arXiv:0708.4233 (also based on LC-TOOL-2001-039 (revised))

WHIZARD 2.0

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

MANUAL [1]

WOLFGANG KILIAN,[2] THORSTEN OHL,[3] JÜRGEN REUTER[4]

Universität Siegen, Emmy-Noether-Campus, Walter-Flex-Str. 3, D–57068 Siegen, Germany
Universität Würzburg, Am Hubland, D–97074 Würzburg, Germany
Universität Freiburg, Hermann-Herder-Str. 3, D–79104 Freiburg, Germany

---

[1] This 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

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

[3] e-mail: `ohl@physik.uni-wuerzburg.de`

[4] e-mail: `reuter@physik.uni-freiburg.de`

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 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, an interface to the LHAPDF library is provided.

For showering, fragmenting and hadronizing the final state, a `PYTHIA` and `HERWIG` interface are provided which follow the Les Houches Accord.

The `WHIZARD` distribution is available at

`http://whizard.event-generator.org`

or at

`http://projects.hepforge.org/whizard`

where also the `svn` repository is located.

Contents

Chapter 1

Introduction

1.1 Disclaimer

This is a very 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://projects.hepforge.org/whizard` *directly.*

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.*

1.2 Overview

1.3 About examples in this manual

Although `WHIZARD` has been designed as a Monte Carlo event generator for LHC physics, several elementary steps and aspects of its usage throughout the manual will be demonstrated with the famous textbook example of $e^+e^- \to \mu^+\mu^-$. This is the same process, the textbook by Peskin/Schroeder [14] 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 Tevatron or LHC.

Chapter 2

Installation

2.1 Prerequisites and Installation

The concept of the WHIZARD installation has been changed from version 1 to version 2. Now WHIZARD is centrally installed on a computer, e.g. in the `/usr/local`, and then the user has a working space which is completely separated from the WHIZARD installation directory. The WHIZARD tarball can be downloaded either from the WHIZARD webpage, `http://whizard.event-generator.org`, or the corresponding HepForge webpage, `http://projects.hepforge.org/whizard`. On the WHIZARD webpage, one can either download the tarball of the most recent version (or older versions), or one can check out the latest version from the subversion (svn) repository. The latter is only recommended for developers and users willing to accept that maybe not all newly installed features are already working. The check-out from the svn repository is done with the following command:

```
svn checkout http://svn.hepforge.org/whizard/trunk/ SomeLocalDir
```

Note again, that the subversion contains the latest developer version. In order to be able, to compile this, one has to first generate the `configure` script out of the file `configure.ac` by running `autoreconf` (NOT `autoconf`) which is part of the `autoconf/automake` (`http://www.gnu.org/software/autoconf/` and `http://www.gnu.org/software/automake`) package. Furthermore, the development version also needs the `noweb` tools to be installed on the system in order to extract the source codes and documentation from several so called `.nw` files. The `noweb` package can be downloaded and installed from `http://www.cs.tufts.edu/~nr/noweb/`.

The general prerequisites for the installation (i.e. also from the tarball, not only from the svn) are standard tools for software development like `make` etc., and two different compilers, a FORTRAN2003 for the WHIZARD core and its corresponding libraries as well as an O'Caml compiler for the O'Mega matrix element generator.

Unpack the tarball, go to the WHIZARD directory, create a new directory and go to it. In that directory, perform a `../configureFC=<yourFORTRANcompiler>--prefix=/usr/local`. Note that this is because the source and compile directories should be different to avoid any problems during compilation and installation. `../configure--help` shows you the options for the configure process you have. The `FC` environment variable allows you to specify your

`FORTRAN` compiler of choice. Note that `WHIZARD` 2 has been written in `FORTRAN2003` in a fully object-oriented way. We highly recommend usage of the standard `gfortran` compiler from version 4.5.0 on. You can access the help menu of configure by `../configure --help`. `./configure -V` shows you the actual version of your downloaded `WHIZARD` distribution. The possible environment variables are:

`CC`	`C compiler command`
`CFLAGS`	`C compiler flags`
`LDFLAGS`	`linker flags, e.g. -L<lib dir> if you have libraries in a nonstandard directory <lib dir>`
`LIBS`	`libraries to pass to the linker, e.g. -l<library>`
`CPPFLAGS`	`C/C++/Objective C preprocessor flags, e.g. -I<include dir> if you have headers in a nonstandard directory <include dir>`
`CPP`	`C preprocessor`
`FC`	`Fortran compiler command`
`FCFLAGS`	`Fortran compiler flags`
`CXX`	`C++ compiler command`
`CXXFLAGS`	`C++ compiler flags`
`CXXCPP`	`C++ preprocessor`

For most of these there is no need to be set during installation.

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 `O'Caml` 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 parts of the program do contain good old `Fortran77` code, e.g. the PYTHIA bundle for showering and hadronization. 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`. 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 `O'Caml` 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. 2.1.2) requiring O'Mega matrix elements are thereby switched off. Note that you can install such a built (e.g. on a batch system without O'Caml installation), but the try to build a distribution (all `make distxxx` targets) will fail.

2.1.1 Installation of optional external programs

There are several optional external programs that can be installed and linked to `WHIZARD`. The probably most important is LHAPDF [16] for using parton distribution functions (PDFs) for hadron colliders. Other external programs or libraries are HepMC [17] and StdHEP [**?**] for the corresponding event formats. As LHAPDF has become a widely accepted tool and is easily available from the Hepforge server, we decided not to ship `WHIZARD` with a standard PDF. Note that because we believe this code is outdated by now, from `WHIZARD` 2.0 on we do no longer support the PDFLIB interface from the CERNLIB library.

As the LHAPDF homepage states, it provides a unified and easy to use interface to modern PDF sets. The LHAPDF package is available from the Hepforge address: `http://projects.hepforge.org/lhapdf/`. A comprehensive manual and description on how to install it. The basic procedure is the same as for `WHIZARD` itself, namely unpack it, configure it with a flag `FC=<your compiler>` for the Fortran 95 compiler and a flag `--prefix=<install dir.>` for the install directory, and then do a `make`. After that you can do an optional `make check`. Finally, install LHAPDF by doing `make install`. It is not mandatory to compile LHAPDF with the same Fortran compiler as `WHIZARD`, but of course desirable. In the worst case, when configuring `WHIZARD` you have to specify the run-time library for the Fortran compiler for LHAPDF by `LIBS=-L<Fortran run time>`. If this library is in a system-accessible library path, this is not necessary. When configuring `WHIZARD`, `WHIZARD` looks for the binary `lhapdf-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 lhapdf-config... /usr/local/bin/lhapdf-config
checking the LHAPDF version... 5.8.2
checking the LHAPDF pdfsets path... /usr/local/share/lhapdf/PDFsets
checking for initpdfsetm 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 `lhapf-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 lhapdf-config... /usr/local/bin/lhapdf-config
checking the LHAPDF version... 5.8.2
checking the LHAPDF pdfsets path... /usr/local/share/lhapdf/PDFsets
checking for initpdfsetm 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`.

The HepMC package [17] is an object oriented event record written in C++ for High Energy Physics Monte Carlo Generators. Many extensions from HEPEVT, the Fortran HEP standard, are supported: the number of entries is unlimited, spin density matrices can be stored with each vertex, flow patterns (such as color) can be stored and traced, integers representing random number generator states can be stored, and an arbitrary number of event weights can be included. The HepMC webpage is: `https://savannah.cern.ch/projects/hepmc/`, and the package can be downloaded from `http://lcgapp.cern.ch/project/simu/HepMC/download/`. Detailed information on the installation and usage can be found there. 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 whether we are using the GNU C++ compiler... (cached) yes
checking whether g++ accepts -g... (cached) yes
checking dependency style of g++... (cached) 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... 2.05.01
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 `WHIZARD` does not automatically find the HepMC distribution (because it is installed in a non-standard path), or you want to use a different version than installed on your system, then set the environment variable `HEPMC_DIR` during the `WHIZARD` configuration to the corresponding HepMC installation directory:

```
./configure HEPMC_DIR=<HEPMC install dir.> CXXFLAGS=<C++ falgs>
```

The environment variable `CXXFLAGS` allows you to set specific `C/C++` preprocessor flags, e.g. non-standard include paths for header files.

Besides the fact that StdHEP contains a set of translation routines which convert Herwig, Jetset, Isajet, or QQ events to and from the standard HEP event format, it also contains utility routines to work with the HEPEVT common block and a set of I/O routines. The second point is the interesting one for the usage with `WHIZARD`, as StdHEP provides a possibility to write machine-independent binary event files, using either the HEPEVT or the HEPRUP/HEPEUP common block. The StdHEP webpage is `http://cepa.fnal.gov/psm/stdhep/` and the package can be downloaded from `http://cepa.fnal.gov/psm/stdhep/getStdHep.shtml`. 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 Fortran90/95/03 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=<fmcfio 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.

2.1.2 `WHIZARD` self tests/checks

`WHIZARD` has a number of self-consistency checks and test 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 `test` 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.

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`.

2.2 Setting up a user work space

When `WHIZARD` is installed on a system it can be used by any user in a multi-user environment.

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`[1]. Now since SINDARIN is a programming language, we honor the old tradition of starting with the famous Hello World program. In SINDARIN this reads simply

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

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

Now we assume that you – or your kind system administrator – has installed `WHIZARD` in your executable path. Then you should open a command shell and execute

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

and if everything works well, you get the output

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

[... here a banner is displayed]

```
|=============================================================================|
|                                 WHIZARD 2.0.1
|=============================================================================|
| Initializing process library 'processes'
| Reading model file 'SM.mdl'
| Using model: SM
```

---

[1] As it is well known, W(h)izards communicate in SINDARIN, Scripting INtegration, Data Analysis, Results display and INterfaces.

```
| 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 initializes a *process library*, and it reads a physics *model*. 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'
|=============================================================================|
|                                WHIZARD 2.0.0_rc1
|=============================================================================|
| Initializing process library 'processes'
| Reading model file 'SM.mdl'
| Using model: SM
| Reading commands from file 'ee.sin'
| Added process to library 'processes':
|   [O] ee = e-, e+ => mu-, mu+
| Generating code for process library 'processes'
| Calling O'Mega for process 'ee'
| command:
| /home/kilian/whizard/build/nagfor/src/omega/bin/omega_SM.opt -o
| ee.f90 -target:whizard -target:parameter_module parameters_SM
| -target:module ee -target:md5sum 6ABA33BC2927925D0F073B1C1170780A
! -fusion:progress -scatter 'e- e+ => mu- mu+'
[1/1] e- e+ => mu- mu+ ... done. [time: 0.03 secs, total: 0.03 secs, remaining: 0.00 secs]
all processes done. [total time: 0.03 secs]
SUMMARY: 6 fusions, 2 propagators, 2 diagrams
| Writing interface code for process library 'processes'
| Compiling process library 'processes'
```

```
| Loading process library 'processes'
| Process 'ee': updating previous configuration
sqrts =    3.6000000000000000E+02
| Integrating process 'ee'
| Generating phase space, writing file 'ee.phs' (this may take a while)
| Found 2 phase space channels.
Warning: No cuts have been defined.
```

```
| Using partonic energy as event scale.
| iterations = 3:1000, 3:10000
| Creating VAMP integration grids:
| Using phase-space channel equivalences.
| 1000 calls, 2 channels, 2 dimensions, 20 bins, stratified = T
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
   1       1000  8.3366006E+02  1.47E+00    0.18    0.06*  40.12
   2       1000  8.3357740E+02  8.16E-01    0.10    0.03*  40.11
   3       1000  8.3214263E+02  1.01E+00    0.12    0.04   57.40
|-----------------------------------------------------------------------------|
   3       3000  8.3311382E+02  5.83E-01    0.07    0.04   57.40    0.69   3
|-----------------------------------------------------------------------------|
   4      10000  8.3325834E+02  1.10E-01    0.01    0.01*  57.02
   5      10000  8.3333796E+02  1.11E-01    0.01    0.01   57.03
   6      10000  8.3323772E+02  1.11E-01    0.01    0.01   57.03
|=============================================================================|
   6      30000  8.3327798E+02  6.41E-02    0.01    0.01   57.03    0.23   3
|=============================================================================|
```

```
n_events =  10
$sample => "ee"
| Initializating simulation for processes ee:
| Simulation mode = unweighted, event_normalization = '1'
| No analysis setup has been provided.
| Writing events in LHEF format to file 'ee.lhef'
| Generating 10 events ...
| Writing events in internal format to file 'ee.evx'
| Event sample corresponds to luminosity [fb-1] =   0.1200E-01
| ... done
| Simulation finished.
| There were no errors and  1 warning(s).
| WHIZARD run finished.
|=============================================================================|
```

The final result is the desired event file, `ee.lhef`.

Chapter 4

SINDARIN: The `WHIZARD` command language

4.1 A specialized command language

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 ist, or it combines these two approaches. `WHIZARD` 1 was built in this way: there were some input files which were written by the user, and it could be called both stand-alone or as an external library.

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

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

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

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

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

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

We do plan to support interfaces for commonly used languages in the future. However, introducing a speal-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 is interested in.

4.2 Overview: Basic concepts

4.2.1 SINDARIN scripts

SINDARIN scripts are contained in files. The user can create them using any editor of his choice. By convention, the files have the extension '`.sin`'. `WHIZARD` executes a script if the filename is given as an argument to the program:

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

Alternatively, scripts can be executed line by line interactively; we describe this below in Sec.5.2.

A SINDARIN script as a whole is a sequence of *commands*, similar to the commands in any imperative language such as Fortran or C. Examples of commands are `integrate` or `simulate`.

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

Nevertheless, we recommend to use some line-oriented format and meaningful identation, so the logical structure of a script is made explicit. How this is done in detail, is up to the script writer.

A command may consist of a *keyword*, a list of *arguments* in brackets `(...)`, and an *option* script. In some cases, there is a zeroth (*suffix*) argument without brackets, immediately following the keyword.

Arguments enclosed in square brackets `[]` also exist. They have a special meaning, they denote subevents (collections of momenta) in event analysis.

The option script, if any, is enclosed in braces {...}. It is also a sequence of commands (possibly with their own options). Usually, it has the purpose of setting specific parameters in a context local to the command. The braces indicate a scoping unit; most 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 upon the end of line. This distinction may be useful; a possible convention could be to initiate actual comments by `#` and use `!` for "commenting out" lines that should temporarily be disabled.

4.2.2 Data types and expressions

SINDARIN data are classified by their types. 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 contains arithmetic expressions and functions very much 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 names of numerical variables consist of alphanumeric characters and underscores. The first character must not be a digit. Character case does matter. In this manual we follow the convention that variable names consist of lower-case letters, digits, and underscores only.

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. Observable names begin with a capital letter.

The language also has the following standard types:

- `logical` (a.k.a. boolean). Logical variable names are prefixed by a `?` (question mark) sign, otherwise the same rules as for numerical variables apply.
- `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.

Furthermore, SINDARIN supports two data types tailored specifically for Monte Carlo:

- `alias` objects denote a set of particle species. Alias names have no prefix, they are distinguished from numerics by context.
- `subevt` objects denote a collection of particle momenta within an event. Their names are prefixed by a `@` (at) sign.

Each variable or object has a well-defined type.

In the current implementation, SINDARIN has no container data types derived from basic types, such as lists, arrays, or hashes. The `subevt` type is a container for particles, but there is no type for an individual particle: this is represented as a one-particle `subevt`.

Grouping of numerical, logical, string, and alias expressions is done using ordinary brackets `()`. For subevent expressions, use square brackets `[]`.

4.2.3 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* variables that are automatically computed by the program. On top of that, the user is free to introduce his own variables (*user variables*).

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.

Intrinsic, independent variables can also be assigned a new value, but they cannot be redeclared. Intrinsic dependent variables can never be assigned a new value explicitly. They get a new value automatically when one of the variables they depend on is assigned a new value.

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

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

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

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

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

4.2.4 Special objects

In addition to the basic data types, SINDARIN contains objects that serve special purposes. Most of them do not occur in expressions, but there are assignment statements and operations acting on them with specific rules. Examples are physics models, processes, cut expressions, iteration specifiers, histograms, and more. These objects are intrinsic.

4.2.5 Control structures

A complete programming language should have some concept of conditionals and loops. In SINDARIN, conditionals are represented by the usual `if-then-elsif-else-endif` sequence. Since parameter scans are the obvious motivation for loops in SINDARIN, the loop syntax is `scan` *variable* `=` `(`*values*`)` {...}.

4.3 Data and expressions

4.3.1 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
 nbar  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`.

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., 0., 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 "fuzzy" comparison operators

`==~  <>~`

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 these operators, in contrast to `==` vs. `<>`, are not mutually exclusive.

4.3.2 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 real values. This includes the operators `==~` and `<>~`.

4.3.3 Complex-valued objects

Complex variables and values are currently not yet used by the physics models implemented in ***WHIZARD****. They are an experimental feature.*

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`.

4.3.4 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 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 a simulated event samples on and off.

4.3.5 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 `$`. There is only one string operation, concatenation

```
$string = "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.

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 output. There is also a `sprint` function and a `print` command; they have no format string but typeset their arguments in a default format.

4.4 Particles and (sub)events

4.4.1 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 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.

4.4.2 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:

```
["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 particle combinations. That is, the four-momenta of distinct particles are combined (added conmponent-wise), and the results become subevent elements just like ordinary particles.

Sometimes, variables (actually, named constants) of type subevent are useful. Subevent variables are declared by the `subevt` keyword, and their names carry the prefix `@`. Within expressions, 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`.

4.4.3 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 usefule 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:

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

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

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 a 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:

```
@sorted_evt = sort by Pt [@evt]
```

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 particle from *new-particles* is only appended if it is not present in *particles*, so the function will not produce duplicate entries unless they had been there in the first place.

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, 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:

```
@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]
```

4.4.4 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 `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.

4.4.5 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.

4.4.6 More particle functions

count

`count [`*`particles`*`]`

`count [`*`particles-1, particles-2`*`]`

`count if` *`logical-expr`* [*particles*] `count if` *`logical-expr`* [*particles-1*, *ref-particles-2*]

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., $\sqrt{\Delta\eta^2 + \Delta\phi^2}$

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).

4.5 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 7.1.

The model definitions are contained in text files with filename extension `.mdl`, e.g., `SM.mdl`, which are located in the `share/models` subdirectory of the `WHIZARD` installation. These files are easily readable, so if you need details of a model implementation, inspect their contents.

The model file contains the complete particle and parameter definitions as well as their default values. It also contains a list of vertices. This is used only for phase-space setup; the vertices used for generating amplitudes and the corresponding Feynman rules are stored in different files within the `O'Mega` source tree.

In a SINDARIN script, a model is a special object of type `model`. There is always a *current* model. Initially, this is the SM, so on startup `WHIZARD` reads the `SM.mdl` model file and assigns its content to the current model object. (You can change the default model by the `--model` option on the command line.) Once the model has been loaded, you can define processes for the model, and you have all independent model parameters at your disposal. As noted before, these are intrinsic parameters which need not be declared when you assign them a value, for instance:

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

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

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

If you would like to generate a SUSY process instead, for instance, you can assign a different model (cf. Table 7.1) 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
print (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
print (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.

4.6 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.

4.6.1 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 must be two or larger[1]. There is no hard upper limit; the complexity of processes that `WHIZARD` can handle depends only on the practical computing limitations (CPU time and memory). Roughly speaking, one can assume that processes up to $2 \to 6$ particles are

---

[1] $2 \to 1$ processes are currently unsupported, they will be enabled in a later version.

safe, 2 → 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.

4.6.2 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`, 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 4.1, 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 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
```

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

Table 4.1: *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.*

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.

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 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.)

4.6.3 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.

If the option string contains a `model =` reassignment, the process is defined for that model, which can be different from the current model that applies to other processes. The process will retain this association during integration and event generation.

Another useful option is the setting

```
$restrictions = string
```

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

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

which is redundant here, however.

The restriction keeps the full energy dependence in the intermediate propagator, so the Breit-Wigner shape can be observed in distributions. This breaks gauge invariance, in particular if the intermediate state is off shell, so you should use the feature only if you know the implications.

All options can also be set globally, i.e., outside the `process` definition. In the case of restrictions this may not be very useful, however.

4.6.4 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` 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.

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 to the compiler are issued. Otherwise, it will detect any modification to the process configuration and regenerate the matrix element or recompile accordingly. Thus, a SINDARIN script can be executed repeatedly without rebuilding everything from scratch, and you can safely add more processes to a script in a subsequent run without having to worry about the processes that have already been treated.

This default behavior can be changed. By setting

```
?rebuild_library = true
```

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

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

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

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

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

4.6.5 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 `processes`. If you do not specify anything else, the processes you compile will be collected by a driver file `processes.f90` which is compiled together with the process code and combined as a libtool archive `processes.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 ("my_lhc_proc", "my_other_proc")
```

will display the contents of the libraries together with a code letter which indicates the status of the libraries and the processes within.

You may generate and compile a process library, and make use of it in a different project. It is not necessary to write `process` definitions for all processes you want to use, if they are already available in compiled form. If process libraries `my_lhc_proc` and `my_other_proc` exist in the working directory, you may simply say

```
load ("my_lhc_proc", "my_other_proc")
```

to make them available for the current project. Note that the last one in the argument list becomes the active library. You may switch to another library via a `library = ...` statement.

4.6.6 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 etc.

For this task, there is a variant of the `compile` command:

```
compile as "my_whizard" ("my_lhc_proc", "my_other_proc")
```

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.

4.7 Beams

Before processes can be integrated and simulated, the program has to know about the collider properties. They can be specified by the `beam` statement.

In the command script, it is irrelevant whether a `beam` statement comes before or after process specification. The `integrate` or `simulate` commands will use the `beam` statement that was issued last.

4.7.1 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 `beam` 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 `beam` 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: some beam properties have not yet been implemented in WHIZARD2.

Beam parameters can be specified either globally or as local options to the `beam` statement (in braces, located before any structure-function settings). These two forms have the same effect on the beam properties:

```
sqrts = 14 TeV   beams = p, p => lhapdf
```

```
beams = p, p { sqrts = 14 TeV } => lhapdf
```

The value of `sqrts`, as well as any other beam parameters, will be memorized by the `beam` statement, so a modification of beam parameters after the statement has no effect on a process where the beams setup is used.

The `beam` 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 particles's mass. However, it is needed for computing polarized decay, e.g.

```
beams = Z { beam_polarization = longitudinal (1) }
```

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. 4.8.

4.7.2 LHAPDF

For incoming hadron beams, the `beam` 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 process setup, namely the example already shown above:

```
beams = p, p => lhapdf
```

This selects a default LHAPDF structure-function set for both proton beams (currently, `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 `lhapdf` keyword:

```
beams = p, p => lhapdf { $lhapdf_file = "MSTW2008lo68cl.LHgrid" }
```

Similarly, a member within the set is selected by the numeric variable `lhapdf_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, lhapdf
```

Here, there is a list of two independent structure functions (each with its own option set, if applicable) which applies to the two beams.

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

```
beams = p, gamma => lhapdf
```

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 `lhapdf_photon_scheme`. (There are also pion structure functions available.) For modifying the default, you have to specify separate structure functions

```
beams = p, gamma => lhapdf { ... }, lhapdf { ... }
```

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

```
beams = p, gamma => lhapdf { ... }, none
```

4.7.3 ISR structure functions

4.7.4 Beamstrahlung

4.7.5 Effective photon approximation

4.8 Polarization

4.8.1 Initial state polarization

WHIZARD supports multiple modes of polarizing the inital state fully or partially by assigning a nontrivial density matrix in helicity space. Initial state polarization requires a beam setup and is initialized by means of the `beam_polarization` statement:

```
beam_polarization = polarization_constructor [, polarization_constructor]
```

This statement assigns the polarization(s) specified by the `polarization_constructor`s to the incoming beam(s). There are seven different constructors available:

- `none`: Unpolarized. 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 multiplicty).

- `circular` (f): $|f|$ of the particles are in the maximum / minimum helicity eigenstate, the remainder is unpolarized. The sign of f determines the sign of the helicity eigenvalue. As only the maximal / minimal entries of the density matrix are populated, `circular` is useful mainly for spin $\frac{1}{2}$ and massless bosons of spin > 0. Parameter range:

  $$f \in [-1\ ;\ 1]$$

  Density matrix:

  $$\rho = \mathrm{diag}\left(\frac{1+f}{2}\ ,\ 0\ ,\ \ldots\ ,\ 0\ ,\frac{1-f}{2}\right)$$

- `longitudinal` (f): $|f|$ of the particles have longitudinal polarization, the remainder is unpolarized. Longitudinal polarization is (obviously) only available for massive bosons of spin > 0. Parameter range:

  $$f \in [0\ ;\ 1]$$

  Density matrix:

  $$\rho = \mathrm{diag}\left(\frac{1-f}{m}\ ,\ \ldots\ ,\ \frac{1-f}{m}\ ,\ \frac{1+f\,(m-1)}{m}\ ,\ \frac{1-f}{m}\ ,\ \ldots\ ,\ \frac{1-f}{m}\right)$$

  (m: particle multiplicity)

- `transverse` (f, ϕ): Polarization along an arbitrary direction in the $x - y$ plane, with $\phi = 0$ being positive x direction and $\phi = 90°$ positive y direction. $|f|$ of the particles are polarized, the remainder is unpolarized. The sign of f determines the sign of the eigenvalue, and flipping it therefore is equivalent to shifting ϕ by 180°. Note that, although transverse polarization 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. Parameter range:

  $$f \in [-1\ ;\ 1] \quad , \quad \phi \in \mathbb{R}$$

  Density matrix:

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

  (for antiparticles, the matrix is conjugated).

- `axis` (f, θ, ϕ): Polarization along an arbitrary axis in polar coordinates (polar axis in positive z direction, polar angle θ, azimuthal angle ϕ). $|f|$ of the particles are polarized, the remainder is unpolarized. The sign of f determines the sign of the eigenvalue.

Note that, although axis polarization defines a valid density matrix for all particles with multiplicity > 1, it is meaningful only for particles with spin $\frac{1}{2}$. Parameter range:

$$f \in [-1\ ;\ 1] \quad , \quad \theta \in \mathbb{R} \quad , \quad \phi \in \mathbb{R}$$

Density matrix:

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

- `density_matrix` (a, b): Allows to define a 2×2 density matrix (or the maximum / minimum helicity entries of a $n \times n$ matrix) explicitly. Parameter range:

  $$a \in [0\ ;\ 1] \quad , \quad b \in \left\{ z \in \mathbb{C} \,\middle|\, |z| \leq \frac{1}{2} \right\}$$

  Density matrix:

  $$\rho = \begin{pmatrix} a & 0 & \cdots & \cdots & b \\ 0 & 0 & \ddots & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 0 & 0 \\ b^* & \cdots & \cdots & 0 & 1 - a \end{pmatrix}$$

  (note that the matrix is conjugated for antiparticles).

- `diagonal_density` $(h_1 : f_1\ ,\ [h_2 : f_2\ ,\ \ldots\ ,\ h_n : f_n])$: This explicitely defines an arbitrary diagonal density matrix. The entries $\rho_{h_1,h_1}\ ,\ \ldots\ ,\ \rho_{h_n,h_n}$ (where the h_i denotes the (doubled) helicity eigenvalues for bosons (fermions)) are initialized with $f_1\ ,\ \ldots\ ,\ f_n$, the remaining matrix elements are set to zero. The h_i must be mutually different; the f_i are required to be positive and are normalized if necessary. Parameter range:

  $$h_i \in \mathbb{Z} \quad , \quad f_i \in \mathbb{R}_+$$

In addition, the `beam_polarization` statement can be used to deactivate beam polarization:

```
beam_polarization = off
```

The statement can be used both globally and locally; the order of the `beam_polarization` and `beams` statements doesn't matter. Some examples are in order to eludicate its use together with the different constructors:

-
  ```
  beams = A, A {
     beam_polarization = circular(1), transverse (1, 90 degree)
  }
  beams = u, ubar
  ```

  The first beam statement 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. As `beam_polarization` is used as a local option to `beams`, its effect does not carry over to the second `beams` statement which sets up an unpolarized u / $\overline{u}$ pair.

-
  ```
  beam_polarization = longitudinal (1)
  beams = "W+"
  beams = Z
  ```

  This example sets up the decay of a longitudinal vector boson. As the statement is used globally, it affects both `beams` definitions with the first `beams` statement defining a longitudinal W^+, and the second one defining a longitudinal Z.

-
  ```
  beams = u, ubar
  integrate (uuzz) {
     beam_polarization =
        diagonal_density (-1:1),
        density_matrix (0.25, 0.2 + exp (90 degree * I))
  }
  integrate (uuzz)
  ```

  The first integrate uses a polarized initial state with the u being in a purely left-handed state and the $\overline{u}$ being assigned the explicit density matrix

  $$\rho_{\overline{u}} = \begin{pmatrix} \frac{1}{4} & \frac{1}{5}\,e^{\frac{\pi}{2}i} \\ \frac{1}{5}\,e^{-\frac{\pi}{2}i} & \frac{3}{4} \end{pmatrix}$$

  (note that the sign of the phase is flipped as the $\overline{u}$ is an antiparticle). As `beam_polarization` is used as a local argument to `integrate`, the second `integrate` is not afflicted and calculates the integral for the unpolarized process.

-
  ```
  beam_polarization = off
  beams = u, ubar
  scan int hel1 = (-1, 1) {
     scan int hel2 = (-1, 1) {
        beam_polarization = circular (hel1), circular (hel2)
        integrate (uuzz)
     }
  }
  integrate (uuzz)
  ```

  This example loops over the different quark helicity combinations and calculates the respective integrals. The `beam_polarization` statement is confined to the loop and, therefore, the last `integrate` calculates the unpolarized integral

Although beam polarization should be straightforward to use, some pitfalls exist for the unwary:

- Once `beam_polarization` is set globally, it persists and is applied every time `beams` is executed. In particular, this means that code like

  ```
  beam_polarization = axis (0.5, 45 degree, 45 degree), none
  beams = "W+", "W-"
  beams = Z
  beam_polarization = transverse (1)
  ```

  will throw an error, event though the beam setup is consistent in the end. In order to avoid this, beam polarization must be explicitly disabled between the two `beams` statements:

  ```
  beam_polarization = axis (0.5, 45 degree, 45 degree), none
  beams = "W+", "W-"
  beam_polarization = off
  beams = Z
  beam_polarization = transverse (1)
  ```

  This kind of trap can be avoided be using `beam_polarization` only locally.

- 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.

4.8.2 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 can't be `polarized` and `unstable` at the same time.

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 select event file format(s)) and can also be accessed in the analysis by means of the `Hel` observable. For example, an analysis defintion 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 extended to cover all possible helicity combinations). Note, however, that helicity information is not available in the integration step; therefore, it is not possible to use `Hel` as a cut observable.

While final state polarization is straightforward to use, there is a caveat when used in combination with flavor products. If a particle in a flavor product is defined as `polarized`, then all particles "originating" from the product will act as if they had been declared as `polarized` — their polarization will be recorded in the generated events. E.g., the example

```
process test = u:d, ubar:dbar => d:u, dbar:ubar, u, ubar

! insert compilation, cuts and integration here

polarized d, dbar
simulate (test) {?polarized_events = true}
```

will generate events including helicity information for all final state d and $\overline{d}$ quarks, but only for part of the final state u and $\overline{u}$ quarks. In this case, if you had wanted to keep the helicity information also for all u and $\overline{u}$, you would have had to explicitely include them into the `polarized` statement.

4.9 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.

4.9.1 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.

Integration uses the `VAMP` algorithm and code. 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 and 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.

Here is an example of the integration output, which illustrates these properties. The SINDARIN script describes the process $e^+e^- \to q\bar{q}q\bar{q}$ with q being any light quark, i.e., W^+W^- and ZZ production and hadronic decay together will any irreducible background. We cut on p_T and energy of jets, and on the invariant mass of jet pairs. Here is the script:

```
alias q = d:u:s:c
alias Q = D:U:S:C
process proc_4f = e1, E1 => q, Q, q, Q

ms = 0  mc = 0
sqrts = 500 GeV
cuts = all (Pt > 10 GeV and E > 10 GeV) [q:Q]
   and all M > 10 GeV [q:Q, q:Q]

integrate (proc_4f)
```

After the run is finished, the integration output looks like

```
| Loading process library 'processes'
| Process 'proc_4f': updating configuration
| Generating phase space configuration ...
| ... done.
| ... found 114 phase space channels, collected in 17 groves.
| Phase space: found 668 equivalences between channels.
| Wrote phase-space configuration file 'proc_4f.phs'.
| iterations = 10:10000, 5:20000
| Applying user-defined cuts.
| Creating VAMP integration grids:
| Using phase-space channel equivalences.
| 10000 calls, 114 channels, 8 dimensions, 8 bins, stratified = T
| Integrating process 'proc_4f':
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
   1      10000  2.8081256E+03  3.57E+02   12.72   12.72*   4.11
   2      10000  3.0181098E+03  1.92E+02    6.36    6.36*   3.50
   3      10000  2.9288866E+03  8.53E+01    2.91    2.91*   7.55
   4      10000  3.0906462E+03  6.75E+01    2.18    2.18*   9.63
   5      10000  2.9092659E+03  5.60E+01    1.93    1.93*  10.82
   6      10000  3.0194199E+03  5.35E+01    1.77    1.77*  10.27
   7      10000  2.9812271E+03  5.31E+01    1.78    1.78   10.63
   8      10000  2.9072033E+03  5.02E+01    1.73    1.73*  13.28
   9      10000  2.9534310E+03  5.05E+01    1.71    1.71*  11.72
  10      10000  2.8998730E+03  4.89E+01    1.69    1.69*  14.10
|-----------------------------------------------------------------------------|
  10     100000  2.9555210E+03  1.95E+01    0.66    2.09   14.10    0.99  10
|-----------------------------------------------------------------------------|
  11      20000  2.9816184E+03  3.45E+01    1.16    1.63*  11.84
  12      20000  2.9773291E+03  2.41E+01    0.81    1.15*  10.30
  13      20000  2.9701919E+03  1.96E+01    0.66    0.93*   9.60
  14      20000  2.9785382E+03  1.70E+01    0.57    0.81*   9.50
  15      20000  2.9821841E+03  1.52E+01    0.51    0.72*   9.05
|-----------------------------------------------------------------------------|
  15     100000  2.9781249E+03  8.79E+00    0.30    0.93    9.05    0.06   5
|-----------------------------------------------------------------------------|
|=============================================================================|
  15     100000  2.9781249E+03  8.79E+00    0.30    0.93    9.05    0.06   5
|=============================================================================|
| Process 'proc_4f':
|    time estimate for generating 10000 unweighted events: 3m:13s
|-----------------------------------------------------------------------------|
```

Each row shows the index of a single iteration, the number of matrix element calls for that iteration, and the integral and error estimate. 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 1.7 and 14 %, 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 2.1, corresponding to 0.7 % error, and a χ^2 value of 0.98, which is just right: apparently, 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.

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.

4.9.2 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 further adapt grids, but not channel weights. This can be changed by the parameters `?adapt_final_grids` (default `true`) and `?adapt_final_weights` (default `false`).

4.9.3 Phase space

Before `integrate` can start its work, it must have a phase-space configuration for the process at hand. This is laid out in an ASCII file *process-name*`.phs`. Normally, you don't have to deal with this file, since `WHIZARD` will generate one automatically if it doesn't 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.

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 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 1. 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 2, so graphs with up to 2 t/u-channel lines are considered. In particular cases, such as $e^+e^- \to n\gamma$, all graphs are multiperipheral, and for $n > 2$ `WHIZARD` would find no parameterizations in the default setup. Increase 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.

Such logarithmic mappings need a dimensionful scale as parameter. There are three such scales, all with default value 10 GeV: `phs_e_scale` (energy), `phs_m_scale` (invariant mass), and `phs_q_scale` (momentum transfer). If cuts and/or masses are such that energies, invariant masses of particle pairs, and momentum transfer values below 10 GeV are excluded or suppressed, the values can be kept. In special cases they should be changed: for instance, if you want to describe $\gamma^* \to \mu^+\mu^-$ splitting well down to the muon mass, no cuts, you may set `phs_m_scale = mmu`. The convergence of the Monte-Carlo integration result will be considerably faster.

4.9.4 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 larger 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 user is encouraged to define his 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")
```

4.9.5 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 `true`). 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.

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 (`?alpha_s_from_mz = false`), the scale Λ_{QCD}, represented by the intrinsic variable `lambda_qcd`, is used for fixing the reference value. `alpha_s_order` and `alpha_s_nf` apply analogously.

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 is is also the factorization scale for the LHAPDF structure functions.

Just like the `cuts` expression, the `scale` expression is evaluated at the point where it is read by an explicit or implicit `integrate` command.

4.9.6 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 = (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.

4.10 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.

4.10.1 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 int 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`.

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. 4.11. In Chap. 8, you will find a more thorough discussion of event generation with `WHIZARD`, which also covers in detail the available event-file formats.

4.10.2 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 $pp \to W^+W^-$, 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., $pp \to u\bar{d}\bar{u}d$. 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. 4.6.3.

Another approach is to factorize the process in production (of W bosons) and decays ($W \to q\bar{q}$). 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.

4.11 Analysis and Output

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.

In the following sections, we first summarize the available data structures, before we consider their graphical display.

4.11.1 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     =   1.5000000
error[abs]  =  0.20412415
error[rel]  =  0.13608276
n_entries   =           4
```

4.11.2 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 further cuts, or 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`.

`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])
```

4.11.3 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), n (number of entries), and excess weight. 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.

4.11.4 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 that x errors.

The plot output, again written by `write_analysis` contains the four values for each point, again in the ordering $x, y, \Delta y, \Delta x$.

4.11.5 Output

There are three basic methods for retrieving the information stored in observables, histograms, and plots.

1. By default, the `write_analysis` command prints all data to standard output. Output is redirected to a file 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). 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.

2. Custom formatting (currently available for datasets of `histogram` and `plot` type) is switched on by `?out_custom = true`. This flag, like the other parameters described here, is meaningful only to the `write_analysis` command. If no other parameters are set, output in custom format will be a variant of a blank-separated column format where only data are printed, and metadata are listed in a comment header for each dataset. For histograms, the columns are x, y, Δy. For plots, the columns are x, y, Δy, Δx.

   The fixed-column format is switched off by `?out_columns = false`. The separator character, by default a blank character, is given by the string `$out_separator`. Hence, a comma-separated-value (CSV) format is defined by

   ```
   write_analysis (tag) {
     ?out_custom = true  ?out_columns = false  $out_separator = ","
   }
   ```

The comment initiator is given by the variable `$out_comment`, by default equal to `"# "`.

In contrast to default formatting, custom-formatted data are never written to standard output. If `$out_file` is the empty string, each dataset with tag ***`analysis-tag`*** will be written to its own file named ***`analysis-tag`***`.dat`.

3. If these switches and parameters are insufficient, it is possible to define an arbitrary output format for each histogram bin or plot point. For histograms, there is a *macro* called `histogram_writer`:

```
histogram_writer = { commands }
```

Similar to variables, the macro can be defined either locally as an option to `write_analysis` or globally.

If `?out_custom` is true and this macro is defined, the macro will be executed once for each bin. (The parameters `?out_columns` and `$out_separator` are ignored.) Within the ***`commands`*** body, the following variables are locally available:

`bin_center`	bin center
`bin_width`	bin width
`bin_sum`	summed weights in the bin
`bin_error`	standard deviation of the bin sum
`bin_excess`	excess weight
`bin_index`	bin index
`n_bins`	number of bins
`$out_file`	current output file

Note that these variables locally override all external variables of the same name.

For plots, the corresponding macro is defined in a similar fashion with

```
plot_writer = { commands }
```

and the predefined variables are

`point_x`	x
`point_y`	y
`point_yerr`	σ_y
`point_xerr`	σ_x
`point_index`	plot point index
`n_points`	total number of plot points
`$out_file`	current output file

If one of the macros is chosen for data output, metadata are still written. This can be disabled by setting `?write_header = false`.

In order make it a bit clearer how custom formats work, here's a small example which outputs all defined histograms and plots in custom format, interleaving the actual bins and plot points with comments containing the index.

```
write_analysis_data () {
   ?out_custom = true
   histogram_writer = {
      printf "# bin no. %i" (bin_index)
      printf "%10.8e   %10.8e   %10.8e"
         (bin_center, bin_sum, bin_excess)
   }
   plot_writer = {
      printf "# plot point %i" (plot_index)
      printf "%10.8e   %10.8e" (plot_x, plot_y)
   }
}
```

4.12 Data Visualization

4.12.1 GAMELAN

The data values and tables that we have introduced in the previous section can be visualized using built-in features of `WHIZARD`. To be precise, `WHIZARD` can write LaTeX code which incorporates code in the graphics language GAMELAN to produce a pretty-printed account of observables, histograms, and plots. GAMELAN is actually a macro package for MetaPost. This is a graphics language originally written by John Hobby as a variant of Knuth's MetaFont language for designing text fonts for TeX.

The GAMELAN language exists independent of `WHIZARD`. It requires a working LaTeX installation together with MetaPost (which is usually bundled with a TeX distribution, but might need a separate switch for installation). On top of that, nothing needs to be installed since the `src/gamelan` subdirectory of the `WHIZARD` build contains the complete package, including an (incomplete) manual. `WHIZARD` uses a subset of GAMELAN's graphics macros directly, but it allows for access to the full package if desired.

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.

4.12.2 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.

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

Table 4.2: *Graph options. The content of strings of type LaTeX must be valid LaTeX code (containing typesetting commands such as math mode). The content of strings of type GAMELAN must be valid GAMELAN code. If a graph bound is kept* undefined, *the actual graph bound is determined such as not to crop the graph contents in the selected direction.*

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

drawing options. Graph options apply to the graph as a whole (title, labels, etc.) and are placed in braces on the lhs of the assigment. Drawing options apply to the individual graph elements representing the contained histograms and plots, and are placed together with the graph element on the rhs of the assignment. Thus, the complete syntax for assigning multiple graph elements is

```
graph graph-tag { graph-options }
= graph-element-tag1 { drawing-options1 }
& graph-element-tag2 { drawing-options2 }
...
```

This form is recommended, but graph and drawing options can also be set as global parameters, as usual.

We list the supported graph and drawing options in Tables 4.2 and 4.3, respectively.

4.12.3 Drawing options

The options for coloring lines, filling curves, or choosing line styles make use of macros in the GAMELAN language. At this place, we do not intend to give a full account of the possiblities, but we rather list a few basic features that are likely to be useful for drawing graphs.

Table 4.3: *Drawing options. The content of strings of type GAMELAN must be valid GAMELAN code. The behavior w.r.t. the flags with* undefined *default value depends on the type of graph element. Histograms: draw baseline, piecewise, fill area, draw curve, no errors, no symbols; Plots: no baseline, no fill, draw curve, no errors, no symbols.*

Variable	Default	Type	Meaning
`?draw_base`	*undefined*	Logical	Whether to draw a baseline for the curve
`?draw_piecewise`	*undefined*	Logical	Whether to draw bins separately (histogram)
`?fill_curve`	*undefined*	Logical	Whether to fill area between baseline and curve
`$fill_options`	`""`	GAMELAN	Options for filling the area
`?draw_curve`	*undefined*	Logical	Whether to draw the curve as a line
`$draw_options`	`""`	GAMELAN	Options for drawing the line
`?draw_errors`	*undefined*	Logical	Whether to draw error bars for data points
`$err_options`	`""`	GAMELAN	Options for drawing the error bars
`?draw_symbols`	*undefined*	Logical	Whether to draw symbols at data points
`$symbol`	Black dot	GAMELAN	Symbol to be drawn
`gmlcode_bg`	`""`	GAMELAN	Code to be executed before drawing
`gmlcode_fg`	`""`	GAMELAN	Code to be executed after drawing

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.

4.13 Control structures

4.13.1 Conditionals

4.13.2 Loops

4.13.3 Include files

4.13.4 External programs

4.14 Miscellaneous commands

4.14.1 Printing messages

Chapter 5

User Interfaces for WHIZARD

5.1 Command line and SINDARIN input files

5.2 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
```

The WHISH can be closed by the `quit` command:

```
whish? quit
```

5.3 Graphical user interface

This is planned, but not implemented yet.

5.4 WHIZARD as a library

This is planned, but not implemented yet.

Chapter 6

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.

Chapter 7

Implemented physics

7.1 The Monte-Carlo integration routine: `VAMP`

7.2 The Phase-Space Setup

7.3 The hard interaction models

7.3.1 The Standard Model and friends

7.3.2 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 anomalous top couplings	`---`	`SM_top`
SM with K matrix	`---`	`SM_KM`
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`
SM with Z'	`---`	`Zprime`
"SQED" with gravitino	`---`	`GravTest`
Augmentable SM template	`---`	`Template`

Table 7.1: *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 8

Event generation and analysis

In order to perform a physics analysis with `WHIZARD` one has to generate events. This seems to be a trivial statement, but as there have been any questions like "My `WHIZARD` does not produce plots – what has gone wrong?" we believe that repeating that rule is worthwile. Of course, it is not mandatory to use `WHIZARD`'s own analysis set-up, the user can always choose to just generate events and use his/her own analysis package like ROOT, or TopDrawer, or you name it for the analysis.

Accordingly, we first start to describe how to generate events and what options there are – different event formats, renaming output files, using weighted or unweighted events with different normalizations. How to re-use and manipulate already generated event samples, how to limit the number of events per file, etc. etc.

8.1 Event generation

To explain how event generation works, we again take our favourite example, $e^+e^- \to \mu^+\mu^-$,

```
process eemm = e1, E1 => e2, E2
compile
```

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: Process 'eemm' must be integrated before simulation.`. This is because you have to provide `WHIZARD` with the information of the corresponding cross section, phase space parameterization and grids, i.e. you have to integrate a process before you could generate events. 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:

```
| Loading process library 'processes'
| Process 'eemm': updating configuration
sqrts =    500.00000000000000
| Integrating process 'eemm'
| Generating phase space configuration ...
| ... found 2 phase space channels, collected in 2 groves.
| Phase space: found 2 equivalences between channels.
| Wrote phase-space configuration file 'eemm.phs'.
Warning: No cuts have been defined.
| Using partonic energy as event scale.
| iterations = 3:10000
| Creating VAMP integration grids:
| Using phase-space channel equivalences.
| 10000 calls, 2 channels, 2 dimensions, 20 bins, stratified = T
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
   1      10000  4.2823916E+02  6.75E-02    0.02    0.02*  40.29
   2      10000  4.2823862E+02  4.37E-02    0.01    0.01*  40.29
   3      10000  4.2824459E+02  3.38E-02    0.01    0.01*  40.29
|=============================================================================|
   3      30000  4.2824192E+02  2.48E-02    0.01    0.01   40.29    0.01   3
|=============================================================================|
| Process 'eemm':
|    time estimate for 10000 unweighted events = 0h 00m 00.469s
|-----------------------------------------------------------------------------|
| Initializating simulation for processes eemm:
| Simulation mode = unweighted, event_normalization = '1'
| No analysis setup has been provided.
| Simulation finished.
| 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 4282 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:

```
| No analysis setup has been provided.
| Generating 500 events ...
| Writing events in internal format to file 'whizard.evx'
| Event sample corresponds to luminosity [fb-1] =    1.167
```

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. Files with the suffix `.evx` are binary format event files, using a machine-independent `WHIZARD`-specific event file format. Before we list the event formats supported by `WHIZARD`, the next two section 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, `WHIZARD` is trying to re-use as much information as possible. The same holds 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 with an already existing event file less or equally many events as generated before, `WHIZARD` will not generate again but re-use the existing events (as will be explained below, 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 8.1.3 then `WHIZARD` has to generate events all the time). 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`.

8.1.1 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 $2 \to 2$ process the efficiency is roughly of the order of several tens of per cent. As a rule of thumb, one can say that with every additional pair of final state particle the average efficiency one can achieve decreases by a factor of five to ten.

The default of `WHIZARD` is to generate *unweighted* events. One can use the logical variable `?unweighted = false` to disable unweighting and generate weighted events. (The command `?unweighted = true` is a tautology, because `true` is the default for this variable.) Note that again this command has to appear *before* the corresponding `simulate` command, otherwise it will be ignored or effective only for any `simulate` command appearing later in the SINDARIN file.

Excess events to be done...

8.1.2 Choice on event normalizations

There are basically four different choices to normalize event weights (... denotes the average) :

1. $\langle w_i \rangle = 1, \qquad \langle \sum_i w_i \rangle = N$
2. $\langle w_i \rangle = \sigma, \qquad \langle \sum_i w_i \rangle = N \times \sigma$
3. $\langle w_i \rangle = 1/N, \qquad \langle \sum_i w_i \rangle = 1$
4. $\langle w_i \rangle = \sigma/N, \qquad \langle \sum_i w_i \rangle = \sigma$

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 weight 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 `$event_normalization`. The default is `$event_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

Format	Type	remark	ext.
Athena	ASCII	variant of HEPEVT	no
debug	ASCII	most verbose `WHIZARD` format	no
default	ASCII	`WHIZARD` verbose 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 Madgraph style	no
LHEF	ASCII	Les Houches accord compliant	no
long	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

Table 8.1: *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.*

the normalization. The unit normalization (option 1) can be switched on also for weighted events by setting the event normalization variable equal to `"1"` or `"unity"`. Option 2 can be demanded by setting `event_normalization = "sigma"`. Options 3 and 4 can be set by `"1/n"` and `"sigma/n"`, respectively. `WHIZARD` accepts small and capital letter for these expressions.

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

8.1.3 Supported event formats

Event formats can either be distinguished whether they 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 8.1 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 sections (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.

Other event formats can be written out by setting the variable `sample_format = <format>`, where `<format>` can be any of the following supported variables:

- `ascii`: a quite verbose ASCII format which contains lots of information (an example is shown in the appendix).
  Standard suffix: `.evt`
- `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`
- `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`
- `lha`: Implementation of the Les Houches Accord as it was in the old MadEvent and `WHIZARD` 1
  Standard suffix: `.lha`
- `lhef`: Formatted Les Houches Accord implementation that contains the XML headers
  Standard suffix: `.lhef`
- `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 HEPRUP/HEPEUP common blocks (only available if StdHEP is installed and correctly linked)
  Standard suffix: `.up.stdhep`

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 format 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 (in the sequel, we gave the numbers out as single precision for better readability), for specifications of the event formats correspond the different accords and publicatios:

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 our example process $e^+e^- \to \mu^+\mu^-$, 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   1.00000000
         2          11           0           0           3           4           0
 0.00000000        0.00000000        249.999999        250.000000       5.11003380E-004
 0.00000000        0.00000000        0.00000000        0.00000000       0.000000000
         2         -11           0           0           3           4           0
 0.00000000        0.00000000       -249.999999        250.000000       5.11003380E-004
 0.00000000        0.00000000        0.00000000        0.00000000       0.00000000
         1          13           1           2           0           0           0
 225.985918       -80.1076510        70.8033735        250.000000       0.10565838
 0.00000000        0.00000000        0.00000000        0.00000000       0.00000000
         1         -13           1           2           0           0           0
-225.985918        80.1076510       -70.8033735        250.000000       0.10565838
 0.00000000        0.00000000        0.00000000        0.00000000       0.00000000
```

ASCII SHORT:

This is basically the same as the HEPEVT standard, but very much abbreviated. The header line for each event is identical, but first line per particle does only contain the PDG and the polarization, while the vertex information line is omitted.

```
         4           2           0   1.00000000
        11           0
 0.00000000        0.00000000        249.999999        250.000000       5.11003380E-004
       -11           0
 0.00000000        0.00000000       -249.999999        250.000000       5.11003380E-004
```

```
          13           0
  225.985918 -80.1076510   70.8033735   250.000000   0.10565838
         -13           0
 -225.985918  80.1076510  -70.8033735   250.000000   0.10565838
```

ASCII LONG:

Identical to the ASCII short format, but after each event there is a line containg two values: the value of the sample function to be integrated over phase space, so basically the squared matrix element including all normalization factors, flux factor, structure functions etc.

```
           4           2           0   1.00000000
          11           0
  0.00000000  0.00000000   249.999999   250.000000   5.11003380E-004
         -11           0
  0.00000000  0.00000000  -249.999999   250.000000   5.11003380E-004
          13           0
  225.985918 -80.1076510   70.8033735   250.000000   0.10565838
         -13           0
 -225.985918  80.1076510  -70.8033735   250.000000   0.10565838
  435.480971  1.00000000
```

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.

```
           1           4
           1           2          11            0           0      3          4
  0.00000000  0.00000000   249.999999   250.000000   5.11003380E-004
  0.00000000  0.00000000   0.00000000   0.00000000
           2           2         -11            0           0      3          4
  0.00000000  0.00000000  -249.999999   250.000000   5.11003380E-004
  0.00000000  0.00000000   0.00000000   0.00000000
           3           1          13            1           2      0          0
  225.985918 -80.1076510   70.8033735   250.000000   0.10565838
  0.00000000  0.00000000   0.00000000   0.00000000
           4           1         -13            1           2      0          0
 -225.985918  80.1076510  -70.8033735   250.000000   0.10565838
  0.00000000  0.00000000   0.00000000   0.00000000
```

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 fr 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 [9].

```
 4     1      1.0000000000    500.000000     -1.000000      0.117800
11   -11    13   -13
 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    250.0000000000      0.0000000000      0.0000000000    249.9999999995
 2    250.0000000000      0.0000000000      0.0000000000   -249.9999999995
 3    250.0000000000    223.6404152843   -102.7925182666     43.8024162280
 4    250.0000000000   -223.6404152843    102.7925182666    -43.8024162280
```

LHEF:

This is the modern version of the Les Houches accord event format (LHEF), for the details confer the corresponding publication [11].

```
<LesHouchesEvents version="1.0">
<header>
  <generator_name>WHIZARD</generator_name>
  <generator_version>2.0.0</generator_version>
</header>
<init>
         11         -11   250.000000        250.000000               -1          -1          -1          -1           3
 0.347536454       1.413672505E-004   1.00000000                1
</init>
<event>
          4           1   1.00000000        500.000000       -1.00000000       0.117800000
         11          -1           0           0           0           0   0.00000000        0.00000000        249.999999
        -11          -1           0           0           0           0   0.00000000        0.00000000       -249.999995
         13           1           1           2           0           0   223.640415       -102.792518        43.8024162
        -13           1           1           2           0           0  -223.640415        102.792518       -43.8024162
</event>
</LesHouchesEvents>
```

Sample files for the default ASCII format as well as for the debug event format are shown in the appendix.

8.1.4 Negative weight events

Chapter 9

Technical details – Advanced Spells

9.0.5 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 10

New Models via FeynRules

Appendix A

SINDARIN Reference

This appendix is out-of-date and needs revision.

In the SINDARIN language, there are certain pre-defined constructors or commands that cannot be used in different context by the user, which are – in alphabetical order – `alias`, `all`, `$analysis_filename`, `and`, `as`, `any`, `beams`, `cmplx`, `combine`, `compile`, `cuts`, `$description`, `echo`, `else`, `exec`, `expect`, `false`, `?fatal_beam_decay`, `if`, `include`, `int`, `integrate`, `iterations`, `$label`, `lhapdf`, `library`, `load`, `luminosity`, `model`, `n_events`, `no`, `observable`, `or`, `$physical_unit`, `plot`, `process`, `read_slha`, `real`, `?rebuild`, `?recompile`, `record`, `$restrictions`, `results`, `$sample`, `sample_format`, `scan`, `seed`, `show`, `simulate`, `sqrts`, `then`, `$title`, `tolerance`, `true`, `unstable`, `?vis_channels`, `write_analysis`, `write_slha`, `$xlabel`, and `$ylabel`. Also units are fixed, like `degree`, `eV`, `keV`, q `MeV`, `GeV`, and `TeV`. Again, these tags are locked and not user-redefinable. There functionality will be listed in detail below. Furthermore, a variable with a preceding question mark, ?, is a logical, while a preceding hash, #, 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 be marked by a hash, #, or an exclamation mark, !.

- `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`)

- `$analysis_filename`

This character variable allows to create a LaTeXfile for the user anaylsis, and to specify its name. If this variable is not set, the analysis will be directed to the screen output. (cf. also `write_analysis`)

- `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 `or`).

- `as`
  cf. `compile`

- `ascii`
  Specifier for the `sample_format` command to demand the generation of the standard `WHIZARD` verbose ASCII event files. (cf. also `$sample`, `sample_format`)

- `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`)

- `athena`
  Specifier for the `sample_format` command to demand the generation of the ATHENA variant for HEPEVT ASCII event files. (cf. also `$sample`, `sample_format`)

- `beams`
  This specifies the contents and structure of the beams. 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 `lhapdf`, `ISR`, `EPA` and so on are switched on as e.g. `beams = p, p => lhapdf`. (cf. also `circe`, `circe2`, `lhapdf`).

- `int checkpoint`
  Setting this variable to a positive integer n instructs simulate to print out a progress summary every n events.

- `cmplx`
  Defines a complex variable. (to be finalized still

- `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 defined, of course).

- `compile`
  The `compile` command is mandatory, 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 `processes.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`, `load`)

- `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`)

- `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).

- `$description`
  String variable that allows to specify a description text for the analysis, `$description = "analysis description text"`. This line appears below the title of a corresponding analysis, on top of the respective plot. (cf. `analysis`, `$title`)

- `echo`
  Allows to put verbose information on the screen during execution, e.g. `echo ("Hello, world!")`. (cf. also `show`)

- `else`
  cf. `if`

- `eV`
  Physical unit, stating that the corresponding number is in electron volt.

- `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.

- `expect`
  The binary function `expect` compares two numerical expressions whether they are fulfill a certain ordering condition or are equal up to a specific uncertainty or tolerance which can bet set by the specifier `tolerance`, i.e. in principle it checks whether a logical expression is true. The `expect` function does actually not just check a value for correctness, but also records its result. If failures are present when the program terminates, the exit code is nonzero. The syntax is `expect (<num1> <log_comp> <num2>)`, where `num1` and `num2` are two numerical values (or corresponding variables) and `log_comp` is one of the following logical comparators: `<`, `>`, `<=`, `>=`, `==~`, `<>`, `~~`, `~`. (cf. also `<`, `>`, `<=`, `>=`, `==`, `<>`, `~~`, `~`, `tolerance`).

- `$extension_ascii`
  String variable that allows via `$extension_ascii = "<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`)

- `$extension_ascii_long`
  String variable that allows via `$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` 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`)

- `$extension_ascii_short`
  String variable that allows via `$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` 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`)

- `$extension_debug`
  String variable that allows via `$extension_debug = "<suffix>"` to specify the suffix for the file `name.suffix` to which events in a 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`)

- `$extension_hepevt`
  String variable that allows via `$extension_hepevt = "<suffix>"` to specify the suffix

for the file `name.suffix` to which events in the `WHIZARD` 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`)

- `$extension_hepmc`
  String variable that allows via `$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`)

- `$extension_lhef`
  String variable that allows via `$extension_lhef = "<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>.lhef`. (cf. also `sample_format`, `$sample`)

- `false`
  Constructor stating that a logical expression or variable is false, e.g. `?<log_var> = false`. (cf. also `true`).

- `?fatal_beam_decay`
  Logical variable that let the user decide whether the possibility of a beam decay is treated as a fatal error or only as a warning. An example is a process $bt \to X$, where the bottom quark as an inital state particle appears as a possible decay product of the second incoming particle, the top quark. This might trigger inconsistencies or instabilities in the phase space set-up.

- `GeV`
  Physical unit, energies in 10^9 electron volt. This is the default energy unit of WHIZARD.

- `hepevt`
  Specifier for the `sample_format` command to demand the generation 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`)

- `if`
  Conditional clause with the construction `if <log_expr> then <expr> else <expr>`. Note that there must be an `end if` statement. For more complicated expressions it is better to use expressions in parentheses: `if (<log_expr>) then {<expr>} else {<expr>}`. 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}`. The `then` constructor is not mandatory and can be omitted.

- `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`.

- `int`
  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>`. (cf. also `real` and `cmplx`)

- `integrate`
  The `integrate (<proc_name>)` { `<integrate_options>` } command invokes the integration (phase-space grid generation and Monte-Carlo sampling of the process `proc_name` (which can also be a list of processes) with the integration options `<integrate_options`. Right now the only option is to specify the number of iterations and calls per integration during the Monte-Carlo phase-space integration via `iterations = <n_iterations>:<n_calls>`. Note that this can be list, separated by colons, which breaks up the integration process into units of the specified number of integrations and calls each.

- `iterations`
  Option to set the number of iterations and calls per iteration during the Monte-Carlo phase-space integration process, cf. `integrate`.

- `keV`
  Physical unit, energies in 10^3 electron volt.

- `$label`
  This is a string variable, `$label = "label_name"` that allows to specify a label `label_name` for analysis plots on the x axis. It is only taken into account if the variable `$xlabel` has not been set, in which case it is overwritten by the string value of that variable. (cf. also `xlabel`, `ylabel`).

- `lha`
  Specifier for the `sample_format` command to demand the generation of the WHIZARD 1 LHA ASCII event format files. (cf. also `$sample`, `sample_format`)

- `lhapdf`
  This is a specifier to demand calling LHAPDF parton densities to integrate processes in hadron collisions. (cf. `beams`)

- `lhef`
  Specifier for the `sample_format` command to demand the generation of the Les Houches Accord (LHEF) event format files, with the XML headers. (cf. also `$sample`, `sample_format`)

- `library`
  The command `library = "<lib_name>"` allows to specify a separate shared object library archive `lib_name.so`, not using the standard library `processes.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`, `load`)

- `load`
  The `load` command allows to load again a library if some details have been changed (processes added, redefined or maybe changed. (cf. also `compile`, `library`)

- `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` This specifier `luminosity = <num>` sets the integrated luminosity 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. Furthermore, the `luminosity` or `n_events` command has to be invoked *after* the corresponding logical variable which tells WHIZARD to write an event file in a specific format. (cf. `n_events`, `$sample`, `sample_format`)

- `MeV`
  Physical unit, energies in 10^6 electron volt.

- `model`
  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 7.1. 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.

- `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`)

- `n_events`
  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. Furthermore, the `n_events`

or `luminosity` command has to be invoked *after* the corresponding logical variable which tells WHIZARD to write an event file in a specific format. (cf. `luminosity`, `$sample`, `sample_format`)

- `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`)

- `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`).

- `$physical_unit`
  This is a string variable, `$physical_unit = "<unit_name>''`, that allows to set a LaTeXname `unit_name` for the physical unit of a label of an analysis plot. This unit is then also used for calculations within the analysis set-up.

- `plot`

  (cf. `record`)

- `process`
  Allows to set a hard interaction process, either for a decay process `decay_proc` as `process <decay_proc> = <mother> => <daughter1>, <daughter2>, ...`, or for a scattering process `scat_proc` as `<incoming1>, <incoming2> => <outgoing1>, <outgoing2>, ...`. Note that there can be arbitrarily many processes to be defined in a SINDARIN input file. (cf. also `restrictions`)

- `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`)

- `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 integer variable `real_var`: `real <real_var> = <real_val>`. (cf. also `int` and `cmplx`)

- `real epsilon`
  Predefined real; the relative uncertainty instrinsic to the floating point type used by WHIZARD.

- `int real_precision`
  Predefined integer; the decimal precision of the floating point type used by WHIZARD.

- `int range`
  Predefined integer; the decimal range of the floating point type used by WHIZARD.

- `real tiny`
  Predefined real; the smallest number which can be represented by the floating point type used by WHIZARD.

- `?rebuild`
  The logical variable `?rebuild = true/false` specifies whether the matrix element code for processes is re-generated by the matrix element generator O'Mega (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`).

- `?recompile`
  The logical variable `?recompile = true/false` specifies whether 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`)

- `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`. (cf. also `eval`)

- `$restrictions`
  This is an optional argument for process definitions. 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 matrix element 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 $e^-e^+ \to \mu^-\mu^+$ to the s-channel Z-boson exchange. (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`)

- `$sample`
  String variable to set the (base) name of the event output format, e.g. `$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_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`, `ascii`, `athena`, `debug`, `long`, `short`, `hepmc`, `lhef`, `lha`, `stdhep`, `stdhep_up`. (cf. also `$sample`, `simulate`, `hepevt`, `ascii`, `athena`, `debug`, `long`, `short`, `hepmc`, `lhef`, `lha`, `stdhep`, `stdhep_up`)

- `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.

- `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.

- `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. (cf. also `echo`, `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`)

- `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.

- `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 <particle_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`)

- `TeV`
  Physical unit, for energies in 10^{12} electron volt.

- `then`
  Alternative option inside a conditional clause, not mandatory, hence maybe be omitted, cf. `if`.

- `$title`
  This string variable sets the title of a plot in a WHIZARD analysis setup, e.g. a histogram or an observable. The syntax is `$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 `$description`, `$label`, `$xlabel`, `$ylabel`).

- `tolerance`
  Real variable that defines the tolerance with which the (logical) function `expect` accepts

```
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.*

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`).

- `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 these have to be provided by the user as in the example in Fig. A.1. First, the Z decays to electrons and up quarks are generated, then ZZ production at a 500 GeV ILC is called, and then both Zs are decayed according to the probability distribution of the two generated decay matrix elements. This obviously allows also for inclusive decays. (cf. also `stable`)

- `?vis_channels`
  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 according to their properties: `integrate (foo)  iterations=3:10000 ?vis_channels = true`. The default is `false`. (cf. also `integrate`)

- `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 `$analysis_filename` is provided, the analysis (including the histograms) are printed out on the screen, otherwise they are written to a file defined by that specific string variable. (cf. also `$analysis_filename`)

- `write_slha`

Demands WHIZARD to write out a file in the SUSY Les Houches accord (SLHA). (Cf. also `read_slha`)

- `$xlabel`
  String variable, `$xlabel = "<LaTeX code>"`, that sets the x axis label in a plot or histogram in a WHIZARD analysis. (cf. also `label` and `$ylabel`)

- `$ylabel`
  String variable, `$ylabel = "<LaTeX code>"`, that sets the y axis label in a plot or histogram in a WHIZARD analysis. (cf. also `label` and `$xlabel`)

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 continuos 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, Comput. Phys. Commun. **120**, 13 (1999) [arXiv:hep-ph/9806432].

[6] T. Ohl, Comput. Phys. Commun. **101**, 269 (1997) [arXiv:hep-ph/9607454].

[7] M. Skrzypek and S. Jadach, Z. Phys. **C49** (1991) 577.

[8] A. Djouadi, J. Kalinowski, M. Spira, Comput. Phys. Commun. **108** (1998) 56-74.

[9] E. Boos *et al.*, arXiv:hep-ph/0109068.

[10] P. Z. Skands *et al.*, JHEP **0407**, 036 (2004) [arXiv:hep-ph/0311123].

[11] J. Alwall *et al.*, Comput. Phys. Commun. **176**, 300 (2007) [arXiv:hep-ph/0609017].

[12] K. Hagiwara *et al.*, Phys. Rev. D **73**, 055005 (2006) [arXiv:hep-ph/0512260].

[13] B. C. Allanach *et al.*, 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].

[14] M.E. Peskin, D.V.Schroeder, *An Introduction to Quantum Field Theory*, Addison-Wesley Publishing Co., 1995.

[15] J. A. Aguilar-Saavedra *et al.*, arXiv:hep-ph/0511344.

[16] W. Giele *et al.*, arXiv:hep-ph/0204316; M. R. Whalley, D. Bourilkov and R. C. Group, arXiv:hep-ph/0508110; D. Bourilkov, R. C. Group and M. R. Whalley, arXiv:hep-ph/0605240.

[17] M. Dobbs and J. B. Hansen, Comput. Phys. Commun. **134**, 41 (2001).