CERN-LCGAPP-2007-03

March 2007
updated July 2007

A Brief Introduction to PYTHIA 8.090

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, comprising a coherent set of physics models for the evolution from a few-body hard process to a complex multihadronic final state. While previous versions were written in Fortran, PYTHIA 8 represents a complete rewrite in C++. This is still a project under development, with some components still missing, and in need of validation and tuning of several aspects. Nevertheless, by now most of the basic structure is in place, and the user interaction is approaching its final form. Therefore the program is ready to be tried out, with feedback e.g. on interoperability with LHC software useful for preparing the first production-quality release later this year. 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, 4] is the product of almost thirty years 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 also to migrate Pythia to C++, in the process cleaning up and modernizing various aspects.

A first attempt in this direction was the Pythia 7 project [5]. However, after the Herwig++ [6] group decided to join in the development of a generic administrative structure split off from Pythia 7, ThePEG [7], work on the left-behind 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 essentially as a one-person effort, with a three-year "road map".

The version presented here is not yet tested and tuned enough to provide a realistic alternative to Pythia 6. Instead it is released to allow feedback, as part of the development and validation process. All subversions in the 8.0xx series should be viewed as development snapshots, with 8.100, sometime in late 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) [8] and with Les Houches Event Files (LHEF) [9] 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 some finer details to be added, and still to be retuned to some of the key experimental data.

The current article provides an introduction to Pythia 8 usage. The programming aspects are covered in more detail in a set of interlinked HTLM (or alternativly PHP) pages that comes with the program files, see below. Much of the physics aspects are unchanged relative to the Pythia 6 manual [4], and so we refer to it and to other physics articles for that. Instead what we here give is 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 to get going with the new program.

2 Physics Summary

This article is not intended to provide a complete description of the physics content. For this we primarily refer to the Pythia 6 manual [4] and associated literature. We would like to draw attention to some key points of difference, however. This is done in a rather rhapsodic form, with further details available on the webpages.

2.1 Hard processes

Currently the program only works with pp, $\overline{\mathrm{p}}\mathrm{p}$ and $\mathrm{e}^+\mathrm{e}^-$ incoming beams. In particular, there is no provision for ep collisions or for incoming photon beams, neither on their own nor as flux around an electron.

The list of processes currently implemented is summarized further down; it corresponds to most of the Standard-Model ones in Pythia 6. The cross-section expressions should be identical, but default scale choices have been changed, so that cross sections may be somewhat different for that reason.

The default parton distribution remains CTEQ 5L, but ones found in the LhaPdf library [10] can easily be linked in. It is now possible to use separate sets for the hard interaction, on one hand, and subsequent showers and multiple interactions, on the other.

2.2 Parton showers

The initial- and final-state algorithms are based on the new $p_\perp$-ordered evolution introduced in Pythia 6.3, while the older mass-ordered ones have not been implemented. It is now possible to have a branching of a photon to a fermion pair as part of the final-state evolution.

Already in Pythia 6 the initial-state evolution and the multiple interactions were interleaved into one common decreasing $p_\perp$ sequence. Now also the final-state evolution is interleaved with the other two. In this context, some of that final-state radiation gets to be associated with dipoles stretched between a final-state parton and the "hole" left by an initial-state one.

2.3 Multiple interactions and beam remnants

The multiple-interactions machinery as such contains the full functionality introduced in Pythia 6.3. Rescaled parton densities are defined after the first interaction, that take into account the nature of the previous partons extracted. Currently there is only one scenario for colour-reconnection in the final state; this scenario is intermediate in character between the original one and the more recent ones. The description of beam remnants is based on the new framework.

In addition to the standard QCD $2 \to 2$ processes, the framework now also includes prompt photons, charmonia and bottomonia, low-mass Drell-Yan pairs, and t-channel $\gamma^*/\mathrm{Z}^0/\mathrm{W}^\pm$ exchange as part of the process mix.

Two hard interactions can now be set in the same event, for dedicated studies of two low-rate processes in coincidence. There are no Sudakov factors included for these two interactions, similarly to normal events with one hard interaction.

2.4 Hadronization

Hadronization is based solely on the Lund string fragmentation framework; older alternative descriptions have been left out.

Particle data have been updated in agreement with the 2006 PDG tables [11]. This also includes a changed content of the scalar meson multiplet. Some charm and bottom decay tables have been obtained from the DELPHI and LHC-B collaborations.

Bose–Einstein effects are not yet implemented.

2.5 Other program components

Standardized procedures have been introduced to link the program to various external programs for specific tasks.

Some of the old jet finders and other analysis routines are made available. Also included are simple one-dimensional histograms.

3 Program Structure

3.1 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.

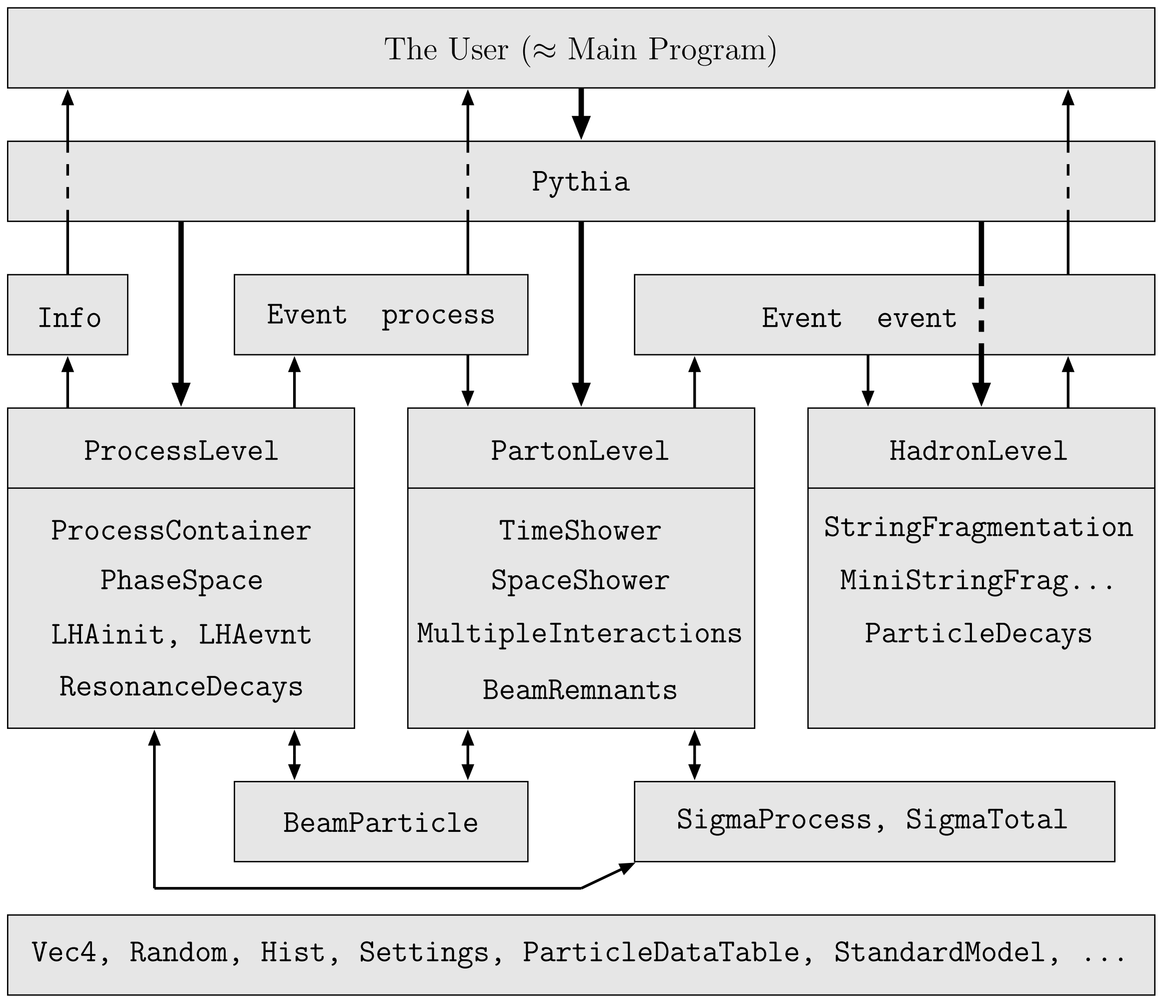

Figure 1: The relationship between the main classes in Pythia 8. The thick arrows show the flow of commands to carry out different physics tasks, whereas the thinner show the flow of information between the tasks. The bottom box contains common utilities that may be used anywhere. Obviously the picture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

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 of point 1 above, and another called `event`, that covers the full story from the incoming beams to the final hadrons. A small `Info` class keeps track of useful one-of-a-kind information, such as kinematical variables of the hard process.

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.

The process library, as well as parametrizations of total, elastic and diffractive cross sections, are used both by the hard-process selection machinery and the multiple-interactions one.

The `Settings` database keeps track of all integer, double, boolean and string variables that can be changed by the user to steer the performance of Pythia, except that

`ParticleDataTable` is its own separate database.

Finally, a number of utilities can be used just about anywhere, for Lorentz four-vectors, random numbers, jet finding and simple histograms, 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.

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.

3.2 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 the lengthier 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 `xmldoc` subdirectory have matching names, but the match is broken by the desire to group topics more by user interaction than internal operation. These files contain information on all settings and particle data, but not in a convenient-to-read format. Instead they are translated into a corresponding set of `.html` files in the `htmldoc` subdirectory and a set of `.php` files in `phpdoc`. The former set can easily be read if you open the `htmldoc/Welcome.html` file in your favourite web browser, but offers no interactivity. The latter set must be installed under a webserver, like where you put your homepage, and then provides a simple Graphical User Interface if you open the `phpdoc/Welcome.php` file in a web browser.

For output to the HepMC event record format [12], an interface is provided in the `hepmcinterface` subdirectory. There are also interfaces to allow parton distribution functions to be used from the LhaPdf library [10] and hard processes from Pythia 6.4.

The installation procedure is described in a `README` file, and involves a `configure` file and a `Makefile`. The former should be invoked with command-line arguments, alternatively be brute-force edited, to provide the path to the HepMC library if it will be used. Compiled libraries are put in the `lib` subdirectory. Default is to build archive libraries, but optionally also shared-object ones can be built. The standard setup is intended for Linux systems, but a simplified alternative is provided for Windows users.

Finally, some examples of main programs, along with input files for them, are found in the `examples` subdirectory. A `configure` file and `Makefile` there will allow you to build executables, see the `examples/README` file. As above, command-line arguments or brute-force editing allows you to set the LhaPdf and Pythia 6.4 paths, if required. The executables are 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 `configure` files may be required during installation. For the rest, files should not be modified, at least not without careful consideration of 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`, `<parm`, `<word`, `<particle`, `<channel`, or `<a`. They contain instructions from which `Settings` and `ParticleDataTable` build up their respective databases 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.

4 Key Program Elements

4.1 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. 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. Further, methods will be introduced to list all or only the changed settings and particle data.

At this stage you can also optionally hook up with some external facilities, see the section on links to external programs.

At the initialization stage all remaining details of the generation are to be specified. The `pythia.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;
```
pythia.init(fileName);
```
assumes that the file obeys the LHEF standard format and that information can be extracted from it accordingly.

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.

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`.

When problems are encountered, in `init(...)` or `next()`, they can be assigned one of three degrees of severity. Abort is the highest. In that case the call could not complete its tasks, and returns the value `false`. If in `init(...)` it is then not possible to generate any events at all. If in `next()` only the current event must be skipped. In a few cases the abort may be predictable and desirable, e.g. when a file of LHA events comes to an end. Errors are less severe, and the program can usually work around them, e.g. by backing up one step and trying again. Should that not succeed, an abort may result. Warnings are of informative character only, and do not require any corrective actions (except, in the longer term, to find more reliable algorithms).

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 aborts, errors and warnings issued.

4.2 The event record

The `Event` class for event records is not much more than a wrapper for a vector of `Particle`s. 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 [11] 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.

4.2.1 The particle

A `Particle` corresponds to one entry/slot/line in the event record. Its properties therefore mix 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, with $c = 1$), alternatively extracted as a `Vec4 p()`.
- `m()` : the particle mass (in GeV).
- `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:

- `isFinal()` : `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 (in radians).
- `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 pointer 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.
- `spinType()` : $2s + 1$, or 0 where undefined spin.
- `charge(), chargeType()` : charge, and three times it to make an integer.
- `isCharged(), isNeutral()` : `bool`s whether `chargeType()` is different from 0 or not.
- `colType()` : 0 for colour singlets, 1 for triplets, -1 for antitriplets and 2 for octets.
- `m0()` : the nominal mass of the particle species.

4.2.2 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, and only contains redundant information. When you translate to another event-record format where the first particle is assigned index 1, such as HepMC, this line should therefore be dropped so as to keep the rest of the indices synchronized. 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 optional arguments 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`. This record is 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. The most important ones are `motherList(i)`, `daughterList(i)` and `sisterList(i)`. These return 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.

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 two further sets of vectors. These are intended for the expert user only, so only a few words on each. The first is a vector of junctions, i.e. vertices where three string pieces meet. This list is often empty or else contains only a very few per event. The second is a storage area for parton indices, classified by subsystem. Such information is needed to interleave multiple interactions, initial-state showers, final-state showers and beam remnants. It can also be used in the hadronization.

4.3 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, but some of the information is also available for external processes.

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

- `list()` : list some information on the current event.
- `eCM(), s()` : the cm energy and its square.
- `name(), code()` : the name and code of the subprocess.
- `id1(), id2()` : 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(), QFac(), Q2Fac()` : parton densities $x\, f_i(x, Q^2)$ evaluated for the two incoming partons, and the associated Q/Q^2 factorization scale.
- `mHat(), sHat(), tHat(), uHat()` : the invariant mass of the hard subprocess and the Mandelstam variables for $2 \to 2$ processes.
- `pTHat(), thetaHat()` : transverse momentum and polar scattering angle of the hard subprocess.
- `alphaS(), alphaEM(), QRen(), Q2Ren()` : α_s and α_em values for the hard process, and the associated Q/Q^2 renormalization scale.
- `nTried(), nAccepted(), sigmaGen(), sigmaErr()` : the number of trial and accepted events, and the resulting estimated cross section and estimated error, in units of mb, summed over the included processes.

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

4.4 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.

There exists a number of methods to set and get values in these data bases, but most of them are of no interest to the normal user. The key method to set a new value is

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

The typical form of a string is

```
variable = value
```

where the equal sign is optional and the variable begins with a letter for settings and a digit for particle data. A string not beginning with either 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 or digit 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. For variables with an allowed range, 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. Everything else gives `false` (including but not limited to `false`, `off`, `no` and 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. The whole file can then be read and processed with a command

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

As above, comments can be freely interspersed.

4.4.1 Settings

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

1. A `Flag` is an on/off switch, and is stored as a `bool`.
2. A `Mode` correspond to an enumeration of separate options, and is stored as an `int`.
3. A `Parm` — short for parameter — takes a continuum of values, and is stored as a `double`.
4. A `Word` is a text string (with no embedded blanks) and is stored as as a `string`.

Collectively the four above kinds of variables are called settings.

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

- The variable name, of the form `class:name` (or `file:name`, or `task:name`, usually these agree), e.g. `TimeShower:pTmin`. The class/file/task 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 or a word (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 four separate maps, one for each kind of variable, with the name used as key. The default values are taken from the

`.xml` files in the `xmldoc` subdirectory. 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 is a public member of the `Pythia` class, so an alternative notation would be `pythia.settings.command(argument)`. As already mentioned, for input the `pythia.readString(...)` method is to be preferred, since it also can handle particle data. A typical example would be

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

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

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

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

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

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

4.4.2 Processes

Currently only a limited set of processes is implemented internally in PYTHIA 8, basically the Standard-Model ones found in PYTHIA 6. These that are there are switched on and off via the ordinary settings machinery, using flags of the type `ProcessGroup:ProcessName`, see Table 1. By default all processes are off. A whole group can be turned on by a `ProcessGroup:all = on` command, then overriding the individual flags.

Note that processes in the `SoftQCD` group are of a kind that cannot be input via the LHA, while essentially all other kinds could. It is not (yet) possible to mix internal processes with those input via the Les Houches Accord.

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) in particular 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 $2 \to 2$ and $2 \to 3$ processes (for each of the products).

In addition, for any resonance with a Breit-Wigner mass distribution, the allowed mass range of that particle species is taken into account, both for $2 \to 1$, $2 \to 2$ and $2 \to 3$ processes, thereby providing a further cut possibility. Note that the `SoftQCD` processes do not use any cuts but generate their respective cross sections in full.

4.4.3 Particle data

The following particle properties are stored in the `ParticleDataTable` class for a given PDG particle identity code `id`, here presented by the method used to access this property:

- `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

Table 1: Currently implemented processes. In the names, a “2” separates initial and final state, an “(s:X)”, “(t:X)” or “(l:X)” occasionally appends info on an s- or t-channel- or loop-exchanged particle X.

`ProcessGroup`	`ProcessName`
`SoftQCD`	`minBias,elastic, singleDiffractive, doubleDiffractive`
`HardQCD`	`gg2gg, gg2qqbar, qg2qg, qq2qq, qqbar2gg, qqbar2qqbarNew, gg2ccbar, qqbar2ccbar, gg2bbbar, qqbar2bbbar`
`PromptPhoton`	`qg2qgamma, qqbar2ggamma, gg2ggamma, ffbar2gammagamma, gg2gammagamma`
`WeakBosonExchange`	`ff2ff(t:gmZ), ff2ff(t:W)`
`WeakSingleBoson`	`ffbar2gmZ, ffbar2W, ffbar2ffbar(s:gm)`
`WeakDoubleBoson`	`ffbar2gmZgmZ, ffbar2ZW, ffbar2WW`
`WeakBosonAndParton`	`qqbar2gmZg, qg2gmZq, ffbar2gmZgm, fgm2gmZf qqbar2Wg, qg2Wq, ffbar2Wgm, fgm2Wf`
`Charmonium`	`gg2QQbar[3S1(1)]g +` 18 more
`Bottomonium`	`gg2QQbar[3S1(1)]g +` 18 more
`Top`	`gg2ttbar, qqbar2ttbar, qq2tq(t:W), ffbar2ttbar(s:gmZ), ffbar2tqbar(s:W)`
`SMHiggs`	`fbar2H, gg2H, gmgm2H, ffbar2HZ, ffbar2HW, ff2Hff(t:ZZ), ff2Hff(t:WW), gg2Httbar, qqbar2Httbar, qg2Hq, gg2Hbbbar, qqbar2Hbbbar, gg2Hg(l:t), qg2Hq(l:t), qqbar2Hg(l:t)`
`SUSY`	`qqbar2chi0chi0` (not yet completed)

an antiparticle.

- `hasAnti(id)` : `bool` whether a distinct antiparticle exists or not.
- `spinType(id)` : $2s + 1$ for particles with defined spin, else 0.
- `chargeType(id)` : three times the charge (to make it an integer); can also be read as a `double charge(id) = chargeType(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).
- `mWidth(id)` : the width Γ of the Breit-Wigner mass distribution (in GeV).
- `mMin(id), mMax(id)` : the allowed mass range generated by the Breit-Wigner, $m_{\mathrm{min}} < m < m_{\mathrm{max}}$ (in GeV).

- `tau0(id)` : the nominal proper lifetime τ_0 (in mm/c).
- `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.
- `mRun(id, massScale)` : the running mass for quarks, else the same as the nominal mass.
- `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 most of these properties.

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

- `onMode()` : whether a channel is on (1) or off (0), or on only for particles (2) or antiparticles (3).
- `bRatio()` : the branching ratio.
- `meMode()` : 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 ≤`i`< 8, with trailing unused ones set to 0.

The original particle data and decay table is read in from the `ParticleData.xml` file.

The `ParticleDataTable` class is purely static, i.e. exists as one global copy, that you can interact directly with 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)`. As already mentioned, for input the `pythia.readString(string)` method is to be preferred, since it also can handle settings.

It is only the form of the `string` that needs to be specified slightly differently, as `id:property = value`. The `id` part is the standard PDG particle code, i.e. a number, and `property` is one of the ones already described above, with a few minor differences: `name`, `antiName`, `spinType`, `chargeType`, `colType`, `m0`, `mWidth`, `mMin`, `mMax`, `tau0`, `mayDecay`, `isResonance`, `isVisible`, and `externalDecay`. As before, several commands can be stored as separate lines in a file, say

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

and then be read with `pythia.readFile(fileName)`.

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 spinTp chargeTp colTp m0 mWidth mMin mMax tau0
```

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. (The flags `mayDecay`, `isResonance`, `isVisible`, and `externalDecay` are in either case reset to their defaults and would have to be changed separately.)

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 `onMode`, `bRatio`, `meMode` and `products`, where the latter expects a list of all the decay products, separated by blanks, 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 = onMode bRatio meMode products
```

To add a new channel at the end, use

```
id:addChannel = onMode bRatio meMode products
```

To remove all existing channels and force decays into one new channel, use

```
id:oneChannel = onMode bRatio meMode products
```

A first `oneChannel` command could be followed by several subsequent `addChannel` ones, to build up a completely new decay table for an existing particle.
It is currently not possible to remove a channel selectively, but setting its branching ratio vanishing is as effective.

Often one may want to allow only a specific subset of decay channels for a particle. This can be achieved e.g. by a repeated use of `id:channel:onMode` commands, but there also is a set of commands that initiates a loop over all decay channels and allows a matching to be carried out. The `id:onMode` command can switch `on` or `off` all channels. The `id:onIfAny` and `id:offIfAny` will switch on/off all channels that contain any of the enumerated particles. For instance

```
23:onMode = off
23:onIfAny = 1 2 3 4 5
```

first switches off all Z^0 decay modes and then switches back on any that contains one of the five lighter quarks. Other methods are `id:onIfAll` and `id:offIfAll`, and `id:onIfMatch` and `id:offIfMatch`, where all the enumerated products must be present for a decay channel to be switched on/off. The difference is that the former two allow further non-matched particles in a decay channel while the latter two do not. There are also further methods to switch on channels selectively either for the particle or for the antiparticle.

When a particle is to be decayed, the branching ratios of the allowed channels is always rescaled to unity. There are also methods for by-hand rescaling of branching ratios.

You may obtain a listing of all the particle data by calling

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

The listing is by increasing `id` number. To list only those particles that have been changed, instead use

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

To list only one specific particle `id`, use `list(id)`. It is also possible to `list` a `vector<int>` of `id`'s.

4.5 Links 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. The HTML/PHP pages contain full information on how the different links should be set up. Here the purpose is mainly to point out the possibilities that exist.

4.5.1 The Les Houches interface

The LHA [8] 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 each a pure virtual method `set()`. 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 a Les Houches Event File (LHEF) [9].

The `LHAinit` class stores information equivalent to the `/HEPRUP/` commonblock, as required to initialize the event-generation chain. 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 `LHAinitFortran` and `LHAevntFortran` are two derived classes, containing `set()` members that read the respective LHA Fortran commonblock for initialization and event information. This can be used for a runtime link to a Fortran library, and is the mechanism used to link to the PYTHIA 6.4 process library, see below.

The `LHAinitLHEF` and `LHAevntLHEF` are two other derived classes, that can read a file with initialization and event information, assuming that the file has been written in the LHEF format. You do not need to declare these classes yourself, since a shortcut is provided by the `pythia.init(fileName)` command.

If you create `LHAinit` and `LHAevnt` objects yourself, pointers to those should be handed in with the `init(...)` call, then of the form `pythia.init(LHAinit*, LHAevnt*)`.

4.5.2 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 `Pythia6Interface.h`, and in addition the PYTHIA 6.4 library must be linked.

The following routines have been interfaced: `pygive(command)`, `pyinit( frame, beam, target, wIn)`, `pyupin()`, `pyupev()`, `pylist( mode)` and `pystat( mode)`. Details on allowed arguments are given in the PYTHIA 6.4 manual [4].

These methods can be used in context of the `LHAinitFortran` and `LHAevntFortran` classes. The existing code there takes care of converting `HEPRUP` and `HEPEUP` commonblock information from Fortran to C++, and of making it available to the PYTHIA 8 methods. What needs to be supplied are the two `LHAinitFortran::fillHepRup()` and `LHAinitFortran::fillHepEup()` methods. The first can contain an arbitrary number of `pygive(...)`, followed by `pyinit(...)` and `pyupin()` in that order. The second only needs to contain `pyupev()`. Finally, the use of `pylist(...)` and `pystat(...)` is entirely optional, and calls are best put directly in the main program.

All hard PYTHIA 6.4 processes should be available, at least to the extent that they are defined for beams of protons and antiprotons, or for e^+e^- annihilation, 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 implemented directly in Pythia 8.

4.5.3 Semi-internal processes

When you implement new processes via the Les Houches Accord you do all flavour, colour and phase-space selection externally, before your process-level events are input for further processing by Pythia. However, it is also possible to implement a new process in exactly the same way as the internal Pythia ones, thus making use of the internal phase-space selection machinery to sample an externally provided cross-section expression.

The matrix-element information has to be put in a new class that derives from one of the existing classes, `Sigma1Process` for $2 \to 1$ processes, `Sigma2Process` for $2 \to 2$ ones, and `Sigma3Process` for $2 \to 3$ ones, which in their turn derive from the `SigmaProcess` base class. Note that `Pythia` is rather good at handling the phase space of $2 \to 1$ and $2 \to 2$ processes, is more primitive for $2 \to 3$ ones and does not at all address higher multiplicities. This limits the set of processes that you can implement in this framework. The produced particles may be resonances, however, so it is possible to end up with bigger "final" multiplicities through sequential decays, and to include further matrix-element weighting in those decays.

In your new class you have to implement a number of methods. Chief among them is one to return the matrix-element weight for an already specified kinematics configuration and another one to set up the final-state flavours and colour flow of the process. Further methods exist, some of more informative character, such as providing the name of the process. Should you actually go ahead, it is strongly recommended to shop around for a similar process that has already been implemented, and to use that existing code as a template. Look for processes with the same combinations of incoming flavours and colour flows, rather than the shape of the cross section itself. With a reasonable such match the task should be of medium difficulty, without it more demanding.

Once a class has been written, a pointer of type `SigmaProcess*` to a `new` instance of your class needs to be created in the main program, and handed in with the `pythia.setSigmaPtr(...)` method. From there on the process will be handled on equal footing with internally implemented processes.

4.5.4 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. Currently the selection is very limited; for protons only CTEQ 5L (default) and GRV 94L are available. However, a built-in interface to LhaPdf library [10] allows a much broader selection, if only LhaPdf is linked together with Pythia.

Should this not be enough, it is possible to write your own derived class. 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. The `xfUpdate(...)` member is called to do the actual updating of PDF's. This is the only pure virtual method, that therefore must be implemented in any derived class.

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

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

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

A word of warning: to switch to a new PDF set implies that a complete retuning of the generator may be required, since the underlying-event activity from multiple interactions and parton showers is changed. There is an option that allows a replacement of the PDF for the hard process only, so that this is not required. Inconsistent but convenient.

4.5.5 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.setDecayPtr(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).
```

When the `decay` method is called, `idProd[0]`, `mProd[0]` and `pProd[0]` contain information on the particle that is to be decayed. When the decay is done, these vectors should have increased by the addition of all the decay products, starting at index 1.

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()`.

4.5.6 User hooks

Sometimes it may be convenient to step in during the generation process: to modify the built-in cross sections, to veto undesirable events or simply to collect statistics at various stages of the evolution. There is a base class `UserHooks` that gives you this access at a few selected places. This class in itself does nothing; the idea is that you should write your own derived class for your task. A few very simple derived classes come with the program, mainly as illustration.

There are four distinct sets of routines. Ordered by increasing complexity, rather than by their appearance in the event-generation sequence, they are:

- Ones that gives you access to the event record in between the process-level and parton-level steps, or in between the parton-level and hadron-level ones. You can study the event record and decide whether to veto this event.

- Ones that allow you to set a scale at with the combined parton-level MI+ISR+FSR downwards evolution in pT is temporarily interrupted, so the event can be studied and either vetoed or allowed to continue the evolution.
- Similar ones that instead gives you access after the first few ISR or FSR branchings of the hardest subprocess.
- Ones that gives you access to the properties of the trial hard process, so that you can modify the internal Pythia cross section by your own correction factors.

4.5.7 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.setRndmEnginePtr(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.

4.5.8 The HepMC event format

The HepMC event format [12] is a standard format for the storage of events in several major experiments. The translation from the Pythia 8 `Event` format should be done after `pythia.next()` has generated an event. Therefore there is no need for a tight linkage, but only to call the `HepMC::I_Pythia8::fill_next_event( pythia.event, hepmcevt )` conversion routine from the main program written by the user. Version 1 of HepMC makes use of the CLHep library [13] for four-vectors, while version 2 is standalone; this requires some adjustments in the interface code based on which version is used.

4.5.9 Parton showers

It is possible to replace the existing timelike and/or spacelike showers in the program by your own. This is truly for experts, since it requires a rather strict adherence to a wide set of rules.

5 Getting Going

After you downloaded the `pythia8090.tgz` package from the Pythia webpage,

`http://www.thep.lu.se/~torbjorn/Pythia.html`

link “Future”, you can unpack it with `tar xvfz pythia8090.tgz`, into a new subdirectory `pythia8090`. The rest of the installation procedure is described in the `README` file in that directory. It is assumed you are on a Linux system; so far there is hardly any multiplatform support.

After this, the main program is up to the user to write. A worksheet (found on the webpage) takes you through as step-by-step procedure, and sample main programs are provided in the `examples` subdirectory. These programs are included to serve as inspiration when starting to write your own program, by illustrating the principles involved.

The information available if you open `htmldoc/Welcome.html` in your web browser will help you expore the program possibilities further. If you install the `phpdoc` subdirectory under a web server you will also get extra help to build a file of commands to the `Settings` and `ParticleDataTable` machineries, to steer the execution of your main program.

Such "cards files" are separate from the main programs 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. Therefore some such have been predefined, e.g. `Main:numberOfEvents`. Whether they actually are used is up to the author of a main program to decide.

6 Outlook

As already explained in the introduction, Pythia 8 is not yet quite of production quality. It is possible to set up and run various processes and get out sensible event records, containing all the major physics aspects, but some further development and tuning is required to become competitive with the existing Pythia 6 code. In addition, a few aspects are still missing. Nevertheless reasonably realistic tests could be undertaken. Any feedback — positive or negative — would be most welcome. Some aspects could still be changed before the 8.100 first "official" release, sometime before the end of 2007.

Acknowledgements

The support and kind hospitality of the SFT group at CERN is gratefully acknowledged. Peter Skands has contributed the implementation of the SUSY processes and the SUSY Les Houches Accord. Mikhail Kirsanov has developed the configure files, the makefiles and the interface to HepMC, and made several valuable suggestions. Ben Lloyd has written the PHP framework. Bertrand Bellenot has provided a simple makefile for Win32/NMAKE. This work was supported in part by the European Union Marie Curie Research Training Network MCnet under contract MRTN-CT-2006-035606.

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

[4] T. Sjöstrand, S. Mrenna and P. Skands, JHEP **05** (2006) 026 [hep-ph/0603175]

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

[6] 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/`

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

[8] 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]

[9] J. Alwall et al., Computer Physics Comm. **176** (2007) 300 [hep-ph/0609017]

[10] M.R. Whalley, D. Bourilkov and R.C. Group, in 'HERA and the LHC', eds. A. De Roeck and H. Jung, CERN-2005-014, p. 575 [hep-ph/0508110]

[11] Particle Data Group, W.-M. Yao et al., J. Phys. **G33** (2006) 1

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

[13] see webpage `http://proj-clhep.web.cern.ch/proj-clhep/`