For tutorials
at Summer Schools
or for self-study
February 2024

PYTHIA 8.3 Worksheet

1 Introduction

The Pythia 8.3 program is a standard particle physics tool for the generation of high-energy collisions, comprising a coherent set of physics models for the evolution from a few-body high-energy ("hard") scattering process to a complex multihadronic final state. The particles are produced in vacuum. Simulation of the interaction of the produced particles with detector material is not included in Pythia but can, if needed, be done by interfacing to external detector-simulation codes.

The Pythia 8.3 code package contains a library of hard interactions and models for initial- and final-state parton showers, multiple parton-parton interactions, beam remnants, string fragmentation and particle decays. It also has a set of utilities and interfaces to external programs, notably to allow the input of a wider range of hard interactions.

The objective of this exercise is to teach you the basics of how to use the Pythia 8.3 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 neither the physics models used in the program nor the full range of methods and options.

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. It does presuppose a working Unix-style environment with C++ compilers and the like; check Appendix D if in doubt. This section also introduces the main sources of further information.

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. Section 3 gives you a simple step-by-step recipe how to write a minimal main program, that can then gradually be expanded in different directions, e.g. as in Section 4.

In Section 5 you will 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 final four sections provide suggestions for optional further studies, and can be addressed in any order. Section 6 studies Z′ production, with particular emphasis on jet properties. Section 7 describes how you can explore various physics aspects of the Standard Model Higgs production and decay. Section 8 deals with the important topic of merging of external matrix-element input of different orders, introducing the CKKW-L scheme as a suitable starting point. Section 9, finally, collects suggestions for a few diverse studies.

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 provided for the interested in Appendix C. Further main programs included with the PYTHIA code provide examples of linking, e.g. to MADGRAPH, POWHEG, LHAPDF, FASTJET, ROOT, and the Les Houches Accords LHEF and SLHA.

Appendix A contains a brief summary of the event-record structure, and Appendix B some notes on simple histogramming and jet finding. Appendices C and D have already been mentioned.

2 Installation and documentation

Denoting a generic PYTHIA 8 version `pythia83xx` (at the time of writing `xx` = 12), here is how to install PYTHIA 8 on a Linux/Unix/macOS system as a standalone package (assuming you have standard Unix-family tools installed, see Appendix D).

1. In a browser, go to
   `https://pythia.org`
2. Download the (current) program package
   `pythia83xx.tgz`
   to a directory of your choice (e.g. by right-clicking on the link).
3. In a terminal window, `cd` to where `pythia83xx.tgz` was downloaded, and type
   `tar xvfz pythia83xx.tgz`
   This will create a new (sub)directory `pythia83xx` where all the PYTHIA source files are now ready and unpacked.
4. Move to this directory (`cd pythia83xx`) and do a `make`. This will take ∼3–15 minutes on a single core, depending on your computer. If it has n cores you can use `make -j`n to speed up the compilation accordingly. The PYTHIA 8 library is now compiled and ready for physics.
5. For test runs, `cd` to the `examples/` subdirectory. An `ls` reveals a list of programs, `mainNNN.cc`. These example programs illustrate different aspects of PYTHIA 8. For a list of what they do, see the “Sample Main Programs” page in the “Getting

Started" section of the online manual (see further below).

Initially only use one or two to check that the installation works, preferably among the first few ones, since several higher-number ones require further program libraries to be linked. Once you have worked your way through the introductory exercises in the next sections, you can return and study more programs and their output.

To execute one of the test programs, do

```
make mainNNN
./mainNNN
```

The output is directed to the terminal, `stdout`. To save the output to a file instead, do `./mainNNN > mainNNN.log`, after which you can study the test output at leisure by opening `mainNNN.log`. See Appendix A for an explanation of the event record that is listed for the first event in almost all of the runs.

There are three main Pythia documentation sources.

- The **online manual**: if you use a web browser to open the file

  ```
  pythia83xx/share/Pythia8/htmldoc/Welcome.html
  ```

  you will gain access to the online manual, where all available methods and settings are described. A setting can be a boolean, an integer or a real number, or a text string, for which the user can set its value. Use the left-column index to navigate among the topics, which are then displayed in the larger right-hand field. The triangle symbols ▶ are used to expand a section.

  While the effect of some settings is obvious, it is less so for others. A deeper understanding then may require that you read the relevant sections of the physics guide (see next), in conjunction with the points raised in the online manual itself.

  It may be useful to note that the `.html` files are built from the corresponding `.xml` ones. The latter are the ones read in at the beginning of a run to specify all settings, including particle data, and their default values. This is intended to ensure that the documentation is always in step with the default data used by the code.
- The **physics guide**: the article "A comprehensive guide to the physics and usage of Pythia 8.3" [1] provides a detailed presentation of the physics behind the Pythia 8.3 code, plus an overview of more practical aspects. Given that the physics description alone is around 200 pages, it contains more material than you can take in one sitting, but it allows you to study up on areas of specific interest. The relevant references to the original literature allows you to dive even deeper. The more practical part has some overlap with the online manual, but does not go into the details of all methods and settings.

  The full text is linked to the online manual above, as the "Introduction" of the "Separate Documents" section, but note that it has been submitted to arXiv and published in SciPost [1]. This should always be your main reference when you use Pythia 8 for an article of yours, but original literature of special relevance should not be forgotten, whether written by Pythia authors or by others.
- A **Doxygen representation** of the code is also available, not as part of the distribution but available at

  ```
  https://pythia.org/latest-doxygen
  ```

  The code proper does not contain any specific Doxygen instructions, so this representation is obtained by an automatic extraction of information on classes and methods, including the comments for code sections. It provides less detailed information than the other two, but may be a more convenient way to navigate the code structure for experts.

3 A “Hello World” program

We will now generate a single $gg \to t\bar{t}$ event at the LHC, using Pythia standalone.

Open a new file `mymain01.cc` in the `examples` subdirectory with a text editor, e.g. Emacs. Then type the following lines (here interspersed with superfluous explanatory comments, the `//` parts):

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

int main() {                // Begin main program.

  // Set up generation.
  Pythia pythia;            // Declare a Pythia object
  pythia.readString("Top:gg2ttbar = on"); // Switch on process.
  pythia.readString("Beams:eCM = 8000."); // 8 TeV CM energy.
  pythia.init(); // Initialize; incoming pp beams is default.

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

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

The `examples/Makefile` has been set up to compile all `mymainNN.cc`, `NN` $= 01 - 99$, and link them to the `lib/libpythia8.a` library, just like the `mainNNN.cc` ones. Therefore you can compile and run `mymain01` as before:

```
make mymain01
./mymain01.exe > mymain01.log
```

If you want to pick another name, or if you need to link to more libraries, you have to edit `examples/Makefile` appropriately.

Thereafter you can study `mymain01.log`, especially the example of a complete event record (preceded by initialization information, and by kinematical-variable and hard-process listing for the same event). 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 top quark produced in the hard interaction, gg → $t\bar{t}$, initially with positive status code. When later a shower branching t → tg occurs, the new t and g are added at the bottom of the then-current event record, but the old t 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 the top. After the shower, when the final top decays, t → bW^+, also that copy receives a negative status code. When you understand the basic principles, see if you can find several copies of the top quarks, 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 `mymain01` program from above, towards what would be needed for a more realistic analysis setup.

- Often, we wish to mix several processes together. To add the process $q\bar{q} \to t\bar{t}$ to the above example, just include a second `pythia.readString` call
```
pythia.readString("Top:qqbar2ttbar = on");
```
- Now we wish to generate more than one event. To do this, introduce a loop around `pythia.next()`, so the code reads
```
for (int iEvent = 0; iEvent < 5; ++iEvent) {
  pythia.next();
}
```
  Hereafter, we will call this the *event loop*. The program will generate 5 events. Each call to `pythia.next()` resets the event record and fills it with a new event. To list more of the events, you also need to add
```
pythia.readString("Next:numberShowEvent = 5");
```
  along with the other `pythia.readString` commands.
- To obtain statistics on the number of events generated of the different kinds, and the estimated cross sections, add a
```
pythia.stat();
```
  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 (except for a few special cases). The above-mentioned `pythia.stat()` will then tell you how many times each problem was encountered over the entire run.

- Studying the event 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.1). The `cout` statement, therefore, will give a list of the PDG code of every particle in the event record.
- As mentioned above, the event listing contains all partons and particles, traced through a number of intermediate steps. Eventually, the top will decay (t → Wb), and by implication it is the last top copy in the event record that defines the definitive top production kinematics, just before the decay. You can obtain the location of this final top e.g. by inserting a line just before the particle loop
  ```
  int iTop = 0;
  ```
  and a line inside the particle loop
  ```
  if (pythia.event[i].id() == 6) iTop = i;
  ```
  The value of `iTop` will be set every time a top is found in the event record. When the particle loop is complete, `iTop` will now point to the final top in the event record (which can be accessed as `pythia.event[iTop]`).
- 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[iTop].pT()`, and pseudorapidity, `pythia.event[iTop].eta()`. Use these methods to print out the values for the final top found above.
- We now want to generate more events, say 1000, to view the shape of these distributions. Inside PYTHIA is a very simple histogramming class, see Appendix B.1, that can be used for rapid check/debug purposes. To book the histograms, insert before the event loop
  ```
  Hist pT("top transverse momentum", 100, 0., 200.);
  Hist eta("top pseudorapidity", 100, -5., 5.);
  ```
  where the last three arguments are the number of bins, the lower edge and the upper edge of the histogram, respectively. Now we want to fill the histograms in each event, so before the end of the event loop insert
  ```
  pT.fill( pythia.event[iTop].pT() );
  eta.fill( pythia.event[iTop].eta() );
  ```
  Finally, to write out the histograms, after the event loop we need a line like
  ```
  cout << pT << eta;
  ```
  Do you understand why the η distribution looks the way it does? Propose and study a related but alternative measure and compare.
- As a final standalone exercise, consider plotting the charged multiplicity of events.

You then need to have a counter set to zero for each new event. Inside the particle loop this counter should be incremented whenever the particle `isCharged()` and `isFinal()`. 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.

5 Input files

With the `mymain01.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 `mymain01.cc` example program. Open a new file, `mymain01.cmnd`, and input the following

```
! t tbar 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 = 8000.    ! the cm energy of collisions.
Top:gg2ttbar = on    ! switch on the process g g -> t tbar.
Top:qqbar2ttbar = on ! switch on the process q qbar -> t tbar.
```

The `mymain01.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. Cut-and-paste of variable names can be used to avoid spelling mistakes.

Any command that begins with a number is for changes in the `ParticleData` database, where the initial number is the PDG identity code of the particle. An initial letter instead is for changes in the `Settings` database, which contains everything else.

The final step is to modify our program to use this input file. Its name can be hardcoded in the main program, but it can also be provided as a command-line argument for more flexibility. 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]);
```

The executable `mymain01` is then run with a command line like

```
./mymain01 mymain01.cmnd > mymain01.log
```

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) {
```

You are now free to play with further options in the input file, such as:

- `6:m0 = 175.`
  change the top mass, which by default is 173 GeV.
- `PartonLevel:FSR = off`
  switch off final-state radiation.
- `PartonLevel:ISR = off`
  switch off initial-state radiation.
- `PartonLevel:MPI = off`
  switch off multiparton interactions.
- `Tune:pp = 20` (or other values between 1 and 32)
  different combined tunes, in particular to radiation and multiparton 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.
- `Random:setSeed = on`
  `Random:seed = 123456789`
  all runs by default use the same random-number sequence, for reproducibility, especially useful during development and debug, but you can pick any number between 1 and 900,000,000 to obtain a unique sequence.

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

The possibility to use command-line input files is further illustrated in many of the `mainNNN.cc` programs, e.g. `main111.cc`, `main132.cc` and `main231.cc`.

You have now completed the core part of the worksheet — congratulations! From now on you should be able to take off in different directions, depending on your interests. The following four sections contain examples of further possible studies, and can be addressed in any order.

6 Jets and other event properties

The purpose of this exercise is to study further how different physics aspects affect event properties, notably jet production. As a simple example, assume that we want to study the properties of a Z′ gauge boson, i.e. more-or-less a heavier copy of the normal Z^0 one. Such particles have been proposed in various scenarios for physics beyond the Standard

Model. The case of a 1 TeV mass has already been ruled out at the LHC, but it offers a good way to illustrate the importance of parton showers, multiparton interactions and hadronization.

To begin, open the `main216.cc` file in your `examples` directory. Have a look at it, to check that you understand the overall flow of the code. The process used is $\mathrm{q}\overline{\mathrm{q}} \to \gamma^*/\mathrm{Z}^0/\mathrm{Z}'^0 \to \mathrm{q}'\overline{\mathrm{q}}'$, with full $\gamma^*/\mathrm{Z}^0/\mathrm{Z}'^0$ interference, but you can concentrate on the Z' peak region by restricting the generated mass window. The Z' identity code is 32, and its allowed decay channels apply for this process at all mass scales. By switching off all its decay modes, and then switching back on those that contain a $\mathrm{d}, \mathrm{u}, \mathrm{s}, \mathrm{c}$ or b one, decays are restricted to "light" quarks, as viewed in relation to the Z' mass scale.

Compile and run the program as before. Study `main216.log` output, notably the histograms at the end (see Appendix B.1). Identify average number and spread, and note other properties, such as how often two jets are (not) found. Do you understand roughly what is going on?

If you have Python 3 and Matplotlib (see beginning of Appendix D) installed, you can also do

```
python3 plot216.py
```

and open `fig216.pdf` for a nicer representation of the histograms, but without the statistics information explicit.

Once you are familiar with the basic setup, we can start to do variations around it, to study various effects. You may not have time for all of them, and do not need to do them in the order listed below. On the contrary, if you do this together with others, it would be useful if you branch out in different directions, using the various suggestions as inspiration. Then compare results and reason whether they make sense, notably that effects go in the expected direction.

Bookkeep key results, at least by tabulating the average and spread of the two-jet mass for each run, and optionally by (re)naming the PDF files so that they are not overwritten. Consider the statistical significance of the difference between scenarios. You may want to increase the number of events generated, but for this exercise not so much that it becomes the limiting factor for the studies.

- Make use of further `pythia.readString("...");` commands to switch off one or more aspects of the full event generation, notably
  `PartonLevel:ISR = off` for no initial-state radiation,
  `PartonLevel:FSR = off` for no final-state radiation,
  `PartonLevel:MPI = off` for no multiparton interactions, and
  `HadronLevel:all = off` for no hadronization.
  How does each of these affect the average reconstructed mass?
- Instead switch all off as a starting point and then add back on one component at a time, and again quantify results.
- How would matters change for a different Z' mass? Use e.g. 200 GeV and 5 TeV as two extremes. Remember also to modify the Z' mass limits, the histogram upper limit, and the `jetpTmin` scale in proportion (more or less).
- How would properties change if the jets were made narrower or broader, say

`jetRadius` 0.4 or 1.0? Similarly if their minimal transverse momentum `jetpTmin` is modified? You could even switch from the anti-$k_\perp$ algorithm to the $k_\perp$ one, `jetPower = 1`, or the Cambridge/Aachen one, ditto `= 0`.

- Histogram the number of found jets. What would the effect be if all found jets are included in the invariant mass calculation for all events with at least two jets?
- The decay to top is not included above, since event topologies are much more complicated there. Nevertheless, do a run with t = 6 as the only `32:onIfAny` decay product.
- Study other event properties. For instance, you can plot the charged multiplicity of events, cf. `main101.cc`, and how it varies when different components are switched on/off. Or similarly the Z′ $p_\perp$, but then for the last Z′ copy in the event, after all effects have been added to it, cf. `main102.cc`.
- If you want to overlay curves from different conditions, it is feasible to have a big `for` loop, within which a new `Pythia` instance is created and destroyed with new settings each time. Preferably the booking of histograms would come before this big loop, and the plotting code after it.

Most of the variations above can be combined, e.g. to study a narrow/broad jet radius at a low Z′ mass.

A good reference to learn more about the theory of jet algorithms is [10].

7 Some studies of Higgs production

The discovery of the Higgs boson has been the main accomplishment of the LHC to date. Generators have been part of that story, right from the early days when the ATLAS and CMS detectors were designed in such a way as to permit that discovery. This section offers exercises intended to explore the physics of Higgs production from various aspects, with Pythia as guide.

7.1 Production kinematics

The dominant production channel is gg → H^0. To study the kinematics distribution of the Higgs, the existing top production program could easily be modified. Instead of switching on top production, use

```
    HiggsSM:gg2H = on
```

And instead of looking for the last top copy before decay, look for the last Higgs copy `iH`, ie. the last particle with `id() == 25`. Once found the `pythia.event[iH]` methods can be used to extract and histogram Higgs kinematics distributions, like for the top. In addition to the transverse momentum `pT()` you can compare the distributions for true rapidity `y()` and for pseudorapidity `eta()`.

7.2 Production processes

While gg $\to$ H^0 is the main Higgs production channel, it is not the only one. Do a run with `HiggsSM:all = on` to check which are implemented and their relative importance. Also figure out how you could generate one of the less frequent processes on its own, either from the online manual or by making some deductions from the output with all processes.

In order to get decent cross-section statistics faster, you can use `PartonLevel:all = off` to switch off everything except the hard-process generation itself. One price to pay is that the kinematics distributions for the Higgs are not meaningful.

If instead complete events are generated you can study how the transverse-momentum distribution varies between processes. What are the reasons behind the significant differences?

7.3 Decay channels

Also the decay channels and branching ratios in the Higgs decay are of interest. Here no ready-made statistics routines exist, so you have to do it yourself. You already have the location `iH` of the decaying Higgs. Since the standard decay modes are two-body, their locations are stored in

```
int iDau1 = pythia.event[iH].daughter1();
int iDau2 = pythia.event[iH].daughter2();
```

and from that you can get the identities of the daughters. Introduce counters for all the decay modes you come to think of, that you use to derive and print branching ratios. Print the daughter identities in the leftover decays, where you did not yet have any counters, and iterate until you catch it all.

In this case, `PartonLevel:all = off` cannot be used, since then one does not get to the decays, at least not with the information in `pythia.event`. But you can combine

`PartonLevel:ISR = off` to switch off initial-state radiation,
`PartonLevel:FSR = off` to switch off final-state radiation,
`PartonLevel:MPI = off` to switch off multiparton interactions, and
`HadronLevel:all = off` to switch off hadronization and decays,

to get almost the same net time saving.

7.4 Mass distribution

By now the Higgs mass is pretty well pinned down, but you can check what the branching ratios would have been with another mass, e.g. by `25:m0 = 150.` for a 150 GeV Higgs.

On a related note, the Higgs mass is generated according to a Breit-Wigner distribution, convoluted with parton densities. Can you resolve this shape for the default Higgs mass? How does it change had the Higgs been heavier, say at 400 GeV?

7.5 Associated jets

Let us return to the different production channels, which give different event characteristics. Well-known is that the $\mathrm{qq} \to \mathrm{qqH}^0$ processes give rise to jets at large rapidities, that can be used for tagging purposes. This can be studied as follows.

The two relevant processes are `HiggsSM:ff2Hff(t:ZZ)` and `HiggsSM:ff2Hff(t:WW)` for $\mathrm{Z}^0\mathrm{Z}^0$ and $\mathrm{W}^+\mathrm{W}^-$ fusion (f here denotes a fermion; the same processes also exist e.g. at $\mathrm{e}^+\mathrm{e}^-$ colliders), that are to be compared with the standard $\mathrm{gg} \to \mathrm{H}^0$ one.

Find jets using the `SlowJet` class, see Appendix B.2, e.g. using the anti-$k_\perp$ algorithm with $R = 0.7$, $p_{\perp\mathrm{min}} = 30$ GeV and $\eta_{\mathrm{max}} = 5$.

One problem is that also the Higgs decay products can give jets, which we are not interested in here. To avoid this, we can switch off Higgs decays by `25:mayDecay = off`. This still leaves the Higgs itself in the event record. A call `pythia.event[iH].statusNeg()` will negate its status code, and since `slowJet.analyze(...)` only considers the final particles, i.e. those with positive status, the Higgs is thus eliminated.

Now study the $p_\perp$ and rapidity spectrum of the hardest jets, and compare those distributions for the two processes. Also study how many jets are produced in the two cases.

7.6 Underlying event

Several mechanisms contribute to the overall particle production in Higgs events. This can be studied e.g. by histogramming the charged particle distribution.

You then need to have a counter set to zero for each new event. Inside the particle loop this counter should be incremented whenever the particle `isFinal()` and `isCharged()`. 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, for instance.

Once you have the distribution down for normal events, study what happens if you remove ISR, FSR and MPI one by one or all together. Also study the contribution of the Higgs decay itself to the multiplicity, e.g. by setting the Higgs stable. Reflect on why different combinations give the pattern they do, e.g. why ISR on/off makes a bigger difference when MPI is on than off.

7.7 Decay properties

The decay mode $\mathrm{H}^0 \to \mathrm{Z}^0\mathrm{Z}^0 \to \ell^+\ell^-\ell'^+\ell'^-$, $\ell, \ell' = \mathrm{e}, \mu$, is called the gold-plated one, since it stands out so well from backgrounds. It also offers angular correlations that probe the spin of a Higgs candidate.

For now consider a simpler, but still interesting, pair of distributions: the mass spectra of the two Z^0 decay products. Plot them for the lighter and the heavier of the two separately, and compare shapes and average values. To improve statistics, you can use `25:onMode = off` to switch off all decay channels, and then `25:onIfMatch = 23 23` to switch back on the decay to Z^0Z^0 (and nothing else). Further, neither ISR, FSR, MPI nor hadronization affect the mass distributions, so this allows some speedup.

How can one qualitatively understand why the two masses tend to be so far apart, rather than roughly comparable?

7.8 Comparison with Z production

One of the key reference processes at hadron colliders is Z^0 production. To lowest order it only involves one process, $q\overline{q} \to \gamma^*/Z^0$, accessible with `WeakSingleBoson:ffbar2gmZ = on`. One complication is that the process involves γ^*/Z^0 interference, and so a significant enhancement at low masses, even if the combined particle always is classified with code 23, however.

Compare the two processes $gg \to H^0$ and $q\overline{q} \to \gamma^*/Z^0$, with respect to the $p_\perp$ distribution of the boson and the total charged multiplicity of the events. So as to remove the dependence on the difference in mass, you can set a specific mass range in the γ^*/Z^0 generation with `PhaseSpace:mHatMin = 124` and `PhaseSpace:mHatMax = 126`, to agree with the H^0 mass to ± 1 GeV.

Can you explain what is driving the differences in the $p_\perp$ and n_{chg} distributions between the two processes?

8 CKKW-L merging

The main programs we have constructed and studied in the previous sections have one common drawback: all start from the Pythia 8 internal library of lowest-order processes, and then add higher-order corrections entirely by the internal parton-shower machinery. This will give reliable results for soft and collinear configurations, but less so for multiple hard, well-separated jets. To model the latter similarly well we need to include external input from higher-order calculations, at least at tree level, but where feasible also at one-loop level. A number of different external programs can provide such input, using the LHA/LHEF standard format [2, 3, 4] to transfer information, usually as LHE files. The hard-process events stored in these files will be accepted or rejected in such a way that double counting between different parton multiplicities is removed, resulting in a smooth transition between the multiplicities, and between the external input and the internal handling of parton showers. These two tasks usually go hand in hand.

Many different schemes have been proposed for matrix element + parton shower merging (MEPS), and a comprehensive selection of such schemes is available with the Pythia 8 distribution, including

- leading-order merging: MLM jet matching [5] (MadGraph- or AlpGen-style), CKKW-L merging [6], and unitarised ME+PS merging (UMEPS) [7]; and
- next-to-leading order merging: NL^3 merging and unitarised NLO+PS merging (UNLOPS) [8].

The setup of such merging schemes is documented in the online manual on the page "Matching and Merging" with further subpages, and is illustrated in several of the example main programs.

Here we will experiment with the CKKW-L scheme, which was the first merging scheme available in PYTHIA 8, and also is among the simpler to work with. We will take the `main162` example main program with the `main162ckkwl.cmnd` input file as a starting point for our studies. In its general structure `main162.cc` closely resembles the main program(s) we already constructed step by step, so we will only need to comment on aspects that are new for the merging game. The process $W^\pm + \leq 2$ jets will be taken as an example. It uses the LHE files

`w_production_tree_0.lhe` for $W^\pm + 0$ partons
`w_production_tree_1.lhe` for $W^\pm + 1$ parton
`w_production_tree_2.lhe` for $W^\pm + 2$ partons

in the `examples` directory to produce a result that simultaneously describes $W^\pm + 0, 1, 2$ jet observables with leading-order matrix elements while also including arbitrarily many shower emissions. Jets are here defined by a clustering procedure on the partons thus generated. (We omit other effects from consideration, such as MPIs or hadronization.)

Say we want to study a one-jet observable, e.g. the transverse momentum of the jet j in events with *exactly* one jet. In this case, we want to take "hard" jets from the $pp \to Wj$ matrix element (ME), while "soft" jets should be modelled by parton-shower (PS) emissions off the $pp \to W$ states. In order to smoothly merge these two samples, we have to know in which measure "hard" is defined, and which value of this measure separates the hard and soft regions. In `main162ckkwl.cmnd`, these definitions are

```
Merging:doPTLundMerging = on
Merging:TMS             = 15.
```

This will enable the merging procedure, with the merging scale defined by the shower evolution variable $p_\perp$ (called `pTLund` in the merging framework and in `MadGraph5`) with a merging scale $t_{\mathrm{MS}} = 15$ GeV. This definition fixes what we mean when we talk about "hard" and "soft" jets:

Hard jets: $\min\{\text{any } p_\perp \text{ between sets of partons}\} > t_{\mathrm{MS}}$
Soft jets: $\min\{\text{any } p_\perp \text{ between sets of partons}\} < t_{\mathrm{MS}}$

Thus, in order for the merging prescription to work, we need to remove phase space regions with $\min\{\text{any } p_\perp\} < t_{\mathrm{MS}}$ from the W + 1-parton matrix element calculation. Otherwise, there would be an overlap between the "soft jet" and "hard jet" samples.

This requirement means that the merging-scale definition should be implemented as a *cut in the matrix element generator*. Alternatively, it is possible to enforce the cut in PYTHIA 8 internally, assuming that the ME is calculated with more inclusive (i.e. loose) cuts. This is illustrated in Figure 1, in which the triangle depicts the whole phase space, with soft or collinear divergences located on the edges. The yellow area symbolises the phase-space region used for the generation of the LHEF events, while the green area represents the phase space after PYTHIA 8 has enforced the merging-scale cut on the input events. In order to correctly apply the merging-scale cut, the green area has to be fully contained inside the yellow one, i.e. the cut in the ME generator has to be more inclusive than the t_{MS}-cut. For optimal efficiency, the yellow and green areas should be identical. This can be

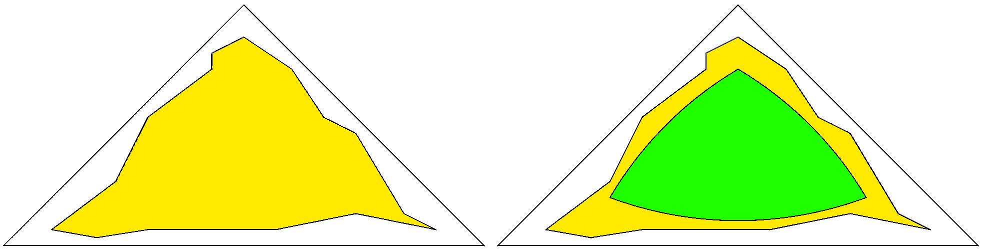

Figure 1: Schematic illustration of how the phase space covered by the external matrix-element generator, yellow region, has to enclose the region passing the Pythia 8 cuts, green region.

the case in MadGraph5 [9], when using the generation cuts `ktdurham` (corresponding to `Merging:doKTMerging = on`) and `ptlund` (corresponding to `Merging:doPTLundMerging = on`).

After the merging-scale definition, we define the underlying process. To tell Pythia 8 that we want to merge additional jets in W-boson production, we specify which is the core process, using MadGraph5 notation without spaces, where the final state is defined by the $W^{\pm}$ decay products rather than by the $W^{\pm}$:

```
Merging:Process          = pp>LEPTONS,NEUTRINOS
```

in `main162ckkwl.cmnd`. Since we include both W^{+} and W^{-} in the LHE files, we use the wildcards `LEPTONS` and `NEUTRINOS` to mean leptons and antileptons as well as neutrinos and antineutrinos, respectively. Finally, the setting

```
Merging:nJetMax          = 2
```

tells the program to include the pre-generated ME events for up to two additional jets.

In `main162.cc`, the CKKW-L input file `main162cckwl.cmnd` is read early on by the `pythia.readFile(...)` command. This gives access to the number of events to be read from each LHE file through `Main:numberOfEvents` and the number of LHE files to be processed via `Main:numberOfSubruns`. The *subrun loop* then handles each LHE file, one at a time. Specifically, the

```
pythia.readFile(cmndFile, iMerge);
```

uses the `iMerge` argument when reading the `main162ckkwl.cmnd` file, so that only those commands following the respective `Main:subrun = iMerge` label are read. (Plus that everything before the first `Main:subrun` is re-read, but that does not matter since it stays the same.) Thus the proper LHE file is picked up for each jet multiplicity. The

```
Beams:frameType          = 4
```

also informs Pythia that beam parameters should be read from the header section of the LHE file, and not set by the user.

Then we enter the event loop. The already-discussed difference in phase-space coverage can lead to a fair fraction of all input events being rejected. Thus the number of produced events can be lower than the requested `Main:numberOfEvents` one if the file is not large enough. (When no further events can be read the `pythia.next()` command will return `false`, so that the event loop can be exited at the end of the LHE file.) Those events that survive come with a weight

```
double weightMerging = pythia.info.mergingWeight();
```

which contains Sudakov factors (to remove the double counting between samples of different multiplicity), α_s ratios (to incorporate the α_s running not available in matrix element generators), and ratios of parton distributions (to include variable factorization scales). If the flag `Merging:includeWeightInXsection` is set, the weight is instead found in

```
double weight = pythia.info.weight();
```

In order to always get the weight, it is recommended to use the product of `weight()` and `mergingWeight()`. This weight *must* be used when filling histogram bins, as is e.g. done by

```
pTWnow.fill(pTW, weight);
```

for the $p_\perp$ of the W boson. The sum of weights also goes into the calculation of the total generated cross section.

After the event loop, the contribution to the $p_\perp$ of the W boson from this particular multiplicity is normalised by

```
pTWnow *= MB2PB*norm/binWidth;
```

where `norm` includes the weight per event,

```
norm = sigmaInc/pythia.info.nAccepted();
```

the variable `MB2PB` is a factor of 10^9 to convert from mb to pb, and `binWidth` compensates for the bin width to yield cross section per GeV. This number and more detailed statistics are printed to the terminal. As a final step, the contribution of the current subrun is added to the total histogram

```
pTWsum += pTWnow;
```

and the subrun loop begins over with the next LHE file. The complete histogram, combining all multiplicities, is printed after the sub-run loop has concluded.

You can compile `main162.cc` by

```
make main162
```

and run CKKW-L merging with

```
./main162 main162ckkwl.cmnd
```

When you run the program, note that some warning messages are issued routinely as part of the merging machinery, in the steps where a clustering history is found and where it is decided whether an event fails the merging scale cuts. Warnings from the SLHA interface also are irrelevant. So no reason to worry about any of that.

After the first run with the main program as is, you can try different variations.

- Convince yourself that the variation of the "merging weight" is moderate.
- Check in which $p_\perp$ regions which jet multiplicity contributes most.
- Study how the individual contributions and the sum changes when you run with a maximum of 1 or 0 jets, instead of the default 2.
- Compare the $p_\perp$ spectrum of the W with what you get from running the internal PYTHIA production process, by straightforward modifications of your `mymain01` program.
- A major limitation is the size of the event files that come with the standard PYTHIA distribution, for storage reasons. If you have a decent internet connection you can download larger files, with 100 000 events for each multiplicity up to W + 4 partons. Do this from the PYTHIA home page, near the bottom of the start page,

files `wp_tree_0.lhe.gz` through `wp_tree_4.lhe.gz`. You should `gunzip` them after downloading. (It is also possible to configure PYTHIA so that it can read `gzipped` files, see the `README` in the PYTHIA main directory, but this is less trivial.) With these files you can repeat the exercise above, and in particular check how much is gained by including the further W + 3 and W + 4 samples. Note that you have to change the `Beams:LHEF` inputs in the respective subruns to reflect the names of the `wp_tree_*.lhe` files. Since these files were generated with a $k_\perp$ cut of $k_{\perp,\mathrm{cut}} = 30$ GeV and $D = 0.4$, you also need to set `Merging:doKTMerging = on` (and `Merging:doPTLundMerging = off`) and `Merging:DParameter = 0.4`. To run through all events in the files takes a while, so check with a fraction of the sample to begin with, and be prepared to do something else while you wait for the full run to complete.

- Alternatively, if you are already familiar with MADGRAPH5, you could generate your own LHE files and merge them. This would take some time, however, in particular for the higher multiplicities, so may not be an option.
- Use a jet finder to analyse the final state, and plot the $p_\perp$ spectra for the first, second, and third hardest jets, combining separate contributions similarly to what is done in `main162.cc` for the W $p_\perp$ spectrum. Instructions how to use the built-in `SlowJet` jet finder can be found in Appendix B.2.
- Check the variation of merged predictions with t_{MS}. You can do this by using an "inclusive" event sample, and having PYTHIA enforce a stronger t_{MS} cut. In which phase-space region is the t_{MS} variation most visible?
- Switch between the "wimpy" and "power" options for maximal shower scales by choosing values for `TimeShower:pTmaxMatch` and `SpaceShower:pTmaxMatch`. Are the effects more visible in merged or non-merged predictions?

Once you have familiarised yourself with the example, you can experiment with more advanced settings in `main162ckkwl.cmnd`. Alternatively, you can also try other merging schemes and check how the results compare to CKKW-L merging, e.g. unitarised leading-order merging (UMEPS) as illustrated in `main162umeps.cmnd`, or unitarised next-to-leading order merging (UNLOPS) as illustrated in `main162cmnd.cc`. Both input files work with the `main162.cc` main program that you have used for CKKW-L merging above. To see how to implement a user-defined merging scale function, take a look at `main161.cc`. Finally, you can try `main164.cc`, which is intended as the matching and merging frontend for PYTHIA and can be run with several `.cmnd` files for different matching/merging options. The `main164.cc` example produces HEPMC events that can either be used directly in RIVET during the run or saved in a file to be used in standard analysis and plotting tools after the run.

9 Further studies

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 multiparton-interactions (MPI) 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 MPI'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 MPI switched on (`PartonLevel:MPI = on/off`) for minimum-bias events (`SoftQCD:minBias = on`).
- Z^0 production to lowest order only involves one process, which is accessible with `WeakSingleBoson:ffbar2gmZ = on`. The problem here is that the process is $\mathrm{f}\overline{\mathrm{f}} \to \gamma^*/\mathrm{Z}^0$ with full γ^*/Z^0 interference and so a significant 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.`
- Use a jet clustering algorithm, e.g. one of the `SlowJet` options described in Appendix B.2, to study the number of jets found in association with the Z^0 above. You can switch off Z^0 decay with `23:mayDecay = no`, and negate its status code by `pythia.event[iZ].statusNeg()`, so that it will not be included in the jet finding. Here `iZ` is the last copy of the Z^0, cf. how the last top copy was found above. Again check the importance of FSR/ISR/MPI.

As a final note, some past summer schools have focused on specific topics for tutorials, and often used preconfigured environments. These tutorials can offer further insights, but

are seldom of the PYTHIA-standalone character that we have aimed for here. If you are interested, some of them are available at
`https://gitlab.com/Pythia8/tutorials`.

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 [11]. An online listing is available from
`http://pdg.lbl.gov/2022/reviews/rpp2022-rev-monte-carlo-numbering.pdf`

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_L^0	113	ρ^0	3112	Σ^-
5	b	15	τ^-	25	H^0	421	D^0	310	K_S^0	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_L^0 and K_S^0 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 (see the online manual for the meaning of each specific code):

code range	explanation
11 – 19	beam particles
21 – 29	particles of the hardest subprocess
31 – 39	particles of subsequent subprocesses in multiparton 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 it in the event listing.

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

A.4 Colour flow information

The colour flow information is based on the Les Houches Accord convention [2]. In it, the number of colours is assumed infinite, so that each new colour line can be assigned a new separate colour. These colours are given consecutive labels: 101, 102, 103, ... A gluon has both a colour and an anticolour label, an (anti)quark only (anti)colour.

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

B Some facilities

The PYTHIA package contains some facilities that are not part of the core generation mission, but are useful for standalone running, notably at summer schools. Here we give some brief info on histograms and jet finding.

B.1 Histograms

For real-life applications you may want to use sophisticated histogramming programs like ROOT, which however take much time to install and learn. Within the time at our disposal, we therefore stick with the very primitive `Hist` class. Here is a simple overview of what is involved.

As a first step you need to declare a histogram, with name, title, number of bins and x range (from, to), like

```
     Hist HpT("Higgs transverse momentum", 100, 0., 200.);
```

Once declared, its contents can be added by repeated calls to fill,

```
     HpT.fill( 22.7, 1.);
```

where the first argument is the x value and the second the weight. Since the weight defaults to 1 the last argument could have been omitted in this case.

```
3.50*10^ 2  9
3.00*10^ 2  X   7
2.50*10^ 2  X  1X
2.00*10^ 2  X6 XX
1.50*10^ 2  XX5XX
1.00*10^ 2  XXXXX
0.50*10^ 2  XXXXX

  Contents
    *10^ 2  31122
    *10^ 1  47208
    *10^ 0  79373

  Low edge  --
    *10^ 1  10001
    *10^ 0  05050
```

Figure 2: Simple output of a histogram.

A set of overloaded operators have been defined, so that histograms can be added, subtracted, divided or multiplied by each other. Then the contents are modified accordingly bin by bin. Thus the relative deviation between two histograms `data` and `theory` can be found as

```
diff = (data - theory) / (data + theory);
```

assuming that `diff`, `data` and `theory` have been booked with the same number of bins and x range.

Also overloaded operations with double real numbers are available. Again these four operations are defined bin by bin, i.e. the corresponding amount is added to, subtracted from, multiplied by or divided by each bin. The double number can come before or after the histograms, with obvious results. Thus the inverse of a histogram result is given by `1./result`. The two kind of operations can be combined, e.g.

```
allpT = ZpT + 2.  * WpT;
```

B.1.1 Line-printer output

A histogram can be printed by making use of the overloaded `<<` operator, e.g.

```
cout << ZpT;
```

The printout format is inspired by the old HBOOK one. To understand how to read it, consider the simplified example in Fig. 2. The key feature is that the `Contents` and `Low edge` have to be read vertically. For instance, the first bin has the contents $3 * 10^2 + 4 * 10^1 + 7 * 10^0 = 347$. Correspondingly, the other bins have contents 179, 123, 207 and 283. The first bin stretches from $-(1 * 10^1 + 0 * 10^0) = -10$ to the beginning of the second bin, at $-(0 * 10^1 + 5 * 10^0) = -5$.

The visual representation above the contents give a simple impression of the shape. An `X` means that the contents are filled up to this level, a digit in the topmost row the fraction to which the last level is filled. So the `9` of the first column indicates this bin is filled 9/10 of the way from $3.00 * 10^2 = 300$ to $3.50 * 10^2 = 350$, i.e. somewhere close to 345, or more precisely in the range 342.5 to 347.5.

The printout also provides some other information, such as the number of entries, i.e. how many times the histogram has been filled, the total weight inside the histogram, the total weight in underflow and overflow, and the mean value and root-mean-square width (disregarding underflow and overflow). The mean and width assumes that all the contents is in the middle of the respective bin. This is especially relevant when you plot a integer quantity, such as a multiplicity. Then it makes sense to book with limits that are half-integers, e.g.

```
Hist multMPI( "number of multiparton interactions", 20, -0.5, 19.5);
```

so that the bins are centred at 0, 1, 2, ..., respectively. This also avoids ambiguities which bin gets to be filled if entries are exactly at the border between two bins. Also note that the `fill( xValue)` method automatically performs a cast to double precision where necessary, i.e. `xValue` can be an integer.

B.1.2 Graphics output

If you have Python 3 with Matplotlib installed (see beginning of Appendix D) then an interface is prepared to produce better-looking plots from the histograms above. Necessary instructions, that are used behind-the-scene to produce relevant Python code, should be put near the end of a run, after all histograms have been properly normalized.

In a first step you must decide on the name of the Python program, e.g.:

```
HistPlot hpl( "myplotNN");
```

where file ending .py is added automatically.

For each new frame you give its file name, title, and x and y axis labels:

```
hpl.frame("myfigNNa", "Transverse momentum spectra",
  "$p_{\\perp}$ (GeV)", "$\\sigma$ (mb)");
```

This will (later) create a file `myfigNNa.pdf`. If you keep the file name empty after the first frame, the new frame will be appended to the current file. Note the possibility to use LaTex symbols, but with backslash doubled up to give the correct escape sequence.

For each frame you can add several curves, by histogram name, plotting style and colour, and text label

```
hpl.add( ZpT, "-,blue", "Z");
hpl.add( WpT, "--,red", "W");
hpl.plot();
```

where the final command finishes the frame. Each `add` command writes out its respective histogram contents in a `*.dat` file, to be used subsequently.

Once the Pythia run is finished, you have to run Python to generate the plots

```
python3 myplotNN.py
```

whereafter you can open `myfigNNa.pdf` (and other frames) in a PDF viewer.

B.2 Jet finding

The `SlowJet` class offer jet finding by the $k_{\perp}$, Cambridge/Aachen and anti-$k_{\perp}$ algorithms. By default it is now a front end to the FJcore subset, extracted from the FastJet package [12] and distributed as part of the PYTHIA package, and is therefore no longer slow. It is good enough for basic jet studies, but does not allow for jet pruning or other more sophisticated applications. (An interface to the full FastJet package is available for such uses.)

You set up `SlowJet` initially with

```
SlowJet slowJet( pow, radius, pTjetMin, etaMax);
```

where `pow = -1` for anti-$k_{\perp}$ (recommended), `pow = 0` for Cambridge/Aachen, `pow = 1` for $k_{\perp}$, while `radius` is the R parameter, `pTjetMin` the minimum $p_{\perp}$ of jets, and `etaMax` the maximum pseudorapidity of the detector coverage.

Inside the event loop, you can analyze an event by a call

```
slowJet.analyze( pythia.event );
```

The jets found can be listed by `slowJet.list();`, but this is only feasible for a few events. Instead you can use the following methods, among others:

`slowJet.sizeJet()` gives the number of jets found,
`slowJet.pT(i)` gives the $p_{\perp}$ for the `i`'th jet, and
`slowJet.y(i)` gives the rapidity for the `i`'th jet.

The jets are ordered in falling $p_{\perp}$.

C Interface to HepMC

The standard HEPMC event-record format is frequently used in the MCnet school training sessions, notably since it is required for comparisons with experimental data analyses implemented in the Rivet package. Then a ready-made installation is used. However, for the ambitious, here is sketched how to set up the PYTHIA interface, assuming you already have installed HEPMC. A similar procedure is required for interfacing to other external libraries, so the points below may be of more general usefulness.

To begin with, you need to go back to the installation procedure of section 2 and insert/redo some steps.

1. Move back to the main `pythia83xx` directory (`cd ..` if you are in `examples`).
2. Do a `make distclean` to remove the old configuration and the compiled library.
3. Configure the program:
   ```
   ./configure --with-hepmc3=path
   ```
   where the directory-tree `path` would depend on your local installation. If the library is in a standard location you can omit the `=path` part.
4. Use `make` as before, to recompile with the new configure information available in `examples/Makefile.inc`, and move back to the `examples` subdirectory.
5. You can now also use the `main131.cc` and `main132.cc` examples to produce HEPMC event files. The latter may be most useful; it presents a slight gener-

alisation of the command-line-driven main program you constructed in Section 5. After you have built the executable you can run it with

```
./main132.exe infile hepmcfile > main132.log
```

where `infile` is an input “card” file (like `mymain01.cmnd` or `main132.cmnd`) and `hepmcfile` is your chosen name for the output file with HepMC events.

Note that the above procedure is based on the assumption that you will be running your main programs from the `examples` subdirectory. For experts there is a `make install` step to install the library and associated components in locations of your choice, and a `bin/pythia8-config` script to help you link to the library from anywhere.

D Preparations before starting the tutorial

Normally, you will run this tutorial on your own (laptop or desktop) computer. It is therefore important to make sure that you will be able to extract, compile, and run the code.

Pythia is not a particularly demanding package by modern standards, but it requires you to have a C++ compiler, able to accept C++11 code. You also need to use the `make` and `tar` command tools, and have access to a text editor, such as Emacs. Below, we give some very basic instructions for standard installations on Linux, macOS, and Windows platforms, respectively.

For a nicer display of histograms, additionally the Python language needs to be installed, including the Matplotlib library. Assuming you have the former, e.g. as `python3`, the latter can be downloaded by

```
python3 -m pip install -U matplotlib
```

In the context of summer schools, students are strongly recommended to make sure that the above-mentioned facilities have been properly installed before traveling to the school, especially if the school is in a location which is likely to offer limited bandwidth.

D.1 Linux (Ubuntu)

The default tutorial instructions are intended for Linux (or other Unix-based) platforms, so this should be the easiest type of system to work with. The presence of the required development tools, including the C++ compiler, should be automatic on most Linux distributions. If this is not the case, you have to download them with the command

```
sudo apt install build-essential
```

in an Ubuntu installation, presumably with similar commands in other distributions. Linux distributions use the GCC compiler suite, so you can type `gcc --version` to confirm that GCC is installed.

D.2 macOS

macOS does not include code development tools by default, but they can relatively easily be obtained by installing Apple's Xcode package, which can be downloaded from the App Store on the Mac; just type "xcode" in the App Store search field to find it. Note that downloading and installing Xcode can take quite some time, and if you don't already have an Apple ID it will take even longer, so this should be done well before starting the tutorial. After downloading Xcode you should open it once, since this is required to trigger the full installation procedure.

An alternative is to type

```
xcode-select --install
```

in a terminal window (preceded by `sudo` if you are not the superuser). This will trigger the installation of the command line tools part of the Xcode package, but not the full development environment, and so is a faster procedure.

The Mac uses the CLANG compiler suite rather than the GCC one, but mimics the GCC front end. The remaining minor structural differences are already taken into account in the `Makefile`s, hidden from the normal user. Warnings issued during compilation may differ, and results may not be identical between CLANG and GCC, but should agree within statistical errors.

There are also (at least) two external package management system for Macs: MacPorts (`www.macports.org`) and Homebrew (`https://brew.sh`). You may want to explore these to install other software, such as ROOT and various EMACS variants.

D.3 Windows

Historically it is a problem to run PYTHIA under Microsoft Windows, and it is not supported. That is, even if you buy and install a commercial C++ compiler, you would still have to write replacements for the `Makefile`s yourself. Typically a Windows/Linux dual boot setup has therefore been the recommended solution, with various virtualization software as an option.

This has changed with the introduction of the Windows Subsystem for Linux (WSL) a few years ago. It now allows the installation of a complete Linux distribution, by default Ubuntu, without the overhead of dual boot or virtualization. There is no experience with this solution within the PYTHIA team, but on paper it seems simple. Basically you have to open a powershell terminal in administrator mode, type

```
wsl --install
```

and afterwards restart your machine. From there on you can open a Linux terminal window and operate as for any native Linux installation.

D.4 Python, Docker, and Jupyter notebooks

A Python interface to PYTHIA is available and can be enabled by configuring PYTHIA as follows:

```
./configure --with-python
```

The interface is compatible with both Python 2 and 3. To configure with a specific Python version, the relevant `python-config` script can be passed. For example, the following configures the bindings with Python 3:

```
./configure --with-python-config=python3-config
```

The Python interface can also be installed through the ANACONDA distribution system:

```
conda install -c conda-forge pythia8
```

Note that typically the most recent PYTHIA release should be available via ANACONDA but it sometimes lags behind the release cycle. The default Python interface is a simplified version of the full C++ interface. It is possible to generate the full interface, however:

```
cd plugins/python && ./generate --full && cd -
```

where the `generate` script requires DOCKER.

In general, DOCKER images containing PYTHIA installations are available at

`https://hub.docker.com/u/pythia8`

but these are mainly used for testing and development.

There is a tutorial available that provides the PYTHIA Python interface through a JUPYTER notebook. This tutorial requires only DOCKER; both the JUPYTER instance and Python interface are provided through the DOCKER container. The DOCKER container for this tutorial can be run with the following command:

```
docker run --rm -u ‘id -u $USER‘ -v $PWD:$PWD -w $PWD -p 8888:8888 \\
   -it pythia8/tutorials:cteq22 $@
```

You will then receive instructions on how to open the JUPYTER notebook. For example,

```
[YYY] Serving notebooks from local directory: XXXXXX
[YYY] The Jupyter Notebook is running at:
[YYY] http://d9c979095ffc:8888/?token=a1024fc584f44...
[YYY]  or http://127.0.0.1:8888/?token=a1024fc584f44...
```

where the `http://127.0.0.1:8888` link can be copied into a browser to open the JUPYTER notebook.

Once connected to the JUPYTER service, choose `worksheet8307.ipynb`. Each cell of the notebook can be run to follow the tutorial in the main body of (the previous version of) this document, but using the Python interface to PYTHIA.

References

[1] C. Bierlich, S. Chakraborty, N. Desai, L. Gellersen, I. Helenius, P. Ilten, L. Lönnblad, S. Mrenna, S. Prestel and C. T. Preuss, T. Sjöstrand, P. Z. Skands, M. Utheim, and R. Verheyen, SciPost Phys. Codebases 8-r8.3 (2022), [arXiv:2203.11601 [hep-ph]]

[2] 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]

[3] J. Alwall et al., Comput. Phys. Commun. **176** (2007) 300 [hep-ph/0609017]

[4] J. Butterworth et al., arXiv:1405.1067 [hep-ph]

[5] M.L. Mangano, M. Moretti, F. Piccinini and M. Treccani, JHEP **01** (2007) 013

[6] L. Lönnblad and S. Prestel, JHEP **03** (2012) 019 [arxiv:1109.4829 [hep-ph]]

[7] L. Lönnblad and S. Prestel, JHEP **02** (2013) 094 [arxiv:1211.4827 [hep-ph]]

[8] L. Lönnblad and S. Prestel, JHEP **03** (2013) 166 [arxiv:1211.7278 [hep-ph]]

[9] J. Alwall et al., JHEP **06** (2011) 128, [arXiv:1106.0522 [hep-ph]]

[10] G. Salam, Eur. Phys. J. C67 (2010) 637 [arxiv:0906.1833 [hep-ph]]

[11] Particle Data Group, R.L. Workman et al., PTEP 2022 (2022) 083C01

[12] M. Cacciari, G.P. Salam and G. Soyez, Eur. Phys. J. C72 (2012) 1896 [arXiv:1111.6097 [hep-ph]]