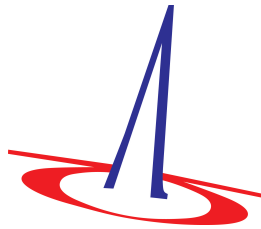


WHIZARD -Tutorial



Terascale Monte Carlo School 2014

March 10, 2014

1 How to use this tutorial

WHIZARD is a program system designed for the efficient calculation of multi-particle scattering cross sections and simulated event samples. The project web page can be reached via the URL

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

This tutorial will walk you through the basic usage of WHIZARD and take you to the point where you can generate event samples which match those which you will be analyzing later. Before you start working on the tutorial, take a little time to familiarize yourself with the virtual machine environment.

Whenever you encounter a line starting with a dollar sign \$, the remainder is a command which you should run in the shell. Boxed frames filled with code are SINDARIN scripts (WHIZARDs flavor of input files) which you should copy into your favorite editor and run through WHIZARD. Typewrite-style paragraphs are sample WHIZARD output.

At several points, you will encounter sections labelled “Self-Study”. Those are suggestions for modifications of the examples presented in this tutorial which you might want to try out in order to get a deeper understanding of how the program works. Feel free try out your own ideas as well. You can find a copy of the WHIZARD manual on the virtual machine in

```
/opt/share/whizard/doc/manual.pdf
```

and several examples in

```
/opt/share/whizard/examples
```

As WHIZARD produces a number of files during its run, please use different directories for different processes.

2 First steps

In this first part of the tutorial, you will learn how to invoke `WHIZARD` and get a short overview over its input language `SINDARIN`. No physics in this section.

2.1 Invoking the program

After installation (its already installed in the VM), `WHIZARD` is available as a standalone program which can be executed by calling the binary:

```
$ whizard
```

The program now starts, prints its banner and then waits for input on `stdin`. For now, we can terminate the run by pressing `ctrl-d` in order to send an end-of-file. Of course, the usual way to run `WHIZARD` is not to read from `stdin` but to supply an input file as an argument when calling the program.

After `WHIZARD` has terminated, inspect the directory in which you ran the program. You will find that it has left you a file called `whizard.log`. This file contains a copy of most of the output also sent to `stdout` during the run.

2.2 Talking to the WHIZARD: SINDARIN and “Hello, World!”

`WHIZARD` expects input in its own scripting language called `SINDARIN`. Therefore, start out by writing the `SINDARIN` flavor of the usual “Hello, World!” program. Create a file called `hello_world.sin` containing

```
printf "Hello, World"
```

and run it through `WHIZARD` via

```
$ whizard ./hello_world.sin
```

2.3 Variables

`WHIZARD` predefines many variables and also allows you to define your own. As with most languages, `SINDARIN`s variables come in different types. Among others, there are `integer`, `real`, `complex`, `string` and `logical` variables, the last two of which are prefixed with `$` and `?` respectively. In order to get a list of the predefined variables, augment your script by another line such that it reads

```
printf "Hello, World"
show ()
```

and rerun it. `show` is the second `WHIZARD` command we encounter. Used without argument, it prints a list of all defined variables, but arguments such as `model` or `???` are possible. You can easily get a scrollable version of the same list via

```
$ echo 'show ()' | whizard | less
```

(a nice trick if you are unsure how a variable is called or want to find out the defaults). Take some time to inspect the result. Most variables act as options controlling the behavior of `WHIZARD` commands¹, e.g. `$restrictions` or `?alpha_s_is_fixed`. Further down the list, you will find variables of real type like `GF` and `mZ` which represent parameters of the currently active model and which you can modify in order to change the model values. Note that some of those are marked by an asterisk `*`: this tells you that they are automatically calculated and cannot be changed by assignment. At the bottom of the variable list you'll find a long list of more exotic definitions like

```
down* = PDG(1)
dbar* = PDG(-1)
```

Those are “PDG array” type variables which bind one or more PDG numbers to a name. They are defined in the model files and are used to refer to particles when defining processes and observables.

In order to see some of the things you can do with variables, try another `SINDARIN` script (or augment your existing one)

```
real conv = 360. / 2. / pi
printf "Weinberg angle, default value [degrees]: %f" (asin (sw) * conv)
mW = 70 GeV
printf "Weinberg angle, new value [degrees]: %f" (asin (sw) * conv)
```

Note that the line breaks are not mandatory; `SINDARIN`'s syntax is not line based. The output should look like

```
[user variable] conv = 5.729577951308E+01
Weinberg angle, default value [degrees]: 28.127416
SM.mW = 7.000000000000E+01
Weinberg angle, new value [degrees]: 39.857282
```

¹ The first entry `seed => [integer]` is not a variable but shows the currently used random number seed.

What happened?

1. We defined a new real variable called `conv` and assigned it the conversion factor from radians to degrees. Evidently, `WHIZARD` has some predefined constants like `pi`. Note you must explicitly declare the type when you use a new variable for the first time: `integer`, `real`, `complex`, `string` or `logical`. Every assignment is reflected in the program output, making it easy to find out what happened after the run.
2. We used `printf` to print the value of the Weinberg angle. The formats of the values are defined in the message string, and the actual values are given as a comma separated list in parenthesis. `WHIZARD` accepts most of the format specifiers also used in C and other languages. We also used a function, `asin`, to get the value of the angle.
3. A new value was assigned to the W mass. Note that although their use is not mandatory, `WHIZARD` supports and encourages the use of units where appropriate. You can find a list of them in the manual; the default energy unit is GeV. The output `SM.mW = ...` confirms that we indeed modified a model input parameter.
4. The next `printf` statement reflects the fact that the Weinberg angle is not a free quantity but does depend on the W mass. The corresponding relation is defined in the model file.

2.4 Other SINDARIN constructs

Although we will not cover them in this tutorial, `SINDARIN` has several additional constructs common to programming languages:

- **Loops.** `SINDARIN` supports scanning over variables, a feature which can be exploited for parameter scans,
- **Conditionals.** The usual `if...then...else` construct exists and can be used for code blocks and in expressions. The latter can be very useful in defining observables for histogramming (more later).
- **`sprintf`.** This works similarly to `printf`, but returns a string. Allows e.g. for automatic generation of different histograms in a loop.

All of these features are documented in the manual.

2.5 Self-Study

Play a bit around with variable assignments and expressions, and try to find out which functions `WHIZARD` supports for use in expressions and how common operators look like and work. If you like, look up loops and conditionals in the manual and try them out.

3 A first stab at physics: $e^+e \rightarrow W^+W$

In this section we will use the trivial example of $e^+e \rightarrow W^+W$ to see how the basic functionality of a tree level Monte Carlo works in WHIZARD: process definition, integration and event generation.

3.1 Process definition and integration

Create a SINDARIN script with the following content

```
! Define the process
process proc = "e+", "e-" => "W+", "W-"

! Deactivate visualization of channel history
?vis_history = false

! Compile the process into a process library
compile

! Set the process energy
sqrt_s = 500 GeV

! Integrate the process
integrate (proc)
```

(note the appearance of comments) and run it through WHIZARD. You will be greeted by a lot of output. What happened?

3.1.1 Process definition and code generation

The first line `process proc = ...` defines our W pair production process and assigns the name `proc` with which we will refer to it in the remainder of the script. Note the appearance of quotation marks—those are needed to prevent WHIZARD from interpreting the `+` and `-` as operators. However, most WHIZARD models define several aliases for each particle. Instead of `e+`, `e-`, `W+` and `W-` one could also use `E1`, `e1`, `Wp` and `Wm`. An exhaustive list of definitions can be found in the model file

```
/opt/share/whizard/models/SM.mdl
```

Matrix elements for WHIZARD are generated by a separate matrix element generator called O'MEGA. For every process, O'MEGA generates a piece of FORTRAN code which is dynamically

compiled and loaded by WHIZARD. The compile statement in the second line triggers the code generation and the compilation into a process library, which is then loaded.²

3.1.2 Phase space parameterization and integration

The third statement in the script, `sqrts = ...`, sets the center of mass energy of the process and is then followed by the final `integrate` statement, which takes the name(s) of the process(es) to be integrated in parenthesis, separated by commas. The first interesting bit of output from the `integrate` command is

```
| Phase space: generating configuration ...
| Phase space: ... success.
| Phase space: writing configuration file 'proc_i1.phs'
| Phase space: 3 channels, 2 dimensions
```

For efficient integration of multileg cross sections, WHIZARD employs a multichannel Monte Carlo integrator (VAMP). Each channel corresponds to a separate phase space parametrization, automatically tailored to map out a class of singularities, combined with a Monte Carlo grid and a weight. During adaption, grids and weights are iteratively optimized. What does the above output signify? WHIZARD starts out looking for an existing phase space parameterization for the process and, upon discovering that none exists, generates a new one. For our trivial example, this step is instantaneous, but for more complicated (multijet) processes, it may take a finite amount of time.

After the channels and grids have been set up, WHIZARD starts the adaption and integration process. The output from this process reads

```
| Starting integration for process 'proc'
| Integrate: iterations not specified, using default
| Integrator: 2 chains, 3 channels, 2 dimensions
| Integrator: Using VAMP channel equivalences
| Integrator: 1000 initial calls, 20 bins, stratified = T
| Integrator: VAMP
|=====|
| It      Calls  Integral[fb]  Error[fb]  Err[%]  Acc  Eff[%]  Chi2 N[It] |
|=====|
| 1        864  7.2138993E+03  5.97E+01   0.83    0.24*  26.15
| 2        776  7.2042129E+03  3.62E+01   0.50    0.14*  44.66
| 3        776  7.2335102E+03  4.02E+01   0.56    0.15   44.11
|-----|
| 3        2416  7.2167572E+03  2.45E+01   0.34    0.17   44.11   0.15   3
|-----|
| 4        9984  7.1964029E+03  4.65E+00   0.06    0.06*  45.49
```

² The explicit invocation of `compile` is not mandatory. If you omit it, the program will automatically generate the matrix element upon integration or simulation.

5	9984	7.1954733E+03	4.72E+00	0.07	0.07	45.49		
6	9984	7.1910422E+03	4.79E+00	0.07	0.07	45.46		

6	29952	7.1943588E+03	2.73E+00	0.04	0.07	45.46	0.36	3
=====								
Time estimate for generating 10000 events: 0d:00h:00m:02s								

Each line corresponds to an adaption run in which the phase space is sampled and the grids and weights of the different channels are adapted. The whole adaption is separated into two batches of iterations, and only the results of the second batch are actually used to compute the integral (the first batch is also different in that only the grids are adapted and the weights are kept fixed). The asterisk denotes the “current best grid”. During event generation, the last one of those is used to sample the phase space. The default choices for the number of iterations and samples (“calls”) depend on the process under consideration and are usually sufficient to achieve a stable integration result. However, there are situations in which more control is desirable. This can be achieved by the `iterations` option. In order to see how it works, modify the example in the following way

```
integrate (proc) { iterations = 3:1000:"gw", 5:3000:"w", 5:10000 }
```

(enclosing the statement in curly braces localizes its effect to this specific `integrate` command) and observe how the output changes. The flags “g”, “w” or “gw” tells WHIZARD to adapt the grid, the weight or both of them at each iteration. If no flag is set neither the grid nor the weight will be adjusted. This is the preferable option for the final run.

3.1.3 Event generation and analysis

In order to see how event generation and analysis works in WHIZARD modify the previous example by appending the lines

```
! Define a histogram for the angular distribution
histogram angular_distribution (-1, 1, 2. / 30.) {
$title = "Angular distribution"
$x_label = "$\cos\theta_{W-}$"
}
analysis = record angular_distribution (eval cos (Theta) ["W-"])

! Generate 1 fb-1 of events
simulate (proc) {
```

```

luminosity = 1 / 1 fbarn
}

! Compile the analysis to a file
compile_analysis

```

The first statement defines a histogram; the three numbers in parenthesis denote the range and the bin width. Since we are going to histogram the cosine of the polar angle, our histogram goes from -1 to 1 , and we choose it to have 30 bins. The second statement `analysis = ...` assigns an analysis expression. This will be executed for every event generated in the simulation. As this expression introduces a lot of new stuff, lets break it up:

`["W-"]` defines a “subevent”. A subevent is a set of momenta associated with final state particles (or combinations of them). The subevent defined above is trivial in that it consists only of the W^- momentum. We could also have built a subevent with two separate momenta via `["W+": "W-"]` (which would have made no sense in this context) or have added up the momenta of both W bosons by writing `[collect["W+": "W-"]]`. There are many ways to manipulate subevents which allow to build quite elaborate observables that can be used in the context of cuts, scale and analysis expressions. You can find a list of them in the manual.

`eval` is a function which takes an expression and evaluates it in the context of a subevent.

`Theta` is an observable. An observable is a quantity which maps one or two four momenta to a number. Observables may only appear in the context of an `eval` function (or in the `all` and `any` functions which will be discussed later). In this example, we have used `Theta` as a unary observable, thus calculating the polar angle, but we could also have evaluated it on a pair of subevents (this is different from a single subevent with multiple momenta!) by doing `eval cos (Theta) ["W+", "W-"]`. In this context, `Theta` would have evaluated to the angle enclosed by the two W momenta. You can find a list of all available observables in the manual.

`record` takes a number and records it in a histogram. It is in fact a function returning a logical value, which allows to chain several `record` calls via the `and` operator in order to fill several histograms at once³. The result tells whether the observable lies in the histogram range.

So, in a nutshell, this definition will cause `WHIZARD` to calculate $\cos \theta_{W^-}$ for every event and bin the values in the previously defined histogram.

The actual simulation is triggered by the `simulate` command, with which we request the program to simulate 1 fb^{-1} of unweighted events. We could also have used `n_events = 10000`

³Obviously, `WHIZARD` does not short-circuit the `and`.

instead of `luminosity` in order to set the number of generated events directly. Finally, after performing the simulation, the `compile_analysis` command tells `WHIZARD` to write the analysis to disk and create a PDF⁴ containing any histograms and plots. The result can be found in `whizard_analysis.pdf`. Inspect it with a PDF viewer (the virtual machine provides `evince` for this purpose). Also, observe how we used \LaTeX when labelling the histogram. This works because `WHIZARD` in fact uses \LaTeX to generate the graphical analysis, so you can use whatever \TeX ish expressions you like.

During simulation, events are written to disk in a `WHIZARD`-specific format which contains all available information for each event and can be read back later (see the next self-study). We will later see how to instruct the program to provide additional event files in different formats.

3.2 More SINDARIN: options and global vs. local variables

Another new thing we encountered in the above `SINDARIN` snippets are command options. Most of these are just ordinary variables, the values of which influence the operation of `WHIZARD`. The only exception was `iterations` which does not correspond to a variable as it does not map to any of the available variable types. In addition, apart from `sqrts`, all of these variables were set inside `{...}` behind commands. The reason is simple: the effect of statements in curly brackets after commands is localized to the execution of this command—any changes are forgotten after the command has been executed. For example, we could also have moved `$title = ...` out of the brackets and put it before the `histogram =` This would have worked just as well, but we would have affected all subsequently defined histograms. Similarly, we could have put `sqrts = ...` into curly brackets after `integrate`, but `simulate` would have complained about a missing value for `sqrts` in this case.

3.3 Self-Study

`WHIZARD` has a checksumming and caching mechanism which tries to reuse as much information as it can from previous runs. Rerun the above example and change the setup and parameters a bit in order to find out how it works. There are also flags which control the caching; try to locate them in the `show` output and see how they work. There are command line options which do the same thing; check out

```
$ whizard --help
```

and try them.

4 Hadron Collider: pp initial state

Simple W pair production at a hadron collider is used as example to show how flavor sums and structure functions work. We will also add an additional jet to the final state and use the opportunity to show how cuts work.

⁴The virtual machine has low memory, if this does not work from the start, please increase the memory of your VM (2 GB). Nevertheless you can also see the analysis in `whizard_analysis.data`

4.1 Basic setup

In order to change our W pair production example to a proton-proton initial state and add a convolution with the parton distributions, change the above example such that the first few lines read

```
! Define the process
alias pr = u:ubar:d:dbar:g
process proc = pr, pr => "W+", "W-"

! Deactivate visualization of channel history
?vis_history = false

! Compile the process into a process library
compile

! Setup the beams
sqrts = 8 TeV
beams = p, p => pdf_builtin
```

What changed?

1. We have to accommodate for the composite initial state at a hadron collider. To this end, final and initial state particles in WHIZARD can be defined as flavor products of particles, separated by colons. In order to avoid repetition, an `alias` can be assigned to a flavor product, in this case `pr`⁵. In fact, assigning an `alias` creates a variable of the `PDG(...)` type which we already encountered in the variable list.
2. The cross section has to be convoluted with a structure function. This is accomplished by providing a beam setup via `beams = .`. The equality sign is followed by a pair of particle identifiers `p, p` which identify the hadronic initial states as protons, followed by the declaration of the requested structure function `=> pdf_builtin`. In this case we are using the PDFs built into WHIZARD, the default being CTEQ6L. If WHIZARD was built with LHAPDF support, `=> lhpdf` would be another choice which we will use later. Options for the choice of PDF set exist and are documented in the manual, and other structure functions are available for adding e.g. initial state photon radiation or simulating the beamstrahlung of a linear collider. Also, structure functions can be chained.

The changes in the resulting program output are not overly exciting, the most noteworthy being the summary of the structure function setup.

⁵ The more fitting identifier `p` is already taken by the actual proton, represented by its proper MCID.

4.2 Adding a jet and defining cuts

We now will add an additional jet to the final state of our W pair production example. The corresponding leading order matrix element has a divergence when the jet momentum becomes soft or collinear to the beam axis, and we therefore need a cut to remove it. Modify the first half of the example to read

```
! Define the process
alias pr = u:ubar:d:dbar:g
alias j = u:ubar:d:dbar:g
process proc = pr, pr => "W+", "W-", j

! Deactivate visualization of channel history
?vis_history = false

! Compile the process into a process library
compile

! Set the process energy
sqrt_s = 8 TeV
beams = p, p => pdf_builtin
cuts = all Pt > 5 GeV [j]
      and all 200 GeV < M < 2 TeV [collect ["W+": "W-": j]]

! Integrate the process
integrate (proc)
```

Apart from the additional jet in the final state, the only other new element is the introduction of two cuts

$$p_{T,\text{jet}} > 5 \text{ GeV}, \quad 200 \text{ GeV} \leq \sqrt{\hat{s}} \leq 2 \text{ TeV}$$

The first cut keeps the jet momentum away from the dangerous soft and collinear regions, the second cut is for demonstration purposes. Here is a detailed explanation:

- `[j]` and `[collect ["W+": "W-": j]]` are subevents. The first consists of the momenta of all final state particles which match the `j` alias (only a single momentum in our case), and the second contains the sum of all final state momenta.
- The `all` function takes a logical expression, evaluates it for all momenta in a subevent and concatenates the results with a logical **and**—a phasespace point passes the cut only if all momenta in the subevent satisfy the condition. On the other hand the function **any** accepts a phasespace point if at least one condition is true.

- The observables `Pt` and `M` evaluate to the transverse energy and the invariant mass.
- Both cuts are concatenated with a logical `and`.

Note that it is possible to define additional cuts which are only applied to the generated events—those are set up with `selection = ...` instead of `cuts =`

The output of the run does not deliver new insights.

4.3 Self-Study

Take a look at the generated angular distribution and compare it the leptonic case—where does the difference come from? If you like, try to extend the example to include the decay of the W bosons by using the inclusive matrix element for $pp \rightarrow e^+ \nu_e \mu^- \bar{\nu}_\mu$.

5 Showering of the semileptonic channel

In this last section you will learn how you can use `WHIZARD` to generate your signal sample and to enable the parton shower in `WHIZARD`.

5.1 Adding decays for the W s

We will change our `SINDARIN` file with the process $pp \rightarrow W^+W^-$ in such a way that the W s decay semileptonically. In order to implement this change, we just have to add three additional lines. Add two additional process definitions

```
process wpdecay = "W+" => u, dbar
process wmdecay = "W-" => e1, N1

integrate (wpdecay,wmdecay)
```

right after the existing one and two additional statements

```
unstable "W+" (wpdecay)
unstable "W-" (wmdecay)
```

just before the `simulate` statement and rerun the program. The `unstable` statement tells `WHIZARD` to treat a particle (the W in this case) as unstable and decay it during event generation

via the decay processes listed in parenthesis. The resulting cascade decays will fully preserve spin and color correlations.

The program output shows you that `WHIZARD` now automatically calculates the integral of the decay matrix element once the simulation is started.⁶ Take a look at the generated histogram—what has gone wrong? Try to fix it.

5.2 Parton shower and underlying event

In order to complete the simulation, we will now add a parton shower and the simulation of underlying event and hadronization to the generated events. Normally it is recommended to use `PYTHIA` for the shower, but this feature is not available in `WHIZARD 2.2.0 beta`. Therefore we will use native shower of `WHIZARD`. In order to activate it, add the following lines right before the `simulate` statement

```
! Define shower options
?allow_shower = true

?ps_fsr_active = true
?ps_isr_active = true
?ps_use_PYTHIA_shower = false
?ps_isr_only_onshell_emitted_partons = true
ps_max_n_flavors = 5
ps_mass_cutoff = 0.5 GeV
#$ps_PYTHIA_PYGIVE = "MSTP(5)=108;MSTP(81)=0;MSTJ(41)=1;MSTJ(21)=0;MSTP(68)=0"
```

lower the luminosity

```
luminosity = 1 / 10 fbarn
```

and rerun the program. Most options should be self-explanatory:

- `?allow_shower` activates the parton shower.
- `?ps_fsr_active` and `?ps_isr_active` add initial and final state radiation via the parton shower.
- `?ps_use_PYTHIA_shower` activates/deactivates `PYTHIA` shower.

⁶We could also have explicitly calculated the integral prior to the simulation using `integrate`.

- `ps_max_n_flavors` sets the number of active flavors in the parton shower.
- `ps_mass_cutoff` sets the cutoff scale of the shower.
- `$ps_PYTHIA_PYGIVE` passes additional options to PYTHIA via the PYGIVE call. Please refer to the PYTHIA manual for more information (not used in this tutorial).

Congratulations, you have now the knowledge to command `WHIZARD` via `SINDARIN` to generate your event samples you will later be analyzing with `RIVET`.

5.3 Self-Study

To pass the final test, execute the following two steps:

1. Change the analysis to reconstruct the invariant mass of W^+ .
2. Compare your result with the full $2 \rightarrow 4$ process $pp \rightarrow u\bar{d}e^-\bar{\nu}_e$.

References

- [1] W. Kilian, T. Ohl and J. Reuter, Eur. Phys. J. C **71** (2011) 1742 [arXiv:0708.4233].