the W as in $W^+ \rightarrow \mu^+\nu_\mu$:

```
process eett = e1, E1 => t, tbar
process t_dec = t => Wp, b
process W_dec = Wp => E2, n2

unstable t (t_dec)
```

```
      unstable Wp (W_dec)

      sqrts = 500
      simulate (eett) { n_events = 1 }
```

Note that in this case the final state in the event file will consist of $\bar{t}b\mu^+\nu_\mu$ because the anti-top is not decayed.

If more than one decay process is being specified like in

```
        process eeww = e1, E1 => Wp, Wm
        process w_dec1 = Wp => E2, n2
        process w_dec2 = Wp => E3, n3

        unstable Wp (w_dec1, w_dec2)

        sqrts = 500
        simulate (eeww) { n_events = 100 }
```

then `WHIZARD` takes the integrals of the specified decay processes and distributes the decays statistically according to the calculated branching ratio. Note that this might not be the true branching ratios if decay processes are missing, or loop corrections to partial widths give large(r) deviations. In the calculation of the code above, `WHIZARD` will issue an output like

```
    | Unstable particle W+: computed branching ratios:
    |   w_dec1: 5.0018253E-01   mu+, numu
    |   w_dec2: 4.9981747E-01   tau+, nutau
    |   Total width = 4.5496085E-01 GeV (computed)
    |               = 2.0490000E+00 GeV (preset)
    |   Decay options: helicity treated exactly
```

So in this case, `WHIZARD` uses 50 % muonic and 50 % tauonic decays of the positively charged W, while the W^- appears directly in the event file. `WHIZARD` shows the difference between the preset W width from the physics model file and the value computed from the two decay channels.

Note that a particle in a `SINDARIN` input script can be also explictly marked as being stable, using the

```
      stable <particle-tag>
```

constructor for the particle `<particle-tag>`.

Resetting branching fractions

As described above, decay processes that appear in a simulation must first be integrated by the program, either explicitly via the `integrate` command, or implicitly by `unstable`. In either case, `WHIZARD` will use the computed partial widths in order to determine branching fractions. In the spirit of a purely leading-order calculation, this is consistent.

However, it may be desired to rather use different branching-fraction values for the decays of a particle, for instance, NLO-corrected values. In fact, after `WHIZARD` has integrated any process, the integration result becomes available as an ordinary `SINDARIN` variable. For instance, if a decay process has the ID `h_bb`, the integral of this process – the partial width, in this case – becomes the variable `integral(h_bb)`. This variable may be reset just like any other variable:

```
integral(h_bb) = 2.40e-3 GeV
```

The new value will be used for all subsequent Higgs branching-ratio calculations and decays, if an unstable Higgs appears in a process for simulation.

Spin correlations in decays

By default, WHIZARD applies full spin and color correlations to the factorized processes, so it keeps both color and spin coherence between productions and decays. Correlations between decay products of distinct unstable particles in the same event are also fully retained. The program sums over all intermediate quantum numbers.

Although this approach obviously yields the optimal description with the limits of production-decay factorization, there is support for a simplified handling of particle decays. Essentially, there are four options, taking a decay `W_ud`: $W^- \to \bar{u}d$ as an example:

1. Full spin correlations: `unstable Wp (W_ud)`

2. Isotropic decay: `unstable Wp (W_ud) { ?isotropic_decay = true }`

3. Diagonal decay matrix: `unstable Wp (W_ud) { ?diagonal_decay = true }`

4. Project onto specific helicity: `unstable Wp (W_ud) { decay_helicity = -1 }`

Here, the isotropic option completely eliminates spin correlations. The diagonal-decays option eliminates just the off-diagonal entries of the W spin-density matrix. This is equivalent to a measurement of spin before the decay. As a result, spin correlations are still present in the classical sense, while quantum coherence is lost. The definite-helicity option is similar and additional selects only the specified helicity component for the decaying particle, so its decay distribution assumes the shape for an accordingly polarized particle. All options apply in the rest frame of the decaying particle, with the particle's momentum as the quantization axis.

Automatic decays

A convenient option is if the user did not have to specify the decay mode by hand, but if they were generated automatically. WHIZARD does have this option: the flag `?auto_decays` can be set to `true`, and is taking care of that. In that case the list for the decay processes of the particle marked as unstable is left empty (we take a W^- again as example):

```
unstable Wm () { ?auto_decays = true }
```

WHIZARD then inspects at the local position within the SINDARIN input file where that `unstable` statement appears the masses of all the particles of the active physics model in order to determine which decays are possible. It then calculates their partial widths. There are a few options to customize the decays. The integer variable `auto_decays_multiplicity` allows to set the maximal multiplicity of the final states considered in the auto decay option. The defaul value of that variable is 2; please be quite careful when setting this to values larger than that. If you do so, the flag `?auto_decays_radiative` allows to specify whether final states simply containing additional resolved gluons or photons are taken into account or not. For the example above, you almost hit the PDG value for the W total width:

```
| Unstable particle W-: computed branching ratios:
|   decay_a24_1: 3.3337068E-01   d, ubar
|   decay_a24_2: 3.3325864E-01   s, cbar
|   decay_a24_3: 1.1112356E-01   e-, nuebar
|   decay_a24_4: 1.1112356E-01   mu-, numubar
|   decay_a24_5: 1.1112356E-01   tau-, nutaubar
|   Total width = 2.0478471E+00 GeV (computed)
|             = 2.0490000E+00 GeV (preset)
|   Decay options: helicity treated exactly
```

Future shorter notation for decays

In an upcoming WHIZARD version there will be a shorter and more concise notation already in the process definition for such decays, which, however, is current not yet implemented. The two first examples above will then be shorter and have this form:

```
process wj = u, gl => (Wp => E1, n1), d
```

as well as

```
process eett = e1, E1 => (t => (Wp => E2, n2), b), tbar
```

5.8.3   Event formats

As mentioned above, the internal WHIZARD event format is a machine-dependent event format. There are a series of human-readable ASCII event formats that are supported: very verbose formats intended for debugging, formats that have been agreed upon during the Les Houches workshops like LHA and LHEF, or formats that are steered through external packages like HepMC. More details about event formats can be found in Sec. 11.5.

5.9   Analysis and Visualization

SINDARIN natively supports basic methods of data analysis and visualization which are frequently used in high-energy physics studies. Data generated during script execution, in particular simulated event samples, can be analyzed to evaluate further observables, fill histograms, and draw two-dimensional plots.

So the user does not have to rely on his/her own external graphical analysis method (like e.g. gnuplot or ROOT etc.), but can use methods that automatically ship with WHIZARD. In many cases, the user, however, clearly will use his/her own analysis machinery, especially experimental collaborations.

In the following sections, we first summarize the available data structures, before we consider their graphical display.

5.9.1   Observables

Analyses in high-energy physics often involve averages of quantities other than a total cross section. SINDARIN supports this by its `observable` objects. An `observable` is a container that collects a single real-valued variable with a statistical distribution. It is declared by a command of the form

```
observable analysis-tag
```

where *analysis-tag* is an identifier that follows the same rules as a variable name.

Once the observable has been declared, it can be filled with values. This is done via the `record` command:

```
record analysis-tag (value)
```

To make use of this, after values have been filled, we want to perform the actual analysis and display the results. For an observable, these are the mean value and the standard deviation. There is the command `write_analysis`:

```
write_analysis (analysis-tag)
```

Here is an example:

```
observable obs
record obs (1.2)  record obs (1.3)  record obs (2.1)  record obs (1.4)
write_analysis (obs)
```

The result is displayed on screen:

```
###############################################################################
# Observable: obs
average     =  1.500000000000E+00
error[abs]  =  2.041241452319E-01
error[rel]  =  1.360827634880E-01
n_entries   = 4
```

5.9.2   The analysis expression

The most common application is the computation of event observables – for instance, a forward-backward asymmetry – during simulation. To this end, there is an `analysis` expression, which behaves very similar to the `cuts` expression. It is defined either globally

```
analysis = logical-expr
```

or as a local option to the `simulate` or `rescan` commands which generate and handle event samples. If this expression is defined, it is not evaluated immediately, but it is evaluated once for each event in the sample.

In contrast to the `cuts` expression, the logical value of the `analysis` expression is discarded; the expression form has been chosen just by analogy. To make this useful, there is a variant of the `record` command, namely a `record` function with exactly the same syntax. As an example, here is a calculation of the forward-backward symmetry in a process `ee_mumu` with final state $\mu^+\mu^-$:

```
observable a_fb
analysis = record a_fb (eval sgn (Pz) ["mu-"])
simulate (ee_mumu) { luminosity = 1 / 1 fbarn }
```

The logical return value of `record` – which is discarded here – is `true` if the recording was successful. In case of histograms (see below) it is true if the value falls within bounds, false otherwise.

Note that the function version of `record` can be used anywhere in expressions, not just in the `analysis` expression.

When `record` is called for an observable or histogram in simulation mode, the recorded value is weighted appropriately. If `?unweighted` is true, the weight is unity, otherwise it is the event weight.

The `analysis` expression can involve any other construct that can be expressed as an expression in SINDARIN. For instance, this records the energy of the 4th hardest jet in a histogram pt_dist, if it is in the central region:

```
analysis =
  record pt_dist (eval E [extract index 4
                          [sort by - Pt
                            [select if -2.5 < Eta < 2.5 [colored]]]])
```

Here, if there is no 4th jet in the event which satisfies the criterion, the result will be an undefined value which is not recorded. In that case, `record` evaluates to `false`.

Selection cuts can be part of the analysis expression:

```
analysis =
  if any Pt > 50 GeV [lepton] then
    record jet_energy (eval E [collect [jet]])
  endif
```

Alternatively, we can specify a separate selection expression:

```
selection = any Pt > 50 GeV [lepton]
analysis = record jet_energy (eval E [collect [jet]])
```

The former version writes all events to file (if requested), but applies the analysis expression only to the selected events. This allows for the simultaneous application of different selections to a single event sample. The latter version applies the selection to all events before they are analyzed or written to file.

The analysis expression can make use of all variables and constructs that are defined at the point where it is evaluated. In particular, it can make use of the particle content and kinematics of the hard process, as in the example above. In addition to the predefined variables and those defined by the user, there are the following variables which depend on the hard process. Some of them are constants, some vary event by event:

```
integer:   event_index
integer:   process_num_id
string:    $process_id
integer:   n_in, n_out, n_tot
real:      sqrts, sqrts_hat
real:      sqme, sqme_ref
real:      event_weight, event_excess
```

The `process_num_id` is the numeric ID as used by external programs, while the process index refers to the current library. By default, the two are identical. The process index itself is not available as a predefined observable. The `sqme` and `sqme_ref` values indicate the squared matrix element and the reference squared matrix element, respectively. The latter applies when comparing with a reference sample (the `rescan` command).

   `record` evaluates to a logical, so several `record` functions may be concatenated by the logical operators `and` or `or`. However, since usually the further evaluation should not depend on the return value of `record`, it is more advisable to concatenate them by the semicolon (`;`) operator. This is an operator (*not* a statement separator or terminator) that connects two logical expressions and evaluates both of them in order. The lhs result is discarded, the result is the value of the rhs:

```
analysis =
  record hist_pt (eval Pt [lepton]) ; record hist_ct (eval cos (Theta) [lepton])
```

5.9.3   Histograms

In SINDARIN, a histogram is declared by the command

```
histogram analysis-tag (lower-bound, upper-bound)
```

This creates a histogram data structure for an (unspecified) observable. The entries are organized in bins between the real values *lower-bound* and *upper-bound*. The number of bins is given by the value of the intrinsic integer variable `n_bins`, the default value is 20.

   The `histogram` declaration supports an optional argument, so the number of bins can be set locally, for instance

```
histogram pt_distribution (0 GeV, 500 GeV) { n_bins = 50 }
```

Sometimes it is more convenient to set the bin width directly. This can be done in a third argument to the `histogram` command.

```
histogram pt_distribution (0 GeV, 500 GeV, 10 GeV)
```

If the bin width is specified this way, it overrides the setting of `n_bins`.

   The `record` command or function fills histograms. A single call

```
record (real-expr)
```

puts the value of `real-expr` into the appropriate bin. If the call is issued during a simulation where `unweighted` is false, the entry is weighted appropriately.

If the value is outside the range specified in the histogram declaration, it is put into one of the special underflow and overflow bins.

The `write_analysis` command prints the histogram contents as a table in blank-separated fixed columns. The columns are: x (bin midpoint), y (bin contents), Δy (error), excess weight, and n (number of entries). The output also contains comments initiated by a `#` sign, and following the histogram proper, information about underflow and overflow as well as overall contents is added.

5.9.4 Plots

While a histogram stores only summary information about a data set, a `plot` stores all data as (x, y) pairs, optionally with errors. A plot declaration is as simple as

    plot *analysis-tag*

Like observables and histograms, plots are filled by the `record` command or expression. To this end, it can take two arguments,

    record (*x-expr, y-expr*)

or up to four:

    record (*x-expr, y-expr, y-error*)
    record (*x-expr, y-expr, y-error-expr, x-error-expr*)

Note that the y error comes first. This is because applications will demand errors for the y value much more often than x errors.

The plot output, again written by `write_analysis` contains the four values for each point, again in the ordering $x, y, \Delta y, \Delta x$.

5.9.5 Analysis Output

There is a default format for piping information into observables, histograms, and plots. In older versions of `WHIZARD` there was a first version of a custom format, which was however rather limited. A more versatile custom output format will be coming soon.

1. By default, the `write_analysis` command prints all data to the standard output. The data are also written to a default file with the name `whizard_analysis.dat`. Output is redirected to a file with a different name if the variable `$out_file` has a nonempty value. If the file is already open, the output will be appended to the file, and it will be kept open. If the file is not open, `write_analysis` will open the output file by itself, overwriting any previous file with the same name, and close it again after data have been written.

   The command is able to print more than one dataset, following the syntax

```
write_analysis (analysis-tag1, analysis-tag2, ...)  { options }
```

The argument in brackets may also be empty or absent; in this case, all currently existing datasets are printed.

The default data format is suitable for compiling analysis data by `WHIZARD`'s built-in `gamelan` graphics driver (see below and particularly Chap. 12). Data are written in blank-separated fixed columns, headlines and comments are initiated by the `#` sign, and each data set is terminated by a blank line. However, external programs often require special formatting.

The internal graphics driver `gamelan` of `WHIZARD` is initiated by the `compile_analysis` command. Its syntax is the same, and it contains the `write_analysis` if that has not been separately called (which is unnecessary). For more details about the `gamelan` graphics driver and data visualization within `WHIZARD`, confer Chap. 12.

2. Custom format. Not yet (re-)implemented in a general form.

5.10 Custom Input/Output

`WHIZARD` is rather chatty. When you run examples or your own scripts, you will observe that the program echoes most operations (assignments, commands, etc.) on the standard output channel, i.e., on screen. Furthermore, all screen output is copied to a log file which by default is named `whizard.log`.

For each integration run, `WHIZARD` writes additional process-specific information to a file $\langle tag \rangle$.`log`, where $\langle tag \rangle$ is the process name. Furthermore, the `write_analysis` command dumps analysis data – tables for histograms and plots – to its own set of files, cf. Sec. 5.9.

However, there is the occasional need to write data to extra files in a custom format. `SINDARIN` deals with that in terms of the following commands:

5.10.1 Output Files

open_out

```
open_out (⟨filename⟩)
open_out (⟨filename⟩) { ⟨options⟩ }
```

Open an external file for writing. If the file exists, it is overwritten without warning, otherwise it is created. Example:

```
open_out ("my_output.dat")
```

close_out

```
close_out (⟨filename⟩)
close_out (⟨filename⟩) { ⟨options⟩ }
```

Close an external file that is open for writing. Example:

```
close_out ("my_output.dat")
```

5.10.2   Printing Data

printf

```
printf ⟨format-string-expr⟩
printf ⟨format-string-expr⟩ (⟨data-objects⟩)
```

Format ⟨**data-objects**⟩ according to ⟨**format-string-expr**⟩ and print the resulting string to standard output if the string variable $out_file is undefined. If $out_file is defined and the file with this name is open for writing, print to this file instead.

Print a newline at the end if ?out_advance is true, otherwise don't finish the line.

The ⟨**format-string-expr**⟩ must evaluate to a string. Formatting follows a subset of the rules for the printf(3) command in the C language. The supported rules are:

- All characters are printed as-is, with the exception of embedded conversion specifications.

- Conversion specifications are initiated by a percent (%) sign and followed by an optional prefix flag, an optional integer value, an optional dot followed by another integer, and a mandatory letter as the conversion specifier.

- A percent sign immediately followed by another percent sign is interpreted as a single percent sign, not as a conversion specification.

- The number of conversion specifiers must be equal to the number of data objects. The data types must also match.

- The first integer indicates the minimum field width, the second one the precision. The field is expanded as needed.

- The conversion specifiers d and i are equivalent, they indicate an integer value.

- The conversion specifier e indicates a real value that should be printed in exponential notation.

- The conversion specifier f indicates a real value that should be printed in decimal notation without exponent.

- The conversion specifier g indicates a real value that should be printed either in exponential or in decimal notation, depending on its value.

- The conversion specifier `s` indicates a logical or string value that should be printed as a string.

- Possible prefixes are `#` (alternate form, mandatory decimal point for reals), `0` (zero padding), `-` (left adjusted), `+` (always print sign), ' ' (print space before a positive number).

For more details, consult the `printf(3)` manpage. Note that other conversions are not supported and will be rejected by `WHIZARD`.

The data arguments are numeric, logical or string variables or expressions. Numeric expressions must be enclosed in parantheses. Logical expressions must be enclosed in parantheses prefixed by a question mark `?`. String expressions must be enclosed in parantheses prefixed by a dollar sign `$`. These forms behave as anonymous variables.

Note that for simply printing a text string, you may call `printf` with just a format string and no data arguments.

Examples:

```
printf "The W mass is %8f GeV" (mW)

int i = 2
int j = 3
printf "%i + %i = %i" (i, j, (i+j))

string $directory = "/usr/local/share"
string $file = "foo.dat"
printf "File path: %s/%s" ($directory, $file)
```

There is a related `sprintf` function, cf. Sec. 5.1.5.

5.11 WHIZARD at next-to-leading order

5.11.1 Prerequisites

A full NLO computation requires virtual matrix elements obtained from loop diagrams. Since `O'Mega` cannot calculate such diagrams, external programs are used. `WHIZARD` has a generic interface to matrix-element generators that are BLHA-compatible. Explicit implementations exist for `Gosam`, `OpenLoops` and `Recola`.

Setting up `Gosam`

The installation of `Gosam` is detailed on the HepForge page https://gosam/hepforge.org. We mention here some of the steps necessary to get it to be linked with `WHIZARD`.

Bug in `Gosam` installation scripts: In many versions of `Gosam` there is a bug in the installation scripts that is only relevant if `Gosam` is installed with superuser privileges. Then all files in `$installdir/share/golem` do not have read privileges for normal users. These privileges must be given manually to all files in that directory.

Prerequisites for `Gosam` to produce code for one-loop matrix elements are the scientific algebra program `form` and the generator of loop topologies and diagrams, `qgraf`. These can

be accessed via their respective webpages http://www.nikhef.nl/~form/ and http://cfif.ist.utl.pt/~paulo/qgraf.html. Note also that both `Java` and the Java runtime environment have to be installed in order for `Gosam` to properly work. Furthermore, `libtool` needs to be installed. A more convenient way to install `Gosam`, is the automatic installation script https://gosam.hepforge.org/gosam_installer.py.

Setting up `OpenLoops`

The installation of `OpenLoops` is explained in detail on the HepForge page https://openloops.hepforge.org. In the following, the main steps for usage with `WHIZARD` are summarized.

Please note that at the moment, `OpenLoops` cannot be installed such that in almost all cases the explicit `OpenLoops` package directory has to be set via `--with-openloops=<openloops_dir>`.

`OpenLoops` can be checked out with

```
git clone https://gitlab.com/openloops/OpenLoops.git
```

Note that `WHIZARD` only supports `OpenLoops` version that are at least 2.1.1 or newer. Alternatively, one can use the public beta version of `OpenLoops`, which can be checked out by the command

```
git clone -b public_beta https://gitlab.com/openloops/OpenLoops.git
```

The program can be build by running `scons` or `./scons`, a local version that is included in the `OpenLoops` directory. This produces the script `./openloops`, which is the main hook for the further usage of the program.

`OpenLoops` works by downloading prebuild process libraries, which have to be installed for each individual process. This requires the file `openloops.cfg`, which should contain the following content:

```
[OpenLoops]
process_repositories=public, whizard
compile_extra=1
```

The first line instructs `OpenLoops` to also look for process libraries in an additional lepton collider repository. The second line triggers the inclusion of $N + 1$-particle tree-level matrix elements in the process directory, so that a complete NLO calculation including real amplitudes can be performed only with `OpenLoops`.

The libraries can then be installed via

```
./openloops libinstall proc_name
```

A list of supported library names can be found on the `OpenLoops` web page. Note that a process library also includes all possible permutated processes. The process library `ppllj`, for example, can also be used to compute the matrix elements for $e^+e^- \to q\bar{q}$ (massless quarks only). The massive case of the top quark is handled in `eett`. Additionally, there are process libraries for top and gauge boson decays, `tbw`, `vjj`, `tbln` and `tbqq`.

Finally, `OpenLoops` can be linked to `WHIZARD` during configuration by including

```
--enable-openloops --with-openloops=$OPENLOOPS_PATH,
```

where `$OPENLOOPS_PATH` is the directory the `OpenLoops` executable is located in. `OpenLoops` one-loop diagrams can then be used with the `SINDARIN` option

```
$loop_me_method = "openloops".
```

The functional tests which check the `OpenLoops` functionality require the libraries `ppllj`, `eett` and `tbw` to be installed (note that `eett` is not contained in `ppll`). During the configuration of `WHIZARD`, it is automatically checked that these two libraries, as well as the option `compile_extra=1`, are present.

OpenLoops `SINDARIN` flags

Several `SINDARIN` options exist to control the behavior of `OpenLoops`.

- `openloops_verbosity`:
  Decide how much `OpenLoops` output is printed. Can have values 0, 1 and 2.

- `?openloops_use_cms`:
  Activates the complex mass scheme. For computations with decaying resonances like the top quark or W or Z bosons, this is the preferred option to avoid gauge-dependencies.

- `openloops_phs_tolerance`:
  Controls the exponent of `extra psp_tolerance` in the BLHA interface, which is the numerical tolerance for the on-shell condition of external particles

- `openloops_switch_off_muon_yukawa`:
  Sets the Yukawa coupling of muons to zero in order to assure agreement with `O'Mega`, which is possibly used for other components and per default does not take $H\mu\mu$ couplings into account.

- `openloops_stability_log`:
  Creates the directory `stability_log`, which contains information about the performance of the matrix elements. Possible values are

    - 0: No output (default),
    - 1: On finish() call,
    - 2: Adaptive,
    - 3: Always

- `?openloops_use_collier`: Use Collier as the reduction method (default true).

Setting up `Recola`

The installation of `Recola` is explained in detail on the HepForge page https://recola.hepforge.org. In the following the main steps for usage with `WHIZARD` are summarized. The minimal required version number of `Recola` is 1.3.0.

Recola can be linked to `WHIZARD` during configuration by including

```
        --enable-recola
```

In case the `Recola` library is not in a standard path or a path accessible in the `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH`) of the operating system, then the option

```
        --with-recola=$RECOLA_PATH
```

can be set, where `$RECOLA_PATH` is the directory the `Recola` library is located in. `Recola` can then be used with the `SINDARIN` option

```
        $method = "recola"
```

or any other of the matrix element methods.

Note that there might be a clash of the `Collier` libraries when you have `Collier` installed both via `Recola` and via `OpenLoops`, but have compiled them with different `Fortran` compilers.

5.11.2   NLO cross sections

An NLO computation can be switched on in `SINDARIN` with

```
        process proc_nlo = in1, in2 => out1, ..., outN { nlo_calculation = <components> },
```

where the `nlo_calculation` can be followed by a list of strings specifying the desired NLO-components to be integrated, i.e. `born`, `real`, `virtual`, `dglap`, (for hadron collisions) or `mismatch` (for the soft mismatch in resonance-aware computations) and `full`. The `full` option switches on all components and is required if the total NLO result is desired. For example, specifying

```
        nlo_calculation = born, virtual
```

will result in the computation of the Born and virtual component.

The integration can be carried out in two different modes: Combined and separate integration. In the separate integration mode, each component is integrated individually, allowing for a good overview of their contributions to the total cross section and a fine tuned control over the iterations in each component. In the combined integration mode, all components are added up during integration so that the sum of them is evaluated. Here, only one integration will be displayed. The default method is the separate integration.

The convergence of the integration can crucially be influenced by the presence of resonances. A better convergence is in this case achieved activating the resonance-aware FKS subtraction,

```
        $fks_mapping_type = "resonances".
```

This mode comes with an additional integration component, the so-called soft mismatch.

Note that you can modify the number of iterations in each component with the multipliers:

- `mult_call_real` multiplies the number of calls to be used in the integration of the real component. A reasonable choice is `10.0` as the real phase-space is more complicated than the Born but the matrix elements evaluate faster than the virtuals.

- `mult_call_virt` multiplies the number of calls to be used in the integration of the virtual component. A reasonable choice is `0.5` to make sure that the fast Born component only contributes a negligible MC error compared to the real and virtual components.

- `mult_call_dglap` multiplies the number of calls to be used in the integration of the DGLAP component.

5.11.3   Fixed-order NLO events

Fixed-order NLO events can also be produced in three different modes: Combined weighted, combined unweighted and separated weighted.

- **Combined weighted**
  In the combined mode, one single integration grid is produced, from which events are generated with the total NLO weight. The corresponding event file contains N events with born-like kinematics and weight equal to $\mathcal{B} + \mathcal{V} + \sum_{\alpha_r} \mathcal{C}_{\alpha_r}$, where $\mathcal{B}$ is the Born matrix element, $\mathcal{V}$ is the virtual matrix element and $\mathcal{C}_{\alpha_r}$ are the subtraction terms in each singular region. For resonance-aware processes, also the mismatch value is added. Each born-like event is followed by N_{phs} associated events with real kinematics, i.e. events where one additional QCD particle is present. The corresponding real matrix elements $\mathcal{R}_\alpha$ form the weight of these events. N_{phs} is the number of distinct phase spaces. Two phase spaces are distinct if they have different resonance histories and/or have different emitters. So, two α_r can share the same phase space index.

  The combined event mode is activated by

  ```
  ?combined_nlo_integration = true
  ?unweighted = false
  ?fixed_order_nlo_events = true
  ```

  Moreover, the process must be specified at next-to-leading-order in its definition using `nlo_calculation = full`. WHIZARD then proceeds as in the usual simulation mode. I.e. it first checks whether integration grids are already present and uses them if they fit. Otherwise, it starts an integration.

- **Combined unweighted**
  The unweighted combined events can be generated by using the POWHEG mode, cf. also the next subsection, but disabling the additional radiation and Sudakov factors with the `?powheg_disable_sudakov` switch:

  ```
  ?combined_nlo_integration = true
  ?powheg_matching = true
  ?powheg_disable_sudakov = true
  ```

  This will produce events with Born kinematics and unit weights (as `?unweighted` is `true` by default). The events are unweighted by using $\mathcal{B} + \mathcal{V} + \sum_{\alpha_r} (\mathcal{C}_{\alpha_r} + \mathcal{R}_{\alpha_r})$. Of course, this only works when these weights are positive over the full phase-space, which is not guaranteed for all scales and regions at NLO. However, for many processes perturbation theory works nicely and this is not an issue.

- **Separate weighted**
  In the separate mode, grids and events are generated for each individual component of

the NLO process. This method is preferable for complicated processes, since it allows to individually tune each grid generation. Moreover, the grid generation is then trivially parallelized. The event files either contain only Born kinematics with weight $\mathcal{B}$ or $\mathcal{V}$ (and mismatch in case of a resonance-aware process) or mixed Born and real kinematics for the real component like in the combined mode. However, the Born events have only the weight $\sum_{\alpha_r} \mathcal{C}_{\alpha_r}$ in this case.

The separate event mode is activated by

```
?unweighted = false
?negative_weights = true
?fixed_order_nlo_events = true
```

Note that negative weights have to be switched on because, in contrast to the combined mode, the total cross sections of the individual components can be negative.

Also, the desired component has to appear in the process NLO specification, e.g. using `nlo_calculation = real`.

Weighted fixed-order NLO events are supported by any output format that supports weights like the `HepMC` format and unweighted NLO events work with any format. The output can either be written to disk or put into a FIFO to interface it to an analysis program without writing events to file.

The weights in the real event output, both in the combined and separate weighted mode, are divided by a factor $N_{\mathrm{phs}} + 1$. This is to account for the fact that we artificially increase the number of events in the output file. Thus, the sum of all event weights correctly reproduces the total cross section.

5.11.4 POWHEG matching

To match the NLO events with a parton shower, `WHIZARD` supports the `POWHEG` matching. It generates a distribution according to

$$d\sigma = d\Phi_n \, \bar{B}_{\mathrm{s}} \left(\Delta_{\mathrm{s}}(p_T^{\mathrm{min}}) + d\Phi_{\mathrm{rad}} \, \Delta_{\mathrm{s}}(k_{\mathrm{T}}(\Phi_{\mathrm{rad}})) \frac{R_{\mathrm{s}}}{B} \right) \quad \text{where} \tag{5.5}$$

$$\bar{B}_{\mathrm{s}} = B + \mathcal{V} + d\Phi_{\mathrm{rad}} \, \mathcal{R}_{\mathrm{s}} \quad \text{and} \tag{5.6}$$

$$\Delta_{\mathrm{s}}(p_T) = \exp \left[-\int d\Phi_{\mathrm{rad}} \frac{R_{\mathrm{s}}}{B} \, \theta \left(k_T^2(\Phi_{\mathrm{rad}}) - p_T^2 \right) \right] . \tag{5.7}$$

The subscript s refers to the singular part of the real component, cf. to the next subsection. Eq. (5.5) produces either no or one additional emission. These events can then either be analyzed directly or passed on to the parton shower[6] for the full simulation. You activate this with

```
?fixed_order_nlo_events = false
?combined_nlo_integration = true
?powheg_matching = true
```

---

[6]E.g. `PYTHIA8` has explicit examples for `POWHEG` input, see also http://home.thep.lu.se/Pythia/pythia82html/POWHEGMerging.html.

The $p_T^{\min}$ of Eq. (5.5) can be set with `powheg_pt_min`. It sets the minimal scale for the `POWHEG` evolution and should be of order 1 GeV and set accordingly in the interfaced shower. The maximal scale is currently given by `sqrts` but should in the future be changeable with `powheg_pt_max`.

Note that the `POWHEG` event generation needs an additional grid for efficient event generation that is generated during integration if `?powheg_matching = true` is set. Thus, this needs to be set before the `integrate` statement. Further options that steer the efficiency of this grid are `powheg_grid_size_xi`, `powheg_grid_size_y` and `powheg_grid_sampling_points`.

5.11.5 Separation of finite and singular contributions

For both the pure NLO computations as well as the `POWHEG` event generation, `WHIZARD` supports the partitioning of the real into finite and singular contributions with the flag

        ?nlo_use_real_partition = true

The finite contributions, which by definition should not contain soft or collinear emissions, will then integrate like an ordinary LO integration with one additional particle. Similarly, the event generation will produce only real events without subtraction terms with Born kinematics for this additional finite component. The `POWHEG` event generation will also only use the singular parts.

The current implementation uses the following parametrization

$$R = R_{\text{fin}} + R_{\text{sing}} ,\tag{5.8}$$

$$R_{\text{sing}} = RF(\Phi_{n+1}) ,\tag{5.9}$$

$$R_{\text{fin}} = R(1 - F(\Phi_{n+1})) ,\tag{5.10}$$

$$F(\Phi_{n+1}) = \begin{cases} 1 & \text{if} \quad \exists\,(i,j) \in \mathcal{P}_{\text{FKS}} \quad \text{with} \quad \sqrt{(p_i + p_j)^2} < h + m_i + m_j \\ 0 & \text{else} \end{cases}\tag{5.11}$$

Thus, a point is singular ($F = 1$), if any of the FKS tuples forms an invariant mass that is smaller than the hardness scale h. This parameter is controlled in `SINDARIN` with `real_partition_scale`. This simplifies in massless case to

$$F(\Phi_{n+1}) = \begin{cases} 1 & \text{if } \exists\,(i,j) \in \mathcal{P}_{\text{FKS}} \quad \text{with} \quad 2E_i E_j(1 - \cos\theta_{ij}) < h^2 \\ 0 & \text{else} \end{cases}\tag{5.12}$$

Chapter 6

Random number generators

6.1   General remarks

The random number generators (RNG) are one of the crucialer points of Monte Carlo calculations, hence, giving those their "randomness". A decent multipurpose random generator covers

- reproducibility

- large period

- fast generation

- independence

of the random numbers. Therefore, special care is taken for the choice of the RNGs in WHIZARD. It is stated that WHIZARD utilizes *pseudo*-RNGs, which are based on one (or more) recursive algorithm(s) and start-seed(s) to have reproducible sequences of numbers. In contrast, a genuine random generator relies on physical processes.

WHIZARD ships with two completely different random number generators which can be selected by setting the SINDARIN option

```
$rng_method = "rng_tao"
```

Although, WHIZARD sets a default seed, it is adviced to use a different one

```
seed = 175368842
```

note that some RNGs do not allow certain seed values (e.g. zero seed).

6.2   The TAO Random Number Generator

The TAO ("The Art Of") random number generator is a lagged Fibonacci generator based upon (signed) 32-bit integer arithmetic and was proposed by Donald E. Knuth and is implemented in the VAMP package. The TAO random number generator is the default RNG of WHIZARD, but can additionally be set as SINDARIN option

```
$rng_method = rng_tao
```

The TAO random number generators is a subtractive lagged Fibonacci generator

$$x_j = (x_{j-k} - x_{j-L}) \mod 2^{30}$$

with lags $k = 100$ and $l = 37$ and period length $\rho = 2^{30} - 2$.

6.3   The RNGStream Generator

The RNGStream [97] was originally implemented in `C++` with floating point arithmetic and has been ported to `Fortran2003`. The RNGstream can be selected by the `SINDARIN` option

```
$rng_method = "rng_stream"
```

The RNGstream supports multiple independent streams and substreams of random numbers which can be directly accessed.

The main advantage of the RNGStream lies in the domain of parallelization where different worker have to access different parts of the random number stream to ensure numerical reproducibility. The RNGstream provides exactly this property with its (sub)stream-driven model.

Unfortunately, the RNGStream can only be used in combination with `VAMP2`.

Chapter 7

Integration Methods

7.1   The Monte-Carlo integration routine: `VAMP`

`VAMP` [32] is a multichannel extension of the `VEGAS` [33] algorithm. For all possible singularities in the integrand, suitable maps and integration channels are chosen which are then weighted and superimposed to build the phase space parameterization. Both grids and weights are modified in the adaption phase of the integration.

The multichannel integration algorithm is implemented as a `Fortran95` library with the task of mapping out the integrand and finding suitable parameterizations being completely delegated to the calling program (`WHIZARD` core in this case). This makes the actual `VAMP` library completely agnostic of the model under consideration.

7.2   The next generation integrator: `VAMP2`

`VAMP2` is a modern implementation of the integrator package `VAMP` written in `Fortran2003` providing the same features. The backbone integrator is still `VEGAS` [33], although implemented differently as in `VAMP`.

The main advantage over `VAMP` is the overall faster integration due to the usage of `Fortran2003`, the possible usage of different random number generators and the complete parallelization of `VEGAS` and the multichannel integration.

`VAMP2` can be set by the `SINDARIN` option

```
$integration_method = "vamp2"
```

It is said that the generated grids between `VAMP` and `VAMP2` are incompatible.

7.2.1   Multichannel integration

The usual matrix elements do not factorise with respect to their integration variables, thus making an direct integration ansatz with VEGAS unfavorable.[1] Instead, we apply the multichannel

---

[1]One prerequisite for the VEGAS algorithm is that the integral factorises, and such produces only the best results for those.

ansatz and let VEGAS integrate each channel in a factorising mapping.

The different structures of the matrix element are separated by a partition of unity and the respective mappings, such that each structure factorise at least once. We define the mappings $\phi_i : U \mapsto \Omega$, where U is the unit hypercube and Ω the physical phase space. We refer to each mapping as a *channel*. Each channel then gives rise to a probability density $g_i : U \mapsto [0, \infty)$, normalised to unity

$$\int_0^1 g_i(\phi_i^{-1}(p)) \left| \frac{\partial \phi_i^{-1}}{\partial p} \right| d\mu(p) = 1, \quad g_i(\phi_i^{-1}(p)) \geq 0,$$

written for a phase space point p using the mapping ϕ_i. The *a-priori* channel weights α_i are defined as partition of unity by $\sum_{i \in I} \alpha_i = 1$ and $0 \leq \alpha_i \leq 1$. The overall probability density g of a random sample is then obtained by

$$g(p) = \sum_{i \in I} \alpha_i g_i(\phi_i^{-1}(p)) \left| \frac{\partial \phi_i^{-1}}{\partial p} \right|,$$

which is also a non-negative and normalized probability density.

We reformulate the integral

$$I(f) = \sum_{i \in I} \alpha_i \int_\Omega g_i(\phi_i^{-1}(p)) \left| \frac{\partial \phi_i^{-1}}{\partial p} \right| \frac{f(p)}{g(p)} d\mu(p).$$

The actual integration of each channel is then done by VEGAS, which shapes the g_i.

7.2.2   VEGAS

VEGAS is an adaptive and iterative Monte Carlo algorithm for integration using importance sampling. After each iteration, VEGAS adapts the probability density g_i using information collected while sampling. For independent integration variables, the probability density factorises $g_i = \prod_{j=1}^d g_{i,j}$ for each integration axis and each (independent) $g_{i,j}$ is defined by a normalised step function

$$g_{i,j}(x_j) = \frac{1}{N \Delta x_{j,k}}, \quad x_{j,k} - \Delta x_{j,k} \leq x_j < x_{j,k},$$

where the steps are $0 = x_{j,0} < \cdots < x_{j,k} < \cdots < x_{j,N} = 1$ for each dimension j. The algorithm randomly selects for each dimension a bin and a position inside the bin and calculates the respective $g_{i,j}$.

7.2.3   Channel equivalences

The automated mulitchannel phasespace configuration can lead to a surplus of degrees of freedom, e.g. for a highly complex process with a large number of channels (VBS). In order to marginalize the redundant degrees of freedom of phasespace configuration, the adaptation distribution of the grids are aligned in accordance to their phasespace relation, hence the binning of the grids is equialized. These equivalences are activated by default for `VAMP` and `VAMP2`, but can be steered by:

```
?use_vamp_equivalences = true
```

Chapter 8

Phase space parameterizations

8.1 General remarks

WHIZARD as a default performs an adaptive multi-channel Monte-Carlo integration. Besides its default phase space algorithm, wood, to be detailed in Sec. 8.3, WHIZARD contains a phase space method phs_none which is a dummy method that is intended for setups of processes where no phase space integration is needed, but the program flow needs a (dummy) integrator for internal consistency. Then, for testing purposes, there is a single-channel phase space integrator, phs_single. From version 2.6.0 of WHIZARD on, there is also a second implementation of the wood phase space algorithm, called fast_wood, cf. Sec. 8.4, whose implementation differs technically and which therefore solves certain technical flaws of the wood implementation. Additionally, WHIZARD supports single-channel, flat phase-space using RAMBO (on diet).

8.2 The flat method: `rambo`

The RAMBO algorithm produces a flat phase-space with constant volume for massless particles. RAMBO was originally published in [100]. We use the slim version, called RAMBO on diet, published in [98]. The overall weighting efficiency of the algorithm is unity for massless final-state particles. For the massive case, the weighting efficiency of unity will decrease rendering the algorithm less efficient. But in most cases, the invariants are in regions of phase space where they are much larger than the masses of the final-state particles.

We provide the RAMBO mainly for cross checking our implementation and do not recommend it for real world application, even though it can be used as one. The RAMBO method becomes useful as a fall-back option if the standard algorithm fails for physical reasons, see, e.g., Sec. 8.6.

8.3 The default method: `wood`

The wood algorithm classifies different phase space channels according to their importance for a full scattering or decay process following heuristic rules. For that purpose, WHIZARD investigates

the kinematics of the different channels depending on the total center-of-mass energy (or the mass of the decaying particle) and the masses of the final-state particles.

The `wood` phase space inherits its name from the naming schemes of structures of increasing complexities, namely trees, forests and groves. Simply stated, a phase-space forest is a collection of phase-space trees. A phase-space tree is a parameterization for a valid channel in the multi-channel adaptive integration, and each variable in the a tree corresponds to an integration dimension, defined by an appropriate mapping of the $(0, 1)$ interval of the unit hypercube to the allowed range of the corresponding integration variable. The whole set of these phase-space trees, collected in a phase-space forest object hence contains all parameterizations of the phase space that `WHIZARD` will use for a single hard process. Note that processes might contain flavor sums of particles in the final state. As `WHIZARD` will use the same phase space parameterization for all channels for this set of subprocesses, all particles in those flavor sums have to have the same mass. E.g. in the definition of a "light" jet consisting of the first five quarks and antiquarks,

```
alias jet = u:d:s:c:b:U:D:S:C:B
```

all quarks including strange, charm and bottom have to be massless for the phase-space integration. `WHIZARD` can treat processes with subprocesses having final-state particles with different masses in an "additive" way, where each subprocess will become a distinct component of the whole process. Each process component will get its own phase-space parameterization, such that they can allow for different masses. E.g. in a 4-flavor scheme for massless u, d, s, c quarks one can write

```
alias jet = u:d:s:c:U:D:S:C
process eeqq = e1, E1 => (jet, jet) + (b, B)
```

In that case, the parameterizations will be for massless final state quarks for the first subprocess, and for massive b quarks for the second subprocess. In general, for high-energy lepton colliders, the difference would not matter much, but performing the integration e.g. for $\sqrt{s} = 11$ GeV, the difference will be tremendous. `WHIZARD` avoids inconsistent phase-space parameterizations in that way.

As a multi-particle process will contain hundred or thousands of different channels, the different integration channels (trees) are grouped into so called *groves*. All channels/trees in the same grove share a common weight for the phase-space integration, following the assumption that they are related by some approximate symmetry. The `VAMP` adaptive multi-channel integrator (cf. Sec. 7.1) allows for equivalences between different integration channels. This means that trees/channels that are related by an exact symmetry are connected by an array of these equivalences.

The phase-space setup, i.e. the detailed structure of trees and forests, are written by `WHIZARD` into a phase-space file that has the same name as the corresponding process (or process component) with the suffix `.phs`. For the `wood` phase-space method this file is written by a `Fortran` module which constructs a similar tree-like structure as the directed acyclical graphs (DAGs) in the `O'Mega` matrix element generator but in a less efficient way.

In some very rare cases with externally generated models (cf. Chapter 17) the phase-space generation has been reported to fail as `WHIZARD` could not find a valid phase-space channel. Such pathological cases cannot occur for the hard-coded model implementations inside `WHIZARD`. They can only happen if there are in principle two different Feynman diagrams contributing to the same phase-space channel and `WHIZARD` considers the second one as extremely subleading

(and would hence drop it). If for some reason however the first Feynman diagram is then absent, no phase-space channel could be found. This problem cannot occur with the `fast_wood` implementation discussed in the next section, cf. 8.4.

The `wood` algorithms orders the different groves of phase-space channels according to a heuristic importance depending on the kinematic properties of the different phase-space channels in the groves. A phase-space (`.phs`) file looks typically like this:

```
process sm_i1

! List of subprocesses with particle bincodes:
!  8  4      1   2
! e+ e- => mu+ mu-
!  8  4      1   2

  md5sum_process    = "1B3B7A30C24664A73D3D027382CFB4EF"
  md5sum_model_par  = "7656C90A0B2C4325AD911301DACF50EB"
  md5sum_phs_config = "6F72D447E8960F50FDE4AE590AD7044B"
  sqrts         =   1.000000000000E+02
  m_threshold_s =   5.000000000000E+01
  m_threshold_t =   1.000000000000E+02
  off_shell = 2
  t_channel = 6
  keep_nonresonant = T

! Multiplicity = 2, no resonances,  0 logs,  0 off-shell,  s-channel graph
grove #1
! Channel #1
  tree  3

! Multiplicity = 1, 1 resonance,   0 logs,  0 off-shell,  s-channel graph
grove #2
! Channel #2
  tree  3
  map   3 s_channel      23 ! Z
```

The first line contains the process name, followed by a list of subprocesses with the external particles and their binary codes. Then there are three lines of MD5 check sums, used for consistency checks. WHIZARD (unless told otherwise) will check for the existence of a phase-space file, and if the check sum matches, it will reuse the existing file and not generate it again. Next, there are several kinematic parameters, namely the center-of-mass energy of the process, `sqrts`, and two mass thresholds, `m_threshold_s` and `m_threshold_t`. The latter two are kinematical thresholds, below which WHIZARD will consider s-channel and t-channel-like kinematic configurations as effectively massless, respectively. The default values shown in the example have turned out to be optimal values for Standard Model particles. The two integers `off_shell` and `t_channel` give the number of off-shell lines and of t-channel lines that WHIZARD will allow for finding valid phase-space channels, respectively. This neglects extremley multi-peripheral background-like diagram constellations which are very subdominamnt compared to resonant signal processes. The final flag specifies whether WHIZARD will keep non-resonant phase-space channels (default), or whether it will focus only on resonant situations.

After this header, there is a list of all groves, i.e. collections of phase-space channels which are connected by quasi-symmetries, together with the corresponding multiplicity of subchannels in that grove. In the phase-space file behind the multiplicity, WHIZARD denotes the number of (massive) resonances, logarithmcally enhanced kinematics (e.g. collinear regions), and number of off-shell lines, respectively. The final entry in the grove header notifies whether the diagrams in that grove have *s*-channel topologies, or count the number of corresponding *t*-channel lines.

Another example is shown here,

```
! Multiplicity = 3, no resonances,  2 logs,  0 off-shell,  1 t-channel line
grove #1
! Channel #1
  tree  3 12
  map   3 infrared        22 ! A
  map  12 t_channel        2 ! u
! Channel #2
  tree  3 11
  map   3 infrared        22 ! A
  map  11 t_channel        2 ! u
! Channel #3
  tree  3 20
  map   3 infrared        22 ! A
  map  20 t_channel        2 ! u
! Channel #4
  tree  3 19
  map   3 infrared        22 ! A
  map  19 t_channel        2 ! u
```

where WHIZARD notifies in different situations a photon exchange as infrared. So it detects a possible infrared singularity where a particle can become arbitrarily soft. Such a situation can tell the user that there might be a cut necessary in order to get a meaningful integration result.

The phase-space setup that is generated and used by the wood phase-space method can be visualized using the SINDARIN option

```
?vis_channels = true
```

The wood phase-space method can be invoked with the SINDARIN command

```
$phs_method = "wood"
```

Note that this line is unnecessary, as wood is the default phase-space method of WHIZARD.

8.4   A new method: fast_wood

This method (which is available from version 2.6.0 on) is an alternative implementation of the wood phase-space algorithm. It uses the recursive structures inside the O'Mega matrix element generator to generate all the structures needed for the different phase-space channels. In that way, it can avoid some of the bottlenecks of the wood Fortran implementation of the algorithm. On the other hand, it is only available if the O'Mega matrix element generator has been enabled (which is the default for WHIZARD). The fast_wood method is then invoked via

```
?omega_write_phs_output = true
$phs_method = "fast_wood"
```

The first option is necessary in order to tell `O'Mega` to write out the output needed for the `fast_wood` parser in order to generate the phase-space file. This is not enabled by default in order not to generate unnecessary files in case the default method `wood` is used.

So the `fast_wood` implementation of the `wood` phase-space algorithm parses the tree-like represenation of the recursive set of one-particle off-shell wave functions that make up the whole amplitude inside `O'Mega` in the form of a directed acyclical graph (DAG) in order to generate the phase-space (`.phs`) file (cf. Sec. 8.3). In that way, the algorithm makes sure that only phase-space channels are generated for which there are indeed (sub)amplitudes in the matrix elements, and this also allows to exclude vetoed channels due to restrictions imposed on the matrix elements from the phase-space setup (cf. next Sec. 8.5).

8.5 Phase space respecting restrictions on subdiagrams

The `Fortran` implementation of the `wood` phase-space does not know anything about possible restrictions that maybe imposed on the `O'Mega` matrix elements, cf. Sec. 5.4.3. Consequently, the `wood` phase space also generates phase-space channels that might be absent when restrictions are imposed. This is not a principal problem, as in the adaptation of the phase-space channels `WHIZARD`'s integrator `VAMP` will recognize that there is zero weight in that channel and will drop the channel (stop sampling in that channel) after some iterations. However, this is a waste of ressources as it is in principle known that this channel is absent. Using the `fast_wood` phase-space algorithm (cf. Sec. 8.4 will take restrictions into account, as `O'Mega` will not generate trees for channels that are removed with the restrictions command. So it advisable for the user in the case of very complicated processes with restrictions to use the `fast_wood` phase-space method to make `WHIZARD` generation and integration of the phase space less cumbersome.

8.6 Phase space for processes forbidden at tree level

The phase-space generators `wood` and `fast_wood` are intended for tree-level processes with their typical patterns of singularities, which can be read off from Feynman graphs. They can and should be used for loop-induced or for externally provided matrix elements as long as `WHIZARD` does not provide a dedicated phase-space module.

Some scattering processes do not occur at tree level but become allowed if loop effects are included in the calculation. A simple example is the elastic QED process

$$A \quad A \longrightarrow A \quad A$$

which is mediated by a fermion loop. Similarly, certain applications provide externally provided or hand-taylored matrix-element code that replaces the standard `O'Mega` code.

Currently, `WHIZARD`'s phase-space parameterization is nevertheless tied to the `O'Mega` generator, so for tree-level forbidden processes the phase-space construction process will fail.

There are two possible solutions for this problem:

1. It is possible to provide the phase-space parameterization information externally, by sup-
   plying an appropriately formatted `.phs` file, bypassing the automatic algorithm. Assuming
   that this phase-space file has been named `my_phase_space.phs`, the SINDARIN code should
   contain the following:

   ```
   ?rebuild_phase_space = false
   $phs_file = "my_phase_space.phs"
   ```

   Regarding the contents of this file, we recommend to generate an appropriate `.phs` for a
   similar setup, using the standard algorithm. The generated file can serve as a template,
   which can be adapted to the particular case.

   In detail, the `.phs` file consists of entries that specify the process, then a standard header
   which contains MD5 sums and such – these variables must be present but their values
   are irrelevant for the present case –, and finally at least one `grove` with `tree` entries that
   specify the parameterization. Individual parameterizations are built from the final-state
   and initial-state momenta (in this order) which we label in binary form as $1, 2, 4, 8, \ldots$.
   The actual tree consists of iterative fusions of those external lines. Each fusion is indicated
   by the number that results from adding the binary codes of the external momenta that
   contribute to it.

   For instance, a valid phase-space tree for the process $AA \to AA$ is given by the simple
   entry

   ```
   tree 3
   ```

   which indicates that the final-state momenta 1 and 2 are combined to a fusion $1 + 2 = 3$.
   The setup is identical to a process such as $e^+e^- \to \mu^+\mu^-$ below the Z threshold. Hence,
   we can take the `.phs` file for the latter process, replace the process tag, and use it as an
   external phase-space file.

2. For realistic applications of WHIZARD together with one-loop matrix-element providers,
   the actual number of final-state particles may be rather small, say $2, 3, 4$. Furthermore,
   one-loop processes which are forbidden at tree level do not contain soft or collinear
   singularities. In this situation, the RAMBO phase-space integration method, cf. Sec. 8.2 is a
   viable alternative which does not suffer from the problem.

Chapter 9

Methods for Hard Interactions

The hard interaction process is the core of any physics simulation within an MC event generator. One tries to describe the dominant particle interaction in the physics process of interest at a given order in perturbation theory, thereby making use of field-theoretic factorization theorems, especially for QCD, in order to separate non-perturbative physics like parton distribution functions (PDFs) or fragmentation functions from the perturbative part. Still, it is in many cases not possible to describe the perturbative part completely by means of fixed-order hard matrix elements: in soft and/or collinear regions of phase space, multiple emission of gluons and quarks (in general QCD jets) and photons necessitates a resummation, as large logarithms accompany the perturbative coupling constants and render fixed-order perturbation theory unreliable. The resummation of these large logarithms can be done analytically or (semi-)numerically, however, usually only for very inclusive quantities. At the level of exclusive events, these phase space regions are the realm of (QCD and also QED) parton showers that approximate multi-leg matrix elements from the hard perturbative into to the soft-/collinear regime.

The hard matrix elements are then the core building blocks of the physics description inside the MC event generator. `WHIZARD` generates these hard matrix elements at tree-level (or sometimes for loop-induced processes using effective operators as insertions) as leading-order processes. This is done by the `O'Mega` subpackage that is automatically called by `WHIZARD`. Besides these physical matrix elements, there exist a couple of methods to generate dummy matrix elements for testing purposes, or for generating beam profiles and using them with externally linked special matrix elements.

Especially for one-loop processes (next-to-leading order for tree-allowed processes or leading-order for loop-induced processes), `WHIZARD` allows to use matrix elements from external providers, so called OLP programs (one-loop providers). Of course, all of these external packages can also generate tree-level matrix elements, which can then be used as well in `WHIZARD`.

We start the discussion with the two different options for test matrix elements, internal test matrix elements with no generated compiled code in Sec. 9.1 and so called template matrix elements with actual `Fortran` code that is compiled and linked, and can also be modified by the user in Sec. 9.2. Then, we move to the main matrix element method by the matrix element generator `O'Mega` in Sec. 9.3. Matrix elements from the external matrix element generators are

discussed in the order of which interfaces for the external tools have been implemented: `Gosam` in Sec. 9.4, `OpenLoops` in Sec. 9.5, and `Recola` in Sec. 9.6.

9.1   Internal test matrix elements

This method is merely for internal consistency checks inside `WHIZARD`, and is not really intended to be utilized by the user. The method is invoked by

```
$method = "unit_test"
```

This particular method is only applicable for the internal test model `Test.mdl`, which just contains a Higgs boson and a top quark. Technically, it will also works within model specifications for the Standard Model, or the Minimal Supersymmetric Standard Model (MSSM), or all models which contain particles named as `H` and `t` with PDG codes 25 and 6, respectively. So, the models `QED` and `QCD` will not work. Irrespective of what is given in the `SINDARIN` file as a scattering input process, `WHIZARD` will always take the process

```
model = SM
process <proc_name>= H, H => H, H
```

or for the test model:

```
model = Test
process <proc_name>= s, s => s, s
```

as corresponding process. (This is the same process, just with differing nomenclature in the different models). No matrix element code is generated and compiled, the matrix element is completely internal, included in the `WHIZARD` executable (or library), with a unit value for the squared amplitude. The integration will always be performed for this particularly process, even if the user provides a different process for that method. Hence, the result will always be the volume of the relativistic two-particle phase space. The only two parameters that influence the result are the collider energy, `sqrts`, and the mass of the Higgs particle with PDG code 25 (this mass parameter can be changed in the model `Test` as `ms`, while it would be `mH` in the Standard Model `SM`.

It is also possible to use a test matrix element, again internal, for decay processes, where again `WHIZARD` will take a predefined process:

```
model = SM
process <proc_name> = H => t, tbar
```

in the `SM` model or

```
model = Test
process <proc_name> = s => f, fbar
```

Again, this is the same process with PDG codes $25 \to 6\ -6$ in the corresponding models. Note that in the model `SM` the mass of the quark is set via the variable `mtop`, while it is `mf` in the model `Test`.

Besides the fact that the user always gets a fixed process and cannot modify any matrix element code by hand, one can do all things as for a normal process like generating events,

different weights, testing rebuild flags, using different setups and reweight events accordingly. Also factorized processes with production and decay can be tested that way.

In order to avoid confusion, it is highly recommended to use this method `unit_test` only with the test model setup, model `Test`.

On the technical side, the method `unit_test` does not produce a process library (at least not an externally linked one), and also not a makefile in order to modify any process files (which anyways do not exist for that method). Except for the logfiles and the phase space file, all files are internal.

9.2 Template matrix elements

Much more versatile for the user than the previous matrix element method in 9.1, are two different methods with constant template matrix elements. These are written out as `Fortran` code by the `WHIZARD` main executable (or library), providing an interface that is (almost) identical to the matrix element code produced by the `O'Mega` generator (cf. the next section, Sec. 9.3. There are actually two different methods for that purpose, providing matrix elements with different normalizations:

        $method = "template"

generates matrix elements which give after integration over phase space exactly one. Of course, for multi-particle final states the integration can fluctuate numerically and could then give numbers that are only close to one but not exactly one. Furthermore, the normalization is not exact if any of the external particles have non-zero masses, or there are any cuts involved. But otherwise, the integral from `WHIZARD` should give unity irrespective of the number of final state particles.

In contrast to this, the second method,

        $method = "template_unity"

gives a unit matrix elements, or rather a matrix element that contains helicity and color averaging factors for the initial state and the square root of the factorials of identical final state particles in the denominator. Hence, integration over the final state momentum configuration gives a cross section that corresponds to the volume of the n-particle final state phase space, divided by the corresponding flux factor, resulting in

$$\sigma(s, 2 \to 2, 0) = \frac{3.8937966 \cdot 10^{11}}{16\pi} \cdot \frac{1}{s[\text{GeV}]^2} \text{ fb} \tag{9.1}$$

for the massless case and

$$\sigma(s, 2 \to 2, m_i) = \frac{3.8937966 \cdot 10^{11}}{16\pi} \cdot \sqrt{\frac{\lambda(s, m_3^2, m_4^2)}{\lambda(s, m_1^2, m_2^2)}} \cdot \frac{1}{s[\text{GeV}]^2} \text{ fb} \tag{9.2}$$

for the massive case. Here, m_1 and m_2 are the masses of the incoming, m_3 and m_4 the masses of the outgoing particles, and $\lambda(x, y, z) = x^2 + y^2 + z^2 - 2xy - 2xz - 2yz$.

For the general massless case with no cuts, the integral should be exactly

$$\sigma(s, 2 \to n, 0) = \frac{(2\pi)^4}{2s} \Phi_n(s) = \frac{1}{16\pi s} \frac{\Phi_n(s)}{\Phi_2(s)}, \tag{9.3}$$

where the volume of the massless n-particle phase space is given by

$$\Phi_n(s) = \frac{1}{4(2\pi)^5} \left(\frac{s}{16\pi^2}\right)^{n-2} \frac{1}{(n-1)!(n-2)!}. \tag{9.4}$$

For $n \neq 2$ the phase space volume is dimensionful, so the units of the integral are fb $\times$ GeV$^{2(n-2)}$. (Note that for physical matrix elements this is compensated by momentum factors from wave functions, propagators, vertices and possibly dimensionful coupling constants, but here the matrix element is just equal to unity.)

Note that the phase-space integration for the `template` and `template_unity` matrix element methods is organized in the same way as it would be for the real $2 \to n$ process. Since such a phase space parameterization is not optimized for the constant matrix element that is supplied instead, good convergence is not guaranteed. (Setting `?stratified = true` may be helpful here.)

The possibility to call a dummy matrix element with this method allows to histogram spectra or structure functions: Choose a trivial process such as $uu \to dd$, select the `template_unity` method, switch on structure functions for one (or both) beams, and generate events. The distribution of the final-state mass squared reflects the x dependence of the selected structure function.

Furthermore, the constant in the source code of the unit matrix elements can be easily modified by the user with their `Fortran` code in order to study customized matrix elements. Just rerun `WHIZARD` with the `--recompile` option after the modification of the matrix element code.

Both methods, `template` and `template_unity` will also work even if no `OCaml` compiler is found or used and consequently the `O'Mega` matrix elemente generator (cf. Sec. 9.3 is disable. The methods produce a process library for their corresponding processes, and a makefile, by which `WHIZARD` steers compilation and linking of the process source code.

9.3   The O'Mega matrix elements

`O'Mega` is a subpackage of `WHIZARD`, written in `OCaml`, which can produce matrix elements for a wide class of implemented physics models (cf. Sec. 10.1.1 and 10.1.2 for a list of all implemented physics models), and even almost arbitrary models when using external Lagrange level tools, cf. Chap. 17. There are two different variants for matrix elements from `O'Mega`: the first one is invoked as

```
$method = "omega"
```

and is the default method for `WHIZARD`. It produces matrix element as `Fortran` code which is then compiled and linked. An alternative method, which for the moment is only available for the Standard Model and its variants as well models which are quite similar to the SM, e.g. the Two-Higgs doublet model or the Higgs-singlet extension. This method is taken when setting

```
$method = "ovm"
```

The acronym `ovm` stands for O'Mega Virtual Machine (OVM). The first (default) method (`omega`) of O'Mega matrix elements produces `Fortran` code for the matrix elements,that is compiled by the same compiler with which `WHIZARD` has been compiled. The OVM method (`ovm`) generates an `ASCII` file with so called op code for operations. These are just numbers which tell what numerical operations are to be performed on momenta, wave functions and vertex expression in order to yield a complex number for the amplitude. The op codes are interpreted by the OVM in the same as a Java Virtual Machine. In both cases, a compiled `Fortran` is generated which for the `omega` method contains the full expression for the matrix element as `Fortran` code, while for the `ovm` method this is the driver file of the OVM. Hence, for the `ovm` method this file always has roughly the same size irrespective of the complexity of the process. For the `ovm` method, there will also be the `ASCII` file that contains the op codes, which has a name with an `.hbc` suffix: `<process_name>.hbc`.

For both O'Mega methods, there will be a process library created as for the template matrix elements (cf. Sec. 9.2) named `default_lib.f90` which can be given a user-defined name using the `library = "<library>"` command. Again, for both methods `omega` and `ovm`, a makefile named `<library>_lib.makefile` is generated by which `WHIZARD` steers compilation, linking and clean-up of the process sources. This makefile can handily be adapted by the user in case she or he wants to modify the source code for the process (in the case of the source code method).

Note that `WHIZARD`'s default ME method via O'Mega allows the user to specify many different options either globally for all processes in the `SINDARIN`, or locally for each process separately in curly brackets behind the corresponding process definition. Examples are

- Restrictions for the matrix elements like the exclusion of intermediate resonances, the appearance of specific vertices or coupling constants in the matrix elments. For more details on this cf. Sec. 5.4.3.

- Choice of a specific scheme for the width of massive intermediate resonances, whether to use constant width, widths only in s-channel like kinematics (this is the default), a fudged-width scheme or the complex-mass scheme. The latter is actually steered as a specific scheme of the underlying model and not with a specific O'Mega command.

- Choice of the electroweak gauge for the amplitude. The default is the unitary gauge.

With the exception of the restrictions steered by the `$restrictions = "<restriction>"` string expression, these options have to be set in their specific O'Mega syntax verbatim via the string command `$omega_flags = "<expr>"`.

9.4   Interface to GoSam

One of the supported methods for automated matrix elements from external providers is for the `Gosam` package. This program package which is a combination of `Python` scripts and `Fortran` libraries, allows both for tree and one-loop matrix elements (which is leading or next-to-leading order, depending on whether the corresponding process is allowed at the tree level or not). In

principle, the advanced version of `Gosam` also allows for the evaluation of two-loop virtual matrix elements, however, this is currently not supported in `WHIZARD`. This method is invoked via the command

```
$method = "gosam"
```

Of course, this will only work correctly of `Gosam` with all its subcomponents has been correctly found during configuration of `WHIZARD` and then subsequently correctly linked.

In order to generate the tables for spin, flavor and color states for the corresponding process, first `O'Mega` is called to provide `Fortran` code for the interfaces to all the metadata for the process(es) to be evaluated. Next, the `Gosam Python` script is automatically invoked that first checks for the necessary ingredients to produce, compile and link the `Gosam` matrix elements. These are the the `Qgraf` topology generator for the diagrams, `Form` to perform algebra, the `Samurai`, `AVHLoop`, `QCDLoop` and `Ninja` libraries for Passarino-Veltman reduction, one-loop tensor integrals etc. As a next step, `Gosam` automatically writes and executes a `configure` script, and then it exchanges the Binoth Les Houches accord (BLHA) contract files between `WHIZARD` and itself [37,38] to check whether it actually generate code for the demanded process at the given order. Note that the contract and answer files do not have to be written by the user by hand, but are generated automatically within the program work flow initiated by the original `SINDARIN` script. `Gosam` then generates `Fortran` code for the different components of the processes, compiles it and links it into a library, which is then automatically accessible (as an external process library) from inside `WHIZARD`. The phase space setup and the integration as well as the LO (and NLO) event generation work then in exactly the same way as for `O'Mega` matrix elements.

As an NLO calculation consists of different components for the Born, the real correction, the virtual correction, the subtraction part and possible further components depending on the details of the calculation, there is the possible to separately choose the matrix element method for those components via the keywords `$loop_me_method`, `$real_tree_me_method`, `$correlation_me_method` etc. These keywords overwrite the master switch of the `$method` keyword.

For more information on the switches and details of the functionality of `Gosam`, cf. http://gosam.hepforge.org.

9.5 Interface to Openloops

Very similar to the case of `Gosam`, cf. Sec. 9.4, is the case for `OpenLoops` matrix elements. Also here, first `O'Mega` is called in order to provide an interface for the spin, flavor and color degrees of freedom for the corresponding process. Information exchange between `WHIZARD` and `OpenLoops` then works in the same automatic way as for `Gosam` via the BLHA interface. This matrix element method is invoked via

```
$method = "openloops"
```

This again is the master switch that will tell `WHIZARD` to use `OpenLoops` for all components, while there are special keywords to tailor-make the setup for the different components of an NLO calculation (cf. Sec. 9.4.

The main difference between `OpenLoops` and `Gosam` is that for `OpenLoops` there is no process code to be generated, compiled and linked for a process, but a precompiled library is called and linked, e.g. `ppllj` for the Drell-Yan process. Of course, this library has to be installed on the system, but if that is not the case, the user can execute the `OpenLoops` script in the source directory of `OpenLoops` to download, compile and link the corresponding dynamic library. This limits (for the moment) the usage of `OpenLoops` to processes where pre-existint libraries for that specific processes have been generated by the `OpenLoops` authors. A new improved generator for general process libraries for `OpenLoops` will get rid of that restriction.

For more information on the installation, switches and details of the functionality of `OpenLoops`, cf. http://openloops.hepforge.org.

9.6 Interface to Recola

The third one-loop provider (OLP) for external matrix elements that is supported by `WHIZARD`, is `Recola`. In contrast to `Gosam`, cf. Sec. 9.4, and `OpenLoops`, cf. Sec. 9.5, `Recola` does not use a BLHA interface to exchange information with `WHIZARD`, but its own tailor-made C interoperable library interface to communicate to the Monte Carlo side. `Recola` matrix elements are called for via

```
$method = "recola"
```

`Recola` uses a highly efficient algorithm to generate process code for LO and NLO SM amplitudes in a fully recursive manner. At the moment, the setup of the interface within `WHIZARD` does not allow to invoke more than one different process in `Recola`: this would lead to a repeated initialization of the main setup of `Recola` and would consequently crash it. It is foreseen in the future to have a safeguard mechanism inside `WHIZARD` in order to guarantee initialization of `Recola` only once, but this is not yet implemented.

Further information on the installation, details and parameters of `Recola` can be found at http://recola.hepforge.org.

9.7 Special applications

There are also special applications with combinations of matrix elements from different sources for dedicated purposes like e.g. for the matched top–anti-top threshold in e^+e^-. For this special application which depending on the order of the matching takes only `O'Mega` matrix elements or at NLO combines amplitudes from `O'Mega` and `OpenLoops`, is invoked by the method:

```
$method = "threshold"
```

Chapter 10

Implemented physics

10.1 The hard interaction models

In this section, we give a brief overview over the different incarnations of models for the description of the realm of subatomic particles and their interactions inside `WHIZARD`. In Sec. 10.1.1, the Standard Model (SM) itself and straightforward extensions and modifications thereof in the gauge, fermionic and Higgs sector are described. Then, Sec. 10.1.2 gives a list and short description of all genuine beyond the SM models (BSM) that are currently implemented in `WHIZARD` and its matrix element generator `O'Mega`. Additional models beyond that can be integrated and handled via the interfaces to external tools like `SARAH` and `FeynRules`, or the universal model format `UFO`, cf. Chap. 17.

10.1.1 The Standard Model and friends

10.1.2 Beyond the Standard Model

Strongly Interacting Models and Composite Models

Higgsless models have been studied extensively before the Higgs boson discovery at the LHC Run I in 2012 in order to detect possible loopholes in the electroweak Higgs sector discovery potential of this collider. The Threesite Higgsless Model is one of the simplest incarnations of these models, and was one of the first BSM models beyond SUSY and Little Higgs models that have been implemented in `WHIZARD` [39]. It is also called the Minimal Higgsless Model (MHM) [40] is a minimal deconstructed Higgsless model which contains only the first resonance in the tower of Kaluza-Klein modes of a Higgsless extra-dimensional model. It is a non-renormalizable, effective theory whose gauge group is an extension of the SM with an extra $SU(2)$ gauge group. The breaking of the extended electroweak gauge symmetry is accomplished by a set of nonlinear sigma fields which represent the effects of physics at a higher scale and make the theory nonrenormalizable. The physical vector boson spectrum contains the usual photon, $W^\pm$ and Z bosons as well as a $W'^\pm$ and Z' boson. Additionally, a new set of heavy fermions are introduced to accompany the new gauge group "site" which mix to form the physical eigenstates. This mixing is controlled by the small mixing parameter ϵ_L which is adjusted to satisfy constraints

MODEL TYPE	with CKM matrix	trivial CKM
Yukawa test model	---	`Test`
QED with e, μ, τ, γ	---	`QED`
QCD with d, u, s, c, b, t, g	---	`QCD`
Standard Model	`SM_CKM`	`SM`
SM with anomalous gauge couplings	`SM_ac_CKM`	`SM_ac`
SM with Hgg, $H\gamma\gamma$, $H\mu\mu$, He^+e^-	`SM_Higgs_CKM`	`SM_Higgs`
SM with bosonic dim-6 operators	---	`SM_dim6`
SM with charge 4/3 top	---	`SM_top`
SM with anomalous top couplings	---	`SM_top_anom`
SM with anomalous Higgs couplings	---	`SM_rx/NoH_rx/SM_ul`
SM extensions for VV scattering	---	`SSC/AltH/SSC_2/SSC_AltT`
SM with Z'	---	`Zprime`
Two-Higgs Doublet Model	`THDM_CKM`	`THDM`
MSSM	`MSSM_CKM`	`MSSM`
MSSM with gravitinos	---	`MSSM_Grav`
NMSSM	`NMSSM_CKM`	`NMSSM`
extended SUSY models	---	`PSSSM`
Littlest Higgs	---	`Littlest`
Littlest Higgs with ungauged $U(1)$	---	`Littlest_Eta`
Littlest Higgs with T parity	---	`Littlest_Tpar`
Simplest Little Higgs (anomaly-free)	---	`Simplest`
Simplest Little Higgs (universal)	---	`Simplest_univ`
SM with graviton	---	`Xdim`
UED	---	`UED`
"SQED" with gravitino	---	`GravTest`
Augmentable SM template	---	`Template`

Table 10.1:   *List of models available in* `WHIZARD`*. There are pure test models or models implemented for theoretical investigations, a long list of SM variants as well as a large number of BSM models.*

from precision observables, such as the S parameter [41]. Here, additional weak gauge boson production at the LHC was one of the focus of the studies with `WHIZARD` [42].

Supersymmetric Models

`WHIZARD`/`O'Mega` was the first multi-leg matrix-element/event generator to include the full Minimal Supersymmetric Standard Model (MSSM), and also the NMSSM. The SUSY implementations in `WHIZARD` have been extensively tested [43,44], and have been used for many theoretical and experimental studies (some prime examples being [45,46,56].

Little Higgs Models

Inofficial models

There have been several models that have been included within the `WHIZARD`/`O'Mega` framework but never found their way into the official release series. One famous example is the non-commutative extension of the SM, the NCSM. There have been several studies, e.g. simulations on the s-channel production of a Z boson at the photon collider option of the ILC [49]. Also, the production of electroweak gauge bosons at the LHC in the framework of the NCSM have been studied [50].

10.2 The SUSY Les Houches Accord (SLHA) interface

To be filled in ... [52,53,54].

The neutralino sector deserves special attention. After diagonalization of the mass matrix expresssed in terms of the gaugino and higgsino eigenstates, the resulting mass eigenvalues may be either negative or positive. In this case, two procedures can be followed. Either the masses are rendered positive and the associated mixing matrix gets purely imaginary entries or the masses are kept signed, the mixing matrix in this case being real. According to the SLHA agreement, the second option is adopted. For a specific eigenvalue, the phase is absorbed into the definition of the relevant eigenvector, rendering the mass negative. However, `WHIZARD` has not yet officially tested for negative masses. For external SUSY models (cf. Chap. 17) this means, that one must be careful using a SLHA file with explicit factors of the complex unity in the mixing matrix, and on the other hand, real and positive masses for the neutralinos. For the hard-coded SUSY models, this is completely handled internally. Especially Ref. [56] discusses the details of the neutralino (and chargino) mixing matrix.

10.3 Lepton Collider Beam Spectra

For the simulation of lepton collider beam spectra there are two dedicated tools, `CIRCE1` and `CIRCE2` that have been written as in principle independent tools. Both attempt to describe the details of electron (and positron) beams in a realistic lepton collider environment. Due to the quest for achieving high peak luminosities at e^+e^- machines, the goal is to make the spatial

extension of the beam as small as possible but keeping the area of the beam roughly constant. This is achieved by forcing the beams in the final focus into the shape of a quasi-2D bunch. Due to the high charge density in that bunch, the bunch electron distribution is modified by classical electromagnetic radiation, so called *beamstrahlung*. The two `CIRCE` packages are intended to perform a simulation of this beamstrahlung and its consequences on the electron beam spectrum as realistic as possible. More details about the two packages can be found in their stand-alone documentations. We will discuss the basic features of lepton-collider beam simulations in the next two sections, including the technicalities of passing simulations of the machine beam setup to `WHIZARD`. This will be followed by a section on the simulation of photon collider spectra, included for historical reasons.

10.3.1   CIRCE1

While the bunches in a linear collider cross only once, due to their small size they experience a strong beam-beam effect. There is a code to simulate the impact of this effect on luminosity and background, called `GuineaPig++` [10,11,12]. This takes into account the details of the accelerator, the final focus etc. on the structure of the beam and the main features of the resulting energy spectrum of the electrons and positrons. It offers the state-of-the-art simulation of lepton-collider beam spectra as close as possible to reality. However, for many high-luminosity simulations, event files produced with `GuineaPig++` are usually too small, in the sense that not enough independent events are available for physics simulations. Lepton collider beam spectra do peak at the nominal beam energy ($\sqrt{s}/2$) of the collider, and feature very steeply falling tails. Such steeply falling distributions are very poorly mapped by histogrammed distributions with fixed bin widths.

The main working assumption to handle such spectra are being followed within `CIRCE1`:

1. The beam spectra for the two beams P_1 and P_2 factorize (here x_1 and x_2 are the energy fractions of the two beams, respectively):

$$D_{P_1 P_2}(x_1, x_2) = D_{P_1}(x_1) \cdot D_{P_2}(x_2)$$

2. The peak is described with a delta distribution, and the tail with a power law:

$$D(x) = d \cdot \delta(1 - x) \; + \; c \cdot x^\alpha \, (1 - x)^\beta$$

The two powers α and β are the main coefficients that can be tuned in order to describe the spectrum with `CIRCE1` as close as possible as the original `GuineaPig++` spectrum. More details about how `CIRCE1` works and what it does can be found in its own write-up in `circe1/share/doc`.

10.3.2   CIRCE2

The two conditions listed in 10.3.1 are too restrictive and hence insufficient to describe more complicated lepton-collider beam spectra, as they e.g. occur in the CLIC drive-beam design. Here, the two beams are highly correlated and also a power-law description does not give good
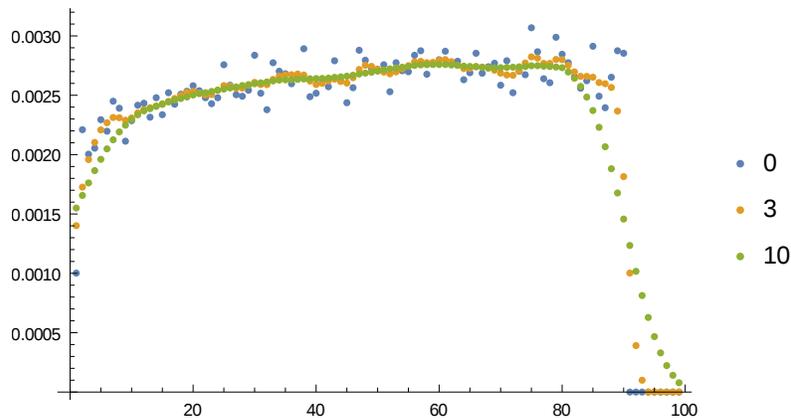
Figure 10.1: *Smoothing the bin at the $x_{e^+} = 1$ boundary with Gaussian filters of 3 and 10 bins width compared to no smoothing.*

enough precision for the tails. To deal with these problems, `CIRCE2` starts with a two-dimensional histogram featuring factorized, but variable bin widths in order to simulate the steep parts of the distributions. The limited statistics from too small `GuineaPig++` event output files leads to correlated fluctuations that would leave strange artifacts in the distributions. To abandon them, Gaussian filters are applied to smooth out the correlated fluctuations. Here care has to be taken when going from the continuum in x momentum fraction space to the corresponding boundaries: separate smoothing procedures are being applied to the bins in the continuum region and those in the boundary in order to avoid artificial unphysical beam energy spreads. Fig. 10.1 shows the smoothing of the distribution for the bin at the $x_{e^+} = 1$ boundary. The blue dots show the direct `GuineaPig++` output comprising the fluctuations due to the low statistics. Gaussian filters with widths of 3 and 10 bins, respectively, have been applied (orange and green dots, resp.). While there is still considerable fluctuation for 3 bin width Gaussian filtering, the distribution is perfectly smooth for 10 bin width. Hence, five bin widths seem a reasonable compromise for histograms with a total of 100 bins. Note that the bins are not equidistant, but shrink with a power law towards the $x_{e^-} = 1$ boundary on the right hand side of Fig. 10.1.

WHIZARD ships (inside its subpackage `CIRCE2`) with prepared beam spectra ready to be used within `CIRCE2` for the ILC beam spectra used in the ILC TDR [13,14,15,16,17]. These comprise the designed staging energies of 200 GeV, 230 GeV, 250 GeV, 350 GeV, and 500 GeV. Note that all of these spectra up to now do not take polarization of the original beams on the beamstrahlung into account, but are polarization-averaged. For backwards compatibility, also the 500 GeV spectra for the TESLA design [27,28], here both for polarized and polarization-averaged cases, are included. Correlated spectra for CLIC staging energies like 350 GeV, 1400 GeV and 3000 GeV are not yet (as of version 2.2.4) included in the `WHIZARD` distribution.

In the following we describe how to obtain such files with the tools included in `WHIZARD`(resp. `CIRCE2`). The procedure is equivalent to the so-called `lumi-linker` construction used by Timothy Barklow (SLAC) together with the legacy version `WHIZARD 1.95`. The workflow to produce such files is to run `GuineaPig++` with the following input parameters:

```
do_lumi = 7;
```

```
num_lumi = 100000000;
num_lumi_eg = 100000000;
num_lumi_gg = 100000000;
```

This demands from `GuineaPig++` the generation of distributions for the e^-e^+, $e^\mp\gamma$, and $\gamma\gamma$ components of the beamstrahlung's spectrum, respectively. These are the files `lumi.ee.out`, `lumi.eg.out`, `lumi.ge.out`, and `lumi.gg.out`, respectively. These contain pairs (E_1, E_2) of beam energies, *not* fractions of the original beam energy. Huge event numbers are out in here, as `GuineaPig++` will produce only a small fraction due to a very low generation efficiency.

The next step is to transfer these output files from `GuineaPig++` into input files used with `CIRCE2`. This is done by means of the tool `circe_tool.opt` that is installed together with the `WHIZARD` main binary and libraries. The user should run this executable with the following input file:

```
{ file="ilc500/ilc500.circe"                      # to be loaded by WHIZARD
  { design="ILC" roots=500 bins=100 scale=250 # E in [0,1]
    { pid/1=electron pid/2=positron pol=0     # unpolarized e-/e+
      events="ilc500/lumi.ee.out" columns=2   # <= Guinea-Pig
      lumi = 1564.763360                       # <= Guinea-Pig
      iterations = 10                          # adapting bins
      smooth = 5 [0,1) [0,1)                    # Gaussian filter 5 bins
      smooth = 5 [1] [0,1) smooth = 5 [0,1) [1] } } }
```

The first line defines the output file, that later can be read in into the beamstrahlung's description of `WHIZARD` (cf. below). Then, in the second line the design of the collider (here: ILC for 500 GeV center-of-mass energy, with the number of bins) is specified. The next line tells the tool to take the unpolarized case, then the `GuineaPig++` parameters (event file and luminosity) are set. In the last three lines, details concerning the adaptation of the simulation as well as the smoothing procedure are being specified: the number of iterations in the adaptation procedure, and for the smoothing with the Gaussian filter first in the continuum and then at the two edges of the spectrum. For more details confer the documentation in the `CIRCE2` subpackage.

This produces the corresponding input files that can be used within `WHIZARD` to describe beamstrahlung for lepton colliders, using a `SINDARIN` input file like:

```
beams = e1, E1 => circe2
$circe2_file = "ilc500.circe"
$circe2_design = "ILC"
?circe2_polarized = false
```

10.3.3   Photon Collider Spectra

For details confer the complete write-up of the `CIRCE2` subpackage.

10.4 Transverse momentum for ISR photons

The structure functions that describe the splitting of a beam particle into a particle pair, of which one enters the hard interaction and the other one is radiated, are defined and evaluated in the strict collinear approximation. In particular, this holds for the ISR structure function which describes the radiation of photons off a charged particle in the initial state.

The ISR structure function that is used by `WHIZARD` is understood to be inclusive, i.e., it implicitly contains an integration over transverse momentum. This approach is to be used for computing a total cross section via `integrate`. In `WHIZARD`, it is possible to unfold this integration, as a transformation that is applied by `simulate` step, event by event. The resulting modified events will show a proper logarithmic momentum-transfer (Q^2) distribution for the radiated photons. The recoil is applied to the hard-interaction system, such that four-momentum and $\sqrt{\hat{s}}$ are conserved. The distribution is cut off by $Q^2_{\max}$ (cf. `isr_q_max`) for large momentum transfer, and smoothly by the parton mass (cf. `isr_mass`) for small momentum transfer.

To activate this modification, set

```
?isr_handler = true
$isr_handler_mode = "recoil"
```

before, or as an option to, the `simulate` command.

Limitations: the current implementation of the p_T modification works only for the symmetric double-ISR case, i.e., both beams have to be charged particles with identical mass (e.g., e^+e^-). The mode `recoil` generates exactly one photon per beam, i.e., it modifies the momentum of the single collinear photon that the ISR structure function implementation produces, for each beam. (It is foreseen that further modes or options will allow to generate multiple photons. Alternatively, the `PYTHIA` shower can be used to simulate multiple photons radiated from the initial state.)

10.5 Transverse momentum for the EPA approximation

For the equivalent-photon approximation (EPA), which is also defined in the collinear limit, recoil momentum can be inserted into generated events in an entirely analogous way. The appropriate settings are

```
?epa_handler = true
$epa_handler_mode = "recoil"
```

Limitations: as for ISR, the current implementation of the p_T modification works only for the symmetric double-EPA case. Both incoming particles of the hard process must be photons, while both beams must be charged particles with identical mass (e.g., e^+e^-). Furthermore, the current implementation does not respect the kinematical limit parameter `epa_q_min`, it has to be set to zero. In effect, the lower Q^2 cutoff is determined by the beam-particle mass `epa_mass`, and the upper cutoff is either given by $Q_{\max}$ (the parameter `epa_q_max`), or by the limit $\sqrt{s}$ if this is not set.

It is possible to combine the ISR and EPA handlers, for processes where ISR is active for one of the beams, EPA for the other beam. For this scenario to work, both handler switches must be on, and both mode strings must coincide. The parameters are set separately for ISR and EPA, as described above.

10.6   Resonances and continuum

10.6.1   Complete matrix elements

Many elementary physical processes are composed of contributions that can be qualified as (multiply) *resonant* or *continuum*. For instance, the amplitude for the process $e^+e^- \to q\bar{q}q\bar{q}$, evaluated at tree level in perturbation theory, contains Feynman diagrams with zero, one, or two W and Z bosons as virtual lines. If the kinematical constraints allow this, two vector bosons can become simultaneously on-shell in part of phase space. To a first approximation, this situation is understood as W^+W^- or ZZ production with subsequent decay. The kinematical distributions show distinct resonances in the quark-pair spectra. Other graphs contain only one s-channel W/Z boson, or none at all, such as graphs with $q\bar{q}$ production and subsequent gluon radiation, splitting into another $q\bar{q}$ pair.

A WHIZARD declaration of the form

```
process q4 = e1, E1 => u, U, d, D
```

produces the full set of graphs for the selected final state, which after squaring and integrating yields the exact tree-level result for the process. The result contains all doubly and singly resonant parts, with correct resonance shapes, as well as the continuum contribution and all interference. This is, to given order in perturbation theory, the best possible approximation to the true result.

10.6.2   Processes restricted to resonances

For an intuitive separation of a two-boson "signal" contribution, it is possible to restrict the set of graphs to a certain intermediate state. For instance, the declaration

```
process q4_zz = e1, E1 => u, U, d, D { $restrictions = "3+4~Z && 5+6~Z" }
```

generates an amplitude that contains only those Feynman graphs where the specified quarks are connected to a Z virtual line. The result may be understood as ZZ production with subsequent decay, where the Z resonances exhibit a Breit-Wigner shape. Combining this with the analogous W^+W^- restricted process, the user can generate "signal" processes.

Adding both "signal" cross sections WW and ZZ will result in a reasonable approximation to the exact tree-level cross section. The amplitude misses the single-resonant and continuum contributions, and the squared amplitude misses the interference terms, however. More importantly, the restricted processes as such are not gauge-invariant (with respect to the electroweak gauge group), and they are no longer dominant away from resonant kinematics. We therefore strongly recommend that such restricted processes are always accompanied by a cut setup that restricts the kinematics to an approximately on-shell pattern for both resonances. For instance:

```
cuts = all 85 GeV < M < 95 GeV [u:U]
    and all 85 GeV < M < 95 GeV [d:D]
```

In this region, the gauge-dependent and continuum contributions are strictly subdominant. Away from the resonance(s), the results for a restricted process are meaningless, and the full process has to be computed instead.

10.6.3   Factorized processes

Another method for obtaining the signal contribution is a proper factorization into resonance production and decay. We would have to generate a production process and two decay processes:

```
process z_uu = Z => u, U
process z_dd = Z => d, D
process zz = e1, E1 => Z, Z
```

All three processes must be integrated. The integration results are partial decay widths and the ZZ production cross section, respectively. (Note that cut expressions in SINDARIN apply to all integrations, so make sure that no production-process cuts are active when integrating the decay processes.)

During a later event-generation step, the Z decays can then be activated by declaring the Z as unstable,

```
unstable Z (z_uu, z_dd)
```

and then simulating the production process

```
simulate (zz)
```

The generated events will consist of four-fermion final states, including all combinations of both decay modes. It is important to note that in this setup, the invariant $u\bar{u}$ and $d\bar{d}$ masses will be always *exactly* equal to the Z mass. There is no Breit-Wigner shape involved. However, in this approximation the results are gauge-invariant, as there is no off-shell contribution involved.

For further details on factorized processes and spin correlations, cf. Sec. 5.8.2.

10.6.4   Resonance insertion in the event record

From the above discussion, we may conclude that it is always preferable to compute the complete process for a given final state, as long as this is computationally feasible. However, in the simulation step this approach also has a drawback. Namely, if a parton-shower module (see below) is switched on, the parton-shower algorithm relies on event details in order to determine the radiation pattern of gluons and further splitting. In the generated event records, the full-process events carry the signature of non-resonant continuum production with no intermediate resonances. The parton shower will thus start the evolution at the process energy scale, the total available energy. By contrast, for an electroweak production and decay process, the evolution

should start only at the vector boson mass, m_Z. In effect, even though the resonant contribution of WW and ZZ constitutes the bulk of the cross section, the radiation pattern follows the dynamics of four-quark continuum production. In general, the number of radiated hadrons will be too high.

To overcome this problem, there is a refinement of the process description available in `WHIZARD`. By modifying the process declaration to

```
?resonance_history = true
resonance_on_shell_limit = 4
process q4 = e1, E1 => u, U, d, D
```

we advise the program to produce not just the complete matrix element, but also all possible restricted matrix elements containing resonant intermediate states. This has no effect at all on the integration step, and thus on the total cross section.

However, when subsequently events are generated with this setting, the program checks, for each event, the kinematics and determines the set of potentially resonant contributions. The criterion is whether the off-shellness of a particular would-be resonance is less than the resonance width multiplied by the value of `resonance_on_shell_limit` (default value = 4). For the set of resonance histories which pass this criterion (which can be empty), their respective squared matrix element is related to the full-process matrix element. The ratio is interpreted as a probability. The random-number generator then selects one or none of the resonance histories, and modifies the event record accordingly. In effect, for an appropriate fraction of the events, depending on the kinematics, the parton-shower module is provided with resonance information, so it can adjust the radiation pattern accordingly.

It has to be mentioned that generating the matrix-element code for all possible resonance histories takes additional computing resources. In the current default setup, this feature is switched off. It has to be explicitly activated via the `?resonance_history` flag.

Also, the feature can be activated or deactivated individually for each process, such as in

```
?resonance_history = true
process q4_with_res = e1, E1 => u, U, d, D  { ?resonance_history = true }
process q4_wo_res   = e1, E1 => u, U, d, D  { ?resonance_history = false }
```

If the flag is `false` for a process, no resonance code will be generated. Similarly, the flag has to be globally or locally active when `simulate` is called, such that the feature takes effect for event generation.

There are two additional parameters that can fine-tune the conditions for resonance insertion in the event record. Firstly, the parameter `resonance_on_shell_turnoff`, if nonzero, enables a Gaussian suppression of the probability for resonance insertion. For instance, setting

```
?resonance_history = true
resonance_on_shell_turnoff = 4
resonance_on_shell_limit = 8
```

will reduce the probability for the event to be qualified as resonant by $e^{-1} = 37\%$ if the kinematics is off-shell by four units of the width, and by $e^{-4} = 2\%$ at eight units of the
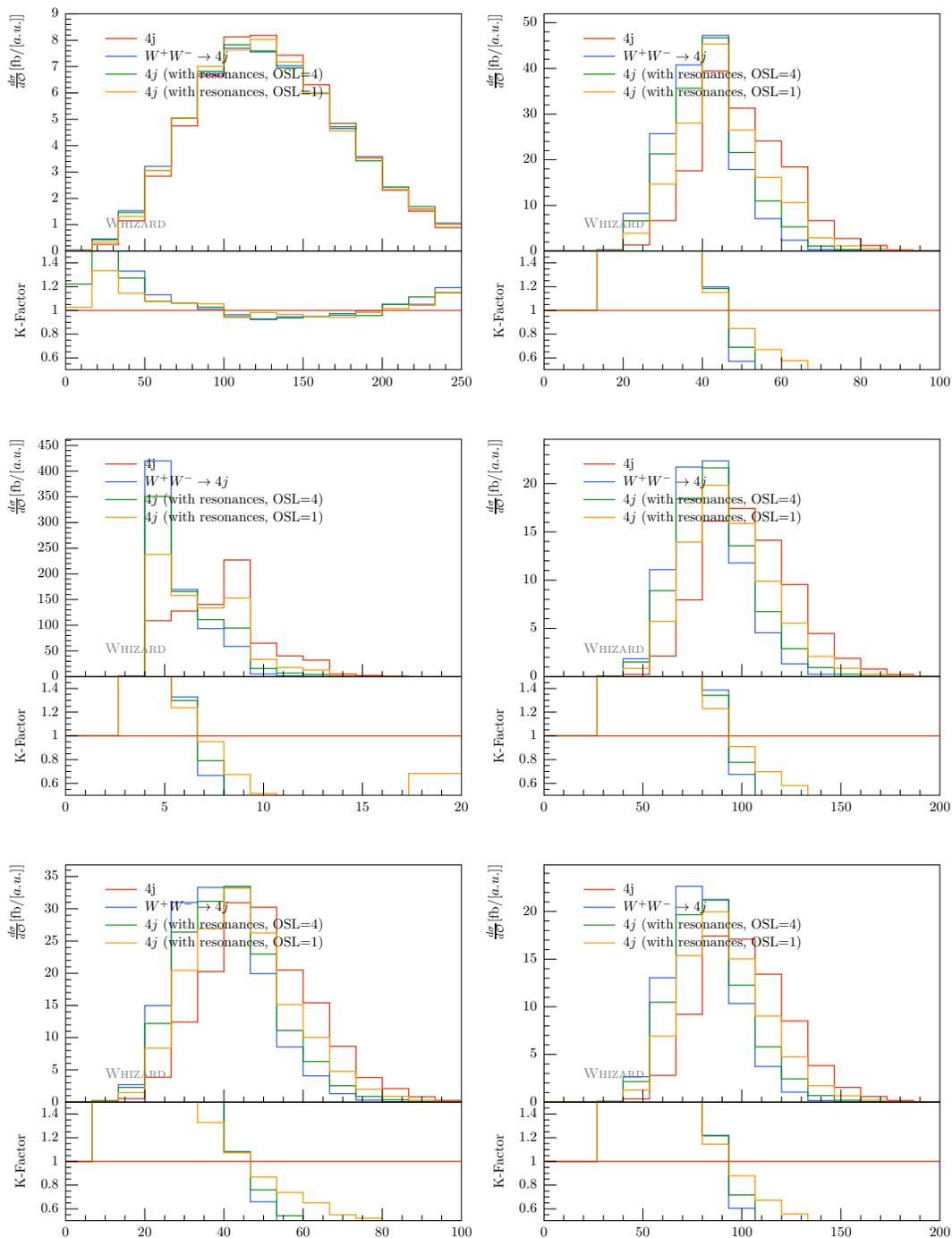
Figure 10.2: *The process $e^+e^- \to jjjj$ at 250 GeV center-of-mass energy is compared transferring the partonic events naively to the parton shower, i.e. without respecting any intermediate resonances (red lines). The blue lines show the process factorized into WW production and decay, where the shower knows the origin of the two jet pairs. The orange and dark green lines show the resonance treatment as mentioned in the text, with `resonance_on_shell_limit = 1` and `= 4`, respectively. `PYTHIA6` parton shower and hadronization with the OPAL tune have been used. The observables are: photon energy distribution and number of charged tracks (upper line left/right, number of hadrons and total number of particles (middle left/right), and number of photons and neutral particles (lower line left/right).*

width. Beyond this point, the setting of the `resonance_on_shell_limit` parameter eliminates resonance insertion altogether. In effect, the resonance-background transition is realized in a smooth way. Secondly, within the resonant-kinematics range the probability for qualifying the event as background can be reduced by the parameter `resonance_background_factor` (default value = 1) to a number between zero and one. Setting this to zero means that the event will be necessarily qualified as resonant, if it falls within the resonant-kinematics range.

Note that if an event, by the above mechanism, is identified as following a certain resonance history, the assigned color flow will be chosen to match the resonance history, not the complete matrix element. This may result in a reassignment of color flow with respect to the original partonic event.

Finally, we mention the order of execution: any additional matrix element code is compiled and linked when `compile` is executed for the processes in question. If this command is omitted, the `simulate` command will trigger compilation.

10.7   Parton showers and Hadronization

In order to produce sensible events, final state QCD (and also QED) radiation has to be considered as well as the binding of strongly interacting partons into mesons and baryons. Furthermore, final state hadronic resonances undergo subsequent decays into those particles showing up in (or traversing) the detector. The latter are mostly pions, kaons, photons, electrons and muons.

The physics associated with these topics can be divided into the perturbative part which is the regime of the parton shower, and the non-perturbative part which is the regime for the hadronization. WHIZARD comes with its own two different parton shower implementations, an analytic and a so-called k_T-ordered parton shower that will be detailed in the next section.

Note that in general it is not advisable to use different shower and hadronization methods, or in other words, when using shower and hadronization methods from different programs these would have to be tuned together again with the corresponding data.

Parton showers are approximations to full matrix elements taking only the leading color flow into account, and neglecting all interferences between different amplitudes leading to the same exclusive final state. They rely on the QCD (and QED) splitting functions to describe the emissions of partons off other partons. This is encoded in the so-called Sudakov form factor [29]:

$$\Delta(t_1, t_2) = \exp\left[\int\limits_{t_1}^{t_2} \mathrm{d}t \int\limits_{z_-}^{z_+} \mathrm{d}z \frac{\alpha_s}{2\pi t} P(z)\right]$$

This gives the probability for a parton to evolve from scale t_2 to t_1 without any further emissions of partons. t is the evolution parameter of the shower, which can be a parton energy, an emission angle, a virtuality, a transverse momentum etc. The variable z relates the two partons after the branching, with the most common choice being the ratio of energies of the parton after and before the branching. For final-state radiation brachings occur after the hard interaction, the evolution of the shower starts at the scale of the hard interaction, $t \sim \hat{s}$, down to a cut-off

scale $t = t_{\rm cut}$ that marks the transition to the non-perturbative regime of hadronization. In the space-like evolution for the initial-state shower, the evolution is from a cut-off representing the factorization scale for the parton distribution functions (PDFs) to the inverse of the hard process scale, $-\hat{s}$. Technically, this evolution is then backwards in (shower) time [30], leading to the necessity to include the PDFs in the Sudakov factors.

The main switches for the shower and hadronization which are realized as transformations on the partonic events within WHIZARD are ?allow_shower and ?allow_hadronization, which are true by default and only there for technical reasons. Next, different shower and hadronization methods can be chosen within WHIZARD:

```
$shower_method = "WHIZARD"
$hadronization_method = "PYTHIA6"
```

The snippet above shows the default choices in WHIZARD namely WHIZARD's intrinsic parton shower, but PYTHIA6 as hadronization tool. (Note that WHIZARD does not have its own hadronization module yet.) The usage of PYTHIA6 for showering and hadronization will be explained in Sec. 10.7.3, while the two different implementations of the WHIZARD homebrew parton showers are discussed in Sec. 10.7.1 and 10.7.2, respectively.

10.7.1 The k_T-ordered parton shower

10.7.2 The analytic parton shower

10.7.3 Parton shower and hadronization from PYTHIA6

Development of the PYTHIA6 generator for parton shower and hadronization (the Fortran version) has been discontinued by the authors several years ago. Hence, the final release of that program is frozen. This allowed to ship this final version, v6.427, with the WHIZARD distribution without the need of updating it all the time. One of the main reasons for that inclusion – besides having the standard tool for showering and hadronization for decays at hand – is to allow for backwards validation within WHIZARD particularly for the event samples generated for the development of linear collider physics: first for TESLA, JLC and NLC, and later on for the Conceptual and Technical Design Report for ILC, for the Conceptual Design Report for CLIC as well as for the Letters of Intent for the LC detectors, ILD and SiD.

Usually, an external parton shower and hadronization program (PS) is steered via the transfer of event files that are given to the PS via LHE events, while the PS program then produces hadron level events, usually in HepMC format. These can then be directed towards a full or fast detector simulation program. As PYTHIA6 has been completely integrated inside the WHIZARD framework, the showered or more general hadron level events can be returned to and kept inside WHIZARD's internal event record, and hence be used in WHIZARD's internal event analysis. In that way, the events can be also written out in event formats that are not supported by PYTHIA6, e.g. LCIO via the output capabilities of WHIZARD.

There are several switches to directly steer PYTHIA6 (the values in brackets correspond to the PYTHIA6 variables):

```
ps_mass_cutoff = 1 GeV              [PARJ(82)]
ps_fsr_lambda = 0.29 GeV           [PARP(72)]
```

```
ps_isr_lambda = 0.29 GeV                 [PARP(61)]
ps_max_n_flavors = 5                     [MSTJ(45)]
?ps_isr_alphas_running = true            [MSTP(64)]
?ps_fsr_alphas_running = true            [MSTJ(44)]
ps_fixed_alphas = 0.2                    [PARU(111)]
?ps_isr_angular_ordered = true           [MSTP(62)]
ps_isr_primordial_kt_width = 1.5 GeV     [PARP(91)]
ps_isr_primordial_kt_cutoff = 5.0 GeV    [PARP(93)]
ps_isr_z_cutoff = 0.999                  [1-PARP(66)]
ps_isr_minenergy = 2 GeV                 [PARP(65)]
?ps_isr_only_onshell_emitted_partons =
        true                             [MSTP(63)]
```

The values given above are the default values. The first value corresponds to the PYTHIA6 parameter PARJ(82), its squared being the minimal virtuality that is allowed for the parton shower, i.e. the cross-over to the hadronization. The same parameter is used also for the WHIZARD showers. ps_fsr_lambda is the equivalent of PARP(72) and is the Λ_{QCD} for the final state shower. The corresponding variable for the initial state shower is called PARP(61) in PYTHIA6. By the next variable (MSTJ(45)), the maximal number of flavors produced in splittings in the shower is given, together with the number of active flavors in the running of α_s. ?ps_isr_alphas_running which corresponds to MSTP(64) in PYTHIA6 determines whether or net a running α_s is taken in the space-like initial state showers. The same variable for the final state shower is MSTJ(44). For fixed α_s, the default value is given by ps_fixed_alpha, corresponding to PARU(111). MSTP(62) determines whether the ISR shower is angular order, i.e. whether angles are increasing towards the hard interaction. This is per default true, and set in the variable ?ps_isr_angular_ordered. The width of the distribution for the primordial (intrinsic) k_T distribution (which is a non-perturbative quantity) is the PYTHIA6 variable PARP(91), while in WHIZARD it is given by pythia_isr_primordial_kt_width. The next variable (PARP(93) gives the upper cutoff for that distribution, which is 5 GeV per default. For splitting in space-like showers, there is a cutoff on the z variable named ps_isr_z_cutoff in WHIZARD. This corresponds to one minus the value of the PYTHIA6 parameter PARP(66). PARP(65), on the other hand, gives the minimal (effective) energy for a time-like or on-shell emitted parton on a space-like QCD shower, given by the SINDARIN parameter ps_isr_minenergy. Whether or not partons emitted from space-like showers are allowed to be only on-shell is given by ?ps_isr_only_onshell_emitted_partons, MSTP(63) in PYTHIA6 language. For more details confer the PYTHIA6 manual [31].

Any other non-standard PYTHIA6 parameter can be fed into the parton shower via the string variable

```
$ps_PYTHIA_PYGIVE = "...."
```

Variables set here get preference over the ones set explicitly by dedicated SINDARIN commands. For example, the OPAL tune for hadronic final states can be set via:

```
$ps_PYTHIA_PYGIVE = "MSTJ(28)=0; PMAS(25,1)=120.; PMAS(25,2)=0.3605E-02; MSTJ(41)=2;
 MSTU(22)=2000; PARJ(21)=0.40000; PARJ(41)=0.11000; PARJ(42)=0.52000; PARJ(81)=0.25000;
 PARJ(82)=1.90000; MSTJ(11)=3; PARJ(54)=-0.03100; PARJ(55)=-0.00200; PARJ(1)=0.08500;
 PARJ(3)=0.45000; PARJ(4)=0.02500; PARJ(2)=0.31000; PARJ(11)=0.60000; PARJ(12)=0.40000;
 PARJ(13)=0.72000; PARJ(14)=0.43000; PARJ(15)=0.08000; PARJ(16)=0.08000;
 PARJ(17)=0.17000; MSTP(3)=1;MSTP(71)=1"
```

A very common error that appears quite often when using `PYTHIA6` for SUSY or any other model having a stable particle that serves as a possible Dark Matter candidate, is the following warning/error message:

```
     Advisory warning type 3 given after        0 PYEXEC calls:
     (PYRESD:) Failed to decay particle  1000022 with mass    15.000
 *****************************************************************************
 *****************************************************************************
 *** FATAL ERROR: Simulation: failed to generate valid event after 10000 tries
 *****************************************************************************
 *****************************************************************************
```

In that case, `PYTHIA6` gets a stable particle (here the lightest neutralino with the PDG code 1000022) handed over and does not know what to do with it. Particularly, it wants to treat it as a heavy resonance which should be decayed, but does not know how do that. After a certain number of tries (in the example abobe 10k), `WHIZARD` ends with a fatal error telling the user that the event transformation for the parton shower in the simulation has failed without producing a valid event. The solution to work around that problem is to let `PYTHIA6` know that the neutralino (or any other DM candidate) is stable by means of

```
    $ps_PYTHIA_PYGIVE = "MDCY(C1000022,1)=0"
```

Here, 1000022 has to be replaced by the stable dark matter candidate or long-lived particle in the user's favorite model. Also note that with other options being passed to `PYTHIA6` the `MDCY` option above has to be added to an existing `$ps_PYTHIA_PYGIVE` command separated by a semicolon.

10.7.4   Parton shower and hadronization from `PYTHIA8`

10.7.5   Other tools for parton shower and hadronization

Chapter 11

More on Event Generation

In order to perform a physics analysis with WHIZARD one has to generate events. This seems to be a trivial statement, but as there have been any questions like "My WHIZARD does not produce plots – what has gone wrong?" we believe that repeating that rule is worthwile. Of course, it is not mandatory to use WHIZARD's own analysis set-up, the user can always choose to just generate events and use his/her own analysis package like ROOT, or TopDrawer, or you name it for the analysis.

Accordingly, we first start to describe how to generate events and what options there are – different event formats, renaming output files, using weighted or unweighted events with different normalizations. How to re-use and manipulate already generated event samples, how to limit the number of events per file, etc. etc.

11.1   Event generation

To explain how event generation works, we again take our favourite example, $e^+e^- \to \mu^+\mu^-$,

```
process eemm = e1, E1 => e2, E2
```

The command to trigger generation of events is `simulate (<proc_name>) { <options> }`, so in our case – neglecting any options for now – simply:

```
simulate (eemm)
```

When you run this SINDARIN file you will experience a fatal error: `FATAL ERROR: Colliding beams:  sqrts is zero (please set sqrts)`. This is because WHIZARD needs to compile and integrate the process `eemm` first before event simulation, because it needs the information of the corresponding cross section, phase space parameterization and grids. It does both automatically, but you have to provide WHIZARD with the beam setup, or at least with the center-of-momentum energy. A corresponding `integrate` command like

```
sqrts = 500 GeV
integrate (eemm) { iterations = 3:10000 }
```

obviously has to appear *before* the corresponding `simulate` command (otherwise you would be punished by the same error message as before). Putting things in the correct order results in an output like:

```
| Reading model file '/usr/local/share/whizard/models/SM.mdl'
| Preloaded model: SM
| Process library 'default_lib': initialized
| Preloaded library: default_lib
| Reading commands from file 'bla.sin'
| Process library 'default_lib': recorded process 'eemm'
sqrts =  5.000000000000E+02
| Integrate: current process library needs compilation
| Process library 'default_lib': compiling ...
| Process library 'default_lib': keeping makefile
| Process library 'default_lib': keeping driver
| Process library 'default_lib': active
| Process library 'default_lib': ... success.
| Integrate: compilation done
| RNG: Initializing TAO random-number generator
| RNG: Setting seed for random-number generator to 29912
| Initializing integration for process eemm:
| ------------------------------------------------------------------------
| Process [scattering]: 'eemm'
|   Library name  = 'default_lib'
|   Process index = 1
|   Process components:
|     1: 'eemm_i1':   e-, e+ => mu-, mu+ [omega]
| ------------------------------------------------------------------------
| Beam structure: [any particles]
| Beam data (collision):
|   e-  (mass = 5.1099700E-04 GeV)
|   e+  (mass = 5.1099700E-04 GeV)
|   sqrts = 5.000000000000E+02 GeV
| Phase space: generating configuration ...
| Phase space: ... success.
| Phase space: writing configuration file 'eemm_i1.phs'
| Phase space: 2 channels, 2 dimensions
| Phase space: found 2 channels, collected in 2 groves.
| Phase space: Using 2 equivalences between channels.
| Phase space: wood
Warning: No cuts have been defined.
| OpenMP: Using 8 threads
| Starting integration for process 'eemm'
| Integrate: iterations = 3:10000
| Integrator: 2 chains, 2 channels, 2 dimensions
| Integrator: Using VAMP channel equivalences
| Integrator: 10000 initial calls, 20 bins, stratified = T
| Integrator: VAMP
|==============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]    Chi2 N[It] |
|==============================================================================|
   1       9216  4.2833237E+02  7.14E-02    0.02    0.02*  40.29
```

```
     2         9216  4.2829071E+02  7.08E-02     0.02     0.02*  40.29
     3         9216  4.2838304E+02  7.04E-02     0.02     0.02*  40.29
|------------------------------------------------------------------------------|
     3        27648  4.2833558E+02  4.09E-02     0.01     0.02   40.29   0.43   3
|==============================================================================|
| Time estimate for generating 10000 events: 0d:00h:00m:04s
| Creating integration history display eemm-history.ps and eemm-history.pdf
| Starting simulation for process 'eemm'
| Simulate: using integration grids from file 'eemm_m1.vg'
| RNG: Initializing TAO random-number generator
| RNG: Setting seed for random-number generator to 29913
| OpenMP: Using 8 threads
| Simulation: requested number of events = 0
|           corr. to luminosity [fb-1] =   0.0000E+00
| Events: writing to raw file 'eemm.evx'
| Events: generating 0 unweighted, unpolarized events ...
| Events: event normalization mode '1'
|         ... event sample complete.
| Events: closing raw file 'eemm.evx'
| There were no errors and     1 warning(s).
| WHIZARD run finished.
|==============================================================================|
```

So, WHIZARD tells you that it has entered simulation mode, but besides this, it has not done anything. The next step is that you have to demand event generation – there are two ways to do this: you could either specify a certain number, say 42, of events you want to have generated by WHIZARD, or you could provide a number for an integrated luminosity of some experiment. (Note, that if you choose to take both options, WHIZARD will take the one which gives the larger event sample. This, of course, depends on the given process(es) – as well as cuts – and its corresponding cross section(s).) The first of these options is set with the command: n_events = <number>, the second with luminosity = <number> <opt. unit>.

Another important point already stated several times in the manual is that WHIZARD follows the commands in the steering SINDARIN file in a chronological order. Hence, a given number of events or luminosity *after* a simulate command will be ignored – or are relevant only for any simulate command potentially following further down in the SINDARIN file. So, in our case, try:

```
 n_events = 500
 luminosity = 10
 simulate (eemm)
```

Per default, numbers for integrated luminosity are understood as inverse femtobarn. So, for the cross section above this would correspond to 4283 events, clearly superseding the demand for 500 events. After reducing the luminosity number from ten to one inverse femtobarn, 500 is the larger number of events taken by WHIZARD for event generation. Now WHIZARD tells you:

```
| Simulation: requested number of events = 500
|           corr. to luminosity [fb-1] =   1.1673E+00
```

```
| Events: reading from raw file 'eemm.evx'
| Events: reading 500 unweighted, unpolarized events ...
| Events: event normalization mode '1'
| ... event file terminates after 0 events.
| Events: appending to raw file 'eemm.evx'
| Generating remaining  500 events ...
|          ... event sample complete.
| Events: closing raw file 'eemm.evx'
```

I.e., it evaluates the luminosity to which the sample of 500 events would correspond to, which is now, of course, bigger than the 1fb^{-1} explicitly given for the luminosity. Furthermore, you can read off that a file `whizard.evx` has been generated, containing the demanded 500 events. (It was there before containing zero events, because to `n_events` or `luminosity` value had been set. `WHIZARD` then tried to get the events first from file before generating new ones). Files with the suffix `.evx` are binary format event files, using a machine-dependent `WHIZARD`-specific event file format. Before we list the event formats supported by `WHIZARD`, the next two sections will tell you more about unweighted and weighted events as well as different possibilities to normalize events in `WHIZARD`.

As already explained for the libraries, as well as the phase space and grid files in Chap. 5, `WHIZARD` is trying to re-use as much information as possible. This is of course also true for the event files. There are special MD5 check sums testing the integrity and compatibility of the event files. If you demand for a process for which an event file already exists (as in the example above, though it was empty) equally many or less events than generated before, `WHIZARD` will not generate again but re-use the existing events (as already explained, the events are stored in a `WHIZARD`-own binary event format, i.e. in a so-called `.evx` file. If you suppress generation of that file, as will be described in subsection 11.5 then `WHIZARD` has to generate events all the time). From version v2.2.0 of `WHIZARD` on, the program is also able to read in event from different event formats. However, most event formats do not contain as many information as `WHIZARD`'s internal format, and a complete reconstruction of the events might not be possible. Re-using event files is very practical for doing several different analyses with the same data, especially if there are many and big data samples. Consider the case, there is an event file with 200 events, and you now ask `WHIZARD` to generate 300 events, then it will re-use the 200 events (if MD5 check sums are OK!), generate the remaining 100 events and append them to the existing file. If the user for some reason, however, wants to regenerate events (i.e. ignoring possibly existing events), there is the command option `whizard --rebuild-events`.

11.2   Unweighted and weighted events

`WHIZARD` is able to generate unweighted events, i.e. events that are distributed uniformly and each contribute with the same event weight to the whole sample. This is done by mapping out the phase space of the process under consideration according to its different phase space channels (which each get their own weights), and then unweighting the sample of weighted events. Only a sample of unweighted events could in principle be compared to a real data sample from some

experiment. The seventh column in the WHIZARD iteration/adaptation procedure tells you about the efficiency of the grids, i.e. how well the phase space is mapped to a flat function. The better this is achieved, the higher the efficiency becomes, and the closer the weights of the different phase space channels are to uniformity. This means, for higher efficiency less weighted events ("calls") are needed to generate a single unweighted event. An efficiency of 10 % means that ten weighted events are needed to generate one single unweighted event. After the integration is done, WHIZARD uses the duration of calls during the adaptation to estimate a time interval needed to generate 10,000 unweighted events. The ability of the adaptive multi-channel Monte Carlo decreases with the number of integrations, i.e. with the number of final state particles. Adding more and more final state particles in general also increases the complexity of phase space, especially its singularity structure. For a $2 \to 2$ process the efficiency is roughly of the order of several tens of per cent. As a rule of thumb, one can say that with every additional pair of final state particle the average efficiency one can achieve decreases by a factor of five to ten.

The default of WHIZARD is to generate *unweighted* events. One can use the logical variable ?unweighted = false to disable unweighting and generate weighted events. (The command ?unweighted = true is a tautology, because true is the default for this variable.) Note that again this command has to appear *before* the corresponding simulate command, otherwise it will be ignored or effective only for any simulate command appearing later in the SINDARIN file.

In the unweighted procedure, WHIZARD is keeping track of the highest weight that has been appeared during the adaptation, and the efficiency for the unweighting has been estimated from the average value of the sampling function compared to the maximum value. In principle, during event generation no events should be generated whose sampling function value exceeds the maximum function value encountered during the grid adaptation. Sometimes, however, there are numerical fluctuations and such events are happening. They are called *excess events*. WHIZARD does keep track of these excess events during event generation and will report about them, e.g.:

```
Warning: Encountered events with excess weight: 9 events (  0.090 %)
| Maximum excess weight = 6.083E-01
| Average excess weight = 2.112E-04
```

Whenever in an event generation excess events appear, this shows that the adaptation of the sampling function has not been perfect. When the number of excess weights is a finite number of percent, you should inspect the phase-space setup and try to improve its settings to get a better adaptation.

Generating *weighted* events is, of course, much faster if the same number of events is requested. Each event carries a weight factor which is taken into account for any internal analysis (histograms), and written to file if an external file format has been selected. The file format must support event weights.

In a weighted event sample, there is typically a fraction of events which effectively have weight zero, namely those that have been created by the phase-space sampler but do not pass the requested cuts. In the default setup, those events are silently dropped, such that the events written to file or available for analysis all have nonzero weight. However, dropping such events affects the overall normalization. If this has happened, the program will issue a warning of the form

```
        | Dropped events (weight zero) = 1142 (total 2142)
        Warning: All event weights must be rescaled by f = 4.66853408E-01
```

This factor has to be applied by hand to any external event files (and to internally generated histograms). The program cannot include the factor in the event records, because it is known only after all events have been generated. To avoid this problem, there is the logical flag `?keep_failed_events` which tells WHIZARD not to drop events with weight zero. The normalization will be correct, but the event sample will include invalid events which have to be vetoed by their zero weight, before any operations on the event record are performed.

11.3 Choice on event normalizations

There are basically four different choices to normalize event weights ($\langle \ldots \rangle$ denotes the average):

1. $\langle w_i \rangle = 1,$ $\langle \sum_i w_i \rangle = N$

2. $\langle w_i \rangle = \sigma,$ $\langle \sum_i w_i \rangle = N \times \sigma$

3. $\langle w_i \rangle = 1/N,$ $\langle \sum_i w_i \rangle = 1$

4. $\langle w_i \rangle = \sigma/N,$ $\langle \sum_i w_i \rangle = \sigma$

So the four options are to have the average weight equal to unity, to the cross section of the corresponding process, to one over the number of events, or the cross section over the event calls. In these four cases, the event weights sum up to the event number, the event number times the cross section, to unity, and to the cross section, respectively. Note that neither of these really guarantees that all event weights individually lie in the interval $0 \leq w_i \leq 1$.

The user can steer the normalization of events by using in SINDARIN input files the string variable `$sample_normalization`. The default is `$sample_normalization = "auto"`, which uses option 1 for unweighted and 2 for weighted events, respectively. Note that this is also what the Les Houches Event Format (LHEF) demands for both types of events. This is WHIZARD's preferred mode, also for the reason, that event normalizations are independent from the number of events. Hence, event samples can be cut or expanded without further need to adjust the normalization. The unit normalization (option 1) can be switched on also for weighted events by setting the event normalization variable equal to `"1"`. Option 2 can be demanded by setting `$sample_normalization = "sigma"`. Options 3 and 4 can be set by `"1/n"` and `"sigma/n"`, respectively. WHIZARD accepts small and capital letters for these expressions.

There are several event formats (based upon the old common block definition HEPRUP) like some of the ASCII formats, LHA, LHE and HepMC that demand cross sections (and corresponding MCintegration errors) to be given in picobarn. So they are converted from the WHIZARD default of femtobarn to picobarn. The only exception is if a (pseudo-)event file for a decay is generated where the unit in those entries is downscaled by a factor of 1000, but remains in GeV as default unit.

In the following section we show some examples when discussing the different event formats available in WHIZARD.

11.4 Event selection

The `selection` expression (cf. Sec. 5.9.2) reduces the event sample during generation or rescanning, selecting only events for which the expression evaluates to `true`. Apart from internal analysis, the selection also applies to writing external files. For instance, the following code generates a $e^+e^- \to W^+W^-$ sample with longitudinally polarized W bosons only:

```
process ww = "e+", "e-" => "W-", "W+"
polarized "W+"
polarized "W-"
?polarized_events = true
sqrts = 500
selection = all Hel == 0 ["W+":"W-"]
simulate (ww) { n_events = 1000 }
```

The number of events that end up in the sample on file is equal to the number of events with longitudinally polarized Ws in the generated sample, so the file will contain less than 1000 events.

11.5 Supported event formats

Event formats can either be distinguished whether they are plain text (i.e. ASCII) formats or binary formats. Besides this, one can classify event formats according to whether they are natively supported by `WHIZARD` or need some external program or library to be linked. Table 11.1 gives a complete list of all event formats available in `WHIZARD`. The second column shows whether these are ASCII or binary formats, the third column contains brief remarks about the corresponding format, while the last column tells whether external programs or libraries are needed (which is the case only for the HepMC formats).

The ".`evx`" is `WHIZARD`'s native binary event format. If you demand event generation and do not specify anything further, `WHIZARD` will write out its events exclusively in this binary format. So in the examples discussed in the previous chapters (where we omitted all details about event formats), in all cases this and only this internal binary format has been generated. The generation of this raw format can be suppressed (e.g. if you want to have only one specific event file type) by setting the variable `?write_raw = false`. However, if the raw event file is not present, `WHIZARD` is not able to re-use existing events (e.g. from an ASCII file) and will regenerate events for a given process. Note that from version v2.2.0 of `WHIZARD` on, the program is able to (partially) reconstruct complete events also from other formats than its internal format (e.g. LHEF), but this is still under construction and not yet complete.

Other event formats can be written out by setting the variable `sample_format = <format>`, where `<format>` can be any of the following supported variables:

- `ascii`: a quite verbose ASCII format which contains lots of information (an example is shown in the appendix).
  Standard suffix: `.evt`

Format	Type	remark	ext.
ascii	ASCII	`WHIZARD` verbose format	no
Athena	ASCII	variant of HEPEVT	no
debug	ASCII	most verbose `WHIZARD` format	no
evx	binary	`WHIZARD`'s home-brew	no
HepMC	ASCII	HepMC format	yes
HEPEVT	ASCII	`WHIZARD` 1 style	no
LCIO	ASCII	LCIO format	yes
LHA	ASCII	`WHIZARD` 1/old Les Houches style	no
LHEF	ASCII	Les Houches accord compliant	no
long	ASCII	variant of HEPEVT	no
mokka	ASCII	variant of HEPEVT	no
short	ASCII	variant of HEPEVT	no
StdHEP (HEPEVT)	binary	based on HEPEVT common block	no
StdHEP (HEPRUP/EUP)	binary	based on HEPRUP/EUP common block	no
Weight stream	ASCII	just weights	no

Table 11.1:   *Event formats supported by* `WHIZARD`, *classified according to ASCII/binary formats and whether an external program or library is needed to generate a file of this format. For both the HEPEVT and the LHA format there is a more verbose variant.*

- `debug`: an even more verbose ASCII format intended for debugging which prints out also information about the internal data structures
  Standard suffix: `.debug`

- `hepevt`: ASCII format that writes out a specific incarnation of the HEPEVT common block (`WHIZARD` 1 back-compatibility)
  Standard suffix: `.hepevt`

- `hepevt_verb`: more verbose version of `hepevt` (`WHIZARD` 1 back-compatibility)
  Standard suffix: `.hepevt.verb`

- `short`: abbreviated variant of the previous HEPEVT (`WHIZARD` 1 back-compatibility)
  Standard suffix: `.short.evt`

- `long`: HEPEVT variant that contains a little bit more information than the short format but less than HEPEVT (`WHIZARD` 1 back-compatibility)
  Standard suffix: `.long.evt`

- `athena`: HEPEVT variant suitable for read-out in the ATLAS ATHENA software environment (`WHIZARD` 1 back-compatibility)
  Standard suffix: `.athena.evt`

- `mokka`: HEPEVT variant suitable for read-out in the MOKKA ILC software environment
  Standard suffix: `.mokka.evt`

- `lcio`: LCIO ASCII format (only available if LCIO is installed and correctly linked)
  Standard suffix: `.lcio`

- `lha`: Implementation of the Les Houches Accord as it was in the old MadEvent and
  `WHIZARD 1`
  Standard suffix: `.lha`

- `lha_verb`: more verbose version of `lha`
  Standard suffix: `.lha.verb`

- `lhef`: Formatted Les Houches Accord implementation that contains the XML headers
  Standard suffix: `.lhe`

- `hepmc`: HepMC ASCII format (only available if HepMC is installed and correctly linked)
  Standard suffix: `.hepmc`

- `stdhep`: StdHEP binary format based on the HEPEVT common block
  Standard suffix: `.hep`

- `stdhep_up`: StdHEP binary format based on the HEPRUP/HEPEUP common blocks
  Standard suffix: `.up.hep`

- `stdhep_ev4`: StdHEP binary format based on the HEPEVT/HEPEV4 common blocks
  Standard suffix: `.ev4.hep`

- `weight_stream`: Format that prints out only the event weight (and maybe alternative
  ones)
  Standard suffix: `.weight.dat`

Of course, the variable `sample_format` can contain more than one of the above identifiers, in which case more than one different event file format is generated. The list above also shows the standard suffixes for these event formats (remember, that the native binary format of `WHIZARD` does have the suffix `.evx`). (The suffix of the different event formats can even be changed by the user by setting the corresponding variable `$extension_lhef = "foo"` or `$extension_ascii_short = "bread"`. The dot is automatically included.)

The name of the corresponding event sample is taken to be the string of the name of the first process in the `simulate` statement. Remember, that conventionally the events for all processes in one `simulate` statement will be written into one single event file. So `simulate (proc1, proc2)` will write events for the two processes `proc1` and `proc2` into one single event file with name `proc1.evx`. The name can be changed by the user with the command `$sample = "<name>"`.

The commands `$sample` and `sample_format` are both accepted as optional arguments of a `simulate` command, so e.g. `simulate (proc) { $sample = "foo" sample_format = hepmc }` generates an event sample in the HepMC format for the process `proc` in the file `foo.hepmc`.

Examples for event formats, for specifications of the event formats correspond the different accords and publications [1]:

---

[1]Some event formats, based on the `HEPEVT` or `HEPEUP` common blocks, use fixed-form ASCII output with a

HEPEVT: The HEPEVT is an ASCII event format that does not contain an event file header. There is a one-line header for each single event, containing four entries. The number of particles in the event (`ISTHEP`), which is four for a fictitious example process $hh \to hh$, but could be larger if e.g. beam remnants are demanded to be included in the event. The second entry and third entry are the number of outgoing particles and beam remnants, respectively. The event weight is the last entry. For each particle in the event there are three lines: the first one is the status according to the HEPEVT format, `ISTHEP`, the second one the PDG code, `IDHEP`, then there are the one or two possible mother particle, `JMOHEP`, the first and last possible daughter particle, `JDAHEP`, and the polarization. The second line contains the three momentum components, p_x, p_y, p_z, the particle energy E, and its mass, m. The last line contains the position of the vertex in the event reconstruction.

```
4 2 0  3.0574068604E+08
2 25 0 0 3 4 0
 0.0000000000E+00   0.0000000000E+00   4.8412291828E+02   5.0000000000E+02   1.2500000000E+02
 0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00
2 25 0 0 3 4 0
 0.0000000000E+00   0.0000000000E+00  -4.8412291828E+02   5.0000000000E+02   1.2500000000E+02
 0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00
1 25 1 2 0 0 0
-1.4960220911E+02  -4.6042825611E+02   0.0000000000E+00   5.0000000000E+02   1.2500000000E+02
 0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00
1 25 1 2 0 0 0
 1.4960220911E+02   4.6042825611E+02   0.0000000000E+00   5.0000000000E+02   1.2500000000E+02
 0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00   0.0000000000E+00
```

ASCII SHORT: This is basically the same as the HEPEVT standard, but very much abbreviated. The header line for each event is identical, but the first line per particle does only contain the PDG and the polarization, while the vertex information line is omitted.

```
4 2 0  3.0574068604E+08
25 0
 0.0000000000E+00   0.0000000000E+00   4.8412291828E+02   5.0000000000E+02   1.2500000000E+02
25 0
 0.0000000000E+00   0.0000000000E+00  -4.8412291828E+02   5.0000000000E+02   1.2500000000E+02
25 0
-1.4960220911E+02  -4.6042825611E+02   0.0000000000E+00   5.0000000000E+02   1.2500000000E+02
25 0
 1.4960220911E+02   4.6042825611E+02   0.0000000000E+00   5.0000000000E+02   1.2500000000E+02
```

ASCII LONG: Identical to the ASCII short format, but after each event there is a line containg two values: the value of the sample function to be integrated over phase space, so basically the squared matrix element including all normalization factors, flux factor, structure functions etc.

---

two-digit exponent for real numbers. There are rare cases (mainly, ISR photons) where the event record can contain numbers with absolute value less than 10^{-99}. Since those numbers are not representable in that format, WHIZARD will set all non-zero numbers below that value to $\pm 10^{-99}$, when filling either common block. Obviously, such values are physically irrelevant, but in the output they are representable and distinguishable from zero.

```
4 2 0  3.0574068604E+08
25 0
 0.0000000000E+00  0.0000000000E+00  4.8412291828E+02  5.0000000000E+02  1.2500000000E+02
25 0
 0.0000000000E+00  0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02  1.2500000000E+02
25 0
-1.4960220911E+02 -4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02
25 0
 1.4960220911E+02  4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02
 1.0000000000E+00  1.0000000000E+00
```

ATHENA: Quite similar to the HEPEVT ASCII format. The header line, however, does contain only two numbers: an event counter, and the number of particles in the event. The first line for each particle lacks the polarization information (irrelevant for the ATHENA environment), but has as leading entry an ordering number counting the particles in the event. The vertex information line has only the four relevant position entries.

```
0 4
1 2 25 0 0 3 4
 0.0000000000E+00  0.0000000000E+00  4.8412291828E+02  5.0000000000E+02  1.2500000000E+02
 0.0000000000E+00  0.0000000000E+00  0.0000000000E+00  0.0000000000E+00
2 2 25 0 0 3 4
 0.0000000000E+00  0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02  1.2500000000E+02
 0.0000000000E+00  0.0000000000E+00  0.0000000000E+00  0.0000000000E+00
3 1 25 1 2 0 0
-1.4960220911E+02 -4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02
 0.0000000000E+00  0.0000000000E+00  0.0000000000E+00  0.0000000000E+00
4 1 25 1 2 0 0
 1.4960220911E+02  4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02
 0.0000000000E+00  0.0000000000E+00  0.0000000000E+00  0.0000000000E+00
```

MOKKA: Quite similar to the ASCII short format, but the event entries are the particle status, the PDG code, the first and last daughter, the three spatial components of the momentum, as well as the mass.

```
4 2 0  3.0574068604E+08
2 25 3 4  0.0000000000E+00  0.0000000000E+00  4.8412291828E+02  1.2500000000E+02
2 25 3 4  0.0000000000E+00  0.0000000000E+00 -4.8412291828E+02  1.2500000000E+02
1 25 0 0 -1.4960220911E+02 -4.6042825611E+02  0.0000000000E+00  1.2500000000E+02
1 25 0 0  1.4960220911E+02  4.6042825611E+02  0.0000000000E+00  1.2500000000E+02
```

LHA: This is the implementation of the Les Houches Accord, as it was used in `WHIZARD` 1 and the old MadEvent. There is a first line containing six entries: 1. the number of particles in the event, `NUP`, 2. the subprocess identification index, `IDPRUP`, 3. the event weight, `XWGTUP`, 4. the scale of the process, `SCALUP`, 5. the value or status of α_{QED}, `AQEDUP`, 6. the value for α_s, `AQCDUP`. The next seven lines contain as many entries as there are particles in the event: the first one has the PDG codes, `IDUP`, the next two the first and second mother of the particles, `MOTHUP`, the fourth and fifth line the two color indices, `ICOLUP`, the next one the status of the particle, `ISTUP`, and the last line the polarization information, `ISPINUP`. At the end of the event there are as lines for each particles with the counter in the event and the four-vector of the particle. For more information on this event format confer [51].

```
25 25  5.0000000000E+02  5.0000000000E+02 -1 -1 -1 -1 3 1
 1.0000000000E-01  1.0000000000E-03  1.0000000000E+00 42
    4    1 3.0574068604E+08  1.000000E+03 -1.000000E+00 -1.000000E+00
    25    25    25    25
     0     0     1     1
     0     0     2     2
     0     0     0     0
     0     0     0     0
    -1    -1     1     1
     9     9     9     9
     1 5.0000000000E+02  0.0000000000E+00  0.0000000000E+00  4.8412291828E+02
     2 5.0000000000E+02  0.0000000000E+00  0.0000000000E+00 -4.8412291828E+02
     3 5.0000000000E+02 -1.4960220911E+02 -4.6042825611E+02  0.0000000000E+00
     4 5.0000000000E+02  1.4960220911E+02  4.6042825611E+02  0.0000000000E+00
```

LHEF: This is the modern version of the Les Houches accord event format (LHEF), for the details confer the corresponding publication [55].

```
<LesHouchesEvents version="1.0">
<header>
  <generator_name>WHIZARD</generator_name>
  <generator_version>3.0.3</generator_version>
</header>
<init>
 25 25  5.0000000000E+02  5.0000000000E+02 -1 -1 -1 -1 3 1
  1.0000000000E-01  1.0000000000E-03  1.0000000000E+00 42
</init>
<event>
 4 42  3.0574068604E+08  1.0000000000E+03 -1.0000000000E+00 -1.0000000000E+00
 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00  4.8412291828E+02  5.0000000000E+02  1.2500000000E+02  0.0000000000E+00  9.00
 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02  1.2500000000E+02  0.0000000000E+00  9.00
 25 1 1 2 0 0 -1.4960220911E+02 -4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02  0.0000000000E+00  9.000
 25 1 1 2 0 0  1.4960220911E+02  4.6042825611E+02  0.0000000000E+00  5.0000000000E+02  1.2500000000E+02  0.0000000000E+00  9.000
</event>
</LesHouchesEvents>
```

Note that for the LHEF format, there are different versions according to the different stages of agreement. They can be addressed from within the SINDARIN file by setting the string variable $lhef_version to one of (at the moment) three values: "1.0", "2.0", or "3.0". The examples above corresponds (as is indicated in the header) to the version "1.0" of the LHEF format. Additional information in form of alternative squared matrix elements or event weights in the event are the most prominent features of the other two more advanced versions. For more details confer the literature.

Sample files for the default ASCII format as well as for the debug event format are shown in the appendix.

11.6   Interfaces to Parton Showers, Matching and Hadronization

This section describes the interfaces to the internal parton shower as well as the parton shower and hadronization routines from PYTHIA. Moreover, our implementation of the MLM matching

making use of the parton showers is described. Sample `SINDARIN` files are located in the `share/examples` directory. All input files come in two versions, one using the internal shower, ending in `W.sin`, and one using `PYTHIA`'s shower, ending in `P.sin`. Thus we state all file names as ending with `X.sin`, where `X` has to be replaced by either `W` or `P`. The input files include `EENoMatchingX.sin` and `DrellYanNoMatchingX.sin` for $e^+e^- \rightarrow hadrons$ and $p\bar{p} \rightarrow Z$ without matching. The corresponding `SINDARIN` files with matching enabled are `EEMatching2X.sin` to `EEMatching5X.sin` for $e^+e^- \rightarrow hadrons$ with a different number of partons included in the matrix element and `DrallYanMatchingX.sin` for Drell-Yan with one matched emission.

11.6.1 Parton Showers and Hadronization

From version 2.1 onwards, `WHIZARD` contains an implementation of an analytic parton shower as presented in [74], providing the opportunity to perform the parton shower from whithin `WHIZARD`. Moreover, an interface to `PYTHIA` is included, which can be used to delegate the parton shower to `PYTHIA`. The same interface can be used to hadronize events using the generated events using `PYTHIA`'s hadronization routines. Note that by `PYTHIA`'s default, when performing initial-state radiation multiple interactions are included and when performing the hadronization hadronic decays are included. If required, these additional steps have to be switched off using the corresponding arguments for `PYTHIA`'s `PYGIVE` routine via the `$ps_PYTHIA_PYGIVE` string.

Note that from version 2.2.4 on the earlier flag `--enable-shower` flag has been abandoned, and there is only a flag to either compile or not compile the interally attached `PYTHIA6` package (`--enable-pythia6`) last release of the `Fortran` `PYTHIA`, v6.427) as well as the interface. It can be invoked by the following `SINDARIN` keywords:

`?ps_fsr_active = true`	master switch for final-state parton showers
`?ps_isr_active = true`	master switch for initial-state parton showers
`?ps_taudec_active = true`	master switch for τ decays (at the moment only via `TAUOLA`
`?hadronization_active = true`	master switch to enable hadronization
`$shower_method = "PYTHIA6"`	switch to use `PYTHIA6`'s parton shower instead of `WHIZARD`'s own shower

If either `?ps_fsr_active` or `?ps_isr_active` is set to `true`, the event will be transferred to the internal shower routines or the `PYTHIA` data structures, and the chosen shower steps (initial- and final-state radiation) will be performed. If hadronization is enabled via the `?hadronization_active` switch, `WHIZARD` will call `PYTHIA`'s hadronization routine. The hadronization can be applied to events showered using the internal shower or showered using `PYTHIA`'s shower routines, as well as unshowered events. Any necessary transfer of event data to `PYTHIA` is automatically taken care of within `WHIZARD`'s shower interface. The resulting (showered and/or hadronized) event will be transferred back to `WHIZARD`, the former final particles will be marked as intermediate. The analysis can be applied to a showered and/or hadronized event just like in the unshowered/unhadronized case. Any event file can be used and will contain the showered/hadronized event.

Settings for the internal analytic parton shower are set via the following `SINDARIN` variables:

ps_mass_cutoff The cut-off in virtuality, below which, partons are assumed to radiate no more. Used for both ISR and FSR. Given in GeV. (Default = 1.0)

ps_fsr_lambda The value for Λ used in calculating the value of the running coupling constant α_S for Final State Radiation. Given in GeV. (Default = 0.29)

ps_isr_lambda The value for Λ used in calculating the value of the running coupling constant α_S for Initial State Radiation. Given in GeV. (Default = 0.29)

ps_max_n_flavors Number of quark flavours taken into account during shower evolution. Meaningful choices are 3 to include u, d, s-quarks, 4 to include u, d, s, c-quarks and 5 to include u, d, s, c, b-quarks. (Default = 5)

?ps_isr_alphas_running Switch to decide between a constant α_S, given by **ps_fixed_alphas**, and a running α_S, calculated using **ps_isr_lambda** for ISR. (Default = true)

?ps_fsr_alphas_running Switch to decide between a constant α_S, given by **ps_fixed_alphas**, and a running α_S, calculated using **ps_fsr_lambda** for FSR. (Default = true)

ps_fixed_alphas Fixed value of α_S for the parton shower. Used if either one of the variables **?ps_fsr_alphas_running** or **?ps_isr_alphas_running** are set to **false**. (Default = 0.0)

?ps_isr_angular_ordered Switch for angular ordered ISR. (Default = true)[2]

ps_isr_primordial_kt_width The width in GeV of the Gaussian assumed to describe the transverse momentum of partons inside the proton. Other shapes are not yet implemented. (Default = 0.0)

ps_isr_primordial_kt_cutoff The maximal transverse momentum in GeV of a parton inside the proton. Used as a cut-off for the Gaussian. (Default = 5.0)

ps_isr_z_cutoff Maximal z-value in initial state branchings. (Default = 0.999)

ps_isr_minenergy Minimal energy in GeV of an emitted timelike or final parton. Note that the energy is not calculated in the labframe but in the center-of-mas frame of the two most initial partons resolved so far, so deviations may occur. (Default = 1.0)

ps_isr_tscalefactor Factor for the starting scale in the initial state shower evolution. (Default = 1.0)

?ps_isr_only_onshell_emitted_partons Switch to allow only for on-shell emitted partons, thereby rejecting all possible final state parton showers starting from partons emitted during the ISR. (Default = false)

---

[2]The FSR is always simulated with angular ordering enabled.

Settings for the `PYTHIA` are transferred using the following `SINDARIN` variables:

`?ps_PYTHIA_verbose`	if set to false, output from `PYTHIA` will be suppressed
`$ps_PYTHIA_PYGIVE`	a string containing settings transferred to `PYTHIA`'s `PYGIVE` subroutine. The format is explained in the `PYTHIA` manual. The limitation to 100 characters mentioned there does not apply here, the string is split appropriately before being transferred to `PYTHIA`.

Note that the included version of `PYTHIA` uses `LHAPDF` for initial state radiation whenever this is available, but the PDF set has to be set manually in that case using the keyword `ps_PYTHIA_PYGIVE`.

11.6.2 Parton shower – Matrix Element Matching

Along with the inclusion of the parton showers, `WHIZARD` includes an implementation of the MLM matching procedure. For a detailed description of the implemented steps see [74]. The inclusion of MLM matching still demands some manual settings in the `SINDARIN` file. For a given base process and a matching of N additional jets, all processes that can be obtained by attaching up to N QCD splittings, either a quark emitting a gluon or a gluon splitting into two quarks ar two gluons, have to be manually specified as additional processes. These additional processes need to be included in the `simulate` statement along with the original process. The `SINDARIN` variable `mlm_nmaxMEjets` has to be set to the maximum number of additional jets N. Moreover additional cuts have to be specified for the additional processes.

```
alias quark = u:d:s:c
alias antiq = U:D:S:C
alias j = quark:antiq:g

?mlm_matching = true
mlm_ptmin = 5 GeV
mlm_etamax = 2.5
mlm_Rmin = 1

cuts = all Dist > mlm_Rmin [j, j]
       and all Pt > mlm_ptmin [j]
       and all abs(Eta) < mlm_etamax [j]
```

Note that the variables `mlm_ptmin`, `mlm_etamax` and `mlm_Rmin` are used by the matching routine. Thus, replacing the variables in the `cut` expression and omitting the assignment would destroy the matching procedure.

The complete list of variables introduced to steer the matching procedure is as follows:

`?mlm_matching_active` Master switch to enable MLM matching. (Default = false)

`mlm_ptmin` Minimal transverse momentum, also used in the definition of a jet

`mlm_etamax` Maximal absolute value of pseudorapidity η, also used in defining a jet

`mlm_Rmin` Minimal $\eta - \phi$ distance R_{min}

`mlm_nmaxMEjets` Maximum number of jets N

`mlm_ETclusfactor` Factor to vary the jet definition. Should be ≥ 1 for complete coverage of phase space. (Default $= 1$)

`mlm_ETclusminE` Minimal energy in the variation of the jet definition

`mlm_etaclusfactor` Factor in the variation of the jet definition. Should be ≤ 1 for complete coverage of phase space. (Default $= 1$)

`mlm_Rclusfactor` Factor in the variation of the jet definition. Should be ≥ 1 for complete coverage of phase space. (Default $= 1$)

The variation of the jet definition is a tool to asses systematic uncertainties introduced by the matching procedure (See section 3.1 in [74]).

11.7   Rescanning and recalculating events

In the simplest mode of execution, `WHIZARD` handles its events at the point where they are generated. It can apply event transforms such as decays or shower (see above), it can analyze the events, calculate and plot observables, and it can output them to file. However, it is also possible to apply two different operations to those events in parallel, or to reconsider and rescan an event sample that has been previously generated.

We first discuss the possibilities that `simulate` offers. For each event, `WHIZARD` calculates the matrix element for the hard interaction, supplements this by Jacobian and phase-space factors in order to obtain the event weight, optionally applies a rejection step in order to gather uniformly weighted events, and applies the cuts and analysis setup. We may ask about the event matrix element or weight, or the analysis result, that we would have obtained for a different setting. To this end, there is an `alt_setup` option.

This option allows us to recalculate, event by event, the matrix element, weight, or analysis contribution with a different parameter set but identical kinematics. For instance, we may evaluate a distribution for both zero and non-zero anomalous coupling `fw` and enter some observable in separate histograms:

```
simulate (some_proc) {
    fw = 0
    analysis = record hist1 (eval Pt [H])
  alt_setup = {
    fw = 0.01
    analysis = record hist2 (eval Pt [H])
  }
}
```

In fact, the `alt_setup` object is not restricted to a single code block (enclosed in curly braces) but can take a list of those,

```
alt_setup = { fw = 0.01 }, { fw = 0.02 }, ...
```

Each block provides the environment for a separate evaluation of the event data. The generation of these events, i.e., their kinematics, is still steered by the primary environment.

The `alt_setup` blocks may modify various settings that affect the evaluation of an event, including physical parameters, PDF choice, cuts and analysis, output format, etc. This must not (i.e., cannot) affect the kinematics of an event, so don't modify particle masses. When applying cuts, they can only reduce the generated event sample, so they apply on top of the primary cuts for the simulation.

Alternatively, it is possible to `rescan` a sample that has been generated by a previous `simulate` command:

```
simulate (some_proc) { $sample = "my_events"
  analysis = record hist1 (eval Pt [H])
}
?update_sqme = true
?update_weight = true
rescan "my_events" (some_proc) {
  fw = 0.01
  analysis = record hist2 (eval Pt [H])
}
rescan "my_events" (some_proc) {
  fw = 0.05
  analysis = record hist3 (eval Pt [H])
}
```

In more complicated situation, rescanning is more transparent and offers greater flexibility than doing all operations at the very point of event generation.

Combining these features with the `scan` looping construct, we already cover a considerable range of applications. (There are limitations due to the fact that `SINDARIN` doesn't provide array objects, yet.) Note that the `rescan` construct also allows for an `alt_setup` option.

You may generate a new sample by rescanning, for which you may choose any output format:

```
rescan "my_events" (some_proc) {
  selection = all Pt > 100 GeV [H]
  $sample = "new_events"
  sample_format = lhef
}
```

The event sample that you rescan need not be an internal raw `WHIZARD` file, as above. You may rescan a LHEF file,

```
rescan "lhef_events" (proc) {
  $rescan_input_format = "lhef"
}
```

This file may have any origin, not necessarily from `WHIZARD`. To understand such an external file, `WHIZARD` must be able to reconstruct the hard process and match it to a process with a known name (e.g., `proc`), that has been defined in the `SINDARIN` script previously.

Within its limits, `WHIZARD` can thus be used for translating an event sample from one format to another format.

There are three important switches that control the rescanning behavior. They can be set or unset independently.

- `?update_sqme` (default: false). If true, `WHIZARD` will recalculate the hard matrix element for each event. When applying an analysis, the recalculated squared matrix element (averaged and summed over quantum numbers as usual) is available as the variable `sqme_prc`. This may be related to `sqme_ref`, the corresponding value in the event file, if available. (For the `alt_env` option, this switch is implied.)

- `?update_weight` (default: false). If true, `WHIZARD` will recalculate the event weight according to the current environment and apply this to the event. In particular, the user may apply a `reweight` expression. In an analysis, the new weight value is available as `weight_prc`, to be related to `weight_ref` from the sample. The updated weight will be applied for histograms and averages. An unweighted event sample will thus be transformed into a weighted event sample. (This switch is also implied for the `alt_env` option.)

- `?update_event` (default: false). If true, `WHIZARD` will generate a new decay chain etc., if applicable. That is, it reuses just the particles in the hard process. Otherwise, the complete event is kept as it is written to file.

For these options to make sense, `WHIZARD` must have access to a full process object, so the `SINDARIN` script must contain not just a definition but also a `compile` command for the matrix elements in question.

If an event file (other than raw format) contains several processes as a mixture, they must be identifiable by a numeric ID. `WHIZARD` will recognize the processes if their respective `SINDARIN` definitions contain appropriate `process_num_id` options, such as

```
process foo = u, ubar => d, dbar { process_num_id = 42 }
```

Certain event-file formats, such as LHEF, support alternative matrix-element values or weights. `WHIZARD` can thus write both original and recalculated matrix-element and weight values. Other formats support only a single event weight, so the `?update_weight` option is necessary for a visible effect.

External event files in formats such as LHEF, HepMC, or LCIO, also may carry information about the value of the strong coupling α_s and the energy scale of each event. This information will also be provided by `WHIZARD` when writing external event files. When such an event file is rescanned, the user has the choice to either user the α_s value that `WHIZARD` defines in the current context (or the method for obtaining an event-specific running α_s value), or override this for each event by using the value in the event file. The corresponding parameter is `?use_alphas_from_file`, which is false by default. Analogously, the parameter

`?use_scale_from_file` may be set to override the scale definition in the current context. Obviously, these settings influence matrix-element recalculation and therefore require `?update_sqme` to be set in order to become operational.

11.8 Negative weight events

For usage at NLO refer to Subsection 5.11.3. In case, you have some other mechanism to produce events with negative weights (e.g. with the `weight = <expr>` command), keep in mind that you should activate `?negative_weights = true` and `unweighted = false`. The generation of unweighted events with varying sign (also known as events and counter events) is currently not supported.

Chapter 12

Internal Data Visualization

12.1 GAMELAN

The data values and tables that we have introduced in the previous section can be visualized using built-in features of WHIZARD. To be precise, WHIZARD can write LaTeX code which incorporates code in the graphics language GAMELAN to produce a pretty-printed account of observables, histograms, and plots.

GAMELAN is a macro package for MetaPost, which is part of the TeX/LaTeX family. MetaPost, a derivative of Knuth's MetaFont language for font design, is usually bundled with the TeX distribution, but might need a separate switch for installation. The GAMELAN macros are contained in a subdirectory of the WHIZARD package. Upon installation, they will be installed in the appropriate directory, including the gamelan.sty driver for LaTeX. WHIZARD uses a subset of GAMELAN's graphics macros directly, but it allows for access to the full package if desired.

An (incomplete) manual for GAMELAN can be found in the share/doc subdirectory of the WHIZARD system. WHIZARD itself uses a subset of the GAMELAN capabilities, interfaced by SINDARIN commands and parameters. They are described in this chapter.

To process analysis output beyond writing tables to file, the write_analysis command described in the previous section should be replaced by compile_analysis, with the same syntax:

    compile_analysis (*analysis-tags*) { *options* }

where ***analysis-tags***, a comma-separated list of analysis objects, is optional. If there are no tags, all analysis objects are processed. The ***options*** script of local commands is also optional, of course.

This command will perform the following actions:

1. It writes a data file in default format, as write_analysis would do. The file name is given by $out_file, if nonempty. The file must not be already open, since the command needs a self-contained file, but the name is otherwise arbitrary. If the value of $out_file is empty, the default file name is whizard_analysis.dat.

2. It writes a driver file for the chosen datasets, whose name is derived from the data
   file by replacing the file extension of the data file with the extension `.tex`. The driver
   file is a LaTeX source file which contains embedded GAMELAN code that handles the
   selected graphics data. In the LaTeX document, there is a separate section for each
   contained dataset. Furthermore, a process-/analysis-specific makefile with the name
   `<process_name>_ana.makefile` is created that can be used to generate postscript or PDF
   output from the LaTeX source. If the steering flag `?analysis_file_only` is set to `true`,
   then the LaTeX file and the makefile are only written, but no execution of the makefile
   resulting in compilation of the LaTeX code (see the next item) is invoked.

3. As mentioned above, if the flag `?analysis_file_only` is set to `false` (which is the
   default), the driver file is processed by LaTeX(invoked by calling the makefile with the name
   `<process_name>_ana.makefile`), which generates an appropriate GAMELAN source file
   with extension `.mp`. This code is executed (calling GAMELAN/MetaPost, and again
   LaTeX for typesetting embedded labels). There is a second LaTeX pass (automatically done
   by the makefile) which collects the results, and finally conversion to PostScript and PDF
   formats.

   The resulting PostScript or PDF file – the file name is the name of the data file with the
extension replaced by `.ps` or `.pdf`, respectively – can be printed or viewed with an appropriate
viewer such as `gv`. The viewing command is not executed automatically by WHIZARD.

   Note that LaTeX will write further files with extensions `.log`, `.aux`, and `.dvi`, and GAMELAN
will produce auxiliary files with extensions `.ltp` and `.mpx`. The log file in particular, could
overwrite WHIZARD's log file if the basename is identical. Be careful to use a value for `$out_file`
which is not likely to cause name clashes.

12.1.1   User-specific changes

In the case, that the SINDARIN `compile_analysis` command is invoked and the flag named
`?analysis_file_only` is not changed from its default value `false`, WHIZARD calls the process-
/analysis-specific makefile triggering the compilation of the LaTeX code and the GAMELAN plots
and histograms. If the user wants to edit the analysis output, for example changing captions,
headlines, labels, properties of the plots, graphs and histograms using GAMELAN specials
etc., this is possible and the output can be regenerated using the makefile. The user can also
directly invoke the GAMELAN script, `whizard-gml`, that is installed in the binary directly along
with the WHIZARD binary and other scripts. Note however, that the LaTeX environment for the
specific style files have to be set by hand (the command line invocation in the makefile does this
automatically). Those style files are generally written into `share/texmf/whizard/` directory.
The user can execute the commands in the same way as denoted in the process-/analysis-specific
makefile by hand.

12.2 Histogram Display

12.3 Plot Display

12.4 Graphs

Graphs are an additional type of analysis object. In contrast to histograms and plots, they do not collect data directly, but they rather act as containers for graph elements, which are copies of existing histograms and plots. Their single purpose is to be displayed by the GAMELAN driver.

Graphs are declared by simple assignments such as

```
graph g1 = hist1
graph g2 = hist2 & hist3 & plot1
```

The first declaration copies a single histogram into the graph, the second one copies two histograms and a plot. The syntax for collecting analysis objects uses the `&` concatenation operator, analogous to string concatenation. In the assignment, the rhs must contain only histograms and plots. Further concatenating previously declared graphs is not supported.

After the graph has been declared, its contents can be written to file (`write_analysis`) or, usually, compiledd by the LaTeX/GAMELAN driver via the `compile_analysis` command.

The graph elements on the right-hand side of the graph assignment are copied with their current data content. This implies a well-defined order of statements: first, histograms and plots are declared, then they are filled via `record` commands or functions, and finally they can be collected for display by graph declarations.

A simple graph declaration without options as above is possible, but usually there are options which affect the graph display. There are two kinds of options: graph options and drawing options. Graph options apply to the graph as a whole (title, labels, etc.) and are placed in braces on the lhs of the assigment. Drawing options apply to the individual graph elements representing the contained histograms and plots, and are placed together with the graph element on the rhs of the assignment. Thus, the complete syntax for assigning multiple graph elements is

```
graph graph-tag { graph-options }
= graph-element-tag1 { drawing-options1 }
& graph-element-tag2 { drawing-options2 }
...
```

This form is recommended, but graph and drawing options can also be set as global parameters, as usual.

We list the supported graph and drawing options in Tables 12.1 and 12.2, respectively.

12.5 Drawing options

The options for coloring lines, filling curves, or choosing line styles make use of macros in the GAMELAN language. At this place, we do not intend to give a full account of the possiblities,

Table 12.1: *Graph options. The content of strings of type LaTeX must be valid LaTeX code (containing typesetting commands such as math mode). The content of strings of type GAMELAN must be valid GAMELAN code. If a graph bound is kept* undefined, *the actual graph bound is determined such as not to crop the graph contents in the selected direction.*

Variable	Default	Type	Meaning
`$title`	""	LaTeX	Title of the graph = subsection headline
`$description`	""	LaTeX	Description text for the graph
`$x_label`	""	LaTeX	x-axis label
`$y_label`	""	LaTeX	y-axis label
`graph_width_mm`	130	Integer	graph width (on paper) in mm
`graph_height_mm`	90	Integer	graph height (on paper) in mm
`?x_log`	false	Logical	Whether the x-axis scale is linear or logarithmic
`?y_log`	false	Logical	Whether the y-axis scale is linear or logarithmic
`x_min`	*undefined*	Real	Lower bound for the x axis
`x_max`	*undefined*	Real	Upper bound for the x axis
`y_min`	*undefined*	Real	Lower bound for the y axis
`y_max`	*undefined*	Real	Upper bound for the y axis
`gmlcode_bg`	""	GAMELAN	Code to be executed before drawing
`gmlcode_fg`	""	GAMELAN	Code to be executed after drawing

Table 12.2: *Drawing options. The content of strings of type GAMELAN must be valid GAMELAN code. The behavior w.r.t. the flags with* undefined *default value depends on the type of graph element. Histograms: draw baseline, piecewise, fill area, draw curve, no errors, no symbols; Plots: no baseline, no fill, draw curve, no errors, no symbols.*

Variable	Default	Type	Meaning
`?draw_base`	*undefined*	Logical	Whether to draw a baseline for the curve
`?draw_piecewise`	*undefined*	Logical	Whether to draw bins separately (histogram)
`?fill_curve`	*undefined*	Logical	Whether to fill area between baseline and curve
`$fill_options`	`""`	GAMELAN	Options for filling the area
`?draw_curve`	*undefined*	Logical	Whether to draw the curve as a line
`$draw_options`	`""`	GAMELAN	Options for drawing the line
`?draw_errors`	*undefined*	Logical	Whether to draw error bars for data points
`$err_options`	`""`	GAMELAN	Options for drawing the error bars
`?draw_symbols`	*undefined*	Logical	Whether to draw symbols at data points
`$symbol`	Black dot	GAMELAN	Symbol to be drawn
`gmlcode_bg`	`""`	GAMELAN	Code to be executed before drawing
`gmlcode_fg`	`""`	GAMELAN	Code to be executed after drawing

but we rather list a few basic features that are likely to be useful for drawing graphs.

Colors

GAMELAN knows about basic colors identified by name:

$$\texttt{black, white, red, green, blue, cyan, magenta, yellow}$$

More generically, colors in GAMELAN are RGB triplets of numbers (actually, numeric expressions) with values between 0 and 1, enclosed in brackets:

$$\texttt{(}r\texttt{, }g\texttt{, }b\texttt{)}$$

To draw an object in color, one should apply the construct `withcolor` *color* to its drawing code. The default color is always black. Thus, this will make a plot drawn in blue:

```
$draw_options = "withcolor blue"
```

and this will fill the drawing area of some histogram with an RGB color:

```
$fill_options = "withcolor (0.8, 0.7, 1)"
```

Dashes

By default, lines are drawn continuously. Optionally, they can be drawn using a *dash pattern*. Predefined dash patterns are

<div align="center">

`evenly`, `withdots`, `withdashdots`

</div>

Going beyond the predefined patterns, a generic dash pattern has the syntax

<div align="center">

`dashpattern (on` *l1* `off` *l2* `on` ...`)`

</div>

with an arbitrary repetition of `on` and `off` clauses. The numbers *l1*, *l2*, ... are lengths measured in pt.

To apply a dash pattern, the option syntax `dashed` *dash-pattern* should be used. Options strings can be concatenated. Here is how to draw in color with dashes:

```
$draw_options = "withcolor red dashed evenly"
```

and this draws error bars consisting of intermittent dashes and dots:

```
$err_options = "dashed (withdashdots scaled 0.5)"
```

The extra brackets ensure that the scale factor $1/2$ is applied only the dash pattern.

Hatching

Areas (e.g., below a histogram) can be filled with plain colors by the `withcolor` option. They can also be hatched by stripes, optionally rotated by some angle. The syntax is completely analogous to dashes. There are two predefined *hatch patterns*:

<div align="center">

`withstripes`, `withlines`

</div>

and a generic hatch pattern is written

<div align="center">

`hatchpattern (on` *w1* `off` *w2* `on` ...`)`

</div>

where the numbers *l1*, *l2*, ... determine the widths of the stripes, measured in pt.

When applying a hatch pattern, the pattern may be rotated by some angle (in degrees) and scaled. This looks like

```
$fill_options = "hatched (withstripes scaled 0.8 rotated 60)"
```

Smooth curves

Plot points are normally connected by straight lines. If data are acquired by statistical methods, such as Monte Carlo integration, this is usually recommended. However, if a plot is generated using an analytic mathematical formula, or with sufficient statistics to remove fluctuations, it might be appealing to connect lines by some smooth interpolation. GAMELAN can switch on spline interpolation by the specific drawing option `linked smoothly`. Note that the results can be surprising if the data points do have sizable fluctuations or sharp kinks.

Error bars

Plots and histograms can be drawn with error bars. For histograms, only vertical error bars are supported, while plot points can have error bars in x and y direction. Error bars are switched on by the `?draw_errors` flag.

There is an option to draw error bars with ticks: `withticks` and an alternative option to draw arrow heads: `witharrows`. These can be used in the `$err_options` string.

Symbols

To draw symbols at plot points (or histogram midpoints), the flag `?draw_symbols` has to be switched on.

Chapter 13

Fast Detector Simulation and External Analysis

Events from a Monte Carlo event generator are further used in an analysis, most often combined with a detector simulation. Event files from the generator are then classified whether they are (i) parton level (coming from the hard matrix element) for which mostly LHE or HepMC event formats are used, particle level (after parton shower and hadronization) - usually in HepMC or LCIO format -, or detector level objects. The latter is the realm of packages like ROOT or specific software from the experimental software frameworks. While detailed experimental studies take into account the best-possible detector description in a so-called full simulation via Geant which takes several seconds per event, fast studies are made with parameterized fast detector simulations like in Delphes or SGV. In the following, we discuss the options to interface external packages for these purposes or to pipe events from WHIZARD to such external packages.

13.1 Interfacing ROOT

One of the most distributed analysis framework is ROOT [101]. In WHIZARD for the moment there is no direct interface to the ROOT framework. The easiest way to write out particle-level events in the ROOT or RootTree format is to use WHIZARD's interface to HepMC3: this modern incarnation of the HepMC format has different writer classes, where the writer class for ROOT and RootTree files is supported by WHIZARD's HepMC3 interface. For this to work, one only has to make sure that HepMC3 has been built with ROOT support, and that the WHIZARD configure has to detect the ROOT setup on the computing environment. For more details cf. the installation section 2.2.9. If this has been successfully linked, then WHIZARD can use its own HepMC3 interface to write out ROOT or RootTree formats.

This can be done by setting the following options in the SINDARIN files:

```
$hepmc3_mode = "Root"
```

or

```
$hepmc3_mode = "RootTree"
```

For more details cf. the ROOT manual and documentation therein.

13.2   Interfacing RIVET

Rivet [102] is a very mighty analysis framework which has been developed to make experimental analyses from the LHC experiments available for non-collaboration members. It can be easily used to analyze events and produce high-quality plots for differential distributions and experimental observables. Since version 3 [103] there is now also a lot of functionality that comes very handy for plotting differential distributions at fixed order in NLO calculations, e.g. negative weights in bins or how to treat imperfectly balanced events and counterevents close to bin boundaries etc. For the moment, WHIZARD does not have a dedicated interface to Rivet, so the preferred method is to write out events, best in the HepMC or HepMC3 format and then read them into Rivet. A more sophisticated interface is foreseen for a future version of WHIZARD, while there are already development versions where WHIZARD detects all the Rivet infrastructure and libraries. But they are not yet used.

For more details and practical examples cf. the Rivet manual. This describes in detail especially the Rivet installation. A typical error that occurs on systems where no ROOT is installed (cf. Sec. 13.1) is the one these `Missing TPython.h` missing headers. Then Rivet can nevertheless be easily built without ROOT support by setting

```
--disable-root
```

in the `rivet-bootstrap` script. For an installation of Rivet it is favorable to include the location of the Rivet Python scripts in the PYTHONPATH environment variable. They can be accessed from the Rivet configuration script as

```
<path_to_rivet-config>/rivet-config --pythonpath
```

If the Python path is not known within the environment variables, then one commonly encounters error like `No module named rivet` or `Import error:  no module named yoda` when running Rivet scripts like e.g. `yodamerge`.

If you use a Rivet version older than `v3.1.1` there is no support for HepMC3 yet, so when using HepMC3 with WHIZARD please use the backwards compatibility mode of HepMC3in the SINDARIN file:

```
$hepmc3_mode = "HepMC2"
```

When using MPI parallelized runs of WHIZARD there will a large number of different `.hepmc` files (also if some grid architecture has produced these event files in junks). Then one has to first merge these event files.

Here, we quickly explain how to steer Rivet for your own analysis. For more details, please confer the Rivet manual.

1. The command

   ```
   rivet-mkanalysis <name>
   ```

   creates a template Rivet plugin for the analysis `<name>.cc`, a template info file `<name>.info` amd a template file for the plot generation `<name>.plot`. Note that this overwrites potentially existing files in this folder with the same name.

2. Now, analysis statements like e.g. cuts etc. can be implemented in `<name>.cc`. For analysis of parton-level events without parton showering, the cuts can be equivalent to those in `WHIZARD`, i.e. the generator-level cuts can be as strict as the analysis cuts to avoid generating unnecessary events. If parton showering is applied it is better to have looser generator than analysis cuts to avoid undesired plot artifacts.

3. Next, one executes the command (the shared library name might be different e.g. on Darwin or BSD OS)

    ```
    rivet-buildplugin Rivet<name>.so <name>.cc
    ```

    This creates an executable `Rivet` analysis library `Rivet<name>.so`. The custom analysis should now appear in the output of

    ```
    rivet --list <name>
    ```

    If this is not the case, the analysis path has to be exported first as `RIVET_ANALYSIS_PATH=$PWD`.

4. We are now ready to use the custom analysis to analyze the `.hepmc` events by executing the command

    ```
    rivet --pwd --analysis=<name> -o <outfile>.yoda <path/to/hepmcfiles>
    ```

    and save the produced histograms of the analysis in the `.yoda` format. In general the option `--ignore-beams` for `Rivet` should be used to prevent `Rivet` to stumble over beam remnants. This is also relevant for lepton collider processes with electron PDFs. For a large number of events, event files can become very big. To avoid writing them all on disk, a FIFO for the `<path/to/hepmcfiles>` can be used.

5. Different `yoda` files can now be merged into a single file using the command

    ```
    <yodamerge --add -o <name>_full.yoda <name>_01.yoda ...
    ```

    This should be applied e.g. for the case of fixed-order NLO differential distributions where Born, real and virtual components have been generated separately.

6. Finally, plots can be produced: after listing all the histograms to be plotted in the plot file `<name>.plot`, the command

    ```
    rivet-mkhtml <name>_full.yoda
    ```

    translates the `.yoda` file into a histogram file in the `.dat` format. These plots can either be visually enhanced by modifying the `<name>.plot` file as is described on the webpage https://rivet.hepforge.org/make-plots.html, or by using any other external plotting tool like e.g. `Gnuplot` for the `.dat` files.

Clearly, this gives only a rough sketch on how to use `Rivet` for an analysis. For more details, please consult the `Rivet` webpage and the `Rivet` manual.

13.3   Fast Detector Simulation with DELPHES

Fast detector simulation allows relatively quick checks whether experimental analyses actually work in a semi-realistic detector study. There are some older tools for fast simulation like e.g. `PGS` (which is no longer actively maintained) and `SGV` which is default fast simulation for ILC studies. For LHC and general future hadron collider studies, `Delphes` [104] is the most commonly used tool for fast detector simulation.

The details on how to obtain and build `Delphes` can be obtained from their webpage, https://cp3.irmp.ucl.ac.be/projects/delphes. It depends both on `Tcl/Tk` as well as `ROOT` (cf. Sec. 13.1. Interfacing any Monte Carlo event generator with a fast detector simulation like `Delphes` is rather trivial: `Delphes` ships with up to five executables

```
DelphesHepMC
DelphesLHEF
DelphesPythia8
DelphesROOT
DelphesSTDHEP
```

`DelphesPythia8` is a direct interface between `PYTHIA8` and `Delphes`, so detector-level events are directly produced via an API interface between `PYTHIA8` and `Delphes`. This is the most convenient method which is foreseen for `WHIZARD`, however not yet implemented. The other four binaries take input files in the `HepMC`, `LHE`, `STDHEP` and `ROOT` format, apply a fast detector simulation according to the chosen input file and give a `ROOT` detector-level event file as output.

Executing one of the binaries above without options, the following message will be displayed:

```
./DelphesHepMC
Usage: DelphesHepMC config_file output_file [input_file(s)]
config_file - configuration file in Tcl format,
output_file - output file in ROOT format,
input_file(s) - input file(s) in HepMC format,
with no input_file, or when input_file is -, read standard input.
```

Using `Delphes` with `HepMC` event files then works as

```
./DelphesHepMC cards/delphes_card_ATLAS.tcl output.root input.hepmc
```

For `STDHEP` files which are directly by `WHIZARD` without external packages (only assuming that the XDR C libraries are present on the system), execute

```
./DelphesSTDHEP cards/delphes_card_ILD.tcl delphes_output.root input.hep
```

For LHE files as input, use

```
./DelphesLHEF cards/delphes_card_CLICdet_Stage1.tcl delphes_output.root input.lhef
```

and for `ROOT` (particle-level) files use

```
./DelphesROOT cards/delphes_card_CMS.tcl delphes_output.root input.root
```

In the `Delphes` cards directory, there is a long list of supported input files for existing and future detectors, a few of which we have displayed here.

`Delphes` detector-level output files can then be analyzed with `ROOT` as described in the `Delphes` manual.

Chapter 14

User Interfaces for WHIZARD

14.1 Command Line and SINDARIN Input Files

The standard way of using WHIZARD involves a command script written in SINDARIN. This script is executed by WHIZARD by mentioning it on the command line:

```
whizard script-name.sin
```

You may specify several script files on the command line; they will be executed consecutively.

If there is no script file, WHIZARD will read commands from standard input. Hence, this is equivalent:

```
cat script-name.sin | whizard
```

When executed from the command line, WHIZARD accepts several options. They are given in long form, i.e., they begin with two dashes. Values that belong to options follow the option string, separated either by whitespace or by an equals sign. Hence, `--prefix /usr` and `--prefix=/usr` are equivalent. Some options are also available in short form, a single dash with a single letter. Short-form options can be concatenated, i.e., a dash followed by several option letters.

The first set of options is intended for normal operation.

`--debug AREA` : Switch on debug output for `AREA`. `AREA` can be one of WHIZARD's source directories or `all`.

`--debug2 AREA` : Switch on more verbose debug output for `AREA`.

`--single-event` : Only compute one phase-space point (for debugging).

`--execute COMMANDS` : Execute `COMMANDS` as a script before the script file (see below). Short version: `-e`

`--file CMDFILE` : Execute commands in `CMDFILE` before the main script file (see below). Short version: `-f`

`--help` : List the available options and exit. Short version: `-h`

--interactive : Run WHIZARD interactively. See Sec. 14.2. Short version: -i.

--library LIB : Preload process library LIB (instead of the default processes). Short version: -l.

--localprefix DIR : Search in DIR for local models. Default is $HOME/.whizard.

--logfile FILE : Write log to FILE. Default is whizard.log. Short version: -L.

--logging : Start logging on startup (default).

--model MODEL : Preload model MODEL. Default is the Standard Model SM. Short version: -m.

--no-banner : Do not display banner at startup.

--no-library : Do not preload a library.

--no-logfile : Do not write a logfile.

--no-logging : Do not issue information into the logfile.

--no-model : Do not preload a specific physics model.

--no-rebuild : Do not force a rebuild.

--query VARIABLE : Display documentation of VARIABLE. Short version: -q.

--rebuild : Do not preload a process library and do all calculations from scratch, even if results exist. This combines all rebuild options. Short version: -r.

--rebuild-library : Rebuild the process library, even if code exists.

--rebuild-phase-space : Rebuild the phase space setup, even if it exists.

--rebuild-grids : Redo the integration, even if previous grids and results exist.

--rebuild-events : Redo event generation, discarding previous event files.

--show-config : Show build-time configuration.

--version : Print version information and exit. Short version: -V.

- : Any further options are interpreted as file names.

The second set of options refers to the configuration. They are relevant when dealing with a relocated WHIZARD installation, e.g., on a batch systems.

--prefix DIR : Specify the actual location of the WHIZARD installation, including all subdirectories.

`--exec-prefix DIR` : Specify the actual location of the machine-specific parts of the `WHIZARD` installation (rarely needed).

`--bindir DIR` : Specify the actual location of the executables contained in the `WHIZARD` installation (rarely needed).

`--libdir DIR` : Specify the actual location of the libraries contained in the `WHIZARD` installation (rarely needed).

`--includedir DIR` : Specify the actual location of the include files contained in the `WHIZARD` installation (rarely needed).

`--datarootdir DIR` : Specify the actual location of the data files contained in the `WHIZARD` installation (rarely needed).

`--libtool LOCAL_LIBTOOL` : Specify the actual location and name of the `libtool` script that should be used by `WHIZARD`.

`--lhapdfdir DIR` : Specify the actual location and of the `LHAPDF` installation that should be used by `WHIZARD`.

The `--execute` and `--file` options allow for fine-tuning the command flow. The `WHIZARD` main program will concatenate all commands given in `--execute` commands together with all commands contained in `--file` options, in the order they are encountered, as a contiguous command stream that is executed *before* the main script (in the example above, `script-name.sin`).

Regarding the `--execute` option, commands that contain blanks must be enclosed in matching single- or double-quote characters since the individual tokens would otherwise be intepreted as separate option strings. Unfortunately, a Unix/Linux shell interpreter will strip quotes before handing the command string over to the program. In that situation, the quote-characters must be quoted themselves, or the string must be enclosed in quotes twice. Either version should work as a command line interpreted by the shell:

```
whizard --execute \'int my_flag = 1\' script-name.sin
whizard --execute "'int my_flag = 1'" script-name.sin
```

14.2   WHISH – The `WHIZARD` Shell/Interactive mode

`WHIZARD` can be also run in the interactive mode using its own shell environment. This is called the `WHIZARD` Shell (WHISH). For this purpose, one starts with the command

```
/home/user$ whizard --interactive
```

or

```
/home/user$ whizard -i
```

`WHIZARD` will preload the Standard Model and display a command prompt:

```
whish?
```

You now can enter one or more `SINDARIN` commands, just as if they were contained in a script file. The commands are compiled and executed after you hit the ENTER key. When done, you get a new prompt. The WHISH can be closed by the `quit` command:

```
whish? quit
```

Obviously, each input must be self-contained: commands must be complete, and conditionals or scans must be closed on the same line.

If `WHIZARD` is run without options and without a script file, it also reads commands interactively, from standard input. The difference is that in this case, interactive input is multi-line, terminated by `Ctrl-D`, the script is then compiled and executed as a whole, and `WHIZARD` terminates.

In WHISH mode, each input line is compiled and executed individually. Furthermore, fatal errors are masked, so in case of error the program does not terminate but returns to the WHISH command line. (The attempt to recover may fail in some circumstances, however.)

14.3   Graphical user interface

This is still experimental.

`WHIZARD` ships with a graphical interface that can be steered in a browser of your choice. It is located in `share/gui`. To use it, you have to run `npm install` (which will install javascript libraries locally in that folder) and `npm start` (which will start a local web server on your machine) in that folder. More technical details and how to get `npm` is discussed in `share/gui/README.md`. When it is running, you can access the GUI by entering `localhost:3000` as address in your browser. The GUI is separated into different tabs for basic settings, integration, simulation, cuts, scans, NLO and beams. You can select and enter what you are interested in and the GUI will produce a `SINDARIN` file. You can use the GUI to run WHIZARD with that `SINDARIN` or just produce it with the GUI and then tweak it further with an editor. In case you run it in the GUI, the log file will be updated in the browser as it is produced. Any `SINDARIN` features that are not supported by the GUI can be added directly as "Additional Code".

14.4   `WHIZARD` as a library

The compiled `WHIZARD` program consists of two libraries (`libwhizard` and `libomega`). In the standard setup, these are linked to a short main program which deals with command line options and top-level administration. This is the stand-alone `whizard` executable program.

Alternatively, it is possible to link the libraries to a different main program of the user's choice. The user program can take complete control of the `WHIZARD` features. The `libwhizard` library provides an API, a well-defined set of procedures which can be called from a foreign main program. The supported languages are `Fortran`, `C`, and `C++`. Using the C API, any other language which supports linking against C libraries can also be interfaced.

14.4.1   Fortran main program

To link a `Fortran` main program with the `WHIZARD` library, the following steps must be performed:

1. Configure, build and install `WHIZARD` as normal.

2. Include code for accessing `WHIZARD` functionality in the user program. The code should initialize `WHIZARD`, execute the intended commands, and finalize. For an example, see below.

3. Compile the user program. The user program must be compiled with the same `Fortran` compiler that has been used for the `WHIZARD` build.

   If necessary, specify an option that finds the installed `WHIZARD` module files. For instance, if `WHIZARD` has been installed in `whizard-path`, this should read

   ```
   -Iwhizard-path/lib/mod/whizard
   ```

4. Link the program (or compile-link in a single step). If necessary, specify options that find the installed `WHIZARD` and `O'Mega` libraries. For instance, if `WHIZARD` has been installed in `whizard-path`, this should read

   ```
   -Lwhizard-path/lib -lwhizard -lwhizard_prebuilt -lomega
   ```

   On some systems, you may have to replace `lib` by `lib64`.

   Such an example compile-link could look like

   ```
   gfortran manual_example_api.f90 -Lwhizard-path/lib -lwhizard -lwhizard_prebuilt -lomega
   ```

   If `WHIZARD` has been compiled with a non-default `Fortran` compiler, you may have to explicitly link the appropriate `Fortran` run-time libraries.

   The `tirpc` library is used by the `StdHEP` subsystem for `xdr` functionality. This library should be present on the host system. This library needs only be linked of the SunRPC library is not installed on the system.

   If additional libraries such as `HepMC` are enabled in the `WHIZARD` configuration, it may be necessary to provide extra options for linking those.

   An example here looks like

   ```
   gfortran manual_example_api.f90 -Lwhizard-path/lib -lwhizard
        -lwhizard_prebuilt -lomega -lHepMC3 -lHepMC3rootIO -llcio
   ```

5. Run the program. If necessary, provide the path to the installed shared libraries. For instance, if `WHIZARD` has been installed in `whizard-path`, this should read

   ```
   export LD_LIBRARY_PATH="whizard-path/lib:$LD_LIBRARY_PATH"
   ```

   On some systems, you may have to replace `lib` by `lib64`, as above.

   The `WHIZARD` subsystem will work with input and output files in the current working directory, unless asked to do otherwise.

Below is an example program, adapted from WHIZARD's internal unit-test suite. The user program controls the WHIZARD workflow in the same way as a SINDARIN script would do. The commands are a mixture of SINDARIN command calls and functionality for passing information between the WHIZARD subsystem and the host program. In particular, the program can process generated events one-by-one.

```
program main

  ! WHIZARD API as a module
  use api

  ! Standard numeric types
  use iso_fortran_env, only: real64, int32

  implicit none

  ! WHIZARD and event-sample objects
  type(whizard_api_t)    :: whizard
  type(simulation_api_t) :: sample

  ! Local variables
  real(real64)   :: integral, error
  real(real64)   :: sqme, weight
  integer(int32) :: idx
  integer(int32) :: i, it_begin, it_end

  ! Initialize WHIZARD, setting some global option
  call whizard%option ("model", "QED")
  call whizard%init ()

  ! Define a process, set some variables
  call whizard%command ("process mupair = e1, E1 => e2, E2")
  call whizard%set_var ("sqrts", 100._real64)
  call whizard%set_var ("seed", 0)

  ! Generate matrix-element code, integrate and retrieve result
  call whizard%command ("integrate (mupair)")
  call whizard%get_integration_result ("mupair", integral, error)

  ! Print result
  print 1, "cross section =", integral / 1000, "pb"
  print 1, "error         =", error    / 1000, "pb"
1 format (2x,A,1x,F5.1,1x,A)
2 format (2x,A,1x,L1)

  ! Settings for event generation
  call whizard%set_var ("$sample", "mupair_events")
  call whizard%set_var ("n_events", 2)

  ! Create an event-sample object and generate events
  call whizard%new_sample ("mupair", sample)
  call sample%open (it_begin, it_end)
```

```
      do i = it_begin, it_end
         call sample%next_event ()
         call sample%get_event_index (idx)
         call sample%get_weight (weight)
         call sample%get_sqme (sqme)
         print "(A,I0)", "Event #", idx
         print 3, "sqme    =", sqme
         print 3, "weight  =", weight
  3      format (2x,A,1x,ES10.3)
      end do

      ! Finalize the event-sample object
      call sample%close ()

      ! Finalize the WHIZARD object
      call whizard%final ()

    end program main
```

The API provides the following commands as `Fortran` subroutines. Most of them are used in the example above.

Module

There is only one module from the `WHIZARD` package which must be `use`d by the user program:

```
    use api
```

You may `use` any other `WHIZARD` module in our program, all module files are part of the installation. Be aware, however, that all other modules are considered internal. Unless explictly mentioned in this manual, interfaces are not documented here and may change between versions.

Changes to the `api` module, if any, will be documented here.

Master object

All functionality is accessed via a master API object which should be declared as follows:

```
    type(whizard_api_t) ::  whizard
```

There should be only one master object.

Pre-Initialization options

Before initializing the API object, it is possible to provide options. The available options mirror the command-line options of the stand-alone program, cf. Sec. 14.1.

```
    call whizard%option (key, value)
```

All keys and values are `Fortran` character strings. The following options are available. For all options, default values exist as listed in Sec. 14.1.

`model` Model that should be preloaded.

`library` Name of the library where matrix-element code should end up.

`logfile` Name of the logfile that `WHIZARD` will write.

`job_id` Name of the current job; can be used for writing unique output files.

`unpack` Comma-separated list of files to be uncompressed and unpacked (via `tar` and `gzip`) when `init` is called on the API object.

`pack` Comma-separated list of files or directories to be packed and compressed when `final` is called.

`rebuild` All of the following:

`rebuild_library` Force rebuilding a matrix-element code library, overwriting results from a previous run.

`recompile` Force recompiling the matrix-element code library.

`rebuild_grids` Force reproducing integration passes.

`rebuild_events` Force regenerating event samples.

Initialization and finalization

After options have been set, the system is initialized via

```
call whizard%init
```

Once initialized, `WHIZARD` can execute commands as listed below. When this is complete, clean up by

```
call whizard%final
```

Variables and values

In the API, `WHIZARD` requires numeric data types according to the IEEE standard, which is available to `Fortran` in the `iso_fortran_env` intrinsic module. Strictly speaking, integer data must have type `int32`, and real data must have type `real64`.

For most systems and default compiler settings, it is not really necessary to `use` the ISO module and its data types. Integers map to default `Fortran integer`, and real values map to default `Fortran double precision`.

As an alternative, you may `use` the `WHIZARD` internal `kinds` module which declares a `real(default)` type

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

On most systems, this will be equivalent to `real(real64)`.

To set a `SINDARIN` variable, use the function that corresponds to the data type:

```
call whizard%set_var (name, value)
```

The name is a `Fortran` string which has to be equal to the name of the corresponding `SINDARIN` variable, including any prefix character ($ or ?). The value depends on the type of the `SINDARIN` variable.

To retrieve the current value of a variable:

```
call whizard%get_var (name, var)
```

The variable must be declared as `integer`, `real(real64)`, `logical`, or `character(:)`, `allocatable`. This depends on the `SINDARIN` variable type.

Commands

Any `SINDARIN` command can be called via

```
call whizard%command (command)
```

command is a `Fortran` character string, as it would appear in a `SINDARIN` script.

This includes, in particular, the important commands `process`, `integrate`, and `simulate`. You may also set variables that way.

Retrieving cross-section results

This call returns the results (integration and error) from a preceding integration run for the process *process-name*:

```
call whizard%get_integration_result ("process-name", integral, error)
```

There is also an optional argument `known` of type `logical` which is set if the integration run was successful, so integral and error are meaningful.

Event-sample object

A `simulate` command will produce an event sample. With the appropriate settings, the sample will be written to file in any chosen format, to be post-processed when it is complete.

However, a possible purpose of using the `WHIZARD` API is to process events one-by-one when they are generated. To this end, there is an event-sample handle, which can be declared in this way:

```
type(simulation_api_t) ::  sample
```

An instance `sample` of this type is created by this factory method:

```
call whizard%new_sample ("process-name(s)", sample)
```

The command accepts a comma-separated list of process names which should be included in the event sample.

To start event generation for this sample, call

```
call sample%open (it_begin, it_end )
```

where the two output parameters (integers) *it_begin* and *it_end* provide the bounds for an event loop in the calling program. (In serial mode, the bounds are equal to 1 and `n_events`, respectively, but in an MPI parallel environment, they depend on the computing node.)

This command generates a new event, to be enclosed within an event loop:

```
call sample%next_event
```

The event will be available by format-specific access methods, see below.

This command closes and deletes an event sample after the event loop has completed:

```
call sample%close
```

Retrieving event data

After a call to `next_event`, the sample object can be queried for specific event data.

```
call sample%get_event_index (value)
```

returns the index (integer counter) of the current event.

```
call sample%get_process_index (value)
call sample%get_process_id (value)
```

returns the numeric (string) ID of the hard process, respectively, that was generated in this event. The variables must be declared as `integer` and `character(:), allocatable`, respectively.

The following methods return `real(real64)` values.

```
call sample%get_sqrts (value)
```

returns the $\sqrt{s}$ value of this event.

```
call sample%get_fac_scale (value)
```

returns the factorization scale of this event (*value*).

```
call sample%get_alpha_s (value)
```

returns the value of the strong coupling for this event (*value*).

```
call sample%get_sqme (value)
```

returns the value of the squared matrix element (summed over final states and averaged over initial states).

```
call sample%get_weight (value)
```

returns the Monte-Carlo weight of this event.

Access to the event record depends on the event format that has been selected. The format must allow access to individual events via data structures in memory. There are three cases where such structures exist and are accessible:

1. If the event format uses a COMMON block, event data is accessible via this COMMON block, which must be declared in the calling routine.

2. The HepMC event format communicates via a C++ object. In Fortran, there is a wrapper which has to be declared as

   ```
   type(hepmc_event_t) ::  hepmc_event
   ```

   To activate this handle, the next_event call must reference it as an argument:

   ```
   call sample%next_event (hepmc_event)
   ```

   The WHIZARD module hepmc_interface contains procedures which can work with this record. A pointer to the actual C++ object can be retrieved as a Fortran c_ptr object as follows:

   ```
   type(c_ptr) ::  hepmc_ptr
   ...
   hepmc_ptr = hepmc_event_get_c_ptr (hepmc_event)
   ```

3. The LCIO event format also communicates via a C++ object. The access methods are entirely analogous, replacing hepmc by lcio in all calls and names.

14.4.2   C main program

To link a C main program with the WHIZARD library, the following steps must be performed:

1. Configure, build and install WHIZARD as normal.

2. Include code for accessing WHIZARD functionality in the user program. The code should initialize WHIZARD, execute the intended commands, and finalize. For an example, see below.

3. Compile the user program with the option that finds the WHIZARD C/C++ interface header file. For instance, if WHIZARD has been installed in whizard-path, this should read

   ```
   -Iwhizard-path/include
   ```

4. Link the program with the necessary libraries (or compile-link in a single step). If WHIZARD has been installed in a system path, this should work automatically. If WHIZARD has been installed in a non-default whizard-path, these are the options:

```
-Lwhizard-path/lib -lwhizard -lwhizard_prebuilt -lomega -ltirpc
```

On some systems, you may have to replace `lib` by `lib64`.

If WHIZARD has been compiled with a non-default `Fortran` compiler, you may have to explicitly link the appropriate `Fortran` run-time libraries.

The `tirpc` library is used by the `StdHEP` subsystem for `xdr` functionality. This library should be present on the host system. Cf. the corresponding remarks in the section for a `Fortran` main program.

If additional libraries such as `HepMC` are enabled in the WHIZARD configuration, it may be necessary to provide extra options for linking those.

5. Run the program. If necessary, provide the path to the installed shared libraries. For instance, if WHIZARD has been installed in `whizard-path`, this should read

```
export LD_LIBRARY_PATH="whizard-path/lib:$LD_LIBRARY_PATH"
```

On some systems, you may have to replace `lib` by `lib64`, as above.

The WHIZARD subsystem will work with input and output files in the current working directory, unless asked to do otherwise.

Below is an example program, adapted from WHIZARD's internal unit-test suite. The user program controls the WHIZARD workflow in the same way as a SINDARIN script would do. The commands are a mixture of SINDARIN command calls and functionality for passing information between the WHIZARD subsystem and the host program. In particular, the program can process generated events one-by-one.

```c
#include <stdio.h>
#include "whizard.h"

int main( int argc, char* argv[] )
{
  /* WHIZARD and event-sample objects */
  void* wh;
  void* sample;

  /* Local variables */
  double integral, error;
  double sqme, weight;
  int idx;
  int it, it_begin, it_end;

  /* Initialize WHIZARD, setting some global option */
  whizard_create( &wh );
  whizard_option( &wh, "model", "QED" );
  whizard_init( &wh );

  /* Define a process, set some variables */
  whizard_command( &wh, "process mupair = e1, E1 => e2, E2" );
  whizard_set_double( &wh, "sqrts", 10. );
```

```
    whizard_set_int( &wh, "seed", 0 );

    /* Generate matrix-element code, integrate and retrieve result */
    whizard_command( &wh, "integrate (mupair)" );

    /* Print result */
    whizard_get_integration_result( &wh, "mupair", &integral, &error);
    printf( "  cross section = %5.1f pb\n", integral / 1000. );
    printf( "  error         = %5.1f pb\n", error / 1000. );

    /* Settings for event generation */
    whizard_set_char( &wh, "$sample", "mupair_events" );
    whizard_set_int( &wh, "n_events", 2 );

    /* Create an event-sample object and generate events */
    whizard_new_sample( &wh, "mupair", &sample );
    whizard_sample_open( &sample, &it_begin, &it_end );
    for (it=it_begin; it<=it_end; it++) {
      whizard_sample_next_event( &sample );
      whizard_sample_get_event_index( &sample, &idx );
      whizard_sample_get_weight( &sample, &weight );
      whizard_sample_get_sqme( &sample, &sqme );
      printf( "Event #%d\n", idx );
      printf( "  sqme    = %10.3e\n", sqme );
      printf( "  weight  = %10.3e\n", weight );
    }

    /* Finalize the event-sample object */
    whizard_sample_close( &sample );

    /* Finalize the WHIZARD object */
    whizard_final( &wh );
  }
```

Header

The necessary declarations are imported by the directive

```
    #include "whizard.h"
```

Master object

All functionality is accessed via a master API object which should be declared as a `void*` pointer:

```
    void* wh;
```

The object must be explicitly created:

```
    whizard_create( &wh );
```

There should be only one master object.

Pre-Initialization options

Before initializing the API object, it is possible to provide options. The available options mirror the command-line options of the stand-alone program, cf. Sec. 14.1.

```
whizard_option( &wh, key, value );
```

All keys and values are null-terminated C character strings. The available options are listed above in the `Fortran` interface documentation.

Initialization and finalization

After options have been set, the system is initialized via

```
whizard_init( &wh );
```

Once initialized, `WHIZARD` can execute commands as listed below. When this is complete, clean up by

```
whizard_final( &wh );
```

Variables and values

In the API, `WHIZARD` requires numeric data types according to the IEEE standard. Integers map to C `int`, and real values map to C `double`. Logical values map to C `int` interpreted as `bool`, and string values map to null-terminated C strings.

To set a `SINDARIN` variable of appropriate type:

```
whizard_set_int ( &wh, name, value );
whizard_set_double ( &wh, name, value );
whizard_set_bool ( &wh, name, value );
whizard_set_char ( &wh, name, value );
```

name is declared `const char*`. It must match the corresponding `SINDARIN` variable name, including any prefix character ($ or ?). *value* is declared `const double/int/char*`.

To retrieve the current value of a variable:

```
whizard_get_int ( &wh, name, &var );
whizard_get_double ( &wh, name, &var );
whizard_get_bool ( &wh, name, &var );
whizard_get_char ( &wh, name, var, len );
```

Here, *var* is a C variable of appropriate type. In the character case, *var* is a C character array declared as *var*[*len*]. The functions return zero if the `SINDARIN` variable has a known value.

Commands

Any `SINDARIN` command can be called via

```
whizard_command( &wh, command );
```

command is a null-terminated C string that contains commands as they would appear in a `SINDARIN` script.

This includes, in particular, the important commands `process`, `integrate`, and `simulate`. You may also set variables that way.

Retrieving cross-section results

This call returns the results (integration and error) from a preceding integration run for the process *process-name*:

```
whizard_get_integration_result( &wh, "process-name", &integral, &error )
```

integral and *error* are C variables of type `double`. The function returns zero if the integration run was successful, so integral and error are meaningful.

Event-sample object

A `simulate` command will produce an event sample. With the appropriate settings, the sample will be written to file in any chosen format, to be post-processed when it is complete.

However, a possible purpose of using the `WHIZARD` API is to process events one-by-one when they are generated. To this end, there is an event-sample handle, which can be declared in this way:

```
void* sample;
```

An instance *sample* of this type is created by this factory method:

```
whizard_new_sample( &wh, "process-name(s)", &sample);
```

The command accepts a comma-separated list of process names which should be included in the event sample.

To start event generation for this sample, call

```
whizard_sample_open( &sample, &it_begin, &it_end );
```

where the two output variables (`int`) *it_begin* and *it_end* provide the bounds for an event loop in the calling program. (In serial mode, the bounds are equal to 1 and `n_events`, respectively, but in an MPI parallel environment, they depend on the computing node.)

This command generates a new event, to be enclosed within an event loop:

```
whizard_sample_next_event( &sample );
```

The event will be available by format-specific access methods, see below.

This command closes and deletes an event sample after the event loop has completed:

```
whizard_sample_close( &sample );
```

Retrieving event data

After a call to `whizard_sample_next_event`, the sample object can be queried for specific event data.

```
whizard_sample_get_event_index( &sample, &value );
whizard_sample_get_process_index( &sample, &value );
whizard_sample_get_process_id( &sample, value, len );
whizard_sample_get_sqrts( &sample, &value );
whizard_sample_get_fac_scale( &sample, &value );
whizard_sample_get_alpha_s( &sample, &value );
whizard_sample_get_sqme( &sample, &value );
whizard_sample_get_weight( &sample, &value );
```

where the *value* is a variable of appropriate type (see above).

Event data are stored in a format-specific way. This may be a COMMON block, or a `HepMC` or `LCIO` event record. In the latter cases, cf. the `C++` API below for access information.

14.4.3   C++ main program

To link a `C++` main program with the `WHIZARD` library, the following steps must be performed:

1. Configure, build and install `WHIZARD` as normal.

2. Include code for accessing `WHIZARD` functionality in the user program. The code should initialize `WHIZARD`, execute the intended commands, and finalize. For an example, see below.

3. Compile the user program with the option that finds the WHIZARD `C/C++` interface header file. For instance, if `WHIZARD` has been installed in `whizard-path`, this should read

   ```
   -Iwhizard-path/include
   ```

4. Link the program with the necessary libraries (or compile-link in a single step). If `WHIZARD` has been installed in a system path, this should work automatically. If `WHIZARD` has been installed in a non-default `whizard-path`, these are the options:

   ```
   -Lwhizard-path/lib -lwhizard -lwhizard_prebuilt -lomega -ltirpc
   ```

   On some systems, you may have to replace `lib` by `lib64`.

   If `WHIZARD` has been compiled with a non-default `Fortran` compiler, you may have to explicitly link the appropriate `Fortran` run-time libraries.

   The `tirpc` library is used by the `StdHEP` subsystem for `xdr` functionality. This library should be present on the host system.

   If additional libraries such as `HepMC` are enabled in the `WHIZARD` configuration, it may be necessary to provide extra options for linking those.

5. Run the program. If necessary, provide the path to the installed shared libraries. For instance, if WHIZARD has been installed in `whizard-path`, this should read

```
export LD_LIBRARY_PATH="whizard-path/lib:$LD_LIBRARY_PATH"
```

On some systems, you may have to replace `lib` by `lib64`, as above.

The WHIZARD subsystem will work with input and output files in the current working directory, unless asked to do otherwise.

Below is an example program, adapted from WHIZARD's internal unit-test suite. The user program controls the WHIZARD workflow in the same way as a SINDARIN script would do. The commands are a mixture of SINDARIN command calls and functionality for passing information between the WHIZARD subsystem and the host program. In particular, the program can process generated events one-by-one.

```cpp
#include <cstdio>
#include <string>
#include "whizard.h"

int main( int argc, char* argv[] )
{
  // WHIZARD and event-sample objects
  Whizard* whizard;
  WhizardSample* sample;

  // Local variables
  double integral, error;
  double sqme, weight;
  int idx;
  int it, it_begin, it_end;

  // Initialize WHIZARD, setting some global option
  whizard = new Whizard();
  whizard->option( "model", "QED" );
  whizard->init();

  // Define a process, set some variables
  whizard->command( "process mupair = e1, E1 => e2, E2" );
  whizard->set_double( "sqrts", 10. );
  whizard->set_int( "seed", 0 );

  // Generate matrix-element code, integrate and retrieve result
  whizard->command( "integrate (mupair)" );

  // Print result
  whizard->get_integration_result( "mupair", &integral, &error );
  printf( "  cross section = %5.1f pb\n", integral / 1000. );
  printf( "  error         = %5.1f pb\n", error / 1000. );

  // Settings for event generation
  whizard->set_string( "$sample", "mupair_events" );
  whizard->set_int( "n_events", 2 );
```

```
  // Create an event-sample object and generate events
  sample = whizard->new_sample( "mupair" );
  sample->open( &it_begin, &it_end );
  for (it=it_begin; it<=it_end; it++) {
    sample->next_event();
    idx = sample->get_event_index();
    weight = sample->get_weight();
    sqme = sample->get_sqme();
    printf( "Event #%d\n", idx );
    printf( "  sqme    = %10.3e\n", sqme );
    printf( "  weight  = %10.3e\n", weight );
  }

  // Finalize the event-sample object
  sample->close();
  delete sample;

  // Finalize the WHIZARD object
  delete whizard;
}
```

Header

The necessary declarations are imported by the directive

```
#include "whizard.h"
```

Master object

All functionality is accessed via a master API object which should be declared as follows:

```
Whizard* whizard;
```

The constructor takes no arguments:

```
whizard = new Whizard();
```

There should be only one master object.

Pre-Initialization options

Before initializing the API object, it is possible to provide options. The available options mirror the command-line options of the stand-alone program, cf. Sec. 14.1.

```
whizard->option( key, value );
```

All keys and values are `C++` strings. The available options are listed above in the `Fortran` interface documentation.

Initialization and finalization

After options have been set, the system is initialized via

```
whizard->init();
```

Once initialized, `WHIZARD` can execute commands as listed below. When all is complete, delete the `WHIZARD` object. This will call the destructor that correctly finalizes the `WHIZARD` workflow.

Variables and values

In the API, `WHIZARD` requires numeric data types according to the IEEE standard. Integers map to C `int`, and real values map to C `double`. Logical values map to C `int` interpreted as `bool`, and string values map to `C++ string`.

To set a `SINDARIN` variable of appropriate type:

```
whizard->set_int ( name, value );
whizard->set_double ( name, value );
whizard->set_bool ( name, value );
whizard->set_string ( name, value );
```

name is a `C++` string value. It must match the corresponding `SINDARIN` variable name, including any prefix character ($ or ?). *value* is a `double/int/string`, respectively.

To retrieve the current value of a variable:

```
whizard->get_int ( name, &var );
whizard->get_double ( name, &var );
whizard->get_bool ( name, &var );
whizard->get_string ( name, &var );
```

Here, *var* is a C variable of appropriate type. The functions return zero if the `SINDARIN` variable has a known value.

Commands

Any `SINDARIN` command can be called via

```
whizard->command( command );
```

command is a `C++` string value that contains commands as they would appear in a `SINDARIN` script.

This includes, in particular, the important commands `process`, `integrate`, and `simulate`. You may also set variables that way.

Retrieving cross-section results

This call returns the results (integration and error) from a preceding integration run for the process *process-name*:

```
whizard->get_integration_result( "process-name", &integral, &error );
```

integral and *error* are variables of type `double`. The function returns zero if the integration run was successful, so integral and error are meaningful.

Event-sample object

A `simulate` command will produce an event sample. With the appropriate settings, the sample will be written to file in any chosen format, to be post-processed when it is complete.

However, a possible purpose of using the `WHIZARD` API is to process events one-by-one when they are generated. To this end, there is an event-sample handle, which can be declared in this way:

```
WhizardSample* sample;
```

An instance *sample* of this type is created by this factory method:

```
sample = whizard->new_sample( "process-name(s)" );
```

The command accepts a comma-separated list of process names which should be included in the event sample.

To start event generation for this sample, call

```
sample->open( &it_begin, &it_end );
```

where the two output variables (`int`) *it_begin* and *it_end* provide the bounds for an event loop in the calling program. (In serial mode, the bounds are equal to 1 and `n_events`, respectively, but in an MPI parallel environment, they depend on the computing node.)

This command generates a new event, to be enclosed within an event loop:

```
sample->next_event();
```

The event will be available by format-specific access methods, see below.

This command closes and deletes an event sample after the event loop has completed:

```
sample->close();
```

Retrieving event data

After a call to `sample->next_event`, the sample object can be queried for specific event data.

```
value = sample->get_event_index();
value = sample->get_process_index();
value = sample->get_process_id();
value = sample->get_sqrts();
value = sample->get_fac_scale();
value = sample->get_alpha_s();
value = sample->get_sqme();
value = sample->get_weight();
```

where the *value* is a variable of appropriate type (see above).

Event data are stored in a format-specific way. This may be a `HepMC` or `LCIO` C++ event record.

For interfacing with the `HepMC` event record, the appropriate declarations must be in place, e.g.,

```
#include "HepMC/GenEvent.h"
using namespace HepMC;
```

An event-record object must be declared,

```
GenEvent* evt;
```

and the `WHIZARD` event call must take the event as an argument

```
sample->next_event ( &evt );
```

This will create a new `evt` object. Then, the `HepMC` event record can be accessed via its own methods. After an event has been processed, the event record should be deleted

```
delete evt;
```

Analogously, for interfacing with the `LCIO` event record, the appropriate declarations must be in place, e.g.,

```
#include "lcio.h"
#include "IMPL/LCEventImpl.h"
using namespace lcio;
```

An event-record object must be declared,

```
LCEvent* evt;
```

and the `WHIZARD` event call must take the event as an argument

```
sample->next_event ( &evt );
```

This will create a new `evt` object. Then, the `LCIO` event record can be accessed via its own methods. After an event has been processed, the event record should be deleted

```
delete evt;
```

14.4.4   Python main program

To create a `Python` executable, `WHIZARD` provides a `Cython` interface that uses `C++` bindings to link a dynamic library which can then be loaded as a module via `Python`. Note that `WHIZARD`'s `Cython`/`Python` interface only works with `Pythonv3`. Also make sure that you do not mix different `Python` versions when linking external programs which also provide `Python` interfaces like `HepMC` or `LCIO`.

To link a `Python` main program with the `WHIZARD` library, the following steps must be performed:

1. Configure, build and install `WHIZARD` as normal.

2. Include code for accessing `WHIZARD` functionality in the user program. The code should initialize `WHIZARD`, execute the intended commands, and finalize. For an example, see below.

3. Run `Python` on the user program. Make sure that the operating system finds the `WHIZARD` `Python` and library path. If `WHIZARD` has been installed in a non-default `whizard-path`, these are the options:

   ```
   export PYTHONPATH=whizard-path/lib/python/site-packages/:$PYTHONPATH
   ```

   If necessary, provide the path to the installed shared libraries. For instance, if `WHIZARD` has been installed in `whizard-path`, this should read

   ```
   export LD_LIBRARY_PATH="whizard-path/lib:$LD_LIBRARY_PATH"
   ```

   On some systems, you may have to replace `lib` by `lib64`, as above.

   The `WHIZARD` subsystem will work with input and output files in the current working directory, unless asked to do otherwise.

4. The `tirpc` library is used by the `StdHEP` subsystem for `xdr` functionality. This library should be present on the host system.

5. Run the program.

Below is an example program, similar to `WHIZARD`'s internal unit-test suite for different external programming languages. The user program controls the `WHIZARD` workflow in the same way as a `SINDARIN` script would do. The commands are a mixture of `SINDARIN` command calls and functionality for passing information between the `WHIZARD` subsystem and the host program. In particular, the program can process generated events one-by-one.

```
import whizard

wz = whizard.Whizard()

wz.option("logfile", "whizard_1_py.log")
wz.option("job_id", "whizard_1_py_ID")
wz.option("library", "whizard_1_py_1_lib")
wz.option("model", "QED")
```

```
wz.init()

wz.set_double("sqrts", 100)
wz.set_int("n_events", 3)
wz.set_bool("?unweighted", True)
wz.set_string("$sample", "foobar")

wz.set_int("seed", 0)

wz.command("process whizard_1_py_1_p = e1, E1 => e2, E2")
wz.command("iterations = 1:100")

integral, error = wz.get_integration_result("whizard_1_py_1_p")
print(integral, error)

wz.command("integrate (whizard_1_py_1_p)")
sqrts = wz.get_double("sqrts")
print(f"sqrts  = {sqrts:5.1f} GeV")
print(f"sigma   = integral:5.1f} pb")
print(f"error    {error:5.1f} pb")

sample = wz.new_sample("whizard_1_py_p1, whizard_1_py_p2, whizard_1_py_p3")
it_begin, it_end = sample.open()
for it in range(it_begin, it_end + 1):
    sample.next_event()
    idx = sample.get_event_index()
    i_proc  = sample.get_process_index()
    proc_id = sample.get_process_id()
    f_scale = sample.get_fac_scale()
    alpha_s = sample.get_alpha_s()
    weight  = sample.get_weight()
    sqme    = sample.get_sqme()
    print(f"Event #{idx}")
    print(f"  process #{i_proc}")
    print(f"  proc_id = {proc_id}")
    print(f"  f_scale = {f_scale:10.3e}")
    print(f"  alpha_s = {f_scale:10.3e}")
    print(f"  sqme    = {f_scale:10.3e}")
    print(f"  weight  = {f_scale:10.3e}")

sample.close()

del(wz)
```

Python module import

There are no necessary headers here as all of this information has been automatically taken care by the Cython interface layer. The WHIZARD module needs to be imported by Python

```
import whizard
```

Master object

All functionality is accessed via a master API object which should be declared as follows:

```
wz = whizard.Whizard()
```

The constructor takes no arguments. There should be only one master object.

Pre-Initialization options

Before initializing the API object, it is possible to provide options. The available options mirror the command-line options of the stand-alone program, cf. Sec. 14.1.

```
wz.option( key, value );
```

All keys and values are `Python` strings. The available options are listed above in the `Fortran` interface documentation.

Initialization and finalization

After options have been set, the system is initialized via

```
wz.init()
```

Once initialized, `WHIZARD` can execute commands as listed below. When all is complete, delete the `WHIZARD` object. This will call the destructor that correctly finalizes the `WHIZARD` workflow.

Variables and values

In the API, `WHIZARD` requires numeric data types according to the IEEE standard. Integers map to `Python int`, and real values map to `Python double`. Logical values map to `True` and `False`, and string values map to `Python` strings.

To set a `SINDARIN` variable of appropriate type:

```
wz.set_int ( name, value );
wz.set_double ( name, value );
wz.set_bool ( name, value );
wz.set_string ( name, value );
```

name is a `Python` string value. It must match the corresponding `SINDARIN` variable name, including any prefix character ($ or ?). *value* is a `double/int/string`, respectively.

To retrieve the current value of a variable:

```
wz.get_int ( name, var );
wz.get_double ( name, var );
wz.get_bool ( name, var );
wz.get_string ( name, var );
```

Here, *var* is a `Python` variable of appropriate type. The functions return zero if the `SINDARIN` variable has a known value.

Commands

Any `SINDARIN` command can be called via

    wz.command(*command*);

command is a `Python` string value that contains commands as they would appear in a `SINDARIN` script.

   This includes, in particular, the important commands `process`, `integrate`, and `simulate`. You may also set variables that way.

Retrieving cross-section results

This call returns the results (integration and error) from a preceding integration run for the process *process-name*:

    wz.get_integration_result("*process-name*", *integral*, *error*);

integral and *error* are variables of type `double`. The function returns zero if the integration run was successful, so integral and error are meaningful.

Event-sample object

A `simulate` command will produce an event sample. With the appropriate settings, the sample will be written to file in any chosen format, to be post-processed when it is complete.

   However, a possible purpose of using the `WHIZARD` API is to process events one-by-one when they are generated. To this end, there is an event-sample handle, which can be declared in this way:

    WhizardSample* sample;

An instance *sample* of this type is created by this factory method:

    sample = wz.new_sample("*process-name(s)*");

The command accepts a comma-separated list of process names which should be included in the event sample.

   To start event generation for this sample, call

    it_begin, *it_end* = wz.sample_open()

where the two output variables (`int`) *it_begin* and *it_end* provide the bounds for an event loop in the calling program. (In serial mode, the bounds are equal to 1 and `n_events`, respectively, but in an MPI parallel environment, they depend on the computing node.)

   This command generates a new event, to be enclosed within an event loop:

    sample.next_event();

The event will be available by format-specific access methods, see below.

   This command closes and deletes an event sample after the event loop has completed:

    sample.close();

Retrieving event data

After a call to `sample.next_event`, the sample object can be queried for specific event data.

```
value = sample.get_event_index();
value = sample.get_process_index();
value = sample.get_process_id();
value = sample.get_sqrts();
value = sample.get_fac_scale();
value = sample.get_alpha_s();
value = sample.get_sqme();
value = sample.get_weight();
```

where the *value* is a variable of appropriate type (see above).

Event data are stored in a format-specific way. This may be a `HepMC` or `LCIO` C++ event record, or some formats supported by `WHIZARD` intrinsically like LHEF etc.

Chapter 15

Examples

In this chapter we discuss the running and steering of WHIZARD with the help of several examples. These examples can be found in the **share/examples** directory of your installation. All of these examples are also shown on the WHIZARD Wiki page: https://whizard.hepforge.org/trac/wiki.

15.1 Z lineshape at LEP I

By this example, we demonstrate how a scan over collision energies works, using as example the measurement of the Z lineshape at LEP I in 1989. The SINDARIN script for this example, Z-lineshape.sin can be found in the **share/examples** folder of the WHIZARD installation.

We first use the Standard model as physics model:

```
model = SM
```

Aliases for electron, muon and their antiparticles as leptons and those including the photon as particles in general are introduced:

```
alias lep = e1:E1:e2:E2
alias prt = lep:A
```

Next, the two processes are defined, $e^+e^- \to \mu^+\mu^-$, and the same with an explicit QED photon: $e^+e^- \to \mu^+\mu^-\gamma$,

```
process bornproc = e1, E1 => e2, E2
process rc = e1, E1 => e2, E2, A
compile
```

and the processes are compiled. Now, we define some very loose cuts to avoid singular regions in phase space, name an infrared cutoff of 100 MeV for all particles, a cut on the angular separation from the beam axis and a di-particle invariant mass cut which regularizes collinear singularities:

```
cuts = all E >= 100 MeV [prt]
    and all abs (cos(Theta)) <= 0.99 [prt]
    and all M2 >= (1 GeV)^2 [prt, prt]
```

For the graphical analysis, we give a description and labels for the x- and y-axis in LATEX syntax:

```
$description = "A WHIZARD Example"
$x_label = "$\sqrt{s}$/GeV"
$y_label = "$\sigma(s)$/pb"
```

We define two plots for the lineshape of the $e^+e^- \to \mu^+\mu^-$ process between 88 and 95 GeV,

```
$title = "The Z Lineshape in $e^+e^-\to\mu^+\mu^-$"
plot lineshape_born { x_min = 88 GeV  x_max = 95 GeV }
```

and the same for the radiative process with an additional photon:

```
$title = "The Z Lineshape in $e^+e^-\to\mu^+\mu^-\gamma$"
plot lineshape_rc { x_min = 88 GeV  x_max = 95 GeV }
```

The next part of the SINDARIN file actually performs the scan:

```
scan sqrts = ((88.0 GeV => 90.0 GeV /+ 0.5 GeV),
              (90.1 GeV => 91.9 GeV /+ 0.1 GeV),
              (92.0 GeV => 95.0 GeV /+ 0.5 GeV)) {
   beams = e1, E1
   integrate (bornproc) { iterations = 2:1000:"gw", 1:2000 }
   record lineshape_born (sqrts, integral (bornproc) / 1000)
   integrate (rc)   { iterations = 5:3000:"gw", 2:5000 }
   record lineshape_rc (sqrts, integral (rc) / 1000)
}
```

So from 88 to 90 GeV, we go in 0.5 GeV steps, then from 90 to 92 GeV in tenth of GeV, and then up to 95 GeV again in half a GeV steps. The partonic beam definition is redundant. Then, the born process is integrated, using a certain specification of calls with adaptation of grids and weights, as well as a final pass. The lineshape of the Born process is defined as a `record` statement, generating tuples of $\sqrt{s}$ and the Born cross section (converted from femtobarn to picobarn). The same happens for the radiative $2 \to 3$ process with a bit more iterations because of the complexity, and the definition of the corresponding lineshape record.

If you run the SINDARIN script, you will find an output like:

```
        | Process library 'default_lib': loading
        | Process library 'default_lib': ... success.
        $description = "A WHIZARD Example"
        $x_label = "$\sqrt{s}$/GeV"
        $y_label = "$\sigma(s)$/pb"
        $title = "The Z Lineshape in $e^+e^-\to\mu^+\mu^-$"
        x_min =  8.800000000000E+01
        x_max =  9.500000000000E+01
        $title = "The Z Lineshape in $e^+e^-\to\mu^+\mu^-\gamma$"
        x_min =  8.800000000000E+01
        x_max =  9.500000000000E+01
        sqrts =  8.800000000000E+01
        | RNG: Initializing TAO random-number generator
        | RNG: Setting seed for random-number generator to 10713
        | Initializing integration for process bornproc:
        | ----------------------------------------------------------------------
        | Process [scattering]: 'bornproc'
        |   Library name  = 'default_lib'
        |   Process index = 1
        |   Process components:
        |     1: 'bornproc_i1':   e-, e+ => mu-, mu+ [omega]
        | ----------------------------------------------------------------------
```

1   The Z Lineshape in $e^+e^- \rightarrow \mu^+\mu^-$

A WHIZARD Example

1   The Z Lineshape in $e^+e^- \rightarrow \mu^+\mu^-\gamma$

A WHIZARD Example
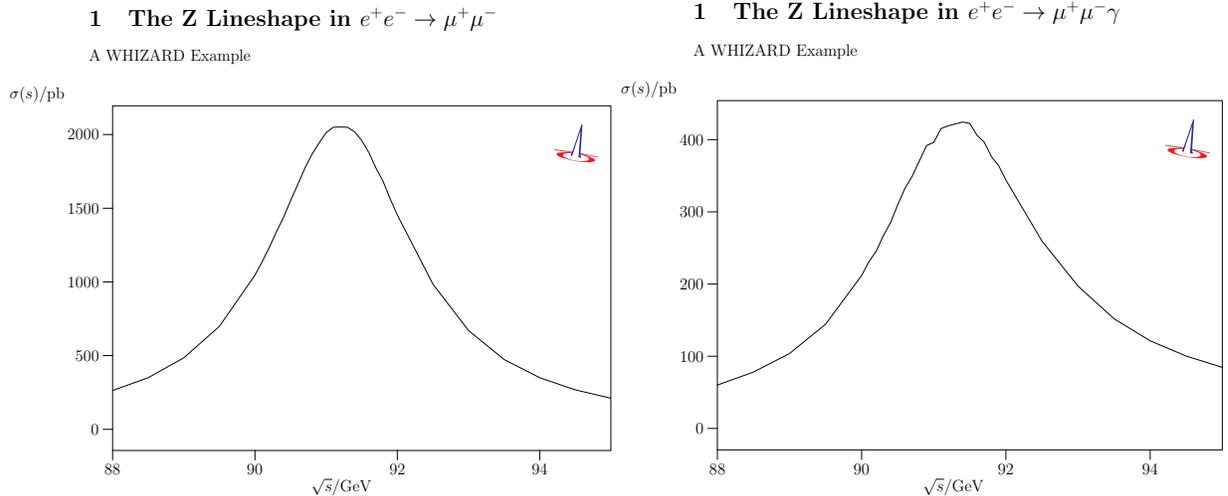
Figure 15.1: *Z lineshape in the dimuon final state (left), and with an additional photon (right)*

```
| Beam structure: e-, e+
| Beam data (collision):
|   e-  (mass = 5.1099700E-04 GeV)
|   e+  (mass = 5.1099700E-04 GeV)
|   sqrts = 8.800000000000E+01 GeV
| Phase space: generating configuration ...
| Phase space: ... success.
| Phase space: writing configuration file 'bornproc_i1.phs'
| Phase space: 1 channels, 2 dimensions
| Phase space: found 1 channel, collected in 1 grove.
| Phase space: Using 1 equivalence between channels.
| Phase space: wood
| Applying user-defined cuts.
| OpenMP: Using 8 threads
| Starting integration for process 'bornproc'
| Integrate: iterations = 2:1000:"gw", 1:2000
| Integrator: 1 chains, 1 channels, 2 dimensions
| Integrator: Using VAMP channel equivalences
| Integrator: 1000 initial calls, 20 bins, stratified = T
| Integrator: VAMP
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
    1       800  2.5881432E+05  1.85E+03    0.72   0.20*  48.97
    2       800  2.6368495E+05  9.25E+02    0.35   0.10*  28.32
|-----------------------------------------------------------------------------|
    2      1600  2.6271122E+05  8.28E+02    0.32   0.13   28.32    5.54    2
|-----------------------------------------------------------------------------|
    3      1988  2.6313791E+05  5.38E+02    0.20   0.09*  35.09
|-----------------------------------------------------------------------------|
    3      1988  2.6313791E+05  5.38E+02    0.20   0.09   35.09
|=============================================================================|
| Time estimate for generating 10000 events: 0d:00h:00m:05s
[.......]
```

and then the integrations for the other energy points of the scan will follow, and finally the same is done for the radiative process as well. At the end of the SINDARIN script we compile the graphical WHIZARD analysis and direct the data for the plots into the file Z-lineshape.dat:

```
compile_analysis { $out_file = "Z-lineshape.dat" }
```

In this case there is no event generation, but simply the cross section values for the scan are dumped into a data file:

```
        $out_file = "Z-lineshape.dat"
        | Opening file 'Z-lineshape.dat' for output
        | Writing analysis data to file 'Z-lineshape.dat'
        | Closing file 'Z-lineshape.dat' for output
        | Compiling analysis results display in 'Z-lineshape.tex'
```

Fig. 15.1 shows the graphical WHIZARD output of the Z lineshape in the dimuon final state from the scan on the left, and the same for the radiative process with an additional photon on the right.

15.2 W pairs at LEP II

This example which can be found as file LEP_cc10.sin in the share/examples directory, shows W pair production in the semileptonic mode at LEP II with its final energy of 209 GeV. Because there are ten contributing Feynman diagrams, the process has been dubbed CC10: charged current process with 10 diagrams. We work within the Standard Model:

```
        model = SM
```

Then the process is defined, where no flavor summation is done for the jets here:

```
        process cc10 = e1, E1 => e2, N2, u, D
```

A compilation statement is optional, and then we set the muon mass to zero:

```
        mmu = 0
```

The final LEP center-of-momentum energy of 209 GeV is set:

```
        sqrts = 209 GeV
```

Then, we integrate the process:

```
        integrate (cc10) { iterations = 12:20000 }
```

Running the SINDARIN file up to here, results in the output:

```
        | Process library 'default_lib': loading
        | Process library 'default_lib': ... success.
        SM.mmu =  0.000000000000E+00
        sqrts =  2.090000000000E+02
        | RNG: Initializing TAO random-number generator
        | RNG: Setting seed for random-number generator to 31255
        | Initializing integration for process cc10:
        | -----------------------------------------------------------------------
        | Process [scattering]: 'cc10'
        |   Library name  = 'default_lib'
        |   Process index = 1
        |   Process components:
        |     1: 'cc10_i1':   e-, e+ => mu-, numubar, u, dbar [omega]
        | -----------------------------------------------------------------------
        | Beam structure: [any particles]
        | Beam data (collision):
        |   e-  (mass = 5.1099700E-04 GeV)
        |   e+  (mass = 5.1099700E-04 GeV)
```

```
|    sqrts = 2.090000000000E+02 GeV
| Phase space: generating configuration ...
| Phase space: ... success.
| Phase space: writing configuration file 'cc10_i1.phs'
| Phase space: 25 channels, 8 dimensions
| Phase space: found 25 channels, collected in 7 groves.
| Phase space: Using 25 equivalences between channels.
| Phase space: wood
Warning: No cuts have been defined.
| OpenMP: Using 8 threads
| Starting integration for process 'cc10'
| Integrate: iterations = 12:20000
| Integrator: 7 chains, 25 channels, 8 dimensions
| Integrator: Using VAMP channel equivalences
| Integrator: 20000 initial calls, 20 bins, stratified = T
| Integrator: VAMP
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]  Chi2 N[It] |
|=============================================================================|
     1    19975  6.4714908E+02  2.17E+01    3.36    4.75*   2.33
     2    19975  7.3251876E+02  2.45E+01    3.34    4.72*   2.17
     3    19975  6.7746497E+02  2.39E+01    3.52    4.98    1.77
     4    19975  7.2075198E+02  2.41E+01    3.34    4.72*   1.76
     5    19975  6.5976152E+02  2.26E+01    3.43    4.84    1.46
     6    19975  6.6633310E+02  2.26E+01    3.39    4.79*   1.43
     7    19975  6.7539385E+02  2.29E+01    3.40    4.80    1.43
     8    19975  6.6754027E+02  2.11E+01    3.15    4.46*   1.41
     9    19975  7.3975817E+02  2.52E+01    3.40    4.81    1.53
    10    19975  7.2284275E+02  2.39E+01    3.31    4.68*   1.47
    11    19975  6.5476917E+02  2.18E+01    3.33    4.71    1.33
    12    19975  7.2963866E+02  2.54E+01    3.48    4.92    1.46
|-----------------------------------------------------------------------------|
    12   239700  6.8779583E+02  6.69E+00    0.97    4.76    1.46    2.18  12
|=============================================================================|
| Time estimate for generating 10000 events: 0d:00h:01m:16s
| Creating integration history display cc10-history.ps and cc10-history.pdf
```

The next step is event generation. In order to get smooth distributions, we set the integrated luminosity to 10 fb^{-1}. (Note that LEP II in its final year 2000 had an integrated luminosity of roughly 0.2 fb^{-1}.)

```
luminosity = 10
```

With the simulated events corresponding to those 10 inverse femtobarn we want to perform a `WHIZARD` analysis: we are going to plot the dijet invariant mass, as well as the energy of the outgoing muon. For the plot of the analysis, we define a description and label the y axis:

```
$description =
  "A WHIZARD Example.
   Charged current CC10 process from LEP 2."
$y_label = "$N_{\textrm{events}}$"
```

We also use LATEX-syntax for the title of the first plot and the x-label, and then define the histogram of the dijet invariant mass in the range around the W mass from 70 to 90 GeV in steps of half a GeV:

```
$title = "Di-jet invariant mass $M_{jj}$ in $e^+e^- \to \mu^- \bar\nu_\mu u \bar d$"
$x_label = "$M_{jj}$/GeV"
histogram m_jets (70 GeV, 90 GeV, 0.5 GeV)
```

And we do the same for the second histogram of the muon energy:

1   Di-jet invariant mass M_{jj} in $e^+e^- \to \mu^-\bar{\nu}_\mu u\bar{d}$

A WHIZARD Example. Charged current CC10 process from LEP 2.
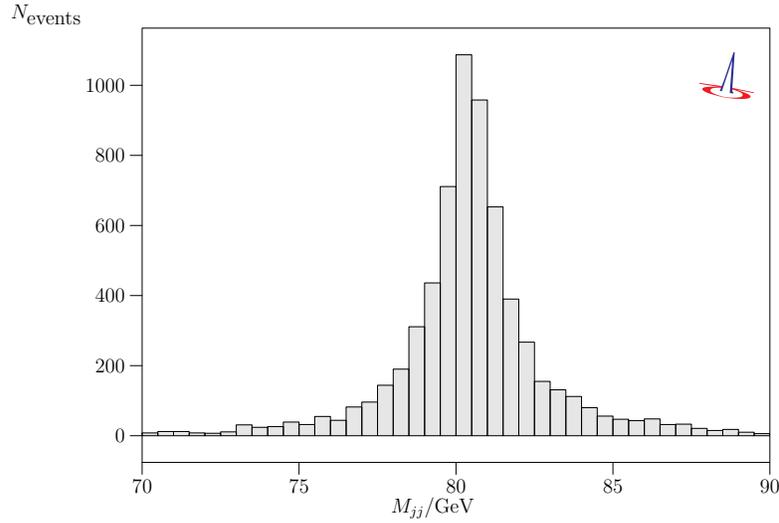
Data within bounds:
$\langle$Observable$\rangle = 80.458 \pm 0.030$ $[n_{\text{entries}} = 6441]$

All data:
$\langle$Observable$\rangle = 80.735 \pm 0.065$ $[n_{\text{entries}} = 6781]$

2   Muon energy E_μ in $e^+e^- \to \mu^-\bar{\nu}_\mu u\bar{d}$

A WHIZARD Example. Charged current CC10 process from LEP 2.
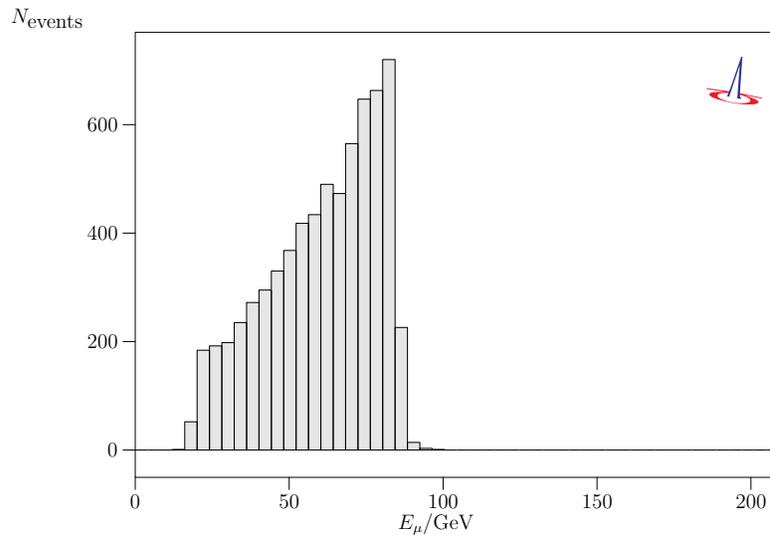
Data within bounds:
$\langle$Observable$\rangle = 60.57 \pm 0.22$ $[n_{\text{entries}} = 6781]$

All data:
$\langle$Observable$\rangle = 60.57 \pm 0.22$ $[n_{\text{entries}} = 6781]$

Figure 15.2: *Histogram of the dijet invariant mass from the CC10 W pair production at LEP II, peaking around the W mass (upper plot), and of the muon energy (lower plot).*

```
$title = "Muon energy $E_\mu$ in $e^+e^- \to \mu^- \bar\nu_\mu u \bar d$"
$x_label = "$E_\mu$/GeV"
histogram e_muon (0 GeV, 209 GeV, 4)
```

Now, we define the `analysis` consisting of two `record` statements initializing the two observables that are plotted as histograms:

```
analysis = record m_jets (eval M [u,D]);
           record e_muon (eval E [e2])
```

At the very end, we perform the event generation

```
simulate (cc10)
```

and finally the writing and compilation of the analysis in a named data file:

```
compile_analysis { $out_file = "cc10.dat" }
```

This event generation part screen output looks like this:

```
luminosity =  1.000000000000E+01
$description = "A WHIZARD Example.
   Charged current CC10 process from LEP 2."
$y_label = "$N_{\textrm{events}}$"
$title = "Di-jet invariant mass $M_{jj}$ in $e^+e^- \to \mu^- \bar\nu_\mu u \bar d$"
$x_label = "$M_{jj}$/GeV"
$title = "Muon energy $E_\mu$ in $e^+e^- \to \mu^- \bar\nu_\mu u \bar d$"
$x_label = "$E_\mu$/GeV"
| Starting simulation for process 'cc10'
| Simulate: using integration grids from file 'cc10_m1.vg'
| RNG: Initializing TAO random-number generator
| RNG: Setting seed for random-number generator to 9910
| OpenMP: Using 8 threads
| Simulation: using n_events as computed from luminosity value
| Events: writing to raw file 'cc10.evx'
| Events: generating 6830 unweighted, unpolarized events ...
| Events: event normalization mode '1'
|         ... event sample complete.
Warning: Encountered events with excess weight: 39 events (  0.571 %)
| Maximum excess weight = 1.027E+00
| Average excess weight = 6.764E-04
| Events: closing raw file 'cc10.evx'
$out_file = "cc10.dat"
| Opening file 'cc10.dat' for output
| Writing analysis data to file 'cc10.dat'
| Closing file 'cc10.dat' for output
| Compiling analysis results display in 'cc10.tex'
```

Then comes the LATEX output of the compilation of the graphical analysis. Fig. 15.2 shows the two histograms as the are produced as result of the WHIZARD internal graphical analysis.

15.3   Higgs search at LEP II

This example can be found under the name LEP_higgs.sin in the share/doc folder of WHIZARD. It displays different search channels for a very light would-be SM Higgs boson of mass 115 GeV at the LEP II machine at its highest energy it finally achieved, 209 GeV. First, we use the Standard Model:

```
model = SM
```

Then, we define aliases for neutrinos, antineutrinos, light quarks and light anti-quarks:

```
alias n = n1:n2:n3
alias N = N1:N2:N3
alias q = u:d:s:c
alias Q = U:D:S:C
```

Now, we define the signal process, which is Higgsstrahlung,

```
process zh = e1, E1 => Z, h
```

the missing-energy channel,

```
process nnbb = e1, E1 => n, N, b, B
```

and finally the 4-jet as well as dilepton-dijet channels:

```
process qqbb = e1, E1 => q, Q, b, B
process bbbb = e1, E1 => b, B, b, B
process eebb = e1, E1 => e1, E1, b, B
process qqtt = e1, E1 => q, Q, e3, E3
process bbtt = e1, E1 => b, B, e3, E3

compile
```

and we compile the code. We set the center-of-momentum energy to the highest energy LEP II achieved,

```
sqrts = 209 GeV
```

For the Higgs boson, we take the values of a would-be SM Higgs boson with mass of 115 GeV, which would have had a width of a bit more than 3 MeV:

```
mH = 115 GeV
wH = 3.228 MeV
```

We take a running b quark mass to take into account NLO corrections to the $Hb\bar{b}$ vertex, while all other fermions are massless:

```
mb = 2.9 GeV
me = 0
ms = 0
mc = 0
```

```
| Process library 'default_lib': loading
| Process library 'default_lib': ... success.
sqrts =   2.090000000000E+02
SM.mH =   1.150000000000E+02
SM.wH =   3.228000000000E-03
SM.mb =   2.900000000000E+00
SM.me =   0.000000000000E+00
SM.ms =   0.000000000000E+00
SM.mc =   0.000000000000E+00
```

To avoid soft-collinear singular phase-space regions, we apply an invariant mass cut on light quark pairs:

```
cuts = all M >= 10 GeV [q,Q]
```

Now, we integrate the signal process as well as the combined signal and background processes:

```
integrate (zh) { iterations = 5:5000}

integrate(nnbb,qqbb,bbbb,eebb,qqtt,bbtt) { iterations = 12:20000 }
```

```
        | RNG: Initializing TAO random-number generator
        | RNG: Setting seed for random-number generator to 21791
        | Initializing integration for process zh:
        | -----------------------------------------------------------------------
        | Process [scattering]: 'zh'
        |   Library name  = 'default_lib'
        |   Process index = 1
        |   Process components:
        |     1: 'zh_i1':   e-, e+ => Z, H [omega]
        | -----------------------------------------------------------------------
        | Beam structure: [any particles]
        | Beam data (collision):
        |   e-  (mass = 0.0000000E+00 GeV)
        |   e+  (mass = 0.0000000E+00 GeV)
        |   sqrts = 2.090000000000E+02 GeV
        | Phase space: generating configuration ...
        | Phase space: ... success.
        | Phase space: writing configuration file 'zh_i1.phs'
        | Phase space: 1 channels, 2 dimensions
        | Phase space: found 1 channel, collected in 1 grove.
        | Phase space: Using 1 equivalence between channels.
        | Phase space: wood
        | Applying user-defined cuts.
        | OpenMP: Using 8 threads
        | Starting integration for process 'zh'
        | Integrate: iterations = 5:5000
        | Integrator: 1 chains, 1 channels, 2 dimensions
        | Integrator: Using VAMP channel equivalences
        | Integrator: 5000 initial calls, 20 bins, stratified = T
        | Integrator: VAMP
        |=============================================================================|
        | It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
        |=============================================================================|
            1      4608  1.6114109E+02  5.52E-04    0.00    0.00*  99.43
            2      4608  1.6114220E+02  5.59E-04    0.00    0.00   99.43
            3      4608  1.6114103E+02  5.77E-04    0.00    0.00   99.43
            4      4608  1.6114111E+02  5.74E-04    0.00    0.00*  99.43
            5      4608  1.6114103E+02  5.66E-04    0.00    0.00*  99.43
        |-----------------------------------------------------------------------------|
            5     23040  1.6114130E+02  2.53E-04    0.00    0.00   99.43   0.82   5
        |=============================================================================|
        [.....]
```

Because the other integrations look rather similar, we refrain from displaying them here, too. As a next step, we define titles, descriptions and axis labels for the histograms we want to generate. There are two of them, one os the invisible mass distribution, the other is the di-*b*-jet invariant mass. Both histograms are taking values between 70 and 130 GeV with bin widths of half a GeV:

```
$description =
  "A WHIZARD Example. Light Higgs search at LEP. A 115 GeV pseudo-Higgs
    has been added. Luminosity enlarged by two orders of magnitude."
$y_label = "$N_{\textrm{events}}$"
```

### 1 Invisible mass distribution in $e^+e^- \to \nu\bar{\nu}b\bar{b}$

A WHIZARD Example. Light Higgs search at LEP. A 115 GeV pseudo-Higgs has been added. Luminosity enlarged by two orders of magnitude.
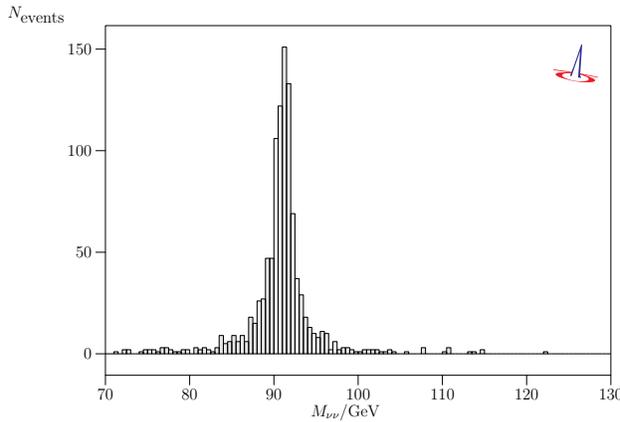
Data within bounds:
$\langle \text{Observable} \rangle = 90.87 \pm 0.14$ $[n_{\text{entries}} = 1034]$

All data:
$\langle \text{Observable} \rangle = 89.59 \pm 0.28$ $[n_{\text{entries}} = 1070]$

2 bb invariant mass distribution in $e^+e^- \to \nu\bar{\nu}b\bar{b}$

A WHIZARD Example. Light Higgs search at LEP. A 115 GeV pseudo-Higgs has been added. Luminosity enlarged by two orders of magnitude.
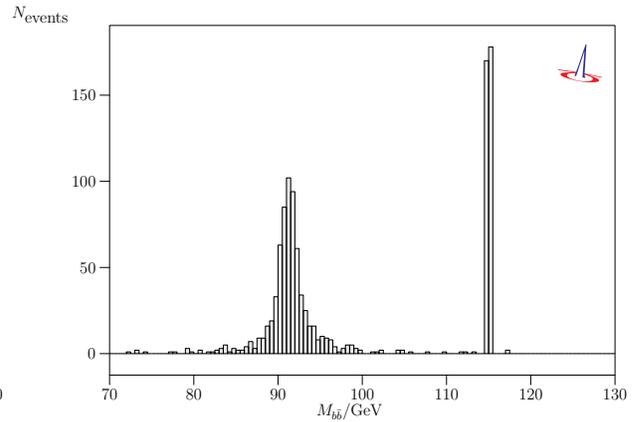
Data within bounds:
$\langle \text{Observable} \rangle = 99.31 \pm 0.36$ $[n_{\text{entries}} = 1050]$

All data:
$\langle \text{Observable} \rangle = 97.86 \pm 0.48$ $[n_{\text{entries}} = 1070]$

3 Dijet invariant mass distribution in $e^+e^- \to q\bar{q}b\bar{b}$

A WHIZARD Example. Light Higgs search at LEP. A 115 GeV pseudo-Higgs has been added. Luminosity enlarged by two orders of magnitude.
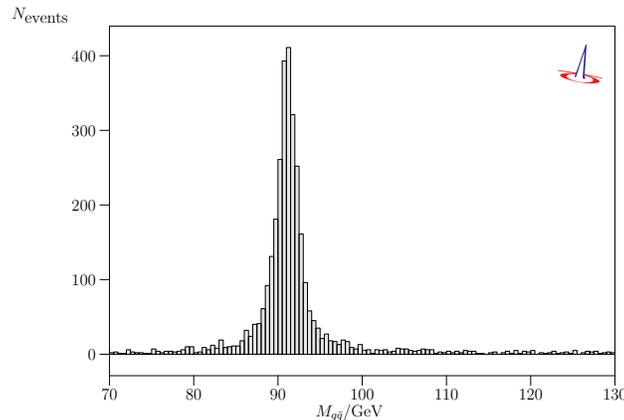
Data within bounds:
$\langle \text{Observable} \rangle = 91.97 \pm 0.12$ $[n_{\text{entries}} = 3186]$

All data:
$\langle \text{Observable} \rangle = 92.99 \pm 0.62$ $[n_{\text{entries}} = 4607]$

Figure 15.3: *Upper line: final state $bb + E_{miss}$, histogram of the invisible mass distribution (left), and of the di-b distribution (right). Lower plot: light dijet distribution in the bbjj final state.*

```
$title = "Invisible mass distribution in $e^+e^- \to \nu\bar\nu b \bar b$"
$x_label = "$M_{\nu\nu}$/GeV"
histogram m_invisible (70 GeV, 130 GeV, 0.5 GeV)

$title = "$bb$ invariant mass distribution in $e^+e^- \to \nu\bar\nu b \bar b$"
$x_label = "$M_{b\bar b}$/GeV"
histogram m_bb (70 GeV, 130 GeV, 0.5 GeV)
```

The analysis is initialized by defining the two records for the invisible mass and the invariant mass of the two b jets:

```
analysis = record m_invisible (eval M [n,N]);
           record m_bb (eval M [b,B])
```

In order to have enough statistics, we enlarge the LEP integrated luminosity at 209 GeV by more than two orders of magnitude:

```
luminosity = 10
```

We start event generation by simulating the process with two b jets and two neutrinos in the final state:

```
simulate (nnbb)
```

As a third histogram, we define the dijet invariant mass of two light jets:

```
$title = "Dijet invariant mass distribution in $e^+e^- \to q \bar q b \bar b$"
$x_label = "$M_{q\bar q}$/GeV"
histogram m_jj (70 GeV, 130 GeV, 0.5 GeV)
```

Then we simulate the 4-jet process defining the light-dijet distribution as a local record:

```
simulate (qqbb) { analysis = record m_jj (eval M / 1 GeV [combine [q,Q]]) }
```

Finally, we compile the analysis,

```
compile_analysis { $out_file = "lep_higgs.dat" }
```

```
        | Starting simulation for process 'nnbb'
        | Simulate: using integration grids from file 'nnbb_m1.vg'
        | RNG: Initializing TAO random-number generator
        | RNG: Setting seed for random-number generator to 21798
        | OpenMP: Using 8 threads
        | Simulation: using n_events as computed from luminosity value
        | Events: writing to raw file 'nnbb.evx'
        | Events: generating 1070 unweighted, unpolarized events ...
        | Events: event normalization mode '1'
        |        ... event sample complete.
        Warning: Encountered events with excess weight: 207 events ( 19.346 %)
        | Maximum excess weight = 1.534E+00
        | Average excess weight = 4.909E-02
        | Events: closing raw file 'nnbb.evx'
        $title = "Dijet invariant mass distribution in $e^+e^- \to q \bar q b \bar b$"
        $x_label = "$M_{q\bar q}$/GeV"
        | Starting simulation for process 'qqbb'
        | Simulate: using integration grids from file 'qqbb_m1.vg'
        | RNG: Initializing TAO random-number generator
        | RNG: Setting seed for random-number generator to 21799
```

```
| OpenMP: Using 8 threads
| Simulation: using n_events as computed from luminosity value
| Events: writing to raw file 'qqbb.evx'
| Events: generating 4607 unweighted, unpolarized events ...
| Events: event normalization mode '1'
|         ... event sample complete.
Warning: Encountered events with excess weight: 112 events (  2.431 %)
| Maximum excess weight = 8.875E-01
| Average excess weight = 4.030E-03
| Events: closing raw file 'qqbb.evx'
$out_file = "lep_higgs.dat"
| Opening file 'lep_higgs.dat' for output
| Writing analysis data to file 'lep_higgs.dat'
| Closing file 'lep_higgs.dat' for output
| Compiling analysis results display in 'lep_higgs.tex'
```

The graphical analysis of the events generated by WHIZARD are shown in Fig. 15.3. In the upper left, the invisible mass distribution in the $b\bar{b} + E_{miss}$ state is shown, peaking around the Z mass. The upper right shows the $M(b\bar{b})$ distribution in the same final state, while the lower plot has the invariant mass distribution of the two non-b-tagged (light) jets in the $bbjj$ final state. The latter shows only the Z peak, while the former exhibits the narrow would-be 115 GeV Higgs state.

15.4   Deep Inelastic Scattering at HERA

15.5 W endpoint at LHC

15.6   SUSY Cascades at LHC

15.7   Polarized WW at ILC

Chapter 16

Technical details – Advanced Spells

16.1   Efficiency and tuning

Since massless fermions and vector bosons (or almost massless states in a certain approximation) lead to restrictive selection rules for allowed helicity combinations in the initial and final state. To make use of this fact for the efficiency of the WHIZARD program, we are applying some sort of heuristics: WHIZARD dices events into all combinatorially possible helicity configuration during a warm-up phase. The user can specify a helicity threshold which sets the number of zeros WHIZARD should have got back from a specific helicity combination in order to ignore that combination from now on. By that mechanism, typically half up to more than three quarters of all helicity combinations are discarded (and hence the corresponding number of matrix element calls). This reduces calculation time up to more than one order of magnitude. WHIZARD shows at the end of the integration those helicity combinations which finally contributed to the process matrix element.

Note that this list – due to the numerical heuristics – might very well depend on the number of calls for the matrix elements per iteration, and also on the corresponding random number seed.

Chapter 17

New External Physics Models

It is never possible to include all incarnations of physics models that can be described by the maybe weirdest form of a quantum field theory in a tailor-made implementation within a program like `WHIZARD`. Users clearly want to be able to use their own special type of model; in order to do so there are external tools to translate models described by their field content and Lagrangian densities into Feynman rules and make them available in an event generator like `WHIZARD`. In this chapter, we describe the interfaces to two such external models, `SARAH` and `FeynRules`.

The `FeynRules` interface had been started already for the legacy version `WHIZARD1` (where it had to be downloaded from https://whizard.hepforge.org as a separate package), but for the `WHIZARDtwo` release series it has been included in the `FeynRules` package (from their version v1.6.0 on). Note that there was a regression for the usage of external models (from either `SARAH` or `FeynRules`) in the first release of series v2.2, v2.2.0. This has been fixed in all upcoming versions.

Besides using `SARAH` or `FeynRules` via their interfaces, there is now a much easier way to let those programs output model files in the "Universal FeynRules Output" (or `UFO`). This option does not have any principle limitations for models, and also does not rely on the never truly constant interfaces between two different tools. Their usage is described in Sec. 17.3.

17.1   New physics models via `SARAH`

`SARAH` [75,76,77,78,79] is a `Mathematica` [80] package which derives for a given model the minimum conditions of the vacuum, the mass matrices, and vertices at tree-level as well as expressions for the one-loop corrections for all masses and the full two-loop renormalization group equations (RGEs). The vertices can be exported to be used with `WHIZARD/O'Mega`. All other information can be used to generate `Fortran` source code for the RGE solution tool and spectrum generator `SPheno` [81,82] to get a spectrum generator for any model. The advantage is that `SPheno` calculates a consistent set of parameters (couplings, masses, rotation matrices, decay widths) which can be used as input for `WHIZARD`. `SARAH` and `SPheno` can be also downloaded from the `HepForge` server:

17.1.1   WHIZARD/O'Mega model files from SARAH

Generating the model files

Here we are giving only the information relevant to generate models for WHIZARD. For more details about the installation of SARAH and an exhaustion documentation about its usage, confer the SARAH manual.

To generate the model files for WHIZARD/O'Mega with SARAH, a new Mathematica session has to be started. SARAH is loaded via

```
<<SARAH-4.2.1/SARAH.m;
```

if SARAH has been stored in the applications directory of Mathematica. Otherwise, the full path has to be given

```
<<[Path_to_SARAH]/SARAH.m;
```

To get an overview which models are delivered with SARAH, the command ShowModels can be used. As an example, we use in the following the triplet extended MSSM (TMSSM) and initialize it in SARAH via

```
Start["TMSSM"];
```

Finally, the output intended for WHIZARD/O'Mega is started via

```
MakeWHIZARD[Options]
```

The possible options of the MakeWHIZARD command are

1. WriteOmega, with values: True or False, default: True
   Defines if the model files for O'Mega should be written

2. WriteWHIZARD, with values: True or False, default: True
   Defines if the model files for WHIZARD should be written

3. Exclude, with values: list of generic type, Default: {SSSS}
   Defines which generic vertices are *not* exported to the model file

4. WOModelName, with values: string, default: name of the model in SARAH followed by _sarah
   Gives the possibility to change the model name

5. MaximalCouplingsPerFile, with values: integer, default: 150
   Defines the maximal number of couplings written per file

6. Version, with values: formatted number, Default: 2.2.1 [1],
   Defines the version of WHIZARD for which the model file is generated

All options and the default values are also shown in the Mathematica session via Options[MakeWHIZARD].

---

[1]Due to a regression in WHIZARD version v2.2.0, SARAH models cannot be successfully linked within that version. Hence, the default value here has been set to version number 2.2.1

Using the generated model files with WHIZARD

After the interface has completed evaluation, the generated files can be found in the subdirectory `WHIZARD_Omega` of `SARAH`s output directory. In order to use it the generated code must be compiled and installed. For this purpose, open a terminal, enter the output directory

```
<PATH_to_SARAH>/Output/TMSSM/EWSB/WHIZARD_Omega/
```

and run

```
./configure
make install
```

By default, the last command installs the compiled model into `.whizard` in current user's home directory where it is automatically picked up by `WHIZARD`. Alternative installation paths can be specified using the `--prefix` option to `WHIZARD`.

```
./configure --prefix=/path/to/installation/prefix
```

If the files are installed into the `WHIZARD` installation prefix, the program will also pick them up automatically, while `WHIZARD`'s `--localprefix` option must be used to communicate any other choice to `WHIZARD`. In case `WHIZARD` is not available in the binary search path, the `WO_CONFIG` environment variable can be used to point `configure` to the binaries

```
./configure WO_CONFIG=/path/to/whizard/binaries
```

More information on the available options and their syntax can be obtained with the `--help` option.

After the model is compiled it can be used in `WHIZARD` as

```
model = tmssm_sarah
```

17.1.2   Linking SPheno and WHIZARD

As mentioned above, the user can also use `SPheno` to generate spectra for its models. This is done by means of `Fortran` code for `SPheno`, exported from `SARAH`. To do so, the user has to apply the command `MakeSPheno[]`. For more details about the options of this command and how to compile and use the `SPheno` output, we refer to the `SARAH` manual.

As soon as the `SPheno` version for the given model is ready it can be used to generate files with all necessary numerical values for the parameters in a format which is understood by `WHIZARD`. For this purpose, the corresponding flag in the Les Houches input file of `SPheno` has to be turned on:

```
Block SPhenoInput    # SPheno specific input
...
75 1                 # Write WHIZARD files
```

Afterwards, `SPheno` returns not only the spectrum file in the standard SUSY Les Houches accord (SLHA) format (for more details about the SLHA and the `WHIZARD` SLHA interface cf. Sec. 10.2), but also an additional file called `WHIZARD.par.TMSSM` for our example. This file can be used in the `SINDARIN` input file via

```
include ("WHIZARD.par.TMSSM")
```

17.1.3   BSM Toolbox

A convenient way to install SARAH together with WHIZARD, SPheno and some other codes are
the BSM Toolbox scripts [2] [83]. These scripts are available at

<p align="center">https://sarah.hepforge.org/Toolbox.html</p>

The Toolbox provides two scripts. First, the configure script is used via

```
toolbox-src-dir> mkdir build
toolbox-src-dir> cd build
toolbox-src-dir> ../configure
```

The configure script checks for the requirements of the different packages and downloads all
codes. All downloaded archives will be placed in the tarballs subdirectory of the directory
containing the configure script. Command line options can be used to disable specific packages
and to point the script to custom locations of compilers and of the Mathematica kernel; a full
list of those can be obtained by calling configure with the --help option.

After configure finishes successfully, make can be called to build all configured packages

```
toolbox-build-dir> make
```

configure creates also the second script which automates the implementation of a new
model into all packages. The butler script takes as argument the name of the model in SARAH,
e.g.

```
> ./butler TMSSM
```

The butler script runs SARAH to get the output in the same form as the WHIZARD/O'Mega model
files and the code for SPheno. Afterwards, it installs the model in all packages and compiles the
new WHIZARD/O'Mega model files as well as the new SPheno module.

---

[2]Those script have been published under the name SUSY Toolbox but SARAH is with version 4 no longer
restricted to SUSY models

17.2 New physics models via `FeynRules`

In this section, we present the interface between the external tool `FeynRules` [84,85,86] and `WHIZARD`. `FeynRules` is a `Mathematica` [80] package that allows to derive Feynman rules from any perturbative quantum field theory-based Lagrangian in an automated way. It can be downloaded from

http://feynrules.irmp.ucl.ac.be/

The input provided by the user is threefold and consists of the Lagrangian defining the model, together with the definitions of all the particles and parameters that appear in the model. Once this information is provided, `FeynRules` can perform basic checks on the sanity of the implementation (e.g. hermiticity, normalization of the quadratic terms), and finally computes all the interaction vertices associated with the model and store them in an internal format for later processing. After the Feynman rules have been obtained, `FeynRules` can export the interaction vertices to `WHIZARD` via a dedicated interface [87]. The interface checks whether all the vertices are compliant with the structures supported by `WHIZARD`'s matrix element generator `O'Mega`, and discard them in the case they are not supported. The output of the interface consists of a set of files organized in a single directory which can be injected into `WHIZARD/O'Mega` and used as any other built-in models. Together with the model files, a framework is created which allows to communicate the new models to `WHIZARD` in a well defined way, after which step the model can be used exactly like the built-in ones. This specifically means that the user is not required to manually modify the code of `WHIZARD/O'Mega`, the models created by the interface can be used directly without any further user intervention. We first describe the installation and general usage of the interface, and then list the general properties like the supported particle types, color quantum numbers and Lorentz structures as well as types of gauge interactions.

17.2.1 Installation and Usage of the `WHIZARD-FeynRules` interface

Installation and basic usage: From `FeynRules` version 1.6.0 onward, the interface to `WHIZARD` is part of the `FeynRules` distribution[3]. In addition, the latest version of the interface can be downloaded from the `WHIZARD` homepage on `HepForge`. There you can also find an installer that can be used to inject the interface into an existing `FeynRules` installation (which allows to use the interface with the `FeynRules` release series1.4.x where it is not part of the package).

Once installed, the interface can be called and used in the same way `FeynRules`' other interfaces described in [84]. The details of how to install and use `FeynRules` itself can be found there, [84,85,86]. Here, we only describe how to use the interface to inject new models into `WHIZARD`. For example, once the `FeynRules` environment has been initialized and a model has been loaded, the command

        WriteWOOutput[L]

---

[3]Note that though the main interface of `FeynRules` to `WHIZARD` is for the most recent `WHIZARD` release, but also the legacy branch `WHIZARD1` is supported.

will call the `FeynmanRules` command to extract the Feynman rules from the Lagrangian `L`, translate them together with the model data and finally write the files necessary for using the model within `WHIZARD` to an output directory (the name of which is inferred from the model name by default). Options can be added for further control over the translation process (see Sec. 17.2.2). Instead of using a Lagrangian, it is also possible to call the interface on a pure vertex list. For example, the following command

```
WriteWOOutput[Input -> list]
```

will directly translate the vertex list `list`. Note that this vertex list must be given in flavor-expanded form in order for the interface to process it correctly.

The interface also supports the `WriteWOExtParams` command described in [84]. Issuing

```
WriteWOExtParams[filename]
```

will write a list of all the external parameters to `filename`. This is done in the form of a `SINDARIN` script. The only option accepted by the command above is the target version of `WHIZARD`, set by the option `WOWhizardVersion`.

During execution, the interface will print out a series of messages. It is highly advised to carefully read through this output as it not only summarizes the settings and the location of the output files, but also contains information on any skipped vertices or potential incompatibilities of the model with `WHIZARD`.

After the interface has run successfully and written the model files to the output directory, the model must be imported into `WHIZARD`. For doing so, the model files have to be compiled and can then be installed independently of `WHIZARD`. In the simplest scenario, assuming that the output directory is the current working directory and that the `WHIZARD` binaries can be found in the current `${PATH}`, the installation is performed by simply executing

```
./configure~\&\&~make clean~\&\&~make install
```

This will compile the model and install it into the directory `${HOME}/.whizard`, making it fully available to `WHIZARD` without any further intervention. The build system can be adapted to more complicated cases through several options to the `configure` which are listed in the `INSTALL` file created in the output directory. A detailed explanation of all options can be found in Sec. 17.2.2.

Supported fields and vertices: The following fields are currently supported by the interface: scalars, Dirac and Majorana fermions, vectors and symmetric tensors. The set of accepted operators, the full list of which can be found in Tab. 17.1, is a subset of all the operators supported by `O'Mega`. While still limited, this list is sufficient for a large number of BSM models. In addition, a future version of `WHIZARD`/`O'Mega` will support the definition of completely general Lorentz structures in the model, allowing the interface to translate all interactions handled by `FeynRules`. This will be done by means of a parser within `O'Mega` of the `UFO` file format for model files from `FeynRules`.

Color: Color is treated in `O'Mega` in the color flow decomposition, with the flow structure being implicitly determined from the representations of the particles present at the vertex. Therefore, the interface has to strip the color structure from the vertices derived by `FeynRules`

Particle spins	Supported Lorentz structures
FFS	All operators of dimension four are supported.
FFV	All operators of dimension four are supported.
SSS	All dimension three interactions are supported.

SVV	Supported operators:
	  dimension 3: $\mathcal{O}_3 = V_1^\mu V_{2\mu} \phi$
	  dimension 5: $\mathcal{O}_5 = \phi \left(\partial^\mu V_1^\nu - \partial^\nu V_1^\mu \right) \left(\partial_\mu V_{2\nu} - \partial_\nu V_{2\mu} \right)$
	Note that $\mathcal{O}_5$ generates the effective gluon-gluon-Higgs couplings obtained by integrating out heavy quarks.

SSV	$\left(\phi_1 \partial^\mu \phi_2 - \phi_2 \partial^\mu \phi_1 \right) V_\mu$ type interactions are supported.
SSVV	All dimension four interactions are supported.
SSSS	All dimension four interactions are supported.
VVV	All parity-conserving dimension four operators are supported, with the restriction that non-gauge interactions may be split into several vertices and can only be handled if all three fields are mutually different.
VVVV	All parity conserving dimension four operators are supported.
TSS, TVV, TFF	The three point couplings in the Appendix of Ref. [89] are supported.

Table 17.1: *All Lorentz structures currently supported by the WHIZARD-FeynRules interface, sorted with respect to the spins of the particles. "S" stands for scalar, "F" for fermion (either Majorana or Dirac) and "V" for vector.*

before writing them out to the model files. While this process is straightforward for all color structures which correspond only to a single flow assignment, vertices with several possible flow configurations must be treated with care in order to avoid mismatches between the flows assigned by O'Mega and those actually encoded in the couplings. To this end, the interface derives the color flow decomposition from the color structure determined by FeynRules and rejects all vertices which would lead to a wrong flow assignment by O'Mega (these rejections are accompanied by warnings from the interface)[4].

At the moment, the $SU(3)_C$ representations supported by both WHIZARD and the interface are singlets (1), triplets (3), antitriplets ($\bar{3}$) and octets (8). Tab. 17.2 shows all combinations of these representations which can form singlets together with the support status of the respective color structures in WHIZARD and the interface. Although the supported color structures do not comprise all possible singlets, the list is sufficient for a large number of SM extensions. Furthermore, a future revision of WHIZARD/O'Mega will allow for explicit color flow assignments, thus removing most of the current restrictions.

---

[4]For the old WHIZARD1 legacy branch, there was a maximum number of external color flows that had to explicitly specified. Essentially, this is $n_8 - \frac{1}{2}n_3$ where n_8 is the maximum number of external color octets and n_3 is the maximum number of external triplets and antitriplets. This can be set in the WHIZARD/FeynRules interface by the WOMaxNcf command, whose default is 4.

$SU(3)_C$ representations	Support status
111, $\bar{3}$31, $\bar{3}$38, 1111, $\bar{3}$311, $\bar{3}$381	Fully supported by the interface
888, 8881	Supported only if at least two of the octets are identical particles.
881, 8811	Fully supported by the interface[5].
$\bar{3}$388	Supported only if the octets are identical particles.
8888	The only supported flow structure is $\cdot\ \Gamma(1,2,3,4)$ + all acyclic permutations where $\Gamma(1,2,3,4)$ represents the Lorentz structure associated with the first flow.
333, $\bar{3}\bar{3}\bar{3}$, 3331, $\bar{3}\bar{3}$31, $\bar{3}\bar{3}$33	Unsupported (at the moment)
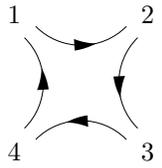
Table 17.2: *All possible combinations of three or four $SU(3)_C$ representations supported by* **FeynRules** *which can be used to build singlets, together with the support status of the corresponding color structures in* **WHIZARD** *and the interface.*

Running α_S: While a running strong coupling is fully supported by the interface, a choice has to be made which quantities are to be reevaluated when the strong coupling is evolved. By default `aS`, `G` (see Ref. [84] for the nomenclature regarding the QCD coupling) and any vertex factors depending on them are evolved. The list of internal parameters that are to be recalculated (together with the vertex factors depending on them) can be extended (beyond `aS` and `G`) by using the option `WORunParameters` when calling the interface [6].

Gauge choices: The interface supports the unitarity, Feynman and R_ξ gauges. The choice of gauge must be communicated to the interface via the option `WOGauge`. Note that massless gauge bosons are always treated in Feynman gauge.

If the selected gauge is Feynman or R_ξ, the interface can automatically assign the proper masses to the Goldstone bosons. This behavior is requested by using the `WOAutoGauge` option. In the R_ξ gauges, the symbol representing the gauge ξ must be communicated to the interface by using the `WOGaugeSymbol` option (the symbol is automatically introduced into the list of external parameters if `WOAutoGauge` is selected at the same time). This feature can be used to automatically extend models implemented in Feynman gauge to the R_ξ gauges.

Since `WHIZARD` (at least until the release series 2.3) is a tree-level tool working with helicity amplitudes, the ghost sector is irrelevant for `WHIZARD` and hence dropped by the interface.

---

[6]As the legacy branch, `WHIZARD1`, does not support a running strong coupling, this is also vetoed by the interface when using `WHIZARD1.x`.

WOWhizardVersion	WHIZARD versions supported
"2.0.3" (default)	2.0.3+
"2.0"	2.0.0 – 2.0.2
"1.96"	1.96+ (deprecated)
"1.93"	1.93 – 1.95 (deprecated)
"1.92"	1.92 (deprecated)

Table 17.3: *Currently available choices for the WOWhizardVersion option, together with the respective WHIZARD versions supported by them.*

17.2.2   Options of the WHIZARD-FeynRules interface

In the following we present a comprehensive list of all the options accepted by `WriteWOOutput`. Additionally, we note that all options of the `FeynRules` command `FeynmanRules` are accepted by `WriteWOOutput`, which passes them on to `FeynmanRules`.

Input
  An optional vertex list to use instead of a Lagrangian (which can then be omitted).

WOWhizardVersion
  Select the WHIZARD version for which code is to be generated. The currently available choices are summarized in Tab. 17.3. This list will expand as the program evolves. To get a summary of all choices available in a particular version of the interface, use the command `?WOWhizardVersion`.

WOModelName
  The name under which the model will be known to WHIZARD[7]. The default is determined from the `FeynRules` model name.

Output
  The name of the output directory. The default is determined from the `FeynRules` model name.

WOGauge
  Gauge choice (*cf.* Sec. 17.2.1). Possible values are: `WOUnitarity` (default), `WOFeynman`, `WORxi`

WOGaugeParameter
  The external or internal parameter representing the gauge ξ in the R_ξ gauges (*cf.* Sec. 17.2.1). Default: `Rxi`

WOAutoGauge
  Automatically assign the Goldstone boson masses in the Feynman and R_ξ gauges and automatically append the symbol for ξ to the parameter list in the R_ξ gauges. Default: `False`

---

[7]For versions 1.9x, model names must start with "fr_" if they are to be picked up by WHIZARD automatically.

`WORunParameters`

> The list of all internal parameters which will be recalculated if α_S is evolved (see above)[8]. Default: {`aS`, `G`}

`WOFast`

> If the interface drops vertices which are supported, this option can be set to `False` to enable some more time consuming checks which might aid the identification. Default: `True`

`WOMaxCouplingsPerFile`

> The maximum number of couplings that are written to a single `Fortran` file. If compilation takes too long or fails, this can be lowered. Default: `500`

`WOVerbose`

> Enable verbose output and in particular more extensive information on any skipped vertices. Default: `False`

17.2.3   Validation of the interface

The output of the interface has been extensively validated. Specifically, the integrated cross sections for all possible $2 \to 2$ processes in the `FeynRules` SM, the MSSM and the Three-Site Higgsless Model have been compared between `WHIZARD`, `MadGraph`, and `CalcHep`, using the respective `FeynRules` interfaces as well as the in-house implementations of these models (the Three-Site Higgsless model not being available in `MadGraph`). Also, different gauges have been checked for `WHIZARD` and `CalcHep`. In all comparisons, excellent agreement within the Monte Carlo errors was achieved. The detailed comparison including examples of the comparison tables can be found in [87].

17.2.4   Examples for the `WHIZARD-/FeynRules` interface

Here, we will use the Standard Model, the MSSM and the Three-Site Higgsless Model as prime examples to explain the usage of the interface. Those are the models that have been used in the validation of the interface in [87]. The examples are constructed to show the application of the different options of the interface and to serve as a starting point for the generation of the user's own `WHIZARD` versions of other `FeynRules` models.

`WHIZARD-FeynRules` example: Standard Model

To start off, we will create WHIZARD 2 versions of the Standard Model as implemented in `FeynRules` for different gauge choices.

---

[8]Not available for versions older than 2.0.0

SM: Unitarity Gauge In order to invoke `FeynRules`, we change to the corresponding directory and load the program in `Mathematica` via

```
$FeynRulesPath =
    SetDirectory["<path-to-FeynRules>"];
<<FeynRules`
```

The model is loaded by

```
LoadModel["Models/SM/SM.fr"];
FeynmanGauge = False;
```

Note that the second line is required to switch the Standard Model to Unitarity gauge as opposed to Feynman gauge (which is the default). Generating a `WHIZARD` model is now simply done by

```
WriteWOOutput[LSM];
```

After invokation, the interface first gives a short summary of the setup

```
Short model name is "fr_standard_model"
Gauge: Unitarity
Generating code for WHIZARD / O'Mega
                    version 2.0.3
Maximum number of couplings per FORTRAN
                        module: 500
Extensive lorentz structure checks disabled.
```

Note that, as we have not changed any options, those settings represent the defaults. The output proceeds with the calculation of the Feynman rules from the Standard Model Lagrangian `LSM`. After the rules have been derived, the interface starts generating output and tries to match the vertices to their `WHIZARD`/`O'Mega` counterparts.

```
10 of 75 vertices processed...
20 of 75 vertices processed...
30 of 75 vertices processed...
40 of 75 vertices processed...
50 of 75 vertices processed...
60 of 75 vertices processed...
70 of 75 vertices processed...
processed a total of 75 vertices, kept 74
    of them and threw away 1, 1 of which
    contained ghosts or goldstone bosons.
```

The last line of the above output is particularily interesting, as it informs us that everything worked out correctly: the interface was able to match all vertices, and the only discarded vertex was the QCD ghost interaction. After the interface has finished running, the model files in the output directory are ready to use and can be compiled using the procedure described in the previous section.

SM: Feynman and R_ξ gauges As the Standard Model as implemented in `FeynRules` also supports Feynman gauge, we can use the program to generate a Feynman gauge version of the model. Loading `FeynRules` and the model proceeds as above, with the only difference being the change

```
FeynmanGauge = True;
```

In order to inform the interface about the modified gauge, we have to add the option `WOGauge`

```
WriteWOOutput[LSM, WOGauge -> WOFeynman];
```

The modified gauge is reflected in the output of the interface

```
Short model name is "fr_standard_model"
Gauge: Feynman
Generating code for WHIZARD / O'Mega
                      version 2.0.3
Maximum number of couplings per FORTRAN
                          module: 500
Extensive lorentz structure checks disabled.
```

The summary of the vertex identification now takes the following form

```
processed a total of 163 vertices, kept 139
  of them and threw away 24, 24 of which
  contained ghosts.
```

Again, this line tells us that there were no problems — the only discarded interactions involved the ghost sector which is irrelevant for the tree-level part of `WHIZARD`.

For a tree-level calculation, the only difference between the different gauges from the perspective of the interface are the gauge boson propagators and the Goldstone boson masses. Therefore, the interface can automatically convert a model in Feynman gauge to a model in R_ξ gauge. To this end, the call to the interface must be changed to

```
WriteWOOutput[LSM, WOGauge -> WORxi,
              WOAutoGauge -> True];
```

The `WOAutoGauge` argument instructs the interface to automatically

1. Introduce a symbol for the gauge parameter ξ into the list of external parameters

2. Generate the Goldstone boson masses from those of the associated gauge bosons (ignoring the values provided by `FeynRules`)

The modified setup is again reflected in the interface output

```
Short model name is "fr_standard_model"
Gauge: Rxi
Gauge symbol: "Rxi"
Generating code for WHIZARD / O'Mega
                      version 2.0.3
Maximum number of couplings per FORTRAN
                          module: 500
Extensive lorentz structure checks disabled.
```

Note the default choice `Rxi` for the name of the ξ parameter – this can be modified via the option `WOGaugeParameter`.

While the `WOAutoGauge` feature allows to generate R_ξ gauged models from models implemented in Feynman gauge, it is of course also possible to use models genuinely implemented in R_ξ gauge by setting this parameter to `False`. Also, note that the choice of gauge only affects the propagators of massive fields. Massless gauge bosons are always treated in Feynman gauge.

Compilation and usage In order to compile and use the freshly generated model files, change to the output directory which can be determined from the interface output (in this example, it is `fr_standard_model-WO`). Assuming that WHIZARD is available in the binary search path, compilation and installation proceeds as described above by executing

```
./configure && make && make install
```

The model is now ready and can be used similarly to the builtin WHIZARD models. For example, a minimal WHIZARD input file for calculating the $e^+e^- \longrightarrow W^+W^-$ scattering cross section in the freshly generated model would look like

```
model = fr_standard_model
process test = "e+", "e-" -> "W+", "W-"
sqrts = 500 GeV
integrate (test)
```

`WHIZARD`/`FeynRules` example: MSSM

In this Section, we illustrate the usage of the interface between FEYNRULES and WHIZARD in the context of the MSSM. All the parameters of the model are then ordered in Les Houches blocks and counters following the SUSY Les Houches Accord (SLHA) [52,53,54] (cf. also Sec. 10.2).

After having downloaded the model from the `FeynRules` website, we store it in a new directory, labelled `MSSM`, of the model library of the local installation of `FeynRules`. The model can then be loaded in `Mathematica` as in the case of the SM example above

```
$FeynRulesPath =
        SetDirectory["<path-to-FeynRules>"];
<<FeynRules`
LoadModel["Models/MSSM/MSSM.fr"];
FeynmanGauge = False;
```

We are again adopting unitarity gauge.

The number of vertices associated to supersymmetric Lagrangians is in general very large (several thousands). For such models with many interactions, it is recommended to first extract all the Feynman rules of the theory before calling the interface between WHIZARD and `FeynRules`. The reason is related to the efficiency of the interface which takes a lot of time in the extraction of the interaction vertices. In the case one wishes to study the phenomenology of several benchmark scenarios, this procedure, which is illustrated below, allows to use the interface in the best way. The Feynman rules are derived from the Lagrangian once and for all and then reused by the interface for each set of WHIZARD model files to be produced, considerably speeding up the generation of multiple model files issued from a single Lagrangian. In addition, the scalar potential of supersymmetric theories contains a large set of four scalar interactions, in general irrelevant for collider phenomenology. These vertices can be neglected with the help of the `Exclude4Scalars->True` option of both interface commands `FeynmanRules` and `WriteWOOutput`. The Feynman rules of the MSSM are then computed within the `Mathematica` notebook by

```
rules = FeynmanRules[lag,
    Exclude4Scalars->True, FlavorExpand->True];
```

where `lag` is the variable containing the Lagrangian.

By default, all the parameters of the model are set to the value of 1. A complete parameter `<slha_params>.dat` file must therefore be loaded. Such a parameter file can be downloaded from the `FeynRules` website or created by hand by the user, and loaded into `FeynRules` as

```
ReadLHAFile[Input -> "<slha_params>.dat"];
```

This command does not reduce the size of the model output by removing vertices with vanishing couplings. However, if desired, this task could be done with the `LoadRestriction` command (see Ref. [90] for details).

The vertices are exported to WHIZARD by the command

```
WriteWOOutput[Input -> rules];
```

Note that the numerical values of the parameters of the model can be modified directly from WHIZARD, without having to generate a second time the WHIZARD model files from `FeynRules`. A SINDARIN script is created by the interface with the help of the instruction

```
WriteWOExtParams["parameters.sin"];
```

and can be further modified according to the needs of the user.

WHIZARD-FeynRules example: Three-Site Higgsless Model

The Three-Site Higgsless model or Minimal Higgsless model (MHM) has been implemented into `LanHEP` [91], `FeynRules` and independently into WHIZARD [39], and the collider phenomenology has been studied by making use of these implementations [91,50,39]. Furthermore, the independent implementations in `FeynRules` and directly into WHIZARD have been compared and found to agree [87]. After the discovery of a Higgs boson at the LHC in 2012, such a model is not in good agreement with experimental data any more. Here, we simply use it as a guinea pig to describe the handling of a model with non-renormalizable interactions with the `FeynRules` interface, and discuss how to generate WHIZARD model files for it. The model has been implemented in Feynman gauge as well as unitarity gauge and contains the variable `FeynmanGauge` which can be set to `True` or `False`. When set to `True`, the option `WOGauge-> WOFeynman` must be used, as explained in [87]. R_ξ gauge can also be accomplished with this model by use of the options `WOGauge -> WORxi` and `WOAutoGauge -> True`.

Since this model makes use of a nonlinear sigma field of the form

$$\Sigma = 1 + i\pi - \frac{1}{2}\pi^2 + \cdots \tag{17.1}$$

many higher dimensional operators are included in the model which are not currently not supported by WHIZARD. Even for a future release of WHIZARD containing general Lorentz structures in interaction vertices, the user would be forced to expand the series only up to a certain order. Although WHIZARD can reject these vertices and print a warning message to the user, it is preferable to remove the vertices right away in the interface by the option `MaxCanonicalDimension->4`. This is passed to the command `FeynmanRules` and restricts the Feynman rules to those of dimension four and smaller[9].

---

[9]`MaxCanonicalDimension` is an option of the `FeynmanRules` function rather than of the interface, itself. In fact, the interface accepts all the options of `FeynmanRules` and simply passes them on to the latter.

As the use of different gauges was already illustrated in the SM example, we discuss the model only in Feynman gauge here. We load `FeynRules`:

```
$FeynRulesPath =
    SetDirectory["<path-to-FeynRules>"];
<<FeynRules`
```

The MHM model itself is then loaded by

```
SetDirectory["<path-to-MHM>"];
LoadModel["3-Site-particles.fr",
    "3-Site-parameters.fr",
    "3-Site-lagrangian.fr"];
FeynmanGauge = True;
```

where `<path-to-MHM>` is the path to the directory where the MHM model files are stored and where the output of the `WHIZARD` interface will be written. The `WHIZARD` interface is then initiated:

```
WriteWOOutput[LGauge, LGold, LGhost, LFermion,
    LGoldLeptons, LGoldQuarks,
    MaxCanonicalDimension->4,
    WOGauge->WOFeynman, WOModelName->"fr_mhm"];
```

where we have also made use of the option `WOModelName` to change the name of the model as seen by `WHIZARD`. As in the case of the SM, the interface begins by writing a short informational message:

```
Short model name is "fr_mhm"
Gauge: Feynman
Generating code for WHIZARD / O'Mega
                    version 2.0.3
Automagically assigning Goldstone
                    boson masses...
Maximum number of couplings per FORTRAN
                    module: 500
Extensive lorentz structure checks disabled.
```

After calculating the Feynman rules and processing the vertices, the interface gives a summary:

```
processed a total of 922 vertices, kept 633
  of them and threw away 289, 289 of which
  contained ghosts.
```

showing that no vertices were missed. The files are stored in the directory `fr_mhm` and are ready to be installed and used with `WHIZARD`.

17.3   New physics models via the `UFO` file format

In this section, we describe how to use the *Universal FeynRules Output* (`UFO`, [88]) format for physics models inside `WHIZARD`. Please refer the manuals of e.g. `FeynRules` manual for details on how to generate a `UFO` file for your favorite physics model. `UFO` files are a collection of `Python`

scripts that encode the particles, the couplings, the Lorentz structures, the decays, as well as parameters, vertices and propagators of the corresponding model. They reside in a directory of the exact name of the model they have been created from.

If the user wants to generate events for processes from a physics model from a UFO file, then this directory of scripts generated by FeynRules is immediately available if it is a subdirectory of the working directory of WHIZARD. The directory name will be taken as the model name. (The UFO-model file name must not start with a non-letter character, i.e. especially not a number. In case such a file name wants to be used at all costs, the model name in the SINDARIN script has to put in quotation marks, but this is not guaranteed to always work.) Then, a UFO model named, e.g., test_model is accessed by an extra ufo tag in the model assignment:

```
model = test_model (ufo)
```

If desired, WHIZARD can access a directory of UFO files elsewhere on the file system. For instance, if FeynRules output resides in the subdirectory MyMdl of /home/users/john/ufo, WHIZARD can use the model named MyMdl as follows

```
model = MyMdl (ufo ('/home/users/john/my_ufo_models'))
```

that is, the SINDARIN keyword ufo can take an argument. Note however, that the latter approach can backfire — in case just the working directory is packed and archived for future reference.

Appendix A

SINDARIN Reference

In the SINDARIN language, there are certain pre-defined constructors or commands that cannot be used in different context by the user, which are e.g. alias, beams, integrate, simulate etc. A complete list will be given below. Also units are fixed, like degree, eV, keV, MeV, GeV, and TeV. Again, these tags are locked and not user-redefinable. Their functionality will be listed in detail below, too. Furthermore, a variable with a preceding question mark, ?, is a logical, while a preceding dollar, $, denotes a character string variable. Also, a lot of unary and binary operators exist, + - \ , = : => < > <= >= ^ () [] {} ==, as well as quotation marks, ". Note that the different parentheses and brackets fulfill different purposes, which will be explained below. Comments in a line can either be marked by a hash, #, or an exclamation mark, !.

A.1 Commands and Operators

We begin the SINDARIN reference with all commands, operators, functions and constructors. The list of variables (which can be set to change behavior of WHIZARD) can be found in the next section.

- +
  1) Arithmetic operator for addition of integers, reals and complex numbers. Example: `real mm = mH + mZ` (cf. also -, *, /, ^). 2) It also adds different particles for inclusive process containers: `process foo = e1, E1 => (e2, E2) + (e3, E3)`. 3) It also serves as a shorthand notation for the concatenation of (→) combine operations on particles/subevents, e.g. `cuts = any 170 GeV < M < 180 GeV [b + lepton + invisible]`.

- -
  Arithmetic operator for subtraction of integers, reals and complex numbers. Example: `real foo = 3.1 - 5.7` (cf. also +, *, /, ^).

- /
  Arithmetic operator for division of integers, reals and complex numbers. Example: `scale`

`= mH / 2` (cf. also `+`, `*`, `-`, `^`).

- `*`

  Arithmetic operator for multiplication of integers, reals and complex numbers. Example:
  `complex z = 2 * I` (cf. also `+`, `/`, `-`, `^`).

- `^`

  Arithmetic operator for exponentiation of integers, reals and complex numbers. Example:
  `real z = x^2 + y^2` (cf. also `+`, `/`, `-`, `^`).

- `<`

  Arithmetic comparator between values that checks for ordering of two values: **`<val1>`**
  `<` **`<val2>`** tests whether **`val1`** is smaller than **`val2`**. Allowed for integer and real values.
  Note that this is an exact comparison if `tolerance` is set to zero. For a finite value of
  `tolerance` it is a "fuzzy" comparison. (cf. also `tolerance`, `<>`, `==`, `>`, `>=`, `<=`)

- `>`

  Arithmetic comparator between values that checks for ordering of two values: **`<val1>`**
  `>` **`<val2>`** tests whether **`val1`** is larger than **`val2`**. Allowed for integer and real values.
  Note that this is an exact comparison if `tolerance` is set to zero. For a finite value of
  `tolerance` it is a "fuzzy" comparison. (cf. also `tolerance`, `<>`, `==`, `>`, `>=`, `<=`)

- `<=`

  Arithmetic comparator between values that checks for ordering of two values: **`<val1>`** `<=`
  `<val2>` tests whether **`val1`** is smaller than or equal **`val2`**. Allowed for integer and real
  values. Note that this is an exact comparison if `tolerance` is set to zero. For a finite
  value of `tolerance` it is a "fuzzy" comparison. (cf. also `tolerance`, `<>`, `==`, `>`, `<`, `>=`)

- `>=`

  Arithmetic comparator between values that checks for ordering of two values: **`<val1>`** `>=`
  `<val2>` tests whether **`val1`** is larger than or equal **`val2`**. Allowed for integer and real
  values. Note that this is an exact comparison if `tolerance` is set to zero. For a finite
  value of `tolerance` it is a "fuzzy" comparison. (cf. also `tolerance`, `<>`, `==`, `>`, `<`, `>=`)

- `==`

  Arithmetic comparator between values that checks for identity of two values: **`<val1>`** `==`
  `<val2>`. Allowed for integer and real values. Note that this is an exact comparison if
  `tolerance` is set to zero. For a finite value of `tolerance` it is a "fuzzy" comparison. (cf.
  also `tolerance`, `<>`, `>`, `<`, `>=`, `<=`)

- `<>`

  Arithmetic comparator between values that checks for two values being unequal: **`<val1>`**
  `<>` **`<val2>`**. Allowed for integer and real values. Note that this is an exact comparison if
  `tolerance` is set to zero. For a finite value of `tolerance` it is a "fuzzy" comparison. (cf.
  also `tolerance`, `==`, `>`, `<`, `>=`, `<=`)

- !

  The exclamation mark tells SINDARIN that everything that follows in that line should be treated as a comment. It is the same as ($\rightarrow$) #.

- #

  The hash tells SINDARIN that everything that follows in that line should be treated as a comment. It is the same as ($\rightarrow$) !.

- &

  Concatenates two or more particle lists/subevents and hence acts in the same way as the subevent function ($\rightarrow$) join: let @visible = [photon] & [colored] & [lepton] in .... (cf. also join, combine, collect, extract, sort).

- $

  Constructor at the beginning of a variable name, $*<string_var>*, that specifies a string variable.

- @

  Constructor at the beginning of a variable name, @*<subevt_var>*, that specifies a subevent variable, e.g. let @W_candidates = combine ["mu-", "numubar"] in ....

- =

  Binary constructor to appoint values to commands, e.g. *<command>* = *<expr>* or *<command>* *<var_name>* = *<expr>*.

- %

  Constructor that gives the percentage of a number, so in principle multiplies a real number by 0.01. Example: 1.23 % is equal to 0.0123.

- :

  Separator in alias expressions for particles, e.g. alias neutrino = n1:n2:n3:N1:N2:N3. (cf. also alias)

- ;

  Concatenation operator for logical expressions: *lexpr1* ; *lexpr2*. Evaluates *lexpr1* and throws the result away, then evaluates *lexpr2* and returns that result. Used in analysis expressions. (cf. also analysis, record)

- /+

  Incrementor for ($\rightarrow$) scan ranges, that increments additively, scan *<num_spec>* *<num>* = (*<lower val>* => *<upper val>* /+ *<step size>*). E.g. scan int i = (1 => 5 /+ 2) scans over the values 1, 3, 5. For real ranges, it divides the interval between upper and lower bound into as many intervals as the incrementor provides, e.g. scan real r = (1 => 1.5 /+ 0.2) runs over 1.0, 1.333, 1.667, 1.5.

- `/+/`

  Incrementor for ($\rightarrow$) `scan` ranges, that increments additively, but the number after the incrementor is the number of steps, not the step size: `scan <num_spec> <num> = (<lower val> => <upper val> /+/ <steps>)`. It is only available for real scan ranges, and divides the interval `<upper val>` – `<lower val>` into `<steps>` steps, e.g. `scan real r = (1 => 1.5 /+/ 3)` runs over `1.0, 1.25, 1.5`.

- `/-`

  Incrementor for ($\rightarrow$) `scan` ranges, that increments subtractively, `scan <num_spec> <num> = (<lower val> => <upper val> /- <step size>)`. E.g. `scan int i = (9 => 0 /+ 3)` scans over the values `9, 6, 3, 0`. For real ranges, it divides the interval between upper and lower bound into as many intervals as the incrementor provides, e.g. `scan real r = (1 => 0.5 /- 0.2)` runs over `1.0, 0.833, 0.667, 0.5`.

- `/*`

  Incrementor for ($\rightarrow$) `scan` ranges, that increments multiplicatively, `scan <num_spec> <num> = (<lower val> => <upper val> /* <step size>)`. E.g. `scan int i = (1 => 4 /* 2)` scans over the values `1, 2, 4`. For real ranges, it divides the interval between upper and lower bound into as many intervals as the incrementor provides, e.g. `scan real r = (1 => 5 /* 2)` runs over `1.0, 2.236` (i.e. $\sqrt{5}$), `5.0`.

- `/*/`

  Incrementor for ($\rightarrow$) `scan` ranges, that increments multiplicatively, but the number after the incrementor is the number of steps, not the step size: `scan <num_spec> <num> = (<lower val> => <upper val> /*/ <steps>)`. It is only available for real scan ranges, and divides the interval `<upper val>` – `<lower val>` into `<steps>` steps, e.g. `scan real r = (1 => 9 /*/ 4)` runs over `1.000, 2.080, 4.327, 9.000`.

- `//`

  Incrementor for ($\rightarrow$) `scan` ranges, that increments by division, `scan <num_spec> <num> = (<lower val> => <upper val> // <step size>)`. E.g. `scan int i = (13 => 0 // 3)` scans over the values `13, 4, 1, 0`. For real ranges, it divides the interval between upper and lower bound into as many intervals as the incrementor provides, e.g. `scan real r = (5 => 1 // 2)` runs over `5.0, 2.236` (i.e. $\sqrt{5}$), `1.0`.

- `=>`

  Binary operator that is used in several different contexts: 1) in process declarations between the particles specifying the initial and final state, e.g. `process <proc_name> = <in1>, <in2> => <out1>, ....`; 2) for the specification of beams when structure functions are applied to the beam particles, e.g. `beams = p, p => pdf_builtin`; 3) for the specification of the scan range in the `scan <var> <var_name> = (<scan_start> => <scan_end> <incrementor>)` (cf. also `process`, `beams`, `scan`)

- `%d`

  Format specifier in analogy to the `C` language for the print out on screen by the ($\rightarrow$)

`printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for decimal integer numbers, e.g. `printf "one = %d" (i)`. The difference between `%i` and `%d` does not play a role here. (cf. also `printf`, `sprintf`, `%i`, `%e`, `%f`, `%g`, `%E`, `%F`, `%G`, `%s`)

- `%e`
  Format specifier in analogy to the C language for the print out on screen by the ($\rightarrow$) `printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for floating-point numbers in standard form `[-]d.ddd e[+/-]ddd`. Usage e.g. `printf "pi = %e" (PI)`. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%f`, `%g`, `%E`, `%F`, `%G`, `%s`)

- `%E`
  Same as ($\rightarrow$) `%e`, but using upper-case letters. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%e`, `%f`, `%g`, `%F`, `%G`, `%s`)

- `%f`
  Format specifier in analogy to the C language for the print out on screen by the ($\rightarrow$) `printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for floating-point numbers in fixed-point form. Usage e.g. `printf "pi = %f" (PI)`. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%e`, `%g`, `%E`, `%F`, `%G`, `%s`)

- `%F`
  Same as ($\rightarrow$) `%f`, but using upper-case letters. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%e`, `%f`, `%g`, `%E`, `%G`, `%s`)

- `%g`
  Format specifier in analogy to the C language for the print out on screen by the ($\rightarrow$) `printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for floating-point numbers in normal or exponential notation, whichever is more approriate. Usage e.g. `printf "pi = %g" (PI)`. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%e`, `%f`, `%E`, `%F`, `%G`, `%s`)

- `%G`
  Same as ($\rightarrow$) `%g`, but using upper-case letters. (cf. also `printf`, `sprintf`, `%d`, `%i`, `%e`, `%f`, `%g`, `%E`, `%F`, `%s`)

- `%i`
  Format specifier in analogy to the C language for the print out on screen by the ($\rightarrow$) `printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for integer numbers, e.g. `printf "one = %i" (i)`. The difference between `%i` and `%d` does not play a role here. (cf. `printf`, `sprintf`, `%d`, `%e`, `%f`, `%g`, `%E`, `%F`, `%G`, `%s`)

- `%s`
  Format specifier in analogy to the C language for the print out on screen by the ($\rightarrow$) `printf` or into strings by the ($\rightarrow$) `sprintf` command. It is used for logical or string variables e.g. `printf "foo = %s" ($method)`. (cf. `printf`, `sprintf`, `%d`, `%i`, `%e`, `%f`, `%g`, `%E`, `%F`, `%G`)

- abarn
  Physical unit, stating that a number is in attobarns (10^{-18} barn). (cf. also nbarn, fbarn, pbarn)

- abs
  Numerical function that takes the absolute value of its argument: abs (*<num_val>*) yields |*<num_val>*|. (cf. also conjg, sgn, mod, modulo)

- acos
  Numerical function asin (*<num_val>*) that calculates the arccosine trigonometric function (inverse of cos) of real and complex numerical numbers or variables. (cf. also sin, cos, tan, asin, atan)

- alias
  This allows to define a collective expression for a class of particles, e.g. to define a generic expression for leptons, neutrinos or a jet as alias lepton = e1:e2:e3:E1:E2:E3, alias neutrino = n1:n2:n3:N1:N2:N3, and alias jet = u:d:s:c:U:D:S:C:g, respectively.

- all
  all is a function that works on a logical expression and a list, all *<log_expr>* [*<list>*], and returns true if and only if log_expr is fulfilled for *all* entries in list, and false otherwise. Examples: all Pt > 100 GeV [lepton] checks whether all leptons are harder than 100 GeV, all Dist > 2 [u:U, d:D] checks whether all pairs of corresponding quarks are separated in R space by more than 2. Logical expressions with all can be logically combined with and and or. (cf. also any, and, no, and or)

- alt_setup
  This command allows to specify alternative setups for a process/list of processes, alt_setup = { *<setup1>* } [, { *<setup2>* } , ...]. An alternative setup can be a resetting of a coupling constant, or different cuts etc. It can be particularly used in a ($\rightarrow$) rescan procedure.

- analysis
  This command, analysis = *<log_expr>*, allows to define an analysis as a logical expression, with a syntax similar to the ($\rightarrow$) cuts or ($\rightarrow$) selection command. Note that a ($\rightarrow$) formally is a logical expression.

- and
  This is the standard two-place logical connective that has the value true if both of its operands are true, otherwise a value of false. It is applied to logical values, e.g. cut expressions. (cf. also all, no, or).

- any
  any is a function that works on a logical expression and a list, any *<log_expr>* [*<list>*], and returns true if log_expr is fulfilled for any entry in list, and false otherwise. Examples: any PDG == 13 [lepton] checks whether any lepton is a muon, any E > 2 *

mW [jet] checks whether any jet has an energy of twice the W mass. Logical expressions with any can be logically combined with and and or. (cf. also all, and, no, and or)

- **as**
  cf. compile

- **ascii**
  Specifier for the sample_format command to demand the generation of the standard WHIZARD verbose/debug ASCII event files. (cf. also $sample, $sample_normalization, sample_format)

- **asin**
  Numerical function asin (*<num_val>*) that calculates the arcsine trigonometric function (inverse of sin) of real and complex numerical numbers or variables. (cf. also sin, cos, tan, acos, atan)

- **atan**
  Numerical function atan (*<num_val>*) that calculates the arctangent trigonometric function (inverse of tan) of real and complex numerical numbers or variables. (cf. also sin, cos, tan, asin, acos)

- **athena**
  Specifier for the sample_format command to demand the generation of the ATHENA variant for HEPEVT ASCII event files. (cf. also $sample, $sample_normalization, sample_format)

- **beam**
  Constructor that specifies a particle (in a subevent) as beam particle. It is used in cuts, analyses or selections, e.g. cuts = all Theta > 20 degree [beam lepton, lepton]. (cf. also incoming, outgoing, cuts, analysis, selection, record)

- **beam_events**
  Beam structure specifier to read in lepton collider beamstrahlung's spectra from external files as pairs of energy fractions: beams: e1, E1 => beam_events. Note that this is a pair spectrum that has to be applied to both beams simultaneously. (cf. also beams, $beam_events_file, ?beam_events_warn_eof)

- **beams**
  This specifies the contents and structure of the beams: beams = *<prt1>*, *<prt2>* [=> *<str_fun1>* ....]. If this command is absent in the input file, WHIZARD automatically takes the two incoming partons (or one for decays) of the corresponding process as beam particles, and no structure functions are applied. Protons and antiprotons as beam particles are predefined as p and pbar, respectively. A structure function, like pdf_builtin, ISR, EPA and so on are switched on as e.g. beams = p, p => lhapdf. Structure functions can be specified for one of the two beam particles only, of the structure function is not a spectrum. (cf. also beams_momentum, beams_theta, beams_phi,

`beams_pol_density`, `beams_pol_fraction`, `beam_events`, `circe1`, `circe2`, `energy_scan`, `epa`, `ewa`, `isr`, `lhapdf`, `pdf_builtin`).

- `beams_momentum`
  Command to set the momenta (or energies) for the two beams of a scattering process:
  `beams_momentum = <mom1>, <mom2>` to allow for asymmetric beam setups (e.g. HERA:
  `beams_momentum = 27.5 GeV, 920 GeV`). Two arguments must be present for a scattering
  process, but the command can be used with one argument to integrate and simulate a
  decay of a moving particle. (cf. also `beams`, `beams_theta`, `beams_phi`, `beams_pol_density`,
  `beams_pol_fraction`)

- `beams_phi`
  Same as ($\rightarrow$) `beams_theta`, but to allow for a non-vanishing beam azimuth angle, too. (cf.
  also `beams`, `beams_theta`, `beams_momentum`, `beams_pol_density`, `beams_pol_fraction`)

- `beams_pol_density`
  This command allows to specify the initial state for polarized beams by the syntax:
  `beams_pol_density = @(<pol_spec_1>), @(<pol_spec_2>)`. Two polarization specifiers
  are mandatory for scattering, while one can be used for decays from polarized probes.
  The specifier `<pol_spec_i>` can be empty (no polarization), has one entry (for a definite
  helicity/spin orientation), or ranges of entries of a spin density matrix. The command
  can be used globally, or as a local argument of the `integrate` command. For detailed
  information, see Sec. 5.6.1. It is also possible to use variables as placeholders in the
  specifiers. Note that polarization is assumed to be complete, for partial polarization use
  ($\rightarrow$) `beams_pol_fraction`. (cf. also `beams`, `beams_theta`, `beams_phi`, `beams_momentum`,
  `beams_pol_fraction`)

- `beams_pol_fraction`
  This command allows to specify the amount of polarization when using polarized beams
  ($\rightarrow$ `beams_pol_density`). The syntax is: `beams_pol_fraction = <frac_1>, <frac_2>`.
  Two fractions must be present for scatterings, being real numbers between 0 and 1. A
  specification with percentage is also possible, e.g. `beams_pol_fraction = 80%, 40%`. (cf.
  also `beams`, `beams_theta`, `beams_phi`, `beams_momentum`, `beams_pol_density`)

- `beams_theta`
  Command to set a crossing angle (with respect to the z axis) for one or both of the beams
  of a scattering process: `beams_theta = <angle1>, <angle2>` to allow for asymmetric
  beam setups (e.g. `beams_angle = 0, 10 degree`). Two arguments must be present for
  a scattering process, but the command can be used with one argument to integrate and
  simulate a decay of a moving particle. (cf. also `beams`, `beams_phi`, `beams_momentum`,
  `beams_pol_density`, `beams_pol_fraction`)

- `by`
  Constructor that replaces the default sorting criterion (according to PDG codes) of the
  ($\rightarrow$) `sort` function on particle lists/subevents by one given by a unary or binary particle

observable: `sort by <observable> [<particles> [, <ref_particles>] ]`. (cf. also `sort`, `extract`, `join`, `collect`, `combine`, `+`)

- `ceiling`
  This is a function `ceiling (<num_val>)` that gives the least integer greater than or equal to `<num_val>`, e.g. `int i = ceiling (4.56789)` gives `i = 5`. (cf. also `int`, `nint`, `floor`)

- `circe1`
  Beam structure specifier for the `CIRCE1` structure function for beamstrahlung at a linear lepton collider: `beams = e1, E1 => circe1`. Note that this is a pair spectrum, so the specifier acts for both beams simultaneously. (cf. also `beams`, `?circe1_photons`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`)

- `circe2`
  Beam structure specifier for the lepton-collider structure function for photon spectra, `CIRCE2`: `beams = A, A => circe2`. Note that this is a pair spectrum, an application to only one beam is not possible. (cf. also `beams`, `?circe2_polarized`, `$circe2_file`, `$circe2_design`)

- `clear`
  This command allows to clear a variable set before: `clear (<clearable var.>)` resets the variable `<clearable var.>` which could be the `beams`, the `unstable` settings, `sqrts`, any kind of `cuts` or `scale` expressions, any user-set variable etc. The syntax of the command is completely analogous to (→) `show`.

- `close_out`
  With the command, `close_out ("<out_file">)` user-defined information like data or (→) `printf` statements can be written out to a user-defined file. The command closes an I/O stream to an external file `<out_file>`. (cf. also `open_out`, `$out_file`, `printf`)

- `cluster`
  Command that allows to cluster all particles in a subevent to a set of jets: `cluster [<particles>]`. It also to cluster particles subject to a certain boolean condition, `cluster if <condition> [<particles>]`. At the moment only available if the `FastJet` package is linked. (cf. also `jet_r`, `combine`, `jet_algorithm`, `kt_algorithm`, `cambridge_[for_passive_]algorithm`, `antikt_algorithm`, `plugin_algorithm`, `genkt_[for_passive_]algorithm`, `ee_kt_algorithm`, `ee_genkt_algorithm`, `?keep_flavors_when_cl`

- `collect`
  The `collect [<list>]` operation collects all particles in the list `<list>` into a one-entry subevent with a four-momentum of the sum of all four-momenta of non-overlapping particles in `<list>`. (cf. also `combine`, `select`, `extract`, `sort`)

- complex
  Defines a complex variable. The syntax is e.g. `complex x = 2 + 3 * I`. (cf. also `int`, `real`)

- combine
  The `combine [<list1>, <list2>]` operation makes a particle list whose entries are the result of adding (the momenta of) each pair of particles in the two input lists `list1`, list2. For example, `combine [incoming lepton, lepton]` constructs all mutual pairings of an incoming lepton with an outgoing lepton (an alias for the leptons has to be defined, of course). (cf. also `collect`, `select`, `extract`, `sort`, `+`)

- compile
  The `compile ()` command has no arguments (the parentheses can also been left out: `/compile ()`. The command is optional, it invokes the compilation of the process(es) (i.e. the matrix element file(s)) to be compiled as a shared library. This shared object file has the standard name `default_lib.so` and resides in the `.libs` subdirectory of the corresponding user workspace. If the user has defined a different library name `lib_name` with the `library` command, then WHIZARD compiles this as the shared object `.libs/lib_name.so`. (This allows to split process classes and to avoid too large libraries.) Another possibility is to use the command `compile as "static_name"`. This will compile and link the process library in a static way and create the static executable `static_name` in the user workspace. (cf. also `library`)

- compile_analysis
  The `compile_analysis` statement does the same as the `write_analysis` command, namely to tell WHIZARD to write the analysis setup by the user for the SINDARIN input file under consideration. If no `$out_file` is provided, the histogram tables/plot data etc. are written to the default file `whizard_analysis.dat`. In addition to `write_analysis`, `compile_analysis` also invokes the WHIZARD LaTeXroutines for producing postscript or PDF output of the data (unless the flag → `?analysis_file_only` is set to `true`). (cf. also `$out_file`, `write_analysis`, `?analysis_file_only`)

- conjg
  Numerical function that takes the complex conjugate of its argument: `conjg (<num_val>)` yields `<num_val>`*. (cf. also `abs`, `sgn`, `mod`, `modulo`)

- cos
  Numerical function `cos (<num_val>)` that calculates the cosine trigonometric function of real and complex numerical numbers or variables. (cf. also `sin`, `tan`, `asin`, `acos`, `atan`)

- cosh
  Numerical function `cosh (<num_val>)` that calculates the hyperbolic cosine function of real and complex numerical numbers or variables. Note that its inverse function is part of the `Fortran2008` status and hence not realized. (cf. also `sinh`, `tanh`)

- count
  Subevent function that counts the number of particles or particle pairs in a subevent: count [*<particles_1>* [, *<particles_2>*]]. This can also be a counting subject to a condition: count if *<condition>* [*<particles_1>* [, *<particles_2>*]].

- cuts
  This command defines the cuts to be applied to certain processes. The syntax is: cuts = *<log_class>* *<log_expr>* [*<unary or binary particle (list) arg>*], where the cut expression must be initialized with a logical classifier log_class like all, any, no. The logical expression log_expr contains the cut to be evaluated. Note that this need not only be a kinematical cut expression like E > 10 GeV or 5 degree < Theta < 175 degree, but can also be some sort of trigger expression or event selection. Whether the expression is evaluated on particles or pairs of particles depends on whether the discriminating variable is unary or binary, Dist being obviously binary, Pt being unary. Note that some variables are both unary and binary, e.g. the invariant mass M. Cut expressions can be connected by the logical connectives and and or. The cuts statement acts on all subsequent process integrations and analyses until a new cuts statement appears. (cf. also all, any, Dist, E, M, no, Pt).

- debug
  Specifier for the sample_format command to demand the generation of the very verbose WHIZARD ASCII event file format intended for debugging. (cf. also $sample, sample_format, $sample_normalization)

- degree
  Expression specifying the physical unit of degree for angular variables, e.g. the cut expression function Theta. (if no unit is specified for angular variables, radians are used; cf. rad, mrad).

- Dist
  Binary observable specifier, that gives the η-ϕ- (pseudorapidity-azimuth) distance $R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2}$ between the momenta of the two particles: eval Dist [jet, jet]. (cf. also eval, cuts, selection, Theta, Eta, Phi)

- dump
  Specifier for the sample_format command to demand the generation of the intrinsic WHIZARD event record format (output of the particle_t type container). (cf. also $sample, sample_format, $sample_normalization

- E
  Unary (binary) observable specifier for the energy of a single (two) particle(s), e.g. eval E ["W+"], all E > 200 GeV [b, B]. (cf. eval, cuts, selection)

- else
  Constructor for providing an alternative in a conditional clause: if *<log_expr>* then *<expr 1>* else *<expr 2>* endif. (cf. also if, elsif, endif, then).

- **elsif**
  Constructor for concatenating more than one conditional clause with each other: `if <log_expr 1> then <expr 1> elsif <log_expr 2> then <expr 2> ...endif`. (cf. also `if`, `else`, `endif`, `then`).

- **endif**
  Mandatory constructor to conclude a conditional clause: `if <log_expr> then ...endif`. (cf. also `if`, `else`, `elsif`, `then`).

- **energy_scan**
  Beam structure specifier for the energy scan structure function: `beams = e1, E1 => energy_scan`. This pair spectrum that has to be applied to both beams simultaneously can be used to scan over a range of collider energies without using the `scan` command. (cf. also `beams`, `scan`, `?energy_scan_normalize`)

- **epa**
  Beam structure specifier for the equivalent-photon approximation (EPA), i.e the Weizsäcker-Williams structure function: e.g. `beams = e1, E1 => epa` (applied to both beams), or e.g. `beams = e1, u => epa, none` (applied to only one beam). (cf. also `beams`, `epa_alpha`, `epa_x_min`, `epa_mass`, `epa_q_max`, `epa_q_min`, `?epa_recoil`, `?epa_keep_energy`)

- **Eta**
  Unary and also binary observable specifier, that as a unary observable gives the pseudorapidity of a particle momentum. The pseudorapidity is given by $\eta = -\log\left[\tan(\theta/2)\right]$, where θ is the angle with the beam direction. As a binary observable, it gives the pseudorapidity difference between the momenta of two particles, where θ is the enclosed angle: `eval Eta [e1]`, `all abs (Eta) < 3.5 [jet, jet]`. (cf. also `eval`, `cuts`, `selection`, `Rap`, `abs`)

- **eV**
  Physical unit, stating that the corresponding number is in electron volt. (cf. also `keV`, `meV`, `MeV`, `GeV`, `TeV`)

- **eval**
  Evaluator that tells WHIZARD to evaluate the following expr: `eval <expr>`. Examples are: `eval Rap [e1]`, `eval M / 1 GeV [combine [q,Q]]` etc. (cf. also `cuts`, `selection`, `record`, `sum`, `prod`)

- **ewa**
  Beam structure specifier for the equivalent-photon approximation (EWA): e.g. `beams = e1, E1 => ewa` (applied to both beams), or e.g. `beams = e1, u => ewa, none` (applied to only one beam). (cf. also `beams`, `ewa_x_min`, `ewa_pt_max`, `ewa_mass`, `?ewa_keep_energy`, `?ewa_recoil`)

- **exec**
  Constructor `exec ("<cmd_name>")` that demands WHIZARD to execute/run the com-

mand `cmd_name`. For this to work that specific command must be present either in the path of the operating system or as a command in the user workspace.

- `exit`
  Command to finish the `WHIZARD` run (and not execute any further code beyond the appearance of `exit` in the `SINDARIN` file. The command (which is the same as → `quit`) allows for an argument, `exit (<expr>)`, where the expression can be executed, e.g. a screen message or an exit code.

- `exp`
  Numerical function `exp (<num_val>)` that calculates the exponential of real and complex numerical numbers or variables. (cf. also `sqrt`, `log`, `log10`)

- `expect`
  The binary function `expect` compares two numerical expressions whether they fulfill a certain ordering condition or are equal up to a specific uncertainty or tolerance which can bet set by the specifier `tolerance`, i.e. in principle it checks whether a logical expression is true. The `expect` function does actually not just check a value for correctness, but also records its result. If failures are present when the program terminates, the exit code is nonzero. The syntax is `expect (<num1> <log_comp> <num2>)`, where `<num1>` and `<num2>` are two numerical values (or corresponding variables) and `<log_comp>` is one of the following logical comparators: `<, >, <=, >=, ==, <>`. (cf. also `<, >, <=, >=, ==, <>`, `tolerance`).

- `extract`
  Subevent function that either extracts the first element of a particle list/subevent: `extract [ <particles>]`, or the element at position `<index_value>` of the particle list: `extract index <index_value> [ <particles>]`. Negative index values count from the end of the list. (cf. also `sort`, `combine`, `collect`, `+`, `index`)

- `factorization_scale`
  This is a command, `factorization_scale = <expr>`, that sets the factorization scale of a process or list of processes. It overwrites a possible scale set by the (→) `scale` command. `<expr>` can be any kinematic expression that leads to a result of momentum dimension one, e.g. `100 GeV`, `eval Pt [e1]`. (cf. also `renormalization_scale`).

- `false`
  Constructor stating that a logical expression or variable is false, e.g. `?<log_var> = false`. (cf. also `true`).

- `fbarn`
  Physical unit, stating that a number is in femtobarns (10^{-15} barn). (cf. also `nbarn`, `abarn`, `pbarn`)

- **floor**
  This is a function `floor (<num_val>)` that gives the greatest integer less than or equal to
  `<num_val>`, e.g. `int i = floor (4.56789)` gives `i = 4`. (cf. also `int`, `nint`, `ceiling`)

- **gaussian**
  Beam structure specifier that imposes a Gaussian energy distribution, separately for each
  beam. The σ values are set by `gaussian_spread1` and `gaussian_spread2`, respectively.

- **GeV**
  Physical unit, energies in 10^9 electron volt. This is the default energy unit of WHIZARD.
  (cf. also `eV`, `keV`, `MeV`, `meV`, `TeV`)

- **graph**
  This command defines the necessary information regarding producing a graph of a function
  in WHIZARD's internal graphical `gamelan` output. The syntax is: `graph <record_name>`
  `{ <optional arguments> }`. The record with name `<record_name>` has to be defined,
  either before or after the graph definition. Possible optional arguments of the `graph`
  command are the minimal and maximal values of the axes (`x_min`, `x_max`, `y_min`, `y_max`).
  (cf. `plot`, `histogram`, `record`)

- **Hel**
  Unary observable specifier that allows to specify the helicity of a particle, e.g. `all Hel`
  `== -1 [e1]` in a selection. (cf. also `eval`, `cuts`, `selection`)

- **hepevt**
  Specifier for the `sample_format` command to demand the generation of HEPEVT ASCII
  event files. (cf. also `$sample`, `sample_format`)

- **hepevt_verb**
  Specifier for the `sample_format` command to demand the generation of the extended or
  verbose version of HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)

- **hepmc**
  Specifier for the `sample_format` command to demand the generation of HepMC ASCII
  event files. Note that this is only available if the HepMC package is installed and correctly
  linked. (cf. also `$sample`, `sample_format`, `?hepmc_output_cross_section`)

- **histogram**
  This command defines the necessary information regarding plotting data as a histogram, in
  the form of: `histogram <record_name> { <optional arguments> }`. The record with
  name `<record_name>` has to be defined, either before or after the histogram definition.
  Possible optional arguments of the `histogram` command are the minimal and maximal
  values of the axes (`x_min`, `x_max`, `y_min`, `y_max`). (cf. `graph`, `plot`, `record`)

- **Ht**
  Subeventary observable specifier for the transverse mass ($\sqrt{p_T^2 + m^2}$ in the c.m. frame)

summed over all particles in the subevent given as argument, e.g. `eval Ht [t:T:Z]`. (cf.
`eval`, `sum`, `prod`, `Pt`, `M`)

- `if`
  Conditional clause with the construction `if <log_expr> then <expr>` [else `<expr>`
  `...]` `endif`. Note that there must be an `endif` statement. For more complicated expres-
  sions it is better to use expressions in parentheses: `if (<log_expr>) then {<expr>}`
  `else {<expr>} endif`. Examples are a selection of up quarks over down quarks depend-
  ing on a logical variable: `if ?ok then u else d`, or the setting of an integer variable
  depending on the rapidity of some particle: `if (eta > 0) then { a = +1} else { a
  = -1}`. (cf. also `elsif`, `endif`, `then`)

- `in`
  Second part of the constructor to let a variable be local to an expression. It has the
  syntax `let <var> = <value> in <expression>`. E.g. `let int a = 3 in let int b
  = 4 in <expression>` (cf. also `let`)

- `include`
  The `include` statement, `include ("file.sin")` allows to include external SINDARIN files
  `file.sin` into the main WHIZARD input file. A standard example is the inclusion of the
  standard cut file `default_cuts.sin`.

- `incoming`
  Constructor that specifies particles (or subevents) as incoming. It is used in cuts, analyses
  or selections, e.g. `cuts = all Theta > 20 degree [incoming lepton, lepton]`. (cf.
  also `beam`, `outgoing`, `cuts`, `analysis`, `selection`, `record`)

- `index`
  Specifies the position of the element of a particle to be extracted by the subevent function
  ($\rightarrow$) extract: `extract index <index_value> [ <particles>]`. Negative index values
  count from the end of the list. (cf. also `extract`, `sort`, `combine`, `collect`, `+`)

- `int`
  1) This is a constructor to specify integer constants in the input file. Strictly speaking,
  it is a unary function setting the value `int_val` of the integer variable `int_var`: `int
  <int_var> = <int_val>`. Note that is mandatory for all user-defined variables. (cf. also
  `real` and `complex`) 2) It is a function `int (<num_val>)` that converts real and complex
  numbers (here their real parts) into integers. (cf. also `nint`, `floor`, `ceiling`)

- `integrate`
  The `integrate (<proc_name>) { <integrate_options> }` command invokes the inte-
  gration (phase-space generation and Monte-Carlo sampling) of the process `proc_name`
  (which can also be a list of processes) with the integration options `<integrate_options>`.
  Possible options are (1) via `$integration_method = "<intg. method>"` the integra-
  tion method (the default being VAMP), (2) the number of iterations and calls per

integration during the Monte-Carlo phase-space integration via the `iterations` specifier; (3) goal for the accuracy, error or relative error (`accuracy_goal`, `error_goal`, `relative_error_goal`). (4) Invoking only phase space generation (`?phs_only = true`), (5) making test calls of the matrix element. (cf. also `iterations`, `accuracy_goal`, `error_goal`, `relative_error_goal`, `error_threshold`)

- `isr`
  Beam structure specifier for the lepton-collider/QED initial-state radiation (ISR) structure function: e.g. `beams = e1, E1 => isr` (applied to both beams), or e.g. `beams = e1, u => isr, none` (applied to only one beam). (cf. also `beams`, `isr_alpha`, `isr_q_max`, `isr_mass`, `isr_order`, `?isr_recoil`, `?isr_keep_energy`)

- `iterations`   (default: internal heuristics)
  Option to set the number of iterations and calls per iteration during the Monte-Carlo phase-space integration process. The syntax is `iterations = <n_iterations>:<n_calls>`. Note that this can be also a list, separated by colons, which breaks up the integration process into passes of the specified number of integrations and calls each. It works for all integration methods. For VAMP, there is the additional option to specify whether grids and channel weights should be adapted during iterations (`"g"`, `"w"`, `"gw"` for both, or `""` for no adaptation). (cf. also `integrate`, `accuracy_goal`, `error_goal`, `relative_error_goal`, `error_threshold`).

- `join`
  Subevent function that concatenates two particle lists/subevents if there is no overlap: `join [<particles>, <new_particles>]`. The joining of the two lists can also be made depending on a condition: `join if <condition> [<particles>, <new_particles>]`. (cf. also `&`, `collect`, `combine`, `extract`, `sort`, `+`)

- `keV`
  Physical unit, energies in 10^3 electron volt. (cf. also `eV`, `meV`, `MeV`, `GeV`, `TeV`)

- `kT`
  Binary particle observable that represents a jet k_T clustering measure: `kT [j1, j2]` gives the following kinematic expression: $2\min(E_{j1}^2, E_{j2}^2)/Q^2 \times (1 - \cos\theta_{j1,j2})$. At the moment, $Q^2 = 1$.

- `let`
  This allows to let a variable be local to an expression. It has the syntax `let <var> = <value> in <expression>`. E.g. `let int a = 3 in let int b = 4 in <expression>` (cf. also `in`)

- `lha`
  Specifier for the `sample_format` command to demand the generation of the `WHIZARD` version 1 style (deprecated) LHA ASCII event format files. (cf. also `$sample`, `sample_format`)

- `lhapdf`
  This is a beams specifier to demand calling `LHAPDF` parton densities as structure functions to integrate processes in hadron collisions. Note that this only works if the external `LHAPDF` library is present and correctly linked. (cf. `beams`, `$lhapdf_dir`, `$lhapdf_file`, `lhapdf_photon`, `$lhapdf_photon_file`, `lhapdf_member`, `lhapdf_photon_scheme`)

- `lhapdf_photon`
  This is a beams specifier to demand calling `LHAPDF` parton densities as structure functions to integrate processes in hadron collisions with a photon as initializer of the hard scattering process. Note that this only works if the external `LHAPDF` library is present and correctly linked. (cf. `beams`, `lhapdf`, `$lhapdf_dir`, `$lhapdf_file`, `$lhapdf_photon_file`, `lhapdf_member`, `lhapdf_photon_scheme`)

- `lhef`
  Specifier for the `sample_format` command to demand the generation of the Les Houches Accord (LHEF) event format files, with XML headers. There are several different versions of this format, which can be selected via the `$lhef_version` specifier (cf. also `$sample`, `sample_format`, `$lhef_version`, `$lhef_extension`, `?lhef_write_sqme_prc`, `?lhef_write_sqme_ref`, `?lhef_write_sqme_alt`)

- `library`
  The command `library = "<lib_name>"` allows to specify a separate shared object library archive `lib_name.so`, not using the standard library `default_lib.so`. Those libraries (when using shared libraries) are located in the `.libs` subdirectory of the user workspace. Specifying a separate library is useful for splitting up large lists of processes, or to restrict a larger number of different loaded model files to one specific process library. (cf. also `compile`, `$library_name`)

- `log`
  Numerical function `log (<num_val>)` that calculates the natural logarithm of real and complex numerical numbers or variables. (cf. also `sqrt`, `exp`, `log10`)

- `log10`
  Numerical function `log10 (<num_val>)` that calculates the base 10 logarithm of real and complex numerical numbers or variables. (cf. also `sqrt`, `exp`, `log`)

- `long`
  Specifier for the `sample_format` command to demand the generation of the long variant of HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)

- `M`
  Unary (binary) observable specifier for the (signed) mass of a single (two) particle(s), e.g. `eval M [e1]`, `any M = 91 GeV [e2, E2]`. (cf. `eval`, `cuts`, `selection`)

- **M2**
  Unary (binary) observable specifier for the mass squared of a single (two) particle(s), e.g.
  `eval M2 [e1]`, `all M2 > 2*mZ [e2, E2]`. (cf. `eval`, `cuts`, `selection`)

- **max**
  Numerical function with two arguments `max (<var1>, <var2>)` that gives the maximum
  of the two arguments: $\max(var1, var2)$. It can act on all combinations of integer and real
  variables. Example: `real heavier_mass = max (mZ, mH)`. (cf. also `min`)

- **meV**
  Physical unit, stating that the corresponding number is in 10^{-3} electron volt. (cf. also `eV`,
  `keV`, `MeV`, `GeV`, `TeV`)

- **MeV**
  Physical unit, energies in 10^6 electron volt. (cf. also `eV`, `keV`, `meV`, `GeV`, `TeV`)

- **min**
  Numerical function with two arguments `min (<var1>, <var2>)` that gives the minimum
  of the two arguments: $\min(var1, var2)$. It can act on all combinations of integer and real
  variables. Example: `real lighter_mass = min (mZ, mH)`. (cf. also `max`)

- **mod**
  Numerical function for integer and real numbers `mod (x, y)` that computes the remainder
  of the division of `x` by `y` (which must not be zero). (cf. also `abs`, `conjg`, `sgn`, `modulo`)

- **model**   (default: `SM`)
  With this specifier, `model = <model_name>`, one sets the hard interaction physics model
  for the processes defined after this model specification. The list of available models can be
  found in Table 10.1. Note that the model specification can appear arbitrarily often in a
  SINDARIN input file, e.g. for compiling and running processes defined in different physics
  models. (cf. also `$model_name`)

- **modulo**
  Numerical function for integer and real numbers `modulo (x, y)` that computes the value
  of x modulo y. (cf. also `abs`, `conjg`, `sgn`, `mod`)

- **mokka**
  Specifier for the `sample_format` command to demand the generation of the MOKKA
  variant for HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)

- **mrad**
  Expression specifying the physical unit of milliradians for angular variables. This default
  in WHIZARD is `rad`. (cf. `degree`, `rad`).

- **nbarn**
  Physical unit, stating that a number is in nanobarns (10^{-9} barn). (cf. also `abarn`, `fbarn`,
  `pbarn`)

- `n_in`
  Integer variable that accesses the number of incoming particles of a process. It can be used in cuts or in an analysis. (cf. also `sqrts_hat`, `cuts`, `record`, `n_out`, `n_tot`)

- `Nacl`
  Unary observable specifier that returns the total number of open anticolor lines of a particle or subevent (i.e., composite particle). Defined only if `?colorize_subevt` is true.. (cf. also `Ncol`, `?colorize_subevt`)

- `Ncol`
  Unary observable specifier that returns the total number of open color lines of a particle or subevent (i.e., composite particle). Defined only if `?colorize_subevt` is true.. (cf. also `Nacl`, `?colorize_subevt`)

- `nint`
  This is a function `nint (<num_val>)` that converts real numbers into the closest integer, e.g. `int i = nint (4.56789)` gives `i = 5`. (cf. also `int`, `floor`, `ceiling`)

- `no`
  `no` is a function that works on a logical expression and a list, `no <log_expr> [<list>]`, and returns `true` if and only if `log_expr` is fulfilled for *none* of the entries in `list`, and `false` otherwise. Examples: `no Pt < 100 GeV [lepton]` checks whether no lepton is softer than 100 GeV. It is the logical opposite of the function `all`. Logical expressions with `no` can be logically combined with `and` and `or`. (cf. also `all`, `any`, `and`, and `or`)

- `none`
  Beams specifier that can used to explicitly *not* apply a structure function to a beam, e.g. in HERA physics: `beams = e1, P => none, pdf_builtin`. (cf. also `beams`)

- `not`
  This is the standard logical negation that converts true into false and vice versa. It is applied to logical values, e.g. cut expressions. (cf. also `and`, `or`).

- `n_out`
  Integer variable that accesses the number of outgoing particles of a process. It can be used in cuts or in an analysis. (cf. also `sqrts_hat`, `cuts`, `record`, `n_in`, `n_tot`)

- `n_tot`
  Integer variable that accesses the total number of particles (incoming plus outgoing) of a process. It can be used in cuts or in an analysis. (cf. also `sqrts_hat`, `cuts`, `record`, `n_in`, `n_out`)

- `observable`
  With this, `observable = <obs_spec>`, the user is able to define a variable specifier `obs_spec` for observables. These can be reused in the analysis, e.g. as a `record`, as functions of the fundamental kinematical variables of the processes. (cf. `analysis`, `record`)

- **open_out**
  With the command, open_out ("*<out_file">*) user-defined information like data or ($\rightarrow$) printf statements can be written out to a user-defined file. The command opens an I/O stream to an external file *<out_file>*. (cf. also close_out, $out_file, printf)

- **or**
  This is the standard two-place logical connective that has the value true if one of its operands is true, otherwise a value of false. It is applied to logical values, e.g. cut expressions. (cf. also and, not).

- **outgoing**
  Constructor that specifies particles (or subevents) as outgoing. It is used in cuts, analyses or selections, e.g. cuts = all Theta > 20 degree [incoming lepton, outgoing lepton]. Note that the outgoing keyword is redundant and included only for completeness: outgoing lepton has the same meaning as lepton. (cf. also beam, incoming, cuts, analysis, selection, record)

- **P**
  Unary (binary) observable specifier for the spatial momentum $\sqrt{\vec{p}^2}$ of a single (two) particle(s), e.g. eval P ["W+"], all P > 200 GeV [b, B]. (cf. eval, cuts, selection)

- **pbarn**
  Physical unit, stating that a number is in picobarns (10^{-12} barn). (cf. also abarn, fbarn, nbarn)

- **pdf_builtin**
  This is a beams specifier for WHIZARD's internal PDF structure functions to integrate processes in hadron collisions. (cf. beams, pdf_builtin_photon, $pdf_builtin_file)

- **pdf_builtin_photon**
  This is a beams specifier for WHIZARD's internal PDF structure functions to integrate processes in hadron collisions with a photon as initializer of the hard scattering process. (cf. beams, $pdf_builtin_file)

- **PDG**
  Unary observable specifier that allows to specify the PDG code of a particle, e.g. eval PDG [e1], giving 11. (cf. also eval, cuts, selection)

- **Phi**
  Unary and also binary observable specifier, that as a unary observable gives the azimuthal angle of a particle's momentum in the detector frame (beam into $+z$ direction). As a binary observable, it gives the azimuthal difference between the momenta of two particles: eval Phi [e1], all Phi > Pi [jet, jet]. (cf. also eval, cuts, selection, Theta)

- **photon_isolation**
  Logical function photon_isolation if *<condition>* [*<list1>* , *<list2>*] that cuts

out event where the photons in **<list1>** do not fulfill the condition **<condition>** and are not isolated from hadronic (and electromagnetic) activity, i.e. the photon fragmentation. (cf. also `cluster`, `collect`, `combine`, `extract`, `select`, `sort`, `+`)

- `photon_recombination`
  Similar to the `cluster` statement takes a subevent as argument and combines a (single) photon with the closest non-photon object given in the subevent. Depends on the `SINDARIN` variable `photon_rec_r0` which gives the R radius within which the photon is recombined. (cf. also `cluster`, `collect`, `combine`)

- `Pl`
  Unary (binary) observable specifier for the longitudinal momentum (p_z in the c.m. frame) of a single (two) particle(s), e.g. `eval Pl ["W+"]`, `all Pl > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)

- `plot`
  This command defines the necessary information regarding plotting data as a graph, in the form of: `plot <record_name> { <optional arguments> }`. The record with name **<record_name>** has to be defined, either before or after the plot definition. Possible optional arguments of the `plot` command are the minimal and maximal values of the axes (`x_min`, `x_max`, `y_min`, `y_max`). (cf. `graph`, `histogram`, `record`)

- `polarized`
  Constructor to instruct `WHIZARD` to retain polarization of the corresponding particles in the generated events: `polarized <prt1> [, <prt2> , ...]`. (cf. also `unpolarized`, `simulate`, `?polarized_events`)

- `printf`
  Command that allows to print data as screen messages, into logfiles or into user-defined output files: `printf "<string_expr>"`. There exist format specifiers, very similar to the C command `printf`, e.g. `printf "%i" (123)`. (cf. also `open_out`, `close_out`, `$out_file`, `?out_advance`, `sprintf`, %d, %i, %e, %f, %g, %E, %F, %G, %s)

- `process`
  Allows to set a hard interaction process, either for a decay process with name **<decay_proc>** as `process <decay_proc> = <mother> => <daughter1>, <daughter2>, ...`, or for a scattering process with name **<scat_proc>** as `process <scat_proc> = <in1>, <in2> => <out1>, <out2>, ...`. Note that there can be arbitrarily many processes to be defined in a `SINDARIN` input file. There are two options for particle/process sums: flavor sums: **<prt1>:<prt2>:...**, where all masses have to be identical, and inclusive sums, **<prt1> + <prt2> + ...**. The latter can be done on the level of individual particles, or sums over whole final states. Here, masses can differ, and terms will be translated into different process components. The `process` command also allows for optional arguments, e.g. to specify a numerical identifier (cf. `process_num_id`), the method how to generate the code for the matrix element(s): `$method`, possible methods are either with the `O'Mega`

matrix element generator, using template matrix elements with different normalizations, or completely internal matrix element; for `O'Mega` matrix elements there is also the possibility to specify possible restrictions (cf. `$restrictions`).

- `prod`
  Takes the product of an expression `<expr>` over the elements of the given subevent `<subevt>`, `prod <expr> [<subevt>]`, e.g. `prod Hel [e1:E1]` (cf. `eval`, `sum`).

- `Pt`
  Unary (binary) observable specifier for the transverse momentum ($\sqrt{p_x^2 + p_y^2}$ in the c.m. frame) of a single (two) particle(s), e.g. `eval Pt ["W+"]`, `all Pt > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)

- `Px`
  Unary (binary) observable specifier for the x-component of the momentum of a single (two) particle(s), e.g. `eval Px ["W+"]`, `all Px > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)

- `Py`
  Unary (binary) observable specifier for the y-component of the momentum of a single (two) particle(s), e.g. `eval Py ["W+"]`, `all Py > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)

- `Pz`
  Unary (binary) observable specifier for the z-component of the momentum of a single (two) particle(s), e.g. `eval Pz ["W+"]`, `all Pz > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)

- `quit`
  Command to finish the `WHIZARD` run (and not execute any further code beyond the appearance of `quit` in the `SINDARIN` file. The command (which is the same as $\rightarrow$ `exit`) allows for an argument, `quit (<expr>)`, where the expression can be executed, e.g. a screen message or an quit code.

- `rad`
  Expression specifying the physical unit of radians for angular variables. This is the default in `WHIZARD`. (cf. `degree`, `mrad`).

- `Rap`
  Unary and also binary observable specifier, that as a unary observable gives the rapidity of a particle momentum. The rapidity is given by $y = \frac{1}{2} \log\left[(E + p_z)/(E - p_z)\right]$. As a binary observable, it gives the rapidity difference between the momenta of two particles: `eval Rap [e1]`, `all abs (Rap) < 3.5 [jet, jet]`. (cf. also `eval`, `cuts`, `selection`, `Eta`, `abs`)

- `read_slha`
  Tells `WHIZARD` to read in an input file in the SUSY Les Houches accord (SLHA), as `read_slha ("slha_file.slha")`. Note that the files for the use in `WHIZARD` should have the suffix `.slha`. (cf. also `write_slha`, `?slha_read_decays`, `?slha_read_input`, `?slha_read_spectrum`)

- `real`
  This is a constructor to specify real constants in the input file. Strictly speaking, it is a unary function setting the value `real_val` of the real variable `real_var`: `real <real_var> = <real_val>`. (cf. also `int` and `complex`)

- `real_epsilon`
  Predefined real; the relative uncertainty intrinsic to the floating point type of the `Fortran` compiler with which `WHIZARD` has been built.

- `real_precision`
  Predefined integer; the decimal precision of the floating point type of the `Fortran` compiler with which `WHIZARD` has been built.

- `real_range`
  Predefined integer; the decimal range of the floating point type of the `Fortran` compiler with which `WHIZARD` has been built.

- `real_tiny`
  Predefined real; the smallest number which can be represented by the floating point type of the `Fortran` compiler with which `WHIZARD` has been built.

- `record`
  The `record` constructor provides an internal data structure in `SINDARIN` input files. Its syntax is in general `record <record_name> (<cmd_expr>)`. The `<cmd_expr>` could be the definition of a tuple of points for a histogram or an `eval` constructor that tells `WHIZARD` e.g. by which rule to calculate an observable to be stored in the record `record_name`. Example: `record h (12)` is a record for a histogram defined under the name `h` with the single data point (bin) at value 12; `record rap1 (eval Rap [e1])` defines a record with name `rap1` which has an evaluator to calculate the rapidity (predefined `WHIZARD` function) of an outgoing electron. (cf. also `eval`, `histogram`, `plot`)

- `renormalization_scale`
  This is a command, `renormalization_scale = <expr>`, that sets the renormalization scale of a process or list of processes. It overwrites a possible scale set by the ($\rightarrow$) `scale` command. `<expr>` can be any kinematic expression that leads to a result of momentum dimension one, e.g. `100 GeV`, `eval Pt [e1]`. (cf. also `factorization_scale`).

- `rescan`
  This command allows to rescan event samples with modified model parameter, beam structure etc. to recalculate (analysis) observables, e.g.:

rescan "*<event_file>*" (*<proc_name>*) { *<rescan_setup>*}.
"*<event_file>*" is the name of the event file and *<proc_name>* is the process whose
(existing) event file of arbitrary size that is to be rescanned. Several flags allow to
reconstruct the beams ($\rightarrow$?recover_beams), to reuse only the hard process but rebuild
the full events ($\rightarrow$?update_event), to recalculate the matrix element ($\rightarrow$?update_sqme)
or to recalculate the individual event weight ($\rightarrow$?update_weight). Further rescan options
are redefining model parameter input, or defining a completely new alternative setup ($\rightarrow$
alt_setup) (cf. also $rescan_input_format)

- results
  Only used in the combination show (results). Forces WHIZARD to print out a results
  summary for the integrated processes. (cf. also show)

- reweight
  The reweight = *<expr>* command allows to give for a process or list of processes an
  alternative weight, given by any kind of scalar expression *<expr>*, e.g. reweight =
  0.2 or reweight = (eval M2 [e1, E1]) / (eval M2 [e2, E2]). (cf. also alt_setup,
  weight, rescan)

- sample_format
  Variable that allows the user to specify additional event formats beyond the WHIZARD
  native binary event format. Its syntax is sample_format = *<format>*, where *<format>*
  can be any of the following specifiers: hepevt, hepevt_verb, ascii, athena, debug,
  long, short, hepmc, lhef, lha, lha_verb, stdhep, stdhep_up, lcio, mokka. (cf. also
  $sample, simulate, hepevt, ascii, athena, debug, long, short, hepmc, lhef, lha,
  stdhep, stdhep_up, lcio, mokka, $sample_normalization, ?sample_pacify,
  sample_max_tries, sample_split_n_evt, sample_split_n_kbytes)

- scale
  This is a command, scale = *<expr>*, that sets the kinematic scale of a process or list
  of processes. Unless overwritten explicitly by ($\rightarrow$) factorization_scale and/or ($\rightarrow$)
  renormalization_scale it sets both scales. *<expr>* can be any kinematic expression
  that leads to a result of momentum dimension one, e.g. scale = 100 GeV, scale = eval
  Pt [e1].

- scan
  Constructor to perform loops over variables or scan over processes in the integration
  procedure. The syntax is scan *<var>* *<var_name>* (*<value list>* or *<value_init>*
  => *<value_fin>* /*<incrementor>* *<increment>*) { *<scan_cmd>* }. The variable var
  can be specified if it is not a real, e.g. an integer. var_name is the name of the variable
  which is also allowed to be a predefined one like seed. For the scan, one can either
  specify an explicit list of values value list, or use an initial and final value and a
  rule to increment. The scan_cmd can either be just a show to print out the scanned
  variable or the integration of a process. Examples are: scan seed (32 => 1 // 2) {
  show (seed_value) } , which runs the seed down in steps 32, 16, 8, 4, 2, 1 (division

by two). `scan mW (75 GeV, 80 GeV => 82 GeV /+ 0.5 GeV, 83 GeV => 90 GeV /*`
`1.2) { show (sw) }` scans over the W mass for the values 75, 80, 80.5, 81, 81.5, 82, 83
GeV, namely one discrete value, steps by adding 0.5 GeV, and increase by 20 % (the latter
having no effect as it already exceeds the final value). It prints out the corresponding
value of the effective mixing angle which is defined as a dependent variable in the model
input file(s). `scan sqrts (500 GeV => 600 GeV /+ 10 GeV) { integrate (proc) }`
integrates the process `proc` in eleven increasing 10 GeV steps in center-of-mass energy
from 500 to 600 GeV. (cf. also `/+`, `/+/`, `/-`, `/*`, `/*/`, `//`)

- `select`
  Subevent function `select if <condition> [<list1> [ , <list2>]]` that selects all
  particles in *<list1>* that satisfy the condition *<condition>*. The second particle list
  <list2> is for conditions that depend on binary observables. (cf. also `collect`, `combine`,
  `extract`, `sort`, `+`)

- `select_b_jet`
  Subevent function `select if <condition> [<list1> [ , <list2>]]` that selects all
  particles in *<list1>* that are b jets and satisfy the condition *<condition>*. The second
  particle list *<list2>* is for conditions that depend on binary observables. (cf. also
  `cluster`, `collect`, `combine`, `extract`, `select`, `sort`, `+`)

- `select_c_jet`
  Subevent function `select if <condition> [<list1> [ , <list2>]]` that selects all
  particles in *<list1>* that are c jets (but *not* b jets) and satisfy the condition *<condition>*.
  The second particle list *<list2>* is for conditions that depend on binary observables. (cf.
  also `cluster`, `collect`, `combine`, `extract`, `select`, `sort`, `+`)

- `select_light_jet`
  Subevent function `select if <condition> [<list1> [ , <list2>]]` that selects all
  particles in *<list1>* that are light(-flavor) jets and satisfy the condition *<condition>*.
  The second particle list *<list2>* is for conditions that depend on binary observables. (cf.
  also `cluster`, `collect`, `combine`, `extract`, `select`, `sort`, `+`)

- `select_non_b_jet`
  Subevent function `select if <condition> [<list1> [ , <list2>]]` that selects all
  particles in *<list1>* that are *not* b jets (c and light jets) and satisfy the condition
  <condition>. The second particle list *<list2>* is for conditions that depend on binary
  observables. (cf. also `cluster`, `collect`, `combine`, `extract`, `select`, `sort`, `+`)

- `selection`
  Command that allows to select particular final states in an analysis selection, `selection`
  `= <log_expr>`. The term `log_expr` can be any kind of logical expression. The syntax
  matches exactly the one of the ($\rightarrow$) `cuts` command. E.g. `selection = any PDG == 13`
  is an electron selection in a lepton sample.

- `sgn`
  Numerical function for integer and real numbers that gives the sign of its argument: `sgn` (`<num_val>`) yields $+1$ if `<num_val>` is positive or zero, and -1 otherwise. (cf. also `abs`, `conjg`, `mod`, `modulo`)

- `short`
  Specifier for the `sample_format` command to demand the generation of the short variant of HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)

- `show`
  This is a unary function that is operating on specific constructors in order to print them out in the WHIZARD screen output as well as the log file `whizard.log`. Examples are `show(<parameter_name>)` to issue a specific parameter from a model or a constant defined in a SINDARIN input file, `show(integral(<proc_name>))`, `show(library)`, `show(results)`, or `show(<var>)` for any arbitrary variable. Further possibilities are `show(real)`, `show(string)`, `show(logical)` etc. to allow to show all defined real, string, logical etc. variables, respectively. (cf. also `library`, `results`)

- `simulate`
  This command invokes the generation of events for the process `proc` by means of `simulate (<proc>)`. Optional arguments: `$sample`, `sample_format`, `checkpoint` (cf. also `integrate`, `luminosity`, `n_events`, `$sample`, `sample_format`, `checkpoint`, `?unweighted`, `safety_factor`, `?negative_weights`, `sample_max_tries`, `sample_split_n_evt`, `sample_split_n_kby`

- `sin`
  Numerical function `sin` (`<num_val>`) that calculates the sine trigonometric function of real and complex numerical numbers or variables. (cf. also `cos`, `tan`, `asin`, `acos`, `atan`)

- `sinh`
  Numerical function `sinh` (`<num_val>`) that calculates the hyperbolic sine function of real and complex numerical numbers or variables. Note that its inverse function is part of the `Fortran2008` status and hence not realized. (cf. also `cosh`, `tanh`)

- `sort`
  Subevent function that allows to sort a particle list/subevent either by increasing PDG code: `sort [<particles>]` (particles first, then antiparticles). Alternatively, it can sort according to a unary or binary particle observable (in that case there is a second particle list, where the first particle is taken as a reference): `sort by <observable>` `[<particles> [, <ref_particles>]]`. (cf. also `extract`, `combine`, `collect`, `join`, `by`, `+`)

- `sprintf`
  Command that allows to print data into a string variable: `sprintf "<string_expr>"`. There exist format specifiers, very similar to the C command `sprintf`, e.g. `sprintf "%i"` (123). (cf. `printf`, `%d`, `%i`, `%e`, `%f`, `%g`, `%E`, `%F`, `%G`, `%s`)

- sqrt
  Numerical function sqrt (*<num_val>*) that calculates the square root of real and complex numerical numbers or variables. (cf. also exp, log, log10)

- sqrts_hat
  Real variable that accesses the partonic energy of a hard-scattering process. It can be used in cuts or in an analysis, e.g. cuts = sqrts_hat > *<num>* [*<phys_unit>*]. The physical unit can be one of the following eV, keV, MeV, GeV, and TeV. (cf. also sqrts, cuts, record)

- stable
  This constructor allows particles in the final states of processes in decay cascade set-up to be set as stable, and not letting them decay. The syntax is stable *<prt_name>* (cf. also unstable)

- stdhep
  Specifier for the sample_format command to demand the generation of binary StdHEP event files based on the HEPEVT common block. (cf. also $sample, sample_format)

- stdhep_up
  Specifier for the sample_format command to demand the generation of binary Std-HEP event files based on the HEPRUP/HEPEUP common blocks. (cf. also $sample, sample_format)

- sum
  Takes the sum of an expression <expr> over the elements of the given subevent <subevt>, sum <expr> [<subevt>], e.g. sum Pt/2 [jets] (cf. eval, prod).

- tan
  Numerical function tan (*<num_val>*) that calculates the tangent trigonometric function of real and complex numerical numbers or variables. (cf. also sin, cos, asin, acos, atan)

- tanh
  Numerical function tanh (*<num_val>*) that calculates the hyperbolic tangent function of real and complex numerical numbers or variables. Note that its inverse function is part of the Fortran2008 status and hence not realized. (cf. also cosh, sinh)

- TeV
  Physical unit, for energies in 10^{12} electron volt. (cf. also eV, keV, MeV, meV, GeV)

- then
  Mandatory phrase in a conditional clause: if *<log_expr>* then *<expr 1>* ...endif. (cf. also if, else, elsif, endif).

- Theta
  Unary and also binary observable specifier, that as a unary observable gives the angle

```
    process zee =   Z => e1, E1
    process zuu =   Z => u, U
    process zz = e1, E1 => Z, Z
    compile
    integrate (zee) { iterations = 1:100 }
    integrate (zuu) { iterations = 1:100 }
    sqrts = 500 GeV
    integrate (zz) { iterations = 3:5000, 2:5000 }
    unstable Z (zee, zuu)
```

Figure A.1: *SINDARIN input file for unstable particles and inclusive decays.*

between a particle's momentum and the beam axis ($+z$ direction). As a binary observable, it gives the angle enclosed between the momenta of the two particles: `eval Theta [e1], all Theta > 30 degrees [jet, jet]`. (cf. also `eval`, `cuts`, `selection`, `Phi`, `Theta_star`)

- `Theta_star`
  Binary observable specifier, that gives the polar angle enclosed between the momenta of the two particles in the rest frame of the mother particle (momentum sum of the two particle): `eval Theta_star [jet, jet]`. (cf. also `eval`, `cuts`, `selection`, `Theta`)

- `true`
  Constructor stating that a logical expression or variable is true, e.g. `?<log_var> = true`. (cf. also `false`).

- `unpolarized`
  Constructor to force `WHIZARD` to discard polarization of the corresponding particles in the generated events: `unpolarized <prt1> [, <prt2> , ...]`. (cf. also `polarized`, `simulate`, `?polarized_events`)

- `unstable`
  This constructor allows to let final state particles of the hard interaction undergo a subsequent (cascade) decay (in the on-shell approximation). For this the user has to define the list of desired decay channels as `unstable <mother> (<decay1>, <decay2>, ....)`, where `mother` is the mother particle, and the argument is a list of decay channels. Note that – unless the `?auto_decays = true` flag has been set – these decay channels have to be provided by the user as in the example in Fig. A.1. First, the Z decays to electrons and up quarks are generated, then ZZ production at a 500 GeV ILC is called, and then both Zs are decayed according to the probability distribution of the two generated decay matrix elements. This obviously allows also for inclusive decays. (cf. also `stable`, `?auto_decays`)

- weight
  This is a command, `weight = <expr>`, that allows to specify a weight for a process or list of processes. `<expr>` can be any expression that leads to a scalar result, e.g. `weight = 0.2`, `weight = eval Pt [jet]`. (cf. also `rescan`, `alt_setup`, `reweight`)

- write_analysis
  The `write_analysis` statement tells `WHIZARD` to write the analysis setup by the user for the `SINDARIN` input file under consideration. If no `$out_file` is provided, the histogram tables/plot data etc. are written to the default file `whizard_analysis.dat`. Note that the related command `compile_analysis` does the same as `write_analysis` but in addition invokes the `WHIZARD` LATEXroutines for producing postscript or PDF output of the data. (cf. also `$out_file`, `compile_analysis`)

- write_slha
  Demands `WHIZARD` to write out a file in the SUSY Les Houches accord (SLHA) format. (cf. also `read_slha`, `?slha_read_decays`, `?slha_read_input`, `?slha_read_spectrum`)

A.2 Variables

A.2.1 Rebuild Variables

- ?rebuild_events   (default: `false`)
  This logical variable, if set `true` triggers `WHIZARD` to newly create an event sample, even if nothing seems to have changed, including the MD5 checksum. This can be used when manually manipulating some settings. (cf also `?rebuild_grids`, `?rebuild_library`, `?rebuild_phase_space`)

- ?rebuild_grids   (default: `false`)
  The logical variable `?rebuild_grids` forces `WHIZARD` to newly create the VAMP grids when using VAMP as an integration method, even if they are already present. (cf. also `?rebuild_events`, `?rebuild_library`, `?rebuild_phase_space`)

- ?rebuild_library   (default: `false`)
  The logical variable `?rebuild_library = true/false` specifies whether the library(-ies) for the matrix element code for processes is re-generated (incl. possible Makefiles etc.) by the corresponding ME method (e.g. if the process has been changed, but not its name). This can also be set as a command-line option `whizard --rebuild`. The default is `false`, i.e. code is never re-generated if it is present and the MD5 checksum is valid. (cf. also `?recompile_library`, `?rebuild_grids`, `?rebuild_phase_space`)

- ?rebuild_phase_space   (default: `false`)
  This logical variable, if set `true`, triggers recreation of the phase space file by `WHIZARD`(cf. also `?rebuild_events`, `?rebuild_grids`, `?rebuild_library`)

- **?recompile_library**   (default: `false`)
  The logical variable `?recompile_library = true/false` specifies whether the library(-
  ies) for the matrix element code for processes is re-compiled (e.g. if the process code
  has been manually modified by the user). This can also be set as a command-line
  option `whizard --recompile`. The default is `false`, i.e. code is never re-compiled if its
  corresponding object file is present. (cf. also `?rebuild_library`)

A.2.2   Standard Variables

- **accuracy_goal**   (default: `0.00000E+00`)
  Real parameter that allows the user to set a minimal accuracy that should be achieved in
  the Monte-Carlo integration of a certain process. If that goal is reached, grid and weight
  adapation stop, and this result is used for simulation. (cf. also `integrate`, `iterations`,
  `error_goal`, `relative_error_goal`, `error_threshold`)

- **?allow_decays**   (default: `true`)
  Master flag to switch on cascade decays for final state particles as an event transform.
  As a default, it is switched on. (cf. also `?auto_decays`, `auto_decays_multiplicity`,
  `?auto_decays_radiative`, `?decay_rest_frame`)

- **?allow_hadronization**   (default: `true`)
  Master flag to switch on hadronization as an event transform. As a default, it is switched
  on. (cf. also `?ps_ ....`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)

- **?allow_shower**   (default: `true`)
  Master flag to switch on (initial and final state) parton shower, matching/merging as an
  event transform. As a default, it is switched on. (cf. also `?ps_ ....`, `$ps_ ...`, `?mlm_
  ...`, `?hadronization_active`)

- **?alpha_evolve_analytic**   (default: `true`)
  Flag that tells WHIZARD to use analytic running formulae for α instead of a numeric Runge-
  Kutta. (cf. also `alpha_order`, `?alpha_is_fixed`, `alpha_nf`, `alpha_nlep`, `?alpha_from_me`)

- **?alpha_is_fixed**   (default: `true`)
  Flag that tells WHIZARD to use a non-running QED α. Note that this has to be set explicitly
  to `false` if the user wants to use one of the running α options. (cf. also `alpha_order`,
  `alpha_nf`, `alpha_lep`, `?alphas_from_me`

- **alpha_nf**   (default: `-1`)
  Integer parameter that sets the number of active quark flavors for the internal evolution
  for running α in WHIZARD. The default, `-1`, keeps it equal to `alphas_nf alpha_is_fixed`,
  `alphas_order`, `?alpha_from_me`, `?alpha_evolve_analytic`

- **alpha_nlep**   (default: `1`)
  Integer parameter that sets the number of active leptons in the running of α in WHIZARD.

The deffault isone, with only the electron considered massless (cf. also `alpha_is_fixed`, `alpha_nf`, `alpha_order`, `?alpha_from_me`, `?alpha_evolve_analytic`)

- `alpha_order`   (default: `0`)
  Integer parameter that sets the order of the internal evolution for running α in `WHIZARD`: the default, `0`, is LO running, `1` is NLO. (cf. also `alpha_is_fixed`, `alpha_nf`, `alphas_lep`, `?alpha_from_me`)

- `alpha_power`   (default: `2`)
  Fixes the electroweak coupling powers used by BLHA matrix element generators. Setting these values is necessary for the correct generation of OLP-files. Having inconsistent values yields to error messages by the corresponding OLP-providers.

- `?alphas_from_lambda_qcd`   (default: `false`)
  Flag that tells `WHIZARD` to use its internal running α_s from $\alpha_s(\Lambda_{QCD})$. Note that in that case `?alphas_is_fixed` has to be set explicitly to `false`. (cf. also `alphas_order`, `?alphas_is_fixed`, `?alphas_from_lhapdf`, `alphas_nf`, `?alphas_from_pdf_builtin`, `?alphas_from_mz`, `lambda_qcd`)

- `?alphas_from_lhapdf`   (default: `false`)
  Flag that tells `WHIZARD` to use a running α_s from the `LHAPDF` library (which has to be correctly linked). Note that `?alphas_is_fixed` has to be set explicitly to `false`. (cf. also `alphas_order`, `?alphas_is_fixed`, `?alphas_from_pdf_builtin`, `alphas_nf`, `?alphas_from_mz`, `?alphas_from_lambda_qcd`, `lambda_qcd`)

- `?alphas_from_mz`   (default: `false`)
  Flag that tells `WHIZARD` to use its internal running α_s from $\alpha_s(M_Z)$. Note that in that case `?alphas_is_fixed` has to be set explicitly to `false`. (cf. also `alphas_order`, `?alphas_is_fixed`, `?alphas_from_lhapdf`, `alphas_nf`, `?alphas_from_pdf_builtin`, `?alphas_from_lambda_qcd`, `lambda_qcd`)

- `?alphas_from_pdf_builtin`   (default: `false`)
  Flag that tells `WHIZARD` to use a running α_s from the internal PDFs. Note that in that case `?alphas_is_fixed` has to be set explicitly to `false`. (cf. also `alphas_order`, `?alphas_is_fixed`, `?alphas_from_lhapdf`, `alphas_nf`, `?alphas_from_mz`, `?alphas_from_lambda_qcd`, `lambda_qcd`)

- `?alphas_is_fixed`   (default: `true`)
  Flag that tells `WHIZARD` to use a non-running QCD α_s. Note that this has to be set explicitly to `false` if the user wants to use one of the running α_s options. (cf. also `alphas_order`, `?alphas_from_lhapdf`, `?alphas_from_pdf_builtin`, `alphas_nf`, `?alphas_from_mz`, `?alphas_from_lambda_qcd`, `lambda_qcd`)

- `alphas_nf`   (default: `5`)
  Integer parameter that sets the number of active quark flavors for the internal evolution for running α_s in `WHIZARD`. (cf. also `alphas_is_fixed`, `?alphas_from_lhapdf`,

?alphas_from_pdf_builtin, alphas_order, ?alphas_from_mz,
?alphas_from_lambda_qcd, lambda_qcd)

- alphas_order   (default: 0)
  Integer parameter that sets the order of the internal evolution for running α_s in WHIZARD:
  the default, 0, is LO running, 1 is NLO, 2 is NNLO. (cf. also alphas_is_fixed,
  ?alphas_from_lhapdf, ?alphas_from_pdf_builtin, alphas_nf, ?alphas_from_mz,
  ?alphas_from_lambda_qcd, lambda_qcd)

- alphas_power   (default: 0)
  Fixes the strong coupling powers used by BLHA matrix element generators. Setting these
  values is necessary for the correct generation of OLP-files. Having inconsistent values
  yields to error messages by the corresponding OLP-providers.

- ?analysis_file_only   (default: false)
  Allows to specify that only LATEX files for WHIZARD's graphical analysis are written out,
  but not processed. (cf. compile_analysis, write_analysis)

- antikt_algorithm   (fixed value: 2)
  Specifies a jet algorithm for the ($\rightarrow$) jet_algorithm command, used in the ($\rightarrow$) cluster
  subevent function. At the moment only available for the interfaced external FastJet pack-
  age. (cf. also kt_algorithm, cambridge_[for_passive_]algorithm, plugin_algorithm,
  genkt_[for_passive_]algorithm, ee_[gen]kt_algorithm, jet_r)

- ?auto_decays   (default: false)
  Flag, particularly as optional argument of the ($\rightarrow$) unstable command, that tells WHIZARD
  to automatically determine the decays of that particle up to the final state multplicity
  ($\rightarrow$) auto_decays_multiplicity. Depending on the flag ($\rightarrow$) ?auto_decays_radiative,
  radiative decays will be taken into account or not. (cf. also unstable, ?isotropic_decay,
  ?diagonal_decay)

- auto_decays_multiplicity   (default: 2)
  Integer parameter, that sets – for the ($\rightarrow$) ?auto_decays option to let WHIZARD automati-
  cally determine the decays of a particle set as ($\rightarrow$) unstable – the maximal final state
  multiplicity that is taken into account. The default is 2. The flag ?auto_decays_radiative
  decides whether radiative decays are taken into account. (cf. also unstable, ?auto_decays)

- ?auto_decays_radiative   (default: false)
  If WHIZARD's automatic detection of decay channels are switched on ($\rightarrow$?auto_decays
  for the ($\rightarrow$) unstable command, this flags decides whether radiative decays (e.g. con-
  taining additional photon(s)/gluon(s)) are taken into account or not. (cf. also unstable,
  auto_decays_multiplicity)

- $beam_events_file
  String variable that allows to set the name of the external file from which a beamstrahlung's

spectrum for lepton colliders as pairs of energy fractions is read in. (cf. also `beam_events`, `?beam_events_warn_eof`)

- `?beam_events_warn_eof`   (default: `true`)
  Flag that tells `WHIZARD` to issue a warning when in a simulation the end of an external file for beamstrahlung's spectra for lepton colliders are reached, and energy fractions from the beginning of the file are reused. (cf. also `beam_events`, `$beam_events_file`)

- `$blha_ew_scheme`   (default: `"alpha_internal"`)
  String variable that transfers the electroweak renormalization scheme via BLHA to the one-loop provider. Possible values are `GF` or `Gmu` for the G_μ scheme, `alpha_internal` (default, G_μ scheme, but value of α_S calculated internally by `WHIZARD`), `alpha_mz` and `alpha_0` (or `alpha_thompson`) for different schemes with α as input.

- `blha_top_yukawa`   (default: `-1.00000E+00`)
  If this value is set, the given value will be used as the top Yukawa coupling instead of the top mass. Note that having different values for y_t and m_t must be supported by your OLP-library and yield errors if this is not the case.

- `$born_me_method`   (default: `""`)
  This string variable specifies the method for the matrix elements to be used in the evaluation of the Born part of the NLO computation. The default is the empty string, i.e. the `$method` being the intrinsic `O'Mega` matrix element generator (`"omega"`), other options are: `"ovm"`, `"unit_test"`, `"template"`, `"template_unity"`, `"threshold"`, `"gosam"`, `"openloops"`. Note that this option is inoperative if no NLO calculation is specified in the process definition. If you want ot use different matrix element methods in a LO computation, use the usual `method` command. (cf. also `$correlation_me_method`, `$dglap_me_method`, `$loop_me_method` and `$real_tree_me_method`.)

- `cambridge_algorithm`   (fixed value: `1`)
  Specifies a jet algorithm for the ($\rightarrow$) `jet_algorithm` command, used in the ($\rightarrow$) `cluster` subevent function. At the moment only available for the interfaced external `FastJet` package. (cf. also `kt_algorithm`, `cambridge_for_passive_algorithm`, `plugin_algorithm`, `genkt_[for_passive_]algorithm`, `ee_[gen]kt_algorithm`, `jet_r`)

- `cambridge_for_passive_algorithm`   (fixed value: `11`)
  Specifies a jet algorithm for the ($\rightarrow$) `jet_algorithm` command, used in the ($\rightarrow$) `cluster` subevent function. At the moment only available for the interfaced external `FastJet` package. (cf. also `kt_algorithm`, `cambridge_algorithm`, `plugin_algorithm`, `genkt_[for_passive_]algorithm`, `ee_[gen]kt_algorithm`, `jet_r`)

- `channel_weights_power`   (default: `2.50000E-01`)
  Real parameter that allows to vary the exponent of the channel weights for the `VAMP` integrator.

- **?check_event_file**   (default: `true`)
  Setting this to false turns off all sanity checks when reading a raw event file with previously generated events. Use this at your own risk; the program may return wrong results or crash if data do not match. (cf. also **?check_grid_file**, **?check_phs_file**)

- **?check_event_weights_against_xsection**   (default: `false`)
  Activates an internal recording of event weights when unweighted events are generated. At the end of the simulation, the mean value of the weights and its standard deviation are displayed. This allows to cross-check event generation and integration, because the value displayed must be equal to the integration result.

- **?check_grid_file**   (default: `true`)
  Setting this to false turns off all sanity checks when reading a grid file with previous integration data. Use this at your own risk; the program may return wrong results or crash if data do not match. (cf. also **?check_event_file**, **?check_phs_file**)

- **?check_phs_file**   (default: `true`)
  Setting this to false turns off all sanity checks when reading a previously generated phase-space configuration file. Use this at your own risk; the program may return wrong results or crash if data do not match. (cf. also **?check_event_file**, **?check_grid_file**)

- **checkpoint**   (default: `0`)
  Setting this integer variable to a positive integer n instructs simulate to print out a progress summary every n events.

- **$circe1_acc**   (default: `"SBAND"`)
  String variable that specifies the accelerator type for the `CIRCE1` structure function for lepton-collider beamstrahlung. (**?circe1_photons**, **?circe1_photon2**, **circe1_sqrts**, **?circe1_generate**, **?circe1_map**, **circe1_eps**, **circe1_mapping_slope**, **circe1_ver**, **circe1_rev**, **circe1_chat**, **?circe1_with_radiation**)

- **circe1_chat**   (default: `0`)
  Chattiness of the `CIRCE1` structure function for lepton-collider beamstrahlung. The higher the integer value, the more information will be given out by the `CIRCE1` package. (**?circe1_photons**, **?circe1_photon2**, **circe1_sqrts**, **?circe1_generate**, **?circe1_map**, **circe1_eps**, **circe1_mapping_slope**, **circe1_ver**, **circe1_rev**, **$circe1_acc**, **?circe1_with_radiation**)

- **circe1_eps**   (default: `1.00000E-05`)
  Real parameter, that takes care of the mapping of the peak in the lepton collider beamstrahlung structure function spectrum of `CIRCE1`. (cf. also **circe1**, **?circe1_photons**, **?circe1_photon2**, **circe1_sqrts**, **?circe1_generate**, **?circe1_map**, **circe1_eps**, **circe1_mapping_slope**, **circe1_ver**, **circe1_rev**, **$circe1_acc**, **circe1_chat**, **?circe1_with_radiation**)

- **?circe1_generate** (default: `true`)
  Flag that determines whether the `CIRCE1` structure function for lepton collider beamstrahlung uses the generator mode for the spectrum, or a pre-defined (semi-)analytical parameterization. Default is the generator mode. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_map`, `circe1_mapping_slope`, `circe1_eps`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **?circe1_map** (default: `true`)
  Flag that determines whether the `CIRCE1` structure function for lepton collider beamstrahlung uses special mappings for s-channel resonances. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `circe1_mapping_slope`, `circe1_eps`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **circe1_mapping_slope** (default: 2.00000E+00)
  Real parameter that allows to vary the slope of the mapping function for the `CIRCE1` structure function for lepton collider beamstrahlung from the default value 2.. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **?circe1_photon1** (default: `false`)
  Flag to tell `WHIZARD` to use the photon of the `CIRCE1` beamstrahlung structure function as initiator for the hard scattering process in the first beam. (cf. also `circe1`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **?circe1_photon2** (default: `false`)
  Flag to tell `WHIZARD` to use the photon of the `CIRCE1` beamstrahlung structure function as initiator for the hard scattering process in the second beam. (cf. also `circe1`, `?circe1_photon1`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **circe1_rev** (default: 0)
  Integer parameter that sets the internal revision number of the `CIRCE1` structure function for lepton-collider beamstrahlung. The default 0 translates always into the most recent version; older versions have to be accessed through the explicit revision date. For more details cf. the `CIRCE1`manual. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_sqrts`, `circe1_ver`, `$circe1_acc`, `circe1_chat`, `?circe1_with_radiation`)

- **circe1_sqrts**
  Real parameter that allows to set the value of the collider energy for the lepton collider beamstrahlung structure function `CIRCE1`. If not set, $\sqrt{s}$ is taken. (cf. also `circe1`,

?circe1_photon1, ?circe1_photon2, ?circe1_generate, ?circe1_map, circe1_eps, circe1_mapping_slope, circe1_ver, circe1_rev, $circe1_acc, circe1_chat, ?circe1_with_radiation)

- circe1_ver   (default: 0)
  Integer parameter that sets the internal versioning number of the CIRCE1 structure function for lepton-collider beamstrahlung. It has to be set by the user explicitly, it takes values from one to ten. (cf. also circe1, ?circe1_photon1, ?circe1_photon2, ?circe1_generate, ?circe1_map, circe1_eps, circe1_mapping_slope, circe1_sqrts, circe1_rev, $circe1_acc, circe1_chat, ?circe1_with_radiation)

- ?circe1_with_radiation   (default: false)
  This logical decides whether the additional photon or electron ("beam remnant") will be considered in the event record or not. (?circe1_photons, ?circe1_photon2, circe1_sqrts, ?circe1_generate, ?circe1_map, circe1_eps, circe1_mapping_slope, circe1_ver, circe1_rev, $circe1_acc)

- $circe2_design   (default: "*")
  String variable that sets the collider design for the CIRCE2 structure function for photon collider spectra. (cf. also circe2, $circe2_file, ?circe2_polarized)

- $circe2_file
  String variable by which the corresponding photon collider spectrum for the CIRCE2 structure function can be selected. (cf. also circe2, ?circe2_polarized, $circe2_design)

- ?circe2_polarized   (default: true)
  Flag whether the photon spectra from the CIRCE2 structure function for lepton colliders should be treated polarized. (cf. also circe2, $circe2_file, $circe2_design)

- ?ckkw_matching   (default: false)
  Master flag that switches on the CKKW(-L) (LO) matching between hard scattering matrix elements and QCD parton showers. Note that this is not yet (completely) implemented in WHIZARD. (cf. also ?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ...)

- ?colorize_subevt   (default: false)
  Flag that enables color-index tracking in the subevent (subevt) objects that are used for internal event analysis.

- ?combined_nlo_integration   (default: false)
  When this option is set to true, the NLO integration will not be performed in the separate components, but instead the sum of all components will be integrated directly. When fixed-order NLO events are requested, this integration mode is possible, but not necessary. However, it is necessary for POWHEG events.

- $compile_workspace
  If set, create process source code and process-driver library code in a subdirectory with

this name. If non-existent, the directory will be created. (cf. also `$job_id`, `$run_id`, `$integrate_workspace`)

- `$correlation_me_method` (default: `""`)
  This string variable specifies the method for the matrix elements to be used in the evaluation of the color (and helicity) correlated part of the NLO computation. The default is the same as the `$method`, i.e. the intrinsic O'Mega matrix element generator (`"omega"`), other options are: `"ovm"`, `"unit_test"`, `"template"`, `"template_unity"`, `"threshold"`, `"gosam"`, `"openloops"`. (cf. also `$born_me_method`, `$dglap_me_method`, `$loop_me_method` and
  `$real_tree_me_method`.)

- `$dalitz_plot` (default: `""`)
  This string variable has two purposes: when different from the empty string, it switches on generation of the Dalitz plot file (ASCII tables) for the real emitters. The string variable itself provides the file name.

- `?debug_decay` (default: `true`)
  Flag that decides whether decay information will be displayed in the ASCII debug event format ($\rightarrow$) debug. (cf. also `sample_format`, `$sample`, `$debug_extension`, `?debug_process`, `?debug_transforms`, `?debug_verbose`)

- `$debug_extension` (default: `"debug"`)
  String variable that allows via `$debug_extension = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in a long verbose format with debugging information are written. If not set, the default file name and suffix is *<process_name>*.debug. (cf. also `sample_format`, `$sample`, `?debug_process`, `?debug_transforms`, `?debug_decay`, `?debug_verbose`)

- `?debug_process` (default: `true`)
  Flag that decides whether process information will be displayed in the ASCII debug event format ($\rightarrow$) debug. (cf. also `sample_format`, `$sample`, `$debug_extension`, `?debug_decay`, `?debug_transforms`, `?debug_verbose`)

- `?debug_transforms` (default: `true`)
  Flag that decides whether information about event transforms will be displayed in the ASCII debug event format ($\rightarrow$) debug. (cf. also `sample_format`, `$sample`, `?debug_decay`, `$debug_extension`, `?debug_process`, `?debug_verbose`)

- `?debug_verbose` (default: `true`)
  Flag that decides whether extensive verbose information will be included in the ASCII debug event format ($\rightarrow$) debug. (cf. also `sample_format`, `$sample`, `$debug_extension`, `?debug_decay`, `?debug_transforms`, `?debug_process`)

- `decay_helicity`
  If this parameter is given an integer value, any particle decay triggered by a subsequent

unstable declaration will receive a projection on the given helicity state for the un-
stable particle. (cf. also unstable, ?isotropic_decay, ?diagonal_decay. The latter
parameters, if true, take precdence over any ?decay_helicity setting.)

- ?decay_rest_frame   (default: false)
  Flag that allows to force a particle decay to be simulated in its rest frame. This simplifies
  the calculation for decays as stand-alone processes, but makes the process unsuitable for
  use in a decay chain.

- $description   (default: "")
  String variable that allows to specify a description text for the analysis, $description
  = "<LaTeX analysis descr.>". This line appears below the title of a corresponding
  analysis, on top of the respective plot. (cf. also analysis, n_bins, ?normalize_bins,
  $obs_unit, $x_label, $y_label, ?y_log, ?x_log, graph_width_mm, graph_height_mm,
  x_min, x_max, y_min, y_max, $gmlcode_bg, $gmlcode_fg, ?draw_base, ?draw_histogram,
  ?fill_curve, ?draw_piecewise, ?draw_curve, ?draw_errors, $symbol, ?draw_symbols,
  $fill_options, $draw_options, $err_options)

- $dglap_me_method   (default: "")
  This string variable specifies the method for the matrix elements to be used in the evaluation
  of the DGLAP remnants of the NLO computation. The default is the same as $method, i.e.
  the O'Mega matrix element generator ("omega"), other options are: "ovm", "unit_test",
  "template", "template_unity", "threshold", "gosam", "openloops". (cf. also
  $born_me_method, $correlation_me_method, $loop_me_method and $real_tree_me_method.)

- ?diagonal_decay   (default: false)
  Flag that – in case of using factorized production and decays using the ($\rightarrow$) unstable
  command – tells WHIZARD instead of full spin correlations to take only the diagonal
  entries in the spin-density matrix (i.e. classical spin correlations). (cf. also unstable,
  ?auto_decays, decay_helicity, ?isotropic_decay)

- ?disable_subtraction   (default: false)
  Disables the subtraction of soft and collinear divergences from the real matrix element.

- ?draw_base
  Settings for WHIZARD's internal graphics output: flag that tells WHIZARD to insert a base
  statement in the analysis code to calculate the plot data from a data set. (cf. also
  ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label,
  graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max,
  $gmlcode_fg, $gmlcode_bg, ?draw_curve, ?draw_piecewise, ?fill_curve, $symbol,
  ?draw_histogram, ?draw_errors, ?draw_symbols, $fill_options, $draw_options,
  $err_options)

- ?draw_curve
  Settings for WHIZARD's internal graphics output: flag that tells WHIZARD to either plot

data as a continuous line or as a histogram (if → ?draw_histogram is set true). (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_histogram, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- ?draw_errors
  Settings for WHIZARD's internal graphics output: flag that determines whether error bars should be drawn or not. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_histogram, ?draw_curve, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- ?draw_histogram
  Settings for WHIZARD's internal graphics output: flag that tells WHIZARD to either plot data as a histogram or as a continuous line (if → ?draw_curve is set true). (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- $draw_options
  Settings for WHIZARD's internal graphics output: $draw_options = "*<LaTeX_code>*" is a string variable that allows to set specific drawing options for plots and histograms. For more details see the gamelan manual. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_histogram, ?draw_errors, ?draw_symbols, $fill_options, ?draw_histogram, $err_options, $symbol)

- ?draw_piecewise
  Settings for WHIZARD's internal graphics output: flag that tells WHIZARD to data from a data set piecewise, i.e. histogram style. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_curve, ?draw_base, ?fill_curve, $symbol, ?draw_histogram, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options)

- ?draw_symbols
  Settings for WHIZARD's internal graphics output: flag that determines whether particular symbols (specified by → $symbol = "*<LaTeX_code>*") should be used for plotting data points (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, $gmlcode_bg, ?draw_base, ?draw_piecewise, ?fill_curve,

?draw_histogram, ?draw_curve, ?draw_errors, $fill_options, $draw_options, $err_options, $symbol)

- ?dump_compressed   (default: false)
  Flag that, if set to true, issues a very compressed and clear version of the dump (→) event format. (cf. also sample_format, $sample, dump, $dump_extension, ?dump_screen, ?dump_summary, ?dump_weights)

- $dump_extension   (default: "pset.dat")
  String variable that allows via $dump_extension = "*<suffix>*" to specify the suffix for the file name.suffix to which events in WHIZARD's internal particle set format are written. If not set, the default file name and suffix is *<process_name>*.pset.dat. (cf. also sample_format, $sample, dump, ?dump_compressed, ?dump_screen, ?dump_summary, ?dump_weights)

- ?dump_screen   (default: false)
  Flag that, if set to true, outputs events for the dump (→) event format on screen instead of to a file. (cf. also sample_format, $sample, dump, ?dump_compressed, $dump_extension, ?dump_summary, ?dump_weights)

- ?dump_summary   (default: false)
  Flag that, if set to true, includes a summary with momentum sums for incoming and outgoing particles as well as for beam remnants in the dump (→) event format. (cf. also sample_format, $sample, dump, ?dump_compressed, $dump_extension, ?dump_screen, ?dump_weights)

- ?dump_weights   (default: false)
  Flag that, if set to true, includes cross sections, weights and excess in the dump (→) event format. (cf. also sample_format, $sample, dump, ?dump_compressed, $dump_extension, ?dump_screen, ?dump_summary)

- ee_genkt_algorithm   (fixed value: 53)
  Specifies a jet algorithm for the (→) jet_algorithm command, used in the (→) cluster subevent function. At the moment only available for the interfaced external FastJet package. (cf. also kt_algorithm, cambridge_[for_passive_]algorithm, plugin_algorithm, genkt_[for_passive_]algorithm, ee_kt_algorithm, jet_r), jet_p)

- ee_kt_algorithm   (fixed value: 50)
  Specifies a jet algorithm for the (→) jet_algorithm command, used in the (→) cluster subevent function. At the moment only available for the interfaced external FastJet package. (cf. also kt_algorithm, cambridge_[for_passive_]algorithm, plugin_algorithm, genkt_[for_passive_]algorithm, ee_genkt_algorithm, jet_r)

- ellis_sexton_scale   (default: -1.00000E+00)
  Real positive paramter for the Ellis-Sexton scale$Q$ used both in the finite one-loop contribution provided bythe OLP and in the virtual counter terms. The NLO cross section

isindependent of $\mathcal{Q}$. Therefore, this allows for debugging ofthe implemention of the virtual counter terms. As the default$\mathcal{Q} = \mu_{\mathrm{R}}$ is chosen. So far, setting this parameteronly works for OpenLoops2, otherwise the default behaviour is invoked.

- `?energy_scan_normalize` (default: `false`)
  Normalization flag for the energy scan structure function: if set the total cross section is normalized to unity. (cf. also `energy_scan`)

- `epa_alpha` (default: 0.00000E+00)
  For the equivalent photon approximation (EPA), this real parameter sets the value of α_{em} used in the structure function. If not set, it is taken from the parameter set of the physics model in use (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_e_max`, `epa_q_min`, `?epa_recoil`, `?epa_keep_energy`, `$epa_mode`, `?epa_handler`, `$epa_handler_mode`)

- `?epa_handler` (default: `false`)
  Activate EPA handler for event generation (no effect on integration). Requires `epa_recoil = false`

- `$epa_handler_mode` (default: `"trivial"`)
  Operation mode for the EPA event handler. Allowed values: `trivial` (no effect), `recoil` (recoil kinematics with two beams)

- `?epa_keep_energy` (default: `false`)
  As the splitting kinematics for the EPA structure function violates Lorentz invariance when the recoil is switched on, this flag forces energy conservation when set to true, otherwise violating energy conservation. (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_alpha`, `epa_q_min`, `?epa_recoil`, `$epa_mode`, `?epa_handler`, `$epa_handler_mode`)

- `epa_mass` (default: 0.00000E+00)
  This real parameter allows to set by hand the mass of the incoming particle for the equivalent-photon approximation (EPA). If not set, the mass for the initial beam particle is taken from the model in use. (cf. also `epa`, `epa_x_min`, `epa_e_max`, `epa_alpha`, `epa_q_min`, `?epa_recoil`, `?epa_keep_energy`, `$epa_mode`. `?epa_handler`, `$epa_handler_mode`)

- `$epa_mode` (default: `"default"`)
  For the equivalent photon approximation (EPA), this string variable defines the mode, i.e. the explicit formula for the EPA distribution. For more details cf. the manual. Possible are `default` (`Budnev_617`), `Budnev_616e`, `log_power`, `log_simple`, and `log`. (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_e_max`, `epa_q_min`, `?epa_recoil`, `?epa_keep_energy`, `?epa_handler`, `$epa_handler_mode`)

- `epa_q_max` (default: 0.00000E+00)
  This real parameter allows to set the upper energy cutoff for the equivalent-photon approximation (EPA). If not set, `WHIZARD` simply takes the collider energy, $\sqrt{s}$. (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_alpha`, `epa_q_min`, `?epa_recoil`, `$epa_mode`, `?epa_keep_energy`, `?epa_handler`, `$epa_handler_mode`)

- `epa_q_min`   (default: 0.00000E+00)
  In the equivalent-photon approximation (EPA), this real parameters sets the minimal
  value for the transferred momentum. Either this parameter or the mass of the beam
  particle has to be non-zero. (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_alpha`, `epa_q_max`,
  `?epa_recoil`, `?epa_keep_energy`, `$epa_mode`, `?epa_handler`, `$epa_handler_mode`)

- `?epa_recoil`   (default: `false`)
  Flag to switch on recoil, i.e. a non-vanishing p_T-kick for the equivalent-photon approx-
  imation (EPA). (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_alpha`, `epa_e_max`, `epa_q_min`,
  `?epa_keep_energy`, `$epa_mode`, `?epa_handler`, `$epa_handler_mode`)

- `epa_x_min`   (default: 0.00000E+00)
  Real parameter that sets the lower cutoff for the energy fraction in the splitting for the
  equivalent-photon approximation (EPA). This parameter has to be set by the user to a non-
  zero value smaller than one. (cf. also `epa`, `epa_e_max`, `epa_mass`, `epa_alpha`, `epa_q_min`,
  `?epa_recoil`, `?epa_keep_energy`, `$epa_mode`, `?epa_handler`, `$epa_handler_mode`)

- `$err_options`
  Settings for `WHIZARD`'s internal graphics output: `$err_options = "<LaTeX_code>"` is
  a string variable that allows to set specific drawing options for errors in plots and
  histograms. For more details see the `gamelan` manual. (cf. also `?normalize_bins`,
  `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_width_mm`,
  `graph_height_mm`, `?y_log`, `?x_log`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_fg`, `$gmlcode_bg`,
  `?draw_base`, `?draw_piecewise`, `?fill_curve`, `?draw_histogram`, `?draw_errors`,
  `?draw_symbols`, `$fill_options`, `?draw_histogram`, `$draw_options`, `$symbol`)

- `error_goal`   (default: 0.00000E+00)
  Real parameter that allows the user to set a minimal absolute error that should be achieved
  in the Monte-Carlo integration of a certain process. If that goal is reached, grid and weight
  adapation stop, and this result is used for simulation. (cf. also `integrate`, `iterations`,
  `accuracy_goal`, `relative_error_goal`, `error_threshold`)

- `error_threshold`   (default: 0.00000E+00)
  The real parameter `error_threshold = <num>` declares that any error value (in absolute
  numbers) smaller than `<num>` is to be considered zero. The units are `fb` for scatterings
  and `GeV` for decays. (cf. also `integrate`, `iterations`, `accuracy_goal`, `error_goal`,
  `relative_error_goal`)

- `event_callback_interval`   (default: 0)
  Setting this integer variable to a positive integer n instructs simulate to print out a progress
  summary every n events.

- `$event_file_version`   (default: `""`)
  String variable that allows to set the format version of the `WHIZARD` internal binary event
  format.

- `event_index_offset`   (default: `0`)
  The value `event_index_offset = <num>` initializes the event counter for a subsequent event sample. By default (value 0), the first event gets index value 1, incrementing by one for each generated event within a sample. The event counter is initialized again for each new sample (i.e., `integrate` command). If events are read from file, and the event file format supports event numbering, the event numbers will be taken from file instead, and the value of `event_index_offset` has no effect. (cf. `luminosity`, `$sample`, `sample_format`, `?unweighted`, `n_events`)

- `?ewa_keep_energy`   (default: `false`)
  As the splitting kinematics for the equivalent W approximation (EWA) violates Lorentz invariance when the recoil is switched on, this flag forces energy conservation when set to true, otherwise violating energy conservation. (cf. also `ewa`, `ewa_x_min`, `ewa_pt_max`, `ewa_mass`, `?ewa_recoil`)

- `ewa_mass`   (default: `0.00000E+00`)
  This real parameter allows to set by hand the mass of the incoming particle for the equivalent W approximation (EWA). If not set, the mass for the initial beam particle is taken from the model in use. (cf. also `ewa`, `ewa_x_min`, `ewa_pt_max`, `?ewa_keep_energy`, `?ewa_recoil`)

- `ewa_pt_max`   (default: `0.00000E+00`)
  This real parameter allows to set the upper p_T cutoff for the equivalent W approximation (EWA). If not set, `WHIZARD` simply takes the collider energy, $\sqrt{s}$. (cf. also `ewa`, `ewa_x_min`, `ewa_mass`, `?ewa_keep_energy`, `?ewa_recoil`)

- `?ewa_recoil`   (default: `false`)
  For the equivalent W approximation (EWA), this flag switches on recoil, i.e. non-collinear splitting. (cf. also `ewa`, `ewa_x_min`, `ewa_pt_max`, `ewa_mass`, `?ewa_keep_energy`)

- `ewa_x_min`   (default: `0.00000E+00`)
  Real parameter that sets the lower cutoff for the energy fraction in the splitting for the equivalent W approximation (EWA). This parameter has to be set by the user to a non-zero value smaller than one. (cf. also `ewa`, `ewa_pt_max`, `ewa_mass`, `?ewa_keep_energy`, `?ewa_recoil`)

- `$exclude_gauge_splittings`   (default: `"c:b:t:e2:e3"`)
  String variable that allows via `$exclude_gauge_splittings = "<prt1>:<prt2>:..."` to exclude fermion flavors from gluon/photon splitting into fermion pairs beyond LO. For example `$exclude_gauge_splittings = "c:s:b:t"` would lead to `gl => u U` and `gl => d D` as possible splittings in QCD. It is important to keep in mind that only the particles listed in the string are excluded! In QED this string would additionally allow for all splittings into lepton pairs `A => l L`. Therefore, once set the variable acts as a replacement of the default value, not as an addition! Note: `"<prt>"` can be both particle or antiparticle. It will always exclude the corresponding fermion pair. An empty string

allows for all fermion flavors to take part in the splitting! Also, particles included in an `alias` are not excluded by `$exclude_gauge_splittings`!

- `$extension_ascii_long`   (default: `"long.evt"`)
  String variable that allows via `$extension_ascii_long = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the so called long variant of the `WHIZARD` version 1 style HEPEVT ASCII format are written. If not set, the default file name and suffix is `<process_name>.long.evt`. (cf. also `sample_format`, `$sample`)

- `$extension_ascii_short`   (default: `"short.evt"`)
  String variable that allows via `$extension_ascii_short = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the so called short variant of the `WHIZARD` version 1 style HEPEVT ASCII format are written. If not set, the default file name and suffix is `<process_name>.short.evt`. (cf. also `sample_format`, `$sample`)

- `$extension_athena`   (default: `"athena.evt"`)
  String variable that allows via `$extension_athena = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the ATHENA file format are written. If not set, the default file name and suffix is `<process_name>.athena.evt`. (cf. also `sample_format`, `$sample`)

- `$extension_default`   (default: `"evt"`)
  String variable that allows via `$extension_default = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in a the standard `WHIZARD` verbose ASCII format are written. If not set, the default file name and suffix is `<process_name>.evt`. (cf. also `sample_format`, `$sample`)

- `$extension_hepevt`   (default: `"hepevt"`)
  String variable that allows via `$extension_hepevt = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the `WHIZARD` version 1 style HEPEVT ASCII format are written. If not set, the default file name and suffix is `<process_name>.hepevt`. (cf. also `sample_format`, `$sample`)

- `$extension_hepevt_verb`   (default: `"hepevt.verb"`)
  String variable that allows via `$extension_hepevt_verb = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the `WHIZARD` version 1 style extended or verbose HEPEVT ASCII format are written. If not set, the default file name and suffix is `<process_name>.hepevt.verb`. (cf. also `sample_format`, `$sample`)

- `$extension_hepmc`   (default: `"hepmc"`)
  String variable that allows via `$extension_hepmc = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the HepMC format are written. If not set, the default file name and suffix is `<process_name>.hepmc`. (cf. also `sample_format`, `$sample`)

- $extension_lcio   (default: "slcio")
  String variable that allows via $extension_lcio = "*<suffix>*" to specify the suffix for
  the file name.suffix to which events in the LCIO format are written. If not set, the default
  file name and suffix is *<process_name>*.slcio. (cf. also sample_format, $sample)

- $extension_lha   (default: "lha")
  String variable that allows via $extension_lha = "*<suffix>*" to specify the suffix for
  the file name.suffix to which events in the (deprecated) LHA format are written. If not
  set, the default file name and suffix is *<process_name>*.lha. (cf. also sample_format,
  $sample)

- $extension_lha_verb   (default: "lha.verb")
  String variable that allows via $extension_lha_verb = "*<suffix>*" to specify the suffix
  for the file name.suffix to which events in the (deprecated) extended or verbose LHA for-
  mat are written. If not set, the default file name and suffix is *<process_name>*.lha.verb.
  (cf. also sample_format, $sample)

- $extension_mokka   (default: "mokka.evt")
  String variable that allows via $extension_mokka = "*<suffix>*" to specify the suffix
  for the file name.suffix to which events in the MOKKA format are written. If not set,
  the default file name and suffix is *<process_name>*.mokka.evt. (cf. also sample_format,
  $sample)

- $extension_raw   (default: "evx")
  String variable that allows via $extension_raw = "*<suffix>*" to specify the suffix for
  the file name.suffix to which events in WHIZARD's internal format are written. If not
  set, the default file name and suffix is *<process_name>*.evx. (cf. also sample_format,
  $sample)

- $extension_stdhep   (default: "hep")
  String variable that allows via $extension_stdhep = "*<suffix>*" to specify the suffix
  for the file name.suffix to which events in the StdHEP format via the HEPEVT common
  block are written. If not set, the default file name and suffix is *<process_name>*.hep. (cf.
  also sample_format, $sample)

- $extension_stdhep_ev4   (default: "ev4.hep")
  String variable that allows via $extension_stdhep_ev4 = "*<suffix>*" to specify the
  suffix for the file name.suffix to which events in the StdHEP format via the HEP-
  EVT/HEPEV4 common blocks are written. *<process_name>*.up.hep is the default file
  name and suffix, if this variable not set. (cf. also sample_format, $sample)

- $extension_stdhep_up   (default: "up.hep")
  String variable that allows via $extension_stdhep_up = "*<suffix>*" to specify the suffix
  for the file name.suffix to which events in the StdHEP format via the HEPRUP/HEPEUP
  common blocks are written. *<process_name>*.up.hep is the default file name and suffix,
  if this variable not set. (cf. also sample_format, $sample)

- `?fatal_beam_decay`   (default: `true`)
  Logical variable that let the user decide whether the possibility of a beam decay is treated
  as a fatal error or only as a warning. An example is a process $bt \to X$, where the bottom
  quark as an inital state particle appears as a possible decay product of the second incoming
  particle, the top quark. This might trigger inconsistencies or instabilities in the phase
  space set-up.

- `$fc`   (default: `"Fortran-compiler"`)
  This string variable gives the `Fortran` compiler used within `WHIZARD`. It can only be
  accessed, not set by the user. (cf. also `$fcflags`, `$fclibs`)

- `$fcflags`   (default: `"Fortran-flags"`)
  This string variable gives the compiler flags for the `Fortran` compiler used within `WHIZARD`.
  It can only be accessed, not set by the user. (cf. also `$fc`, `$fclibs`)

- `$fclibs`   (default: `"Fortran-libs"`)
  This string variable gives the linked libraries for the `Fortran` compiler used within `WHIZARD`.
  It can only be accessed, not set by the user. (cf. also `$fc`, `$fcflags`)

- `?fill_curve`
  Settings for `WHIZARD`'s internal graphics output: flag that tells `WHIZARD` to fill data curves
  (e.g. as a histogram). The style can be set with $\to$ `$fill_options = "<LaTeX_code>"`.
  (cf. also `?normalize_bins`, `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`,
  `$y_label`, `graph_width_mm`, `graph_height_mm`, `?y_log`, `?x_log`, `x_min`, `x_max`, `y_min`, `y_max`,
  `$gmlcode_fg`, `$gmlcode_bg`, `?draw_base`, `?draw_piecewise`, `?draw_curve`, `?draw_histogram`,
  `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- `$fill_options`
  Settings for `WHIZARD`'s internal graphics output: `$fill_options = "<LaTeX_code>"` is a
  string variable that allows to set fill options when plotting data as filled curves with the $\to$
  `?fill_curve` flag. For more details see the `gamelan` manual. (cf. also `?normalize_bins`,
  `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_width_mm`,
  `graph_height_mm`, `?y_log`, `?x_log`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_fg`, `$gmlcode_bg`,
  `?draw_base`, `?draw_piecewise`, `?draw_curve`, `?draw_histogram`, `?draw_errors`,
  `?draw_symbols`, `?fill_curve`, `$draw_options`, `$err_options`, `$symbol`)

- `?fixed_order_nlo_events`   (default: `false`)
  Induces the generation of fixed-order NLO events.

- `fks_delta_i`   (default: `2.00000E+00`)
  Real parameter for the FKS phase space that applies a cut to the y variable with $0 < \delta_\mathrm{I} \le 2$
  for initial state singularities only. The dependence on the parameter vanishes between
  real subtraction and integrated subtraction term. For debugging purposes.

- `fks_delta_o`   (default: `2.00000E+00`)
  Real parameter for the FKS phase space that applies a cut to the y variable with $0 < \delta_o \le 2$

for final state singularities only. The dependence on the parameter vanishes between real subtraction and integrated subtraction term. For debugging purposes.

- `fks_dij_exp1`   (default: 1.00000E+00)
  Fine-tuning parameters of the FKS final state partition functions. The exact meaning depends on the mapping implementation. (cf. also `fks_dij_exp2`, `$fks_mapping_type`, `fks_xi_min`, `fks_y_max`)

- `fks_dij_exp2`   (default: 1.00000E+00)
  Fine-tuning parameters of the FKS initial state partition functions. The exact meaning depends on the mapping implementation. (cf. also `fks_dij_exp1`, `$fks_mapping_type`, `fks_xi_min`, `fks_y_max`)

- `$fks_mapping_type`   (default: `"default"`)
  Sets the FKS mapping type. Possible values are `"default"` and `"resonances"`. The latter option activates the resonance-aware subtraction mode and induces the generation of a soft mismatch component. (cf. also `fks_dij_exp1`, `fks_dij_exp2`, `fks_xi_min`, `fks_y_max`)

- `fks_xi_cut`   (default: 1.00000E+00)
  (Experimental) Real parameter for the FKS phase space that applies a cut to ξ variable with $0 < \xi_{\text{cut}} \leq \xi_{\text{max}}$. The dependence on the parameter vanishes between real subtraction and integrated subtraction term. Could thus be used for debugging. This is not implemented properly, use at your own risk!

- `fks_xi_min`   (default: 0.00000E+00)
  Real parameter for the FKS phase space that sets the numerical lower value of the ξ variable. Valid for the value range [`tiny_07`, 1], where value inputs out of bounds will take the value of the closest bound. Here, `tiny_07` = `1E0_default` * `epsilon (0._default)`, where `epsilon` is an intrinsic Fortran function. (cf. also `fks_dij_exp1`, `fks_dij_exp2`, `$fks_mapping_type`, `fks_y_max`)

- `fks_y_max`   (default: 1.00000E+00)
  Real parameter for the FKS phase space that sets the numerical upper value of the $|y|$ variable. Valid for ranges [0, 1], where value inputs out of bounds will take the value of the closest bound. Only supported for massless FSR. (cf. also `fks_dij_exp1`, `$fks_mapping_type`, `fks_dij_exp2`)

- `form_threads`   (default: 2)
  The number of threads used by `Gosam` when matrix elements are evaluated using `FORM`

- `form_workspace`   (default: 1000)
  The size of the workspace `Gosam` requires from `FORM`. Inside `FORM`, it corresponds to the heap size used by the algebra processor.

- `gaussian_spread1`   (default: 0.00000E+00)
  Parameter that sets the energy spread (σ value) of the first beam for a Gaussian spectrum. (cf. `gaussian`)

- **gaussian_spread2**   (default: 0.00000E+00)
  Ditto, for the second beam.

- **genkt_algorithm**   (fixed value: 3)
  Specifies a jet algorithm for the ($\rightarrow$) jet_algorithm command, used in the ($\rightarrow$) cluster subevent function. At the moment only available for the interfaced external FastJet package. (cf. also kt_algorithm, cambridge_for_passive_algorithm, plugin_algorithm, genkt_for_passive_algorithm, ee_[gen]kt_algorithm, jet_r), jet_p

- **genkt_for_passive_algorithm**   (fixed value: 13)
  Specifies a jet algorithm for the ($\rightarrow$) jet_algorithm command, used in the ($\rightarrow$) cluster subevent function. At the moment only available for the interfaced external FastJet package. (cf. also kt_algorithm, cambridge_for_passive_algorithm, plugin_algorithm, genkt_algorithm, ee_[gen]kt_algorithm, jet_r)

- **gks_multiplicity**   (default: 0)
  Jet multiplicity for the GKS merging scheme.

- **$gmlcode_bg**   (default: "")
  Settings for WHIZARD's internal graphics output: string variable that allows to define a background for plots and histograms (i.e. it is overwritten by the plot/histogram), e.g. a grid: $gmlcode_bg = "standardgrid.lr(5);". For more details, see the gamelan manual. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_fg, ?draw_histogram, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- **$gmlcode_fg**   (default: "")
  Settings for WHIZARD's internal graphics output: string variable that allows to define a foreground for plots and histograms (i.e. it overwrites the plot/histogram), e.g. a grid: $gmlcode_bg = "standardgrid.lr(5);". For more details, see the gamelan manual. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, $gmlcode_bg, ?draw_histogram, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- **$gosam_fc**   (default: "")
  The Fortran compiler used by Gosam.

- **$gosam_filter_lo**   (default: "")
  The filter string given to Gosam in order to filter out tree-level diagrams. (cf. also $gosam_filter_nlo, $gosam_symmetries)

- **$gosam_filter_nlo**   (default: `""`)
  The same as **$gosam_filter_lo**, but for loop matrix elements. (cf. also **$gosam_filter_nlo**, **$gosam_symmetries**)

- **$gosam_symmetries**   (default: `"family,generation"`)
  String variable that is transferred to `Gosam` configuration file to determine whether certain helicity configurations are considered to be equal. Possible values are `flavour`, `family` etc. For more info see the `Gosam` manual.

- **graph_height_mm**   (default: `90`)
  Settings for `WHIZARD`'s internal graphics output: integer value that sets the height of a graph or histogram in millimeters. (cf. also **?normalize_bins**, **$obs_label**, **$obs_unit**, **$title**, **$description**, **$x_label**, **$y_label**, **graph_width_mm**, **?y_log**, **?x_log**, **x_min**, **x_max**, **y_min**, **y_max**, **$gmlcode_bg**, **$gmlcode_fg**, **?draw_histogram**, **?draw_base**, **?draw_piecewise**, **?fill_curve**, **?draw_curve**, **?draw_errors**, **?draw_symbols**, **$fill_options**, **$draw_options**, **$err_options**, **$symbol**)

- **graph_width_mm**   (default: `130`)
  Settings for `WHIZARD`'s internal graphics output: integer value that sets the width of a graph or histogram in millimeters. (cf. also **?normalize_bins**, **$obs_label**, **$obs_unit**, **$title**, **$description**, **$x_label**, **$y_label**, **graph_height_mm**, **?y_log**, **?x_log**, **x_min**, **x_max**, **y_min**, **y_max**, **$gmlcode_bg**, **$gmlcode_fg**, **?draw_histogram**, **?draw_base**, **?draw_piecewise**, **?fill_curve**, **?draw_curve**, **?draw_errors**, **?draw_symbols**, **$fill_options**, **$draw_options**, **$err_options**, **$symbol**)

- **hadron_enhanced_fraction**   (default: `1.00000E-02`)
  Fraction of Lund strings that break with enhanced width. [not yet active]

- **hadron_enhanced_width**   (default: `2.00000E+00`)
  Enhancement factor for the width of breaking Lund strings. [not yet active]

- **?hadronization_active**   (default: `false`)
  Master flag to switch hadronization (through the attached `PYTHIA` package) on or off. As a default, it is off. (cf. also **?allow_shower**, **?ps_** ..., **$ps_** ..., **?mlm_** ...)

- **$hadronization_method**   (default: `"PYTHIA6"`)
  Determines whether `WHIZARD`'s own hadronization or the (internally included) `PYTHIA6` should be used.

- **?helicity_selection_active**   (default: `true`)
  Flag that decides whether `WHIZARD` uses a numerical selection rule for vanishing helicities: if active, then, if a certain helicity combination yields an absolute (`O'Mega`) matrix element smaller than a certain threshold ($\rightarrow$ `helicity_selection_threshold`) more often than a certain cutoff ($\rightarrow$ `helicity_selection_cutoff`), it will be dropped.

- `helicity_selection_cutoff`   (default: 1000)
  Integer parameter that gives the number a certain helicity combination of an (O'Mega)
  amplitude has to be below a certain threshold ($\rightarrow$ `helicity_selection_threshold`) in
  order to be dropped from then on. (cf. also `?helicity_selection_active`)

- `helicity_selection_threshold`   (default: 1.00000E+10)
  Real parameter that gives the threshold for the absolute value of a certain helicity combi-
  nation of an (O'Mega) amplitude. If a certain number ($\rightarrow$ `helicity_selection_cutoff`)
  of calls stays below this threshold, that combination will be dropped from then on. (cf.
  also `?helicity_selection_active`)

- `?hepevt_ensure_order`   (default: `false`)
  Flag to ensure that the particle set confirms the HEPEVT standard. This involves some
  copying and reordering to guarantee that mothers and daughters are always next to each
  other. Usually this is not necessary.

- `$hepmc3_mode`   (default: `"HepMC3"`)
  This specifies the writer mode for HepMC3. Possible values are `HepMC2`, `HepMC3` (default),
  `HepEVT`, `Root`. and `RootTree` (cf. also `hepmc`)

- `?hepmc3_write_flows`   (default: `false`)
  Flag for the HepMC3 event format that decideswhether to write out color flows. The
  default is `false`. (cf. also `hepmc`)

- `?hepmc_output_cross_section`   (default: `false`)
  Flag for the HepMC event format that allows to write out the cross section (and error)
  from the integration together with each HepMC event. This can be used by programs like
  Rivet to scale histograms according to the cross section. (cf. also `hepmc`)

- `?hoppet_b_matching`   (default: `false`)
  Flag that switches on the matching between 4- and 5-flavor schemes for hadron collider
  b-parton initiated processes. Works either with builtin PDFs or with the external `LHAPDF`
  interface. Needs the external `HOPPET` library to be linked. (cf. `beams`, `pdf_builtin`,
  `lhapdf`)

- `$integrate_workspace`
  Character string that tells `WHIZARD` the subdirectory where to find the run-specific phase-
  space configuration and the `VAMP` and `VAMP2` grid files. If undefined (as per default),
  `WHIZARD` creates them and searches for them in the current directory. (cf. also `$job_id`,
  `$run_id`, `$compile_workspace`)

- `$integration_method`   (default: `"vamp"`)
  This string variable specifies the method for performing the multi-dimensional phase-
  space integration. The default is the `VAMP` algorithm (`"vamp"`), other options are via
  the numerical midpoint rule (`"midpoint"`) or an alternate `VAMP2` implementation that is
  MPI-parallelizable (`"vamp2"`).

- `integration_results_verbosity` (default: 1)
  Integer parameter for the verbosity of the integration results in the process-specific logfile.

- `?integration_timer` (default: `true`)
  This flag switches the integration timer on and off, that gives the estimate for the duration of the generation of 10,000 unweighted events for each integrated process.

- `?isotropic_decay` (default: `false`)
  Flag that – in case of using factorized production and decays using the ($\rightarrow$) `unstable` command – tells `WHIZARD` to switch off spin correlations completely (isotropic decay). (cf. also `unstable`, `?auto_decays`, `decay_helicity`, `?diagonal_decay`)

- `isr_alpha` (default: 0.00000E+00)
  For lepton collider initial-state QED radiation (ISR), this real parameter sets the value of α_{em} used in the structure function. If not set, it is taken from the parameter set of the physics model in use (cf. also `isr`, `isr_q_max`, `isr_mass`, `isr_order`, `?isr_recoil`, `?isr_keep_energy`)

- `?isr_handler` (default: `false`)
  Activate ISR handler for event generation (no effect on integration). Requires `isr_recoil` = `false`

- `?isr_handler_keep_mass` (default: `true`)
  If `true` (default), force the incoming partons of the hard process (after radiation) on their mass shell. Otherwise, insert massless on-shell momenta. This applies only for event generation (no effect on integration, cf. also `?isr_handler`)

- `$isr_handler_mode` (default: `"trivial"`)
  Operation mode for the ISR event handler. Allowed values: `trivial` (no effect), `recoil` (recoil kinematics with two photons)

- `?isr_keep_energy` (default: `false`)
  As the splitting kinematics for the ISR structure function violates Lorentz invariance when the recoil is switched on, this flag forces energy conservation when set to true, otherwise violating energy conservation. (cf. also `isr`, `isr_q_max`, `isr_mass`, `isr_order`, `?isr_recoil`, `?isr_alpha`)

- `isr_log_order` (default: 0)
  For lepton collider initial-state QED radiation (ISR), this integer parameters sets the logarithmic order: 0 (default) is LL, 1 is NLL. (cf. also `isr`, `isr_q_max`, `isr_mass`, `isr_alpha`, `?isr_recoil`, `?isr_keep_energy`, `isr_order`)

- `isr_mass` (default: 0.00000E+00)
  This real parameter allows to set by hand the mass of the incoming particle for lepton collider initial-state QED radiation (ISR). If not set, the mass for the initial beam particle is taken from the model in use. (cf. also `isr`, `isr_q_max`, `isr_alpha`, `isr_order`, `?isr_recoil`, `?isr_keep_energy`, `isr_log_order`)

- **isr_order**   (default: 3)
  For lepton collider initial-state QED radiation (ISR), this integer parameter allows to set the order up to which hard-collinear radiation is taken into account. Default is the highest available, namely third order. (cf. also isr, isr_q_max, isr_mass, isr_alpha, ?isr_recoil, ?isr_keep_energy, isr_log_order)

- **isr_q_in**   (default: -1.00000E+00)
  This is the starting scale for the running of the QED coupling alpha. If negative, the electron mass is taken. (cf. also isr, isr_q_max, isr_alpha, isr_order, ?isr_recoil, ?isr_keep_energy, isr_log_order)

- **isr_q_max**   (default: 0.00000E+00)
  This real parameter allows to set the scale of the initial-state QED radiation (ISR) structure function. If not set, it is taken internally to be $\sqrt{s}$. (cf. also isr, isr_alpha, isr_mass, isr_order, ?isr_recoil, ?isr_keep_energy)

- **?isr_recoil**   (default: false)
  Flag to switch on recoil, i.e. a non-vanishing p_T-kick for the lepton collider initial-state QED radiation (ISR). (cf. also isr, isr, isr_alpha, isr_mass, isr_order, isr_q_max, isr_log_order)

- **jet_algorithm**   (default: 999)
  Variable that allows to set the type of jet algorithm when using the external FastJet library. It accepts one of the following algorithms: ($\rightarrow$) kt_algorithm, ($\rightarrow$) cambridge_[for_passive_]algorithm, ($\rightarrow$) antikt_algorithm, ($\rightarrow$) plugin_algorithm, ($\rightarrow$) genkt_[for_passive_]algorithm, ($\rightarrow$) ee_[gen]kt_algorithm). (cf. also cluster, jet_p, jet_r, jet_ycut)

- **jet_dcut**   (default: 0.00000E+00)
  Value for the d_{ij} separation measure used in the $e^+e^-k_T$ algorithms that are available via the interface to the FastJet package. (cf. also cluster, combine, kt_algorithm, jet_algorithm, cambridge_[for_passive_]algorithm, antikt_algorithm, plugin_algorithm, genkt_[for_passive_]algorithm, ee_[gen]kt_algorithm, jet_p, jet_r)

- **jet_p**   (default: 0.00000E+00)
  Value for the exponent of the distance measure R in the generalized k_T algorithms that are available via the interface to the FastJet package. (cf. also cluster, combine, jet_algorithm, kt_algorithm, cambridge_[for_passive_]algorithm, antikt_algorithm, plugin_algorithm, genkt_[for_passive_]algorithm, ee_[gen]kt_algorithm, jet_r, jet_ycut)

- **jet_r**   (default: 0.00000E+00)
  Value for the distance measure R used in some algorithms that are available via the interface to the FastJet package. (cf. also cluster, combine, jet_algorithm, kt_algorithm,

cambridge_[for_passive_]algorithm, antikt_algorithm,
plugin_algorithm, genkt_[for_passive_]algorithm, ee_[gen]kt_algorithm, jet_p,
jet_ycut)

- jet_ycut   (default: 0.00000E+00)
  Value for the y separation measure used in the Cambridge-Aachen algorithms that are available via the interface to the FastJet package. (cf. also cluster, combine, kt_algorithm, jet_algorithm, cambridge_[for_passive_]algorithm, antikt_algorithm, plugin_algorithm, genkt_[for_passive_]algorithm, ee_[gen]kt_algorithm, jet_p, jet_r)

- $job_id
  Arbitrary string that can be used for creating unique names. The variable is initialized with the value of the job_id option on startup. (cf. also $compile_workspace, $run_id)

- ?keep_beams   (default: false)
  The logical variable ?keep_beams = true/false specifies whether beam particles and beam remnants are included when writing event files. For example, in order to read Les Houches accord event files into PYTHIA, no beam particles are allowed.

- ?keep_failed_events   (default: false)
  In the context of weighted event generation, if set to true, events with failed kinematics will be written to the event output with an associated weight of zero. This way, the total cross section can be reconstructed from the event output.

- ?keep_flavors_when_clustering   (default: false)
  The logical variable ?keep_flavors_when_clustering = true/false specifies whether the flavor of a jet should be kept during cluster when a jet consists of one quark and zero or more gluons. Especially useful for cuts on b-tagged jets (cf. also cluster).

- ?keep_flavors_when_recombining   (default: true)
  The logical variable ?keep_flavors_when_recombining = true/false specifies whether the flavor of a particle should be kept during photon_recombination when a jet/lepton consists of one lepton/quark and zero or more photons (cf. also photon_recombination).

- ?keep_remnants   (default: true)
  The logical variable ?keep_beams = true/false is respected only if ?keep_beams is set. If true, beam remnants are tagged as outgoing particles if they have been neither showered nor hadronized, i.e., have no children. If false, beam remnants are also included in the event record, but tagged as unphysical. Note that for ISR and/or beamstrahlung spectra, the radiated photons are considered as beam remnants.

- kt_algorithm   (fixed value: 0)
  Specifies a jet algorithm for the $(\rightarrow)$ jet_algorithm command, used in the $(\rightarrow)$ cluster subevent function. At the moment only available for the interfaced external FastJet

package. (cf. also `cambridge_[for_passive_]algorithm`, `plugin_algorithm`, `genkt_[for_passive_]algorithm`, `ee_[gen]kt_algorithm`, `jet_r`)

- `lambda_qcd`   (default: 2.00000E-01)
  Real parameter that sets the value for Λ_{QCD} used in the internal evolution for running α_s in WHIZARD. (cf. also `alphas_is_fixed`, `?alphas_from_lhapdf`, `alphas_nf`, `?alphas_from_pdf_builtin`, `?alphas_from_mz`, `?alphas_from_lambda_qcd`, `alphas_order`)

- `lcio_run_id`   (default: 0)
  Allows to set an integer run ID for the LCIO event format. Normally, the process ID is taken as run ID, unless the flag (cf.) `?proc_as_run_id` is set to `false`, cf. also `process`.

- `$lhapdf_dir`   (default: "")
  String variable that tells the path where the LHAPDF library and PDF sets can be found. When the library has been correctly recognized during configuration, this is automatically set by WHIZARD. (cf. also `lhapdf`, `$lhapdf_file`, `lhapdf_photon`, `$lhapdf_photon_file`, `lhapdf_member`, `lhapdf_photon_scheme`)

- `$lhapdf_file`   (default: "")
  This string variable `$lhapdf_file = "<pdf_set>"` allows to specify the PDF set `<pdf_set>` from the external LHAPDF library. It must match the exact name of the PDF set from the LHAPDF library. The default is empty, and the default set from LHAPDF is taken. Only one argument is possible, the PDF set must be identical for both beams, unless there are fundamentally different beam particles like proton and photon. (cf. also `lhapdf`, `$lhapdf_dir`, `lhapdf_photon`, `$lhapdf_photon_file`, `lhapdf_photon_scheme`, `lhapdf_member`)

- `lhapdf_member`   (default: 0)
  Integer variable that specifies the number of the corresponding PDF set chosen via the command ($\rightarrow$) `$lhapdf_file` or ($\rightarrow$) `$lhapdf_photon_file` from the external LHAPDF library. E.g. error PDF sets can be chosen by this. (cf. also `lhapdf`, `$lhapdf_dir`, `$lhapdf_file`, `lhapdf_photon`, `$lhapdf_photon_file`, `lhapdf_photon_scheme`)

- `$lhapdf_photon_file`   (default: "")
  String variable `$lhapdf_photon_file = "<pdf_set>"` analagous to ($\rightarrow$) `$lhapdf_file` for photon PDF structure functions from the external LHAPDF library. The name must exactly match the one of the set from LHAPDF. (cf. `beams`, `lhapdf`, `$lhapdf_dir`, `$lhapdf_file`, `$lhapdf_photon_file`, `lhapdf_member`, `lhapdf_photon_scheme`)

- `lhapdf_photon_scheme`   (default: 0)
  Integer parameter that controls the different available schemes for photon PDFs inside the external LHAPDF library. For more details see the LHAPDF manual. (cf. also `lhapdf`, `$lhapdf_dir`, `$lhapdf_file`, `lhapdf_photon`, `$lhapdf_photon_file`, `lhapdf_member`)

- `$lhef_extension`   (default: "lhe")
  String variable that allows via `$lhef_extension = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the LHEF format are written. If not set, the default

file name and suffix is *<process_name>*.lhe. (cf. also sample_format, $sample, lhef, $lhef_extension, $lhef_version, ?lhef_write_sqme_prc, ?lhef_write_sqme_ref, ?lhef_write_sq

- $lhef_version   (default: "2.0")
  Specifier for the Les Houches Accord (LHEF) event format files with XML headers to discriminate among different versions of this format. (cf. also $sample, sample_format, lhef, $lhef_extension, $lhef_extension, ?lhef_write_sqme_prc, ?lhef_write_sqme_ref, ?lhef_write_

- ?lhef_write_sqme_alt   (default: true)
  Flag that decides whether in the ($\rightarrow$) lhef event format alternative weights of the squared matrix element shall be written in the LHE file. (cf. also $sample, sample_format, lhef, $lhef_extension, $lhef_extension, ?lhef_write_sqme_prc, ?lhef_write_sqme_ref)

- ?lhef_write_sqme_prc   (default: true)
  Flag that decides whether in the ($\rightarrow$) lhef event format the weights of the squared matrix element of the corresponding process shall be written in the LHE file. (cf. also $sample, sample_format, lhef, $lhef_extension, $lhef_extension, ?lhef_write_sqme_ref, ?lhef_write_sqme_alt)

- ?lhef_write_sqme_ref   (default: false)
  Flag that decides whether in the ($\rightarrow$) lhef event format reference weights of the squared matrix element shall be written in the LHE file. (cf. also $sample, sample_format, lhef, $lhef_extension, $lhef_extension, ?lhef_write_sqme_prc, ?lhef_write_sqme_alt)

- $library_name
  Similar to $model_name, this string variable is used solely to access the name of the active process library, e.g. in printf statements. (cf. compile, library, printf, show, $model_name)

- ?logging   (default: true)
  This logical – when set to false – suppresses writing out a logfile (default: whizard.log) for the whole WHIZARD run, or when WHIZARD is run with the --no-logging option, to suppress parts of the logging when setting it to true again at a later part of the SINDARIN input file. Mainly for debugging purposes. (cf. also ?openmp_logging, ?mpi_logging)

- $loop_me_method   (default: "")
  This string variable specifies the method for the matrix elements to be used in the evaluation of the virtual part of the NLO computation. The default is the empty string, i.e. the same as $method. Working options are: "threshold", "openloops", "recola", "gosam". (cf. also $real_tree_me_method, $correlation_me_method and $born_me_method.)

- luminosity   (default: 0.00000E+00)
  This specifier luminosity = *<num>* sets the integrated luminosity (in inverse femtobarns, fb^{-1}) for the event generation of the processes in the SINDARIN input files. Note that WHIZARD itself chooses the number from the luminosity or from the n_events specifier, whichever would give the larger number of events. As this depends on the cross section

under consideration, it might be different for different processes in the process list. (cf. `n_events`, `$sample`, `sample_format`, `?unweighted`)

- `max_bins`   (default: 20)
  Integer parameter that modifies the settings of the `VAMP` integrator's grid parameters. It sets the maximal number of bins per integration dimension. (cf. `iterations`, `min_bins`, `min_calls_per_channel`, `min_calls_per_bin`)

- `?me_verbose`   (default: `false`)
  Flag determining whether the makefile command for generating and compiling the `O'Mega` matrix element code is silent or verbose. Default is silent.

- `$method`   (default: `"omega"`)
  This string variable specifies the method for the matrix elements to be used in the evaluation. The default is the intrinsic `O'Mega` matrix element generator (`"omega"`), other options are: `"ovm"`, `"unit_test"`, `"template_unity"`, `"threshold"`. For processes defined `"template"`, with `nlo_calculation = ...`, please refer to `$born_me_method`, `$real_tree_me_method`, `$loop_me_method` and `$correlation_me_method`.

- `min_bins`   (default: 3)
  Integer parameter that modifies the settings of the `VAMP` integrator's grid parameters. It sets the minimal number of bins per integration dimension. (cf. `iterations`, `max_bins`, `min_calls_per_channel`, `min_calls_per_bin`)

- `min_calls_per_bin`   (default: 10)
  Integer parameter that modifies the settings of the `VAMP` integrator's grid parameters. It sets the minimal number every bin in an integration dimension must be called. If the number of calls from the iterations is too small, `WHIZARD` will automatically increase the number of calls. (cf. `iterations`, `min_calls_per_channel`, `min_bins`, `max_bins`)

- `min_calls_per_channel`   (default: 10)
  Integer parameter that modifies the settings of the `VAMP` integrator's grid parameters. It sets the minimal number every channel must be called. If the number of calls from the iterations is too small, `WHIZARD` will automatically increase the number of calls. (cf. `iterations`, `min_calls_per_bin`, `min_bins`, `max_bins`)

- `mlm_ETclusfactor`   (default: 2.00000E-01)
  This real parameter is a factor that enters the calculation of the y_{cut} measure for jet clustering after the parton shower in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)

- `mlm_ETclusminE`   (default: 5.00000E+00)
  This real parameter is a minimal energy that enters the calculation of the y_{cut} measure for jet clustering after the parton shower in the MLM jet matching between hard matrix

elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Eclusfactor`   (default: 1.00000E+00)
  This real parameter is a factor that enters the calculation of the y_{cut} measure for jet clustering after the parton shower in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Emin`   (default: 0.00000E+00)
  Real parameter that sets a minimal energy E_{min} value as an infrared cutoff in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Qcut_ME`   (default: 0.00000E+00)
  Real parameter that in the MLM jet matching between hard matrix elements and QCD parton shower sets a possible virtuality cut on jets from the hard matrix element. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Qcut_PS`   (default: 0.00000E+00)
  Real parameter that in the MLM jet matching between hard matrix elements and QCD parton shower sets a possible virtuality cut on jets from the parton shower. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Rclusfactor`   (default: 1.00000E+00)
  This real parameter is a factor that enters the calculation of the y_{cut} measure for jet clustering after the parton shower in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_Rmin`   (default: 0.00000E+00)
  Real parameter that sets a minimal R distance value that enters the y_{cut} jet clustering measure in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_etaclusfactor`   (default: 1.00000E+00)
  This real parameter is a factor that enters the calculation of the y_{cut} measure for jet clustering after the parton shower in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- `mlm_etamax`   (default: 0.00000E+00)
  This real parameter sets a maximal pseudorapidity that enters the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `mlm_` ..., `?hadronization_active`)

- **?mlm matching**   (default: `false`)
  Master flag to switch on MLM (LO) jet matching between hard matrix elements and the QCD parton shower. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)

- **mlm nmaxMEjets**   (default: `0`)
  This integer sets the maximal number of jets that are available from hard matrix elements in the MLM jet matching between hard matrix elements and QCD parton shower. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)

- **mlm ptmin**   (default: `0.00000E+00`)
  This real parameter sets a minimal p_T that enters the y_{cut} jet clustering measure in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)

- **$model name**
  This variable makes the locally used physics model available as a string, e.g. as `show` (`$model_name`). However, the user is not able to change the current model by setting this variable to a different string. (cf. also `model`, `$library_name`, `printf`, `show`)

- **?mpi logging**   (default: `false`)
  This logical – when set to `false` – suppresses writing out messages about MPI parallelization (number of used workers etc.) on screen and into the logfile (default name `whizard.log`) for the whole `WHIZARD` run. Mainly for debugging purposes. (cf. also `?logging`, `?openmp_logging`)

- **?muli active**   (default: `false`)
  Master flag that switches on `WHIZARD`'s module for multiple interaction with interleaved QCD parton showers for hadron colliders. Note that this feature is still experimental. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`)

- **mult call dglap**   (default: `1.00000E+00`)
  (Real-valued) multiplier for the number of calls used in the integration of the DGLAP remnant NLO component. This way, a higher accuracy can be achieved for this component, while simultaneously avoiding redundant integration calls for the other components. (cf. also `mult_call_real`, `mult_call_virt`)

- **mult call real**   (default: `1.00000E+00`)
  (Real-valued) multiplier for the number of calls used in the integration of the real subtraction NLO component. This way, a higher accuracy can be achieved for the real component, while simultaneously avoiding redundant integration calls for the other components. (cf. also `mult_call_dglap`, `mult_call_virt`)

- **mult call virt**   (default: `1.00000E+00`)
  (Real-valued) multiplier for the number of calls used in the integration of the virtual NLO component. This way, a higher accuracy can be achieved for this component, while

simultaneously avoiding redundant integration calls for the other components. (cf. also `mult_call_dglap`, `mult_call_real`)

- `n_bins`   (default: `20`)
  Settings for `WHIZARD`'s internal graphics output: integer value that sets the number of bins in histograms. (cf. also `?normalize_bins`, `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_width_mm`, `graph_height_mm`, `?y_log`, `?x_log`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_histogram`, `?draw_base`, `?draw_piecewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- `n_calls_test`   (default: `0`)
  Integer variable that allows to set a certain number of matrix element sampling test calls without actually integrating the process under consideration. (cf. `integrate`)

- `n_events`   (default: `0`)
  This specifier `n_events = <num>` sets the number of events for the event generation of the processes in the `SINDARIN` input files. Note that WHIZARD itself chooses the number from the `n_events` or from the `luminosity` specifier, whichever would give the larger number of events. As this depends on the cross section under consideration, it might be different for different processes in the process list. (cf. `luminosity`, `$sample`, `sample_format`, `?unweighted`, `event_index_offset`)

- `$negative_sf`   (default: `"default"`)
  String variable to set the behavior to either keep negative structure function/PDF values or set them to zero. The default (`"default"`) takes the first option for NLO and the second for LO processes. Explicit behavior can be set with `"positive"` or `"negative"`.

- `?negative_weights`   (default: `false`)
  Flag that tells `WHIZARD` to allow negative weights in integration and simulation. (cf. also `simulate`, `?unweighted`)

- `$nlo_correction_type`   (default: `"QCD"`)
  String variable which sets the NLO correction type via `nlo_correction_type = "<type>"` to either `"QCD"`, `"EW"`, or to all with `<type>` set to `"Full"`. Must be set before the `process` statement.

- `?nlo_cut_all_real_sqmes`   (default: `false`)
  Flag that decides whether in the case that the real component does not pass a cut, its subtraction term shall be discarded for that phase space point as well or not. (cf. also `?nlo_use_born_scale`)

- `?nlo_reuse_amplitudes_fks`   (default: `false`)
  Only compute real and virtual amplitudes for subprocesses that give a different amplitude and reuse the result for equivalent subprocesses. Might give a speed-up for some processes.

Might break others, especially in cases where resonance histories are needed. Experimental feature, use at your own risk!

- **?nlo_use_born_scale**   (default: `false`)
  Flag that decides whether a scale expression defined for the Born component of an NLO process shall be applied to all other components as well or not. (cf. also **?nlo_cut_all_real_sqmes**)

- **?normalize_bins**   (default: `false`)
  Settings for WHIZARD's internal graphics output: flag that determines whether the weights shall be normalized to the bin width or not. (cf. also **n_bins**, **$obs_label**, **$obs_unit**, **$title**, **$description**, **$x_label**, **$y_label**, **graph_width_mm**, **graph_height_mm**, **?y_log**, **?x_log**, **x_min**, **x_max**, **y_min**, **y_max**, **$gmlcode_bg**, **$gmlcode_fg**, **?draw_histogram**, **?draw_base**, **?draw_piecewise**, **?fill_curve**, **?draw_curve**, **?draw_errors**, **$symbol**, **?draw_symbols**, **$fill_options**, **$draw_options**, **$err_options**)

- **$obs_label**   (default: `""`)
  Settings for WHIZARD's internal graphics output: this is a string variable **$obs_label = "*<LaTeX_Code>*"** that allows to attach a label to a plotted or histogrammed observable. (cf. also **n_bins**, **?normalize_bins**, **$obs_unit**, **$title**, **$description**, **$x_label**, **$y_label**, **?y_log**, **?x_log**, **graph_width_mm**, **graph_height_mm**, **x_min**, **x_max**, **y_min**, **y_max**, **$gmlcode_bg**, **$gmlcode_fg**, **?draw_base**, **?draw_histogram**, **?draw_piecewise**, **?fill_curve**, **?draw_curve**, **?draw_errors**, **$symbol**, **?draw_symbols**, **$fill_options**, **$draw_options**, **$err_options**)

- **$obs_unit**   (default: `""`)
  Settings for WHIZARD's internal graphics output: this is a string variable **$obs_unit = "*<LaTeX_Code>*"** that allows to attach a LaTeX physical unit to a plotted or histogrammed observable. (cf. also **n_bins**, **?normalize_bins**, **$obs_unit**, **$title**, **$description**, **$x_label**, **$y_label**, **?y_log**, **?x_log**, **graph_width_mm**, **graph_height_mm**, **x_min**, **x_max**, **y_min**, **y_max**, **$gmlcode_bg**, **$gmlcode_fg**, **?draw_base**, **?draw_histogram**, **?fill_curve**, **?draw_piecewise**, **?draw_curve**, **?draw_errors**, **$symbol**, **?draw_symbols**, **$fill_options**, **$draw_options**, **$err_options**)

- **$omega_flags**   (default: `""`)
  String variable that allows to pass flags to the **O'Mega** matrix element generator. Normally, WHIZARD takes care of all flags automatically. Note that for restrictions of intermediate states, there is a special string variable: (cf. →) **$restrictions**.

- **?omega_openmp**   (default: `false`)
  Flag to switch on or off OpenMP multi-threading for **O'Mega** matrix elements. (cf. also **$method**, **$omega_flag**)

- **?omega_write_phs_output**   (default: `false`)
  This flag decides whether a the phase-space output is produced by the **O'Mega** matrix element generator. This output is written to file(s) and contains the Feynman diagrams

which belong to the process(es) under consideration. The file is mandatory whenever the variable $phs_method has the value fast_wood, i.e. if the phase-space file is provided by cascades2.

- $openloops_allowed_libs   (default: "")
  String variable to restrict the allowed OpenLoopsprocess libraries for a process. (cf. also $method, openloos_verbosity, ?openloops_use_cms, openloops_stability_log, ?openloops_switch_off_muon_yukawa)

- $openloops_extra_cmd   (default: "")
  String variable to transfer customized special commands to OpenLoops. The three supported examples $openloops_extra_command = "extra approx top/stop/not" are for selection of subdiagrams in top production. (cf. also $method, openloos_verbosity, ?openloops_use_cms, openloops_stability_log, ?openloops_switch_off_muon_yukawa)

- openloops_phs_tolerance   (default: 7)
  This integer parameter gives via openloops_phs_tolerance = <n> the relative numerical tolerance 10^{-n} for the momentum conservation of the external particles within OpenLoops. (cf. also openloos_verbosity, $method, ?openloops_switch_off_muon_yukawa, openloops_stability_log, $openloops_extra_cmd)

- openloops_stability_log   (default: 0)
  Creates the directory stability_log containing information about the performance of the OpenLoopsmatrix elements. Possible values are 0 (No output), 1 (On finish()-call), 2 (Adaptive) and 3 (Always).

- ?openloops_switch_off_muon_yukawa   (default: false)
  Sets the Yukawa coupling of muons for OpenLoops to zero. (cf. also openloos_verbosity, $method, ?openloops_use_cms, openloops_stability_log, $openloops_extra_cmd)

- ?openloops_use_cms   (default: true)
  Activates the complex mass scheme in OpenLoops. (cf. also openloos_verbosity, $method, ?openloops_switch_off_muon_yukawa, openloops_stability_log, $openloops_extra_cmd)

- openloops_verbosity   (default: 1)
  Decides how much OpenLoops output is printed. Can have values 0, 1 and 2, where 2 is the highest verbosity level.

- ?openmp_is_active   (fixed value: false)
  Flag to switch on or off OpenMP multi-threading for WHIZARD. (cf. also ?openmp_logging, openmp_num_threads, openmp_num_threads_default, ?omega_openmp)

- ?openmp_logging   (default: true)
  This logical – when set to false – suppresses writing out messages about OpenMP parallelization (number of used threads etc.) on screen and into the logfile (default name

whizard.log) for the whole WHIZARD run. Mainly for debugging purposes. (cf. also ?logging, ?mpi_logging)

- openmp_num_threads   (default: 1)
  Integer parameter that sets the number of OpenMP threads for multi-threading. (cf. also ?openmp_logging, openmp_num_threads_default, ?omega_openmp)

- openmp_num_threads_default   (fixed value: 1)
  Integer parameter that shows the number of default OpenMP threads for multi-threading. Note that this parameter can only be accessed, but not reset by the user. (cf. also ?openmp_logging, openmp_num_threads, ?omega_openmp)

- ?out_advance   (default: true)
  Flag that sets advancing in the printf output commands, i.e. continuous printing with no line feed etc. (cf. also printf)

- $out_file   (default: "")
  This character variable allows to specify the name of the data file to which the histogram and plot data are written (cf. also write_analysis, open_out, close_out)

- ?pacify   (default: false)
  Flag that allows to suppress numerical noise and give screen and log file output with a lower number of significant digits. Mainly for debugging purposes. (cf. also ?sample_pacify)

- $pdf_builtin_set   (default: "CTEQ6L")
  For WHIZARD's internal PDF structure functions for hadron colliders, this string variable allows to set the particular PDF set. (cf. also pdf_builtin, pdf_builtin_photon)

- photon_iso_eps   (default: 1.00000E+00)
  Photon isolation parameter ϵ_γ (energy fraction) from hep-ph/9801442 (cf. also photon_iso_n, photon_iso_r0)

- photon_iso_n   (default: 1.00000E+00)
  Photon isolation parameter n (cone function exponent) from hep-ph/9801442 (cf. also photon_iso_eps, photon_iso_r0)

- photon_iso_r0   (default: 4.00000E-01)
  Photon isolation parameter R_0^γ (isolation cone radius) from hep-ph/9801442 (cf. also photon_iso_eps, photon_iso_n)

- photon_rec_r0   (default: 1.00000E-01)
  Photon recombination parameter R_0^γ for photon recombination in NLO EW calculations

- phs_e_scale   (default: 1.00000E+01)
  Real parameter that sets the energy scale that acts as a cutoff for parameterizing radiation-like kinematics in the wood phase space method. WHIZARD takes the maximum of this value and the width of the propagating particle as a cutoff. (cf. also phs_threshold_t,

phs_threshold_s, phs_t_channel, phs_off_shell, phs_m_scale, phs_q_scale,
?phs_keep_resonant, ?phs_step_mapping, ?phs_step_mapping_exp, ?phs_s_mapping)

- **$phs_file** (default: "")
  This string variable allows the user to set an individual file name for the phase space parameterization for a particular process: $phs_file = "*<file_name>*". If not set, the default is *<proc_name>*_*<proc_comp>*.*<run_id>*.phs. (cf. also $phs_method)

- **?phs_keep_nonresonant** (default: true)
  Flag that decides whether the wood phase space method takes into account also non-resonant contributions. (cf. also phs_threshold_t, phs_threshold_s, phs_t_channel, phs_off_shell, phs_m_scale, phs_q_scale, phs_e_scale, ?phs_step_mapping, ?phs_step_mapping_exp, ?phs_s_mapping)

- **phs_m_scale** (default: 1.00000E+01)
  Real parameter that sets the mass scale that acts as a cutoff for parameterizing collinear and infrared kinematics in the wood phase space method. WHIZARD takes the maximum of this value and the mass of the propagating particle as a cutoff. (cf. also phs_threshold_t, phs_threshold_s, phs_t_channel, phs_off_shell, phs_e_scale, phs_q_scale, ?phs_keep_resonant, ?phs_step_mapping, ?phs_step_mapping_exp, ?phs_s_mapping)

- **$phs_method** (default: "default")
  String variable that allows to choose the phase-space parameterization method. The default is the "wood" method that takes into account electroweak/BSM resonances. Note that this might not be the best choice for (pure) QCD amplitudes. (cf. also $phs_file)

- **phs_off_shell** (default: 2)
  Integer parameter that sets the number of off-shell (not t-channel-like, non-resonant) lines that are taken into account to find a valid phase-space setup in the wood phase-space method. (cf. also phs_threshold_t, phs_threshold_s, phs_t_channel, phs_e_scale, phs_m_scale, phs_q_scale, ?phs_keep_resonant, ?phs_step_mapping, ?phs_step_mapping_exp, ?phs_s_mapping)

- **?phs_only** (default: false)
  Flag (particularly as optional argument of the → integrate command) that allows to only generate the phase space file, but not perform the integration. (cf. also $phs_method, $phs_file)

- **phs_q_scale** (default: 1.00000E+01)
  Real parameter that sets the momentum transfer scale that acts as a cutoff for parameterizing t- and u-channel like kinematics in the wood phase space method. WHIZARD takes the maximum of this value and the mass of the propagating particle as a cutoff. (cf. also phs_threshold_t, phs_threshold_s, phs_t_channel, phs_off_shell, phs_e_scale, phs_m_scale, ?phs_keep_resonant, ?phs_step_mapping, ?phs_step_mapping_exp, ?phs_s_mapping)

- `?phs_s_mapping`   (default: `true`)
  Flag that allows special mapping for s-channel resonances. (cf. also `phs_threshold_t`, `phs_threshold_s`, `phs_t_channel`, `phs_off_shell`, `phs_e_scale`, `phs_m_scale`, `?phs_keep_resonant`, `?phs_q_scale`, `?phs_step_mapping`, `?phs_step_mapping_exp`)

- `?phs_step_mapping`   (default: `true`)
  Flag that switches on (or off) a particular phase space mapping for resonances, where the mass and width of the resonance are explicitly set as channel cutoffs. (cf. also `phs_threshold_t`, `phs_threshold_s`, `phs_t_channel`, `phs_off_shell`, `phs_e_scale`, `phs_m_scale`, `?phs_keep_resonant`, `?phs_q_scale`, `?phs_step_mapping_exp`, `?phs_s_mapping`)

- `?phs_step_mapping_exp`   (default: `true`)
  Flag that switches on (or off) a particular phase space mapping for resonances, where the mass and width of the resonance are explicitly set as channel cutoffs. This is an exponential mapping in contrast to ($\rightarrow$) `?phs_step_mapping`. (cf. also `phs_threshold_t`, `phs_threshold_s`, `phs_t_channel`, `phs_off_shell`, `phs_e_scale`, `phs_m_scale`, `?phs_q_scale`, `?phs_keep_resonant`, `?phs_step_mapping`, `?phs_s_mapping`)

- `phs_t_channel`   (default: `6`)
  Integer parameter that sets the number of t-channel propagators in multi-peripheral diagrams that are taken into account to find a valid phase-space setup in the `wood` phase-space method. (cf. also `phs_threshold_t`, `phs_threshold_s`, `phs_off_shell`, `phs_e_scale`, `phs_m_scale`, `phs_q_scale`, `?phs_keep_resonant`, `?phs_step_mapping`, `?phs_step_mapping_exp`, `?phs_s_mapping`)

- `phs_threshold_s`   (default: `5.00000E+01`)
  For the phase space method `wood`, this real parameter sets the threshold below which particles are assumed to be massless in the s-channel like kinematic regions. (cf. also `phs_threshold_t`, `phs_off_shell`, `phs_t_channel`, `phs_e_scale`, `phs_m_scale`, `phs_q_scale`, `?phs_keep_resonant`, `?phs_step_mapping`, `?phs_step_mapping_exp`, `?phs_s_mapping`)

- `phs_threshold_t`   (default: `1.00000E+02`)
  For the phase space method `wood`, this real parameter sets the threshold below which particles are assumed to be massless in the t-channel like kinematic regions. (cf. also `phs_threshold_s`, `phs_off_shell`, `phs_t_channel`, `phs_e_scale`, `phs_m_scale`, `phs_q_scale`, `?phs_keep_resonant`, `?phs_step_mapping`, `?phs_step_mapping_exp`, `?phs_s_mapping`)

- `plugin_algorithm`   (fixed value: `99`)
  Specifies a jet algorithm for the ($\rightarrow$) `jet_algorithm` command, used in the ($\rightarrow$) `cluster` subevent function. At the moment only available for the interfaced external `FastJet` package. (cf. also `kt_algorithm`, `cambridge_for_passive_algorithm`, `genkt_[for_passive_]algorithm`, `ee_[gen]kt_algorithm`, `jet_r`)

- **$polarization_mode**   (default: `"helicity"`)
  String variable that specifies the mode in which the polarization of particles is handled when polarized events are written out. Possible options are `"ignore"`, `"helicity"`, `"factorized"`, and `"correlated"`. For more details cf. the detailed section.

- **?polarized_events**   (default: `false`)
  Flag that allows to select certain helicity combinations in final state particles in the event files, and perform analysis on polarized event samples. (cf. also `simulate`, `polarized`, `unpolarized`)

- **?powheg_disable_sudakov**   (default: `false`)
  This flag allows to set the Sudakov form factor to one. This effectively results in a version of the matrix-element method (MEM) at NLO.

- **powheg_grid_size_xi**   (default: 5)
  Number of ξ points in the POWHEG grid.

- **powheg_grid_size_y**   (default: 5)
  Number of y points in the POWHEG grid.

- **powheg_lambda**   (default: 0.00000E+00)
  Reference scale of the α_s evolution in the POWHEG matching algorithm. Per default we use $\Lambda_{\overline{MS}}^{n_f=5}$.

- **?powheg_matching**   (default: `false`)
  Activates Powheg matching. Needs to be combined with the `?combined_nlo_integration`-method.

- **powheg_pt_min**   (default: 1.00000E+00)
  Lower p_T-cut-off for the POWHEG hardest emission.

- **?powheg_test_sudakov**   (default: `false`)
  Performs an internal consistency check on the POWHEG event generation.

- **?powheg_use_singular_jacobian**   (default: `false`)
  This allows to give a different normalization of the Jacobian, resulting in an alternative POWHEG damping in the singular regions.

- **?proc_as_run_id**   (default: `true`)
  Normally, for LCIO the process ID (cf. `process_num_id`) is used as run ID, unless this flag is set to `false`, cf. also `process`, `lcio_run_id`.

- **process_num_id**
  Using the integer `process_num_id = <int_var>` one can set a numerical identifier for processes within a process library. This can be set either just before the corresponding `process` definition or as an optional local argument of the latter. (cf. also `process`, `?proc_as_run_id`, `lcio_run_id`)

- **$ps_PYTHIA8_config**   (default: "")
  String variable that allows to pass options for tunes etc. to the attached PYTHIA8 parton shower or hadronization, e.g.: **$ps_PYTHIA8_config = "PartonLevel:MPI = off"**. (cf. also
  ?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active)

- **$ps_PYTHIA8_config_file**   (default: "")
  String variable that allows to pass a filename to a PYTHIA8 configuration file.

- **$ps_PYTHIA_PYGIVE**   (default: "")
  String variable that allows to pass options for tunes etc. to the attached PYTHIA parton shower or hadronization, e.g.: **$ps_PYTHIA_PYGIVE = "MSTJ(41)=1"**. (cf. also
  ?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active)

- **ps_fixed_alphas**   (default: 0.00000E+00)
  This real parameter sets the value of α_s if it is (cf. → **?ps_isr_alphas_running**,
  ?ps_fsr_alphas_running) not running in initial and/or final-state QCD showers. (cf. also
  ?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active)

- **?ps_fsr_active**   (default: **false**)
  Flag that switches final-state QCD radiation (FSR) on. (cf. also **?allow_shower, ?ps_
  ..., $ps_ ..., ?mlm_ ..., ?hadronization_active**)

- **?ps_fsr_alphas_running**   (default: **true**)
  Flag that decides whether a running α_s is taken in time-like QCD parton showers. (cf. also **?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active**)

- **ps_fsr_lambda**   (default: 2.90000E-01)
  By this real parameter, the value of Λ_{QCD} used in running α_s for time-like showers is set (except for showers in the decay of a resonance). (cf. also **?allow_shower, ?ps_ ..., $ps_
  ..., ?mlm_ ..., ?hadronization_active**)

- **?ps_isr_active**   (default: **false**)
  Flag that switches initial-state QCD radiation (ISR) on. (cf. also **?allow_shower, ?ps_
  ..., $ps_ ..., ?mlm_ ..., ?hadronization_active**)

- **?ps_isr_alphas_running**   (default: **true**)
  Flag that decides whether a running α_s is taken in space-like QCD parton showers. (cf. also **?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active**)

- **?ps_isr_angular_ordered**   (default: **true**)
  If switched one, this flag forces opening angles of emitted partons in the QCD ISR shower to be strictly ordered, i.e. increasing towards the hard interaction. (cf. also **?allow_shower,
  ?ps_ ..., $ps_ ..., ?mlm_ ..., ?hadronization_active**)

- `ps_isr_lambda` (default: 2.90000E-01)
  By this real parameter, the value of Λ_{QCD} used in running α_s for space-like showers is set. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_isr_minenergy` (default: 1.00000E+00)
  By this real parameter, the minimal effective energy (in the c.m. frame) of a time-like or on-shell-emitted parton in a space-like QCD shower is set. For a hard subprocess that is not in the rest frame, this number is roughly reduced by a boost factor $1/\gamma$ to the rest frame of the hard scattering process. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `?ps_isr_only_onshell_emitted_partons` (default: `false`)
  This flag if set true sets all emitted partons off space-like showers on-shell, i.e. it would not allow associated time-like showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_isr_primordial_kt_cutoff` (default: 5.00000E+00)
  Real parameter that sets the upper cutoff for the primordial k_T distribution inside a hadron. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?hadronization_active`, `?mlm_` ...)

- `ps_isr_primordial_kt_width` (default: 0.00000E+00)
  This real parameter sets the width $\sigma = \langle k_T^2 \rangle$ for the Gaussian primordial k_T distribution inside the hadron, given by: $\exp[-k_T^2/\sigma^2]k_T dk_T$. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `?ps_isr_pt_ordered` (default: `false`)
  By this flag, it can be switched between the analytic QCD ISR shower (`false`, default) and the p_T ISR QCD shower (`true`). (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_isr_tscalefactor` (default: 1.00000E+00)
  The Q^2 scale of the hard scattering process is multiplied by this real factor to define the maximum parton virtuality allowed in time-like QCD showers. This does only apply to t- and u-channels, while for s-channel resonances the maximum virtuality is set by m^2. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_isr_z_cutoff` (default: 9.99000E-01)
  This real parameter allows to set the upper cutoff on the splitting variable z in space-like QCD parton showers. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_mass_cutoff` (default: 1.00000E+00)
  Real value that sets the QCD parton shower lower cutoff scale, where hadronization sets in. (cf. also `?allow_shower`, `?ps_` ..., `$ps_` ..., `?mlm_` ..., `?hadronization_active`)

- `ps_max_n_flavors`   (default: 5)
  This integer parameter sets the maxmimum number of flavors that can be produced in a QCD shower $g \to q\bar{q}$. It is also used as the maximal number of active flavors for the running of α_s in the shower (with a minimum of 3). (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)

- `?ps_taudec_active`   (default: `false`)
  Flag to switch on τ decays, at the moment only via the included external package `TAUOLA` and `PHOTOS`. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)

- `ps_tauola_dec_mode1`   (default: 0)
  Integer code to request a specific τ decay within `TAUOLA` for the decaying τ, and – in correlated decays – for the second τ. For more information cf. the comments in the code or the `TAUOLA` manual. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `ps_tauola_dec_mode2`   (default: 0)
  Integer code to request a specific τ decay within `TAUOLA` for the decaying τ, and – in correlated decays – for the second τ. For more information cf. the comments in the code or the `TAUOLA` manual. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `?ps_tauola_dec_rad_cor`   (default: `true`)
  Flag to switch radiative corrections for τ decays in `TAUOLA` on or off. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `ps_tauola_mh`   (default: 1.25000E+02)
  Real option to set the Higgs mass for Higgs decays into τ leptons in the interface to `TAUOLA`. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `ps_tauola_mix_angle`   (default: 9.00000E+01)
  Option to set the mixing angle between scalar and pseudoscalar Higgs bosons for Higgs decays into τ leptons in the interface to `TAUOLA`. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `?ps_tauola_photos`   (default: `false`)
  Flag to switch on `PHOTOS` for photon showering inside the `TAUOLA` package. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `?ps_tauola_pol_vector`   (default: `false`)
  Flag to decide whether for transverse τ polarization, polarization information should be taken from `TAUOLA` or not. The default is just based on random numbers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- `?ps_tauola_transverse`   (default: `false`)
  Flag to switch transverse τ polarization on or off for Higgs decays into τ leptons. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?ps_taudec_active`)

- **?read_color_factors**   (default: **true**)
  This flag decides whether to read QCD color factors from the matrix element provided by each method, or to try and calculate the color factors in **WHIZARD** internally.

- **?read_raw**   (default: **true**)
  This flag demands **WHIZARD** to (try to) read events (from the internal binary format) first before generating new ones. (cf. **simulate**, **?write_raw**, **$sample**, **sample_format**)

- **real_epsilon**   (fixed value: 0.00000E+00)
  This gives the smallest number E of the same kind as the float type for which $1 + E > 1$. It cannot be set by the user. (cf. also **real_range**, **real_tiny**, **real_precision**).

- **$real_partition_mode**   (default: **"default"**)
  String variable to choose which parts of the real cross section are to be integrated. With the default value (**"default"**) or **"off"** the real cross section is integrated as usual without partition. If set to **"on"** or **"all"**, the real cross section is split into singular and finite part using a partition function F, such that $\mathcal{R} = [1 - F(p_T^2)]\mathcal{R} + F(p_T^2)\mathcal{R} = \mathcal{R}_{\text{fin}} + \mathcal{R}_{\text{sing}}$. The emission generation is then performed using $\mathcal{R}_{\text{sing}}$, whereas $\mathcal{R}_{\text{fin}}$ is treated separately. If set to **"singular"** (**"finite"**), only the singular (finite) real component is integrated.(cf. also **real_partition_scale**)

- **real_partition_scale**   (default: 1.00000E+01)
  This real variable sets the invariant mass of the FKS pair used as a separator between the singular and the finite part of the real subtraction terms in an NLO calculation, e.g. in $e^+e^- \rightarrow t\bar{t}j$. (cf. also **$real_partition_mode**)

- **real_precision**   (fixed value: 15)
  This integer gives the precision of the numeric model for the real float type in use. It cannot be set by the user. (cf. also **real_range**, **real_epsilon**, **real_tiny**).

- **real_range**   (fixed value: 307)
  This integer gives the decimal exponent range of the numeric model for the real float type in use. It cannot be set by the user. (cf. also **real_precision**, **real_epsilon**, **real_tiny**).

- **real_tiny**   (fixed value: 0.00000E+00)
  This gives the smallest positive (non-zero) number in the numeric model for the real float type in use. It cannot be set by the user. (cf. also **real_range**, **real_epsilon**, **real_precision**).

- **$real_tree_me_method**   (default: **""**)
  This string variable specifies the method for the matrix elements to be used in the evaluation of the real part of the NLO computation. The default is the same as the **$method**, i.e. the intrinsic O'Mega matrix element generator (**"omega"**), other options are: **"ovm"**, **"unit_test"**, **"template"**, **"template_unity"**, **"threshold"**, **"gosam"**,

"openloops". (cf. also \$born_me_method, \$correlation_me_method, \$dglap_me_method and \$loop_me_method.)

- ?recover_beams   (default: true)
  Flag that decides whether the beam particles should be reconstructed when reading event/rescanning files into WHIZARD. (cf. rescan, ?update_event, ?update_sqme, ?update_weight)

- relative_error_goal   (default: 0.00000E+00)
  Real parameter that allows the user to set a minimal relative error that should be achieved in the Monte-Carlo integration of a certain process. If that goal is reached, grid and weight adaptation stop, and this result is used for simulation. (cf. also integrate, iterations, accuracy_goal, error_goal, error_threshold)

- ?report_progress   (default: true)
  Flag for the O'Mega matrix element generator whether to print out status messages about progress during matrix element generation. (cf. also \$method, \$omega_flags)

- ?rescan_force   (default: false)
  Flag that allows to bypass essential checks on the particle set when reading event/rescanning files into WHIZARD. (cf. rescan, ?update_event, ?update_sqme, ?update_weight)

- \$rescan_input_format   (default: "raw")
  String variable that allows to set the event format of the event file that is to be rescanned by the ($\rightarrow$) rescan command.

- resonance_background_factor   (default: 1.00000E+00)
  The real variable resonance_background_factor controls resonance insertion if a resonance history applies to a particular event. In determining whether event kinematics qualifies as resonant or non-resonant, the non-resonant probability is multiplied by this factor Setting the factor to zero removes the background configuration as long as the kinematics qualifies as on-shell as qualified by resonance_on_shell_limit.

- ?resonance_history   (default: false)
  The logical variable ?resonance_history = true/false specifies whether during a simulation pass, the event generator should try to reconstruct intermediate resonances. If activated, appropriate resonant subprocess matrix element code will be automatically generated.

- resonance_on_shell_limit   (default: 4.00000E+00)
  The real variable resonance_on_shell_limit = <num> specifies the maximum relative distance from a resonance peak, such that the kinematical configuration can still be considered on-shell. This is relevant only if ?resonance_history = true.

- `resonance_on_shell_turnoff`   (default: 0.00000E+00)
  The real variable `resonance_on_shell_turnoff = <num>`, if positive, controls the smooth transition from resonance-like to background-like events. The relative strength of a resonance is reduced by a Gaussian with width given by this variable. In any case, events are treated as background-like when the off-shellness is greater than `resonance_on_shell_limit`. All of this applies only if `?resonance_history = true`.

- `$resonances_exclude_particles`   (default: `"default"`)
  Accepts a string of particle names. These particles will be ignored when the resonance histories are generated. If `$fks_mapping_type` is not `"resonances"`, this option does nothing.

- `$restrictions`   (default: `""`)
  This is an optional argument for process definitions for the matrix element method `"omega"`. Using the following construction, it defines a string variable, `process`
  `<process_name> = <particle1>, <particle2> => <particle3>, <particle4>, ...`
  `{ $restrictions = "<restriction_def>" }`. The string argument `<restriction_def>` is directly transferred during the code generation to the ME generator O'Mega. It has to be of the form `n1 + n2 + ...  ~ <particle (list)>`, where `n1` and so on are the numbers of the particles above in the process definition. The tilde specifies a certain intermediate state to be equal to the particle(s) in `particle (list)`. An example is `process eemm_z = e1, E1 => e2, E2 { $restrictions = "1+2 ~ Z" }` restricts the code to be generated for the process $e^- e^+ \rightarrow \mu^- \mu^+$ to the s-channel Z-boson exchange. For more details see Sec. 9.3 (cf. also `process`)

- `$rng_method`   (default: `"tao"`)
  String variable that allows to set the method for the random number generation. Default is Donald Knuth' RNG method `TAO`.

- `$run_id`   (default: `""`)
  String variable `$run_id = "<id>"` that allows to set a special ID for a particular process run, e.g. in a scan. The run ID is then attached to the process log file:
  `<proc_name>_<proc_comp>.<id>.log`, the VAMP grid file:
  `<proc_name>_<proc_comp>.<id>.vg`, and the phase space file:
  `<proc_name>_<proc_comp>.<id>.phs`. The run ID string distinguishes among several runs for the same process. It identifies process instances with respect to adapted integration grids and similar run-specific data. The run ID is kept when copying processes for creating instances, however, so it does not distinguish event samples. (cf. also `$job_id`, `$compile_workspace`

- `safety_factor`   (default: 1.00000E+00)
  This real variable `safety_factor = <num>` reduces the acceptance probability for unweighting. If greater than one, excess events become less likely, but the reweighting efficiency also drops. (cf. `simulate`, `?unweighted`)

- **$sample**   (default: `""`)
  String variable to set the (base) name of the event output format, e.g. `$sample = "foo"` will result in an intrinsic binary format event file `foo.evx`. (cf. also `sample_format`, `simulate`, `hepevt`, `ascii`, `athena`, `debug`, `long`, `short`, `hepmc`, `lhef`, `lha`, `stdhep`, `stdhep_up`, `$sample_normalization`, `?sample_pacify`, `sample_max_tries`)

- **sample_max_tries**   (default: `10000`)
  Integer variable that sets the maximal number of tries for generating a single event. The event might be vetoed because of a very low unweighting efficiency, errors in the event transforms like decays, shower, matching, hadronization etc. (cf. also `simulate`, `$sample`, `sample_format`, `?sample_pacify`, `$sample_normalization`, `sample_split_n_evt`, `sample_split_n_kbytes`)

- **$sample_normalization**   (default: `"auto"`)
  String variable that allows to set the normalization of generated events. There are four options: option `"1"` (events normalized to one), `"1/n"` (sum of all events in a sample normalized to one), `"sigma"` (events normalized to the cross section of the process), and `"sigma/n"` (sum of all events normalized to the cross section). The default is `"auto"` where unweighted events are normalized to one, and weighted ones to the cross section. (cf. also `simulate`, `$sample`, `sample_format`, `?sample_pacify`, `sample_max_tries`, `sample_split_n_evt`, `sample_split_n_kbytes`)

- **?sample_pacify**   (default: `false`)
  Flag, mainly for debugging purposes: suppresses numerical noise in the output of a simulation. (cf. also `simulate`, `$sample`, `sample_format`, `$sample_normalization`, `sample_max_tries`, `sample_split_n_evt`, `sample_split_n_kbytes`)

- **?sample_select**   (default: `true`)
  Logical that determines whether a selection should be applied to the output event format or not. If set to `false` a selection is only considered for the evaluation of observables. (cf. `select`, `selection`, `analysis`)

- **sample_split_index**   (default: `0`)
  Integer number that gives the starting index `sample_split_index = <split_index>` for the numbering of event samples `<proc_name>.<split_index>.<evt_extension>` split by the `sample_split_n_evt = <num>`. The index runs from `<split_index>` to `<split_index>` + `<num>`. (cf. also `simulate`, `$sample`, `sample_format`, `$sample_normalization`, `sample_max_tries`, `?sample_pacify`)

- **sample_split_n_evt**   (default: `0`)
  When generating events, this integer parameter `sample_split_n_evt = <num>` gives the number `<num>` of breakpoints in the event files, i.e. it splits the event files into `<num>` + 1 parts. The parts are denoted by `<proc_name>.<split_index>.<evt_extension>`. Here, `<split_index>` is an integer running from 0 to `<num>`. The start can be reset by ($\rightarrow$)

sample_split_index. (cf. also simulate, $sample, sample_format, sample_max_tries, $sample_normalization, ?sample_pacify, sample_split_n_kbytes)

- sample_split_n_kbytes   (default: 0)
  When generating events, this integer parameter sample_split_n_kbytes = <num> limits the file size of event files. Whenever an event file has exceeded this size, counted in kilobytes, the following events will be written to a new file. The naming conventions are the same as for sample_split_n_evt. (cf. also simulate, $sample, sample_format, sample_max_tries, $sample_normalization, ?sample_pacify)

- seed   (default: 0)
  Integer variable seed = <num> that allows to set a specific random seed num. If not set, WHIZARD takes the time from the system clock to determine the random seed.

- $select_alpha_regions   (default: "")
  Fixes the α_r in the real subtraction as well as the DGLAP component. Allows for testing in a list of selected singular regions.

- ?sf_allow_s_mapping   (default: true)
  Flag that determines whether special mappings for processes with structure functions and s-channel resonances are applied, e.g. Drell-Yan at hadron colliders, or Z production at linear colliders with beamstrahlung and ISR.

- ?sf_trace   (default: false)
  Debug flag that writes out detailed information about the structure function setup into the file <proc_name>_sftrace.dat. This file name can be changed with ($\rightarrow$) $sf_trace_file.

- $sf_trace_file   (default: "")
  $sf_trace_file = "<file_name>" allows to change the detailed structure function information switched on by the debug flag ($\rightarrow$) ?sf_trace into a different file <file_name> than the default <proc_name>_sftrace.dat.

- $shower_method   (default: "WHIZARD")
  String variable that allows to specify which parton shower is being used, the default, "WHIZARD", is one of the in-house showers of WHIZARD. Other possibilities at the moment are only "PYTHIA6".

- ?shower_verbose   (default: false)
  Flag to switch on verbose messages when using shower and/or hadronization. (cf. also ?allow_shower, ?ps_ ..., $ps_ ..., ?mlm_ ...,

- ?slha_read_decays   (default: false)
  Flag which decides whether WHIZARD reads in the widths and branching ratios from the DCINFO common block of the SUSY Les Houches Accord files. (cf. also read_slha, write_slha, ?slha_read_spectrum, ?slha_read_input)

- `?slha_read_input`   (default: `true`)
  Flag which decides whether `WHIZARD` reads in the SM and parameter information from the `SMINPUTS` and `MINPAR` common blocks of the SUSY Les Houches Accord files. (cf. also `read_slha`, `write_slha`, `?slha_read_spectrum`, `?slha_read_decays`)

- `?slha_read_spectrum`   (default: `true`)
  Flag which decides whether `WHIZARD` reads in the whole spectrum and mixing angle information from the common blocks of the SUSY Les Houches Accord files. (cf. also `read_slha`, `write_slha`, `?slha_read_decays`, `?slha_read_input`)

- `sqrts`
  Real variable in order to set the center-of-mass energy for the collisions (collider energy $\sqrt{s}$, not hard interaction energy $\sqrt{\hat{s}}$): `sqrts = <num> [ <phys_unit> ]`. The physical unit can be one of the following `eV`, `keV`, `MeV`, `GeV`, and `TeV`. If absent, `WHIZARD` takes `GeV` as its standard unit. Note that this variable is absolutely mandatory for integration and simulation of scattering processes.

- `?stratified`   (default: `true`)
  Flag that switches between stratified and importance sampling for the `VAMP` integration method.

- `$symbol`
  Settings for `WHIZARD`'s internal graphics output: `$symbol = "<LaTeX_code>"` is a string variable for the symbols that should be used for plotting data points. (cf. also `$obs_label`, `?normalize_bins`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_width_mm`, `graph_height_mm`, `?y_log`, `?x_log`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_fg`, `$gmlcode_bg`, `?draw_base`, `?draw_piecewise`, `?fill_curve`, `?draw_histogram`, `?draw_curve`, `?draw_errors`, `$fill_options`, `$draw_options`, `$err_options`, `?draw_symbols`)

- `?test_anti_coll_limit`   (default: `false`)
  Sets the fixed values $\tilde{\xi} = 0.5$ and $y = -0.9999999$ as radiation variables. This way, only anti-collinear, but non-soft phase space points are generated, which allows for testing subtraction in this region. Can be combined with `?test_soft_limit` to probe soft-collinear regions.

- `?test_coll_limit`   (default: `false`)
  Sets the fixed values $\tilde{\xi} = 0.5$ and $y = 0.9999999$ as radiation variables. This way, only collinear, but non-soft phase space points are generated, which allows for testing subtraction in this region. Can be combined with `?test_soft_limit` to probe soft-collinear regions.

- `?test_soft_limit`   (default: `false`)
  Sets the fixed values $\tilde{\xi} = 0.00001$ and $y = 0.5$ as radiation variables. This way, only soft, but non-collinear phase space points are generated, which allows for testing subtraction in this region.

- `threshold_calls`   (default: 10)
  This integer variable gives a limit for the number of calls in a given channel which acts as a lower threshold for the channel weight. If the number of calls in that channel falls below this threshold, the weight is not lowered further but kept at this threshold. (cf. also `channel_weights_power`)

- `$title`   (default: `""`)
  This string variable sets the title of a plot in a WHIZARD analysis setup, e.g. a histogram or an observable. The syntax is `$title = "<your title>"`. This title appears as a section header in the analysis file, but not in the screen output of the analysis. (cf. also `n_bins`, `?normalize_bins`, `$obs_unit`, `$description`, `$x_label`, `$y_label`, `?y_log`, `?x_log`, `graph_width_mm`, `graph_height_mm`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?fill_curve`, `?draw_piecewise`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)

- `tolerance`   (default: 0.00000E+00)
  Real variable that defines the absolute tolerance with which the (logical) function `expect` accepts equality or inequality: `tolerance = <num>`. This can e.g. be used for cross-section tests and backwards compatibility checks. (cf. also `expect`)

- `undefined_jet_algorithm`   (fixed value: 999)
  This is just a place holder for any kind of jet jet algorithm that is not further specified. (cf. also `kt_algorithm`, `cambridge_for_passive_algorithm`, `genkt_[for_passive_]algorithm`, `ee_[gen]kt_algorithm`, `jet_r`, `plugin_algorithm`)

- `?unweighted`   (default: `true`)
  Flag that distinguishes between unweighted and weighted event generation. (cf. also `simulate`, `n_events`, `luminosity`, `event_index_offset`)

- `?update_event`   (default: `false`)
  Flag that decides whether the events in an event file should be rebuilt from the hard process when reading event/rescanning files into WHIZARD. (cf. `rescan`, `?recover_beams`, `?update_sqme`, `?update_weight`)

- `?update_sqme`   (default: `false`)
  Flag that decides whether the squared matrix element in an event file should be updated/recalculated when reading event/rescanning files into WHIZARD. (cf. `rescan`, `?recover_beams`, `?update_event`, `?update_weight`)

- `?update_weight`   (default: `false`)
  Flag that decides whether the weights in an event file should be updated/recalculated when reading event/rescanning files into WHIZARD. (cf. `rescan`, `?recover_beams`, `?update_event`, `?update_sqme`)

- **?use_alphas_from_file**   (default: `false`)
  Flag that decides whether the current α_s definition should be used when recalculating matrix elements for events read from file, or the value that is stored in the file for that event. (cf. `rescan`, `?update_sqme`, `?use_scale_from_file`)

- **?use_scale_from_file**   (default: `false`)
  Flag that decides whether the current energy-scale expression should be used when recalculating matrix elements for events read from file, or the value that is stored in the file for that event. (cf. `rescan`, `?update_sqme`, `?use_alphas_from_file`)

- **?use_vamp_equivalences**   (default: `true`)
  Flag that decides whether equivalence relations (symmetries) between different integration channels are used by the `VAMP` integrator.

- **vamp_grid_checkpoint**   (default: `1`)
  Integer parameter for setting checkpoints to save the current state of the grids and the results so far of the integration. Allowed are all positive integer. Zero values corresponds to a checkpoint after each integration pass, a one value to a checkpoint after each iteration (default) and an N value correspond to a checkpoint after N iterations or after each pass, respectively.

- **$vamp_grid_format**   (default: `"ascii"`)
  Character string that tells `WHIZARD` the file format for `vamp2` to use for writing and reading the configuration for the multi-channel integration setup and the `VAMP2` (only) grid data. The values can be `ascii` for a single human-readable grid file with ending `.vg2` or `binary` for two files, a human-readable header file with ending `.vg2` and binary file with ending `.vgx2` storing the grid data.The main purpose of the binary format is to perform faster I/O, e.g. for HPC runs.`WHIZARD` can convert between the different file formats automatically.

- **?vamp_history_channels**   (default: `false`)
  Flag that decides whether the history of the grid adaptation of the `VAMP` integrator for every single channel are written into the process logfiles. Only for debugging purposes. (cf. also `?vamp_history_global_verbose`, `?vamp_history_global`, `?vamp_verbose`, `?vamp_history_channels_verbose`)

- **?vamp_history_channels_verbose**   (default: `false`)
  Flag that decides whether the history of the grid adaptation of the `VAMP` integrator for every single channel are written into the process logfiles in an extended version. Only for debugging purposes. (cf. also `?vamp_history_global`, `?vamp_history_channels`, `?vamp_verbose`, `?vamp_history_global_verbose`)

- **?vamp_history_global**   (default: `true`)
  Flag that decides whether the global history of the grid adaptation of the `VAMP` integrator are written into the process logfiles. (cf. also `?vamp_history_global_verbose`, `?vamp_history_channels`, `?vamp_history_channels_verbose`, `?vamp_verbose`)

- `?vamp_history_global_verbose`   (default: `false`)
  Flag that decides whether the global history of the grid adaptation of the `VAMP` integrator are written into the process logfiles in an extended version. Only for debugging purposes. (cf. also `?vamp_history_global`, `?vamp_history_channels`, `?vamp_verbose`, `?vamp_history_channels_verbose`)

- `$vamp_parallel_method`   (default: `"simple"`)
  Character string that tells `WHIZARD` the parallel method to use for parallel integration within `vamp2`. (i) `simple` (default) is a local work sharing approach without the need of communication between all workers except for the communication during result collection. (ii) `load` is a global queue approach where the master worker acts as agovernor listening and providing work for each worker. The queue is filled and assigned with workers a-priori with respect to the assumed computational impact of each channel.Both approaches use the same mechanism for result collection using non-blocking communication allowing for a efficient usage of the computing resources.

- `?vamp_verbose`   (default: `false`)
  Flag that sets the chattiness of the `VAMP` integrator. If set, not only errors, but also all warnings and messages will be written out (not the default). (cf. also `?vamp_history_global`, `?vamp_history_global_verbose`, `?vamp_history_channels`, `?vamp_history_channels_verbose`)

- `?virtual_collinear_resonance_aware`   (default: `true`)
  This flag allows to switch between two different implementations of the collinear subtraction in the resonance-aware FKS setup.

- `$virtual_selection`   (default: `"Full"`)
  String variable to select either the full or only parts of the virtual components of an NLO calculation. Possible modes are `"Full"`, `"OLP"` and `"Subtraction."`. Mainly for debugging purposes.

- `?vis_channels`   (default: `false`)
  Optional logical argument for the `integrate` command that demands `WHIZARD` to generate a PDF or postscript output showing the classification of the found phase space channels (if the phase space method `wood` has been used) according to their properties: `integrate (foo) { iterations=3:10000 ?vis_channels = true }`. The default is `false`. (cf. also `integrate`, `?vis_history`)

- `?vis_diags`   (default: `false`)
  Logical variable that allows to give out a Postscript or PDF file for the Feynman diagrams for a `O'Mega` process. (cf. `?vis_diags_color`).

- `?vis_diags_color`   (default: `false`)
  Same as `?vis_diags`, but switches on color flow instead of Feynman diagram generation. (cf. `?vis_diags`).

- **?vis_fks_regions**   (default: `false`)
  Logical variable that, if set to `true`, generates LaTeX code and executes it into a PDF to produce a table of all singular FKS regions and their flavor structures. The default is `false`.

- **?vis_history**   (default: `false`)
  Optional logical argument for the `integrate` command that demands WHIZARD to generate a PDF or postscript output showing the adaptation history of the Monte-Carlo integration of the process under consideration. (cf. also `integrate`, `?vis_channels`)

- **?write_raw**   (default: `true`)
  Flag to write out events in WHIZARD's internal binary format. (cf. `simulate`, `?read_raw`, `sample_format`, `$sample`)

- **$x_label**   (default: `""`)
  String variable, `$x_label = "<LaTeX code>"`, that sets the x axis label in a plot or histogram in a WHIZARD analysis. (cf. also `analysis`, `n_bins`, `?normalize_bins`, `$obs_unit`, `$y_label`, `?y_log`, `?x_log`, `graph_width_mm`, `graph_height_mm`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?fill_curve`, `?draw_piecewise`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)

- **?x_log**   (default: `false`)
  Settings for WHIZARD's internal graphics output: flag that makes the x axis logarithmic. (cf. also `?normalize_bins`, `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_height_mm`, `graph_width_mm`, `?y_log`, `x_min`, `x_max`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_histogram`, `?draw_base`, `?draw_piecewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- **x_max**
  Settings for WHIZARD's internal graphics output: real parameter that sets the upper limit of the x axis plotting or histogram interval. (cf. also `?normalize_bins`, `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_height_mm`, `?y_log`, `?x_log`, `graph_width_mm`, `x_min`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?draw_piecewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- **x_min**
  Settings for WHIZARD's internal graphics output: real parameter that sets the lower limit of the x axis plotting or histogram interval. (cf. also `?normalize_bins`, `$obs_label`, `$obs_unit`, `$title`, `$description`, `$x_label`, `$y_label`, `graph_height_mm`, `?y_log`, `?x_log`, `graph_width_mm`, `x_max`, `y_min`, `y_max`, `$gmlcode_bg`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?draw_piecewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- $y_label   (default: "")
  String variable, $y_label = "<LaTeX_code>", that sets the y axis label in a plot or histogram in a WHIZARD analysis. (cf. also analysis, n_bins, ?normalize_bins, $obs_unit, ?y_log, ?x_log, graph_width_mm, graph_height_mm, x_min, x_max, y_min, y_max, $gmlcode_bg, $gmlcode_fg, ?draw_base, ?draw_histogram, ?fill_curve, ?draw_piecewise, ?draw_curve, ?draw_errors, $symbol, ?draw_symbols, $fill_options, $draw_options, $err_options)

- ?y_log   (default: false)
  Settings for WHIZARD's internal graphics output: flag that makes the y axis logarithmic. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_height_mm, graph_width_mm, ?y_log, x_min, x_max, y_min, y_max, $gmlcode_bg, $gmlcode_fg, ?draw_histogram, ?draw_base, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- y_max
  Settings for WHIZARD's internal graphics output: real parameter that sets the upper limit of the y axis plotting or histogram interval. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_height_mm, ?y_log, ?x_log, graph_width_mm, x_max, x_min, y_max, $gmlcode_bg, $gmlcode_fg, ?draw_base, ?draw_histogram, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

- y_min
  Settings for WHIZARD's internal graphics output: real parameter that sets the lower limit of the y axis plotting or histogram interval. (cf. also ?normalize_bins, $obs_label, $obs_unit, $title, $description, $x_label, $y_label, graph_height_mm, ?y_log, ?x_log, graph_width_mm, x_max, y_max, x_min, $gmlcode_bg, $gmlcode_fg, ?draw_base, ?draw_histogram, ?draw_piecewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options, $draw_options, $err_options, $symbol)

Acknowledgements

Bibliography

[1] T. Sjöstrand, Comput. Phys. Commun. **82** (1994) 74.

[2] A. Pukhov, *et al.*, Preprint INP MSU 98-41/542, `hep-ph/9908288`.

[3] T. Stelzer and W.F. Long, Comput. Phys. Commun. **81** (1994) 357.

[4] T. Ohl, *Proceedings of the Seventh International Workshop on Advanced Computing and Analysis Technics in Physics Research*, ACAT 2000, Fermilab, October 2000, IKDA-2000-30, `hep-ph/0011243`; M. Moretti, Th. Ohl, and J. Reuter, LC-TOOL-2001-040

[5] T. Ohl, *Vegas revisited: Adaptive Monte Carlo integration beyond factorization*, Comput. Phys. Commun. **120**, 13 (1999) [arXiv:hep-ph/9806432].

[6] T. Ohl, *CIRCE version 1.0: Beam spectra for simulating linear collider physics*, Comput. Phys. Commun. **101**, 269 (1997) [arXiv:hep-ph/9607454].

[7] V. N. Gribov and L. N. Lipatov, *e+ e- pair annihilation and deep inelastic e p scattering in perturbation theory*, Sov. J. Nucl. Phys. **15**, 675 (1972) [Yad. Fiz. **15**, 1218 (1972)].

[8] E. A. Kuraev and V. S. Fadin, *On Radiative Corrections to e+ e- Single Photon Annihilation at High-Energy*, Sov. J. Nucl. Phys. **41**, 466 (1985) [Yad. Fiz. **41**, 733 (1985)].

[9] M. Skrzypek and S. Jadach, *Exact and approximate solutions for the electron nonsinglet structure function in QED*, Z. Phys. C **49**, 577 (1991).

[10] D. Schulte, *Beam-beam simulations with Guinea-Pig*, eConf C **980914**, 127 (1998).

[11] D. Schulte, *Beam-beam simulations with GUINEA-PIG*, CERN-PS-99-014-LP.

[12] D. Schulte, M. Alabau, P. Bambade, O. Dadoun, G. Le Meur, C. Rimbault and F. Touze, *GUINEA PIG++ : An Upgraded Version of the Linear Collider Beam Beam Interaction Simulation Code GUINEA PIG*, Conf. Proc. C **070625**, 2728 (2007).

[13] T. Behnke, J. E. Brau, B. Foster, J. Fuster, M. Harrison, J. M. Paterson, M. Peskin and M. Stanitzki *et al.*, *The International Linear Collider Technical Design Report - Volume 1: Executive Summary*, arXiv:1306.6327 [physics.acc-ph].

[14] H. Baer, T. Barklow, K. Fujii, Y. Gao, A. Hoang, S. Kanemura, J. List and H. E. Logan *et al.*, *The International Linear Collider Technical Design Report - Volume 2: Physics*, arXiv:1306.6352 [hep-ph].

[15] C. Adolphsen, M. Barone, B. Barish, K. Buesser, P. Burrows, J. Carwardine, J. Clark and Hélèn. M. Durand *et al.*, *The International Linear Collider Technical Design Report - Volume 3.I: Accelerator & in the Technical Design Phase*, arXiv:1306.6353 [physics.acc-ph].

[16] C. Adolphsen, M. Barone, B. Barish, K. Buesser, P. Burrows, J. Carwardine, J. Clark and Hélèn. M. Durand *et al.*, *The International Linear Collider Technical Design Report - Volume 3.II: Accelerator Baseline Design*, arXiv:1306.6328 [physics.acc-ph].

[17] T. Behnke, J. E. Brau, P. N. Burrows, J. Fuster, M. Peskin, M. Stanitzki, Y. Sugimoto and S. Yamada *et al.*, arXiv:1306.6329 [physics.ins-det].

[18] M. Aicheler, P. Burrows, M. Draper, T. Garvey, P. Lebrun, K. Peach and N. Phinney *et al.*, *A Multi-TeV Linear Collider Based on CLIC Technology : CLIC Conceptual Design Report*, CERN-2012-007.

[19] P. Lebrun, L. Linssen, A. Lucaci-Timoce, D. Schulte, F. Simon, S. Stapnes, N. Toge and H. Weerts *et al.*, *The CLIC Programme: Towards a Staged e+e- Linear Collider Exploring the Terascale : CLIC Conceptual Design Report*, arXiv:1209.2543 [physics.ins-det].

[20] L. Linssen, A. Miyamoto, M. Stanitzki and H. Weerts, *Physics and Detectors at CLIC: CLIC Conceptual Design Report*, arXiv:1202.5940 [physics.ins-det].

[21] C. F. von Weizsäcker, *Radiation emitted in collisions of very fast electrons*, Z. Phys. **88**, 612 (1934).

[22] E. J. Williams, *Nature of the high-energy particles of penetrating radiation and status of ionization and radiation formulae*, Phys. Rev. **45**, 729 (1934).

[23] V. M. Budnev, I. F. Ginzburg, G. V. Meledin and V. G. Serbo, *The Two photon particle production mechanism. Physical problems. Applications. Equivalent photon approximation*, Phys. Rept. **15** (1974) 181.

[24] I. F. Ginzburg, G. L. Kotkin, V. G. Serbo and V. I. Telnov, *Colliding gamma e and gamma gamma Beams Based on the Single Pass Accelerators (of Vlepp Type)*, Nucl. Instrum. Meth. **205**, 47 (1983).

[25] V. I. Telnov, *Problems of Obtaining $\gamma\gamma$ and $\gamma\epsilon$ Colliding Beams at Linear Colliders*, Nucl. Instrum. Meth. A **294**, 72 (1990).

[26] V. I. Telnov, *Principles of photon colliders*, Nucl. Instrum. Meth. A **355**, 3 (1995).

[27] J. A. Aguilar-Saavedra *et al.* [ECFA/DESY LC Physics Working Group Collaboration], *TESLA: The Superconducting electron positron linear collider with an integrated x-ray laser laboratory. Technical design report. Part 3. Physics at an e+ e- linear collider*, hep-ph/0106315.

[28] F. Richard, J. R. Schneider, D. Trines and A. Wagner, *TESLA, The Superconducting Electron Positron Linear Collider with an Integrated X-ray Laser Laboratory, Technical Design Report Part 1 : Executive Summary*, hep-ph/0106314.

[29] V. V. Sudakov, Sov. Phys. JETP **3**, 65 (1956) [Zh. Eksp. Teor. Fiz. **30**, 87 (1956)].
[30]

[30] T. Sjostrand, Phys. Lett. **157B**, 321 (1985). doi:10.1016/0370-2693(85)90674-4

[31] T. Sjostrand, S. Mrenna and P. Z. Skands, JHEP **0605**, 026 (2006) doi:10.1088/1126-6708/2006/05/026 [hep-ph/0603175].

[32] T. Ohl, *Vegas revisited: Adaptive Monte Carlo integration beyond factorization*, Comput. Phys. Commun. **120**, 13 (1999) [hep-ph/9806432].

[33] G. P. Lepage, CLNS-80/447.

[34] A. Djouadi, J. Kalinowski, M. Spira, Comput. Phys. Commun. **108** (1998) 56-74.

[35] M. Beyer, W. Kilian, P. Krstonošic, K. Mönig, J. Reuter, E. Schmidt and H. Schröder, *Determination of New Electroweak Parameters at the ILC - Sensitivity to New Physics*, Eur. Phys. J. C **48**, 353 (2006) [hep-ph/0604048].

[36] A. Alboteanu, W. Kilian and J. Reuter, *Resonances and Unitarity in Weak Boson Scattering at the LHC*, JHEP **0811**, 010 (2008) [arXiv:0806.4145 [hep-ph]].

[37] T. Binoth *et al.*, Comput. Phys. Commun. **181**, 1612 (2010) doi:10.1016/j.cpc.2010.05.016 [arXiv:1001.1307 [hep-ph]].

[38] S. Alioli *et al.*, Comput. Phys. Commun. **185**, 560 (2014) doi:10.1016/j.cpc.2013.10.020 [arXiv:1308.3462 [hep-ph]].

[39] C. Speckner, *LHC Phenomenology of the Three-Site Higgsless Model*, PhD thesis, arXiv:1011.1851 [hep-ph].

[40] R. S. Chivukula, B. Coleppa, S. Di Chiara, E. H. Simmons, H. -J. He, M. Kurachi and M. Tanabashi, *A Three Site Higgsless Model*, Phys. Rev. D **74**, 075011 (2006) [hep-ph/0607124].

[41] R. S. Chivukula, E. H. Simmons, H. -J. He, M. Kurachi and M. Tanabashi, *Ideal fermion delocalization in Higgsless models*, Phys. Rev. D **72**, 015008 (2005) [hep-ph/0504114].

[42] T. Ohl and C. Speckner, *Production of Almost Fermiophobic Gauge Bosons in the Minimal Higgsless Model at the LHC*, Phys. Rev. D **78**, 095008 (2008) [arXiv:0809.0023 [hep-ph]].

[43] T. Ohl and J. Reuter, *Clockwork SUSY: Supersymmetric Ward and Slavnov-Taylor identities at work in Green's functions and scattering amplitudes*, Eur. Phys. J. C **30**, 525 (2003) [hep-th/0212224].

[44] J. Reuter and F. Braam, *The NMSSM implementation in WHIZARD*, AIP Conf. Proc. **1200**, 470 (2010) [arXiv:0909.3059 [hep-ph]].

[45] J. Kalinowski, W. Kilian, J. Reuter, T. Robens and K. Rolbiecki, *Pinning down the Invisible Sneutrino*, JHEP **0810**, 090 (2008) [arXiv:0809.3997 [hep-ph]].

[46] T. Robens, J. Kalinowski, K. Rolbiecki, W. Kilian and J. Reuter, *(N)LO Simulation of Chargino Production and Decay*, Acta Phys. Polon. B **39**, 1705 (2008) [arXiv:0803.4161 [hep-ph]].

[47] W. Kilian, D. Rainwater and J. Reuter, *Pseudo-axions in little Higgs models*, Phys. Rev. D **71**, 015008 (2005) [hep-ph/0411213].

[48] W. Kilian, D. Rainwater and J. Reuter, *Distinguishing little-Higgs product and simple group models at the LHC and ILC*, Phys. Rev. D **74**, 095003 (2006) [Erratum-ibid. D **74**, 099905 (2006)] [hep-ph/0609119].

[49] T. Ohl and J. Reuter, *Testing the noncommutative standard model at a future photon collider*, Phys. Rev. D **70**, 076007 (2004) [hep-ph/0406098].

[50] T. Ohl and C. Speckner, *The Noncommutative Standard Model and Polarization in Charged Gauge Boson Production at the LHC*, Phys. Rev. D **82**, 116011 (2010) [arXiv:1008.4710 [hep-ph]].

[51] E. Boos *et al.*, *Generic user process interface for event generators*, arXiv:hep-ph/0109068.

[52] P. Z. Skands *et al.*, *SUSY Les Houches Accord: Interfacing SUSY Spectrum Calculators, Decay Packages, and Event Generators*, JHEP **0407**, 036 (2004) [arXiv:hep-ph/0311123].

[53] J. A. Aguilar-Saavedra, A. Ali, B. C. Allanach, R. L. Arnowitt, H. A. Baer, J. A. Bagger, C. Balazs and V. D. Barger *et al.*, *Supersymmetry parameter analysis: SPA convention and project*, Eur. Phys. J. C **46**, 43 (2006) [hep-ph/0511344].

[54] B. C. Allanach, C. Balazs, G. Belanger, M. Bernhardt, F. Boudjema, D. Choudhury, K. Desch and U. Ellwanger *et al.*, Comput. Phys. Commun. **180**, 8 (2009) [arXiv:0801.0045 [hep-ph]].

[55] J. Alwall *et al.*, *A standard format for Les Houches event files*, Comput. Phys. Commun. **176**, 300 (2007) [arXiv:hep-ph/0609017].

[56] K. Hagiwara *et al.*, *Supersymmetry simulations with off-shell effects for LHC and ILC*, Phys. Rev. D **73**, 055005 (2006) [arXiv:hep-ph/0512260].

[57] B. C. Allanach *et al.*, *The Snowmass points and slopes: Benchmarks for SUSY searches*, in *Proc. of the APS/DPF/DPB Summer Study on the Future of Particle Physics (Snowmass 2001)* ed. N. Graf, Eur. Phys. J. C **25** (2002) 113 [eConf **C010630** (2001) P125] [arXiv:hep-ph/0202233].

[58] M.E. Peskin, D.V.Schroeder, *An Introduction to Quantum Field Theory*, Addison-Wesley Publishing Co., 1995.

[59] U. Klein, O. Fischer, *private communications.*

[60] L. Garren, *StdHep, Monte Carlo Standardization at FNAL*, Fermilab CS-doc-903, http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=903

[61] S. Frixione, Phys. Lett. B **429**, 369 (1998) doi:10.1016/S0370-2693(98)00454-7 [hep-ph/9801442].

[62] W. Giele *et al.*, *The QCD / SM working group: Summary report*, arXiv:hep-ph/0204316; M. R. Whalley, D. Bourilkov and R. C. Group, *The Les Houches Accord PDFs (LHAPDF) and Lhaglue*, arXiv:hep-ph/0508110; D. Bourilkov, R. C. Group and M. R. Whalley, *LHAPDF: PDF use from the Tevatron to the LHC*, arXiv:hep-ph/0605240.

[63] M. Dobbs and J. B. Hansen, *The HepMC C++ Monte Carlo event record for High Energy Physics*, Comput. Phys. Commun. **134**, 41 (2001).

[64] E. Boos *et al.* [CompHEP Collaboration], Nucl. Instrum. Meth. A **534**, 250 (2004) [hep-ph/0403113].

[65] J. Pumplin, D. R. Stump, J. Huston *et al.*, *New generation of parton distributions with uncertainties from global QCD analysis*, JHEP **0207**, 012 (2002). [hep-ph/0201195].

[66] A. D. Martin, R. G. Roberts, W. J. Stirling *et al.*, *Parton distributions incorporating QED contributions*, Eur. Phys. J. **C39**, 155-161 (2005). [hep-ph/0411040].

[67] A. D. Martin, W. J. Stirling, R. S. Thorne *et al.*, *Parton distributions for the LHC*, Eur. Phys. J. **C63**, 189-285 (2009). [arXiv:0901.0002 [hep-ph]].

[68] H. L. Lai, M. Guzzi, J. Huston, Z. Li, P. M. Nadolsky, J. Pumplin and C. P. Yuan, *New parton distributions for collider physics*, Phys. Rev. D **82**, 074024 (2010) [arXiv:1007.2241 [hep-ph]].

[69] J. F. Owens, A. Accardi and W. Melnitchouk, *Global parton distributions with nuclear and finite-Q^2 corrections*, Phys. Rev. D **87**, no. 9, 094012 (2013) [arXiv:1212.1702 [hep-ph]].

[70] A. Accardi, L. T. Brady, W. Melnitchouk, J. F. Owens and N. Sato, arXiv:1602.03154 [hep-ph].

[71] L. A. Harland-Lang, A. D. Martin, P. Motylinski and R. S. Thorne, arXiv:1412.3989 [hep-ph].

[72] S. Dulat *et al.*, arXiv:1506.07443 [hep-ph].

[73] G. P. Salam and J. Rojo, *A Higher Order Perturbative Parton Evolution Toolkit (HOPPET)*, Comput. Phys. Commun. **180**, 120 (2009) [arXiv:0804.3755 [hep-ph]].

[74] W. Kilian, J. Reuter, S. Schmidt and D. Wiesler, *An Analytic Initial-State Parton Shower*, JHEP **1204** (2012) 013 [arXiv:1112.1039 [hep-ph]].

[75] F. Staub, *Sarah*, arXiv:0806.0538 [hep-ph].

[76] F. Staub, *From Superpotential to Model Files for FeynArts and CalcHep/CompHep*, Comput. Phys. Commun. **181**, 1077 (2010) [arXiv:0909.2863 [hep-ph]].

[77] F. Staub, *Automatic Calculation of supersymmetric Renormalization Group Equations and Self Energies*, Comput. Phys. Commun. **182**, 808 (2011) [arXiv:1002.0840 [hep-ph]].

[78] F. Staub, *SARAH 3.2: Dirac Gauginos, UFO output, and more*, Computer Physics Communications **184**, pp. 1792 (2013) [Comput. Phys. Commun. **184**, 1792 (2013)] [arXiv:1207.0906 [hep-ph]].

[79] F. Staub, *SARAH 4: A tool for (not only SUSY) model builders*, Comput. Phys. Commun. **185**, 1773 (2014) [arXiv:1309.7223 [hep-ph]].

[80] `Mathematica` is a registered trademark of Wolfram Research, Inc., Champain, IL, USA.

[81] W. Porod, *SPheno, a program for calculating supersymmetric spectra, SUSY particle decays and SUSY particle production at e+ e- colliders*, Comput. Phys. Commun. **153**, 275 (2003) [hep-ph/0301101].

[82] W. Porod and F. Staub, *SPheno 3.1: Extensions including flavour, CP-phases and models beyond the MSSM*, Comput. Phys. Commun. **183**, 2458 (2012) [arXiv:1104.1573 [hep-ph]].

[83] F. Staub, T. Ohl, W. Porod and C. Speckner, Comput. Phys. Commun. **183**, 2165 (2012) [arXiv:1109.5147 [hep-ph]].

[84] N. D. Christensen and C. Duhr, *FeynRules - Feynman rules made easy*, Comput. Phys. Commun. **180**, 1614 (2009) [arXiv:0806.4194 [hep-ph]].

[85] N. D. Christensen, P. de Aquino, C. Degrande, C. Duhr, B. Fuks, M. Herquet, F. Maltoni and S. Schumann, *A Comprehensive approach to new physics simulations*, Eur. Phys. J. C **71**, 1541 (2011) [arXiv:0906.2474 [hep-ph]].

[86] C. Duhr and B. Fuks, Comput. Phys. Commun. **182**, 2404 (2011) [arXiv:1102.4191 [hep-ph]].

[87] N. D. Christensen, C. Duhr, B. Fuks, J. Reuter and C. Speckner, *Introducing an interface between WHIZARD and FeynRules*, Eur. Phys. J. C **72**, 1990 (2012) [arXiv:1010.3251 [hep-ph]].

[88] C. Degrande, C. Duhr, B. Fuks, D. Grellscheid, O. Mattelaer and T. Reiter, Comput. Phys. Commun. **183**, 1201 (2012) doi:10.1016/j.cpc.2012.01.022 [arXiv:1108.2040 [hep-ph]].

[89] T. Han, J. D. Lykken and R. -J. Zhang, *On Kaluza-Klein states from large extra dimensions*, Phys. Rev. D **59**, 105006 (1999) [hep-ph/9811350].

[90] B. Fuks, *Beyond the Minimal Supersymmetric Standard Model: from theory to phenomenology*, Int. J. Mod. Phys. A **27**, 1230007 (2012) [arXiv:1202.4769 [hep-ph]].

[91] H. -J. He, Y. -P. Kuang, Y. -H. Qi, B. Zhang, A. Belyaev, R. S. Chivukula, N. D. Christensen and A. Pukhov *et al.*, *CERN LHC Signatures of New Gauge Bosons in Minimal Higgsless Model*, Phys. Rev. D **78**, 031701 (2008) [arXiv:0708.2588 [hep-ph]].

[92] W. Kilian, J. Reuter and T. Robens, *NLO Event Generation for Chargino Production at the ILC*, Eur. Phys. J. C **48**, 389 (2006) [hep-ph/0607127].

[93] J. R. Andersen *et al.* [SM and NLO Multileg Working Group Collaboration], *Les Houches 2009: The SM and NLO Multileg Working Group: Summary report*, arXiv:1003.1241 [hep-ph].

[94] J. M. Butterworth, A. Arbey, L. Basso, S. Belov, A. Bharucha, F. Braam, A. Buckley and M. Campanelli *et al.*, *Les Houches 2009: The Tools and Monte Carlo working group Summary Report*, arXiv:1003.1643 [hep-ph], arXiv:1003.1643 [hep-ph].

[95] T. Binoth, N. Greiner, A. Guffanti, J. Reuter, J.-P. .Guillet and T. Reiter, *Next-to-leading order QCD corrections to pp –¿ b anti-b b anti-b + X at the LHC: the quark induced case*, Phys. Lett. B **685**, 293 (2010) [arXiv:0910.4379 [hep-ph]].

[96] N. Greiner, A. Guffanti, T. Reiter and J. Reuter, *NLO QCD corrections to the production of two bottom-antibottom pairs at the LHC* Phys. Rev. Lett. **107**, 102002 (2011) [arXiv:1105.3624 [hep-ph]].

[97] P. LéEcuyer, R. Simard, E. J. Chen, and W. D. Kelton, *An Object-Oriented Random-Number Package with Many Long Streams and Substreams*, Operations Research, vol. 50, no. 6, pp. 1073-1075, Dec. 2002.

[98] S. Plätzer, *RAMBO on diet*, [arXiv:1308.2922 [hep-ph]].

[99] R. Kleiss and W. J. Stirling, *Massive multiplicities and Monte Carlo*, Nucl. Phys. B **385**, 413 (1992). doi:10.1016/0550-3213(92)90107-M

[100] R. Kleiss, W. J. Stirling and S. D. Ellis, *A New Monte Carlo Treatment of Multiparticle Phase Space at High-energies*, Comput. Phys. Commun. **40** (1986) 359. doi:10.1016/0010-4655(86)90119-0

[101] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389**, 81-86 (1997) doi:10.1016/S0168-9002(97)00048-X

[102] A. Buckley, J. Butterworth, L. Lönnblad, D. Grellscheid, H. Hoeth, J. Monk, H. Schulz and F. Siegert, *Rivet user manual*, Comput. Phys. Commun. **184**, 2803-2819 (2013) doi:10.1016/j.cpc.2013.05.021 [arXiv:1003.0694 [hep-ph]].

[103] C. Bierlich, A. Buckley, J. Butterworth, C. H. Christensen, L. Corpe, D. Grellscheid, J. F. Grosse-Oetringhaus, C. Gutschow, P. Karczmarczyk, J. Klein, L. Lönnblad, C. S. Pollard, P. Richardson, H. Schulz and F. Siegert, *Robust Independent Validation of Experiment and Theory: Rivet version 3*, SciPost Phys. **8**, 026 (2020) doi:10.21468/SciPostPhys.8.2.026 [arXiv:1912.05451 [hep-ph]].

[104] J. de Favereau *et al.* [DELPHES 3], *DELPHES 3, A modular framework for fast simulation of a generic collider experiment*, JHEP **02**, 057 (2014) doi:10.1007/JHEP02(2014)057 [arXiv:1307.6346 [hep-ex]]