Contents

Introductory Remarks

The material contained here will cover two of the tutorial sessions: the first one on Monte Carlo generators in general, and the second one on a BSM signal and its backgrounds. We will also introduce how to obtain generator output in the `HepMC` format and how to analyse the generated events using the `Rivet` framework.

In the first session we expect people to sign up for getting to know one particular generator. In particular, there should be **an about equal number of persons per generator!** In later tutorials people should arrange into groups of four people each to collaboratively work on the topics covered by the other tutorials, such that in each group experience with one particular generator is present.

In later tutorials we will use pre-generated event samples obtained by the four event generators covered here. This has become necessary for time constraints, as running the simulation to obtain decent statistics would have taken way too long for the time being available for the tutorials. If you are interested in the precise settings to obtain the samples, just ask one of us.

The event sample files are contained on a NFS network directory to be mounted within the virtual machine. For performance reasons, we have set up several NFS servers, intended to be shared equally amongst the students. For this to work, little paper snippets have been prepared for everyone mentioning a particular server name. Given this name, issue the command

```
$ mount-samples SERVER
```

in the virtual machine, replacing `SERVER` by the server name on the snippet and entering `hamburg0312` when asked for a password.

Using Rivet to Analyse Generator Predictions

1. Introduction

Rivet is a small framework for analysing simulated collider events in HepMC format. As well as providing the infrastructure for such analyses, it provides a set of efficient observable calculators and a large collection of standard analyses from a variety of colliders and experiments. Rivet is designed to be an efficient and simple-to-use system for generator validation, tuning, and regression testing.

2. Getting started

Rivet is pre-installed in your virtual machine and should be usable right away. To test that it's working properly, you can list all built-in analyses and display details about one of them (*Hint: tab completion should work for these commands!*):

```
$ rivet --list-analyses
ALEPH_1991_S2435284
ALEPH_1996_S3196992
ALEPH_1996_S3486095
[...]
UA5_1982_S875503
UA5_1986_S1583476
UA5_1987_S1640666
UA5_1988_S1867512
UA5_1989_S1926373

$ rivet --show-analyses ATLAS_2010_S8894728

ATLAS_2010_S8894728
===================

Track-based underlying event at 900 GeV and 7 TeV in ATLAS

Status: VALIDATED

Spires ID: 8894728
Spires URL:  http://www.slac.stanford.edu/spires/find/hep/www?rawcmd=key+8894728
HepData URL: http://hepdata.cedar.ac.uk/view/irn8894728
Experiment: ATLAS (LHC)
Year of publication: 2010
[...]
```

3. Running analyses on top of MC events

The main interface to Rivet is the `rivet` command. We will first demonstrate how to use this to analyse HepMC events from a file in the HepMC format. Let's assume we want to run the analysis `MC_ZJETS` on an existing file of $Z \to e^+e^-$ events which have been pre-generated for this tutorial. Due to space and network constraints, the pre-generated event samples are gzipped and thus have to be decompressed on-the-fly using `zcat`. They can simply be piped into Rivet:

```
$ zcat /mcschool/events/background-sherpa-zee/file.1.hepmc.gz | rivet \
    -a MC_ZJETS -n 10000 -H Zee.aida
```

Go ahead and run this command, it should write out histograms in the file `Zee.aida`. We'll see later how to plot them. The switch `-n` specifies how many events to read from the file; it can be left out to read the full file. All options for the `rivet` command can be displayed by running

```
$ rivet --help
```

If you want to run Rivet "live" on events generated by a Monte-Carlo, we recommend using a named pipe (or 'FIFO') so that your events don't create a huge file that takes all your disk space. The idea is that the generator will push events into what looks like a file, and Rivet will read from the same 'file'. In fact, though, the 'file' is a disguised pipe between the two processes, so no slow filesystem access needs to take place, and the system will automatically balance the data flow between the writing and reading processes: the generator will only write more event data when Rivet has read that currently available in the FIFO buffer. All this is completely transparent to the user: good old Unix! Here's how you do it (with a fictional generator command, as an example; see the generator specific instructions for writing HepMC to a file in the next sections):

```
$ mkfifo hepmc.fifo
$ my-generator --num-events=10000 --hepmc-output=hepmc.fifo &
$ rivet -a ANALYSIS_NAME hepmc.fifo
```

The backgrounding of the generator process is important: the generator will wait until the `hepmc.fifo` pipe is being read by Rivet, so unless it is backgrounded you will never get the terminal focus back to run `rivet`! Alternatively, you can also start the generator and Rivet in two different shells, albeit in the same working directory.

4. Plotting the analysis output

By default, Rivet outputs its histograms in the AIDA XML format, which is not particularly easy to read or plot directly. Rivet thus comes with the command `rivet-mkhtml` for comparing and plotting AIDA files. It produces nice comparison plots of publication quality from the AIDA format text files and will automatically create a webpage containing all those plots. Example:

```
$ rivet-mkhtml -o plots/ Zee.aida:'Title=$Z\to e^{+}e^{-}$'
```

Here we used the histogram file created by the Rivet run above. You can specify multiple AIDA files at the same time (e.g. from analysing events of different generators), which will result in comparison plots for their common histograms. A webpage displaying the plots will be create in the directory specified by `-o` and can be viewed using `firefox plots/index.html`. Run `rivet-mkhtml --help` to find out about all features and options.

5. Writing an analysis

Here we are going to write a new analysis for use with Rivet. This is done "stand-alone", i.e. you don't have to modify the code of Rivet itself: in fact, you can follow these instructions using a system install of Rivet to which you have no write permissions.

All analysis routines are implemented as sub-classes of the Rivet "Analysis" class: pretty much all the magic that binds the analysis object into the Rivet system is handled in this base class, meaning that your code can really concentrate on implementing the physics goals of the analysis.

5.1. An analysis skeleton

We have prepared an analysis skeleton which you will extend later in the tutorial. You can find it in:

```
/mcschool/analysis/MY_ANALYSIS.cc
```

Please copy that file to a working directory.

For simplicity, Rivet analysis classes are usually written in just one `.cc` file, i.e. no header declaration. This is because classes are almost always not inherited from, and all that the Rivet system needs to know is that it can be treated as an `Analysis*` pointer: avoiding header files makes everything more compact and removes a source of errors and annoyance.

An analysis has the following components:

- a no-argument constructor;
- three analysis event loop methods: `init`, `analyze` and `finalize`;
- a minimal hook into the plugin system

It is also possible to add some metadata methods which describe the analysis, references to publications, experiment, etc. as class methods directly. Analyses may also contain member variables for the analysis: event weight counters and histograms are the most common of these.
The three event loop methods are used for the following:

- `init`: book histograms and define projections (we'll get to these later)
- `analyze`: select particles, filter according to cuts, loop over combinations, construct observables, fill histograms. This is where the per-event aspect of the analysis algorithm goes.
- `finalize`: normalize/scale/divide histograms, tuples, etc.

This probably looks similar to every analysis system you've ever used, so hopefully you're not worried about Rivet being weird or difficult to learn ;)
Rivet provides implementations of many calculational tools, called "projections". These are just observable calculator objects with a silly name, so don't get worried. They implement particle selections, cuts, jet algorithms, missing E_T and more. In our example, a projection has been registered under the name `"electrons"` in the `init` method as referring to a projection of type "IdentifiedFinalState" — a calculator which provides a list of particles of a certain flavour (here: electrons) and with certain basic cuts applied (here: $p_\perp > 25$ GeV and $|\eta| < 5.0$). This is done via the `addProjection` method:

```
IdentifiedFinalState electrons(-5.0, 5.0, 25.0*GeV);
electrons.acceptIdPair(ELECTRON);
addProjection(electrons, "electrons");
```

The projections are then used in each event by calling the analysis' `applyProjection(event)` method. This will return a const reference to the completed projection object and takes the type of the reference as a template argument, e.g.

```
const IdentifiedFinalState& electrons =
  applyProjection<IdentifiedFinalState>(event, "electrons");
```

Note that a) you don't have to manage the memory yourself, and b) polymorphism via the reference is both allowed and encouraged. If b) means nothing to you, don't worry... we just want to reassure C++ fiends who might think we're cramping their OO style!
You'll find more details and examples for adding your own projections in Secs. 5.4 and 6 below.

5.2. Compiling and linking

To use your new analysis, you need to build it into a Rivet analysis plugin library. You can do this manually, but to make life easier there is a helper script, used as follows:

```
$ rivet-buildplugin MY_ANALYSIS.cc
```

The default name of the resulting library is `RivetAnalysis.so`. Note that the name of the library has to start with the word `Rivet` or it will not get loaded at runtime.

5.3. Running

You can now use your newly built analysis. Use the `--analysis-path` flag such that it contains the directory where the `RivetAnalysis.so` shared library file was created and test that it is found with the `rivet` command:

```
$ ls
RivetAnalaysis.so MY_ANALYSIS.cc
$ rivet --analysis-path=$PWD --list-analyses
[...]
MY_ANALYSIS
```

5.4. Filling the analysis

Major idea: projections These are where the computational meat of Rivet resides. They are just observable calculators: given an `Event` object, they project out physical observables. They are registered with a name in the `init` method and can then be applied to the current event, also by name, in the `analyze` method. They can then be queried about the things they have computed, cf. the two simple examples in `MY_ANALYSIS.cc` Check the code documentation on the Rivet website for a list of all projections and their abilities: `http://projects.hepforge.org/rivet/code/dev/hierarchy.html`. Projections you might need for this tutorial are e.g. `IdentifiedFinalState`, `MissingMomentum` and `FastJets`.

Example: Getting final state particles To iterate over the identified particles as defined by the `IdentifiedFinalState` projection in the example, you could use code like the following:

```
ParticleVector particles =
        applyProjection<IdentifiedFinalState>(event, "electrons").particlesByPt();
MSG_INFO("Total electron multi = " << particles.size());
foreach (const Particle& p, particles) {
  const double eta = p.momentum().eta();
  MSG_INFO("Particle eta = " << eta);
}
```

Physics vectors Objects that you get out of projections, like particles or jets, typically have a `momentum()` method which returns a `FourMomentum`. It has typical methods like `eta()`, `pT()`, `phi()`, `rapidity()`, `E()`, `px()`, `mass()`, . . . , all of which are documented in the code documentation.

Histogram booking Rivet has `Histogram1D` and `Profile1D` histograms similar to the `TH1D` and `TProfile` histograms in ROOT. They can be booked via helper methods like `bookHistogram1D` and `bookProfile1D` as exemplified in `MY_ANALYSIS`.

6. First modifications

As preparation for later, you can now add two more features to the analysis:

- Analyse the distribution of missing E_T
  (Hint: Use the `MissingMomentum` projection and its `vectorEt()` method.)
- Analyse the jet multiplicity distribution, and the $p_\perp$ and η of the 2 leading jets
  (Hint: Use the `FastJets` projection.)

Try running the modified analysis with the example event file as in Secs. 3 and 4 to see whether it works as expected. The plot settings for all observables defined in your analysis can be modified in a separate text file, an example of which you can copy from `/mcschool/analysis/MY_ANALYSIS.plot`. For `rivet-mkhtml` to pick up the settings from that file, you need to set the environment variable `export RIVET_ANALYSIS_PATH=$PWD` to the directory containing the `.plot` file beforehand. The syntax of that file is hopefully self-explanatory from the example, and available switches are documented in `http://rivet.hepforge.org/make-plots.html`.

Further Information and Resources

This was only a brief introduction to get you started with running and writing a simple analysis. If you need anything more complicated, here are a few pointers with more information:

- Rivet online documentation
  `http://rivet.hepforge.org/trac/wiki`
- Code documentation
  `http://rivet.hepforge.org/code/dev/hierarchy.html`
- Code repository of standard analyses (useful as examples, e.g. the `MC_*` analyses)
  `http://rivet.hepforge.org/trac/browser/trunk/src/Analyses`

- PDF manual
  `/mcschool/opt/share/Rivet/rivet-manual.pdf`
- Standard analyses list
  `http://rivet.hepforge.org/analyses`
- Contact the Rivet authors
  `mailto:rivet@projects.hepforge.org`

Herwig++ Tutorial

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

Herwig++ is a multi-purpose event generator for lepton, lepton-hadron and hadron-hadron collider physics. The focus is on physics beyond the standard model, simulating hard processes at NLO accuracy and parton showers incorporating QCD coherence through angular ordered showers.

1. Preparation

To speed up the setup we have pre-installed Herwig++ for the tutorials in `/mcschool/opt`. For the first steps please copy one of the input files shipped with Herwig++ to your working directory (we'll explain their roles in Sec. 3 below):

```
$ cd /path/to/your/working/directory
$ cp /mcschool/opt/share/Herwig++/LHC.in .
```

2. Simple LHC Events

As a first step, we will generate 100 LHC Drell-Yan events in the default setup that comes with the distribution:

```
$ Herwig++ read LHC.in
$ Herwig++ run LHC.run -N100 -d1
```

The commands are explained in the next section. We have actually passed a debug flag, `-d1`, to explicitly print out some of the generated events. Looking at the file `LHC.log`, you should see the detailed record of the first 10 events of this *run*. Each *event* is made up of individual *steps* that reflect the treatment of the event as it passes through the various stages of the generator (hard subprocess, parton shower, hadronization and decays).

Every *particle* in a step has an entry like

```
16    g     21 [13] (42,43)    {+6,-5}
                 -1.040    -2.805   177.756   177.783     0.750
```

The first line contains `16`, the particle's label in this event; `g 21`, the particle's name and PDG code; `[13]`, the label(s) of parent particle(s); `(42,43)` the label(s) of child particle(s); and `{+6,-5}`, the colour structure: this particle is connected via colour lines 5 and 6 to other coloured particles. Sometimes you'll also see something like `7v` or `2^`; they signify that the current particle is a clone of particle 7 below / 2 above. The second line shows p_x, p_y, p_z, E and $\pm\sqrt{|E^2 - p^2|}$.

Note that everybody has generated the exact same events (go and compare!), with exactly the same momenta. Adding 10 of these runs together will *not* be equivalent to running 1000 events! To make statistically independent runs, you need to specify a random seed, either with `$ Herwig++ run LHC.run -N100 -seed 123456` or, as we'll see now, in the `LHC.in` file.

3. Input Files

Any ThePEG-based generator like Herwig++ is controlled mainly through input files (.in files). They create a *Repository* of component objects (each one is a C++ class of its own), arranged in a directory structure, which is **not** related to the file system on the hard disk. The input files then assemble the repository objects, which provide interfaces for parameter settings, into an event generator. Herwig++ already comes with a pre-prepared default setup[1]. As a user, you will only need to write a file with a few lines (like `LHC.in`) for your own parameter modifications. The next few sections will go through this.

[1]in `/mcschool/opt/share/Herwig++/defaults`

The first command we ran (`Herwig++ read LHC.in`) takes the default repository provided with the installation[2], and reads in the additional instructions from `LHC.in` to modify the repository accordingly. A complete setup for a generator run will now be saved to disk in a `.run` file, for use with a command like `Herwig++ run LHC.run -N100`. The run can also be started directly from the `LHC.in` file, which is especially useful for batch jobs or parameter scans.

Writing new .in files is the main way of interacting with Herwig++. Have a look at the other examples we have provided for LEP, Tevatron, or ILC (`LEP.in, TVT.in` and `ILC.in`) and see if you can understand the differences:

```
$ cp -u /mcschool/opt/share/Herwig++/???.in .
```

The two most useful repository commands are `create`, which registers a C++ object with the repository under a chosen name, and `set`, which is used to modify parameters of an object. Note that all this can be done without recompiling any code!

Take your time to play with the options in the example files. Some suggestions for things you can try:

1. Run 100 Tevatron events.
2. Control a run directly from the .in file. Be careful with the number of events you generate, the default is 10^7, and we don't have that much time today!
3. Compare the Drell-Yan cross sections for Tevatron and LHC. The cross sections are written to `TVT.out` and `LHC.out`, respectively.

4. Analysis handlers

There is an easier way to analyse the generated events than looking at the `.log` file. `ThePEG` provides the option to attach multiple *analysis handlers* to a generator object. Every analysis handler initializes itself before a run (*e.g.* to book histograms), analyses each event in turn (fill histograms) and then runs some finalization code at the end of the run (output histograms).

As part of the default setup, one analysis handler has always been running already. The *BasicConsistency* handler does what its name promises: checking for charge and momentum conservation.

4.1. Graphviz plot

Before we go on to a physics analysis, let's briefly look at a useful handler that allows us to visualize the internal structure of an event within Herwig++. Enable the *GraphvizPlot* analysis for LHC (the line in `LHC.in` which mentions `/Herwig/Analysis/Plot`) and run one LHC event. *Plot* should have produced a file `LHC-Plot-1.dot`, which contains the description of a directed graph for the generated event. The `graphviz` package will plot the graph for us:

```
$ dot -Tsvg LHC-Plot-1.dot > LHC-plot.svg
```

Have a look with `$ inkscape LHC-plot.svg`

1. Identify the Drell-Yan process. Has there been initial state radiation?
2. Keep track of the incoming protons and proton remnants. Did only one $2 \to 2$ scattering take place?

It is important to note that these plots only reflect the internal event structure in the generator. Many internal lines do *not* have a physical significance!

[2]The default repository can also be re-created by the user, by copying over the files from `/mcschool/opt/share/Herwig++/defaults`, and running `$ Herwig++ init`. This reads the file `HerwigDefaults.in` (which in turn includes most of the other .in files) to prepare a default *LEPGenerator* and *LHCGenerator* object. This default setup is now stored in `HerwigDefaults.rpo`. A regular user does not really need to run `init` at all, everything can be modified at the `read` stage.

5. Writing HepMC Event Files

As mentioned in the preceding section, analysis handlers can be used to directly analyse the events, i.e. to write out histograms etc. However, we will use Rivet for our BSM analysis project during the next days, which is a *generator-independent* analysis framework. As Rivet reads the Monte Carlo data in the HepMC format, you will learn in this section how to produce HepMC files with Herwig++.

Herwig++ contains a special analysis handler, `/Herwig/Analysis/HepMCFile`, which writes the generated events to a HepMC file. Enabling it is as easy as this:

```
insert LHCGenerator:AnalysisHandlers 0 /Herwig/Analysis/HepMCFile
```

You can find this line already included in the analysis section of `LHC.in`, merely uncomment it. A filename can be specified using the `Filename` interface to `HepMCFile`. You should also tell `HepMCFile` how many events are to be written. This is done with the `PrintEvent` interface. But beware: HepMC files can get really large!

5.1. Connecting Herwig++ to Rivet

— *You can skip this section, if you want to, as this is also going to be covered in the* Rivet *tutorial.* —
—*We just provide this as a reference here.*—

To avoid having to store large HepMC files on the hard disk just to analyse them later on with Rivet, a *fifo* file useful. In that way, for instance, the HepMC stream can directly be piped into Rivet. All you have to do is e.g.

```
$ Herwig++ read LHC.in     // where LHC.in contains: "set HepMCFile:Filename fifo.hepmc"
$ mkfifo fifo.hepmc
$ Herwig++ run LHC.run -N1000 &
$ rivet -a [...] -H histograms.aida fifo.hepmc
```

NB: It's absolutely crucial to start Herwig++ in the background by adding `&` at the end of the line. Otherwise Herwig++ will never start writing events because Rivet will never start reading from the fifo.

6. Changing Default Settings

Take a look at the default settings in `/mcschool/opt/share/Herwig++/defaults`, we have commented them extensively. Ask the tutors to explain parameters. Can you identify which four lines in `HerwigDefaults.in` control the hard subprocess, the parton shower, the hadronization and the decays?

6.1. Switching On or Off Simulation Steps

So far, we did look at completely generated events including parton showers, hadronization, decays of hadrons and multiple parton interactions. The first three of these steps may be switched off by setting the corresponding *step handler* interfaces of the event handler to `NULL`. Multiple parton interactions are switched off by setting the `MPIHandler` interface of the `ShowerHandler` to `NULL`. For the remainder of the tutorial we suggest to switch off multiple parton interactions, as this part of the simulation is very complex, taking too much time in running a reasonable number of events.

Add repository commands to your local `LHC.in` switching on or off successive steps and look at the effects by generating few events. The default settings are provided in `HerwigDefaults.in` and `Shower.in`. Take care of the directories, in which the different objects reside.

6.2. Changing the Hard Process

The default hard process for LHC is Drell-Yan vector boson production with leptonic decays. Edit `LHC.in` to replace the matrix element for vector boson production by the one for jet pair production for `LHCGenerator`'s default `SubProcessHandler`.

The relevant matrix element is contained in the default repository, `/Herwig/MatrixElements/MEQCD2to2`. Generate few events as for the default settings, using the flag `-d1` to see explicit event printouts.

6.3. Initial and final state radiation

Initial and final state QCD radiation may be switched off separately. Work out the relevant interface settings by looking at the comments and repository commands in `Shower.in`, for

1. switching off initial state radiation,
2. switching off final state radiation,

Notice the difference between the `newdef` command, referring to a default setting, and the `set` command changing defaults. Again, you can just add the respective `set` repository commands for setting interface values to your local `LHC.in` file.

If you want to look at some events in detail, it is reasonable to switch off hadronization, decays and multiple parton interactions when looking at the effect of the different settings (just to have clearer output).

6.4. Changing Decay Modes

The default setup for decay modes is contained in `Decays.in` and all files read in there. Out of these, individual decay modes can be switched of by statements like

```
set /Herwig/Particles/W+/W+->nu_tau,tau+;:OnOff Off
```

Note that the decay products are to be given in the order as specified in the decay input files.

6.5. Matrix Element Options

There are often also switches for the selection of a particular subprocess given with a matrix element. For W production, the relevant switch for e.g. the leptonic W–channel is
`set MEqq2W2ff:Process Leptons`.

7. VBF Higgs production with invisible Higgs decays

In the tutorials on Wednesday and Thursday the focus is on Higgs production via vector boson fusion (VBF) with invisible Higgs decays. For reasons of time, we will have to work on pre-generated samples. There will be a number of samples from different event generators, some of which contain the VBF signal, others containing background events to this process.

In particular, there is a sample of Herwig++ signal events generated with the input file located here:

```
/mcschool/opt/share/Herwig++/HiggsVBFPowheg.in
```

7.1. Hard processes at NLO accuracy

This file contains a setup to simulate the VBF Higgs process at next-to-leading order (NLO) QCD accuracy, using the POWHEG algorithm. Step through this file to see what ingredients are needed. Don't hesitate to ask us in case of questions.

Several processes are available in Herwig++ at NLO QCD accuracy, for example Higgs production in association with a Z boson. Have a look at `LHC-Powheg.in` (in the same directory) for other examples.

7.2. SUSY Higgs decay

Although this all looks much like Standard-Model Higgs production (and in fact it is), the idea of Wednesday's and Thursday's project is to have the light neutral, i.e. standard-model-like Higgs (in a 2-Higgs-doublet model) decay to stable supersymmetric particles. The Higgs then only appears in the detector as missing transverse energy and momentum.

The SUSY spectrum file defining the model and parameter point can be found at
`/mcschool/SLHA/bpoint_w1_slha.in`. It contains the mass spectrum of the SUSY particles and the decay modes. In particular, the branching ratio of the Higgs decay to a pair of neutralinos is (artificially) set to 100% there.

7.3. Try yourself

If you want to generate events yourself using `HiggsVBFPowheg.in`, please note:

- Be sure to also copy `MSSM.in` from the same directory to your working directory. This file is needed for MSSM runs. It generates non-standard-model particles, sets up new decay modes etc.
- While the input file is read, Herwig++ complains a lot about incomplete decay tables in the provided `.slha` file. In this case this is nothing to worry about since the relevant Higgs decay to neutralinos is included.
- The initialization process takes some time with this NLO example. Alternatively you might try generating the same process at leading order in QCD. A complete setup to do so is provided in `/mcschool/opt/share/Herwig++/HiggsVBF.in` .

Further Information and Resources

Thanks for trying Herwig++!

If you have any questions later on, please email us at `herwig@projects.hepforge.org` or have a look at `http://projects.hepforge.org/herwig/`, where many how-tos can be found, and we'll add more on request. For detailed documentation refer to our manual, `arxiv:0803.0883`.

Simon Plätzer, Stefan Prestel, Christian Röhr, Frank Siegert, Christian Speckner March, 12th-15th, 2012

Pythia 8 Tutorial

The objective of this exercise is to teach you the basics of how to use the PYTHIA 8.1 event generator to study multi-jet backgrounds. As you become more familiar you will better understand the tools at your disposal, and can develop your own style to use them. Within this first exercise it is not possible to describe the physics models used in the program; for this we refer to the PYTHIA 8.1 brief introduction [1], to the full PYTHIA 6.4 physics description [2] (and all the further references found in them), and [5] concerning matrix element merging.

PYTHIA 8 is, by today's standards, a small package. As such, it should be noted that PYTHIA8 includes a selection $2 \to 1$ and $2 \to 2$ processes, as well as a limited variety of $2 \to 3$ processes, but does not contain a general matrix element generator. New processes, particularly for two or more additional jets, can be made available in form of Les Houches Event (LHE) files. This means that to estimate backgrounds with many jets, you can use a matrix element generator like e.g. MADGRAPH/MADEVENT to improve the PYTHIA8 description of well-separated jets [5].

1. The manual

For this school, a precompiled PYTHIA 8 version `pythia8160` (linked to `fastjet`, HEPMC and LHAPDF) is already available on the virtual machine image. You can start right away with experimenting. A comprehensive and useful online manual is available at

`http://home.thep.lu.se/~torbjorn/php8160/Welcome.php`

or if you open

`/mcschool/opt/share/Pythia8/pythia8160/htmldoc/Welcome.html`

A large set of example main programs can be found by typing `ls` in

`/mcschool/opt/share/Pythia8/pythia8160/examples`

Information on these examples can be found in the online manual. If you would still like to install PYTHIA 8 on your private machine, some instructions are given in Appendix B.

2. Simple LHC Events

When using PYTHIA, you are expected to write the main program yourself, for maximal flexibility and power. Several examples of such main programs are included with the code, to illustrate common tasks and help getting started. You will also see how the parameters of a run can be read in from a file, so that the main program can be kept fixed. Many of the provided main programs therefore allow you to create executables that can be used for different physics studies without recompilation, but potentially at the cost of some flexibility.

The focus of the next sessions will be on extracting signals from Standard Model backgrounds. So, to get to know the PYTHIA 8 syntax, we will generate one $\mathrm{pp} \to \mathrm{W} + \mathrm{jets}$ event at the LHC, using PYTHIA standalone to produce all jets.

Open a new file `mymain.cc` in the `examples` subdirectory with a text editor, e.g. Emacs. Then type the following lines (here with explanatory comments added):

```
// Headers and Namespaces.
#include "Pythia.h"      // Include Pythia headers.
using namespace Pythia8; // Let Pythia8:: be implicit.

int main() {             // Begin main program.

  // Set up generation.
  // Declare Pythia object
  Pythia pythia;
  // Initialise pythia on LHE file for qqbar-> W
  pythia.init("./w+_production_lhc_0.lhe");

  // Generate event(s).
  pythia.next();         // Generate an(other) event. Fill event record.

  return 0;
}                        // End main program with error-free return.
```

Next you need to edit the `Makefile` (the one in the `examples` subdirectory) so it knows what to do with `mymain.cc`. The line

```
# Create an executable for one of the normal test programs

main00 main01 main02 main03 ...  main09 main10 main10 \
```

together with the following three lines enumerate the main programs that do not need any external libraries. Edit the last of these lines to include also `mymain`:

```
 main40 mymain: \
```

Now you can compile and run your main program by typing

```
make mymain
./mymain.exe > mymain.out
```

You can then study `mymain.out`, especially the example of an event record. At this point you need to turn to Appendix A for a brief overview of the information stored in the event record.

An important part of the event record is that many copies of the same particle may exist, but only those with a positive status code are still present in the final state. For illustration, consider a quark q produced in the first initial state splitting of the pp $\to$ W hard interaction. Initially, this parton will have a positive status code. When a shower branching q $\to$ qg occurs later, the new q and g are added at the bottom of the then-current event record, but the old q is not removed. It is marked as decayed, however, by negating its status code. At any stage of the shower there is thus only one "current" copy of this quark. When you understand the basic principles, see if you can find several copies of the a parton produced in the hard interaction, and check the status codes to figure out why each new copy has been added. Also note how the mother/daughter indices tie together the various copies.

3. A first realistic analysis

We will now gradually expand the skeleton `mymain` program from above, towards what would be needed for a more realistic analysis setup.

- Generally, we wish to generate more than one event. To do this, introduce a loop around `pythia.next()` and `pythia.event.list()`, so the code now reads

  ```
  for (int iEvent = 0; iEvent < 5; ++iEvent) {
    pythia.next();
    pythia.event.list();
  }
  ```

  Hereafter, we will call this the **event loop**. The program will now generate and print 5 events; each call to `pythia.next()` resets the event record and fills it with a new event. Once you start generating many events, it might be convenient to remove the `pythia.event.list()` call. By default, Pythia 8 will still print a record of the very first event.

- To obtain statistics on the number of events generated of the different kinds, and the estimated cross sections, add

  ```
  pythia.statistics();
  ```

  just before the end of the program.

- During the run you may receive problem messages. These come in three kinds:

  - a *warning* is a minor problem that is automatically fixed by the program, at least approximately;
  - an *error* is a bigger problem, that is normally still automatically fixed by the program, by backing up and trying again;
  - an *abort* is such a major problem that the current event could not be completed; in such a rare case `pythia.next()` is `false` and the event should be skipped.

  Thus the user need only be on the lookout for aborts. During event generation, a problem message is printed only the first time it occurs. The above-mentioned `pythia.statistics()` will then tell you how many times each problem was encountered over the entire run.

- Looking at the `pythia.event.list()` listing for a few events at the beginning of each run is useful to make sure you are generating the right kind of events, at the right energies, etc. For real analyses, however, you need automated access to the event record. The Pythia event record provides many utilities to make this as simple and efficient as possible. To access all the particles in the event record, insert the following loop after `pythia.next()` (but fully enclosed by the **event loop**)

```
for (int i = 0; i < pythia.event.size(); ++i)
   cout << "i = " << i << ", id = " << pythia.event[i].id() << endl;
```

  which we will call the **particle loop**. Inside this loop, you can access the properties of each particle `pythia.event[i]`. For instance, the method `id()` returns the PDG identity code of a particle (see Appendix A). The `cout` statement, therefore, will give a list of the PDG code of every particle in the event record. All methods that give particle properties can be found following the "Particle properties" link in the section "Study output" in the left-hand menu in the manual, or in the file `/mcschool/opt/share/Pythia8/pythia8160/htmldoc/ParticleProperties.html`.

- If you are e.g. only interested in final state partons, the `isFinal()` and `isParton()` methods can be applied to the event record entry:

```
for (int i = 0; i < pythia.event.size(); ++i)
   if(pythia.event[i].isFinal() && pythia.event[i].isParton())
         cout << "i = " << i << ", id = " << pythia.event[i].id() << endl;
```

  This will only print the PDG code of final state gluons, (anti)quarks and diquarks.

- In addition to the particle properties in the event listing, there are also methods that return many derived quantities for a particle, such as transverse momentum (`pythia.event[i].pT()`). Use this method to find the transverse momentum of the W^+-boson after the evolution, i.e. the last W-boson in the event record (which is of course cheating a bit, since in an experimental environment, it is a lot more complicated to isolate W-candidates).

- We now want to generate more events, say 1000, to study the shape the W-p_T-spectrum. Inside PYTHIA is a very simple histogramming class, that can be used for rapid check/debug purposes. To book the histograms, insert before the **event loop**

```
Hist pTW("pT of W-boson", 100, 0., 100.);
```

  where the last three arguments are the number of bins, the lower edge and the upper edge of the histogram, respectively. As an exercise, fill this histogram for each event with the transverse momentum of the W-boson after the evolution. For this, initialise a variable before the **particle loop**, and find the p_T inside a **particle loop**:

```
double pT = 0.;
for (int i = pythia.event.size(); i > 0; --i)
   if( pythia.event[i].idAbs() == 24 ) {
         pT = pythia.event[i].pT();
         break;
   }
```

Then, before the end of the **event loop**, insert

```
pTW.fill(pT);
```

to fill the histogram. To arrive at a correctly normalised histogram, include

```
pTW *= pythia.info.sigmaGen() / pythia.info.nAccepted();
```

after the **event loop** and the `pythia.statistics()` call. In this way, the sum of the heights of all histogram bins will give the correct cross section, irrespectively of how many events you requested[3]. Finally, to print the histograms to the terminal, add a line like

```
cout << pTW;
```

For comparison with merged results, it might be useful to save the output of this run.

4. A first merged prediction

The main program we have constructed in the previous section still has a drawback: All radiation will be produced by PYTHIA 8. This will give reliable results for soft and collinear configurations, but less so for multiple hard, well-separated jets. To model both soft/collinear and well-separated jets at the same time, we need to include matrix element calculations – which describe the production of multiple hard jets nicely – into the jet evolution of the parton shower. This can be done by supplying LHE files to PYTHIA 8, which will then internally be processed to perform a smooth transition from n-jet to $n+1$-jet events [5]. Several main programs illustrating matrix element + parton shower merging (MEPS) are included in the PYTHIA 8 distribution.

Take as an example one-jet merging in W + jets. In this case, we want to take a "hard" jet from the pp $\rightarrow$ Wj matrix element (ME), while soft jets should be modelled by emissions off pp $\rightarrow$ W states, generated by Pythia.

We will now adapt the `mymain` program from above, towards what would be needed for merged predictions.

- First, to produce the two samples mentioned above, we need to run Pythia both with pp $\rightarrow$ Wj, and with pp $\rightarrow$ W input. You can use the same Pythia object to run both samples consecutively:

```
Hist pTW0("pT of W-boson, zero-jet ME", 100, 0., 100.);
pythia.init("./w+_production_lhc_0.lhe");
   ...
      event loop, particle loop, histogramming for 0-jet sample, normalisation of histograms, etc.
   ...
Hist pTW1("pT of W-boson, one-jet ME", 100, 0., 100.);
pythia.init("./w+_production_lhc_1.lhe");
   ...
      event loop, particle loop, histogramming for 1-jet sample, normalisation of histograms, etc.
   ...
```

- In the following, it is possible that you try to read more events than are available. This can e.g. happen when vetoing events internally to generate Sudakov form factors. To make sure that the normalisation introduced in the previous section is still valid, change the `pythia.next()` statement to

```
if( !pythia.next() ) {
   if( pythia.info.atEndOfFile() ) break;
   else continue;
}
```

so that you exit the event loop one you have reached the end of the LHE file. For the exercise sessions, we have prepared LHE files with 100 000 events for each jet multiplicity. Once you request the generation of a comparable number of events, you need to implement this change.

- So far, your adaptions mean that PYTHIA 8 will shower the pp $\rightarrow$ W process and the pp $\rightarrow$ Wj process, without any merging. For a merged prediction, you need to include

```
pythia.readString("Merging:doKTMerging = on");
```

at the beginning of the program. This will enable the merging procedure, with the merging scale defined

[3]Pythia cross sections are given in units of mb. If you instead prefer e.g. pb then multiply by `1e9`.

as $k_\perp$-separation of jets in the $k_\perp$-algorithm. This definition fixes what we mean when we talk about "hard" and "soft" jets:

Hard jets: $\min\{k_\perp(\text{Hard jet})\} > t_{\text{MS}}$
Soft jets: $\min\{k_\perp(\text{Soft jet})\} < t_{\text{MS}}$

There is of course a plethora of hardness criteria. For this reason, Pythia 8 allows you to define your very own merging scale (see below).

- When you have fixed the merging scale definition, you need to pick a merging scale value by adding

  ```
  pythia.settings.readString("Merging:TMS = 30.");
  ```

  at the beginning of the program. Note that t_{MS} has dimension GeV. Here, we have chosen a value of $t_{\text{MS}} = 30$ GeV since the example LHE files have been generated with this cut – no physics is tied to this particular value[4].

- Then, decide on the maximal number of additional jets available from ME calculation, and the hard process, by including

  ```
  pythia.settings.readString("Merging:nJetMax = 1");
  pythia.settings.readString("Merging:Process = pp>e+ve");
  ```

  at the beginning of the program.

- You have almost generated a merged prediction. Being pragmatic, a merged calculation is only a sophisticated reweighting procedure, which will add effects of Sudakov resummation, and momentum-scale running of α_s and parton distributions, to the ME calculation. This means that each event, after the merging procedure, comes with a multiplicative weight to include these effects. The weight can be accessed by calling the function `pythia.info.mergingWeight()`. You will have to fill histograms with this weight, by e.g. changing

  ```
  pTW0.fill(pT);
  ```

  to

  ```
  pTW0.fill(pT,pythia.info.mergingWeight());
  ```

  You can convince yourself that the variation of this weight is moderate.

- If the program is run now, you will have generated the two necessary samples for one-jet merging. You only need to add these predictions to arrive at the merged result[5]. To add the histograms, include

  ```
  Hist pTWSum("pT of W-boson, merged prediction", 100, 0., 100.);
  pTWSum = pTW0 + pTW1;
  cout << pTWSum;
  ```

  after you have filled and normalised `pTW0` and `pTW1`. With the histogram `pTWSum`, you have now produced a merged prediction for the transverse momentum of the W-boson.

5. Merging with two jets

The Pythia 8 distribution is shipped with a three LHE files (`examples/w+_production_*`) with a very small number of events for $W^+ + \leq 2$ jets. These files are regularised in $k_\perp$, so that by changing

```
pythia.settings.readString("Merging:nJetMax = 1");
```

to

```
pythia.settings.readString("Merging:nJetMax = 2");
```

you can perform a merging for up to two additional hard jets. Of course, that means that you now need to generate events from three LHE files, histogram all three samples, and at the end, add three instead of two contributions. Due to the tiny number of events, this does not allow for a reasonable analysis, but gives a hint how easily your program can be generalised to multi-jet merging.

[4]In fact, co-varying t_{MS} in the matrix element calculation and the merging algorithm, and analysing differences in the combined result, gives the classical error estimate for MEPS methods.
[5]This will not double-count contributions: The removal of double counting exactly what merging procedures are intended for.

6. Input Files

With the `mymain.cc` structure developed above it is necessary to recompile the main program for each minor change, e.g. if you want to rerun with more statistics. This is not time-consuming for a simple standalone run, but may become so for more realistic applications. Therefore, parameters can be put in special input "card" files that are read by the main program.

We will now create such a file, with the same settings used in the `mymain` example program. Open a new file, `mymain.cmnd`, and input the following

```
! W + jet merging at the LHC
Merging:doKTMerging = on        ! Merging scale definition used to regularise ME
Merging:TMS         = 30        ! Merging scale value (= cut value in ME)
Merging:nJetMax     = 1         ! Number of additional jets available from ME
Merging:Process     = pp>e+ve   ! Process to be merged
```

The `mymain.cmnd` file can contain one command per line, of the type

```
variable = value
```

All variable names are case-insensitive (the mixing of cases has been chosen purely to improve readability) and non-alphanumeric characters (such as !, # or $) will be interpreted as the start of a comment. All valid variables are listed in the online manual (see Section 2, point 6, above). Cut-and-paste of variable names can be used to avoid spelling mistakes.

The final step is to modify our program to use this input file. The name of this input file can be hardcoded in the main program, but for more flexibility, it can also be provided as a command-line argument. To do this, replace the

```
int main() {
```

line by

```
int main(int argc, char* argv[]) {
```

and replace all `pythia.readString(...)` commands with the single command

```
pythia.readFile(argv[1]);
```

after the declaration of the `pythia` object. The executable `mymain.exe` is then run with a command line like

```
./mymain.exe mymain.cmnd > mymain.out
```

and should give the same output as before.

In addition to all the internal PYTHIA variables there exist a few defined in the database but not actually used. These are intended to be useful in the main program, and thus begin with `Main:`. The most basic of those is `Main:numberOfEvents`, which you can use to specify how many events you want to generate. To make this have any effect, you need to read it in the main program, after the `pythia.readFile(...)` command, by a line like

```
int nEvent = pythia.mode("Main:numberOfEvents");
```

and set up the **event loop** like

```
for (int iEvent = 0; iEvent < nEvent; ++iEvent) {
```

The online manual also exists in an interactive variant, where you semi-automatically can construct a file with all the command lines you wish to have. This requires that somebody installs the `pythia81xx/phpdoc` directory in a webserver. If you lack a local installation you can use the one at

```
http://home.thep.lu.se/~torbjorn/php81xx/Welcome.php
```

This is not a commercial-quality product, however, and requires some user discipline. Full instructions are provided on the "Save Settings" page.

7. Changing Default Settings

You are now free to play with further options in the input file (or use the `pythia.readString` method directly in your code), and make changes such as:

- `Tune:pp = 5` (or other values between 1 and 7)
  Different combined tunes, in particular to radiation and multiple interactions parameters. In part this reflects that no generator is perfect, and also not all data is perfect, so different emphasis will result in different optima. Currently, the default tune for LHC is Tune 4C, (which can also be explicityly set by `Tune:pp = 5`). What happens if you switch to Tune A2, which is a recent tune that can be used by putting `Tune:pp = 7`?
- `SpaceShower:rapidityOrder = off`
  In Tune 4C, the initial state radiation is ordered both in decreasing $p_{\perp,evol}$ (the PYTHIA evolution $p_{\perp}$) and in decreasing rapidity. This switch allows to remove the enforced rapidity ordering. How does this affect the MEPS predictions?

The philosophy of PYTHIA is to make every parameter available to the user. A complete list of changeable settings can be found in the online manual. Have a look at the switches related to MEPS merging. As additional challenge, think about which switches are dangerous, i.e. will maximally corrupt your predictions. Can you isolate a singularly bad setting? Why does the prediction deteriorate with the changes?

7.1. Switching On or Off Simulation Steps

In the same way, you can switch off parts of the simulation, e.g.

- `PartonLevel:FSR = off`
  switch off final-state radiation.
- `PartonLevel:ISR = off`
  switch off initial-state radiation.
- `PartonLevel:MPI = off`
  switch off multiple interactions.

For debugging your code for instance, it might be reasonable to switch off multiple interactions, so that you can produce plots more quickly.

8. Additional challenges

If you have time left, you should take the opportunity to try a few other processes or options. Below are given some examples, but feel free to pick something else that you would be more interested in. Don't hesitate to contact us [6].

8.1. Changing the Merging Scale Definition

If you are a bit more ambitious, or you would like to use PYTHIA's merging facilities in the future, it might be nice to know how you communicate your favourite merging scale definition to PYTHIA. This involves only a little more work, and would be useful if e.g. you already have LHE files that you would like to use with PYTHIA.

The merging procedure can be influenced with methods of the class `Pythia::MergingHooks`. The philosophy here is close to the one of `Pythia::UserHooks`: Like `Pythia::UserHooks` allows the user to influence the event generation at different stages in the evolution, so does `Pythia::MergingHooks` allow the user to intervene inside the merging.

The starting point is to tell Pythia that you want more control by setting

```
Merging:doUserMerging = on
```

instead of setting `Merging:doKTMerging = on`. Then you need to derive your own `Pythia::MergingHooks` object, by adding e.g.

```
// Class for interaction with the merging
class MyMergingHooks : public MergingHooks {
  public:
  // Default constructor
  MyMergingHooks() {};
  // Default destructor
 ~MyMergingHooks() {};
```

```
    // User defined functional form of the merging scale
    virtual double tmsDefinition( const Event& event);
  };
```

between the `using namespace Pythia8;` and `int main()` statements.

As an example, let us consider minimal kinematical p_T as a merging scale, and a matrix element cut of $p_{T,min} = 20$ GeV. Then, your merging scale definition could be included by writing

```
// Definition of the merging scale
double MyMergingHooks::tmsDefinition( const Event& event){
  // Declare overall veto
  bool doVeto = false;
  // Declare cut used in matrix element
  double pTjmin = 20.;
  // Declare minimum value
  double minPT  = 8000.;
  // Check matrix element cuts
  for( int i=0; i < event.size(); ++i){
    if( event[i].isFinal() && event[i].isParton() ) {
      // Save pT value
      minPT = min(minPT,event[i].pT());
    }
  }
  // Check if all partons are above the merging scale
  if(minPT  > pTjmin) doVeto  = true;
  // If event is above merging scale, veto
  if(doVeto) return 1.;
  // Else, do nothing
  return -1.;
}
```

after the `MyMergingHooks` declaration, and setting

```
     Merging:TMS = 0
```

as input for Pythia 8. Note the convention that for a state produced by the matrix element generator, the merging scale definition should always give a value larger than the cut `Merging:TMS`.

Now, after you have chosen a definition, construct an object and tell Pythia about the existence of your definition by including

```
  MergingHooks* myMergingHooks = new MyMergingHooks();
  pythia.setMergingHooksPtr( myMergingHooks );
```

after the declaration of the `pythia` object.

Once you have decided on a `tmsDefinition`, and communicated `myMergingHooks` to Pythia, your definition will take precedence over the default choice whenever the merging scale definition is invoked internally.

In this example, kinematical p_T would have been used as merging scale. Clearly, this will not be enough to regularise multi-jet matrix elements. Finite MEs could e.g. be achieved by applying a combined ΔR_{ij}, $p_{T,i}$ and m_{ij} cut on all combinations of (light) jets i and j. This cut is then a potential candidate for a user-defined merging scale.

8.2. Merging with more jets, and user-defined merging scale

So far, you have learned how to set up a merged prediction with one additional hard jet. For most background studies however, it would be prudent to allow for a larger number of well-separated jets. In the installation on your virtual machine, we have included LHE files for W+ ≤ 4 jets and for Z+ ≤ 4 jets[6]. These files have been regularised by requiring the cuts

$$\begin{aligned} \Delta R_{ij} &= 0.1 && (1)\\ p_{T,i} &= 20\text{ GeV} && (2)\\ m_{ij} &= 20\text{ GeV} && (3) \end{aligned}$$

[6]During the course of the tutorial, we will inform you where in the local file system to find the LHE files.

Define this combined cut as the merging scale, i.e. classify an event as "inside the matrix element region" if all (combinations of) jets pass all three cuts, and as "inside the parton shower region" otherwise. For this, you will need to derive a `myMergingHooks::tmsDefinition` function.

After this, you can expand `mymain` to two-, three-, or four-jet merging by reusing the `pythia` object. Some inspiration on how to do reuse the `pythia` object within a loop over jet multiplicities can be taken from the `main84.cc` sample program.

Once you have done this, it might be nice to have a look at

- The W-boson p_T again. Does it change? If it does, can you explain why?
- The $k_\perp$ of jets. For this, you need to define jets with a jet algorithm first. In `main84.cc`, you will find a function (`pTfirstJet`) using `fastjet` to define jets. Can you use this to find the $k_\perp$-separation between e.g. the hardest and second hardest jet? How does this change when including additional jets?
- Changing the default choices in the merging procedure[7]. For example, check `Merging:unorderedScalePrescrip`. How are observables influenced by your choices?

8.3. Writing HepMC output for Matrix-Element-merged predictions

Finally, we can move from just looking at plots to analysing complete events. We will use the HepMC event-record format for this. HepMC events can also be used as input for detector simulations leading up to a full-fledged experimental analysis.

For this training session, HepMC is already installed in `/mcschool/opt`. Also, this installation is prepared for HepMC usage. To use the prepared installation, you need to in the `Makefile` include `mymain` in the list of main programs linking to HepMC, by e.g. changing the line

```
main81 main82 main83 main84: \
```

to

```
main81 main82 main83 main84 mymain: \
```

and remove `mymain` elsewhere in the `Makefile`. If you want to want to link your own installation of Pythia 8 to HepMC, please consult Appendix B, Steps 4ff. In the following, assume that you have linked to a running installation of HepMC. Then, if you want to include HepMC-event output in your main program, start with the following:

1. Include the HepMC libraries in your main program by adding the lines

   ```
   #include "HepMCInterface.h"
   #include "HepMC/GenEvent.h"
   #include "HepMC/IO_GenEvent.h"
   ```

   after the `#include "Pythia.h"` statement.

2. Define an object converting the Pythia event to HepMC, and a file where the HepMC events will be stored by adding the lines

   ```
   HepMC::I_Pythia8 ToHepMC;
   HepMC::IO_GenEvent ascii_io(filename, std::ios::out);
   ```

   before the **event loop**.

3. Inside the **event loop**, create a HepMC event for each event:

   ```
   HepMC::GenEvent* hepmcevt = new HepMC::GenEvent();
   ```

   then convert the Pythia event to HepMC and write in to the file

   ```
   ToHepMC.fill_next_event( pythia, hepmcevt );
   ascii_io << hepmcevt;
   delete hepmcevt;
   ```

[7]This will result in changing how contributions formally beyond the accuracy of Ckkw-l MEPS method. Together with variations of the merging scale value, changing default choices can give (very) conservative error estimates for the method.

It might be good to know that the files `main41.cc` and `main42.cc` are intended as examples to produce HepMC event files.

In case of matrix element merging, this is however not the end. As discussed above, events in MEPS come with weights to include effects of Sudakov resummation, α_s and PDF running. For a merged prediction all events need to have the correct relative weight, consisting of the accepted cross section, and the "merging weight" of the current event. The accepted cross section is not known a priori. We solved this puzzle earlier by rescaling the Pythia-histograms with `pythia.info.sigmaGen()` after the **event loop**, for each jet multiplicity separately. HepMC does not allow rescaling of all event weights after the event generation, so this trick is not an option. Instead, we need to estimate the cross section before we print merged HepMC events to an output file. How this can be done is illustrated in `main84.cc`. If you have estimated the accepted cross section, you can set the correct weight of each HepMC output event by invoking

```
hepmcevt->weights().push_back( pythia.info.mergingWeight() * normhepmc );
```

before filling the event with the `fill_next_event` method. Here, `normhepmc` is the accepted cross section for the current jet multiplicity. Finally, Rivet analyses sometimes require the full, merged cross section to e.g. normalise plots to cross sections, which is also not know in advance. Since Rivet reads this cross section from the cross section information stored in the last event, here, it is enough to sum the correct weight of each HepMC event, and set the cross section of the last event in the HepMC file to this sum. This is also illustrated in `main84.cc`.

Have you come this far?
Then you hold a general main program for multi-jet merging, which you can use to generate LHC events.
Congratulations!

A. The Event Record

The event record is set up to store every step in the evolution from an initial low-multiplicity partonic process to a final high-multiplicity hadronic state, in the order that new particles are generated. The record is a vector of particles, that expands to fit the needs of the current event (plus some additional pieces of information not discussed here). Thus `event[i]` is the `i`'th particle of the current event, and you may study its properties by using various `event[i].method()` possibilities.

The `event.list()` listing provides the main properties of each particles, by column:

- `no`, the index number of the particle (`i` above);
- `id`, the PDG particle identity code (method `id()`);
- `name`, a plain text rendering of the particle name (method `name()`), within brackets for initial or intermediate particles and without for final-state ones;
- `status`, the reason why a new particle was added to the event record (method `status()`);
- `mothers` and `daughters`, documentation on the event history (methods `mother1()`, `mother2()`, `daughter1()` and `daughter2()`);
- `colours`, the colour flow of the process (methods `col()` and `acol()`);
- `p_x`, `p_y`, `p_z` and `e`, the components of the momentum four-vector (p_x, p_y, p_z, E), in units of GeV with $c = 1$ (methods `px()`, `py()`, `pz()` and `e()`);
- `m`, the mass, in units as above (method `m()`).

For a complete description of these and other particle properties (such as production and decay vertices, rapidity, $p_\perp$, etc), open the program's online documentation in a browser (see Section 2, point 6, above), scroll down to the "Study Output" section, and follow the "Particle Properties" link in the left-hand-side menu. For brief summaries on the less trivial of the ones above, read on.

A.1. Identity codes

A complete specification of the PDG codes is found in the Review of Particle Physics [3]. An online listing is available from

`http://pdg.lbl.gov/2008/mcdata/mc_particle_id_contents.html`

A short summary of the most common `id` codes would be

1	d	11	e^-	21	g	211	π^+	111	π^0	213	ρ^+	2112	n
2	u	12	ν_e	22	γ	311	K^0	221	η	313	K^{*0}	2212	p
3	s	13	μ^-	23	Z^0	321	K^+	331	η'	323	K^{*+}	3122	Λ^0
4	c	14	ν_μ	24	W^+	411	D^+	130	K^0_L	113	ρ^0	3112	Σ^-
5	b	15	τ^-	25	H^0	421	D^0	310	K^0_S	223	ω	3212	Σ^0
6	t	16	ν_τ			431	D_s^+			333	ϕ	3222	Σ^+

Antiparticles to the above, where existing as separate entities, are given with a negative sign.
Note that simple meson and baryon codes are constructed from the constituent (anti)quark codes, with a final spin-state-counting digit $2s+1$ (K^0_L and K^0_S being exceptions), and with a set of further rules to make the codes unambiguous.

A.2. Status codes

When a new particle is added to the event record, it is assigned a positive status code that describes why it has been added, as follows:

code range	explanation
11 – 19	beam particles
21 – 29	particles of the hardest subprocess
31 – 39	particles of subsequent subprocesses in multiple interactions
41 – 49	particles produced by initial-state-showers
51 – 59	particles produced by final-state-showers
61 – 69	particles produced by beam-remnant treatment
71 – 79	partons in preparation of hadronization process
81 – 89	primary hadrons produced by hadronization process
91 – 99	particles produced in decay process, or by Bose-Einstein effects

Whenever a particle is allowed to branch or decay further its status code is negated (but it is *never* removed from the event record), such that only particles in the final state remain with positive codes. The `isFinal()` method returns `true/false` for positive/negative status codes.

A.3. History of parton shower branchings

The two mother and two daughter indices of each particle provide information on the history relationship between the different entries in the event record. The detailed rules depend on the particular physics step being described, as defined by the status code. As an example, in a $2 \to 2$ process $ab \to cd$, the locations of a and b would set the mothers of c and d, with the reverse relationship for daughters. When the two mother or daughter indices are not consecutive they define a range between the first and last entry, such as a string system consisting of several partons fragment into several hadrons.

There are also several special cases. One such is when "the same" particle appears as a second copy, e.g. because its momentum has been shifted by it taking a recoil in the dipole picture of parton showers. Then the original has both daughter indices pointing to the same particle, which in its turn has both mother pointers referring back to the original. Another special case is the description of ISR by backwards evolution, where the mother is constructed at a later stage than the daughter, and therefore appears below in the event listing.

If you get confused by the different special-case storage options, the two `pythia.event.motherList(i)` and `pythia.event.daughterList(i)` methods are able to return a `vector` of all mother or daughter indices of particle `i`.

A.4. Colour flow information

The colour flow information is based on the Les Houches Accord convention [4]. In it, the number of colours is assumed infinite, so that each new colour line can be assigned a new separate colour. These colours are given consecutive labels: 101, 102, 103, ... A gluon has both a colour and an anticolour label, an (anti)quark only (anti)colour.

While colours are traced consistently through hard processes and parton showers, the subsequent beam-remnant-handling step often involves a drastic change of colour labels. Firstly, previously unrelated colours and anticolours taken from the beams may at this stage be associated with each other, and be relabelled accordingly. Secondly, it appears that the close space–time overlap of many colour fields leads to reconnections, i.e. a swapping of colour labels, that tends to reduce the total length of field lines.

B. Installing Pythia 8

While Pythia can be run standalone, it can also be interfaced with a set of other libraries. One example is HepMC, which is the standard format used by experimentalists to store generated events. Since the the location of external libraries is naturally installation-dependent, it is not possible to give a fool-proof linking procedure, but some hints are given below.

If you would like to install Pythia 8 on your private machine, and you have a C++ compiler, here is how to install the latest Pythia 8 version on a Linux/Unix/MacOSX system as a standalone package.

1. In a browser, go to
   ```
   http://www.thep.lu.se/∼torbjorn/Pythia.html
   ```
2. Download the (current) program package
   ```
   pythia81xx.tgz
   ```
   to a directory of your choice (e.g. by right-clicking on the link).
3. In a terminal window, `cd` to where `pythia81xx.tgz` was downloaded, and type
   ```
   tar xvfz pythia81xx.tgz
   ```
   This will create a new (sub)directory `pythia81xx` where all the Pythia source files are now ready and unpacked.
4. Move to this directory (`cd pythia81xx`). If you are only interested in directly producing plots from Pythia event records, you can directly go to the next step. If you want to produce and store HepMC event output, configure the program in preparation for the compilation by typing
   ```
   ./configure --with-hepmc=path
   ```
   where the directory-tree `path` would depend on your local HepMC installation. Should `configure` not recognise the version number you can supply that with an optional argument, like
   ```
   ./configure --with-hepmc=path --with-hepmcversion=2.04.01
   ```
5. Do a `make`. This will take 2–3 minutes (computer-dependent). The Pythia 8 libraries are now compiled and ready for physics.
6. For test runs, `cd` to the `examples/` subdirectory. An `ls` reveals a list of programs, `mainNN`, with `NN` from `01` through `30`. These example programs each illustrate an aspect of Pythia 8. For a list of what they do, see the `README` file in the same directory or look at the online documentation.
   Initially only use one or two of them to check that the installation works. Once you have worked your way though the introductory exercises in the next sections you can return and study the programs and their output in more detail.
   If you want to produce HepMC output, do either of
   ```
   source config.csh
   source config.sh
   ```
   the former when you use the csh or tcsh shells, otherwise the latter. (Use `echo $SHELL` if uncertain.). If you are not interested in HepMC, this step can be skipped.
   To execute one of the test programs, do
   ```
   make mainNN
   ./mainNN.exe
   ```
   The output is now just written to the terminal, `stdout`. To save the output to a file instead, do `./mainNN.exe > mainNN.out`, after which you can study the test output at leisure by opening `mainNN.out`. See Appendix A for an explanation of the event record that is listed in several of the runs.
7. If you open the file
   ```
   pythia81xx/htmldoc/Welcome.html
   ```

you will gain access to the online manual, where all available methods and parameters are described. Use the left-column index to navigate among the topics, which are then displayed in the larger right-hand field.

References

[1] T. Sjöstrand, S. Mrenna and P. Skands, Comput. Phys. Comm. **178** (2008) 852 [arXiv:0710.3820]

[2] T. Sjöstrand, S. Mrenna and P. Skands, JHEP **05** (2006) 026 [hep-ph/0603175]

[3] Particle Data Group, C. Amsler et al., Physics Letters **B667** (2008) 1

[4] E. Boos et al., in the Proceedings of the Workshop on Physics at TeV Colliders, Les Houches, France, 21 May - 1 Jun 2001 [hep-ph/0109068]

[5] L. Lönnblad and S. Prestel, arXiv:1109.4829 [hep-ph]

[6] For merging related questions, email `stefan.prestel@thep.lu.se`
In case of general problems, contact us under `pythia8@projects.hepforge.org`

Simon Plätzer, Stefan Prestel, Christian Röhr, Frank Siegert, Christian Speckner March, 12th-15th, 2012

Sherpa Tutorial

1. Introduction

Sherpa is a full-featured event generator which puts its emphasis on an improved description of multi-jet final states which are relevant as SM backgrounds to many BSM signals. These are mainly described by the perturbative stages of event generation, i.e. the hard scattering process described by matrix elements and the parton shower stage with its resummation of soft and collinear enhancements.

One main ingredient for improvement is the generation of hard QCD emissions with an exact matrix element, because the parton shower approximation is not valid in that case. To run a parton-shower on top of such a matrix element which potentially already contains hard emissions, a prescription called “ME+PS merging” is implemented. Information about this merging can be found in the Sherpa reference publication [1] and in more detail at [2] and its references. In addition to these perturbative event phases, Sherpa also allows for the simulation of QED radiation resummed in the YFS approach, hadronisation, hadron decays, and multiple parton interactions.

Sherpa 1.3.1 is installed in `/mcschool/opt` in the VM, but if you want to install it on your laptop later, that is easily done as described in the Sherpa manual [3].

2. Input file

The way a particular simulation runs in Sherpa is defined by several parameters, which are listed by the user in the steering file `Run.dat`. The first step in running Sherpa is to adjust all parameters to the needs of the desired simulation. Instructions for properly constructing these files are given in the Sherpa manual [3], but for now we will discuss just a few of the most important features using the example of Z+jet production at the LHC.

Start in a fresh working directory and copy over the run card for this example from

```
/mcschool/opt/share/SHERPA-MC/Examples/LHC_ZJets/Run.dat
```

and open it in your favourite editor.

In the “processes” section of the Run.dat file, the hard scattering processes are specified. The particles are identifed by their PDG codes [4]. There are also so-called particle containers, which allow you to specify several processes with one line. For example, the particle container for jets, ”93”, includes all processes with $d, \bar{d}, u, \bar{u}, s, \bar{s}, c, \bar{c}, b, \bar{b}, g$ in this place. A list of particle codes and particle containers is displayed when Sherpa is run.

Look at the run card and the manual [3] to find the following information about the run:

- Beam settings
- Hard scattering process
  - Which physics process are we running? Which lepton is being produced?
  - Up to how many hard jets are generated by exact matrix elements?
    (Hint: Look for curly brackets)
- Parton level cuts
  - Are any cuts imposed on the hard scattering?
    (Hint: Look at the “selector” section)
  - Why are they necessary?

(Note that this run card is also documented in the Sherpa manual [3].)

3. Changing some settings

Later in this tutorial we plan on studying VBF Higgs production with the Higgs decaying invisibly into LSPs. Sherpa will be used for the SM background samples, and one potential background could be Z+jets production with the Z decaying into neutrinos. To start our preparations for that, let's make a few changes to the run card:

- Change the beam settings to LHC at 8 TeV
- Switch the hard scattering process to $pp \to \nu_e \bar{\nu}_e$
  (Hint: The PDG code for ν_e is 12.)
- Consider whether the cuts on the hard scattering are still sensible.
- As you have hopefully found out, the example run card generates Drell-Yan events with up to 4 additional jets in the matrix element. We will later use samples that were generated with up to 4 additional jets, but for the time being, go ahead and reduce the number of additional jets to `93{0}`, which amounts to running just a LO generator + parton shower.

4. Running it

When run for the first time in your working directory, Sherpa will integrate the cross sections of the hard scattering processes you have requested. Depending on the complexity of the processes this may take a rather long time. The integration results will be stored in the directory `Results` and are automatically re-used in later runs in the same working directory..

Now it is time to run Sherpa for the first time. This is as easy as typing

```
Sherpa
```

This should start with the phase space integration, and the directory `Results` will contain the stored integration results at the end of the run. The run card specifies that 10000 events should be generated. If this takes too long, you can simply abort the event generation using `Ctrl-c`. You can also change the number of events (as well as any other parameter) on the command line like:

```
Sherpa EVENTS=100 OUTPUT=2
```

So far, the events were only generated and not written out to file or analysed yet. The `OUTPUT` setting will let you switch between different levels of detail in the on-screen output, e.g.:

- `OUTPUT=2` Information messages only (the default)
- `OUTPUT=3` Information messages + event output on screen
- `OUTPUT=15` Information messages + event output + full debugging output (a lot!)

Let's first switch to `EVENTS=1` and `OUTPUT=3` and actually look at the structure of one event printed on screen.

5. Writing HepMC Event Files

In the following, we want to use the Rivet Monte-Carlo analysis framework to analyse the events generated by different Monte-Carlo programs. This is documented at the beginning of this hand-out and in addition for the tutorial today you only have to know how to write out HepMC event files with Sherpa. All of this is documented in section 6.1.12 of the Sherpa manual [3], but the upshot is: With the keyword `HEPMC2_SHORT_OUTPUT=<filename>` you specify the filename to be written to, either in the run section of the run card, or on the command line. The extension `.hepmc` will be appended to that filename.

So to write into a fifo-file named `fifo.hepmc` you could run:

```
mkfifo fifo.hepmc
Sherpa HEPMC2_SHORT_OUTPUT=fifo &
rivet --analysis-path=[...] -a MY_ANALYSIS -H z+0jet.aida fifo.hepmc
```

This should create the file `z+0jet.aida` containing the histograms from your analysis and the events you generated with Sherpa. Together with the information from the beginning of this hand-out you should now be able to create plots. To see the effects of any changes we make in the following, the `MY_ANALYSIS` that you have extended in the Rivet part of this tutorial comes in handy.

6. Playing with the physics

Here are a few ideas for settings which you can play around with based on the example run card from above.

6.1. Generating Feynman graphs for the hard scattering processes

To plot all Feynman graphs included in the simulation of the specified core process, the `Print_Graphs` option is available as documented in Sec. 6.6.15 of the Sherpa manual [3]. It has to be added to the `Process` block before the `End process` line. Convince yourself that only the Feynman graphs you would expect are included in this example run.

6.2. Including matrix element corrections

In the example run card above we were running just a LO + parton shower simulation of the $pp \to \nu\bar{\nu}$ process. Now let's improve on that by including the tree-level matrix elements for $\nu\bar{\nu}+1$ and $\nu\bar{\nu}+2$ jets production in the simulation using ME+PS merging. For that you would change `93{0}` back to `93{1}` (for 0 and 1 jet accuracy) or even `93{2}` (for 0, 1 and 2 jet accuracy). Now run these three setups in separate working directories through your Rivet analysis and compare the results:

- What impact does the correct modelling of the multi-jet final states have on the E_T^{miss} distribution? Why?
- How are the jet properties (multiplicity, $p_\perp$ spectra) affected?

6.3. Varying the ME+PS merging cut

ME+PS merging is based on a slicing of the QCD radiation phase space into a region of hard emissions ($\approx$ high $p_\perp$) and soft/collinear emissions ($\approx$ low $p_\perp$). The hard region is filled by the exact matrix element, while the soft/collinear region is filled by the parton shower approximation. Obviously, the exact value of the cut between these two regions is somewhat arbitrary and unphysical. It is set by the `CKKW` line in the processes section, and in the example was chosen to be 30 GeV. Find out how sensitive the observables are to variations of this unphysical ME+PS merging cut! Try changing it in a reasonable range (15-60 GeV) around the original 30 GeV. (If you have enough time left: What happens if you set it above the factorisation scale (e.g. 150 GeV)?)

Note that you should work in separate working directories for different values of the merging cut since the integration results will be different.

6.4. Hadronisation and Underlying Event

In the run card so far, multiple parton interactions were disabled, but the hadronisation was enabled. Effects from hadronisation and multiple parton interactions can be studied by comparing a run where both are disabled:

```
FRAGMENTATION=Off
MI_HANDLER=None
```

with a run where both are enabled:

```
FRAGMENTATION=Ahadic
MI_HANDLER=Amisic
```

6.5. Other hard scattering processes

For the study of BSM signal and backgrounds on Wednesday, you are going to be provided with pre-generated samples due to time constraints. There will be Sherpa samples for 4 different processes available. You can find the respective run cards with which these have been generated in `/mcschool/opt/share/SHERPA-MC/samples`. The samples are as follows:

- `zee`:
  Drell-Yan production of electrons with up to 4 jets.
- `wlv`:
  $W \to \ell\nu$ production with up to 4 jets.

- `zvv`:
  $Z \to \nu\bar{\nu}$ production with up to 4 jets.

- `vbfzvv`:
  Electroweak $Z[\to \nu\bar{\nu}] + 2$ jets production, which also contains VBF-like graphs.

With the information learned so far you will probably understand most aspects of these run cards. For the last process it might be interesting to plot the Feynman graphs. To do so, copy over the run card and modify it as you have done earlier in the tutorial.

If you have any questions about these samples, please don't hesitate to ask!

Final Remarks

Congratulations, you've made it this far! You should now be equipped with enough knowledge about what is contained in the background samples that you will be using in Wednesday's tutorial. If any questions have been left open please ask during the tutorial.

Further information about Sherpa can be obtained from the following references.

References

[1] T. Gleisberg, S. .Höche, F. Krauss, M. Schönherr, S. Schumann, F. Siegert and J. Winter, JHEP **0902**, 007 (2009) [arXiv:0811.4622 [hep-ph]].

[2] S. Höche, F. Krauss, S. Schumann and F. Siegert, JHEP **0905** (2009) 053 [arXiv:0903.1219 [hep-ph]].

[3] Sherpa 1.3.1 manual: `http://sherpa.hepforge.org/doc/SHERPA-MC-1.3.1.html`

[4] `http://pdg.lbl.gov/2011/mcdata/mc_particle_id_contents.html`

[5] Sherpa homepage: `http://sherpa.hepforge.org`

[6] E-mail the Sherpa authors with your questions and comments at `mailto:sherpa@projects.hepforge.org`

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[8], 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.

[8]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.[9]

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 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

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

```
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[10]. 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 $1\,\mathrm{fb}^{-1}$ 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

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

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`[11]. In fact, assigning an `alias` creates a variable of the `PDG(...)` type which we already encountered in the variable list.

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

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.

4.2. 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

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

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 try out

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[12] 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
```

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

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

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 try out

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 `$sample = "xxx"`, and you can specify an alternate file extension with `$extension_hepmc`.

Isolating a VBF Higgs signal with invisible decay

In this part of the tutorial we will focus on properties of Higgs production in vector boson fusion, with a subsequent decay of the Higgs to a pair of neutralinos in a model where neutralinos constitute the stable, lightest supersymmetric particles.

The Higgs boson will thus decay invisibly, meaning that its decay products contribute to the missing energy in each event. The most significant backgrounds to this signal are QCD and electroweak production of Z and $W^{\pm}$ bosons in association with several jets.

We have prepared a number of signal and background processes, from different generators. These can be found in `/mcschool/events` after issuing the command `mount-samples SERVER`.

- Try to develop a `Rivet` analysis for looking at event properties process by process.
- Compare different generator predictions. What are the similarities/differences?
- Try to define cuts and vetoes to reduce the number of background events.
- Evaluate the efficiencies of your cuts for signal and background.

In order to design the analysis, we will first look at the properties of signal and background processes. Here are a few hints of useful properties to look at:

- What is the average number of jets in a certain class of events?
- What are their kinematic properties?
- What are the properties of isolated leptons?[13]
- What is the distribution of missing energy in the events?

The final goal of this tutorial is to develop an analysis which will be able to extract the Higgs signal out of the major backgrounds. To this end, we will provide an event sample containing both signal and background events (though not with realistic cross sections to make the signal more significant). We provide several combinations of different signal and background predictions in `/mcschool/events`.

- Apply your analysis to the mixed signal/background samples.
- Assuming that each sample contains equal contributions from each background process, and the signal to background cross sections satisfy $\sigma_S : \sigma_B = 1 : 9$ how many signal events did your analysis extract?
- Compare this number to the generated 'truth' by looking for events with neutralinos in the final state. How close can you get to the total number of signal events contained in the mixed samples?

[13]An isolated lepton is a lepton with no hadronic activity nearby above a certain $p_{\perp}$ threshold. Isolated leptons should not enter the jet clustering!