HGF Monte Carlo School 2012

—

WHIZARD Tutorial

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 an 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 (WHIZARD's 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 "Things to try out". 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

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

and several examples in

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

As WHIZARD produces a number of files during its run, it is prudent to work with the examples in separate directories.

2 First steps

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

2.1 Invoking the program

After installation (it's 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!"

Once you start it, WHIZARD expects to be fed input in it's own scripting language called SINDARIN. Therefore, we will start out by writing the SINDARIN flavor of the usual "Hello, World!" program. Fire up your favorite editor and create a file called `hello_world.sin` with the following content

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

Then,run it through WHIZARD via

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

As expected, the output is now augmented by

```
Hello, World!
```

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 any arguments, it prints a list of all defined variables. 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 commands[1], e.g. `$restrictions` or `?alpha_s_is_fixed`. Further down the list, you will find variables of real type like `GF` and `mZ` which repesent parameters of the currently active model and which you can modifiy 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 number of PDG numbers to a name. They are defined in the model files and are used to refer to partciles when defining processes and observables. Again, the asterisk tells you that they cannot be modified.

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 linebreaks are not mandatory; SINDARIN's syntax is not line based. The resulting output should look like

```
[user variable] conv =    57.295779513082323
Weinberg angle, default value [degrees]: 28.127416
SM.mW =    70.000000000000000
Weinberg angle, new value [degrees]: 39.857282
```

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: `real`, `int`, `string`, `complex` 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.

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

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 appropiate. 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 Things to try out

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: $e^+e^- \to W^+W^-$

In this section we will use the trivial example of $e^+e^- \to W^+W^-$ 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-"

! Compile the process into a process library
compile
```

```
! Set the process energy
sqrts = 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. The syntax should be self-explanatory. Note the appearance of quotation marks — those are needed to prevent WHIZARD from interpreting the `+-` as operators. If you prefer to leave them out, you'll be happy to learn that most WHIZARD models define several aliases for each particle. Instead of `e+`, `e-`, `W+` and `W-` we could also have used `E1`, `e1`, `Wp` and `Wm`. If you'd like to have a look at the different aliases, you'll find their definitions in the model file

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

(or in the `show` variable dump).

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 Phasespace 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 commata.

The first interesting bit of output from the `integrate` commend is

```
| Phase space file 'proc.phs' not found.
| Generating phase space configuration ...
| ... done.
| ... found 3 phase space channels, collected in 2 groves.
| Phase space: found 3 equivalences between channels.
| Wrote phase-space configuration file 'proc.phs'.
| iterations = 3:1000, 3:10000
```

For efficient integration of multileg cross sections, WHIZARD employs a multichannel Monte Carlo integrator (VAMP). Each channel corresponds to a separate phasespace 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

[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.

above output signifify? WHIZARD starts out looking for a 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 setup, WHIZARD starts the adaption and integration process. The output from this process reads

```
| Integrating process 'proc':
|=============================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|=============================================================================|
   1       1000  7.2274951E+03  5.31E+01    0.74    0.23*  25.88
   2       1000  7.1931106E+03  2.60E+01    0.36    0.11*  42.06
   3       1000  7.2498115E+03  2.86E+01    0.39    0.12   50.50
|-----------------------------------------------------------------------------|
   3       3000  7.2198088E+03  1.81E+01    0.25    0.14   50.50    1.09   3
|-----------------------------------------------------------------------------|
   4      10000  7.1922502E+03  4.36E+00    0.06    0.06*  50.01
   5      10000  7.1900971E+03  3.04E+00    0.04    0.04*  50.56
   6      10000  7.1915657E+03  2.59E+00    0.04    0.04*  57.99
|-----------------------------------------------------------------------------|
   6      30000  7.1911688E+03  1.80E+00    0.02    0.04   57.99    0.10   3
|-----------------------------------------------------------------------------|
|=============================================================================|
   6      30000  7.1911688E+03  1.80E+00    0.02    0.04   57.99    0.10   3
|=============================================================================|
```

Each line corresponds to an adaption run in which the phasespace 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 phasespace.

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 desireable. 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 = 3:1000,5:3000,5:10000}
```

(enclosing the statement in curly braces localizes its effect to this specific `integrate` command) and observe how the output changes.

3.2 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 statements `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, let's 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 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.

- The `eval` function 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 we will discuss later). In this example, we have used `Theta` as an 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 once[3]. The result tells whether the observable lies in the histogram range.

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

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 $1\,\mathrm{fb}^{-1}$ of unweighted events. We could also have used `n_events = 10000` 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 PDF containing any histograms and plots. The result can be found in `whizard_analysis.pdf`. Fire up a PDF viewer (the virtual machine provides `xpdf` for this purpose) and inspect the result. Also, observe how we used LaTeX when labelling the histogream. This works because WHIZARD in fact uses LaTeX to generate the graphical analysis, so you can use whatever TeXish 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 things-to-try-out). We will later see how to instruct the program to provide additional event files in different formats.

3.3 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 curly braces 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 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'd 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.

3.4 Things to try out

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 fiddle around a bit with the setup and parameters 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 Going to a hadron collider: pp initial state

We now take the simple W pair production example to a hadron collider to show how flavor sums and structure functions work. We'll also add an additional jet to the final state and use the opportunity to show how cuts work.

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-"

! 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 accomodate for the composite intial state at a hadron collider. To this end, final and initial state particles in WHIZARD can be defined as flavor products of particles, seperated by colons. In order to avoid repetition, an `alias` can be assigned to a flavor product, in this case `pr`[4]. 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 CTE6L. If WHIZARD was built with LHAPDF support, `=> lhapdf` 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 summery of the structure function setup.

[4]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 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

! Compile the process into a process library
compile

! Set the process energy
sqrts = 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

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

The first cut keeps the jet momentum away from the dangerous soft and collinear regions, the second cut is only for demonstration purposes. Let's try to understand the structure of the cuts.

- `[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. `all` has a sibling called `any` with obvious semantics.
- 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 add anything to what we already know.

4.3 Things to do

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 $p, p \to e^+, \nu_e, \mu^-, \bar{\nu}_\mu$.

5 BSM Physics: Higgs production in vector boson fusion and decay to neutralino pairs in the MSSM

In the analysis session later you will be analyzing an event sample and looking for a signal coming from a MSSM Higgs produced in vector boson fusion and decaying to a pair of LSPs. In the simulated scenario, the LSP is the lightest neutralino, and the branching ration is nearly 50%.

In the last section of this tutorial we will now learn how this signal can be simulated with WHIZARD. You can also use the opportunity to study the characteristics of the signal; this might help you in determining suitable cuts to separate signal from background later.

5.1 Vector boson fusion in the MSSM

Create a new SINDARIN file in a new directory with the content

```
! Reset model and read in SLHA benchmark point
model = MSSM
read_slha ("bpoint_w1_slha.in") {?slha_read_decays = true}

mc = 0
ms = 0
mb = 0

! Process definition
alias pr = u:ubar:d:dbar:c:cbar:s:sbar:b:bbar:g
alias j = u:ubar:d:dbar:c:cbar:s:sbar:b:bbar:g

process vfusion = pr, pr => j, j, h

compile

! Integration
sqrts = 8 TeV
beams = p, p => lhapdf

cuts = all Pt > 10 GeV [j]
   and all abs (Eta) < 5 [j]

integrate (vfusion)

! Histogram: Higgs p_T
```

```
histogram hist_pt (0., 200., 10.) {
   $title = "Higgs $p_T$"
   $x_label = "$p_T$ [GeV]"
}
analysis = record hist_pt (eval Pt [h])

! Simulate
simulate (vfusion) {
   n_events = 10000
   checkpoint = 100
   sample_format = hepmc
}
compile_analysis
```

The necessary SLHA file can be found in

`/mcschool/opt/SLHA/bpoint_w1_slha.in`

Copy it to your working directory so that WHIZARD can find it before running this example.
What is new in the above code?

1. In the first line, the physics model is reset to the MSSM using the `model = ...` statement.

2. The next two lines read in the SLHA file and replace the default model parameters accordingly. Note that the option `?slha_read_decays = true` is necessary for `read_slha` to take the particle widths from the SLHA.

3. The charm, strange and bottom masses are reset to 0. This is necessary as we want all 5 light quark flavors to appear in the flavor sums which define the proton and the jets. As all particles in a flavor product are treated as a single particle during phase space generation, their masses have to match. If you like to fiddle around, try to comment them out and see what happens.

4. Instead of the builtin PDFs, we now use LHAPDF. This is a requirement of the built-in PYTHIA interface which we want to use later for showering and hadronizing the events. As we didn't specify a particular set, WHIZARD uses its default; look for it in the program output.

5. We added two new options to the `simulate` command: the `checkpoint` variable instructs WHIZARD to keep us updated on the progress of the generation every 100 events, and the `sample_formats = ...` is used to specify a comma separated list of additional event formats in which the events will be written to disk. Here, we generate HEPMC events which can be analyzed by RIVET. A list of the supported event formats can be found in the manual.

The run now takes considerably longer compared to our previous W pair production example, so, depending on your machine, you might want to reduce the number of generated events

if you get impatient. Just terminate WHIZARD using ctrl-c and edit the file; once you rerun WHIZARD, it will read in any events it already generated and pick up where it was interrupted.

The only exciting new piece of output is the progress report created by `simulate` due to the presence of then `checkpoint` option. Every 100 events, it gives you a progress indicator and estimates the time remaining until the full sample is generated. After the program has finished, take a look at the generated files and try to identify the generated HEPMC event file. Take a look at the p_T distribution; it might give you a hint at the cuts which you might later want to apply in your signal analysis.

5.2 Adding the Higgs decay

We will now add the decay of the Higgs to a neutralino pair. While we could do this by just simulating the full inclusive $p, p \to j, j, \chi_1^0, \chi_1^0$, we are just interested in the Higgs production signal, so we will simply add a cascade decay (which is considerably faster).

In order to implement this change, we just have to add two additonal lines. Add an additonal process definition

```
process hdecay = h => neu1, neu1
```

right after the existing one and the additional statement

```
unstable h (hdecay)
```

just before the `simulate` statement and rerun the program. The `unstable` statement tells WHIZARD to treat a particle (the Higgs 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. Of course, this is of no consquence here as we are decaying a scalar particle.

The program output shows you that WHIZARD now automatically calculates the integral of the decay matrix element once the simulation is started[5] Take a look at the generated histogram — what has gone wrong? Try to fix it.

5.3 Parton shower, underlying event and hadronization using PYTHIA

In order to complete the simulation, we now will add a parton shower and the simulation of underlying event and hadronization to the generated events. Although WHIZARD has finally gotten a native shower, it is still considered beta, and the PYTHIA shower via the built-in PYTHIA interface is the recommended choice at the moment. In order to activate it, add the following lines right before the simulate statement

```
?ps_fsr_active = true
?ps_isr_active = true
?hadronization_active = true
ps_max_n_flavors = 5
ps_mass_cutoff = 1 GeV
$ps_PYTHIA_PYGIVE = "MDCY(C1000022,1)=0;MSTP(68)=0;MSTP(5)=108"
```

[5] We could also have explicitly calculated the integral prior to the simulation using `integrate`.

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

- `?ps_fsr_active` and `?ps_isr_active` activate the PYTHIA interface and add initial and final state radiation via the parton shower.
- `?hadronization_active` activates hadronization.
- `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. In this case, we prevent PYTHIA from trying to decay the neutralino, switch off matching and select a particular PYTHIA tune. Please refer to the PYTHIA 6 manual for more information.

That's it. Congratulations, you now have a working version of the very same WHIZARD configuration which was used to create the event samples you will later be analyzing with RIVET. Take a look at the generated HEPMC event file and convince yourself that it contains a whole bunch of hadrons instead of the two partons in the final state ;)

5.4 Things to do

Look at the distributions of various observables for the two parton level jets (without shower and hadronization) and try to identify suitable discrimination cuts. Try to run the generated events (with shower and hadronization) through the RIVET analysis that you will be using later. Hint: you can change the name of the generated event file to `xxx.hepmc` via `$analysis = "xxx"`, and you can specify an alternate file extension with `$extension_hepmc`.