CERN-LCGAPP-2005-05

LU TP 05–43

July 2005
Updated edition, April 2006

A brief introduction to PYTHIA 8.048

T. Sjöstrand

CERN/PH, CH–1211 Geneva 23, Switzerland

and

Department of Theoretical Physics, Lund University, Sölvegatan 14A, SE–223 62 Lund, Sweden

Abstract

The PYTHIA program is a standard tool for the generation of high-energy collisions. While previous versions were written in Fortran, PYTHIA 8 represents a complete rewrite in C++. This is a project still very much under development, and will not offer a production-run-quality replacement for some time to come. Nevertheless, enough of the basic structure is in place that the program can be tried out. Early feedback, e.g. on interoperability with LHC software, could avoid problems later on. The current introduction should provide enough details to get going with such an exploratory phase.

1 Introduction

The development of Jetset [1], containing several of the components that later were merged with Pythia [2], was begun in 1978. Thus the current Pythia 6 generator [3] is the product of more than a quarter of a century of development, and some of the code has not been touched in a very long time. New options have been added, but old ones seldom removed. The basic structure has been expanded in different directions, well beyond what it was once intended for, making it rather cumbersome by now.

From the onset, all code has been written in Fortran 77. For the LHC era, the experimental community has made the decision to move heavy computing completely to C++. Fortran support may be poor to non-existing, and young experimenters will not be conversant in Fortran any longer. Therefore it is logical to move Pythia to C++, in the process cleaning up and modernizing various aspects.

A first attempt in this direction was the Pythia 7 project [4]. However, after the Herwig++ [5] group decided to join in the development of a common administrative structure, ThePEG [6], work on the physics aspects stalled.

Pythia 8 is a clean new start, to provide a successor to Pythia 6. In a return to the traditional Pythia spirit, it is a completely standalone generator, thus not relying on ThePEG or any other external library. Some hooks for links to other programs are already provided, however, and others may be added. Work on Pythia 8 was begun from scratch in September 2004, so far as a one-person effort, with a three-year "road map".

The second public version, presented here, clearly is not yet developed enough to provide a realistic alternative to Pythia 6. Instead it it made public to allow early feedback from the experimental community, especially within the context of LHC software activities. It is intended that all subversions in the 8.0xx series should be viewed as development snapshots, with 8.100, sometime in early 2007, the first one to be taken seriously. Even so, that version will not be a complete replacement in all respects, but strongly focused on LHC applications.

Further, with the rise of automatic matrix-element code generation and phase-space sampling, input of process-level events via the Les Houches Accord (LHA) [7] reduces the need to have extensive process libraries inside Pythia itself. Thus emphasis is on providing a good description of subsequent steps of the story, involving elements such as initial- and final-state parton showers, multiple parton–parton interactions, string fragmentation, and decays. All the latter components now exist as C++ code, even if in a preliminary form, with finer details to be added, and still to be better integrated and tuned. At the current stage, however, there is only the beginning of a Pythia 8 process library; instead an interface is provided to Pythia 6, so that essentially all hard processes available there can be generated and sent on to Pythia 8, transparent to the user. This Fortran link is not intended as a long-term solution, of course.

The current document provides an introduction to Pythia 8 usage. It is not an attempt to compete with the Pythia 6 manual [3] in comprehensiveness, but is rather more lightweight. Much of the physics aspects are unchanged relative to the Pythia 6 manual, and so we refer to it and to other physics articles for that. Instead we here give an overview for potential users who already have some experience with event generators in general, and Pythia 6 in particular, who want to understand what is different about Pythia 8, and how the new program should be used.

2 Program Structure

2.1 Program flow

The physics topics that have to come together in a complete event generator can crudely be subdivided into three stages:

1. The generation of a "process" that decides the nature of the event. Often it would be a "hard process", such as $\mathrm{gg} \to \mathrm{h}^0 \to \mathrm{Z}^0\mathrm{Z}^0 \to \mu^+\mu^-\mathrm{q}\overline{\mathrm{q}}$, that is calculated in perturbation theory, but a priori we impose no requirement that a hard scale must be involved. Only a very small set of partons/particles is defined at this level, so only the main aspects of the event structure are covered.
2. The generation of all subsequent activity on the partonic level, involving initial- and final-state radiation, multiple parton–parton interactions and the structure of beam remnants. Much of the phenomena are under an (approximate) perturbative control, but nonperturbative physics aspects are also important. At the end of this step, a realistic partonic structure has been obtained, e.g. with broadened jets and an underlying-event activity.
3. The hadronization of this parton configuration, by string fragmentation, followed by the decays of unstable particles. This part is almost completely nonperturbative, and so requires extensive modelling and tuning or, especially for decays, parametrizations of existing data. It is only at the end of this step that realistic events are available, as they could be observed by a detector.

This division of tasks is not watertight — parton distributions span and connect the two first steps, to give one example — but it still helps to focus the discussion.

The structure of the Pythia 8 generator, as illustrated in Fig. 1, is based on this subdivision. The main class for all user interaction is called `Pythia`. It calls on the three classes `ProcessLevel`, `PartonLevel` and `HadronLevel`, corresponding to points 1, 2 and 3 above. Each of these, in their turn, call on further classes that perform the separate kinds of physics tasks.

Information is flowing between the different program elements in various ways, the most important being the event record, represented by the `Event` class. Actually, there are two objects of this class, one called `process`, that only covers the few partons of the "hard" process, and another called `event`, that covers the full story from the incoming beams to the final hadrons.

There are also two incoming `BeamParticle`s, that keep track of the partonic content left in the beams after a number of interactions and initial-state radiations, and rescales parton distributions accordingly.

Finally, a number of utilities can be used just about anywhere, for Lorentz four-vectors, random numbers and simple histograms, for setting and getting various parameters and particle data, and for a number of other "minor" tasks.

Orthogonally to the subdivision above, there is another, more technical classification, whereby the user interaction with the generator occurs in three phases:

- Initialization, where the tasks to be performed are specified.
- Generation of individual events (the "event loop").
- Finishing, where final statistics is made available.

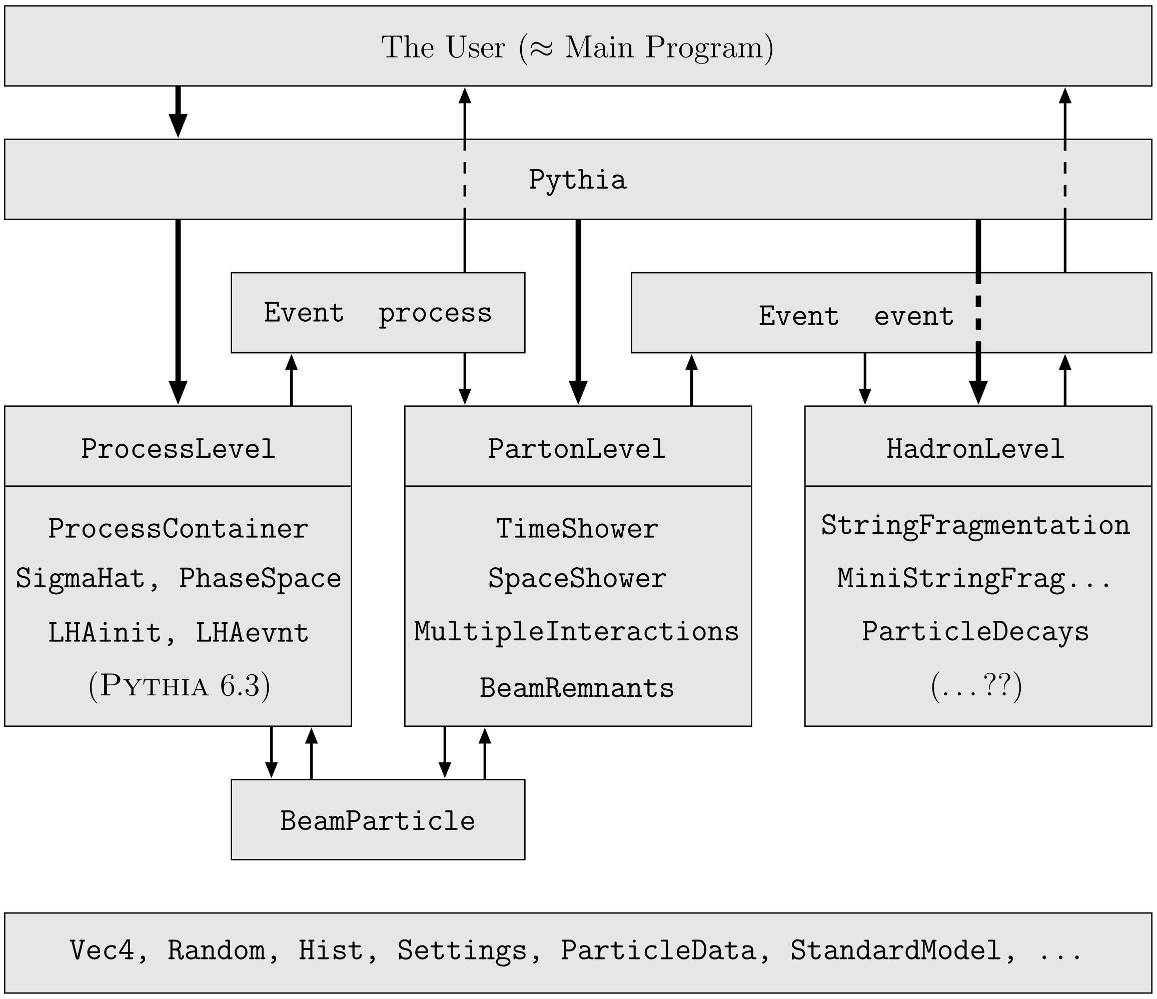

Figure 1: The relationship between the main classes in Pythia 8. The thick arrows show the flow of commands to carry out different physics tasks, whereas the thinner show the flow of information between the tasks. The bottom box contains common utilities that may be used anywhere. Obviously the picture is strongly simplified.

Again the subdivision (and orthogonality) is not strict, with many utilities and tasks stretching across the borders, and with no finishing step required for many aspects. Nevertheless, as a rule, these three phases are represented by different methods inside the class of a specific physics task.

2.2 Program files

The code is subdivided into a set of files, mainly by physics task. Each file typically contains one main class, but often with a few related helper classes that are not used elsewhere in the program. Normally the files come in pairs.

1. A header file, `.h` in the `include` subdirectory, where the public interface of the class is declared, and inline methods are defined.
2. A source code file, `.cc` in the `src` subdirectory, where most of the methods are implemented.

During compilation, related dependency files, `.d`, and compiled code, `.o` are created in the `tmp` subdirectory.

In part the `.xml` documentation files in the `doc` subdirectory have matching names, but the match is broken by the desire to group topics more by user interaction than internal operation. The introduction you read now only represent an extract of the full information available in the `.xml` files, hopefully eventually to evolve into a nice GUI.

Since the subprocess library currently is rather primitive, a copy of the Fortran code in PYTHIA 6.4 is provided in the `pythia6` subdirectory, so that hard processes can be fed in from it.

For output to the HEPMC event record format [8], an interface is provided in the `hepmcinterface` subdirectory.

The installation procedure is described in a `README` file, and involves a `Makefile` and either of the two files `config.csh` and `config.sh`. The `libpythia8.a`, `libpythia6.a` and `libhepmcinterface.a` libraries are put in the `lib` subdirectory. The latter two may or may not be built, depending on choice of values in the `config` files.

Finally, some examples of main programs, along with input files to them, are found in the `examples` subdirectory. A `Makefile` there will allow you to build executables (placed in the `bin` directory, but with links from `examples`).

Important warnings

Playing with the files in the `examples` subdirectory is encouraged, to familiarize oneself with the program. Modifying the config files may be required during installation. For the rest, files should not be modified, at least not without careful consideration of possible consequences.

In particular, the `.xml` files are set read-only, and should not be tampered with. Interspersed in them, there are lines beginning with `<flag`, `<mode`, `<parameter`, or `<a`. They contain instructions from which Settings builds up its database of user-accessible variables, see further below. Any stupid changes here will cause difficult-to-track errors!

Further, sometimes you will see two question marks, "??", in the text or code. This is for internal usage, to indicate loose ends or preliminary thoughts. Please disregard.

3 Key Program Elements

3.1 The main generation class

As has already been mentioned, the `Pythia` class is the main means of communication between the user and the event generation process. We here present the key methods that should be used, put in context.

Already at the top of the main program file you need to include the proper header file

```
#include "Pythia.h"
```

To simplify typing, it also makes sense to declare

```
using namespace Pythia8;
```

Given this, the first step in the main program is to create a generator object, e.g. with

```
Pythia pythia;
```

In the following we will assume that the `pythia` object has been created with this name, but of course you are free to pick another one.

The `Pythia` constructor will initialize the default values for the `Settings` and the `ParticleDataTable` data bases. In addition, the old Fortran 77 Pythia 6 obtained its default values already at the loading stage. These data can now be modified in a number of ways, but most conveniently by the two methods

```
    pythia.readString(string);
```

for changing a single variable, and

```
    pythia.readFile(fileName);
```

for changing a set of variables, one per line in the input file. The allowed form for a string/line will be explained as we go along to consider the data bases in more detail. A short summary is that

- a non-alphanumeric first non-blank character signals a comment line,
- a `Pythia6:command = value` string is taken as a command to Pythia 6,
- one like `id:command = value`, with `id` a number, as a modification to the particle database, and
- anything else as a setting.

If the database in question then fails to parse the string, a warning is normally issued, but in the end the relevant line is again considered as a comment and ignored.

To check which changes have actually taken effect in the settings database, it is convenient to insert a

```
    pythia.settings.listChanged();
```

while a complete listing with

```
    pythia.settings.listAll();
```

gives a survey of all the possibilities. For the particle data there are corresponding methods either to show only those particles that were changed (one way or another)

```
    pythia.particleData.listChanged();
```

or to give the complete listing

```
    pythia.particleData.listAll();
```

A specific particle can be listed with `list(id)`, or a set as a `vector<int>` of `id`'s.

At this stage you can also insert hooks to some external facilities.

At the initialization stage all remaining details of the generation are to be specified. The `init` method allows a few different input formats, so you can pick the one convenient for you:

```
    pythia.init( idA, idB, eA, eB);
```

lets you specify the identities and energies of the two incoming beam particles, with A (B) assumed moving in the $+z$ ($-z$) direction.

```
    pythia.init( idA, idB, eCM);
```

is similar, but you specify the CM energy, and you are assumed in the rest frame.

```
    pythia.init( LHAinit*, LHAevnt*);
```

assumes LHA initialization information is available in an `LHAinit` class object, and that LHA event information will be provided by the `LHAevnt` class object, see below.

It is when the `init` call is executed that all the settings values are propagated to the various program elements, and in some cases used to precalculate quantites that will be used at later stages of the generation. Further settings changed after the `init` call will be ignored (unless methods are used to force a partial or complete re-initialization). By contrast, the particle properties database is queried all the time, and so a later change would take effect immediately, for better or worse.

The bulk of the code is concerned with the event generation proper. However, all the information on how this should be done has already been specified. Therefore only a command

```
pythia.next();
```

is required to generate the next event. This method would be located inside an event loop, where a required number of events are to be generated. However, the `pythia.next()` command returns a `bool` value telling whether the generation succeeded or not. This is useful e.g. when a file of LHA events comes to an end. Other unsolvable errors may also result in a `false` value, in which case that event should be skipped. Most internal warning and error messages are related to problems that can be fixed before it gets that far, however, e.g. by taking a step back and trying again.

The key output of the `pythia.next()` command is the event record found in `pythia.event()`, see below. A process-level summary of the event is stored in `pythia.process()`.

At the end of the generation process, you can call

```
pythia.statistics();
```

to get some run statistics, both on cross sections for the subprocesses generated and on the number of errors and warnings issued.

3.2 The event record

The `Event` class for event records is nothing but a wrapper for a vector of `Particle`s, to first approximation. This vector can expand to fit the event size. The index operator is overloaded, so that `event[i]` corresponds to the `i`'th particle of an `Event` object called `event`. For instance, given that the PDG identity code [9] of a particle is provided by the `id()` method, `event[i].id()` returns the identity of the `i`'th particle.

In this section, first the `Particle` methods are surveyed, and then the further aspects of the event record.

3.2.1 The particle

A `Particle` corresponds to one entry/slot/line in the event record. Its properties therefore is a mix of ones belonging to a particle-as-such, like its identity code or four-momentum, and ones related to the event-as-a-whole, like which mother it has.

The following properties are stored for each particle, listed by the member functions you can use to extract the information:

- `id()` : the identity of a particle, according to the PDG particle codes.
- `status()` : status code. The full set of codes provides info on where and why a given particle was produced. The key feature, however, is that positive status codes correspond to remaining particles of the event, while negative codes give ones that have been processed further. If the latter has happened, the reason can be gleaned by considering the status code of daughters.

- `mother1(), mother2()` : the indices in the event record where the first and last mothers are stored, if any. A few different cases are possible, to allow for one or many mothers. The `motherList(i)` method of the `Event` class can return a vector with all the mothers, based on this info.
- `daughter1(), daughter2()` : the indices in the event record where the first and last daughters are stored, if any. A few different cases are possible, to allow for one or many daughters. The `daughterList(i)` method of the `Event` class can return a vector with all the daughters, based on this info.
- `col(), acol()` : the colour and anticolour tags, LHA style.
- `px(), py(), pz(), e()` : the particle four-momentum components (in GeV/c or GeV), alternatively extracted as a `Vec4 p()`.
- `m()` : the particle mass (in GeV/c^2).
- `scale()` : the scale at which a parton was produced (in GeV); model-specific but relevant in the processing of an event.
- `xProd(), yProd(), zProd(), tProd()` : the production vertex coordinates (in mm or mm/c), alternatively extracted as a `Vec4 vProd()`.
- `tau()` : the proper lifetime (in mm/c).

The same method names, with a value inserted between the brackets, set these quantities.

In addition, a number of derived quantities can easily be obtained, but cannot be set, such as:

- `statusAbs()` : the absolute value of the status code.
- `remains()` : true for a remaining particle, i.e. one with positive status code, else false.
- `pT(), pT2()` : (squared) transverse momentum.
- `mT(), mT2()` : (squared) transverse mass.
- `pAbs(), pAbs2()` : (squared) three-momentum magnitude.
- `theta(), phi()` : polar and azimuthal angle.
- `thetaXZ()` : angle in the (p_x, p_z) plane, relative to $+z$ axis, between $-\pi$ and $+\pi$.
- `pPlus(), pMinus()` : $E \pm p_z$.
- `y(), eta()` : rapidity and pseudorapidity.
- `xDec(), yDec(), zDec(), tDec()` : the decay vertex coordinates, alternatively extracted as a `Vec4 vDec()`.

Each `Particle` contains a pointer to the respective `ParticleDataEntry` object in the particle data tables. This gives access to properties of the particle species as such. It is there mainly for convenience, and should be thrown if an event is written to disk, to avoid any problems of object persistency. This pointer is used by member functions such as:

- `name()` : the name of the particle, as a string.
- `nameWithStatus()` : as above, but for negative-status particles the name is given in brackets, to emphasize that they are intermediaries.
- `m0()` : the nominal mass of the particle species (while `mass()` selects a new mass, according to a Breit–Wigner, where relevant).
- `colType()` : 0 for colour singlets, 1 for triplets, -1 for antitriplets and 2 for octets.
- `charge(), icharge()` : charge, and three times it to make an integer.
- `isCharged(), isNeutral()` : `bool`s whether `icharge()` is different from 0 or not.

3.2.2 Other methods in the event record

While the `Particle` vector is the key component of an `Event`, a few further methods are available. The event size can be found with `size()`, i.e. valid particles are stored in the range $0 \leq$`i`$<$ `event.size()`. Line 0 is used to represent the event as a whole, with its total four-momentum and invariant mass, but does not form part of the event history. It is only with lines 1 and 2, which contain the two incoming beams, that the history tracing begins. That way unassigned mother and daughter indices can be put 0 without ambiguity.

A listing of the whole event is obtained with `list()`. The basic identity, status, mother, daughter, colour, four-momentum and mass data are always given, but switches can be set to provide further information, on the complete lists of mothers and daughters, and on production vertices.

The user would normally be concerned with the `Event` object that is a public member `event` of the `Pythia` class. Thus `pythia.event[i].id()` would be used to return the identity of the `i`'th particle, and `pythia.event.size()` to give the size of the event record.

A `Pythia` object contains a second event record for the hard process alone, similar to the LHA process specification, called `process`, used as input for the generation of the complete event. Thus one may e.g. call either `pythia.process.list()` or `pythia.event.list()`. To distinguish those two rapidly at visual inspection, the "Pythia Event Listing" header is printed out differently, adding either "(hard process)" or "(complete event)".

There are also a few methods with an individual particle index `i` as input, but requiring some search operations in the event record, and therefore not possible to define as methods of the `Particle` class. An incomplete list is:

- `motherList(i), daughterList(i), sisterList()` : returns a `vector<int>` containing a list of all the mothers, daughters or sisters of a particle. This list may be empty or arbitrarily large, and is given in ascending order.
- `iTopCopy(i), iBotCopy(i)` : traces "carbon copies" (i.e. the same particle, but shifted by various recoil effects) of the particle at index `i` up to its top mother or down to its bottom daughter. If there are no such carbon copies, `i` itself will be returned.
- `iTopCopyId(i), iBotCopyId(i)` : also trace top mother and bottom daughter, but do not require carbon copies, only that one can find an unbroken chain, of mothers or daughters, with the same flavour `id` code. The behaviour for common quarks and gluons may be unpredictable (what to do in a $\mathrm{g} \to \mathrm{gg}$ branching?), but should work well for "rare" particles.

One data member in an Event object is used to keep track of the largest `col()` or `acol()` tag set so far, so that new ones do not clash.

The event record also contains a vector of junctions, which often is empty or else contains only a very few per event. Methods are available to add further junctions or query the current junction list. This is only for the expert user, however, and is not discussed further here.

3.3 Other event information

A set of one-of-a-kind pieces of event information is stored in the `Info info` object in the `Pythia` class. This is mainly intended for processes generated internally, while it is likely not available for external processes.

You can use `pythia.info.method()` to extract e.g. the following information:

- `list()` : list some information on output.
- `eCM(), s()` : the cm energy and its square.
- `name(), code()` : the name and code of the subprocess.
- `id1(), id(2)` : the identities of the two partons coming in to the hard subprocess.
- `x1(), x2()` : x fractions of the two partons coming in to the hard subprocess.
- `pdf1(), pdf2(), Q2pdf()` : parton densities $x\,f(x,Q^2)$ evaluated for the two incoming partons, and the associated Q^2 scale.
- `mHat(), sHat(), tHat(), uHat()` : the invariant mass of the hard subprocess and the Mandelstam variables.
- `pTHat(), thetaHat()` : transverse momentum and polar scattering angle of the hard subprocess.

In other classes there are also methods that can be called to do a sphericity analysis or search for jets with a simple cone jet finder. These take the event record as input.

3.4 Databases

Inevitably one wants to be able to modify the default behaviour of a generator. Currently there are two PYTHIA 8 databases with modifiable values. One deals with general settings, the other specifically with particle data. In addition, PYTHIA 6 variables can be set for the hard-process generation.

3.4.1 Settings

We distinguish three kinds of user-modifiable variables, by the way they have to be stored:

1. Flags are on/off switches, and are stored as `bool`.
2. Modes correspond to a finite enumeration of separate options, and are stored as `int`.
3. Parameters take a continuum of values, and are stored as `double`.

Collectively the three above kinds of variables are called settings (but if you have a better term, please let me know). One could imagine strings as a fourth kind, but so far there has been no need.

The `Settings` class keeps track of all the flags, modes and parameters in the program. As such, it serves the other program elements from one central repository. The `Settings` class is purely static, i.e. exists only as one global copy, that you can interact with directly by `Settings::command(argument)`. However, a `settings` object of the `Settings` class is a public member of the `Pythia` class, so an alternative notation would be `pythia.settings.command(argument)`.

Each variable stored in `Settings` is associated with a few pieces of information:

- The variable name, of the form `class:name` (or `file:name`, usually these agree), e.g. `TimeShower:pTmin`. The class/file part often, but not always, specifies the only part of the program where the setting is used.
- The default value, set in the original declaration, and intended to represent a reasonable choice.
- The current value, which differs from the default when the user so requests.

- An allowed range of values, represented by meaningful minimum and maximum values. This has no sense for a flag (and is not used there), is usually rather well-defined for a mode, but less so for a parameter. Either of the minimum and maximum may be left free, giving an open-ended range. Often the allowed range exaggerates the degree of our current knowledge, so as not to restrict too much what the user can do.

Technically, the Settings class is implemented with the help of three separate maps, one for each kind of variable, with the name used as key.

Many methods exist that can be used to set or get values. Here we only describe the two most convenient input ones, `readString` and `readFile`.

You can directly set values with `Settings::readString(string)`, where both the variable name and the value are contained inside the character string, separated by blanks and/or an equal sign, e.g. `"TimeShower:pTmin = 1.0"`. A string not beginning with an alphanumeric character is considered as a comment and ignored. Therefore inserting an initial !, #, $, %, or another such character, is a good way to comment out a command. For non-commented strings, the match of the name to the database is case-insensitive. Strings that do begin with a letter and still are not recognized cause a warning to be issued, unless a second argument `false` is used in the call. Any further text after the value is ignored, so the rest of the string can be used for any comments. Values below the minimum or above the maximum are set at the respective border. For `bool` values, the following notation may be used interchangeably: `true` = `on` = `yes` = `ok` = 1, while everything else gives `false` (including but not limited to `false`, `off`, `no` and 0).

As already mentioned, the `Pythia` class contains a `readString` method that hands on to the similarly-named `Settings` method, or to corresponding methods in `ParticleData` or PYTHIA 6, the case being, and therefore offers the most flexible form:

```
  pythia.readString("TimeShower:pTmin = 1.0");
```

The `readString` method is convenient for changing one or two settings, but becomes cumbersome for more extensive modifications. In addition, a recompilation and relinking of the main program is necessary for any change of values. Alternatively, the changes can therefore be collected in a file, where each line corresponds to a character string (without the quotes) of the same kind as above. Thus we could have a file, `main.cmnd` say, collecting commands such as

```
   TimeShower:pTmin = 1.0
   PartonLevel:MI = off !  no multiple interactions
   SpaceShower:alphaSorder = 2 !  second order running alpha-strong
```

The whole file can then be read and processed with `Settings::readFile("main.cmnd")`. A file read this way would only process commands specifically for `Settings`, while the alternative `pythia.readFile("main.cmnd")` allows commands also for `ParticleData` and PYTHIA 6 to be freely mixed in, and so is preferable.

You may obtain a listing of all variables in the database by calling

```
   Settings::listAll();
```

The listing is strictly alphabetical, which at least means that names from the same file/class are kept together, but otherwise may not be so well-structured: important and unimportant ones will appear mixed. A more relevant alternative is

```
   Settings::listChanged();
```

where you will only get those variables that differ from their defaults.

Table 1: Currently implemented processes.

Flag	Process	Code
`SoftQCD:all`	sum of below	102 – 105
`SoftQCD:elastic`	$AB \to AB$	102
`SoftQCD:singleDiffractive`	$AB \to XB$, $AB \to AX$	103, 104
`SoftQCD:doubleDiffractive`	$AB \to X_1X_2$	105
`HardQCD:all`	sum of below	111 – 118
`HardQCD:gg2gg`	$\mathrm{gg} \to \mathrm{gg}$	111
`HardQCD:gg2qqbar`	$\mathrm{gg} \to \mathrm{q\overline{q}}$	112
`HardQCD:qg2qg`	$\mathrm{qg} \to \mathrm{qg}$	113
`HardQCD:qq2qq`	$\mathrm{qq'} \to \mathrm{qq'}$	114, 115, 116
`HardQCD:qqbar2qqbarNew`	$\mathrm{q\overline{q}} \to \mathrm{q'\overline{q}'}$	117
`HardQCD:qqbar2gg`	$\mathrm{q\overline{q}} \to \mathrm{gg}$	118
`PromptPhoton:all`	sum of below	131 – 135
`PromptPhoton:qg2qgamma`	$\mathrm{qg} \to \mathrm{q}\gamma$	131
`PromptPhoton:qqbar2ggamma`	$\mathrm{q\overline{q}} \to \mathrm{g}\gamma$	132
`PromptPhoton:gg2ggamma`	$\mathrm{gg} \to \mathrm{g}\gamma$	133
`PromptPhoton:qqbar2gammagamma`	$\mathrm{q\overline{q}} \to \gamma\gamma$	134
`PromptPhoton:gg2gammagamma`	$\mathrm{gg} \to \gamma\gamma$	135

3.4.2 Processes

Currently only very few processes are implemented internally in Pythia 8. These that are there are switched on an off via the ordinary settings machinery, using the flags in Table 1. By default all processes are off. The processes are grouped, so that a whole group can be turned on by a `GroupName:all = on` command, then overriding the individual flags.

Currently it is not possible to mix internal processes with those from Pythia 6 or those input via the Les Houches accord. Specifically, if any `Pythia6:` command is given, even if not referring to a process, then Pythia 6 will be given control over the process generation.

Each process is assigned an integer code. This code is not used in the internal administration of events, but only intended to allow a simpler user separation of different processes. Also the process name is available, as a string.

For many processes it makes sense to apply phase space cuts. The ones currently available (in the `Settings` database) include

- `PhaseSpace:mHatMin, PhaseSpace:mHatMax` : the range of invariant masses of the scattering process.
- `PhaseSpace:pTHatMin, PhaseSpace:pTHatMax` : the range of transverse momenta in the rest frame of the process for $2 \to 2$ processes.

Note that the SoftQCD processes do not use any cuts but generate their respective cross sections in full.

3.4.3 Particle data

The `ParticleDataTable` class is purely static, i.e. you can interact with it directly by `ParticleDataTable::command(argument)`. However, a `particleData` object of the `ParticleDataTable` class is a public member of the `Pythia` class, so an alternative notation would be `pythia.particleData.command(argument)`. Further, for some of the most frequent user tasks, `Pythia` methods have been defined, so that `pythia.command(argument)` would work, see further below.

This database does not have its final form, since further properties will need to be added when processes with resonance decays are implemented. Currently the following particle properties are stored in the `ParticleDataTable` for a given PDG particle identity code `id`, here presented by the name used to access this property:

- `hasAnti(id)` : `bool` whether a distinct antiparticle exists or not.
- `name(id)` : particle and antiparticle names are stored separately, the sign of `id` determines which of the two is returned, with "void" used to indicate the absence of an antiparticle.
- `charge3(id)` : three times the charge (to make it an integer), alternatively `charge(id)`, which is a `double` equal to `charge3(id)`/3.
- `colType(id)` : the colour type, with 0 uncoloured, 1 triplet, -1 antitriplet and 2 octet.
- `m0(id)` : the nominal mass m_0 (in GeV).
- `constituentMass(id)` : the constituent mass for a quark, hardcoded as $m_{\mathrm{u}} = m_{\mathrm{d}} = 0.325$, $m_{\mathrm{s}} = 0.50$, $m_{\mathrm{c}} = 1.60$ and $m_{\mathrm{b}} = 5.0$ GeV, for a diquark the sum of quark constituent masses, and for everything else the same as the ordinary mass.
- `width(id)` : the width Γ of the Breit-Wigner distribution (in GeV).
- `range(id)` : the allowed mass range generated by the Breit-Wigner, $|m - m_0| <$ `range` (in GeV).
- `tau0(id)` : the nominal proper lifetime τ_0 (in mm/c).
- `mayDecay(id)` : a flag telling whether a particle species may decay or not, offering the main user switch (whether a given particle of this kind then actually will decay also depends on other flags in the `ParticleDecays` class).

Similar methods can also be used to set these properties.

Each particle kind in the `ParticleDataTable` also has a a vector of `DecayChannels` associated with it. The following properties are stored for each decay channel:

- `branchingRatio()` : the branching ratio.
- `modeME()` : the mode of processing this channel, possibly with matrix-element information.
- `multiplicity()` : the number of decay products in a channel, at most 8.
- `product(i)` : a list of the decay products, 8 products 0 $\leq$`i`$<$ 8, with trailing unused ones set to 0.

The original particle data and decay table is read in from the `ParticleData.xml` file. Like for the `Settings` class, the two most convenient methods to change that data are called `readString` and `readFile`, and the pattern is the same: `readString` takes as argument a character string that contains one change, while `readFile` takes as argument a file name, where many changes can be stored, one per line. If you use the methods in the ParticleDataTable class, e.g. ParticleDataTable::readFile("main.cmnd") commands are specific to this database, while the similarly-named methods in Pythia, e.g. pythia.readFile("main.cmnd"), allow a free mixture of commands to Settings, ParticleDataTable and Pythia6. Comments about case-insensitivity, alternative notation for bool values, and more, also carries over.

It is thus only the form of the particle properties that needs to be specified slightly differently from those of the Settings variables. The general form is id:property = value. The id part is the standard PDG particle code, and property is one of the ones already described above, with a few minor differences: name, antiName, charge3, colType, m0, width, range, tau0 and mayDecay. A few examples would be:

```
    111:name = piZero
    3122:mayDecay = false !  Lambda0 stable
    431:tau0 = 0.15 !  D_s proper lifetime
```

For major changes of the properties of a particle, the above one-at-a-time changes can become rather cumbersome. Therefore a few extended input formats are available, where a whole set of properties can be given after the equal sign, separated by blanks and/or by commas. One line like

```
  id:all = name antiName charge3 colType m0 width range tau0 mayDecay
```

replaces all the current information on the particle itself, but keeps its decay channels, if any, while using new instead of all also removes any previous decay channels.

In order to change the decay data, the decay channel number needs to be given right after the particle number, i.e. the command form becomes id:channel:property = value. Recognized properties are branchingRatio, modeME and products, where the latter expects a list of all the decay products, separated by blanks and/or by commas, up until the end of the line, or until a non-number is encountered.. The property all will replace all the information on the channel, i.e.

```
  id:channel:all = branchingRatio modeME products
```

To add a new channel at the end, use

```
  id:addChannel = branchingRatio modeME products
```

It is currently not possible to remove a channel selectively, but setting its branching ratio vanishing is as effective.

When adding new channels or changing branching ratios is general, note that, once a particle is to be decayed, the sum of then existing branching ratios is always rescaled to unity. The construction id:rescaleBR = newSumBR may be used to rescale all the then provided decay channels to a total branching ratio of newSumBR, e.g. to make place for new decay channels to make up the remainder. The id:channel:rescaleBR = factor multiplies the branching ratio of a given channel by the factor provided.

You may obtain a listing of all the particle data by calling ParticleDataTable::listAll(). The listing is by increasing id number. To list only those particles that have been changed, instead use listChanged(). To list only one specific particle id, use list(id). It is also possible to list a vector<int> of id's.

3.4.4 PYTHIA 6

In order to give access to the Fortran Pythia process library at runtime (and not only by writing/reading event files) an interface is provided to C++. This interface is residing in `Pythia6.h`, while the Fortran code is in `Pythia6.f`. The latter should normally be the most recent Fortran Pythia version, but must be at least 6.314, since this is the first version that allows internally generated processes to be stored in the LHA format.

The `readString` and `readFile` methods in `Pythia` can be used to modify the performance of the Pythia 6 generator. The name of any variable must be preceded by `"Pythia6:"` to distinguish it from normal Pythia 8 variables either in the `Settings` or the `ParticleDataTable` classes. An example would be

```
    pythia.readString("Pythia6:ckin(3) = 10.");
```

The first part is peeled off, to give the string `"ckin(3) = 10."`. This string is handed as argument to `pygive`, accessed via an `extern "C" pygive_` interface. Thus all parameters that could be set with the Fortran routine `PYGIVE` can also be set by `readString` and `readFile`. Only those commands that influence the generation of the hard process have any impact, however, since this is the only part of the Fortran code that is used.

All hard Pythia 6 processes should be available, at least to the extent that they are defined for beams of protons and antiprotons, which are the only ones fully implemented in Pythia 8 so far. Soft processes, i.e. elastic and diffractive scattering, as well as minimum-bias events, require a different kinematics machinery, and are not currently available.

3.5 Hooks to external programs

While Pythia 8 is intended to be self-contained, to the extent that you can run it without reference to any external library, often you do want to make use of other programs that are specialized on some aspect of the generation process.

3.5.1 The Les Houches interface

The LHA [7] for user processes is the standard way to input parton-level information from a matrix-elements based generator into Pythia. The conventions for which information should be stored has been defined in a Fortran context, as two commonblocks. Here a C++ equivalent is defined, as two separate classes.

The `LHAinit` and `LHAevnt` classes are base classes, containing reading and printout methods, plus a pure virtual method each. Derived classes have to provide these two virtual methods to do the actual work. Currently the only derived classes are for reading information at runtime from the respective Fortran commonblock or for reading it from previously produced Pythia 6.3 files.

The `LHAinit` class stores information equivalent to the `/HEPRUP/` commonblock, as required to initialize the event generation chain. The main difference is that the vector container now allows a flexible number of subprocesses to be defined. For the rest, names have been modified, since the 6-character-limit does not apply, and variables have been regrouped for clarity, but nothing fundamental.

The pure virtual function `set()` has to be implemented in the derived class, to set relevant information when called. It should return false if it fails. Information can be set by the following methods:

- `beamA( identity, energy, pdfGroup, pdfSet)` : sets the properties of the first incoming beam (`IDBMUP(1)`, `EBMUP(1)`, `PDFGUP(1)`, `PDFSUP(1)`), and similarly for beamB. The parton distribution information defaults to zero, meaning that internal sets are used.

- strategy(choice) : sets the event weighting and cross section strategy (IDWTUP of the Fortran version).
- process(idProcess, xSec, xErr, xMax) : sets info on an allowed process (LPRUP, XSECUP, XERRUP, XMAXUP). Each new call will append one more entry to the list of processes.

The information can be printed using the overloaded << operator, e.g.
cout << LHAinitObject; .

The LHAevnt class stores information equivalent to the /HEPEUP/ commonblock, as required to hand in the next parton-level configuration for complete event generation. The main difference is that the vector container now allows a flexible number of partons to be defined. For the rest, names have been modified, since the 6-character-limit does not apply, and variables have been regrouped for clarity, but nothing fundamental.

The LHA is based on Fortran arrays beginning with index 1, and mother information is defined accordingly. In order to be compatible with this convention, the zeroth line of the C++ particle array is kept empty, so that index 1 also here corresponds to the first particle. One small incompatibility is that the size() method returns the full size of the particle array, including the empty zeroth line, and thus is one larger than the true number of particles (NUP).

The pure virtual function set() has to be implemented in the derived class, to set relevant information when called. It should return false if it fails, e.g. if the supply of events in a file is exhausted. Information can be set by the following methods:

- process(idProcess, weight, scale, alphaQED, alphaQCD) : tells which kind of process occurred, with what weight, at what scale, and which α_{em} and α_{s} were used (IDPRUP, XWTGUP, SCALUP, AQEDUP, AQCDUP of the Fortran version). This method also resets the size of the particle list, and adds the empty zeroth line, so it has to be called before the particle method below.
- particle(id, status, mother1, mother2, colourTag1, colourTag2, p_x, p_y, p_z, e, m, tau, spin) : gives the properties of the next particle handed in (IDUP, ISTUP, MOTHUP(1,..), MOTHUP(2,..), ICOLUP(1,..), ICOLUP(2,..), PUP(J,..), j from 1 through 5, VTIMUP, SPINUP) .

The information can be printed using the overloaded << operator, e.g.
cout << LHAevntObject; .

The LHAinitFortran and LHAevntFortran are two derived classes, containing set members that read the respective Fortran commonblock for initialization and event information. This can be used for a runtime link to a Fortran library, and is the mechanism used (transparently to the user) to link to the PYTHIA 6 process library.

The LHAinitPythia6 and LHAevntPythia6 are two other derived classes, that can read files with initialization and event information, assuming that the files have been written in the PYTHIA 6 format. The respective file name has to be given as argument at the instantiation. Even if this specific implementation is not particularly interesting — the same functionality can be obtained much simpler with the runtime interface to PYTHIA 6 — it can serve as a template for reading files written by more interesting other programs.

Once LHAinit and LHAevnt objects have been created, pointers to those should be handed in with the `init` call, `pythia.init( LHAinit*, LHAevnt*)`.

3.5.2 Parton distribution functions

The `PDF` class is the base class for all parton distribution function parametrizations, from which specific `PDF` classes are derived. The choice of which PDF to use is made by a switch in the `Pythia` class. Thus, a priori, there is no need for a normal user to study this class. Currently the selection is very limited; for protons only CTEQ 5L (default) and GRV 94L are available. It is therefore natural to want to interface to external PDF libraries, and then the structure must be understood.

The constructor requires the incoming beam species to be given: even if used for a proton PDF, one needs to know whether the beam is actually an antiproton. This is one of the reasons why `Pythia` always defines two PDF objects in an event, one for each beam.

Once a PDF object has been constructed, call it `pdf`, the main method used in the event generation is `pdf.xf( id, x, Q2)`, which returns $xf_{\mathrm{id}}(x, Q^2)$, properly taking into account whether the beam is an antiparticle or not.

Whenever the `xf` member is called with a new flavour, x or Q^2, the `xfUpdate` member is called to do the actual updating. This is the only pure virtual method, that therefore must be implemented in any derived class. It may either update that particular flavour or all flavours at this (x, Q^2) point. The choice is to be made by the producer of a given set, based on what he/she deems most effective, given that sometimes only one flavour need be evaluated, and about equally often all flavours are needed at the same x and Q^2. Anyway, the latest value is always kept in memory. This is the other reason why `Pythia` has one separate `PDF` object for each beam.

Once you have created two distinct `PDF` objects, `pdfA` and `pdfB`, you should supply pointers to these as arguments in a `PDFptr` method call

```
    pythia.PDFptr( pdfA*, pdfB*);
```

This has to be made before the `pythia.init(...)` call.

3.5.3 External decays

While `Pythia` is set up to handle any particle decays, decay products are often (but not always) distributed isotropically in phase space, i.e. polarization effects and nontrivial matrix elements usually are neglected. Especially for the τ lepton and for some B mesons it is therefore common practice to rely on dedicated decay packages.

To this end, `DecayHandler` is a base class for the external handling of decays. The user-written derived class is called if a pointer to it has been given with the `pythia.decayPtr( DecayHandler*, vector<int>)` method. The second argument to this method should contain the `id` codes of all the particles that should be decayed by the external program. It is up to the author of the derived class to send different of these particles on to separate packages, if so desired.

There is only one pure virtual method in `DecayHandler`, to do the decay:
`decay( idProd, mProd, pProd, iDec, event)`. Here

- `idProd` is a `vector<int>` of particle PDG identity codes,
- `mProd` is a `vector<double>` of their respective masses (in GeV), and

- `pProd` is a `vector<Vec4>` of their respective four-momenta (components `px()`, `py()`, `pz()` and `e()`),
- `iDec` is the position in the event record of the decaying particle, and
- `event` is a read-only reference to the complete event record, so that the full prehistory of the decaying particle can be extracted, should this be of interest for the decay treatment.

When the `decay` method is called, the three vectors each have size one, so that `idProd[0]`, `mProd[0]` and `pProd[0]` contain information on the particle that is to be decayed. When the decay is done, the vectors should have increased by the addition of all the decay products, starting at index 1. Even if initially defined in the rest frame of the mother, the products should have been boosted so that their four-momenta add up to the `pProd[0]` of the decaying particle. (Given a `Vec4 pNew` defined in the decaying particle rest frame, the boost can be performed by `pNew.bst(pProd[0])`.)

The routine should return `true` if it managed to do the decay and `false` otherwise. In the latter case `Pythia` will try to do the decay itself. Thus one may implement some decay channels externally and leave the rest for `Pythia`, assuming the `Pythia` decay tables are adjusted accordingly.

Note that the decay vertex is always set by `Pythia`, and that B–$\overline{\mathrm{B}}$ oscillations have already been taken into account, if they were switched on. Thus the decaying code `idProd[0]` may be the opposite to the produced one, stored in `event[iDec].id()`.

3.5.4 Random number generators

`RndmEngine` is a base class for the external handling of random number generation. The user-written derived class is called if a pointer to it has been handed in with the `pythia.rndmEnginePtr( RndmEngine*)` method. Since the default Marsaglia-Zaman algorithm is quite good, there is absolutely no physics reason to replace it, but this may still be required for consistency with other program elements in big experimental frameworks.

There is only one pure virtual method in `RndmEngine`, called `flat()`, that should return one random number uniformly distributed in the range between 0 and 1 each time it is called.

Note that methods for initialization are not provided in the base class, in part since input parameters may be specific to the generator used, in part since initialization is likely to be taken care of externally to `Pythia`.

4 Getting Going

After you downloaded the `pythia8048.tgz` package from the PYTHIA webpage, you can unpack it with `tar xvfz pythia8048.tgz`. The rest of the installation procedure is described in the `README` file. It is assumed you are on a Linux system, so far no multiplatform support.

After this, the main program is up to the user to write. However, sample main programs are provided in the `examples` subdirectory, currently `main01` through `main12`. These programs are included to serve as inspiration when starting to write your own program, by illustrating the principles involved. However, clearly they are not intended to be realistic.

Frequently commands to the `Settings`, `ParticleDataTable` or `Pythia6` machineries are collected in a single "cards file", separate from the main program proper, so that minor changes can be made without any recompilation. It is then convenient to collect in the same place some run parameters, such as the number of events to generate, that could be used inside the main program. Whether they actually are used is up to the author of a main program to decide.

The following variables have been defined, and can be extracted with the `Settings` methods:

- `Main:numberOfEvents` : the number of events to be generated.
- `Main:numberToList` : the number of events to list, at the beginning of the run.
- `Main:numberToShow` : print the number of events generated so far, to show how the run is progressing, once every `numberOfEvents/numberToShow` events.
- `Main:timesAllowErrors` : abort the run after this many errors have been found.
- `Main:showChangedSettings` : print a list of the changed flag/mode/parameter settings.
- `Main:showAllSettings` : print a list of all flag/mode/parameter settings.
- `Main:showChangedParticleData` : print a list of changed particle and decay data.
- `Main:showAllParticleData` : print a list of all particle and decay data.
- `Main:idBeamA, Main:idBeamB` : the PDG `id` codes for the two incoming particles.
- `Main:inCMframe` : whether collisions occur in the CM frame or not.
- `Main:eCM` : collision CM energy, if `Main:inCMframe` is true.
- `Main:eBeamA, Main:eBeamB` : the energies of the two incoming beam particles, moving in the $\pm z$ directions, if `Main:inCMframe` is false. If the particle energy is smaller than its mass then it is assumed to be at rest.

5 Outlook

As already explained in the introduction, PYTHIA 8 is still at an early stage. It is possible to set up and run various processes and get out sensible-looking event records, containing all the major physics aspects. However, none of the respective classes is fully developed or debugged. Many aspects are still missing, like a library of at least the most common processes of interest, and a coherent picture of colour flow, involving timelike and spacelike showers, multiple interactions and beam remnants. So do not bother to contact me about PYTHIA 6 features you miss in the new program; such shortcomings are already known. However, if you have comments about what *is* there — positive or negative ones — then do let me know. Several aspects could still be changed, before a first production-quality version, hopefully sometime in early 2007.

Acknowledgements

The support and kind hospitality of the SFT group at CERN is gratefully acknowledged. Discussions with Stephen Mrenna and Peter Skands have been helpful. Mikhail Kirsanov has contributed the interface to `HepMC`, the makefiles and other expertise to simplify the installation and operation of the package.

References

[1] T. Sjöstrand, Computer Physics Commun. **27** (1982) 243, **28** (1983) 229, **39** (1986) 347;
T. Sjöstrand and M. Bengtsson, Computer Physics Commun. **43** (1987) 367

[2] H.-U. Bengtsson, Computer Physics Commun. **31** (1984) 323;
H.-U. Bengtsson and G. Ingelman, Computer Physics Commun. **34** (1985) 251;
H.-U. Bengtsson and T. Sjöstrand, Computer Physics Commun. **46** (1987) 43;
T. Sjöstrand, Computer Physics Commun. **82** (1994) 74

[3] T. Sjöstrand, P. Edén, C. Friberg, L. Lönnblad, G. Miu, S. Mrenna and E. Norrbin, Computer Physics Commun. **135** (2001) 238;
T. Sjöstrand, S. Mrenna and P. Skands, LU TP 06–13, FERMILAB-PUB-06-052-CD-T [hep-ph/0603175]

[4] L. Lönnblad, Computer Physics Commun. **118** (1999) 213;
M. Bertini, L. Lönnblad and T. Sjöstrand, Computer Physics Commun. **134** (2001) 365

[5] S. Gieseke, A. Ribon, M.H. Seymour, P. Stephens and B.R. Webber, JHEP **0402** (2004) 005;
see webpage `http://hepforge.cedar.ac.uk/herwig/`

[6] see webpage `http://www.thep.lu.se/ThePEG/`

[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]

[8] M. Dobbs and J.B. Hansen, Computer Physics Comm. **134** (2001) 41

[9] Particle Data Group, S. Eidelman et al., Phys. Lett. **B592** (2004) 1