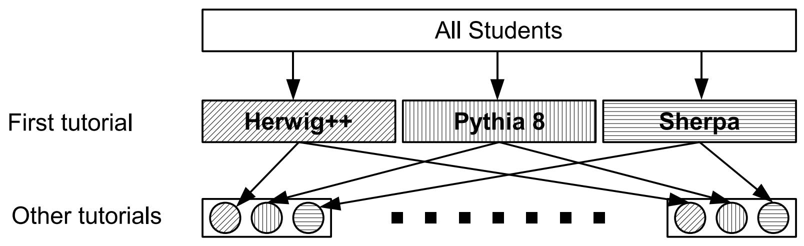
Contents

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 the underlying event. 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 three 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. The procedure is sketched below:



In later tutorials we will use pre-generated event samples obtained by the three 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 `hamburg0311` 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 can be initialised by sourcing the following environment settings:

```
$ source /mc-school/opt/rivet/rivetenv.sh
```

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 demonstrate how to use this to analyse HepMC events from a text file in the HepMC format.

Firstly, 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=500000 --hepmc-output=hepmc.fifo &
$ rivet --analysis=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`!

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. An easier format is available if you convert the AIDA file to a flat text format (the long-term Rivet data format) with the aida2flat command:

```
$ aida2flat Rivet.aida | less
```

Rivet comes with three commands — `rivet-mkhtml`, `compare-histos` and `make-plots` — for comparing and plotting data files. These commands produce nice comparison plots of publication quality from the AIDA format text files.

The high level program `rivet-mkhtml` will automatically create a plot webpage from the given AIDA files. It searches for reference data automatically and uses the other two commands internally. Example:

```
$ rivet-mkhtml -o plots/ withUE.aida:'Title=With UE' withoutUE.aida:'LineColor=blue'
```

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 later extend in the UE tutorial. You can find it in:

```
/mc-school/opt/rivet/more-analyses/MY_ANALYSIS.cc
```

Please copy that file to a working directory.
Note, that it is also very simple to create a new analysis skeleton using the `rivet-mkanalysis` script.

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. All analyses bundled with Rivet store their metadata in external files in the YAML format though. Useful analyses also contain member variables for the analysis: event weight counters and histograms are the most common of these.

The constructor and three event loop methods are used for the following:

- Constructor: set whether the generator cross-section is needed. Minimal!
- `init`: book histograms, projections, ROOT tuples, etc.
- `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. The projections are used 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 ChargedFinalState& cfs = applyProjection<ChargedFinalState>(event, "CFS");
```

The name "CFS" has been registered in the `init` method as referring to a projection of type "ChargedFinalState" — a calculator which provides a list of charged particles with certain basic cuts applied. This is done via the `addProjection` method. 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!

5.2 Compiling and linking

To use your new analysis, you need to build it into a Rivet analysis plugin library, with a name of the form `Rivet*.so` library. You can do this manually, but to make life easier there is again a helper script, used as follows:

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

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 new analysis right away. Set the `RIVET_ANALYSIS_PATH` variable such that it contains the directory where the `RivetMyAnalysis.so` shared library file was created and test it with the `rivet` command:

```
$ ls
RivetMyAnalaysis.so MY_ANALYSIS.cc
$ export RIVET_ANALYSIS_PATH=$PWD
$ rivet --list-analyses
[...]
MY_ANALYSIS
```

Alternatively, you can use the `--analysis-path` flag to `rivet`:

```
$ rivet --list-analyses --analysis-path=$PWD
```

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 their abilities: `http://projects.hepforge.org/rivet/code/dev/hierarchy.html` Projections you might need for this tutorial are `FinalState`, `ChargedFinalState` and `FastJets`.

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

```
ParticleVector particles =
        applyProjection<ChargedFinalState>(event, "CFS").particlesByPt();
MSG_INFO("Total charged multi = " << particles.size());
foreach (const Particle& p, particles) {
  const double eta = p.momentum().eta();
  MSG_DEBUG("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 MC_ANALYSIS.

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://projects.hepforge.org/rivet/trac/wiki/
- PDF manual
  http://projects.hepforge.org/rivet/rivet-manual.pdf
- Slides from an ATLAS Rivet tutorial
  http://projects.hepforge.org/rivet/rivet-tutorial.pdf
- Code documentation
  http://projects.hepforge.org/rivet/code/dev/hierarchy.html
- Standard analyses list
  http://projects.hepforge.org/rivet/analyses
- Code repository of standard analyses (useful as examples, e.g. the MC_* analyses)
  http://projects.hepforge.org/rivet/trac/browser/trunk/src/Analyses
- Contact the Rivet authors
  mailto:rivet@projects.hepforge.org

Herwig++ Tutorial

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

The Herwig++ homepage is at `http://projects.hepforge.org/herwig/`. To speed up the setup, we have pre-installed Herwig++ version 2.5.0 for the tutorials. To use it, you should copy the configuration files to the working directory (we'll explain their role in section 3 below). For the DESY tutorial,

```
    HERWIGPATH=/mc-school/opt/Herwig++/
```

```
$ cd /intro-school/Herwig++
$ cp /HERWIGPATH/share/Herwig++/LHC.in .
```

For convenience, set a the path variable to include the Herwig++ binary directory:

```
$ export PATH=$PATH:/HERWIGPATH/bin
```

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
```

We'll explain the commands 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) and their parameter settings, assemble them into an event generator and run it. 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 `/HERWIGPATH/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 /HERWIGPATH/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!

5 Changing Default Settings

Take a look at the default settings in `/HERWIGPATH/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?

[2]The default repository can also be re-created by the user, by copying over the files from `/HERWIGPATH/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.

5.1 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 an event handler to `NULL`. Multiple parton interactions are switched off by setting the `MPI` 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.

5.2 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.

5.3 Hard Processes at NLO Accuracy

Herwig++ contains several hard processes which can be simulated at next-to-leading order (NLO) QCD accuracy using the POWHEG algorithm. Have a look at `LHC-Powheg.in`, which contains several processes available at NLO QCD, for example Higgs production in association with a Z boson.

5.4 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).

5.5 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.

5.6 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`.

6 Writing HepMC Event Files

For later analysis we will produce event files in the HepMC format. Herwig++ contains a special analysis handler which is capable of writing the generated events to a HepMC file, `/Herwig/Analysis/HepMCFile`. You can change the default output format by using the `Format` interface of `HepMCFile`. Possible values are `GenEvent,AsciiParticles,Ascii,ExtendedAscii` and `Dump`, which produces human readable output. A filename can be specified using the `Filename` interface. Have a look at `LHC.in`, in the analysis section we already provide an example of a HepMC analysis handler. For the analysis you should create files in the format `Ascii`.

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`.

Stefan Gieseke, Simon Plätzer, Stefan Prestel, Frank Siegert March, 15th-16th, 2011

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 various physics aspects. 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 to all the further references found in them.

PYTHIA 8 is, by today's standards, a small package. It is completely self-contained, and is therefore easy to install for standalone usage, e.g. if you want to have it on your own laptop, or if you want to explore physics or debug code without any danger of destructive interference between different libraries. Section 2 describes the installation procedure, which is what we will need for this introductory session.

When you use 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.

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 HEPMC library location is installation-dependent it is not possible to give a fool-proof linking procedure, but some hints are given below. In this tutorial, there will be no need to produce HEPMC output, but for the ambitious, some hints on how to link HEPMC are given below.

1 Preparation

For this school, a precompiled PYTHIA 8 version `pythia8145`, linked to HEPMC, 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/php8145/Welcome.php`

or if you open

`/mc-school/opt/pythia8145/htmldoc/Welcome.html`

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

`/mc-school/opt/pythia8145/examples`

Information on these examples can be found in the online manual. Before continuing, please note that HEPMC quickly get large when many events are stored.

If you would still 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 (`pythia8145`) 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.

2 Simple LHC Events

The focus of the next sessions will be on Underlying Event and Jet Physics. So, as a start-up excercise, we will now generate $gg \to q\overline{q}$ event at the LHC, using Pythia standalone.

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.
  Pythia pythia;         // Declare Pythia object
  pythia.readString("HardQCD:gg2qqbar = on"); // Switch on gg->qqbar process.
  pythia.readString("PhaseSpace:pTHatMin = 20."); // Regularize the
                                                  // pT-> divergence of QCD
                                                  // cross sections by a pT cut
  pythia.init( 2212, 2212, 7000.); // Initialise pp (PDG 2212) at LHC.

  // Show settings
  pythia.settings.listChanged();      // Show changed settings
  pythia.particleData.listChanged();  // Show changed particle data

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

```
    pythia.event.list();   // Print contents of 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 lines

```
# Create an executable for one of the normal test programs
main00 main01 main02 main03 ...  main09 main10 main10 \
```

and the three next enumerate the main programs that do not need any external libraries. Edit the last of these lines to include also mymain:

```
 main44 main71 mymain: \
```

Now it should work as before with the other examples:

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

whereafter you can 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. To exemplify, consider a quark produced in the hard interaction, initially with positive status code. When later, a shower branching q $\to$ qg occurs, 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.

- Often, we wish to mix several processes together. To add all other Dijet production channels to the above example, just replace the pythia.readString call
  ```
  pythia.readString("HardQCD:gg2qqbar = on");
  ```
  by
  ```
  pythia.readString("HardQCD:all = on");
  ```
- Now 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.
- To obtain statistics on the number of events generated of the different kinds, and the estimated cross sections, add a
  ```
  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.

- If you are only interested in final state particles, the `isFinal()` method can be applied to the event record entry by adding
  ```
  for (int i = 0; i < pythia.event.size(); ++i) {
     if(pythia.event[i].isFinal()) {
           cout << "i = " << i
                 << ", id = " << pythia.event[i].id() << endl;
     }
  }
  ```
  This will only print the PDG code of final particles. You can also check if a particle is charged by inferring the `isCharged()` method.

- 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()`, and azimuthal angle, `pythia.event[i].phi()`. Use these methods to find the azimuthal angle of the hardest charged track.

- We now want to generate more events, say 1000, to study the shape a simple distributions, say the charged multiplicity. 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 mult("charged multiplicity", 100, -0.5, 799.5);
  ```
  where the last three arguments are the number of bins, the lower edge and the upper edge of the histogram, respectively. For the histogram, note that it can be treacherous to have bin limits at integers, where roundoff errors decide whichever way they go. In this particular case only even numbers are possible, so 100 bins from -1 to 399 would still be acceptable.

  Now we want to fill this histogram in each event, so we have to count the number of charged particles. For this, initialize a counter before the **particle loop** and increment inside the **particle loop** for each charged particle:
  ```
  int nCh = 0;
  for (int i = 0; i < pythia.event.size(); ++i) {
     if(pythia.event[i].isFinal() && pythia.event[i].isCharged()) {
           nCh++;
     }
  }
  ```
  Then, before the end of the **event loop**, insert
  ```
  mult.fill(nCh);
  ```
  to fill the histogram. Finally, to write out the histograms, after the **event loop** we need a line like
  ```
  cout << mult;
  ```

- As a final standalone exercise, consider plotting the average charged multiplicity over the azimuthal angle difference with respect to the hardest track.

4 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

```
! QCD jet production at the LHC
Beams:idA   = 2212     ! first incoming beam is a 2212, i.e. a proton.
Beams:idB   = 2212     ! second beam is also a proton.
Beams:eCM   = 7000.    ! the cm energy of collisions.
HardQCD:all = on       ! switch on all QCD Dijet processes
```

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]);
```

We now have the beam parameters specified both in the `mymain.cmnd` file and in the main program itself by the line

```
pythia.init( 2212, 2212, 7000.);
```

The arguments to `pythia.init()` will always take precedence over the settings in the input file, so this line should be changed to

```
pythia.init();
```

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.

5 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 = 3` (or other values between 1 and 6)
  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 best tune for LHC is Tune 4C, i.e. `Tune:pp = 5`. This will be the default tune in release `8.150`.

- `MultipleInteractions:bProfile = 3` (or other values between 0 and 3)
  Choice of impact parameter profile for the incoming hadron beams. The default value is

`MultipleInteractions:bProfile = 1`, corresponding to a simple Gaussian matter distribution.
`MultipleInteractions:bProfile = 3`, with the default `MultipleInteractions:expPow = 1` corresponds to an Exponential matter distribution.

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.

5.1 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:MI = off`
  switch off multiple interactions.

For instance, check the importance of FSR, ISR and MI on the charged multiplicity of events by switching off one component at a time.

5.2 Changing the Hard Process

An extensive list of Pythia processes is given in the online manual. It is easy to change the process by replacing e.g. `HardQCD:all = on` with another string. All options available for the desired process are given in the online manual. Processes that are not implemented Pythia-internally can be passed via the Les Houches Interface.

6 Writing HepMC Event Files

The standard HepMC event-record format will be frequently used in subsequent training sessions, with a ready-made installation. However, for the ambitious, here is described how to set up the Pythia interface, assuming you already know where HepMC is installed. Note: the interface to HepMC version 1 is no longer supported; you must use version 2.

To begin with, if you have not already linked to HepMC, you need to go back to the installation procedure of section 2 and redo the steps below step 3.4. Then, do 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;
   ```

It might be good to know that the files `main31.cc` and `main32.cc` are intended as examples to produce HepMC event files.

Further Information and Resources

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.

- One popular misconception is that the energy and momentum of a B meson has to be smaller than that of its mother b quark, and similarly for charm. The fallacy is twofold. Firstly, if the b quark is surrounded by nearby colour-connected gluons, the B meson may also pick up some of the momentum of these gluons. Secondly, the concept of smaller momentum is not Lorentz-frame-independent: if the other end of the b colour force field is a parton with a higher momentum (such as a beam remnant) the "drag" of the hadronization process may imply an acceleration in the lab frame (but a deceleration in the beam rest frame).
  To study this, simulate b production, e.g. the process `HardQCD:gg2bbbar`. Identify B/B* mesons that come directly from the hadronization, for simplicity those with status code -83 or -84. In the former case the mother b quark is in the `mother1()` position, in the latter in `mother2()` (study a few event listings to see how it works). Plot the ratio of B to b energy to see what it looks like.

- One of the characteristics of multiple-interactions (MI) models is that they lead to strong long-range correlations, as observed in data. That is, if many hadrons are produced in one rapidity range of an event, then most likely this is an event where many MI's occurred (and the impact parameter between the two colliding protons was small), and then one may expect a larger activity also at other rapidities.
  To study this, select two symmetrically located, one unit wide bins in rapidity (or pseudorapidity), with a variable central separation Δy: $[\Delta y/2, \Delta y/2+1]$ and $[-\Delta y/2-1, -\Delta y/2]$. For each event you may find n_F and n_B, the charged multiplicity in the "forward" and "backward" rapidity bins. Suitable averages over a sample of events then gives the forward–backward correlation coefficient

$$\rho_{FB}(\Delta y) = \frac{\langle n_F\, n_B\rangle - \langle n_F\rangle\langle n_B\rangle}{\sqrt{(\langle n_F^2\rangle - \langle n_F\rangle^2)(\langle n_B^2\rangle - \langle n_B\rangle^2)}} = \frac{\langle n_F\, n_B\rangle - \langle n_F\rangle^2}{\langle n_F^2\rangle - \langle n_F\rangle^2} ,$$

  where the last equality holds for symmetric distributions such as in pp and $\overline{\mathrm{p}}$p.
  Compare how $\rho_{FB}(\Delta y)$ changes for increasing $\Delta y = 0, 1, 2, 3, \ldots$, with and without MI switched on (`PartonLevel:MI = on/off`) for minimum-bias events (`SoftQCD:minBias = on`).

- Higgs production can proceed through several different production processes. For the Standard Model Higgs some process switches are:
  `HiggsSM:ffbar2H` for $\mathrm{f\overline{f}} \to \mathrm{H}^0$ (f generic fermion, here mainly $\mathrm{b\overline{b}} \to \mathrm{H}^0$);
  `HiggsSM:gg2H` for $\mathrm{gg} \to \mathrm{H}^0$;
  `HiggsSM:ffbar2HZ` for $\mathrm{f\overline{f}} \to \mathrm{H}^0\mathrm{Z}^0$;
  `HiggsSM:ffbar2HW` for $\mathrm{f\overline{f}} \to \mathrm{H}^0\mathrm{W}^\pm$;
  `HiggsSM:ff2Hff(t:ZZ)` for $\mathrm{f\overline{f}} \to \mathrm{H}^0\mathrm{f\overline{f}}$ via $\mathrm{Z}^0\mathrm{Z}^0$ fusion;
  `HiggsSM:ff2Hff(t:WW)` for $\mathrm{f\overline{f}} \to \mathrm{H}^0\mathrm{f\overline{f}}$ via $\mathrm{W}^+\mathrm{W}^-$ fusion;
  `HiggsSM:all` for all of the above (and some more).
  Study the $p_\perp$ and η spectrum of the Higgs in these processes, and compare.

- You can also vary the Higgs mass with a `25:m0 = ...` and switch off FSR/ISR/MI as above for top.

- Z^0 production to lowest order only involves one process, accessible with `WeakSingleBoson:ffbar2gmZ = on`. The problem here is that the process is $\mathrm{f\overline{f}} \to \gamma^*/\mathrm{Z}^0$ with full γ^*/Z^0 interference and so a signficiant enhancement at low masses. The combined particle is always classified with code 23, however. So generate events and study the γ^*/Z^0 mass and $p_\perp$ distributions. Then restrict to a more "Z^0-like" mass range with `PhaseSpace:mHatMin = 75.` and `PhaseSpace:mHatMax = 120.`

- Using your favourite jet cluster algorithm, study the number of jets found in association with the Z^0 above. You can switch off Z0 decay with `23:mayDecay = no`. If you do not have a jet finder around, to begin with you can use the simple `CellJet` one that comes with PYTHIA, see the "Event Analysis" page in the online manual. Again check the importance of FSR/ISR/MI.

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 plaintext 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 information

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.

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]

Sherpa Tutorial

Sherpa is a full-featured event generator which puts its emphasis on an improved description of 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 towards that aim 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 "CKKW merging" is implemented. Information about this merging can be found in the Sherpa publication at arXiv:0811.4622 and in more detail at arXiv:0903.1219 and its references. In addition to these perturbative event phases, Sherpa also has a cluster fragmentation module and hadron decays including QED radiation resummed in the YFS approach.

1 Preparation

Sherpa 1.2.3 is installed in `/mc-school/opt/sherpa1.2.3`. To run it you don't need any special preparation, but it might be handy to add the directory to your `PATH` variable to avoid having to type the full directory name:

```
$ export PATH=/mc-school/opt/sherpa1.2.3/bin:$PATH
```

2 Input File

The way a particular simulation runs in Sherpa is defined by several parameters, which can all be listed in a common file. The default name of this steering file is `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 (`http://sherpa-mc.de/doc/SHERPA-MC-1.2.3.html`), but for now we will discuss a couple of the most important features for the example of Z+jet production at the LHC.

Copy over the run card for this example from `/mc-school/opt/sherpa1.2.3/Examples/LHC_ZJets/Run.dat` and open it in your favourite editor.

Look at the run card to find the following information about the run:

- Beam settings
- Hard scattering process:
  - Which physics process are we running? Which lepton is being produced (Hint: The PDG code for electrons is 11, while the electron neutrino is 12)?
  - Up to how many hard jets are being accounted for by exact matrix elements (Hint: Look for curly brackets)?
  - Are any cuts imposed on the hard scattering (Hint: Look at the "selector" section)? Why?

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

In the "processes" section of the Run.dat file, the hard scattering processes are specified. The particles are identifed by their PDG codes. 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.

3 Simple LHC events

When you run Sherpa for the first time, it will integrate the cross sections. Depending on the hard processes specified in the run card this may take a rather long time. The integration results can be saved and re-used in later runs, for this the directory `Results` has to be created before running Sherpa.

As you have hopefully found out, the example run card above is for production of Drell-Yan events with up to 4 additional jets in the matrix element. Since that is quite a challenging process, for the purposes of this tutorial you should reduce the number of additional jets to just 1 or 2. 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 if you have created it, the directory `Results` should 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
```

Now you can switch to `EVENTS=1` and `OUTPUT=3` and actually look at the structure of one event printed on screen.

4 Writing HepMC Event Files

In the following today and also in the next days, 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 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, but the upshot is:

- `HEPMC2_GENEVENT_OUTPUT=<filename>`: With this keyword the filename's root can be specified, i.e. Sherpa will create files named `<filename>.#.hepmc2g`, where the hash mark numerates the files if events are split into multiple files.
- `FILE_SIZE=<n>`: Number of events per file (default: 1000).
- `EVT_FILE_PATH=<path>`: Directory where the files will be stored

So to write into a fifo-file named `fifo.hepmc2g` you would set:

```
HEPMC2_GENEVENT_OUTPUT=fifo
FILE_SIZE=10000000
```

Together with the information from the beginning of this hand-out you should now be able to create plots from the events you generate with Sherpa. To see the effects of any changes we make in the following the `MC_ZJETS` analysis that comes with Rivet comes in handy. It contains typical observables in the Z+jets process, like the transverse momentum of the Z boson and the jet multiplicity distribution.

5 Changing Default Settings

Here are a few ideas for settings which you can play around with based on the example run card from above. All parameters can be set either on the command line or in the `run` section of the run card.

5.1 Matrix elements and parton shower

- What happens if you use a pure parton shower without ME+PS merging?
  Hint: Simply remove the additional jets in the hard scattering.
- How sensitive are the observables to variations of the unphysical ME+PS merging cut (which is specified in the CKKW line)? Try changing it in a reasonable range (15-60 GeV) around the original 30 GeV. 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.

5.2 Hadronisation and Underlying Event

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
```

5.3 QED radiation

As well as the QCD effects that produce jets, there are also QED effects from radiated photons. In the YFS formalism used by Sherpa, the external lepton lines are dressed with resummed collinear photon radiation. The hardest emission is corrected to the exact matrix element, but the cross section is not affected.

By default, QED radiation is enabled (`ME_QED=On`). To disable it, set

```
ME_QED = Off
```

and check which influence this has on observables like the mass of the Z boson.

In `MC_ZJETS`, the QED radiation in a cone around the lepton has been accounted for in the Z reconstruction. This makes the analysis less susceptible for QED effects. If you are interested in the bare effect of QED radiation, you can enable an additional custom analysis prepared for this tutorial, `MC_ZJETS_NOCLUS`. Here, the Z reconstruction only uses the bare leptons after QED FSR. Since the analysis is not part of the Rivet distribution, you have to let Rivet know where to find it by setting:

```
export RIVET_ANALYSIS_PATH=/mc-school/opt/rivet/more-analyses
```

Note, that you can run both analyses in the same Rivet run.

Further Information and Resources

- Sherpa homepage: `http://sherpa-mc.de`
- Sherpa 1.2.3 manual: `http://sherpa-mc.de/doc/SHERPA-MC-1.2.3.html`
- E-mail us with your questions and comments at `mailto:info@sherpa-mc.de`

The Underlying Event

We have prepared a number of event samples in `hepmc` format. They contain partly minimum bias events and partly events with jets. They have been prepared with various event generators where different options have been used, some better and some worse. The sample files are

```
sampleX1.hepmc.gz
sampleX2.hepmc.gz
sampleX3.hepmc.gz
sampleX4.hepmc.gz
sampleY1.hepmc.gz
sampleY2.hepmc.gz
sampleY3.hepmc.gz
sampleZ1 (various files)
```

All samples can be found in `/mc-school/data`.

All samples contain an underlying event. Your task is to isolate the underlying event and study its properties with the help of a rivet analysis. As a guide and starting point you can use the example skeleton analysis provided on the virtual machine, `/mc-school/ue-tutorial/MY_ANALYSIS.cc`. Please refer to the Rivet guide you received with the school material for technical details.

The following suggestions might be useful:

- Study global properties of the event like η and $p_\perp$ spectra.
- Isolate the underlying event after identifying a leading object in every event.
- Start your study of the underlying event with very inclusive observables like $p_\perp^{\rm sum}$, $N_{\rm ch}$.
- Study more different properties like the $p_\perp$ spectrum of the underlying events.
- Some of your observables might depend on the properties of the leading objects you have isolated before.

If you still have time, you can try to search direct evidence for multiple partonic interactions in the event samples.

The meaning and origin of the samples will only be revealed at the end of the tutorial session.