WHIZARD 辅导

北京大学

2014年8月28日

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 (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 "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

`Desktop/manual.pdf`

and several examples in

`/usr/local/share/whizard/examples`

As WHIZARD produces a number of files during its run, you may want to use different directories for different processes.

Remarks on Installation

The VM provides you with a complete environment, so for the purpose of this tutorial, you don't have to worry about program installation. However, for further studies, you may wish to install WHIZARD yourself, outside the VM. You will find a detailed description of the options, necessary steps and requirements in the manual.

At this place, we just list a few important issues: (i) WHIZARD can be installed centrally on a machine, or in a user directory. In both cases, your work projects remain separate from the installation. (ii) WHIZARD's programming languages are modern Fortran and Objective Caml. For both languages, there are free compilers that are available for all relevant platforms, packaged for standard OS distributions. In particular, the free gfortran compiler will work if you have at least version 4.7, preferably 4.8 or higher.[1] (iii) WHIZARD can optionally make use of external packages, which should be installed if you want their functionality. Notable examples are LHAPDF, StdHEP, or HepMC.

As a shortcut, there is also the instant-whizard script which automatically downloads, compiles, and installs are necessary software.

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!"

WHIZARD expects input in it's 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

[1] Note that there are Fortran compilers that don't work with WHIZARD, yet.

```
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 variants such as `show (model)` or `show (mW, mZ, mtop)` 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 commands, 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 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 = 180 / 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. You won't see characters for terminating or separating statements, either. Statements just follow each other.

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
```

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 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.

Incidentally, the variable `conv` that we defined above is already available in form of a unit. You may simplify the SINDARIN code above to

```
printf "Weinberg angle, default value [degrees]: %f" (asin (sw) / 1 degree)
mW = 70 GeV
printf "Weinberg angle, new value [degrees]: %f" (asin (sw) / 1 degree)
```

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 filenames for output 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: $e^+e^- \to W^+W^-$

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. 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

```
/usr/local/share/whizard/models/SM.mdl
```

Matrix elements for WHIZARD are generated automatically 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 commata. The first interesting bit of output from the integrate commend 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

[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.

```
| 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
   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 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:"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, both the grid and the weight will be adjusted. The exception is the final integration pass, in which grid and weights are frozen, unless specified otherwise.

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) {
n_bins = 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, 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 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 once [3] . 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` 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`. 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 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 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

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

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.

3.4 Event Files

You may have noticed that so far, all data stayed within `WHIZARD`, and only final results were printed on screen or ended up in a PDF document. However, for an actual analysis you would like to see the events themselves.

Actually, `WHIZARD` did generate an event file. You can recognize it by the extension `.evx`. This file is written in a private binary format for internal use, so you can't make much use of it outside `WHIZARD`. If you want to have events in a readable format (by a human or by a computer), you can generate it. For instance, this simulation command converts the events in LHE (Les Houches Event) format:

```
! Generate 1 fb^-1 of events and write to file
simulate (proc) {
  n_events = 10
  $sample = "my_events"
  sample_format = lhef
}
```

Such event files can be fed into external programs. If quarks and gluons are present (later we'll see how to produce complete events directly), you may wish to run an external shower and hadronization program over this file, before the events enter analysis.

3.5 Beam Properties

In our first example, the incoming particles had a fixed, well-defined energy. Since reality is different, we should mention how to come to a more detailed beam description.

Let us look first at ILC physics. (If you are interested in LHC only, you may skip to the next section.) In e^+e^- collisions the initial electrons can radiate a significant fraction of their initial energy before the collision. This is summarized in the ISR (initial-state radiation) approximation, which you can turn on by a `SINDARIN` command, just before `integrate`.

```
beams = "e+", "e-" => isr
```

This should slightly modify the final results.

At a Linear Collider, you must also consider the beamstrahlung effect which also reduces the available energy, before the ISR effect comes into play. This spectrum can become rather complex, so WHIZARD relies on external code. The simplest approach uses the CIRCE 1 beam-events generator:

```
beams = "e+", "e-" => circe1 => isr
```

Of course, this approach is available only for specific beam setups, for which the beam parameters are known in some detail.

3.6 Self-Study

Look up the options for ILC beam description in the manual and try to apply them to the example. For instance, you may polarize the beam particles and watch the cross section change.

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.

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 CTEQ6L. 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 summary of the structure function setup.

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

! 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]]
```

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

```
! 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

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

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

5 The `FeynRules` Interface

The `FeynRules` interface is intended to import new physics models formulated in terms of a Lagrangian in a `FeynRules` model file into `WHIZARD`. More precisely, it is actually a `WHIZARD` interface included in the `FeynRules` release from version 1.6.0 onward. It should be used together with `WHIZARD` v2.0.3 or higher by default. The usage of the interface is documented in the manuals of both packages: the `FeynRules` manual can be found here, but the respective section in the `WHIZARD` manual is somewhat more detailed, including particularly a discussion of the limitations which are still present, and which we will come back to later on in Sec. 5.3.

5.1 Getting started with FeynRules

In order to do this part of the tutorial you must have an up-to-date WHIZARD installation on your own machine. If you have not compiled the package so far, you can obtain a tarball of the current version here (follow the instructions in Sec. 7 to compile the source). In addition, you need a working Mathematica installation as well as FeynRules at hand on your own laptop or desktop machine. If you lack the former, then unfortunately you will have to conclude this section by reading through the documentation sections of the manuals (if you still like), or asking your tutorial instructor to demonstrate it on his machine. Otherwise, if necessary obtain the FeynRules source from here, unpack it into a fresh directory, and finally in a new Mathematica notebook do

```
$FeynRulesPath = SetDirectory["<FEYNRULES-DIR>"];
<< FeynRules`
```

to load the package.

If you're already familiar with FeynRules, feel free to go ahead to Sec. 5.2. If not, note first and foremost that the package was developed to convert the Lagrangian of a given model into the resulting Feynman rules, so any input to FeynRules will be in the form of a proper Lagrangian which should fulfill basic sanity requirements such as hermiticity. Try to get a first idea of the way it works by inspecting the FeynRules version of the standard model in the respective model file under

```
<FEYNRULES-DIR>/Models/SM/SM.fr
```

As you will discover, the file in principle just consists of

- a list of the SM gauge groups and their representations,
- a list of all fields (both unphysical gauge interaction states as well as physical external states including the diagonalization transformation),
- a list of external model parameters to be set freely by the user,
- a list of internal model parameters which are fixed by the values of the external parameters and other model constraints,
- and finally a set of Lagrangian pieces of which the SM Lagrangian consists, which are all added up in the end to form the SM Lagrangian LSM.

Since WHIZARD is a tree-level tool, you can safely ignore anything related to ghosts in the model. However, in order to solve the tasks in the following hands-on sections, it is fruitful to have a closer look at the structure of the Lagrangian pieces LYukawa for Sec. 5.2, and LGauge for Sec. 5.3. Furthermore, FeynRules provides a routine to automatically check the hermiticity of a Lagrangian: load the model and try it, stating

```
LoadModel["Models/SM/SM.fr"];
CheckHermiticity[LSM];
```

in your notebook.

As a first example, export the SM via the WHIZARD interface and employ it with your SM processes from the previous tutorial sections. This is done by invoking

```
FeynmanGauge = False;
WriteWOOutput[LSM];
```

Check the Mathematica messages: the interface calculates the Feynman rules and gives an account of the number of vertices processed. You should find something like

```
processed a total of 75 vertices, kept 74 of them and threw
  away 1, 1 of which contained ghosts or goldstone bosons.
```

which tells you that the only discarded vertex is the ghost interaction of QCD which decouples at tree level, as stated before. Note that the FeynmanGauge flag is mandatory in order to switch to unitary gauge, because FeynRules defaults to Feynman gauge. You could of course check consistency by keeping it at True, but you'll have to communicate this to the interface with an option to the Write command, WriteWOOutput[LSM, WOGauge -> WOFeynman]. However, as you'll see from the messages once you try it, the model becomes less efficient, because now the goldstone bosons contribute, leading to a whole bunch of extra vertices (general R_ξ gauges are possible as well, cf. the WHIZARD manual). Next, cd to the output directory of the interface, to be found directly under the FeynRules top-level directory. The name is inferred from the model name in the .fr file, so in this case it should be fr_standard_model-WO/. Inside the directory, do

```
./configure --prefix=<YOUR-WHIZARD-INSTALLATION>
make install
```

to compile the code and inject the model into your WHIZARD installation. If everything worked out, the new model is now available to WHIZARD under the model name fr_standard_model. Try your SINDARIN scripts with it, and e. g. compare to the built-in version of the SM to check whether they agree.

5.2 A simple example: SM with a hadrophobic heavy Z'

Now let's move on to something non-trivial: extend the SM by adding one new particle and one new coupling. For instance, try to implement a new heavy vector boson Z' with mass $m_{Z'} = 500\,\mathrm{GeV}$ which couples exclusively to the *right-handed* τ leptons. To that end, start from the FeynRules SM again, browsing the model file SM.fr and extending it wherever necessary. Particularly, you'll have to add a new piece to the Lagrangian in the end, which contains the new interaction in correct FeynRules syntax. After making the required statements to introduce the new field and parameter, peek into the LYukawa piece in order to get an idea how the solution might look like.

When the model is correctly loaded in your notebook (you'd always start a new Mathematica kernel before loading new models in order to avoid conflicts with previously loaded models) and

the interface call runs through, use WHIZARD to test whether the Z' is really there. To that end, forgetting about realistic notions of technically feasible future collider options in the real world for the moment, think about the simplest process and energy setup where this should immediately become obvious in the WHIZARD output. Finally, think about actual e^+e^- colliders planned for the near future: What would be the final state and observable of choice to discover our new particle? What would be the minimum collision energy required? Produce a "smoking gun" discovery plot where the presence of the Z' is immediately visible to the plain eye.

5.3 Including the loop-induced ggH coupling: Effective operators and known limitations

By the end of this section, we will arrive at the current limitations of the FeynRules interface. The physics case to be considered is the coupling between gluons and the Higgs particle, which dominates the Higgs production at hadron colliders although it is purely loop-induced and hence a priori not present at tree level tools such as WHIZARD. However, the effect can still be incorporated at tree level using the language of effective operators. These operators are pretty handy for our purposes, because they can simply be added to the Lagrangian in order to derive tree level Feynman rules and study the phenomenological implications of loop effects: A superb application for our FeynRules–WHIZARD machinery!

Technically, in order to find these operators from the underlying theory including higher orders, one has to *integrate out* all the heavy particles running only in the loop: in our case, there is a top quark loop which induces the largest effect because of the huge top Yukawa coupling (if this topic is new to you: try to draw the Feynman diagram). As you can learn from more or less any of the more recent LHC physics review articles or textbooks, after electroweak symmetry breaking the resulting operator in this case reads

$$HG^a_{\mu\nu}G^{a\mu\nu} \tag{1}$$

with the physical Higgs field H, the QCD field strength $G^a_{\mu\nu}$ and an implicit sum over the gluon color indices a. Note that this operator is of mass dimension $d = 5$, and therefore gets normalized in the Lagrangian by a dimensionful constant prefactor

$$-\frac{\alpha_s}{8\pi v}\tau\left[1 + (1-\tau)\arcsin^2\left(\sqrt{1/\tau}\right)\right], \qquad \tau \equiv \frac{4m_t^2}{m_H^2} \tag{2}$$

with the Higgs vev v in the denominator. Your task is to add this effective operator to the FeynRules SM, starting again from a plain SM.fr. The way the gauge sector of the SM is implemented in LGauge might help here.

Once you have FeynRules load the model without errors, try the WHIZARD interface. Inspect the messages, particularly the account of the processed vertices: You will encounter something like

```
WARNING: unidentified vertex of arity 4 (spin structure: SVVV), skipping...
Skipped vertex: H , g , g , g
```

```
{{H,4},{G,1},{G,2},{G,3}}
Vertices of arity > 4 are not implemented yet, skipping vertex....
Skipped vertex: H , g , g , g , g
{{H,5},{G,1},{G,2},{G,3},{G,4}}
```

The meaning of this output is twofold: first, the quartic vertex $ggg H$ which comes from the non-abelian part of the field strengths in the operator has obviously been skipped on the grounds that the respective spin structure `SVVV` (indicating a scalar plus three vectors) could not be identified. This is of course not a limitation of the `FeynRules` side of the interface, but comes from the limited set of Lorentz structures implemented into the matrix element generator on the `WHIZARD` side: Lorentz structures which are not hard-coded in the matrix element generator cannot be supported within `WHIZARD`. For a quite similar reason, the interface also drops the $gggg H$ vertex which is also generated by the operator, simply because there is not a single 5-point vertex structure supported by `WHIZARD` so far. At the moment, this is the main bottleneck of the interface, because particularly in effective theories encoding beyond-the-SM effects one typically encounters operators of higher mass dimension $d > 4$ with respective new Lorentz structures; only a small subset of these has been made available to `WHIZARD` so far[5]. However, note that there has been *no* warning referring to the $gg H$ 3-point vertex: indeed, this is one of the few $d = 5$ interaction structures which are already available. The reason for this selection is clear: it is one of the key ingredients of a plethora of Higgs-related hadron collider studies, and of course every serious Monte Carlo tool should incorporate it in one or the other way. This is also why there is a dedicated model `SM_Higgs` shipped with the `WHIZARD` package which conveys all the non-tree level Higgs–vector interactions $gg H$, $\gamma\gamma H$ and $\gamma Z H$.

In any case, once you injected your model into the `WHIZARD` installation, you can again test it with relevant processes such as the total $H \to gg$ decay width or Higgs production in proton–proton collisions.

5.4 Self-study

For a proper phenomenological study, complete your model by also including the $\gamma\gamma H$ coupling. Of course, the effective operator generating this coupling is the same as the one in Eq. (1) with the QCD field strength replaced by the QED one, $G^a_{\mu\nu} \to A_{\mu\nu}$. consequently, the prefactor of the top quark contribution is also the same, with the sole replacement $\alpha_s \to \alpha_{\text{QED}}$ in Eq. (2). However, the photon couples to electromagnetic charge, so there are more loop contributions to the vertex in excess of the top loop: can you draw them? In any case, the prefactor of the additional contributions (to be added to the top quark piece) reads

$$\frac{\alpha_{\text{QED}}}{48\pi v}\left[2 + 3\tau' + 3\tau'\,(2-\tau')\arcsin^2\left(\sqrt{1/\tau'}\right)\right]\,, \qquad \tau' \equiv \frac{4m_W^2}{m_H^2} \tag{3}$$

with the W boson mass m_W. Once you got it working, produce a plot of the relevant LHC observable, and also compare your model to the one included in the package, `SM_Higgs`.

[5] You can find an exhaustive list of all Lorentz structures supported by `WHIZARD` in the manual. Although there is the plan to support general Lorentz structures at a vertex in the matrix element generator in future versions, no dependable release date can be quoted yet.

6 The $t\bar{t}$ threshold at future lepton colliders

One of the most intriguing physics cases for the various layouts of future high energy e^+e^- colliders is a precise measurement of the $t\bar{t}$ production cross section close to the threshold energy, $\sqrt{s} \sim 2m_t$. In this kinematic region, large effects from the nonrelativistic QCD potential binding the tops to each other will be visible in the total cross section, allowing for a very precise determination of m_t and α_s by a threshold scan. From one of the imminent release versions onwards ($> 2.2.2$), `WHIZARD` will provide a dedicated model `SM_tt_threshold` to incorporate and study this effect up to the next-to-leading log (NLL) order in the nonrelativistic QCD expansion. For the moment, if you want to play around with the model right now, there is a dedicated pre-release version of the `WHIZARD` package which contains the model in its current development state[6]. You can obtain it here ($\sim 23\,\mathrm{MB}$) or from one of the USB sticks going around during the tutorial. Follow the instructions in Sec. 7 to compile the source.

6.1 Leading order and phase space

Before actually going on to the new model with the threshold enhancement, let's start by examining the $t\bar{t}$ production threshold at e^+e^- colliders at leading order (i. e. keep `model = SM` in your `SINDARIN` script for the moment)[7]. The great advantage of such a lepton collider is that you can precisely adjust the partonic center-of-mass energy $\sqrt{s}$ of the process (in principle; of course there are technical details and complications which we will happily ignore for the time being). This means that you can tune your machine to scan $\sqrt{s}$ over the expected threshold region: use `WHIZARD` to produce such a scan over the $t\bar{t}$ threshold. Start with the most naive process, namely the $2 \to 2$ on-shell production $e^+e^- \to t\bar{t}$. Produce a plot to inspect the shape of the total cross section $\sigma(\sqrt{s})$. The crucial `SINDARIN` snippets you'll need for this are

```
process eett = "e+", "e-" => t, tbar                ! the 2->2 process
...
plot thresh ( x_min = Emin x_max = Emax )           ! initialize plot
scan sqrts = (Emin => Emax /+ Estep) {              ! parameter scan syntax
  beams = e1, E1
  integrate (eett)
  record thresh (sqrts, integral(eett))             ! record the plot points
}
compile_analysis
```

[6] Note that the effective parametrization of resonant effects in the threshold region is already fully functional and validated up to NLL; what is still missing is a properly matched transition from the threshold region into the $t\bar{t}$ continuum production region.

[7] Skip this section if you're already familiar with the topic itself and only want to see how `WHIZARD` performs.

where it is remains for you to define the interval (`Emin`, `Emax`) as well as the step size for the scan: around what `sqrts` value would you expect the $t\bar{t}$ threshold? Which parameter of the `SM` model decides whether you are "close" to the threshold, or "far away"? Once you settled for a scan setup and ran `WHIZARD`, what do you observe in the resulting plot?

Of course, you will never observe actual on-shell tops in a detector, they are just an intermediate resonance which you might or might not be able to reconstruct from the decay products. The naive approach to including this would be to augment our $2 \to 2$ process by the dominating top decay process $t \to bW$. However, this will still keep the tops on-shell, which turns out to be an invalid approximation particularly in the threshold region: you'll have to find a more sophisticated formulation of the process, which includes the tops only as intermediate resonances, not as external particles. What is the corresponding $2 \to 4$ process of interest? Which cuts are appropriate to enhance the signal region (i. e. the kinematic region where the top resonances dominate the matrix element), and how large are the effects from non-resonant irreducible backgrounds (i. e. those pieces of the full matrix element which do not contain the top resonances)? Produce a similar plot to the one you already have from the $2 \to 2$ process, and compare the shapes of $\sigma(\sqrt{s})$. What is the qualitative difference between these two shapes, and where does it come from?

6.2 Engaging the model: `SM_tt_threshold`

Now let's switch on the new model `SM_tt_threshold`. But before running it, have a look at the model parameters: the respective snippet of the model file reads[8]

```
parameter m1S    = 172.0        # t-quark m1S mass
parameter wtop   = 1.5          # t-quark width
parameter nloop  = 1            # vNRQCD order (0/1: LL/NLL)
parameter sh     = 1.0          # hard scale: mu_h = m1S * sh
parameter sf     = 1.0          # soft scale: mu_s = m1s * sf * v*(sqrts)
...
external mtpole                 # depends on nloop and sqrts
```

Comparing to the SM, note that there is no top mass parameter `mtop` any more: if you browse the respective literature, you will learn that in the special case of the top quark there are some intricacies in defining its "mass" in the first place, and strictly speaking a value for m_t is worthless without the definition it refers to. However, there is some implicit consent among Monte Carlo developers and experimental physicists concerning the "standard" definition: it is simply the square root of the real part of the constant which is plugged into the top propagator, hence the illustrative term "Monte Carlo mass" (termed `mtop` in all `WHIZARD` models except for

[8] Again, if you are familiar with these parameters, feel free to go on to the next paragraph following the explanation of the parameters.

this, while the imaginary part is then fixed by the top width `wtop`). At leading order it coincides with the so-called pole mass, which is defined as the value of the squared top 4-momentum p_t^2 where the matrix element has its maximum. Of course, there is a respective parameter `mtpole` also in this model, but it is tagged `external` which means that you're not free to vary it by hand. In short, the reason is that there is an intrinsic lower bound on the theoretical uncertainty of its definition, which is set by the QCD confinement scale $\Lambda_{\mathrm{QCD}} \sim 200\,\mathrm{MeV}$. However, the experimental precision at a lepton collider is expected to be much smaller, of order $\sim 50\,\mathrm{MeV}$ or even better, so the theoretical calculation should rely on an input parameter which is at least as well defined as that. A very useful quantity for this is the so-called $1S$ mass, because the uncertainties related to Λ_{QCD} cancel in its definition. It encodes the binding energy of the $t\bar{t}$ system inside the QCD potential at threshold—the ground state of what would be a "top meson" if it didn't decay so fast, hence the term $1S$ from meson spectroscopy. The corresponding model parameter is `m1S`, from which the program then dynamically computes `mtpole` at runtime, depending on the value of `sqrts` and the non-relativistic QCD (NRQCD) loop order `nloop`. The two remaining parameters are `sh` and `sf`, allowing you to set the hard renormalization scale μ_h resp. the soft scale μ_s which enter the threshold resummation calculation. Varying them gives you an estimate of the theoretical uncertainty entering the simulation.

After this lengthy introduction, employ the model to repeat the threshold scan over the $2 \to 4$ process from Sec. 6.1, including the NRQCD threshold corrections this time. As already mentioned, one of the drawbacks of the current model implementation is that it still lacks a sensible matching to the continuum production region, which means that you should not go too far above the threshold region in order to avoid running into unphysical effects: to be on the safe side, don't go more than 3–4 top widths above the threshold with $\sqrt{s}$. Compare the plot once more to the previous ones and look at the qualitative and quantitative changes of the cross section shape as a function of $\sqrt{s}$.

6.3 Self-study

If you still have some time left, try to estimate the theoretical uncertainty by simultaneously varying the parameters `sh` and `sf` (typically this is done in the range $0.5 \cdots 2$, but there is nothing fundamentally physical behind this choice; it's just pretty common). If you manage to do this both for LL and NLL precision, compare them to each other: would you say that the uncertainty estimate is reasonable?

7 Building your own WHIZARD installation

For several sections of this tutorial, you will need to have your own compiled installation of one or the other `WHIZARD` version on your machine. Here is how to achieve this—always supposed your system fulfills all necessary prerequisites; but don't bother for the moment, just try it. You'll learn from the console output when/if there are problems. If you don't manage to build the source on your system, you can always ask one of the tutorial instructors to help you. As

a last resort, consider compiling the package inside the VM where everything is well prepared to serve all WHIZARD needs.

There are several versions packed into tarballs which you can download using the links provided in the respective sections of the tutorial. Otherwise, you can ask one of the tutorial instructors to give you a USB stick with the tarballs. Once you obtained the correct tarball for the exercise that interests you, do[9]

```
tar -xvf <path_to_tarball>
mkdir build inst
cd build
<path_to_unpacked_source>/configure --prefix=$PWD/../inst
make install
. ../inst/bin/whizard-setup.sh
```

After that, the default `whizard` command in your `$PATH` will the one you just compiled, so you're ready to start with the exercise.

8 Setting up a shared folder between the VM and your host system

If you have not done so already, execute the `initial.sh` script inside the VM:

```
. ~/scripts/initial.sh
```

This will install the `VBoxGuestAdditions` needed to share a folder with your host system. Then you need to create the folder to be shared somewhere in your host system, say

```
mkdir ~/vmshare
```

Now, from inside the VM go to "Devices→Shared Folders Settings..." and add a new shared folder definition. In the following pop-up window, choose the folder you just created on your host system as "Folder Path" and `share` as "Folder Name". Finally, execute the `mount-shared.sh` script:

```
. ~/scripts/mount-shared.sh
```

The shared folder of your host system should now be mounted inside the VM at `~/host`.

References

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

[9] Note that the following procedure will overwrite any previously compiled `WHIZARD` compilation under the given installation target specified by the `--prefix` flag; be sure to avoid any unwanted chaos. Specifically, don't forget to set the `--prefix` flag when compiling inside the VM, otherwise you'll overwrite the existing executable located in `/usr/local/bin`.