


For the tutorials
at TRISEP 2013
Vancouver, Canada
4 & 5 July 2013

PYTHIA 8 Worksheet

Torbjörn Sjöstrand
Department of Astronomy and Theoretical Physics, Lund University

1 Introduction

The objective of this exercise is to teach you the basics of how to use event generators, in this case exemplified by Pythia 8.1. Various applications to Higgs physics will be used as illustrations. As you become more familiar you will better understand the tools at your disposal, and can develop your own style to use them. Within this first exercise it is not possible to describe the physics models used in the program; for this we refer to the Pythia 8.1 brief introduction [1], to the full Pythia 6.4 physics description [2], and to all the further references found in them.

Pythia 8 is, by today's standards, a small package. It is completely self-contained, and is therefore easy to install for standalone usage, e.g. if you want to have it on your own laptop, or if you want to explore physics or debug code without any danger of destructive interference between different libraries. Section 2 describes the basic installation procedure, what we will need for this introductory session. The possibility to link to other programs is important for use inside experimental collaborations, in particular, but will not be covered here.

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, Sections 4 – 6. In subsequent sections you then use this platform to study a number of aspects of Higgs physics Pythia.

Finally, Appendix A contains a brief summary of the event-record structure, and Appendix B of the built-in simple histogramming class.

2 Installation

Those of you who already have Pythia 8 installed on your laptop can skip this section. Else, if you have a Linux, Mac OS X or other Unix-based installation, proceed as follows.

1. In a browser, go to
   `http://home.thep.lu.se/~torbjorn/Pythia.html`
2. Download the (current) program package
   `pythia8176.tgz`
   to a directory of your choice.
3. In a terminal window, `cd` to where `pythia8176.tgz` was downloaded, and type
   `tar xvfz pythia8176.tgz`
   This will create a new (sub)directory `pythia8176` where all the PYTHIA files are now unpacked and ready.
4. Move to this directory and compile
   ```
   cd pythia8176
   make
   ```
   The compilation will take a few minutes (computer-dependent; if you have two or more kernels `make -j2` will speed it up). The PYTHIA 8 libraries are now compiled and ready for physics.
5. For test runs,
   `cd examples`
   will bring you to a subdirectory with several `mainNN` programs, with `NN` from `01` through `28` (and beyond). These example programs illustrate different aspects of PYTHIA 8 usage. For now you only need to use one or two of them to check that the installation works, say `NN = 01`. To execute one of them, do
   ```
   make mainNN
   ./mainNN.exe > outNN
   ```
   The output is written to the `outNN` file. Open it to check that you obtain sensible-looking output. The details are not important for now, only that the installation works. (The `outref/outNN` files contain examples of successful runs, if you feel uncertain.)

If you want to work under Windows, you are pretty much on your own. For step 4 a `Makefile.msc` exists, so if you have `Win32/nmake` installed

```
nmake -f Makefile.msc CFG="Win32 Release"
```

hopefully should be enough to compile the library (but no guarantees). Thereafter you have to figure out how to do the test runs.

3 A "Hello World" program

We will now generate a single gg → H event at the LHC.

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. Incoming pp beams is default.
  Pythia pythia;         // Declare Pythia object
  pythia.readString("Beams:eCM = 8000."); // 8 TeV CM energy.
  pythia.readString("HiggsSM:gg2H = on"); // Switch on g g -> H.
  pythia.init();         // Initialize.

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

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

Next you need to edit the `examples/Makefile` so it knows what to do with `mymain.cc`. The lines

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

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

```
 main31 ...  main40 mymain: \
```

Now it should work as before with the other examples:

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

whereafter you can study `mymain.out`, especially the example of 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 $g_1g_2 \to H^0$ hard interaction, with the indices used to distinguish the gluons. Initially the Higgs has a positive status code, while the gluons have a negative one. When an initial-state $g_3 \to g_1g_4$ shower branching is added, the new net process $g_3g_2 \to H^0g_4$ is appended to the end of the event record, whereof the new H^0 and g_4 with positive status. At the same time the status code of the original H^0 is negated. At any stage of the shower there is thus only one "current" copy of the Higgs. After the shower, when the final Higgs decays, e.g. $H^0 \to b\bar{b}$, also that copy receives a negative status code.

When you understand the basic principles, see if you can find several copies of the Higgs, 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.

- We almost always wish to generate more than one event. To do this, introduce a loop around `pythia.next()`, so the code now reads

```
for (int iEvent = 0; iEvent < 10; ++iEvent) {
  pythia.next();
}
```

Hereafter, we will call this the *event loop*. The program will now generate 10 events; each call to `pythia.next()` resets the event record and fills it with a new event. By default only the first event is listed, but you could change this e.g. to 3 by a line

```
pythia.readString("Next:numberShowEvent = 3");
```

along with the other `pythia.readString` commands.

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

  ```
  pythia.stat();
  ```

  just before the end of the program. Do runs with 10, 100 and 1000 events and study how the cross section value and error estimate changes. Begin the `./mymain.exe` command by `time` to obtain timing information.

  Note: the error quoted is only the statistical one. As mentioned in the lectures, $\sigma_{\mathrm{gg}\to\mathrm{H}}$ has unexpectedly large NLO and NNLO corrections, not included in the PYTHIA code. To a good first approximation this can be viewed just as an overall rescaling of the amount of Higgs production, leaving the Higgs event properties unaffected, so it is not too bad.

- 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:

    ```
    if (!pythia.next()) continue;
    ```

  During event generation, a problem message is printed only the first time that particular problem occurs. The above-mentioned `pythia.stat()` will tell you how many times each problem was encountered over the entire run. In particular, repeated aborts here indicate some significant problem.

- Looking at 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). The `cout` statement, therefore, will give a list of the PDG code of every particle in the event record. Do this for one event and compare with the corresponding event-listing information.

- As mentioned above, the event listing contains all partons and particles, traced through a number of intermediate steps, notably by parton-shower emissions. Eventually, the Higgs will decay, and by implication it is the last copy of this Higgs in the event record that gives the "final" answer. You can obtain the location of this final H^0 e.g. by a line just before the *particle loop*
  ```
  int iH = 0;
  ```
  and a line inside the *particle loop*
  ```
  if (pythia.event[i].id() == 25) iH = i;
  ```
  The value of `iH` will be set every time a Higgs is found in the event record. When the *particle loop* is complete, `iH` will now point to the final Higgs in the event record (which can be accessed as `pythia.event[iH]`). Check that your code works for one or a few events, by comparing event listings with the `iH` value.
- In addition to the particle properties in the event listing, there are also methods that return many derived quantities for any particle `i`, such as transverse momentum, `pythia.event[i].pT()`, and rapidity, `pythia.event[i].y()`. Use these methods to print out the values for the final Higgs 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, that can be used for rapid check/debug purposes, see Appendix B. To book the histograms, insert before the *event loop*
  ```
  Hist pTH("Higgs transverse momentum", 100, 0., 200.);
  Hist yH("Higgs pseudorapidity", 100, -5., 5.);
  ```
  Next we want to fill the histograms in each event, so before the end of the *event loop* insert
  ```
  pTH.fill( pythia.event[iH].pT() );
  yH.fill( pythia.event[iH].y() );
  ```
  Finally, to write out the histograms, after the *event loop* insert a line like
  ```
  cout << pTH << yH;
  ```

5 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. This also saves the effort of repeatedly typing (or copying) `pythia.readString("");`, but otherwise the two approaches are equivalent.

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

```
! Higgs production at the LHC
Beams:eCM = 8000.    ! the cm energy of collisions.
HiggsSM:gg2H = on    ! the g g -> H0 SM process
```

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*, along with much other useful information. Open the file

```
pythia8176/htmldoc/Welcome.html
```

to gain access to it. Use the left-column index to navigate among the topics, which are then displayed in the larger right-hand field. Cut-and-paste of variable names from the manual can be used to avoid spelling mistakes.

The final step is to modify our program to use this input file. The file name can be hardcoded in the main program but, for more flexibility, it can also be provided as a command-line argument. To illustrate the latter, replace the `int main() {` line by

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

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

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

The executable `mymain.exe` is then run with a command line like

```
time ./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 in the `Pythia` code itself. These are intended for the main program, and thus begin with `Main:`. The most basic of those is `Main:numberOfEvents`, which you can set in the `cmnd` file 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) {
```

6 Code consolidation

At this point we have a main program that (apart from the comments) should look something like

```
// Headers and Namespaces.
#include "Pythia.h"
using namespace Pythia8;

// Beginning of main program.
int main(int , char* argv[]) {

  // Set up generation and initialize.
  Pythia pythia;
  pythia.readFile(argv[1]);
  pythia.init();

  // Number of events. Book histograms.
  int nEvent = pythia.mode("Main:numberOfEvents");
```

```
  Hist pTH("Higgs transverse momentum", 100, 0., 500.);
  Hist yH("Higgs pseudorapidity", 100, -5., 5.);

  // Begin the event loop.
  for (int iEvent = 0; iEvent < nEvent; ++iEvent) {
    if (!pythia.next()) continue;

    // Locate the final copy of the Higgs.
    int iH = 0;
    for (int i = 0; i < pythia.event.size(); ++i) {
      if (pythia.event[i].id() == 25) iH = i;
    }

    // Histogram some Higgs properties.
    pTH.fill( pythia.event[iH].pT() );
    yH.fill( pythia.event[iH].y() );

  // End the event loop.
  }

  // Statistics and histograms.
  pythia.stat();
  cout << pTH << yH;

  // End of main program.
  return 0;
}
```

and a data file like

```
! Number of events to generate and to list.
Main:numberOfEvents = 1000
Next:numberShowEvent = 0

! Higgs production at the LHC
Beams:eCM = 8000.    ! the cm energy of collisions.
HiggsSM:gg2H = on    ! the g g -> H0 SM process
```

where it has been assumed that you now are tired of getting lengthy event listings. You will still get a summary of the main properties of the first hard process, but that could be switched off as well.

A small extra comment: you may have noticed that all runs by default use the same random-number sequence, which is convenient for reproducibility. However, with `Random:setSeed = on` and `Random:seed = 123456789` you could pick another sequence. Any number between 1 and 900,000,000 gives a unique sequence.

7 Some studies of Higgs production

You can now expand the program in different directions. Here we list a few examples. Go through as many as time allows. You don't have to do them in any particular order, although some of the hints in the earlier studies can be helpful also in the later ones.

7.1 Production processes

While gg $\rightarrow$ 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 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.

7.2 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.

7.3 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?

7.4 Associated jets

Let's return to the different production channels, which give different event characteristics. Well-known is that the qq $\to$ 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 Z^0Z^0 and W^+W^- fusion (f here denotes a fermion; the same processes also exist e.g. at e^+e^- colliders), that are to be compared with the standard gg $\to$ H^0 one.

The `SlowJet` class offer jet finding by the $k_\perp$, Cambridge/Aachen and anti-$k_\perp$ algorithms, equivalent to what can be obtained with the FastJet package [5], less streamlined for speed but better integrated into Pythia. You set it up initially with

```
SlowJet slowJet( -1, radius, pTjetMin, etaMax);
```

where `-1` denotes the use of anti-$k_\perp$, `radius` the R parameter, say 0.7, `pTjetMin` the minimum $p_\perp$ of jets, say 30 (GeV), and `etaMax` the maximum pseudorapidity of the detector coverage, say 5.

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:

`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 $p_\perp$.

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.

7.5 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, see section 7.2 for switch names. Also study the contribution of the Higgs decay itself to the multiplicity, e.g. by setting the Higgs stable, see section 7.4 for a recipe. 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.

8 Further studies for credit

If you expect to receive university credit for this summer school, you should also do the following two exercises on your own, after the tutorials, and hand in the solutions.

8.1 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 $\mathrm{Z}^0\mathrm{Z}^0$ (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?

8.2 Comparison with Z^0 production

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

Compare the two processes $\mathrm{gg} \to \mathrm{H}^0$ and $\mathrm{q\overline{q}} \to \gamma^*/\mathrm{Z}^0$, 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 $\pm$ 1 GeV.

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

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 5), 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/2012/reviews/rpp2012-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^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 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 in the event listing.

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

A.4 Colour flow information

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

While colours are traced consistently through hard processes and parton showers, the subsequent beam-remnant-handling step often involves a drastic change of colour labels. Firstly, previously unrelated colours and anticolours taken from the beams may at this stage be associated with each other, and be relabelled accordingly. Secondly, it appears

that the close space–time overlap of many colour fields leads to reconnections, i.e. a swapping of colour labels, that tends to reduce the total length of field lines.

B 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 that comes with PYTHIA. We here provide 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 pTH("Higgs transverse momentum", 100, 0., 200.);
```

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

```
pTH.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.

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
```

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

```
 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
```

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 centered 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.

Histogram values can also be output to a file

```
pTH.table("filename");
```

which produces a two-column table, where the first column gives the center of each bin and the second one the corresponding bin content. This may be used for plotting e.g. with Gnuplot.

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] M. Cacciari and G.P. Salam, Phys. Lett. B641 (2006) 57 [hep-ph/0512210]