PYTHIA 8 Worksheet

Torbjörn Sjöstrand, Richard Corke
Department of Theoretical Physics, Lund University

Peter Skands
Theoretical Physics, CERN

Introduction

The PYTHIA program is a standard tool for the generation of high-energy collisions (specifically, it focuses on centre-of-mass energies greater than about 10 GeV), 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 full (such as GEANT or FLUKA) or fast (such as PGS or DELPHES) detector-simulation codes.

The PYTHIA 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.

The objective of this exercise is to teach you the basics of how to use the PYTHIA 8.1 event generator to study various physics aspects. As you become more familiar you will better understand the tools at your disposal, and can develop your own style to use them. Within this first exercise it is not possible to describe the physics models used in the program; for this we refer to the PYTHIA 8.1 brief introduction [1], to the full PYTHIA 6.4 physics description [2], and to all further references found in them. More general introductions to the physics of event generators can be found in the recent *MCnet* review on general-

purpose event generators for LHC physics [3], and in summer-school lecture notes such as [4, 5].

PYTHIA 8 is written in C++. If you are not familiar with C or C++, an additional programming reference in addition to these notes will therefore be useful. Nonetheless, most of the examples we will cover should be fairly straightforward to follow given just some basic knowledge of computer programming. By today's standards, PYTHIA 8 is 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 1 describes the installation procedure, which is what we will need for this introductory session. In section 1, we will also use a browser to open the PYTHIA HTML documentation pages, which are included together with the code, and which contain a complete list of all parameters and switches, descriptions of the available methods, example programs, and interfaces, and explanations of how to use PYTHIA's output and physics analysis tools.

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 2 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 3.

In Section 4 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.

While PYTHIA can be run standalone, it can also be interfaced with a set of other libraries, such as FASTJET (for jet clustering), HEPMC (the standard format used by experimentalists to store generated events), LHAPDF (for additional parton distribution functions), MADGRAPH (for additional hard processes, in particular for scenarios beyond the Standard Model), ROOT (the standard analysis and plotting tool used by experimentalists), VINCIA (for an alternative parton-shower model and a ROOT-based runtime display), and several others. We refer to the example programs included both with PYTHIA and with these other packages for how to extend your work with PYTHIA in these directions. The PYTHIA HTML manual also contains a section labeled "Link to Other Programs" that you can consult.

Finally, Section 5 gives some suggestions for the study of other possible physics topics and non-standard options.

1   Installation

Denoting a generic PYTHIA 8 version `pythia81xx` (at the time of writing `xx` = 65), here is how to install PYTHIA 8 on a Linux/Unix/MacOSX system as a standalone package.

1. In a browser, go to

2. Download the (current) program package
   pythia81xx.tgz
   to a directory of your choice (e.g. by right-clicking on the link).

3. In a terminal window, `cd` to where `pythia81xx.tgz` was downloaded, and type
   `tar xvfz pythia81xx.tgz`
   This will create a new (sub)directory `pythia81xx` where all the PYTHIA source files are now ready and unpacked.

4. Move to this directory (`cd pythia81xx`) and do a `make`. This will take ∼3 minutes (computer-dependent). The PYTHIA 8 libraries are now compiled and ready for physics.

5. For test runs, `cd` to the `examples/` subdirectory. An `ls` reveals a list of programs, `mainNN`, with `NN` from `01` through `28` (and beyond). These example programs each illustrate an aspect of PYTHIA 8. For a list of what they do, see the "Sample Main Programs" page in the online manual (point 6 below).

   Initially only use one or two of them to check that the installation works. Once you have worked your way though the introductory exercises in the next sections you can return and study the programs and their output in more detail.

   To execute one of the test programs, do
   ```
   make mainNN
   ./mainNN.exe
   ```
   The output is now just written to the terminal, `stdout`. To save the output to a file instead, do `./mainNN.exe > mainNN.out`, after which you can study the test output at leisure by opening `mainNN.out`. See Section 2 for an explanation of the event record that is listed in several of the runs.

6. If you use a web browser to open the file
   `pythia81xx/htmldoc/Welcome.html`
   you will gain access to the online manual, where all available methods and parameters are described. Use the left-column index to navigate among the topics, which are then displayed in the larger right-hand field.

2 A "Hello World" program

We will now generate a single proton-proton collision at the LHC, in which two gluons interact to produce a top quark pair, $gg \to t\bar{t}$.

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

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

int main() {             // Begin main program.
```

```
   // Set up generation.
   Pythia pythia;          // Declare Pythia object
   pythia.readString("Top:gg2ttbar = on"); // Switch on process.
   pythia.readString("Beams:eCM = 7000."); // 7 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.
```

If you are unfamiliar with C++, pay particular attention to getting all quotation marks, semicolons, and opening and closing brackets exactly right, as in the above example.

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

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

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

```
  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).

2.1   The Event Record

The event record is set up to store every step in the evolution from an initial low-multiplicity partonic process (such as $gg \to t\bar{t}$) to a final high-multiplicity hadronic state (such as, to take an arbitrary example, $pp \to n^0 \pi^+ \rho^0 B^- \mu^+ \nu_\mu B_s^{*+} e^- \bar{\nu}_e K^0 K^- \Delta^{++}$, which could be produced via the aforementioned partonic subprocess), in the order that new particles are generated by the program. 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.

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 (explained below) are still present in the final state. To exemplify, consider a top quark produced in the hard interaction, initially with positive status code. When later, a shower branching $t \to tg$ 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 Markov-chain evolution there is thus only one "current" copy of the top. After the shower, when the final top decays, $t \to bW^+$, 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.

The event-record output (produced by the `event.list();` method) from your example program 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()`), explained below;
- `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()`), explained below;
- `mothers` and `daughters`, documentation on the event history (methods `mother1()`, `mother2()`, `daughter1()` and `daughter2()`); explained below;
- `colours`, the colour flow of the process (methods `col()` and `acol()`); explained below;
- `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.

2.1.1 Identity codes

To distinguish between different types of particles (e.g., electrons, muons, up quarks, down quarks, etc), PYTHIA uses a numbering scheme standardized by the Particle Data Group (PDG) (together with a few PYTHIA-specific extensions). A complete specification of the PDG codes is found in the Review of Particle Physics [6]. An online listing is available from

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

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

1	d	11	e^-	21	g	211	π^+	111	π^0	213	ρ^+	2112	n
2	u	12	ν_e	22	γ	311	K^0	221	η	313	K^{*0}	2212	p
3	s	13	μ^-	23	Z^0	321	K^+	331	η'	323	K^{*+}	3122	Λ^0
4	c	14	ν_μ	24	W^+	411	D^+	130	K_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.

2.1.2 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.

Note: while the *identity* codes described in the preceding subsection are standardized by the PDG and the same for all high-energy Monte Carlo event-generator codes, the *status* codes here described are intrinsic to PYTHIA. Other generator codes (such as HERWIG or SHERPA) and/or other event-record standards (such as the HEPMC one) will in general have different status codes, with different meanings from the ones we use.

2.1.3 History information

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

There are also several special cases. One such is when "the same" particle appears as a second copy, e.g. because its momentum has been shifted by it taking a recoil in the dipole picture of parton showers. Then the original has both daughter indices pointing to the same particle, which in its turn has both mother pointers referring back to the

original. Another special case is the description of ISR by backwards evolution, where the mother is constructed at a later stage than the daughter, and therefore appears below in the event listing.

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

2.1.4 Colour flow information

The colour flow information is based on the Les Houches Accord convention [7]. In it, the number of colours is assumed infinite (called the *planar* or *leading-colour* limit), 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.

3   A first realistic analysis

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

- Often, we wish to mix several processes together. To add the process $q\overline{q} \to t\overline{t}$ 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 now reads
      ```
      for (int iEvent = 0; iEvent < 5; ++iEvent) {
        pythia.next();
      }
      ```
  Hereafter, we will call this the **event loop**. The program will now 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. The above-mentioned `pythia.stat()` will then tell you how many times each problem was encountered over the entire run.

- 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 Section 2). 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 copy of this top in the event record that gives the "final" answer. You can obtain the location of this final top e.g. by 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, 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;
```
This will append to the output some simple ASCII histograms. If you look in the online HTML manual under "Study Output", the "Histograms" page can tell you about further methods to output the histograms as tables, for later plotting with programs like GNUPLOT or ROOT. If you have ROOT installed, there is even an option for opening runtime displays for realtime histogram plotting, by using the VINCIA plugin, see section 5.

- 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.

4   Input files

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

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

```
    ! 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 = 7000.     ! 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 `mymain.cmnd` file can contain one command per line, of the type
```
        variable = value
```
All variable names are case-insensitive (the mixing of cases has been chosen purely to improve readability) and non-alphanumeric characters (such as !, # or $) will be interpreted as the start of a comment. All valid variables are listed in the online manual (see

Section 2, point 6, above). Cut-and-paste of variable names can be used to avoid spelling mistakes.

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

```
int main(int argc, char* argv[]) {
```
and replace all `pythia.readString(...)` commands with the single command
```
pythia.readFile(argv[1]);
```

The executable `mymain.exe` is then run with a command line like
```
./mymain.exe mymain.cmnd > mymain.out
```
and should give the same output as before.

In addition to all the internal `Pythia` variables there exist a few defined in the database but not actually used. These are intended to be useful in the main program, and thus begin with `Main:`. The most basic of those is `Main:numberOfEvents`, which you can use to specify how many events you want to generate. To make this have any effect, you need to read it in the main program, after the `pythia.readFile(...)` command, by a line like
```
int nEvent = pythia.mode("Main:numberOfEvents");
```
and set up the **event loop** like
```
for (int iEvent = 0; iEvent < nEvent; ++iEvent) {
```

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 171 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 = 3` (or other values between 1 and 7)
  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. In addition, detailed tuning of `Pythia` 8 is still in its infancy.
- `Random:setSeed = on`
  `Random:seed = 123456789`
  all runs by default use the same random-number sequence, for reproducibility, 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 usage of further `Main:` variables is illustrated e.g. in `main03.cc`, and the possibility to use command-line input files in `main16.cc` and `main42.cc`.

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

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

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

5 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 $\bar{\mathrm{p}}$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`).

- Higgs production can proceed through several different production processes. For the Standard Model Higgs some process switches are:

  `HiggsSM:ffbar2H` for $f\bar{f} \to H^0$ (f generic fermion, here mainly $b\bar{b} \to H^0$);

  `HiggsSM:gg2H` for $gg \to H^0$;

  `HiggsSM:ffbar2HZ` for $f\bar{f} \to H^0 Z^0$;

  `HiggsSM:ffbar2HW` for $f\bar{f} \to H^0 W^{\pm}$;

  `HiggsSM:ff2Hff(t:ZZ)` for $f\bar{f} \to H^0 f\bar{f}$ via $Z^0 Z^0$ fusion;

  `HiggsSM:ff2Hff(t:WW)` for $f\bar{f} \to H^0 f\bar{f}$ via $W^+ W^-$ fusion;

  `HiggsSM:all` for all of the above (and some more).

  Study the $p_\perp$ and η spectrum of the Higgs in these processes, and compare.

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

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

- Using your favourite jet cluster algorithm, study the number of jets found in association with the Z^0 above. You can switch off Z0 decay with `23:mayDecay = no`. If you do not have a jet finder around, to begin with you can use the simple `SlowJet` one that comes with PYTHIA, see the "Event Analysis" page in the online manual, which offers a choice of the $k_\perp$, Cambridge/Aachen and anti-$k_\perp$ algorithms. Again check the importance of FSR/ISR/MPI.

- In Section 2, we saw how to book and output simple ASCII histograms, and you may also have experimented with writing them to files for later plotting. If you have ROOT installed, there is even an option for opening runtime displays for real-time histogram plotting, via the VINCIA plugin. (Note for ASP 2012: the ROOT installation on the ASP 2012 desktops does not appear to be compatible with this functionality, but you may want to try it out on your laptops.) The VINCIA plug-in also contains an alternative parton-shower formalism (which can be disabled, if you just wish to use its runtime display facilities for vanilla PYTHIA). It can be obtained from the web page:

      http://projects.hepforge.org/vincia

  Go to the web page, follow the instructions there to download and install the plugin, and try out the `vincia03-root` example program included with that code. If everything is set up correctly, including ROOT (you may have to edit the VINCIA Makefile, under "Linking to ROOT", in particular make sure the ROOTSYS, ROOTINCDIR, and ROOTLIBDIR environment variables are set appropriately for your ROOT installation), then you should be seeing runtime displays appearing on your screen, which are updated in realtime as the program runs. The example program and the (separate) VINCIA online HTML documentation can point you to how to adapt this functionality for your own purposes.

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] *MCnet Collaboration*, A. Buckley *et al.*, Phys. Rept. **504** (2011) 145

[4] T. Sjöstrand, Lectures given at SUSSP65, arXiv:0911.5286 [hep-ph].

[5] P. Skands, Lectures given at TASI 2012, arXiv:1207.2389 [hep-ph].

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

[7] 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]