LUND UNIVERSITY

For tutorials
at Summer Schools

PYTHIA 8 Merging Tutorial

Torbjörn Sjöstrand, Stefan Prestel
Department of Theoretical Physics, Lund University

1 Introduction

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].

2 Getting started: Installing Pythia 8

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.

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.

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.

To produce a matrix-element improved prediction with Pythia 8, you will also need to supply Les Houches Event (LHE) files as input. You will have to generate these yourself, with your own cuts, and your preferred matrix element generator. Pythia 8 comes with three very short LHE files containing 100 events each. You can use these as input for the programs you will develop in this tutorial. If your application requires more events, for the sake of this tutorial, you can find sample LHE files under

`http://home.thep.lu.se/~prestel/LHE_Files/`

The names of the files reflect the hard process:

`w2emveb`:	$pp \to W^- \to e^- \bar{\nu}_e +$ jets
`w2epve`:	$pp \to W^+ \to e^+ \nu_e +$ jets
`z2epem`:	$pp \to Z \to e^+ e^- +$ jets
`z2veveb`:	$pp \to Z \to \nu_e \bar{\nu}_e +$ jets

and the number immediately before the `.lhe` suffix gives the number of additional partons in the LHE file. These files come with no guarantees and should only be used for this tutorial.

To download *all* sample LHE files to the examples directory, go to

`/path/to/Pythia/pythia8160/examples`

and (under Linux) execute

```
wget -r -np -nd --reject "index*" http://home.thep.lu.se/~prestel/LHE_Files/
```

in your terminal.

3 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 $pp \to W + \text{jets}$ 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).
  // Generate an(other) event. Fill event record.
  pythia.next();

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

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.

4 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 `/path/to//Pythia/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[1]. 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.

5 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 $pp \to Wj$ matrix element (ME), while soft jets should be modelled by emissions off $pp \to W$ 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 $pp \to Wj$, and with $pp \to W$ 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.
  ...
```

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

```
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. Once you request the generation of a number of events comparable to the number stored in the LHE files[2], you need to implement these commands.

- So far, your adaptions mean that PYTHIA 8 will shower the pp $\to$ W process and the pp $\to$ 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 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 = 20.");
```

  at the beginning of the program. Note that t_{MS} has dimension GeV. Here, we have chosen a value of $t_{\text{MS}} = 20$ GeV since the example LHE files have been generated with this cut – no physics is tied to this particular value[3].

- Then, decide on the maximal number of additional jets available from ME calcula-

[2]The LHE files that can be downloaded from `http://home.thep.lu.se/~prestel/LHE_Files/` contain 100 000 events for each jet multiplicity.

[3]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.

tion, 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[4]. 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.

6 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,

[4]This will not double-count contributions: The removal of double counting exactly what merging procedures are intended for.

this does not allow for a reasonable analysis, but gives a hint how easily your program can be generalised to multi-jet merging.

7 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 scale definition used to regularise ME
Merging:doKTMerging = on
! Merging scale value (= cut value in ME)
Merging:TMS         = 20
! Number of additional jets available from ME
Merging:nJetMax     = 1
! Process to be merged
Merging:Process     = pp>e+ve
```

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.

8 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?

8.1 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.

9 Additional challenges

If you have finished these challenges, 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].

9.1 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.

9.2 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[5]. These files have been regularised by requiring the cuts

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

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?

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

- Changing the default choices in the merging procedure[6]. For example, check `Merging:unorderedScalePrescrip`. How are observables influenced by your choices?

- Define jets with a jet algorithm, by e.g. using the `Pythia::Slowjet` class. How does the ΔR_{ij} separation between jets change when including additioal jets?

9.3 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.

If you want to want to link your own installation of Pythia 8 to HepMC, please consult Section 2, 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;
   ```

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

[6]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.

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.

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`