

WHIZARD 2.2

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

MANUAL ¹

WOLFGANG KILIAN,² THORSTEN OHL,³ JÜRGEN REUTER,⁴ WITH CONTRIBUTIONS FROM
FABIAN BACH,⁵ SEBASTIAN SCHMIDT, CHRISTIAN SPECKNER ⁶

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



when using WHIZARD please cite:
W. Kilian, T. Ohl, J. Reuter,
WHIZARD: Simulating Multi-Particle Processes at LHC and ILC,
Eur.Phys.J.C71 (2011) 1742, arXiv: 0708.4233 [hep-ph]

¹This work is supported by Helmholtz-Alliance “Physics at the Terascale”. In former stages this work has also been supported by the Helmholtz-Gemeinschaft VH-NG-005

²e-mail: kilian@hep.physik.uni-siegen.de

³e-mail: ohl@physik.uni-wuerzburg.de

⁴e-mail: juergen.reuter@desy.de

⁵e-mail: fabian.bach@desy.de

⁶e-mail: cnspeckn@googlemail.com

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, 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 and CTEQ/CT10 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

<http://projects.hepforge.org/whizard>

where also the **svn** repository is located.

Contents

1	Introduction	11
1.1	Disclaimer	11
1.2	Overview	12
1.3	Historical remarks	13
1.4	About examples in this manual	15
2	Installation	17
2.1	Package Structure	17
2.2	Prerequisites	17
2.2.1	No Binary Distribution	17
2.2.2	Tarball Distribution	18
2.2.3	SVN Repository Version	19
2.2.4	Fortran Compilers	20
2.2.5	LHAPDF	20
2.2.6	HOPPET	21
2.2.7	HepMC	22
2.2.8	STDHEP	23
2.3	Installation	24
2.3.1	Central Installation	24
2.3.2	Installation in User Space	25
2.3.3	Configure Options	26
2.3.4	Details on the Configure Process	27
2.3.5	WHIZARD self tests/checks	28
2.4	Working With WHIZARD	28
2.4.1	Working on a Single Computer	28
2.4.2	Stopping And Resuming WHIZARD Jobs	29
2.4.3	Submitting Batch Jobs With WHIZARD I / Relocation of WHIZARD	30
2.4.4	Submitting Batch Jobs With WHIZARD II	32
2.5	Troubleshooting	34
2.5.1	Possible (uncommon) build problems	34
2.5.2	What happens if WHIZARD throws an error?	34
2.5.3	Debugging, testing, and validation	40

3	Getting Started	41
3.1	Hello World	41
3.2	A Simple Calculation	43
4	Steering WHIZARD: SINDARIN Overview	47
4.1	The command language for WHIZARD	47
4.2	SINDARIN scripts	48
4.3	Errors	49
4.4	Statements	50
4.4.1	Process Configuration	50
4.4.2	Parameters	51
4.4.3	Integration	53
4.4.4	Events	54
4.5	Control Structures	56
4.5.1	Conditionals	57
4.5.2	Loops	57
4.5.3	Including Files	59
4.6	Expressions	59
4.6.1	Numeric	59
4.6.2	Logical and String	59
4.6.3	Special	60
4.7	Variables	60
5	Detailed WHIZARD Steering: SINDARIN	63
5.1	Data and expressions	63
5.1.1	Real-valued objects	63
5.1.2	Integer-valued objects	65
5.1.3	Complex-valued objects	65
5.1.4	Logical-valued objects	66
5.1.5	String-valued objects and string operations	66
5.2	Particles and (sub)events	67
5.2.1	Particle aliases	67
5.2.2	Subevents	67
5.2.3	Subevent functions	68
5.2.4	Calculating observables	71
5.2.5	Cuts and event selection	71
5.2.6	More particle functions	72
5.3	Physics Models	74
5.4	Processes	75
5.4.1	Process definition	76
5.4.2	Particle names	76
5.4.3	Options for processes	79
5.4.4	Process components	81

5.4.5	Compilation	83
5.4.6	Process libraries	84
5.4.7	Stand-alone WHIZARD with precompiled processes	84
5.5	Beams	85
5.5.1	Beam setup	85
5.5.2	Asymmetric beams and Crossing angles	86
5.5.3	LHAPDF	87
5.5.4	Built-in PDFs	89
5.5.5	HOPPET b parton matching for LHAPDF	89
5.5.6	Lepton Collider ISR structure functions	90
5.5.7	Lepton Collider Beamstrahlung	91
5.5.8	Beam events	94
5.5.9	Equivalent photon approximation	94
5.5.10	Effective W approximation	95
5.5.11	Energy scans using structure functions	96
5.5.12	Photon collider spectra	97
5.5.13	Concatenation of several structure functions	98
5.5.14	User-defined structure functions	99
5.6	Polarization	99
5.6.1	Initial state polarization	99
5.6.2	Final state polarization	104
5.7	Cross sections	106
5.7.1	Integration	106
5.7.2	Integration run IDs	110
5.7.3	Controlling iterations	111
5.7.4	Phase space	112
5.7.5	Cuts	113
5.7.6	QCD scale and coupling	115
5.7.7	Reweighting factor	116
5.8	Events	116
5.8.1	Simulation	117
5.8.2	Decays	118
5.8.3	Event formats	120
5.9	Analysis and Visualization	121
5.9.1	Observables	121
5.9.2	The analysis expression	122
5.9.3	Histograms	123
5.9.4	Plots	124
5.9.5	Analysis Output	125
5.10	Custom Input/Output	125
5.10.1	Output Files	126
5.10.2	Printing Data	126

6	Random number generators	129
6.1	General remarks	129
6.2	The TAO Random Number Generator	129
7	Integration Methods	131
7.1	The Monte-Carlo integration routine: VAMP	131
8	Phase space parameterizations	133
8.1	General remarks	133
8.2	The default method: wood	133
9	Methods for Hard Interactions	135
9.1	Internal unit matrix elements	135
9.2	Template matrix elements	135
9.3	The O’Mega matrix element generator	135
10	Implemented physics	137
10.1	The hard interaction models	137
10.1.1	The Standard Model and friends	137
10.1.2	Beyond the Standard Model	137
11	More on Event Generation	139
11.1	Event generation	139
11.2	Unweighted and weighted events	142
11.3	Choice on event normalizations	143
11.4	Supported event formats	144
11.5	Interfaces to Parton Showers, Matching and Hadronization	149
11.5.1	Parton Showers and Hadronization	150
11.5.2	Parton shower – Matrix Element Matching	152
11.6	Negative weight events	153
12	User Code Plug-Ins	155
12.1	The plug-in mechanism	155
12.2	Data Types Used for Communication	156
12.3	User-defined Observables and Functions	157
12.3.1	Cut function	157
12.3.2	Event-shape function	158
12.3.3	Observable	159
12.3.4	Examples	159
12.4	Spectrum or Structure Function	161
12.4.1	Definition	161
12.4.2	Example	164
12.5	User Code and Static Executables	166

13 Data Visualization	167
13.1 GAMELAN	167
13.2 Histogram Display	168
13.3 Plot Display	168
13.4 Graphs	168
13.5 Drawing options	170
14 User Interfaces for WHIZARD	173
14.1 Command Line and SINDARIN Input Files	173
14.2 WHISH – The WHIZARD Shell/Interactive mode	175
14.3 Graphical user interface	175
14.4 WHIZARD as a library	175
15 Examples	177
15.1 W pairs at LEP	177
16 Technical details – Advanced Spells	179
16.1 Efficiency and tuning	179
17 New Models via FeynRules	181
A SINDARIN Reference	183

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

<http://whizard.event-generator.org>

*or consult the current version in the **svn** repository on <http://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*

<http://projects.hepforge.org/whizard/trac/wiki>.

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 possible 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 definition 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



Figure 1.1: *General structure of the WHIZARD package.*

weights. Then, in Chap. 12, options for user to plug-in self-written code into the WHIZARD framework are detailed, e.g. for observables, selections and cut functions, or for spectra and structure functions. Also, static executables are discussed.

Chap. 13 is a stand-alone documentation of GAMELAN, the internal 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 the `FeynRules` program 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.

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: **W**HiZard, which stood for **W**, **H**iggs, **Z**, and **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 unofficial versions, the official version 1 of **WHIZARD** was released 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 improvements 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 show some of the major steps (physics implementation and/or technical improvements) in the development of **WHIZARD**:

0.99	08/1999	Beta version
1.00	12/2000	First public version
1.10	03/2001	Libraries; PYTHIA interface
1.11	04/2001	PDF support; anomalous couplings
1.20	02/2002	0'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 0'Mega (color flows); LHA format
1.51	06/2006	Hgg , $H\gamma\gamma$; Spin $3/2 + 2$; BSM models
1.90	11/2007	0'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.
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; 2HDM; reweighting; LHE v2/3; BLHA; HOPPET interface; inclusive processes

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 $e^+e^- \rightarrow \mu^+\mu^-$. This is the same process, the textbook by Peskin/Schroeder [34] 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: <http://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

<http://whizard.hepforge.org>

The distribution is a single file, say `whizard-2.2.0.tgz` for version 2.2.0.

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. 2.2.4 for details).
- The **OCaml** system. **OCaml** is a functional and object-oriented language. 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 privileges¹.

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**.

- **L^AT_EX** and **MetaPost** for data visualization. Both are part of the **T_EX** program family. These programs are not absolutely necessary, but **WHIZARD** will lack the tools for visualization without them.
- The **LHAPDF** structure-function library. See Sec. 2.2.5.

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. 2.5.1.

- The HOPPET structure-function matching tool. See Sec. 2.2.6.
- The HepMC event-format package. See Sec. 2.2.7.
- The STDHEP event-format package. See Sec. 2.2.8.

Once these prerequisites are met, you may unpack the package in a directory of your choice

```
some-directory> tar xzf whizard-2.2.0.tgz
```

and proceed.²

The directory will then contain a subdirectory `whizard-2.2.0` 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.

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.
- The `noweb` package, a light-weight tool for literate programming. This package is nowadays often part of Linux distributions³. You can obtain the source code from⁴

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

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

```
your-src-directory> svn checkout http://whizard.hepforge.org/svn/trunk/ .
```

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, run⁵

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

This will generate a `configure` script.

²Without GNU `tar`, this would read `gunzip -c whizard-2.2.0.tgz | tar xz -`

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

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

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

2.2.4 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 4.7.1 or higher⁶.
- **nagfor** (NAG). You will need version 5.2 or higher.

There are some commercial compilers that might be able to compile WHIZARD 2.2 in the near future, but at the time of writing, all of the compilers listed below contained compiler bugs. Consult the WHIZARD website for updates on this situation.

- **ifort** (Intel). You will need at least version 15 or higher than 14.2.
- **pgfortran** (PGI). You will need a more modern version than 14.4.

2.2.5 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 required⁷.

Please note, that the new release series of LHAPDF, version 6.0 and higher, is not yet supported within WHIZARD 2.2.0. A working interface is planned for one of the next minor releases.

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

<http://lhapdf.hepforge.org/lhapdf5>

for LHAPDF version 5 and older, or

<http://lhapdf.hepforge.org>

for version 6 and newer, 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.

⁶Note that WHIZARD versions 2.0.0 until 2.1.1 compiled with **gfortran** 4.5.x and 4.6.x, but the object-oriented refactoring of the WHIZARD code from 2.2 on made a switch to **gfortran** 4.7.1 or higher necessary. **gcc** 4.7.0 contains a major bug in the static **libstdc++** library which prevented static builds of WHIZARD with external C++ libraries.

⁷ Note that PDF sets which contain photons as partons are only supported with WHIZARD for 5.7.1 or higher

When configuring WHIZARD, WHIZARD looks for the binary `lhpdf-config` (which is present since LHAPDF version 4.1.0): if this file is in an executable path, 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:

```
configure: -----
configure: --- LHAPDF ---
configure:
checking for lhpdf-config... /usr/local/bin/lhpdf-config
checking the LHAPDF version... 5.9.1
checking the LHAPDF pdfsets path... /usr/local/share/lhpdf/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 `lhpdf-config` in an executable path that is checked before the system paths, e.g. `<home>/bin`.

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 lhpdf-config... /usr/local/bin/lhpdf-config
checking the LHAPDF version... 5.9.1
checking the LHAPDF pdfsets path... /usr/local/share/lhpdf/PDFsets
checking for standard PDF sets... all standard PDF sets installed
checking for initpdfsetm 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 case, LHAPDF is enabled and correctly linked. If some of them are missing, then this test will result in a failure. The last check is for the `has_photon` flag, which tests whether photon PDFs are available in the found LHAPDF installation.

2.2.6 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
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.

2.2.7 HepMC

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 webpage:

<http://lcgapp.cern.ch/project/simu/HepMC/>

or

<https://sft.its.cern.ch/jira/browse/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`.

A WHIZARD configuration for HepMC is a bit lengthier as the C++ details have to be checked first:

```

configure: -----
configure: --- HepMC ---
configure:
checking for g++... g++
checking whether we are using the GNU C++ compiler... yes
checking whether g++ accepts -g... yes
checking dependency style of g++... gcc3
checking how to run the C++ preprocessor... g++ -E
checking for ld used by g++... /usr/bin/ld
checking if the linker (/usr/bin/ld) is GNU ld... yes
checking whether the g++ linker (/usr/bin/ld) supports shared libraries... yes
checking for g++ option to produce PIC... -fPIC -DPIC
checking if g++ PIC flag -fPIC -DPIC works... yes
checking if g++ static flag -static works... yes
checking if g++ supports -c -o file.o... yes
checking if g++ supports -c -o file.o... (cached) yes
checking whether the g++ linker (/usr/bin/ld) supports shared libraries... yes
checking dynamic linker characteristics... GNU/Linux ld.so
checking how to hardcode library paths into programs... immediate
checking the HepMC version... 2.06.09
checking for LDFLAGS_STATIC: host system is linux-gnu: static flag...
checking for GenEvent class in -lHepMC... yes
checking whether we are using the GNU Fortran compiler... (cached) yes
checking whether /usr/bin/gfortran accepts -g... (cached) 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.

2.2.8 STDHEP

STDHEP is a library for handling collider scattering events. In particular, it provides a portable format for event files. If you do not need this format, you may skip this section.

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

<http://cepa.fnal.gov/psm/stdhep>

You will need only the libraries for file I/O, not the various translation tools for PYTHIA, HERWIG, etc. Note that STDHEP has largely been replaced by the HepMC format, and conversion tools exist.

If the STDHEP library is linked with the installation, WHIZARD is able to write files in the STDHEP format,

STDHEP is written in Fortran77. Although not really necessary, we strongly advice to compile STDHEP with the same compiler as WHIZARD. Otherwise, one has to add the corresponding Fortran77 run-time libraries to the configure command for WHIZARD. In order to compile STDHEP with a modern Fortran compiler, add the line `F77 = <your Fortran compiler>` below the

`MAKE=make` statement in the GNUmakefile of the `STDHEP` distribution after you extracted the tarball (Note that there might be some difficulties that some modern compilers do not understand the `D` debugging precompiler statements in some of the files. In that case just replace them by comment characters, `C`. Also, some of the hard-coded compiler flags are tailor-made for old-fashioned `g77`). After that just do `make`. Copy the libraries created in the `lib` directory of your `STDHEP` distribution to a directory which is in the `LD_LIBRARY_PATH` of your computer.

The `WHIZARD` configure script will search for the two libraries `libFmcfio.a` and `libstdhep.a`. When `WHIZARD` does not find the `STDHEP` library, you have to set the location of the two libraries explicitly:

```
./configure ... .. STDHEP=<stdhep path>/libstdhep.a
                FMCFIO=<fmc fio path>/libFmcfio.a
```

The corresponding configure output will look like this:

```
configure: -----
configure: --- STDHEP ---
configure:
checking for libFmcfio.a... /usr/local/lib/libFmcfio.a
checking for libstdhep.a... /usr/local/lib/libstdhep.a
checking for stdxwinit in -lstdhep -lFmcfio... yes
configure: -----
```

In the last line, `WHIZARD` checks whether it can correctly access functions from the library. If some symbols could not be resolved, it will put a “no” in the last entry. Then the `config.log` will tell you more about what went wrong in detail.

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

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 administrator 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 be 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. 2.4.1.

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-4.8 FCFLAGS="-g -O3" --enable-fc-omp
```

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-omp**: 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.
- **LHAPDF_DIR** (variable): The location of the optional LHAPDF 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.
- **STDHEP_DIR** (variable): The location of the optional STDHEP 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 `-llibrary`
- `LDFLAGS` (variable): non-standard linker flags

For other options (like e.g. `--enable-fc-quadruple` 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 <http://projects.hepforge.org/whizard/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. 2.3.5) 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 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 2.2.0 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 setting the configure option **--enable-extnum-checks**. On the other hand, the standard self-consistency checks can be completely disabled with the option **--disable-default-checks**.

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.

2.4 Working With WHIZARD

2.4.1 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.

2.4.2 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.⁸ 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.

⁸This 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.

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.

2.4.3 Submitting Batch Jobs With **WHIZARD** I / Relocation of **WHIZARD**

For long-running calculations, you may want to submit a **WHIZARD** job to a remote machine. The challenge lies in the fact that **WHIZARD** needs a complete installation with all auxiliary programs and data files to run, including a **Fortran** compiler.

If the submitting machine where **WHIZARD** has been compiled is binary- or OS-incompatible with the batch machine, there is no way around doing the complete **WHIZARD** installation and compilation on the batch machine, possibly as part of the batch job.

In this section, we describe batch-job preparation in the case that the batch machine has a compatible operating system, and the necessary system tools are available, albeit possibly in different locations. In that case, an existing **WHIZARD** installation can be transferred to the remote machine without recompilation.

The option to completely relocate a configured, compiled and pre-installed **WHIZARD** installation to another location cannot only be used for the setup of **WHIZARD** on batch clusters, but also for automated installations, for testing purposes, for tutorial installations etc. It does not play a role whether the relocation is on the same machine, or on a different computer, as long as the other machine is OS-compatible, compiler and compiler paths are available on the other machine, and external libraries (like **HepMC**, **LHAPDF**) are being found in the same locations.

We assume that it is possible to transfer files from and to the batch machine, and that the batch job is controlled by some script. You (interactively) or the script should perform the following steps, as far as necessary.

To relocate a **WHIZARD** installation, perform the following steps:

- 1.
2. Pack the complete **WHIZARD** installation including all subdirectories (**bin**, **include**, **lib**, **share**, **var** and the **libtool** script) and unpack it on an arbitrary location, say **reloc-dir**. Pack the complete **WHIZARD** installation including all subdirectories and unpack it on the batch machine in an arbitrary location, say **reloc-dir**. [only for relocation to a different computer]

3. Copy the **SINDARIN** script file (say, `run.sin`) to the batch machine in the projected working directory [only relocation to a different computer]
4. Check whether the correct (compatible!) **Fortran** compiler is available in the standard path. If not, create a symbolic link or extend `PATH` accordingly. [only for relocation to a different computer]
5. Check whether the correct (compatible!) **Fortran** runtime library is available in the standard load path, and has priority over any conflicting libraries. If not, create symbolic links or extend `LD_LIBRARY_PATH` accordingly. [only for relocation to a different computer]
6. Do the same for any external libraries as far as they have been linked with the original installation (e.g., **LHAPDF**, **HepMC**). You should verify that the `stdc++` library can be loaded. [only for relocation to a different computer]
7. Check whether the batch machine has a working **L^AT_EX** and **MetaPost** installation. If it doesn't, this is not a severe problem, you just may get some extra error messages, and there won't be graphical output from analysis requests. [only for relocation to a different computer]
8. Execute the relocation script in the `bin` directory of the unpacked **WHIZARD** installation with a prefix option pointing to the new location:

```
reloc-dir/bin/whizard-relocate.sh --whizard_prefix reloc-dir
```

The `reloc-dir` is the path of the relocated **WHIZARD** installation, i.e. the directory that contains at least a `bin` subdirectory, and in case the **WHIZARD** libraries are not found in an accessible path, i.e `lib`, `include`, `share` paths.

This relocation script does the following steps, that can also be run individually:

8a. Run

```
reloc-dir/bin/whizard-setup.sh --prefix reloc-dir
```

where `reloc-dir` is the directory where you unpacked the **WHIZARD** installation, to add **WHIZARD**'s `bin` and `lib` directories to the run and load path, respectively. Note that without the prefix this script adds the paths of the install directory of the original build, i.e. `/usr/local` or the path set in the original configure.

8b. The **WHIZARD** installation is self-contained, but the steering files for the dynamically loaded libraries contain paths that will likely be wrong on the batch system. Fix this with

```
libtool-relocate.sh --prefix reloc-dir
```

If you need **LHAPDF**, and the library is not in the same location as on your host, run instead

```
libtool-relocate.sh --prefix reloc-dir --lhpdf directory-of-liblhpdf
```

- 8c. The next obstacle might be WHIZARD's libtool script. Libtool is a standard tool, but contains machine-specific configurations. If there is – or might be – a problem, run

```
libtool-config.sh --prefix reloc-dir
```

This will create a tailored libtool in the current working directory.

9. Now, the WHIZARD binary can be successfully launched. If WHIZARD doesn't even start, there is something wrong with the preceding steps.

Still, WHIZARD has to be told where to find its files. This is taken care of by the script `whizard.sh`. Please do not call the `whizard` binary directly, use this shell wrapper `whizard.sh` instead. You might want to add a line like:

```
alias "whizard=reloc-dir/bin/whizard.sh"
```

to a `.bash_profile` or `.bashrc` file.

Alternatively, one can run the WHIZARD binary directly with the `--prefix` option

```
whizard --prefix=reloc-dir run.sin
```

You may want to catch standard output and standard error. This depends on your batch system.

If you had to rebuild libtool (see above), you need the additional option

```
--libtool=my-libtool
```

where `my-libtool` is the tailored libtool that you created, e.g., `$pwd/libtool`.

If you need LHAPDF, and its location is different, you need the additional option

```
--lhpdf-dir=directory-where-lhpdf-is-installed
```

If these switches are not set correctly, WHIZARD will fail while running.

10. If all works well, WHIZARD will run as requested. Copy back all files of interest in the working directory, and you are done.

As a rule, the more similar the batch machine is to the local machine, the more steps can be omitted or are trivial. However, with some trial and error it should be able to run batch jobs even if there are substantial differences.

2.4.4 Submitting Batch Jobs With WHIZARD II

There is another possibility that avoids some of the difficulties discussed above. You can suggest WHIZARD to make a statically linked copy of itself, which includes all processes that you want to study, hard-coded. The external libraries (**Fortran**, and possibly **HepMC** and **stdc++**) must be available on the target system, and it must be binary-compatible, but there is no need for transferring the complete WHIZARD installation or relocating paths. The drawback is that generating, compiling and linking matrix element code is done on the submitting host.

Since this procedure is accomplished by `SINDARIN` commands, it is explained below in Sec. 5.4.7.

2.5 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.

2.5.1 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 corrected 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-2.0.0.tar.gz | tar xf -
$ cd whizard-2.0.0
$ 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 advised to keep build and source directory apart from each other.

2.5.2 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 which 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 $t\bar{t} \rightarrow b\bar{b}$). 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 possible 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     = 1.00000000
  pure state     = F
  @(+1: +1: ( 3.333333333333333E-01, 0.000000000000000E+00))
  @(-1: -1: ( 6.666666666666667E-01, 0.000000000000000E+00))
  @(-1: +1: ( 6.666666666666667E-01, 0.000000000000000E+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 $e^+e^- \rightarrow 8\gamma$ 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 $pp \rightarrow jj$ 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.

2.5.3 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 3

Getting Started

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 script is written in SINDARIN. This 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

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.

```
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
```


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 2.2.0                                     |
|=====|
| 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 2.2.0                                     |
|=====|
| 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 `0'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. 8.2) 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.

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 possibilities 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 assignments 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 parentheses (...), 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
integrate (foo)
```

WHIZARD will fail with the error message

```
sqrts = 1 TeV
integrate (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  $\langle tag \rangle$  =  $\langle incoming \rangle \Rightarrow \langle outgoing \rangle$ 
```

Define a process. You give the process a name $\langle tag \rangle$ 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 =  $\langle energy-value \rangle$ 
```

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 =  $\langle beam-particles \rangle$   
beams =  $\langle beam-particles \rangle \Rightarrow \langle structure-function-setup \rangle$ 
```

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

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

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**

```
 $\langle parameter \rangle$  =  $\langle value \rangle$   
 $\langle type \rangle$   $\langle user-parameter \rangle$   
 $\langle type \rangle$   $\langle user-parameter \rangle$  =  $\langle value \rangle$ 
```

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. ??.

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 =  $\langle$ analysis-expression $\rangle$ 
```

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  $\langle$ particle $\rangle$  ( $\langle$ decay-channels $\rangle$ )
```

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 =  $\langle$ integer $\rangle$ 
```

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. 13.4.

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 L^AT_EX 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>
elseif <logical-expression> then <statements>
else <statements>
endif
```

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

```
if (sqrt > 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⟩ /* ⟨multiplier⟩
⟨start-value⟩ => ⟨end-value⟩ // ⟨divisor⟩

```

The iterator can be put in place of an expression in the $\langle value-list \rangle$. 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 $\langle statements \rangle$ 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⟩ /* ⟨multiplier⟩
⟨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 $\langle start-value \rangle$ and $\langle end-value \rangle$ 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** objects 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` construct. 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

Detailed WHIZARD Steering: SINDARIN

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

+ - * / ^

with their obvious meaning and the usual precedence rules.

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

```
abs  sgn  mod  modulo
sqrt exp  log  log10
sin  cos  tan  asin  acos  atan
sinh  cosh  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:

max min

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

The following functions of a real convert to integer:

int nint floor ceiling

and this converts to complex type:

complex

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

== <>
> < >= <=

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

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 a 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 exclusive¹.

¹In 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:

```
1 -98765 0123
```

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:

```
+ - * / ^
```

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:

```
abs  sgn  mod  modulo
      max  min
```

and the conversion functions

```
real  complex
```

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,

```
complex c = 1 + 2 * I
```

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

```
or and 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 be 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

```
%d %i (integer)
%e %f %g %E %F %G (real)
%s (string and logical)
```

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

```
sprintf format-string (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 output².

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:

²In older versions of **WHIZARD**, until v2.1.1, there also used to be a `sprintfd` function and a `printfd` 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 component-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 particle 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.

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 (or two particle momenta) 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.

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.
- **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, $Q^2 = 1 \text{ GeV}^2$.

There is also an integer-valued observable:

- **PDG**
 - One argument: PDG code of the particle. For a composite particle, the code is undefined (value 0).

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 `0'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 location³.)

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.

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.

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

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 $2 \rightarrow 1$ 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 $2 \rightarrow 6$ particles are safe, $2 \rightarrow 8$ 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 [38], 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

```

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

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

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.

Restriction on intermediate states

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 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 $e^-e^+ \rightarrow \nu\bar{\nu}H$, summed over all neutrino generations, contains both ZH pair production (Higgs-strahlung) and $W^+W^- \rightarrow H$ 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 `0'Mega` manual.

Other options

There are some further options that the `0'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 `0'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 `0'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, `0'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.

Another possibility is

```
$omega_flags = "-model:constant_width"
```

which puts the width both in the s and in the t channel everywhere.

Note that both options apply only to charged unstable particles, such as the W boson.

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 appropriate 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. $e^-e^+ \rightarrow \mu^-\gamma$ 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 unnecessary. 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`. This 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 => lhpdf
```

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 particular 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 => lhpdf
```

(Note that for the moment, only the release series 5 of LHAPDF is automatically interfaced to WHIZARD. Support for the release series 6 of LHAPDF is foreseen for the near future). The above `beams` statement selects a default LHAPDF structure-function set for both proton beams (in the current setup, this is `cteq6ll.LHpdf`, member 0). 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 `lhpdf` keyword:

```
beams = p, p => lhpdf
$lhpdf_file = "MSTW2008lo68cl.LHgrid"
```

Similarly, a member within the set is selected by the numeric variable `lhpdf_member`.

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, lhpdf
```

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 $p\gamma$ collisions, where the photon is to be resolved as a hadron. The simple assignment

```
beams = p, gamma => lhpdf, lhpdf_photon
```

will be understood as follows: WHIZARD selects the appropriate default structure functions, `cteq6ll.LHpdf` for the proton and `GSG960.LHgrid` for the photon. The photon case has an additional integer-valued parameter `lhpdf_photon_scheme`. (There are also pion structure functions available.) For modifying the default, you have to specify separate structure functions

```
beams = p, gamma => lhpdf, lhpdf_photon
$lhpdf_file = ...
$lhpdf_photon_file = ...
```

Finally, the scattering of elementary photons on partons is described by

```
beams = p, gamma => lhpdf, 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 `$lhpdf_dir = "<path-to-lhpdf>"`. Usually, it is not necessary to set this because WHIZARD detects this path via the `lhpdf-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.

Tag	Name	Notes	References
cteq6l	CTEQ6L	—	[39]
cteq6l1	CTEQ6L1	—	[39]
cteq6d	CTEQ6D	—	[39]
cteq6m	CTEQ6M	—	[39]
mrst2004qedp	MRST2004QED (proton)	includes photon	[40]
mrst2004qedn	MRST2004QED (neutron)	includes photon	[40]
mstw2008lo	MSTW2008LO	—	[41]
mstw2008nlo	MSTW2008NLO	—	[41]
mstw2008nnlo	MSTW2008NNLO	—	[41]
ct10	CT10	—	[42]

Table 5.2: All PDF sets available as builtin sets. The two MRST2004QED sets also contain a photon.

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 appropriate value. E.g, modifying the above setup to

```
beams = p, p => pdf_builtin
$pdf_builtin_set = "mrst2004qedp"
```

would select the proton PDF from the MRST2004QED set. A list of all currently available PDFs can be found in Table 5.2.

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.

5.5.5 HOPPET b parton matching for LHAPDF

When both LHAPDF for hadron-collider PDF structure functions and the HOPPET tool [43] for their manipulations are correctly linked to WHIZARD, both 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. Here, it depends on the corresponding process and the energy scales involved whether it is a better description to use the $g \rightarrow b\bar{b}$ 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 $\tan\beta$: $pp \rightarrow H$

with a partonic subprocess $b\bar{b} \rightarrow H$, or directly take the gluon PDFs and use $pp \rightarrow b\bar{b}H$ with a partonic subprocess $gg \rightarrow b\bar{b}H$. 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 => lhpdf
sqrts = 14 TeV
?lhpdf_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 $g \rightarrow b\bar{b}$ 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 LHAPDF (note that it is not possible to use WHIZARD's built-in PDFs for the b parton matching schemes), and the HOPPET matching routines are switched on by the flag `?lhpdf_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
```

There are several options for the lepton-collider ISR structure function that are summarized in the following:

Parameter	Default	Meaning
<code>isr_alpha</code>	0/intrinsic	value of α_{QED} for ISR
<code>isr_order</code>	3	max. order of hard-collinear photon emission
<code>isr_mass</code>	0/intrinsic	mass of the radiating lepton
<code>isr_q_max</code>	$0/\sqrt{s}$	upper cutoff for ISR
<code>?isr_recoil</code>	false	flag to switch on recoil/ p_T

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. Finally, with the flag `?isr_recoil`, the p_T recoil of the emitting lepton against the photon radiation can be switched on; per default it is off. 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.

For more information on the underlying physics, see Chap. 9.

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 unofficial 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.

In this section, we discuss the usage of beamstrahlung spectra by means of the **circeone** 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 $e \rightarrow \gamma$ 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
```

In all cases (one or both beams with photon conversion) the beam spectrum applies to both beams simultaneously.

This is an overview over all options and flags for the `CIRCE1` setup for lepton collider beamstrahlung:

Parameter	Default	Meaning
<code>?circe1_photon1</code>	<code>false</code>	$e \rightarrow \gamma$ conversion for beam 1
<code>?circe1_photon2</code>	<code>false</code>	$e \rightarrow \gamma$ conversion for beam 2
<code>circe1_sqrts</code>	\sqrt{s}	collider energy for the beam spectrum
<code>?circe1_generate</code>	<code>true</code>	flag for the <code>CIRCE1</code> generator mode
<code>?circe1_map</code>	<code>true</code>	flag to apply special phase-space mapping
<code>circe1_mapping_slope</code>	2.	value of PS mapping exponent
<code>circe1_eps</code>	1E-5	parameter for mapping of spectrum peak position
<code>circe1_ver</code>	0	internal version of <code>CIRCE1</code> package
<code>circe1_rev</code>	0/most recent	internal revision of <code>CIRCE1</code>
<code>\$circe1_acc</code>	SBAND	accelerator type
<code>circe1_chat</code>	0	chattiness/verbosity of <code>CIRCE1</code>

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.

In general, 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 [17,18,19] 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 = "ilc_ee_500_polavg.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. 5.5.12.

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 Equivalent photon approximation

The equivalent photon approximation (EPA) uses an on-shell approximation for the $e \rightarrow e\gamma$ collinear splitting to allow the simulation of photon-induced backgrounds in lepton collider physics. The original concept is that of the Weizsäcker-Williams approximation [20,21,22]. This is a single-beam structure function that can be applied to both beams, or also to one beam only. Examples 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
<code>epa_alpha</code>	0/intrinsic	value of α_{QED} for EPA
<code>epa_x_min</code>	0.	soft photon cutoff in x (mandatory)
<code>epa_q_min</code>	0.	minimal γ momentum transfer (mandatory)
<code>epa_mass</code>	0/intrinsic	mass of the radiating fermion
<code>epa_e_max</code>	0/ \sqrt{s}	upper cutoff for EPA
<code>?epa_recoil</code>	false	flag to switch on recoil/ p_T

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 energy cutoff for the EPA structure function, `epa_e_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 $e^+e^- \rightarrow e^+e^-\gamma\gamma$ 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.

5.5.10 Effective W approximation

An approach similar to the equivalent photon approximation (EPA) discussed in the previous section Sec. 5.5.9, 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 $2 \rightarrow 4$ processes was still very challenging and $2 \rightarrow 6$ processes almost impossible, such that this approximation was the only viable solution for the simulation of processes like $pp \rightarrow jjVV$ 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 truly 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. 5.5.13:

```
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
<code>ewa_x_min</code>	0.	soft W/Z cutoff in x (mandatory)
<code>ewa_mass</code>	0/intrinsic	mass of the radiating fermion
<code>ewa_pt_max</code>	$0/\sqrt{\hat{s}}$	upper cutoff for EWA
<code>?ewa_keep_energy</code>	false	recoil switch, energy conservation
<code>?ewa_keep_momentum</code>	false	recoil switch, momentum conservation

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 $f \rightarrow fV$ 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 `ewa_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 are two flags to switch on a recoil for the electroweak boson against the radiating fermion. Note that this is an experimental feature that is not completely tested. In any case, the non-collinear kinematics violates 4-momentum conservation, so there are two choices: either to conserve the energy (`?ewa_keep_energy`) or to conserve 3-momentum (`?ewa_keep_momentum`). Both flags are false per default, and kinematics is collinear as a standard.

More details about the physics can be found in Chap. 9.

5.5.11 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 $e^+e^- \rightarrow \mu^+\mu^-$, 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 let 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.

5.5.12 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 [23,24,25]. 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 `$circe2_design`, where the default setting `"*`" corresponds to the default of CIRCE2 (which is the TESLA 500 GeV machine as discussed in the TESLA Technical Design Report [26,27]). Note that up to now there have not been any setups for a photon collider option for the modern linear collider concepts like ILC and CLIC. The string variable `$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 `src/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.

5.5.13 Concatenation of several structure functions

As has been shown already in Sec. 5.5.9 and Sec. 5.5.10, 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
```

5.5.14 User-defined structure functions

Note that in WHIZARD version v2.2.0 this mechanism is temporarily disabled due to the refactoring procedure from v2.1 to v2.2. It is expected to be switched on again as soon as possible, most likely in one of the upcoming minor versions, v2.2.x

There is also a sort of plug-in mechanism that allows to user to write his/her own **Fortran** code for a structure function or a pair spectrum, compile it and use it in a simulation within WHIZARD. The matching rules of beam and hard scattering particles with the structure function setups do apply here as well. The syntax is similar to the cases above, except that there is a mandatory string argument

```
beams = e1, E1 => user_strfun ("my_sf")
```

This syntax expects that there is a **Fortran95** file named `my_sf.f90` in the same directory as the SINDARIN file containing the command above. Furthermore, there are certain standard which functions and subroutines this code has to have. An example interface of such a file is given by `share/tests/user_strfun.f90`.

More details about user-plugin mechanisms in WHIZARD can be found in Chap. 12.

5.6 Polarization

5.6.1 Initial state polarization

WHIZARD supports polarizing the initial 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` statement⁴:

```
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.

⁴Note 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.

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:

```
| -----
| 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:

$$\rho(e^-) = \text{diag}(0.10, 0.90) \quad \rho(e^+) = \text{diag}(0.65, 0.35) \quad ,$$

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:**

`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 = @(±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:

$$\rho = \text{diag} \left(\frac{1 \pm f}{2}, 0, \dots, 0, \frac{1 \mp f}{2} \right)$$

- **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:

$$\rho = \text{diag} \left(\frac{1-f}{|m|}, \dots, \frac{1-f}{|m|}, \frac{1+f(|m|-1)}{|m|}, \frac{1-f}{|m|}, \dots, \frac{1-f}{|m|} \right)$$

($|m| = 2j + 1$: 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 $\phi = 90^\circ$ 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:

$$\rho = \begin{pmatrix} 1 & 0 & \dots & \dots & \frac{f}{2} e^{-i\phi} \\ 0 & 0 & \ddots & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 0 & 0 \\ \frac{f}{2} e^{i\phi} & \dots & \dots & 0 & 1 \end{pmatrix}$$

(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:

$$\rho = \frac{1}{2} \cdot \begin{pmatrix} 1 - f \cos \theta & 0 & \cdots & \cdots & f \sin \theta e^{-i\phi} \\ 0 & 0 & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & & & \ddots & 0 \\ f \sin \theta e^{i\phi} & \cdots & \cdots & 0 & 1 + f \cos \theta \end{pmatrix}$$

- **Diagonal density matrix:**

```
beams_pol_density = @(j:j:hj, j-1:j-1:hj-1, ..., -j:-j:h-j)
```

This defines an arbitrary diagonal density matrix with entries $\rho_{j,j}, \dots, \rho_{-j,-j}$.

- **Arbitrary density matrix:**

```
beams_pol_density = @({m : m' : xm,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:-1)
```

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 wwa = Wp, Wm => A, A
process zee = Z => e1, E1

sqrts = 200 GeV
beams_pol_density = @(1, -1, 1:-1:-1), @()
beams = Wp, Wm
integrate (wwa)
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 by 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 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 \bar{d} quarks, but only for part of the final state u and \bar{u} quarks. In this case, if you had wanted to keep the helicity information also for all u and \bar{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 $e^+e^- \rightarrow q\bar{q}q\bar{q}$ with q being any light quark, i.e., W^+W^- and ZZ production and hadronic decay together with 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

```

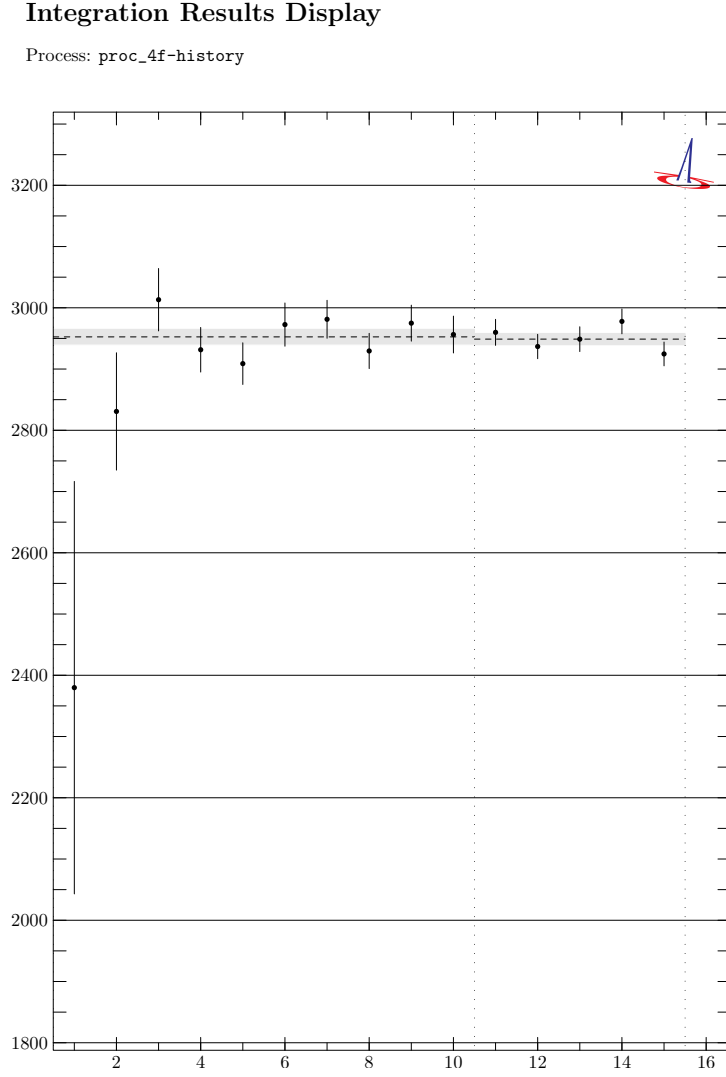


Figure 5.1: Graphical output of the convergence of the adaptation during the integration of a *WHIZARD* process.

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 $\sqrt{n_{\text{calls}}}$. 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, 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 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 $\gamma^* \rightarrow \mu^+\mu^-$ 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. 8.2.

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 `?alpha_s_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 `?alpha_s_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 `?alpha_s_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 `?alpha_s_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 `alpha_s_order` (default 0, meaning leading-log; maximum 2) and `alpha_s_nf` (default 5).

Otherwise there is the option to set `?alpha_s_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 $\Lambda_{\text{QCD}} = 200$ MeV. `alpha_s_order` and `alpha_s_nf` apply analogously.

Note that for using one of the running options for α_s , always `?alpha_s_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.

There are two things that are usually done with an event sample. It can be analyzed directly when it is generated or read, and it can be written to file in a standard format that a human or an external program can understand. The basic analysis features of `WHIZARD` are described below in Sec. 5.9. 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.

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`. We give an example for a $pp \rightarrow Wj$ final state:

```
process wj = u, gl => d, Wp
process wen = Wp => E1, n1

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.

Usually, WHIZARD applies full spin and color correlations to the factorized processes, so it keeps both color and spin coherence between productions and decays. It sums over all quantum numbers. At the moment, it is not yet possible to generate polarized particles and let them decay in a factorized approach. This will be covered in a future version of WHIZARD. Another restriction is that momentarily the factorized approach is always treated in narrow-width approximation; this is motivated by the fact that whenever the width plays an important role, the usage of the factorized approach will not be fully justified. There are again plans for a future WHIZARD version to also include Breit-Wigner or Gaussian distributions when using the factorized approach.

There are several options to play around with the spin correlations: the default is that

they are completely taken into account. However, one can restrict them to the classical spin correlations, i.e. to the diagonal elements of the spin-density matrix between production and decay. This is done by setting the flag `?diagonal_decay = true`. In order to test whether spin correlations might play a crucial role in a process or not, the user can even completely switch off spin correlations by setting `?isotropic_decay = true`. Both flags are off by default.

Decays can also 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 $t\bar{t}\mu^+\nu_\mu$ 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 explicitly marked as being stable, using the

```
stable <particle-tag>
```

constructor for the particle `<particle-tag>`.

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 default 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. 11.4.

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. 13). 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. 13.

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 `<tag>.log`, where `<tag>` 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 parentheses. Logical expressions must be enclosed in parentheses prefixed by a question mark `?`. String expressions must be enclosed in parentheses 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](#).

Chapter 6

Random number generators

6.1 General remarks

6.2 The TAO Random Number Generator

Chapter 7

Integration Methods

7.1 The Monte-Carlo integration routine: VAMP

Chapter 8

Phase space parameterizations

8.1 General remarks

8.2 The default method: wood

Chapter 9

Methods for Hard Interactions

9.1 Internal unit matrix elements

9.2 Template matrix elements

9.3 The O'Mega matrix element generator

Chapter 10

Implemented physics

10.1 The hard interaction models

10.1.1 The Standard Model and friends

10.1.2 Beyond the Standard Model

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 $Hgg, H\gamma\gamma, H\mu\mu$	---	SM_Higgs
SM with charge 4/3 top	---	SM_top
SM with anomalous top couplings	---	SM_top_anom
SM with anomalous Higgs couplings	---	SM_rx/NoH
SM extensions for VV scattering	---	SSC/AltH
SM with Z'	---	Zprime
Two-Higgs Doublet Model	2HDM	2HDM_CKM
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.*

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 worthwhile. 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, $e^+e^- \rightarrow \mu^+\mu^-$,

```
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'
sqrt_s = 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)
|   sqrt_s = 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 11.4 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 mult-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 $2 \rightarrow 2$ 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.

11.3 Choice on event normalizations

There are basically four different choices to normalize event weights ($\langle \dots \rangle$ 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. $\langle w_i \rangle = 1/N$, $\langle \sum_i w_i \rangle = 1$

$$4. \langle w_i \rangle = \sigma/N, \quad \langle \sum_i w_i \rangle = \sigma$$

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.

In the following section we show some examples when discussing the different event formats available in WHIZARD.

11.4 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 StdHEP and 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

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
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	yes
StdHEP (HEPRUP/EUP)	binary	based on HEPRUP/EUP common block	yes
Weight stream	ASCII	just weights	yes

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*

shown in the appendix).

Standard suffix: `.evt`

- **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`
- **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 (only available if StdHEP is installed and correctly linked)
Standard suffix: `.stdhep`
- **stdhep_up**: StdHEP binary format based on the HEPUP/HEPEUP common blocks (only available if StdHEP is installed and correctly linked)
Standard suffix: `.up.stdhep`
- **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:

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 $hh \rightarrow hh$, 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 containing 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.

```

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 [29].

```

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 [31].

```

<LesHouchesEvents version="1.0">
<header>
  <generator_name>WHIZARD</generator_name>
  <generator_version>2.2.0</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 9.00
  25 1 1 2 0 0 1.4960220911E+02 4.6042825611E+02 0.0000000000E+00 5.0000000000E+02 1.2500000000E+02 0.0000000000E+00 9.00
</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.

11.5 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 $e^+e^- \rightarrow \text{hadrons}$ and $p\bar{p} \rightarrow Z$ without matching. The corresponding `SINDARIN` files with matching enabled are `EEMatching2X.sin` to `EEMatching5X.sin` for $e^+e^- \rightarrow \text{hadrons}$ with a different number of partons included in the matrix element and `DrallYanMatchingX.sin` for Drell-Yan with one matched emission.

11.5.1 Parton Showers and Hadronization

From version 2.1 onwards, `WHIZARD` contains an implementation of an analytic parton shower as presented in [44], providing the opportunity to perform the parton shower from within `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.

During configuration the `--enable-shower` flag has to be set (this is the default from version 2.1.0 on), which then triggers automatic compilation of the shower subpackage and the `PYTHIA` version included in the `WHIZARD` package (which is the last release of the `Fortran PYTHIA`, v6.427) as well as the interface. It can be invoked by the following `SINDARIN` keywords:

<code>?ps_fsr_active = true</code>	master switch for final-state parton showers
<code>?ps_isr_active = true</code>	master switch for initial-state parton showers
<code>?hadronization_active = true</code>	master switch to enable hadronization
<code>?ps_use_PYTHIA_shower = true</code>	switch to use <code>PYTHIA</code> 's parton shower instead of <code>WHIZARD</code> '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_alpha_s_running** Switch to decide between a constant α_S , given by **ps_fixed_alpha_s**, and a running α_S , calculated using **ps_isr_lambda** for ISR. (Default = true)
- ?ps_fsr_alpha_s_running** Switch to decide between a constant α_S , given by **ps_fixed_alpha_s**, and a running α_S , calculated using **ps_fsr_lambda** for FSR. (Default = true)
- ps_fixed_alpha_s** Fixed value of α_S for the parton shower. Used if either one of the variables **?ps_fsr_alpha_s_running** or **?ps_isr_alpha_s_running** are set to false. (Default = 0.0)
- ?ps_isr_angular_ordered** Switch for angular ordered ISR. (Default = true)¹
- 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)

¹The FSR is always simulated with angular ordering enabled.

Settings for the `PYTHIA` are transferred using the following `SINDARIN` variables:

<code>?ps_PYTHIA_verbose</code>	if set to false, output from <code>PYTHIA</code> will be suppressed
<code>\$ps_PYTHIA_PYGIVE</code>	a string containing settings transferred to <code>PYTHIA</code> 's <code>PYGIVE</code> subroutine. The format is explained in the <code>PYTHIA</code> manual. The limitation to 100 characters mentioned there does not apply here, the string is split appropriately before being transferred to <code>PYTHIA</code> .

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`.

11.5.2 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 [44]. 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 or 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 [\[44\]](#)).

11.6 Negative weight events

Chapter 12

User Code Plug-Ins

Note that the user-code plug-in mechanism has been currently (for version 2.2.0) disabled, as the huge refactoring of the code between versions 2.1.X and 2.2.X has completely changed many of the interfaces. We plan to bring the interface for user code for spectra, structure functions and event shapes, cuts and observables back online as soon as possible, at latest for version 2.3.0.

12.1 The plug-in mechanism

The capabilities of `WHIZARD` and its `SINDARIN` command language are not always sufficient to adapt to all users' needs. To make the program more versatile, there are several spots in the workflow where the user may plug in his/her own code, to enhance or modify the default behavior.

User code can be injected, without touching `WHIZARD`'s source code, in the following places:

- Cuts, weights, analysis, etc.:
 - Cut functions that operate on a whole subevent.
 - Observable (e.g., event shapes) calculated from a whole subevent.
 - Observable calculated for a particle or particle pair.
- Spectra and structure functions.

Additional plug-in locations may be added in the future.

User code is loaded dynamically by `WHIZARD`. There are two possibilities:

1. The user codes the required procedures in one or more Fortran source files that are present in the working directory of the `WHIZARD` program. `WHIZARD` is called with the `-u` flag:

```
whizard -u --user-src=user-source-code-file ...
```

The file must have the extension `.f90`, and the file name must be specified without extension.

There may be an arbitrary number of user source-code files. The compilation is done in order of appearance. If the name of the user source-code file is `user.f90`, the flag `--user-src` can be omitted.

This tells the program to compile and dynamically link the code at runtime. The base-name of the linked library is `user`.

If a compiled (shared) library with that name already exists, it is taken as-is. If the user code changes or the library becomes invalid for other reasons, recompilation of the user-code files can be forced by the flag `--rebuild-user` or by the generic `-r` flag.

2. The user codes and compiles the required procedures him/herself. They should be provided in form of a library, where the interfaces of the individual procedures follow C calling conventions and exactly match the required interfaces as described in the following sections. The library must be compiled in such a way that it can be dynamically linked. If the calling conventions are met, the actual user code may be written in any programming language. E.g., it may be coded in Fortran, C, or C++ (with `extern(C)` specifications).

WHIZARD is called with the `-u` flag and is given the name of the user library as

```
whizard -u --user-lib=user-library-file ...
```

The library file should either be a dynamically loadable (shared) library with appropriate extension (`.so` on Linux), or a libtool archive (`.la`).

The user-provided procedures may have arbitrary names; the user just has to avoid clashes with procedures from the Fortran runtime library or from the operating-system environment.

12.2 Data Types Used for Communication

Since the user-code interface is designed to be interoperable with C, it communicates with WHIZARD only via C-interoperable data types. The basic data types (Fortran: integer and real kinds) `c_int` and `c_double` are usually identical with the default kinds on the Fortran side. If necessary, explicit conversion may be inserted.

For transferring particle data, we are using a specific derived type `c_prt_t` which has the following content:

```
type, bind(C) :: c_prt_t
  integer(c_int) :: type
  integer(c_int) :: pdg
  integer(c_int) :: polarized
  integer(c_int) :: h
  real(c_double) :: pe
```

```

    real(c_double) :: px
    real(c_double) :: py
    real(c_double) :: pz
    real(c_double) :: p2
end type c_prt_t

```

The meaning of the entries is as follows:

type: The type of the particle. The common type codes are 1=incoming, 2=outgoing, and 3=composite. A composite particle in a subevent is created from a combination of individual particle momenta, e.g., in jet clustering. If the status code is not defined, it is set to zero.

pdg: The particle identification code as proposed by the Particle Data Group. If undefined, it is zero.

polarized: If nonzero, the particle is polarized. The only polarization scheme supported at this stage is helicity. If zero, particle polarization is ignored.

h: If the particle is polarized, this is the helicity. 0 for a scalar, ± 1 for a spin-1/2 fermion, $-1, 0, 1$ for a spin-1 boson.

pe: The energy in GeV.

px/py: The transversal momentum components in GeV.

pz: The longitudinal momentum component in GeV.

p2: The invariant mass squared of the actual momentum in GeV^2 .

WHIZARD does not provide tools for manipulating `c_prt_t` objects directly. However, the four-momentum can be used in Lorentz-algebra calculations from the `lorentz` module. To this end, this module defines the transformational functions `vector4_from_c_prt` and `vector4_to_c_prt`.

12.3 User-defined Observables and Functions

12.3.1 Cut function

Instead of coding a cut expression in SINDARIN, it may be coded in Fortran, or in any other language with a C-compatible interface. A user-defined cut expression is referenced in SINDARIN as follows:

```
cuts = user_cut (name-string) [subevent]
```

The *name-string* is an expression that evaluates to string, the name of the function to call in the user code. The *subevent* is a subevent expression, analogous to the built-in cut definition syntax. The result of the `user_cut` function is a logical value in SINDARIN. It is true if the event passes the cut, false otherwise.

If coded in Fortran, the actual user-cut function in the user-provided source code has the following form:

```
function user_cut_fun (prt, n_prt) result (iflag) bind(C)
  use iso_c_binding
  use c_particles
  type(c_prt_t), dimension(*), intent(in) :: prt
  integer(c_int), intent(in) :: n_prt
  integer(c_int) :: iflag
  ! ... code that evaluates iflag
end function user_cut_fun
```

Here, `user_cut_fun` can be replaced by an arbitrary name by which the function is referenced as *name-string* above. The `bind(C)` attribute in the function declaration is mandatory.

The argument `prt` is an array of objects of type `c_prt_t`, as described above. The integer `n_prt` is the number of entries in the array. It is passed separately in order to determine the actual size of the assumed-size `prt` array.

The result `iflag` is an integer. A nonzero value indicates **true** (i.e., the event passes the cut), zero value indicates **false**. (We do not use boolean values in the interface because their interoperability might be problematic on some systems.)

12.3.2 Event-shape function

An event-shape function is similar to a cut function. It takes a subevent as argument and returns a real (i.e., C double) variable. It can be used for defining subevent observables, event weights, or the event scale, as in

```
analysis = record hist-id (user_event_fun (name-string) [subevent])
```

or

```
scale = user_event_fun (name-string) [subevent]
```

The corresponding Fortran source code has the form

```
function user_event_fun (prt, n_prt) result (rval) bind(C)
  use iso_c_binding
  use c_particles
  type(c_prt_t), dimension(*), intent(in) :: prt
  integer(c_int), intent(in) :: n_prt
  real(c_double) :: rval
  ! ... code that evaluates rval
end function user_event_fun
```

with `user_event_fun` replaced by *name-string*.

12.3.3 Observable

In SINDARIN, an observable-type function is a function of one or two particle objects that returns a real value. The particle objects result from scanning over subevents. In the SINDARIN code, the observable is used like a variable; the particle-object arguments are implicitly assigned.

A user-defined observable is used analogously, e.g.,

```
cuts = all user_obs (name-string) > 0 [subevent]
```

The user observable is defined, as Fortran code, as either a unary or as a binary C-double-valued function of `c_prt_t` objects. The use in SINDARIN (unary or binary) must match the definition.

```
function user_obs_unary (prt1) result (rval) bind(C)
  use iso_c_binding
  use c_particles
  type(c_prt_t), intent(in) :: prt1
  real(c_double) :: rval
  ! ... code that evaluates rval
end function user_obs_unary
```

or

```
function user_obs_binary (prt1, prt2) result (rval) bind(C)
  use iso_c_binding
  use c_particles
  type(c_prt_t), intent(in) :: prt1, prt2
  real(c_double) :: rval
  ! ... code that evaluates rval
end function user_obs_binary
```

with `user_obs_unary/binary` replaced by *name-string*.

12.3.4 Examples

For an example, we implement three different ways of computing the transverse momentum of a particle. This observable is actually built into WHIZARD, so the examples are not particularly useful. However, implementing kinematical functions that are not supported (yet) by WHIZARD (and not easily computed via SINDARIN expressions) proceeds along the same lines.

Cut

The first function is a complete cut which can be used as

```
cuts = user_cut("ptcut") [subevt]
```

It is equivalent to

```
cuts = all Pt > 50 [subevt]
```

The implementation reads

```

function ptcut (prt, n_prt) result (iflag) bind(C)
  use iso_c_binding
  use c_particles
  use lorentz
  type(c_prt_t), dimension(*), intent(in) :: prt
  integer(c_int), intent(in) :: n_prt
  integer(c_int) :: iflag
  logical, save :: first = .true.
  if (all (transverse_part (vector4_from_c_prt (prt(1:n_prt)))) > 50)) then
    iflag = 1
  else
    iflag = 0
  end if
end function ptcut

```

The procedure makes use of the kinematical functions in the `lorentz` module, after transforming the particles into a `vector4` array.

Event Shape

Similar functionality can be achieved by implementing an event-shape function. The function computes the minimum p_T among all particles in the subevent. The SINDARIN expression reads

```
cuts = user_event_shape("pt_min") [subevt] > 50
```

and the function is coded as

```

function pt_min (prt, n_prt) result (rval) bind(C)
  use iso_c_binding
  use c_particles
  use lorentz
  type(c_prt_t), dimension(*), intent(in) :: prt
  integer(c_int), intent(in) :: n_prt
  real(c_double) :: rval
  rval = minval (transverse_part (vector4_from_c_prt (prt(1:n_prt))))
end function pt_min

```

Observable

The third (and probably simplest) user implementation of the p_T cut computes a single-particle observable. Here, the usage is

```
cuts = all user_obs("ptval") > 50 [subevt]
```

and the subroutine reads

```

function ptval (prt1) result (rval) bind(C)
  use iso_c_binding
  use c_particles
  use lorentz
  type(c_prt_t), intent(in) :: prt1
  real(c_double) :: rval
  rval = transverse_part (vector4_from_c_prt (prt1))
end function ptval

```


12.4 Spectrum or Structure Function

12.4.1 Definition

User-defined spectra or structure functions can be used in a `beams` definition just like ordinary ones (`isr`, `pdf_builtin`, etc.), for instance:

```
beams = p, p => user_sf (name-string)
```

or

```
beams = p, "e-" => user_sf (name-string), none
```

To implement a particular spectrum or structure function, the user must code five different procedures. Their names all must begin with *name-string* with specific suffixes appended. In the following, replace `user_sf` by whatever *name-string* has been chosen:

1. `user_sf_info`:

This subroutine tells WHIZARD about static information on the spectrum. In the user-provided source code, the function takes the form (if it is coded in Fortran):

```
subroutine user_sf_info (n_in, n_out, n_states, n_col, n_dim, n_var) bind(C)
  use iso_c_binding
  integer(c_int), intent(inout) :: n_in, n_out, n_states, n_col
  integer(c_int), intent(inout) :: n_dim, n_var
  ! ... code that sets the parameters
end subroutine user_sf_info
```

The subroutine arguments describe the overall properties of the spectrum. For all arguments, there exist default values which apply if the parameter is not set in the subroutine.

n_in Number of incoming particles. This is 1 for a single-particle spectrum (default), or 2 for a two-particle spectrum.

n_out Number of outgoing particles. Should be greater or equal to **n_in**. Default is 2.

n_states Number of distinct quantum states that are supported by the spectrum. Each possible combination of flavor, color, and helicity counts as a separate state. Quantum numbers that are ignored (e.g., unpolarized particles) do not count. Default is 1.

n_col Maximal number of distinct color-flow lines attached to any particle. In the Standard Model, this is 2 or less. Usually, the default value (2) should be left untouched.

n_dim Number of independent integration parameters on which the spectrum depends. For instance, a parton structure function depends on a single parameter, while ISR radiation with generated transverse momentum depends on three parameters. Default is 1.

n_var Number of real parameters that have to be communicated from the kinematics routine to the evaluation routine. For instance, a PDF depends on one variable (x) that is used both for computing the momentum and for evaluating the structure function, so it must be communicated. Default value is 1.

2. `user_sf_mask`:

This subroutine tells **WHIZARD** which quantum numbers are explicit and which ones are ignored. The default is that all quantum numbers are explicit, but it may be appropriate to ignore the polarization of specific particles, e.g., a radiated photon.

```
subroutine user_sf_mask (i_prt, m_flv, m_col, m_hel, i_lock) bind(C)
  use iso_c_binding
  integer(c_int), intent(in) :: i_prt
  integer(c_int), intent(inout) :: m_flv, m_hel, m_col, i_lock
  ! ... code that sets m_flv, m_hel, m_col, i_lock
end subroutine user_sf_mask
```

The arguments define the properties of a specific particle within the interaction that the spectrum describes.

i_prt This is the index of the particle for which the info is requested. **i_prt**, which is a value between 1 and **n_in** + **n_out**. For each possible value of **i_prt**, the subroutine should set the appropriate flags, unless the default values are correct.

m_flv If nonzero, the flavor of this particle is unspecified. In the process, the spectrum interaction matches any particle. Default is 0.

m_col If nonzero, the color of this particle is unspecified. In the process, the spectrum interaction matches any color assignment, and the attached color lines are ignored for the whole process. Default is 0.

m_hel If nonzero, the helicity of this particle is unspecified. In the process, the spectrum interaction matches any helicity assignment, i.e., the particle is treated as unpolarized. Default is 0.

i_lock If nonzero, this indicates a helicity conservation rule. The current particle (**i_prt**) and the particle with index **i_lock** are declared to have the same helicity.

The particle with index **i_lock** must have a corresponding entry pointing to index **i_prt**. Of course, the conservation rule must be satisfied by all quantum-number combination.

(The conservation rule improves efficiency when a beam is declared as unpolarized. (De-)polarization is carried through the structure-function chain, so that it is the hard matrix element which is effectively averaged over polarizations.)

3. `user_sf_state`:

This subroutine tells **WHIZARD** which quantum number combinations are allowed in the spectrum. For each particles, only the quantum numbers allowed by the corresponding mask value (see above) have to be set.

```

subroutine user_sf_state (i_state, i_prt, flv, hel, col) bind(C)
  use iso_c_binding
  integer(c_int), intent(in) :: i_state, i_prt
  integer(c_int), intent(inout) :: flv, hel
  integer(c_int), dimension(*), intent(inout) :: col
  ! ... code that sets flv, hel, col
end subroutine user_sf_state

```

Given `i_state` and `i_prt`, the routine should return appropriate values for the other parameters or leave them at zero (default).

`i_state` Index of a quantum number combination, a number between 1 and `n_states`.

`i_prt` Index of a particle in the interaction, a number between 1 and `n_in + n_out`. The incoming particles come before the outgoing particles in the interaction.

`flv` PDG code of the particle.

`hel` Helicity value of the particle, as described above in Sec. 12.2.

`col` Array of color-flow indices of the particle. The size of the array is `n_col`. Color connections between particles are indicated by coinciding color-flow indices. By convention, color-flow indices are integer values of 500 and greater.

4. `user_sf_kinematics`:

This subroutine generates the momenta for the outgoing particles, given the momenta of incoming particles and an array of parameters.

```

subroutine user_sf_kinematics (prt_in, rval, prt_out, xval) bind(C)
  use iso_c_binding
  use c_particles
  type(c_prt_t), dimension(*), intent(in) :: prt_in
  real(c_double), dimension(*), intent(in) :: rval
  type(c_prt_t), dimension(*), intent(inout) :: prt_out
  real(c_double), dimension(*), intent(out) :: xval
  ! ... code that computes prt_out and xval
end subroutine user_sf_kinematics

```

`prt_in` Array of incoming particles as generated by WHIZARD. Only the energy-momentum entries in the `c_prt_t` objects are relevant, the others are meaningless at this stage. The array size is `n_in`.

`rval` Array of real parameters, sufficient for calculating the outgoing four-momenta. The array size is `n_dim`.

`prt_out` Array of outgoing particles. They must be computed by the user-defined code. Only the energy-momentum entries will be used. The array size is `n_out`.

`xval` Array of parameters that are needed for the evaluation routine below, to uniquely determine the spectrum values. For instance, these could be the x momentum fraction(s) and possibly an extra Jacobian factor. The array size is `n_var`.

5. `user_sf_evaluate`:

This subroutine computes the values (the squared matrix elements) of the spectrum, one for each quantum-number combination, given the variables returned by the kinematics routine above and the energy scale of the event.

```
subroutine user_sf_evaluate (xval, scale, fval) bind(C)
  use iso_c_binding
  real(c_double), dimension(*), intent(in) :: xval
  real(c_double), intent(in) :: scale
  real(c_double), dimension(*), intent(out) :: fval
end subroutine user_sf_evaluate
```

xval Array of parameters as returned by the kinematics routine. The array size is **n_var**.

scale Energy scale in GeV computed by **WHIZARD** (using the **SINDARIN** expression for **scale**) for the current kinematics. The spectrum evaluation can use this scale. Alternatively, the kinematics routine may compute the scale and transfer it to the evaluation via the **xval** array, which ignores the **scale** argument.

fval Value array of the structure function. For each state (combination of quantum numbers) there must be one value. The array size is **n_states**.

12.4.2 Example

For a simple example, we reproduce the effect of an **energy_scan** spectrum for electrons, analogous to the generic one that is built into **WHIZARD**. We assume that a fraction of the energy is transferred from the beam to the incoming parton. In contrast to the actual energy-scan spectrum where no further particle is involved, we transfer the remaining energy to a radiated photon. The user-defined spectrum can be used as in

```
beams = "e-", "e-" => user_sf ("escan")
```

where it is applied to both beams, independently.

We are considering an interaction with one incoming and two outgoing colorless particles. There are two quantum-number states, one input parameter, and no output parameters since the matrix element of this spectrum is constant (unity).

```
subroutine escan_info (n_in, n_out, n_states, n_col, n_dim, n_var) bind(C)
  use iso_c_binding
  integer(c_int), intent(inout) :: n_in, n_out, n_states, n_col
  integer(c_int), intent(inout) :: n_dim, n_var
  n_in = 1
  n_out = 2
  n_states = 2
  n_dim = 1
  n_var = 0
end subroutine escan_info
```

The mask is set up such that polarization is transferred from the incoming to the outgoing particle (locking them together), and the radiated photon is unpolarized.

```

subroutine escan_mask (i_prt, m_flv, m_col, m_hel, i_lock) bind(C)
  use iso_c_binding
  integer(c_int), intent(in) :: i_prt
  integer(c_int), intent(inout) :: m_flv, m_hel, m_col, i_lock
  select case (i_prt)
  case (1)
    i_lock = 3
  case (2)
    m_hel = 1
  case (3)
    i_lock = 1
  end select
end subroutine escan_mask

```

There are two quantum-number states, one with negative and one with positive helicity for both incoming and outgoing electron. They are color singlets (no color index), and the flavor of the three particles is electron, photon, electron.

```

subroutine escan_state (i_state, i_prt, flv, hel, col) bind(C)
  use iso_c_binding
  integer(c_int), intent(in) :: i_state, i_prt
  integer(c_int), intent(inout) :: flv, hel
  integer(c_int), dimension(*), intent(inout) :: col
  select case (i_prt)
  case (1, 3)
    flv = 11
    select case (i_state)
    case (1); hel = -1
    case (2); hel = 1
    end select
  case (2)
    flv = 22
  end select
end subroutine escan_state

```

The kinematics routine computes the x value as $x = r^2$, where r is the integration parameter. The four-momenta are simply scaled by the momentum fraction, assuming zero mass.

```

subroutine escan_kinematics (prt_in, rval, prt_out, xval) bind(C)
  use iso_c_binding
  use c_particles
  use kinds
  use lorentz
  type(c_prt_t), dimension(*), intent(in) :: prt_in
  real(c_double), dimension(*), intent(in) :: rval
  type(c_prt_t), dimension(*), intent(inout) :: prt_out
  real(c_double), dimension(*), intent(out) :: xval
  type(vector4_t), dimension(3) :: p
  real(default) :: x
  x = rval(1)**2
  p(1) = vector4_from_c_prt (prt_in(1))
  p(2) = (1-x) * p(1)

```

```

    p(3) = x * p(1)
    prt_out(1:2) = vector4_to_c_prt (p(2:3))
end subroutine escan_kinematics

```

Finally, the evaluation is trivial. For each quantum-number state, the matrix element is unity.

```

subroutine escan_evaluate (xval, scale, fval) bind(C)
  use iso_c_binding
  real(c_double), dimension(*), intent(in) :: xval
  real(c_double), intent(in) :: scale
  real(c_double), dimension(*), intent(out) :: fval
  fval(1) = 1
  fval(2) = 1
end subroutine escan_evaluate

```

12.5 User Code and Static Executables

In Sec. 5.4.7 we describe how to build a static executable that can be submitted to batch jobs, e.g., on the grid, where a compiler may not be available.

If there is user plug-in code, it would require the same setup of libtool, compiler and linker on the target host, as physical process code. To avoid this, it is preferable to link the user code statically with the executable, which is then run as a monolithic program.

This is actually simple. Two conditions have to be met:

1. The WHIZARD job that creates the executable has to be given the appropriate options (`-u`, `--user-src`, `--user-lib`) such that the user code is dynamically compiled and linked.
2. The compile command in the SINDARIN script which creates the executable takes options that list the procedures which the stand-alone program should access:

```

compile as "executable-name" {
  $user_procs_cut = "cut-proc-names"
  $user_procs_event_shape = "event-shape-proc-names"
  $user_procs_obs1 = "obs1-proc-names"
  $user_procs_obs2 = "obs2-proc-names"
  $user_procs_sf = "strfun-names"
}

```

The values of these option variables are comma-separated lists of procedure names, grouped by their nature. `obs1` and `obs2` refer to unary and binary observables, respectively. The `strfun-names` are the names of the user-defined spectra or structure functions as they would appear in the SINDARIN file which uses them.

With these conditions met, the stand-alone executable will have the user code statically linked, and it will be able to use exactly those user-defined routines that have been listed in the various option strings. (It is possible nevertheless, to plug in additional user code into the stand-alone executable, using the same options as for the original WHIZARD program.)

Chapter 13

Data Visualization

13.1 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.
3. The driver file is processed by \LaTeX , 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 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.

13.2 Histogram Display

13.3 Plot Display

13.4 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, compiled 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.

Table 13.1: *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
<code>\$title</code>	<code>""</code>	\LaTeX	Title of the graph = subsection headline
<code>\$description</code>	<code>""</code>	\LaTeX	Description text for the graph
<code>\$x_label</code>	<code>""</code>	\LaTeX	x -axis label
<code>\$y_label</code>	<code>""</code>	\LaTeX	y -axis label
<code>graph_width_mm</code>	130	Integer	graph width (on paper) in mm
<code>graph_height_mm</code>	90	Integer	graph height (on paper) in mm
<code>?x_log</code>	false	Logical	Whether the x -axis scale is linear or logarithmic
<code>?y_log</code>	false	Logical	Whether the y -axis scale is linear or logarithmic
<code>x_min</code>	<i>undefined</i>	Real	Lower bound for the x axis
<code>x_max</code>	<i>undefined</i>	Real	Upper bound for the x axis
<code>y_min</code>	<i>undefined</i>	Real	Lower bound for the y axis
<code>y_max</code>	<i>undefined</i>	Real	Upper bound for the y axis
<code>gmlcode_bg</code>	<code>""</code>	GAMELAN	Code to be executed before drawing
<code>gmlcode_fg</code>	<code>""</code>	GAMELAN	Code to be executed after drawing

A simple graph declaration without options as above is possible, but usually there are option 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 assignment. 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 13.1 and 13.2, respectively.

Table 13.2: *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	<i>undefined</i>	Logical	Whether to draw a baseline for the curve
?draw_pieewise	<i>undefined</i>	Logical	Whether to draw bins separately (histogram)
?fill_curve	<i>undefined</i>	Logical	Whether to fill area between baseline and curve
\$fill_options	" "	GAMELAN	Options for filling the area
?draw_curve	<i>undefined</i>	Logical	Whether to draw the curve as a line
\$draw_options	" "	GAMELAN	Options for drawing the line
?draw_errors	<i>undefined</i>	Logical	Whether to draw error bars for data points
\$err_options	" "	GAMELAN	Options for drawing the error bars
?draw_symbols	<i>undefined</i>	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

13.5 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 possibilities, 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:

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:

(r, g, b)

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

`evenly, withdots, withdashdots`

Going beyond the predefined patterns, a generic dash pattern has the syntax

`dashpattern (on l1 off l2 on ...)`

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*:

`withstripes, withlines`

and a generic hatch pattern is written

`hatchpattern (on w1 off w2 on ...)`

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 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.

`--execute COMMANDS` : Execute COMMANDS as a script before the script file. Short version: `-e`

`--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-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.
`--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.
`--version` : Print version information and exit. Short version: `-V`.
`-` : Any further options are interpreted as filenames.

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. Cf. Sec. 2.4.3:

`--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.
`--lhpdfdir DIR` : Specify the actual location and of the LHAPDF installation that should be used by WHIZARD.

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 planned, but not implemented yet.

14.4 WHIZARD as a library

This is planned, but not implemented yet.

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 W pairs at LEP

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 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 = 0.0000000000000000E+00
sqrts = 2.0900000000000000E+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

```

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 Models via FeynRules

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, `!`.

- `+`
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 (\rightarrow) 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 `(→) #`.

- #
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, %F, %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 appropriate. 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, %G, %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 `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`)
- **accuracy_goal** (default: 0./off)
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 adaptation stop, and this result is used for simulation. (cf. also `integrate`, `iterations`, `error_goal`, `relative_error_goal`, `error_threshold`)
- **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`)
- **?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_shower** (default: true)
Master flag to switch on (initial and final state) parton shower, matching/merging and hadronization as an event transform. As a default, it is switched on. (cf. also `?ps_`, `$ps_`, `?mlm_`, `?hadronization_active`)
- **?alpha_s_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 `?alpha_s_is_fixed` has to be set explicitly to `false`. (cf. also `alpha_s_order`, `?alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `alpha_s_nf`, `?alpha_s_from_pdf_builtin`, `?alpha_s_from_mz`, `lambda_qcd`)
- **?alpha_s_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 `?alpha_s_is_fixed` has to be set explicitly to `false`. (cf. also `alpha_s_order`, `?alpha_s_is_fixed`, `?alpha_s_from_pdf_builtin`, `alpha_s_nf`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)

- `?alpha_s_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 `?alpha_s_is_fixed` has to be set explicitly to false. (cf. also `alpha_s_order`, `?alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `alpha_s_nf`, `?alpha_s_from_pdf_builtin`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)
- `?alpha_s_from_pdf_builtin` (default: false)
Flag that tells WHIZARD to use a running α_s from the internal PDFs. Note that in that case `?alpha_s_is_fixed` has to be set explicitly to false. (cf. also `alpha_s_order`, `?alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `alpha_s_nf`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)
- `?alpha_s_is_fixed` (default: true)
Flag that tells WHIZARD to use a non-running 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 `alpha_s_order`, `?alpha_s_from_lhapdf`, `?alpha_s_from_pdf_builtin`, `alpha_s_nf`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)
- `alpha_s_nf` (default: 5)
Integer parameter that sets the number of active quark flavors for the internal evolution for running α_s in WHIZARD: the default is 5. (cf. also `alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `?alpha_s_from_pdf_builtin`, `alpha_s_order`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)
- `alpha_s_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 `alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `?alpha_s_from_pdf_builtin`, `alpha_s_nf`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `lambda_qcd`)
- `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**)
- **?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 multiplicity (\rightarrow) **auto_decays_multiplicity**. Depending on the flag (\rightarrow) **?auto_decays_radiative**, radiative decays will be taken into account or not. (cf. also **unstable**, **?decay_rest_frame**, **?isotropic_decay**, **?diagonal_decay**)
- **auto_decays_multiplicity** (default: 2)
 Integer parameter, that sets – for the (\rightarrow) **?auto_decays** option to let WHIZARD automatically 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 **?decay_rest_frame**, **?isotropic_decay**, **?diagonal_decay**)
- **?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 flag decides whether radiative decays (e.g. containing additional photon(s)/gluon(s)) are taken into account or not.
(cf. also `auto_decays_multiplicity`, `?decay_rest_frame`, `?isotropic_decay`, `?diagonal_decay`)

- `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`)

- `$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`)

- `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 => lhpdf`. Structure functions can be specified for one of the two beam particles only, if 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`, `lhpdf`, `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`)

- `channel_weights_power` (default: 0.25)

Real parameter that allows to vary the exponent of the channel weights for the VAMP integrator.

- `?check_events_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_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_events_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_events_file`, `?check_grid_file`)
- `checkpoint` (default: 0/off)
Setting this integer variable to a positive integer n instructs simulate to print out a progress summary every n events.
- `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`, `$circe1_acc`, `circe1_chat`)
- `$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_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`, `$circe1_acc`)
- `circe1_eps` (default: 10^{-5})
Real parameter, that takes care of the mapping of the peak in the lepton collider beamstrahlung structure function spectrum of `CIRCE1`. (cf. also `circe1`, `?circe1_photons`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`)
- `?circe1_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_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`, `$circe1_acc`, `circe1_chat`)
- `circe1_mapping_slope` (default: 2.)
Real parameter that allows to vary the slope of the mapping function for the CIRCE1 structure function for lepton collider beamstrahlung from the default value 2.. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`)
- `?circe1_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`, `$circe1_acc`, `circe1_chat`)
- `?circe1_photon2` (default: `false`)
Flag to tell WHIZARD to use the photon of the CIRCE1 beamstrahlung structure function as initiator for the hard scattering process in the second beam. (cf. also `circe1`, `?circe1_photon1`, `circe1_sqrts`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`)
- `circe1_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 CIRCE1manual. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_sqrts`, `circe1_ver`, `$circe1_acc`, `circe1_chat`)
- `circe1_sqrts` (default: 0/internal \sqrt{s})
Real parameter that allows to set the value of the collider energy for the lepton collider beamstrahlung structure function CIRCE1. If not set, \sqrt{s} is taken. (cf. also `circe1`, `?circe1_photon1`, `?circe1_photon2`, `?circe1_generate`, `?circe1_map`, `circe1_eps`, `circe1_mapping_slope`, `circe1_ver`, `circe1_rev`, `$circe1_acc`, `circe1_chat`)
- `circe1_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`)

- **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`)
- **\$circe2_design** (default: `"*/CIRCE2 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_ ...`, `$ps_ ...`, `?mlm_ ...`)
- **clear**
This command allows to clear a variable set before: `clear (<clearable var.>)` resets the variable `<clearable var.>` which could be the `beams`, the `unstable` settings, `sqrts`, any kind of `cuts` or `scale` expressions, any user-set variable etc. The syntax of the command is completely analogous to (\rightarrow) `show`.
- **close_out**
With the command, `close_out ("<out_file">)` user-defined information like data or (\rightarrow) `printf` statements can be written out to a user-defined file. The command closes an I/O stream to an external file `<out_file>`. (cf. also `open_out`, `$out_file`, `printf`)
- **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 be 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 `$out_file` is provided, the histogram tables/plot data etc. are written to the default file `whizard_analysis.dat`. In addition to `write_analysis`, `compile_analysis` also invokes the WHIZARD L^AT_EX routines for producing postscript or PDF output of the data. (cf. also `$out_file`, `compile_analysis`)

- **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, e.g. `PDG == 15` would select a tau lepton. 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`)
- `?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`)
- `$debug_extension`
String variable that allows via `$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_rest_frame` (default: `false`)
Flag that allows to force a particle decay to be simulated in its rest frame. Usually, particularly as part of the `unstable` command, this is not the case. (cf. also `?auto_decays`,

auto_decays_multiplicity, ?auto_decays_radiative, ?isotropic_decay,
?diagonal_decay)

- **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`).
- **\$description** (default: `"/off`)
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_pieewise`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)
- **?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 `?decay_rest_frame`, `?auto_decays`, `auto_decays_multiplicity`, `?auto_decays_radiative`, `?isotropic_decay`)
- **?diags** (default: `false`)
Logical variable that allows to give out a Postscript or PDF file for the Feynman diagrams for a 0'Mega process. (cf. `?diags_color`).
- **?diags_color** (default: `false`)
Same as `?diags`, but switches on color flow instead of Feynman diagram generation. (cf. `?diags`).
- **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`)
- **?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_pieewise`, `?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 \rightarrow `?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_pieewise`, `?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_pieewise`, `?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 \rightarrow `?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_pieewise`, `?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_pieewise`, `?fill_curve`, `?draw_histogram`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `?draw_histogram`, `$err_options`, `$symbol`)
- `?draw_pieewise`
Settings for WHIZARD's internal graphics output: flag that tells WHIZARD to data from a data set pieewise, i.e. histogram style. (cf. also `?normalize_bins`, `$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`, `$err_options`)
- `?draw_symbols`
Settings for WHIZARD's internal graphics output: flag that determines whether particular symbols (specified by \rightarrow `$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_pieewise`, `?fill_curve`, `?draw_histogram`, `?draw_curve`, `?draw_errors`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)

- **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> thenendif`. (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`)
- **?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**
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_e_max`, `epa_q_min`, `?epa_recoil`)
- **epa_alpha** (default: 0/internal from model)
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_e_max** (default: 0/internal \sqrt{s})
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_mass` (default: 0/internal from model)
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_q_min` (default: 0)
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_e_max`, `?epa_recoil`)
- `?epa_recoil` (default: false)
Flag to switch on recoil, i.e. a non-vanishing p_T -kick for the equivalent-photon approximation (EPA). (cf. also `epa`, `epa_x_min`, `epa_mass`, `epa_alpha`, `epa_e_max`, `epa_q_min`)
- `epa_x_min` (default: 0)
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`)
- `$err_options`
Settings for WHIZARD's internal graphics output: `$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_pieewise`, `?fill_curve`, `?draw_histogram`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `?draw_histogram`, `$draw_options`, `$symbol`)
- `error_goal` (default: 0./off)
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 adaption stop, and this result is used for simulation. (cf. also `integrate`, `iterations`, `accuracy_goal`, `relative_error_goal`, `error_threshold`)
- `error_threshold` (default: 0.)
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`)
- `Eta`
Unary and also binary observable specifier, that as a unary observable gives the pseudorapidity of a particle momentum. The pseudorapidity is given by $\eta = -\log[\tan(\theta/2)]$,

where θ is the angle with the beam direction. As a binary observable, it gives the pseudo-rapidity 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`)
- **\$event_file_version** (default: `"/internal`)
String variable that allows to set the format version of the WHIZARD internal binary event format.
- **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_keep_momentum`)
- **?ewa_keep_energy** (default: `false`)
For the equivalent W approximation (EWA), this flag switches on recoil, i.e. non-collinear splitting. As the splitting kinematics violates Lorentz invariance, this flag forces energy conservation, while violating momentum conservation. (cf. also `ewa`, `ewa_x_min`, `ewa_pt_max`, `ewa_mass`, `?ewa_keep_momentum`)
- **?ewa_keep_momentum** (default: `false`)
For the equivalent W approximation (EWA), this flag switches on recoil, i.e. non-collinear splitting. As the splitting kinematics violates Lorentz invariance, this flag forces momentum conservation, while violating energy conservation. (cf. also `ewa`, `ewa_x_min`, `ewa_pt_max`, `ewa_mass`, `?ewa_keep_energy`)
- **ewa_mass** (default: `0/internal from model`)
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_keep_momentum`)
- **ewa_pt_max** (default: `0/internal \sqrt{s}`)
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_keep_momentum`)

- `ewa_x_min` (default: 0)
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_keep_momentum`)
- `exec`
Constructor `exec ("<cmd_name>")` that demands WHIZARD to execute/run the command `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 \rightarrow `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 be 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`).
- `$extension_ascii_long` (default: "long.evt")
String variable that allows via `$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_debug`: old version of (cf. \rightarrow) `$debug_extension`
- `$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_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 format are written. If not set, the default file name and suffix is `<process_name>.lha.verb`. (cf. also `sample_format`, `$sample`)
- `$extension_lhef`: old version of (cf. \rightarrow) `$lhef_extension`

- `$extension_mokka` (default: "mokka.evt")
String variable that allows via `$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`)
- `$extension_stdhep` (default: "stdhep")
String variable that allows via `$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>.stdhep`. (cf. also `sample_format`, `$sample`)
- `$extension_stdhep_up` (default: "up.stdhep")
String variable that allows via `$extension_stdhep = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in the StdHEP format via the HEPRUP/HEPEUP common blocks are written. `<process_name>.up.stdhep` is the default file name and suffix, if this variable not set. (cf. also `sample_format`, `$sample`)
- `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 (\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 `renormalization_scale`).
- `false`
Constructor stating that a logical expression or variable is false, e.g. `?<log_var> = false`. (cf. also `true`).
- `?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 $bt \rightarrow X$, where the bottom quark as an initial 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.

- **fbarn**
Physical unit, stating that a number is in femtobarns (10^{-15} barn). (cf. also nbarn, abarn, pbarn)
- **?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 \rightarrow `$fill_options = "<LaTeX_code>"`. (cf. also ?normalize_bins, \$obs_label, \$obs_unit, \$title, \$description, \$x_label, \$y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, \$gmlcode_fg, \$gmlcode_bg, ?draw_base, ?draw_pieewise, ?draw_curve, ?draw_histogram, ?draw_errors, ?draw_symbols, \$fill_options, \$draw_options, \$err_options, \$symbol)
- **\$fill_options**
Settings for WHIZARD's internal graphics output: `$fill_options = "<LaTeX_code>"` is a string variable that allows to set fill options when plotting data as filled curves with the \rightarrow ?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_pieewise, ?draw_curve, ?draw_histogram, ?draw_errors, ?draw_symbols, ?fill_curve, \$draw_options, \$err_options, \$symbol)
- **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)
- **GeV**
Physical unit, energies in 10^9 electron volt. This is the default energy unit of WHIZARD. (cf. also eV, keV, MeV, meV, TeV)
- **\$gmlcode_bg** (default: ""/off)
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: `$gmlcode_bg = "standardgrid.lr(5);"`. For more details, see the gamelan manual. (cf. also ?normalize_bins, \$obs_label, \$obs_unit, \$title, \$description, \$x_label, \$y_label, graph_width_mm, graph_height_mm, ?y_log, ?x_log, x_min, x_max, y_min, y_max, \$gmlcode_fg, ?draw_histogram, ?draw_base, ?draw_pieewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, \$fill_options, \$draw_options, \$err_options, \$symbol)
- **\$gmlcode_fg** (default: ""/off)
Settings for WHIZARD's internal graphics output: string variable that allows to define a foreground for plots and histograms (i.e. it overwrites the plot/histogram), e.g. a grid: `$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_pieewise,
?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, $fill_options,
$draw_options, $err_options, $symbol)
```

- **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**)

- **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_pieewise*, *?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_pieewise*, *?fill_curve*, *?draw_curve*, *?draw_errors*, *?draw_symbols*, *\$fill_options*, *\$draw_options*, *\$err_options*, *\$symbol*)

- **?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_ ...*)

- **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**)

- **?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 (0'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 (0'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: 10^{10})
Real parameter that gives the threshold for the absolute value of a certain helicity combination of an (0'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`
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`)
- `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`)
- `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 expressions 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 depending 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 `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 integration (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 `$integration_method = "<intg. method>"` the integration method (the default being VAMP), (2) the number of iterations and calls per integration during the Monte-Carlo phase-space integration via the `iterations` specifier; (3) goal for the accuracy, error or relative error (`accuracy_goal`, `error_goal`, `relative_error_goal`). (4) Invoking only phase space generation (`?phs_only = true`), (5) making test calls of the matrix element. (cf. also `iterations`, `accuracy_goal`, `error_goal`, `relative_error_goal`, `error_threshold`)
- `$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").
- `?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 `?decay_rest_frame`, `?auto_decays`, `auto_decays_multiplicity`, `?diagonal_decay`, `?auto_decays_radiative`)

- **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_alpha** (default: 0/internal from model)
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_mass** (default: 0/internal from model)
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_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_q_max** (default: 0/internal \sqrt{s})
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_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_alpha`, `isr_mass`, `isr_order`, `isr_q_max`)
- **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`, `+`)

- `?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.
- `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, $Q^2 = 1$.
- `lambda_qcd` (default: `200 MeV`)
Real parameter that sets the value for Λ_{QCD} used in the internal evolution for running α_s in WHIZARD. (cf. also `alpha_s_is_fixed`, `?alpha_s_from_lhapdf`, `alpha_s_nf`, `?alpha_s_from_pdf_builtin`, `?alpha_s_from_mz`, `?alpha_s_from_lambda_qcd`, `alpha_s_order`)
- `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 `$sample`, `sample_format`)
- `lhpdf`
This is a beams specifier to demand calling LHAPDF parton densities as structure functions to integrate processes in hadron collisions. Note that this only works if the external LHAPDF library is present and correctly linked. (cf. `beams`, `$lhpdf_dir`, `$lhpdf_file`, `lhpdf_photon`, `$lhpdf_photon_file`, `lhpdf_member`, `lhpdf_photon_scheme`)
- `$lhpdf_dir` (default: `"/external`)
String variable that tells the path where the LHAPDF library and PDF sets can be found. When the library has been correctly recognized during configuration, this is automatically set by WHIZARD. (cf. also `lhpdf`, `$lhpdf_file`, `lhpdf_photon`, `$lhpdf_photon_file`, `lhpdf_member`, `lhpdf_photon_scheme`)
- `$lhpdf_file` (default: `"/external`)
This string variable `$lhpdf_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_hoppet_b_matching` (default: `false`)
If the external libraries LHAPDF and HOPPET are correctly linked, this allows to switch on the matching for 4-/5-flavor schemes for *b*-initiated processes at hadron colliders.
- `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`, `$lhapdf_photon_file`, `lhapdf_photon_scheme`)
- `lhapdf_photon`
This is a beams specifier to demand calling LHAPDF parton densities as structure functions to integrate processes in hadron collisions with a photon as initializer of the hard scattering process. Note that this only works if the external LHAPDF library is present and correctly linked. (cf. `beams`, `lhapdf`, `$lhapdf_dir`, `$lhapdf_file`, `$lhapdf_photon_file`, `lhapdf_member`, `lhapdf_photon_scheme`)
- `$lhapdf_photon_file` (default: `"/external`)
String variable `$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`, `$lhapdf_photon_file`, `lhapdf_member`)
- `lhef`
Specifier for the `sample_format` command to demand the generation of the Les Houches Accord (LHEF) event format files, with XML headers. There are several different versions of this format, which can be selected via the `$lhef_version` specifier (cf. also `$sample`, `sample_format`, `$lhef_version`, `$lhef_extension`, `?lhef_write_sqme_prc`, `?lhef_write_sqme_ref`, `?lhef_write_sqme_alt`)
- `$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_sqme_alt`)

- `$lhef_version` (default: "2.0")
 Specifier for the Les Houches Accord (LHEF) event format files with XML headers to discriminate among different versions of this format. (cf. also `$sample`, `sample_format`, `lhef`, `$lhef_extension`, `$lhef_extension`, `?lhef_write_sqme_prc`, `?lhef_write_sqme_ref`, `?lhef_write_sqme_alt`)
- `$lhef_write_sqme_alt` (default: true)
 Flag that decides whether in the (\rightarrow) `lhef` event format alternative weights of the squared matrix element shall be written in the LHE file. (cf. also `$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`
 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`, `$library_name`)
- `$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`)
- `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`)

- `?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`)
- `long`
Specifier for the `sample_format` command to demand the generation of the long variant of HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)
- `luminosity` (default: 0)
This specifier `luminosity = <num>` sets the integrated luminosity (in inverse femtobarns, fb^{-1}) for the event generation of the processes in the SINDARIN input files. Note that WHIZARD itself chooses the number from the `luminosity` or from the `n_events` specifier, whichever would give the larger number of events. As this depends on the cross section under consideration, it might be different for different processes in the process list. (cf. `n_events`, `$sample`, `sample_format`, `?unweighted`)
- `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`)
- `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`)
- `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`)

- **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_Eclusfactor** (default: 0.2)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_Emin** (default: 0.)
Real parameter that sets a minimal energy E_{min} value as an infrared cutoff in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_etaclusfactor** (default: 1.)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_etamax** (default: 0.)
This real parameter sets a maximal pseudorapidity that enters the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_ETclusfactor** (default: 0.2)
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 GeV)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **?mlm_matching** (default: false)
Master flag to switch on MLM (LO) jet matching between hard matrix elements and the QCD parton shower. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_nmaxMEjets** (default: 0/off)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_Qcut_ME** (default: 0 GeV)
Real parameter that in the MLM jet matching between hard matrix elements and QCD parton shower sets a possible virtuality cut on jets from the hard matrix element. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_Qcut_PS** (default: 0 GeV)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_ptmin** (default: 0 GeV)
This real parameter sets a minimal p_T that enters the y_{cut} jet clustering measure in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_Rclusfactor** (default: 1.)
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_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **mlm_Rmin** (default: 0.)
Real parameter that sets a minimal R distance value that enters the y_{cut} jet clustering measure in the MLM jet matching between hard matrix elements and QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `mlm_ ...`, `?hadronization_active`)
- **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`, `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 `$model_name`)
- `$model_name` (default: `SM`)
 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`)
- `modulo`
 Numerical function for integer and real numbers `modulo (x, y)` that computes the value of x modulo y . (cf. also `abs`, `sgn`, `mod`)
- `mokka`
 Specifier for the `sample_format` command to demand the generation of the MOKKA variant for HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)
- `mrاد`
 Expression specifying the physical unit of milliradians for angular variables. This default in WHIZARD is `rad`. (cf. `degree`, `rad`).
- `?multi_active` (default: `false`)
 Master flag that switches on WHIZARD's module for multiple interaction with interleaved QCD parton showers for hadron colliders. Note that this feature is still experimental. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`)
- `nbarn`
 Physical unit, stating that a number is in nanobarns (10^{-9} barn). (cf. also `abarn`, `fbarn`, `pbarn`)
- `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_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`, `$symbol`)
- `n_calls_test` (default: 0/off)
 Integer variable that allows to set a certain number of matrix element sampling test calls without actually integrating the process under consideration. (cf. `integrate`)

- `?negative_weights` (default: `false`)
Flag that tells WHIZARD to allow negative weights in integration and simulation. (cf. also `simulate`, `?unweighted`)
- `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`, `$sample`, `sample_format`, `?unweighted`)
- `n_in`
Integer variable that accesses the number of incoming particles of a process. It can be used in cuts or in an analysis. (cf. also `sqrts_hat`, `cuts`, `record`, `n_out`, `n_tot`)
- `nint`
This is a function `nint (<num_val>)` that converts real numbers into the closest integer, e.g. `int i = nint (4.56789)` gives `i = 5`. (cf. also `int`, `floor`, `ceiling`)
- `no`
`no` is a function that works on a logical expression and a list, `no <log_expr> [<list>]`, and returns `true` if and only if `log_expr` is fulfilled for *none* of the entries in `list`, and `false` otherwise. Examples: `no Pt < 100 GeV [lepton]` checks whether no lepton is softer than 100 GeV. It is the logical opposite of the function `all`. Logical expressions with `no` can be logically combined with `and` and `or`. (cf. also `all`, `any`, `and`, and `or`)
- `none`
Beams specifier that can be used to explicitly *not* apply a structure function to a beam, e.g. in HERA physics: `beams = e1, P => none, pdf_builtin`. (cf. also `beams`)
- `?normalize_bins` (default: `false`)
Settings for WHIZARD's internal graphics output: flag that determines whether the weights shall be normalized to the bin width or not. (cf. also `n_bins`, `$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`, `$gmlcode_fg`, `?draw_histogram`, `?draw_base`, `?draw_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)
- `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`)
- `$obs_label` (default: `"/off`)
Settings for WHIZARD's internal graphics output: this is a string variable `$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`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?draw_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)
- `$obs_unit` (default: `"/off`)
Settings for WHIZARD's internal graphics output: this is a string variable `$obs_unit = "<LaTeX_Code>"` that allows to attach a L^AT_EX physical unit to a plotted or histogrammed observable. (cf. also `n_bins`, `?normalize_bins`, `$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_pieewise`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)
- `$omega_flag` (default: `""`)
String variable that allows to pass flags to the O'Mega matrix element generator. Normally, WHIZARD takes care of all flags automatically. Note that for restrictions of intermediate states, there is a special string variable: (cf. →) `$restrictions`.
- `?omega_omp` (default: set by OpenMP)
Flag to switch on or off OpenMP multi-threading for O'Mega matrix elements. (cf. also `$method`, `$omega_flag`)
- `?omp_is_active` (default: set by OpenMP)
Flag to switch on or off OpenMP multi-threading for WHIZARD. (cf. also `?omp_logging`, `omp_num_threads`, `omp_num_threads_default`, `?omega_omp`)
- `?omp_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`)

- `openmp_num_threads` (default: set by OpenMP)
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` (default: set by OpenMP)
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`)
- `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`, `$out_file`, `printf`)
- `or`
This is the standard two-place logical connective that has the value true if one of its operands is true, otherwise a value of false. It is applied to logical values, e.g. cut expressions. (cf. also `and`, `not`).
- `?out_advance` (default: `true`)
Flag that sets advancing in the `printf` output commands, i.e. continuous printing with no line feed etc. (cf. also `printf`)
- `$out_file` (default: `"/off`)
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`)
- `P`
Unary (binary) observable specifier for the spatial momentum $\sqrt{p^2}$ of a single (two) particle(s), e.g. `eval P ["W+"], all P > 200 GeV [b, B]`. (cf. `eval`, `cuts`, `selection`)
- `?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`)
- `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`, `$pdf_builtin_file`)
- `$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`)
- `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`)
- `phs_e_scale` (default: 10 GeV)
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`)
- `$phs_file` (default: ""/internal)
This string variable allows the user to set an individual file name for the phase space parameterization for a particular process: `$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_method` (default: "default","wood")
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_m_scale` (default: 10 GeV)
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_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 \rightarrow `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: 10 GeV)
 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: 50 GeV)
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: 100 GeV)
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`)
- `P1`
Unary (binary) observable specifier for the longitudinal momentum (p_z in the c.m. frame) of a single (two) particle(s), e.g. `eval P1 ["W+"]`, `all P1 > 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 <p1> [, <p2> , ...]`. (cf. also `unpolarized`, `simulate`, `?polarized_events`)
- `?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`)
- `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: `<p1>:<p2>:...`, where all masses have to be identical, and inclusive sums, `<p1> + <p2> + ...`. 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`).
- **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`)
- **ps_fixed_alpha_s** (default: 0.)
This real parameter sets the value of α_s if it is (cf. $\rightarrow ?ps_isr_alpha_s_running$, $?ps_fsr_alpha_s_running$) 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_alpha_s_running** (default: true)
Flag that decides whether a running α_s is taken in time-like QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- **ps_fsr_lambda** (default: 0.29 GeV)
By this real parameter, the value of Λ_{QCD} used in running α_s for time-like showers is set (except for showers in the decay of a resonance). (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- **?ps_isr_active** (default: false)
Flag that switches initial-state QCD radiation (ISR) on. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- **?ps_isr_alpha_s_running** (default: true)
Flag that decides whether a running α_s is taken in space-like QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$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_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- `ps_isr_lambda` (default: 0.29 GeV)
By this real parameter, the value of Λ_{QCD} used in running α_s for space-like showers is set. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- `ps_isr_minenergy` (default: 2 GeV)
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 GeV)
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 GeV/off)
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.)
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_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- `ps_isr_z_cutoff` (default: 0.999)
This real parameter allows to set the upper cutoff on the splitting variable z in space-like QCD parton showers. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)

- `ps_mass_cutoff` (default: 1 GeV)
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 maximum number of flavors that can be produced in a QCD shower $g \rightarrow q\bar{q}$. 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_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_PYTHIA_verbose` (default: false)
Flag to switch on verbose messages when using the PYTHIA shower and/or hadronization. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- `?ps_use_PYTHIA_shower` (default: false)
Flag whether to use WHIZARD's internal shower(s) or the parton shower from the included PYTHIA. (cf. also `?allow_shower`, `?ps_ ...`, `$ps_ ...`, `?mlm_ ...`, `?hadronization_active`)
- `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`, `mrاد`).
- **Rap**
Unary and also binary observable specifier, that as a unary observable gives the rapidity of a particle momentum. The rapidity is given by $y = \frac{1}{2} \log [(E + p_z)/(E - p_z)]$. 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_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`)
- **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.
- **?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**)
- **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**)
- **?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./off)
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`)
- **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`).
- **?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**
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 reconstruct only kinematics but recalculate the events (\rightarrow `?update_event`), to recalculate the matrix element (\rightarrow `?update_sqme`) or to recalculate the corresponding 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`)
- **\$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.
- **\$restrictions**
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`)

- **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`)

- **\$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: ""/no run ID)

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.

- **safety_factor** (default: 1.)

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: ""/internal)

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_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_vrb`, `ascii`, `athena`, `debug`, `long`, `short`, `hepmc`, `lhef`, `lha`, `lha_vrb`, `stdhep`, `stdhep_up`. (cf. also `$sample`, `simulate`, `hepevt`, `ascii`, `athena`, `debug`, `long`, `short`, `hepmc`, `lhef`, `lha`, `stdhep`, `stdhep_up`, `$sample_normalization`, `?sample_pacify`, `sample_max_tries`, `sample_split_n_evt`)

- `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`)
- `$sample_normalization` (default: "auto")
String variable that allows to set the normalization of generated events. There are four options: option "1" (events normalized to one), "1/n" (sum of all events in a sample normalized to one), "sigma" (events normalized to the cross section of the process), and "sigma/n" (sum of all events normalized to the cross section). The default is "auto" where unweighted events are normalized to one, and weighted ones to the cross section. (cf. also `simulate`, `$sample`, `sample_format`, `?sample_pacify`, `sample_max_tries`, `sample_split_n_evt`)
- `?sample_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_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`)
- `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 `/+`, `/+/,` `/-`, `/*,` `/*/,` `//`)

- **seed**

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**

Subevent function `select if <condition> [<list1> [, <list2>]]` that select 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`, `+`)

- **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 `(→)` `cuts` command. E.g. `selection = any PDG == 13` is an electron selection in a lepton sample.

- **?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 `(→) $sf_trace_file`.

- **\$sf_trace_file** (default: `"/off`)

`$sf_trace_file = "<file_name>"` allows to change the detailed structure function information switched on by the debug flag `(→) ?sf_trace` into a different file `<file_name>` than the default `<proc_name>_sftrace.dat`.

- **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**, **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**)
- **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**)
- **?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 in-

formation from the common blocks of the SUSY Les Houches Accord files. (cf. also `read_slha`, `write_slha`, `?slha_read_decays`, `?slha_read_input`)

- **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**

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.

- **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. Note that this is only available if the StdHEP package is installed and correctly linked. (cf. also `$sample`, `sample_format`)

- **stdhep_up**

Specifier for the `sample_format` command to demand the generation of binary StdHEP event files based on the HEPRUP/HEPEUP common blocks. Note that this is only

available if the StdHEP package is installed and correctly linked. (cf. also `$sample`, `sample_format`)

- `?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_pieewise`, `?fill_curve`, `?draw_histogram`, `?draw_curve`, `?draw_errors`, `$fill_options`, `$draw_options`, `$err_options`, `?draw_symbols`)
- `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 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_RF`
Binary observable specifier, that gives the angle enclosed between the momenta of the two particles in the rest frame of the collider: `eval Theta_RF [jet, jet]`. (cf. also `eval`, `cuts`, `selection`, `Theta`)
- `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

```

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.

below this threshold, the weight is not lowered further but kept at this threshold. (cf. also `channel_weights_power`)

- **\$title** (default: `"/off`)
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`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?fill_curve`, `?draw_pieewise`, `?draw_curve`, `?draw_errors`, `$symbol`, `?draw_symbols`, `$fill_options`, `$draw_options`, `$err_options`)
- **tolerance** (default: 0.)
Real variable that defines the 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`)
- **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 Z s 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`)

- `?unweighted` (default: `true`)
Flag that distinguishes between unweighted and weighted event generation. (cf. also `simulate`, `n_events`, `luminosity`)
- `?update_event` (default: `false`)
Flag that decides whehter the events in an event file should be updated when reading event/rescanning files into WHIZARD. (cf. `rescan`, `?recover_beams`, `?update_sqme`, `?update_weight`)
- `?update_sqme` (default: `false`)
Flag that decides whehter 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 whehter 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`)
- `user_strfun`
Beam specifier, `beams = <b1>, <b2> => user_strfun ("<user_sf>")` for a user-defined structure function as an external code. For this command to work, there needs to be a file `<user_sf>.f90` being present in the working directory. Note that this command is not yet reactivated again.
- `?use_vamp_equivalences` (default: `true`)
Flag that decides whether equivalence relations (symmetries) between different integration channels are used by the VAMP integrator.
- `?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_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`)
- `?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_history` (default: `true`)
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`)
- `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 `$out_file` is provided, the histogram tables/plot data etc. are written to the default file `whizard.analysis.dat`. Note that the related command `compile_analysis` does the same as `write_analysis` but in addition invokes the WHIZARD \LaTeX routines for producing postscript or PDF output of the data. (cf. also `$out_file`, `compile_analysis`)

- `?write_raw` (default: `true`)
Flag to write out events in WHIZARD's internal binary format. (cf. `simulate`, `?read_raw`, `sample_format`, `$sample`)
- `write_slha`
Demands WHIZARD to write out a file in the SUSY Les Houches accord (SLHA) format. (cf. also `read_slha`, `?slha_read_decays`, `?slha_read_input`, `?slha_read_spectrum`)
- `$x_label` (default: `""/off`)
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`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?fill_curve`, `?draw_pieewise`, `?draw_curve`, `?draw_errors`, `$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_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `?draw_options`, `$err_options`, `$symbol`)
- `x_max` (default: `internal`)
Settings for WHIZARD's internal graphics output: real parameter that sets the upper limit of the x axis plotting or histogram interval. (cf. also `?normalize_bins`, `$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`, `$gmlcode_fg`, `?draw_base`, `?draw_histogram`, `?draw_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `?draw_options`, `$err_options`, `$symbol`)
- `x_min` (default: `internal`)
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_pieewise`, `?fill_curve`, `?draw_curve`, `?draw_errors`, `?draw_symbols`, `$fill_options`, `?draw_options`, `$err_options`, `$symbol`)
- `$y_label` (default: `""/off`)
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_pieewise, ?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_pieewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, \$fill_options, \$draw_options, \$err_options, \$symbol)
- **y_max** (default: internal)
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_pieewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, \$fill_options, \$draw_options, \$err_options, \$symbol)
- **y_min** (default: internal)
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_pieewise, ?fill_curve, ?draw_curve, ?draw_errors, ?draw_symbols, \$fill_options, \$draw_options, \$err_options, \$symbol)

Acknowledgements

We would like to thank E. Boos, R. Chierici, K. Desch, M. Kobel, F. Krauss, P.M. Manakos, N. Meyer, K. Mönig, H. Reuter, T. Robens, S. Rosati, J. Schumacher, M. Schumacher, and C. Schwinn who contributed to **WHIZARD** by their suggestions, bits of codes and valuable remarks and/or used several versions of the program for real-life applications and thus helped a lot in debugging and improving the code. Special thanks go to A. Vaught and J. Weill for their continuous efforts on improving the g95 and gfortran compilers, respectively.

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] 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.
- [18] 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].
- [19] L. Linssen, A. Miyamoto, M. Stanitzki and H. Weerts, *Physics and Detectors at CLIC: CLIC Conceptual Design Report*, arXiv:1202.5940 [physics.ins-det].
- [20] C. F. von Weizsäcker, *Radiation emitted in collisions of very fast electrons*, Z. Phys. **88**, 612 (1934).
- [21] E. J. Williams, *Nature of the high-energy particles of penetrating radiation and status of ionization and radiation formulae*, Phys. Rev. **45**, 729 (1934).
- [22] 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.
- [23] 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).
- [24] V. I. Telnov, *Problems of Obtaining $\gamma\gamma$ and γe Colliding Beams at Linear Colliders*, Nucl. Instrum. Meth. A **294**, 72 (1990).
- [25] V. I. Telnov, *Principles of photon colliders*, Nucl. Instrum. Meth. A **355**, 3 (1995).
- [26] 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.

- [27] 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.
- [28] A. Djouadi, J. Kalinowski, M. Spira, *Comput. Phys. Commun.* **108** (1998) 56-74.
- [29] E. Boos *et al.*, *Generic user process interface for event generators*, arXiv:hep-ph/0109068.
- [30] 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].
- [31] J. Alwall *et al.*, *A standard format for Les Houches event files*, *Comput. Phys. Commun.* **176**, 300 (2007) [arXiv:hep-ph/0609017].
- [32] K. Hagiwara *et al.*, *Supersymmetry simulations with off-shell effects for LHC and ILC*, *Phys. Rev. D* **73**, 055005 (2006) [arXiv:hep-ph/0512260].
- [33] 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].
- [34] M.E. Peskin, D.V.Schroeder, *An Introduction to Quantum Field Theory*, Addison-Wesley Publishing Co., 1995.
- [35] 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].
- [36] 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 Lhaglu*, 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.
- [37] M. Dobbs and J. B. Hansen, *The HepMC C++ Monte Carlo event record for High Energy Physics*, *Comput. Phys. Commun.* **134**, 41 (2001).
- [38] E. Boos *et al.* [CompHEP Collaboration], *Nucl. Instrum. Meth. A* **534**, 250 (2004) [hep-ph/0403113].
- [39] 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].
- [40] 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].
- [41] 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]].

- [42] 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]].
- [43] G. P. Salam and J. Rojo, *A Higher Order Perturbative Parton Evolution Toolkit (HOP-PET)*, Comput. Phys. Commun. **180**, 120 (2009) [arXiv:0804.3755 [hep-ph]].
- [44] W. Kilian, J. Reuter, S. Schmidt and D. Wiesler, *An Analytic Initial-State Parton Shower*, JHEP **1204** (2012) 013 [arXiv:1112.1039 [hep-ph]].