An Introduction to Pythia: The Event Generator

ROCKY BALA
Department of Physics and Astrophysics
University Of Delhi

July 31, 2012

Abstract

The PYTHIA program is frequently used for event generation in high energy physics. The emphasis is on multiparticle production in collision between elementry particles. Most of the components of the program represent original research. Some of the main physics features of PYTHIA has been described here. A brief note on the use of Monte Carlo techniques has also been included. And the application of Pythia is described by considering the process $q + g \rightarrow \gamma + jet$.

Contents

1 Introduction **2**
1.1 Why event generators are required: 2
1.2 Monte Carlo methods in event generators 3
1.2.1 Basics of Monte Carlo 3
1.2.2 Monte Carlo method in high energy physics 3

2 Pythia: The Event Generator **5**
2.1 Need of a new version in C++ 5
2.2 Main Physics aspects of Pythia 5
2.3 Installation 6
2.4 Program Flow 7
2.4.1 Initialisation 7
2.4.2 Generation: The event loop 9
2.4.3 Finishing 11
2.5 Program files and Documentation 11

3 Pythia program for the process q + g → γ + jet **11**

4 Histograms **14**

5 Conclusions **15**

1 Introduction

Multiparticle production is the most characteristic feature of current high-energy physics. Today, observed particle multiplicities are typically between ten and a hundred, and with future machines this range will be extended upward. The bulk of the multiplicity is found in jets, i.e. in collimated bunches of hadrons (or decay products of hadrons) produced by the hadronization of partons, i.e. quarks and gluons.

Nature of the Problem: *The complexity of High-Energy Processes*

High-energy collisions between elementary particles normally give rise to complex final states, with large multiplicities of hadrons, leptons, photons and neutrinos. The relation between these final states and the underlying physics description is not a simple one, for two main reasons. Firstly, we do not even in principle have a complete understanding of the physics. Secondly, any analytical approach is made intractable by the large multiplicities. To first approximation, all processes have a simple structure at the level of interactions between the fundamental objects of nature, i.e. quarks, leptons and gauge bosons. But there are correction to these. Firstly, there are bremsstrahlung-type modifications, i.e. the emission of additional final-state particles by branchings such as $e \rightarrow e\gamma$ or $q \rightarrow qg$. We therefore speak about 'parton showers', wherein a single initial parton may give rise to a whole bunch of partons in the final state. Secondly, we have true higher-order corrections. The necessary perturbative calculations are usually very difficult; only rarely have results been presented that include more than one non-trivial order i.e. more than one loop. Thirdly, quarks and gluons are confined. Due to which they result into the hadronization process, wherein the coloured partons are transformed into jets of colourless hadrons, photons and leptons. The hadronization can be further subdivided into fragmentation and decays, where the higher mass hadrons produced by hadronization process decay into the lower mass ones. This process is still not yet understood from first principles, but has to be based on models. So the simple structure that we started out with has now become considerably more complex - instead of maybe two final-state partons we have a hundred final particles.

Solution Method: *Event Generators*

It is here that event generators come to the rescue. In an event generator, the objective striven for is to use computers to generate events as detailed as could be observed by a perfect detector. This is not done in one step, but rather by 'factorizing' the full problem into a number of components, each of which can be handled reasonably accurately. In the actual generation procedure, most steps therefore involve the branching of one object into two or at least into a very small number, with the daughters free to branch in their turn. In generators, Monte Carlo techniques are used to select all relevant variables according to the desired probability distributions. Complete events are generated by Monte Carlo methods. The complexity is mastered by a subdivision of the full problem into a set of simpler separate tasks. All main aspects of the events are simulated, such as hard-process selection, initial- and final-state radiation, beam remnants, fragmentation, decays, and so on. Therefore events should be directly comparable with experimentally observable ones. The programs can be used to extract physics from comparisons with existing data, or to study physics at future experiments. In real life, the machine produces interactions. These events are observed by detectors. In the Monte Carlo world, the role of the machine, namely to produce events, is taken by the event generators.

1.1 Why event generators are required:

An event generator can be used in many different ways. The five main applications are probably the following:

- To give physicists a feeling for the kind of events one may expect/hope to find, and at what rates.

- As a help in the planning of a new detector, so that detector performance is optimized, within other constraints, for the study of interesting physics scenarios.
- As a tool for devising the analysis strategies that should be used on real data, so that signal-to-background conditions are optimized.
- As a method for estimating detector acceptance corrections that have to be applied to raw data, in order to extract the true physics signal.
- As a convenient framework within which the observed phenomena can be interpreted in terms of a more fundamental underlying theory (usually the Standard Model).

1.2 Monte Carlo methods in event generators

1.2.1 Basics of Monte Carlo

Monte Carlo (MC) methods are stochastic techniques–meaning they are based on the use of random numbers and probability statistics to investigate problems. You can find MC methods used in everything from economics to nuclear physics to regulating the flow of traffic. Of course the way they are applied varies widely from field to field. But, strictly speaking, to call something a "Monte Carlo" experiment, all you need to do is use random numbers to examine some problem.
"Hit and miss" integration is the simplest type of MC method to understand. The idea is that we find some region in space of known volume, which encloses the volume we want to evaluate, then generate random points everywhere in this region, and count the points which actually do hit the volume we want to compute. Suppose the volume of the external region is V_e and the fraction of hits is f_h. Then the volume of the region to be evaluated is simply V $= V_e f_h$.
One of the main applications of MC is integrating functions. At the simplest, this takes the form of integrating an ordinary 1- or multidimensional analytical function. But very often nowadays the function itself is a set of values returned by a simulation and the actual function form need not be known at all. Most of the same principles of MC integration hold regardless of whether we are integrating an analytical function or a simulation.

1.2.2 Monte Carlo method in high energy physics

In HEP, we use two type of monte carlo methods:

- MC for event generation and calculations of process cross-sections.
- MC simulation of generators.

Monte Carlo methods are used to generate events in event generators using random number and probabilities. To understand this, let us assume that the particle is an electron in a gas. So it can

- With probability p1 : ionize the gas, loose some momentum, produce N secondary electrons with momenta P1,P2,.....
- Do nothing with probability 1-p1
- Generate random number r.

- if r<p1: Generate momenta of secondary electrons, add them to your list of particles, reduce the momentum of initial electron.
- if r>p1 then do nothing.

Similarly if the particle is photon then it can

- Convert and produce electron-positron pair with probability p1.
- Compton scatter with probability p2.
- Ionize the matter with probability p3.
- Generate random number r.
- If r<p1, convert the photon to the electron-positron pair.
- If p1<r<p1+p2 generate Compton electron, reduce photon momentum.
- If p1+p2<r<p1+p2+p3 ionize the matter.

Similarly if we are considering an hadron then we have to consider the parton showers and add them to the list of particles. This is continued until all the particles are considered.

For the calculation of process cross section MC are used in the following way. A particle production/decay can be described by matrix element of a particular process. Module of matrix element squared is a measure of probability density for that particular process and the differential cross-section is proportional to this matrix element squared and is given by

$$d\sigma \propto |\mathcal{M}(p_1, p_2......p_n)|^2 \left(\prod_n \frac{d^3 p_n}{(2\pi)^3} \frac{1}{2E_n} \right) \times (2\pi)^4 \delta^4 (P_1 + P_2 - \sum p_n)$$

which represents the cross-section for the collision of two particles of 4-momenta P_1 and P_2 that results into n final states of 4-momenta p_1, $p_2.....p_n$ and the portion of dσ, that is

$$d \prod_n = \left(\prod_n \frac{d^3 p_n}{(2\pi)^3} \frac{1}{2E_n} \right) \times (2\pi)^4 \delta^4 (P_1 + P_2 - \sum p_n)$$

is called the phase space part.
Now what is the probability that the final states 4-momenta will be in some part of available phase space? The answer to this question can be obtained from the basics of probability theory which says that if a random variable x has a distribution g(x) then the probability of finding x beteween a and b is given by

$$P(a < x < b) = \int_a^b g(x) dx$$

Similarly to know the probability of finding the final states 4-momenta to be in some part of available phase space we just have to integrate their probability density function which is proportional to the differential cross section dσ over the available phase space. If n is very high then this becomes an intergral over several dimensions. Forget about evaluating it by standard methods. So this kind of integrals are evaluated by monte carlo methods by randomly generating 4-momenta of final state particles by using some random number generators.

In detector simulation, Particle production generators provide input to detector simulation program. Detector simulation tracks the particles through detector material simulating their interaction with material Then it simulates the detector response and produces output.

2 Pythia: The Event Generator

The Pythia program can be used to generate high-energy-physics events, i.e. sets of outgoing particles produced in the interactions between two incoming particles. The objective is to provide as accurate as possible a representation of event properties in a wide range of reactions, within and beyond the Standard Model, with emphasis on those where strong interactions play a role, directly or indirectly, and therefore multihadronic final states are produced.

The Pythia program is a standard tool for the generation of high-energy physics collisions, comprising a coherent set of physics models for the evolution from a few-body hard process to a complex multihadronic final state. It contains a library of hard processes and models for initial- and final-state parton showers, multiple parton-parton interactions, beam remnants, string fragmentation and particle decays. It also has a set of utilities and interfaces to external programs. While previous versions were written in Fortran, Pythia 8 represents a complete rewrite in C++.

2.1 Need of a new version in C++

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++, and in the process clean up and modernise various aspects. It does contain some new physics aspects, on the other hand, that should make it an attractive option especially for LHC physics studies.

2.2 Main Physics aspects of Pythia

For the description of a typical high-energy event, an event generator should contain a simulation of several physics aspects. If we try to follow the evolution of an event in some semblance of a time order, one may arrange these aspects as follows:

1. Initially two beam particles are coming in towards each other. Normally each particle is characterized by a set of parton distributions, which defines the partonic substructure in terms of flavour composition and energy sharing.

2. One shower initiator parton from each beam starts off a sequence of branchings, such as $q \to qg$, which build up an initial-state shower.

3. One incoming parton from each of the two showers enters the hard process, where then a number of outgoing partons are produced, usually two. It is the nature of this process that determines the main characteristics of the event.

4. The hard process may produce a set of short-lived resonances, like the $Z^0/W^\pm$ gauge bosons, whose decay to normal partons has to be considered in close association with the hard process itself.

5. The outgoing partons may branch, just like the incoming did, to build up final-state showers.

6. In addition to the hard process considered above, further semihard interactions may occur between the other partons of two incoming hadrons.

7. When a shower initiator is taken out of a beam particle, a beam remnant is left behind. This remnant may have an internal structure, and a net colour charge that relates it to the rest of the final state.

8. The QCD confinement mechanism ensures that the outgoing quarks and gluons are not observable, but instead fragment to colour neutral hadrons.

9. Many of the produced hadrons are unstable and decay further.

The Event Generator-Pythia is designed to consider all these aspects. Here for convenince we can call electron and photon as the partonic substructure of an electron with the branching e → eγ similar to parton shower branching q → qg. So that above points applies equally well to an interaction between two leptons, between a lepton and a hadron, and between two hadrons.

2.3 Installation

Denoting a generic Pythia 8 version Pythia81xx (here xx = 65 is used), here is how to install Pythia 8 on a Linux/Unix/MacOSX system as a standalone package:

1. In a browser, go to

$$http://www.thep.lu.se/\sim torbjorn/Pythia.html$$

2. Download the (current) program package

$$pythia81xx.tgz$$

to a directory of your choice (e.g. by right-clicking on the link).

3. In a terminal window, cd to where pythia81xx.tgz was downloaded, and type

$$tar\ \ xvfz\ \ pythia81xx.tgz$$

This will create a new (sub)directory pythia81xx where all the Pythia source files are now ready and unpacked.

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

5. For test runs, cd to the examples/ subdirectory. An ls reveals a list of programs, mainNN, with NN from 01 through 28 (and beyond). These example programs each illustrate an aspect of Pythia 8.

6. To execute one of the test programs, do

$$make\ \ mainNN$$
$$./mainNN.exe$$

The output is now just written to the terminal, stdout. To save the output to a file instead, do ./mainNN.exe > mainNN.out, after which you can study the test output by opening mainNN.out.

7. If you use a web browser to open the file

$$pythia81xx/htmldoc/Welcome.html$$

   you will gain access to the online manual, where all available methods and parameters are described. Use the left-column index to navigate among the topics, which are then displayed in the larger right-hand field.

2.4 Program Flow

The event generation process can be subdivided into three stages:

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

The nature of the run is defined at the initialization stage so this is where most of the PYTHIA user code has to be written.

2.4.1 Initialisation

1. The Pythia class is the main means of communication between the user and the event-generation process. Firstly, at the top of the main program, the proper header file must be included:

   #include "Pythia.h"

2. To simplify typing, it also makes sense to declare

   using namespace Pythia8;

3. Begin main program:

   int main() {

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

   Pythia pythia;

   Here we have assumed that the pythia object has been created with this name, but of course you are free to pick another one. When this object is declared, the Pythia constructor initialises all the default values for the Settings and the ParticleDataTable data bases. These data are now present in memory and can be modified in a number of ways. Normally a run will only contain one Pythia object.

5. Most conveniently, Pythia's settings and parameters can be changed by the two methods. The first one is

   pythia.readString(string);

   for changing a single variable. 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 some special 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 but upper and lower case letters are used for clarity. Strings that do begin with a letter or digit and still are not recognised 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. The readString(...) method is convenient for changing one or two settings, but becomes cumbersome for more extensive modifications.Some examples would be

   pythia.readString("TimeShower:pTmin = 1.0");
   pythia.readString("111:mayDecay = false");

   The second method is

   pythia.readFile("fileName");

   in which all the changes are collected in a file which have .cmnd extension and which for historical reasons often called a "card file", where each line is a character string defined in the same manner as above (without quotation marks). The whole file can then be read and processed by using the above command. Each line in this file will be processed by the readString(string) method introduced above. You can thus freely mix comment lines and lines handed on to Settings or to ParticleData. This approach is better suited for more extensive changes than a direct usage of readString(string), and can also avoid having to recompile and relink your main program between runs. Changes are made sequentially in the order the commands are encountered during execution, meaning that if a parameter is changed several times it is the last one that counts.

6. Next comes the initialization stage, where all the remaining details of the generation are specified. There is one standard method to use for this, that is

   pythia.init();

   with no arguments. It will read all relevant information from the Settings and ParticleData databases. Specifically the setup of incoming beams and energies is governed by the the beam parameters from the Beams group of variables. If you don't change any of those you will default to proton-proton collisions at 14 TeV, i.e. the nominal LHC values.
   A few alternative forms are available, where the arguments of the init(...) call can be used to set the beam parameters. Some of these are

   - pythia.init(idA, idB, eCM); which lets you specify the identities and the CM energy of the two incoming beam particles, with A (B) assumed moving in the +z (-z) direction.
   - pythia.init(idA, idB, eA, eB); is similar, but the two beam energies can be different, so the collisions do not occur in the CM frame. If one of the beam energies is below the particle mass you obtain a fixed-target topology.

- pythia.init(idA, idB, pxA, pyA, pzA, pxB, pyB, pzB); is similar, but here you provide the three-momenta (p_x, p_y, p_z) of the two incoming particles, to allow for arbitrary beam directions.

7. If you want to have a list of the generator and particle data used, either only what has been changed or everything, you can use

$$pythia.settings.listChanged();$$
$$pythia.settings.listAll();$$
$$pythia.particleData.listChanged();$$
$$pythia.particleData.listAll();$$

2.4.2 Generation: The event loop

1. To generate the next event, we just have to give a command

   pythia.next();

   This method takes no arguments, everything has already been specified. This command pythia.next() will generate only next event. If we want to generate more than one event then we need to put this command inside the event loop, so the code now reads

   ```
   for ( int iEvent = 0; iEvent < 5; ++iEvent)
   {
   pythia.next();
   }
   ```

   This is called event loop.The program will now generate 5 events; each call to pythia.next() resets the event record and fills it with a new event.
   To list more of the events, you also need to add

   pythia.readString(”Next:numberShowEvent = 5”);

   along with the other pythia.readString commands.

2. The generated event is now stored in the event object, of type Event, which is a public member of pythia. The event record is set up to store every step in the evolution from an initial low-multiplicity partonic process to a final high-multiplicity hadronic state, in the order in which new particles are generated. Thus pythia.event[i] is the i’th particle of the current event, and you may study its properties by using various pythia.event[i].method() possibilities. For example pythia.event[i].id() returns the PDG identity code of the i’th particle. An event can be listed with pythia.event.list().

   The event record also provides a particle’s status code which shows whether a particle is decayed or is present in the final state. A positive status code shows that the particle is still present and negative code shows that the particle is decayed. When a new particle is added to the event record, it is assigned a positive status code and whenever a particle is allowed to branch or decay further its status code is negated (but it is never removed from the event record), such that only particles in the final state remain with positive codes. The pythia.event[i].isFinal() returns true/false for positive/negative status codes.

   The event record also provides information about the history of the particle. The two mother and two daughter indices of each particle provide information on the history relationship between the different entries in the event record. As an example, in a 2 $\rightarrow$ 2 process ab $\rightarrow$ cd, the locations of a and b would set the mothers of c and d, with the reverse relationship for daughters. The commands pythia.event[i].motherList() and pythia.event[i].daughterList() provides a list of all mother or daughter indices of particle i. Similarly many more methods are available that provides information about various properties of the particles.

3. To access all the particles in the event record, insert the following loop after pythia.next() (but fully enclosed by the event loop)

   ```
   for (int i = 0; i < pythia.event.size(); ++i) {
   cout << ”i = ” << i
   << ”, id = ” << pythia.event[i].id() << endl;
   }
   ```

   which we call the particle loop. Inside this loop, you can access the propertie of each particle pythia.event[i] using various methods. As the method id() returns the PDG identity code of ith particle.

4. For many processes it makes sense to apply phase space cuts. The ones currently available in particular include

   - PhaseSpace:mHatMin, PhaseSpace:mHatMax : to set the range of invariant masses of the scattering process.
   - PhaseSpace:pTHatMin, PhaseSpace:pTHatMax : to set the range of transverse momenta in the rest frame of the process.

   We can also apply some other cuts. For example

   - 6:m0 = 175 will change the top mass, which by default is 171 GeV.
   - PartonLevel:FSR = off to switch off final-state radiation.
   - PartonLevel:ISR = off to switch off initial-state radiation.
   - PartonLevel:MPI = off to switch off multiparton interactions.

   and so on.

5. Now suppose we want to generate more events, say 1000, and view the shape of these distributions. Inside Pythia is a very simple histogramming class, that can be used for rapid check/debug purposes. To book the histograms, insert before the event loop

   Hist name(”title”,number of bins,x_{min},x_{max});

   Now we want to fill the histograms in each event, so before the end of the event loop insert

   ```
   name.fill();
   ```

   Finally, to write out the histograms, after the event loop we need a line like

   ```
   cout << name;
   ```

   In Pythia, the output of histogram comes in line printer mode which has a different format and is somewhat difficult to understand. So to fill Histograms we can use root as well. The benefit using root is that the visulization of the histogram is good and is easy to understand. To use root in pythia we just need to call the root library in pythia program which is not very difficult to do so. In pythia package, there is a subdirectory named rootexamples which contains some root examples and a README file which tells how to use root in pythia.

2.4.3 Finishing

1. At the end of the generation process, you can call

   pythia.stat();

   to get some run statistics, on cross sections and the number of errors and warnings encountered.

2. In the end return 0; command is written to end the main program with error free return.

2.5 Program files and Documentation

The code is subdivided into a set of files, mainly by physics task. 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.

The .xml documentation 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. This can easily be read if you open the htmldoc/Welcome.html file in your favourite web browser.

The installation procedure is described in a README file. Compiled libraries are put in the lib subdirectory. Finally, some examples of main programs, along with input files, or ”cards”, for them, are found in the examples subdirectory. This directory contains its own README file, configure script and Makefile which will allow you to build executables. The executables are placed in the bin directory, but with links from examples.

3 Pythia program for the process q + g → γ + jet

The code for the program is

```
//program for q qbar going to gamma + jet and showing
//transverse momentum, longitudinal momentum, pseudorapidity
//and energy distribution for 1000 events.

// Stdlib header file for input and output.
#include <iostream>

// Header file to access Pythia 8 program elements.
#include "Pythia.h"

// ROOT, for histogramming.
#include "TH1.h"

// ROOT, for interactive graphics.
#include "TVirtualPad.h"
```

```
#include "TApplication.h"

// ROOT, for saving file.
#include "TFile.h"

using namespace Pythia8;

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

// Create the ROOT application environment.
TApplication theApp("hist", &argc, argv);

// create Pythia object and set up generation
Pythia pythia;
pythia.readString("PromptPhoton:qg2qgamma = on");
pythia.readString("Beams:eCM = 7000.");
pythia.readString("PhaseSpace:pTHatMin = 20.");
pythia.init();

// Create file on which histograms can be saved.
TFile* outFile = new TFile("hist.root", "RECREATE");

// Book histograms.
TH1F *pTgamma = new TH1F("pTgamma","Transverse momentum of gamma", 200,0.,200.);
TH1F *pTjet = new TH1F("pTjet","Transverse momentum of jet", 200,0.,200.);
TH1F *pzgamma = new TH1F("pzgamma","Longitudinal momentum of gamma", 200,0.,200.);
TH1F *pzjet = new TH1F("pzjet","Longitudinal momentum of jet", 200,0.,200.);
TH1F *etagamma = new TH1F("etagamma","Pseudorapidity of gamma", 200, -5., 5.);
TH1F *etajet = new TH1F("etajet","Pseudorapidity fo jet",200,-5.,5.);
TH1F *energygamma = new TH1F("energygamma","energy distribution of gamma",500, 0.,1000.);
TH1F *energyjet = new TH1F("energyjet","energy distribution of jet",500,0.,1000.);

// Begin event loop. Generate event; skip if generation aborted.
for (int iEvent = 0; iEvent<1000; ++iEvent)
    {
                  double k = 0;
  int l = 0;
  double m = 0;
  int n = 0;
      if(!pythia.next()) continue;
      for (int i = 0; i< pythia.event.size(); ++i)
{
  if(pythia.event[i].id() == 22)
    {
  if(k < pythia.event[i].pT())
    {
      k = pythia.event[i].pT();
      l = i;
    }
}
      else if(pythia.event[i].id() == 1 || pythia.event[i].id() == 2
           || pythia.event[i].id() == 3 || pythia.event[i].id() == 4
           || pythia.event[i].id() == 5 || pythia.event[i].id() == 6)
           {
                if(m < pythia.event[i].pT())
             {
           m = pythia.event[i].pT();
           n = i;
            }
        }
}

pTgamma->Fill(k);
pzgamma->Fill(pythia.event[l].pz());
etagamma->Fill(pythia.event[l].eta());
energygamma->Fill(pythia.event[l].e());
pTjet->Fill(m);
pzjet->Fill(pythia.event[n].pz());
etajet->Fill(pythia.event[n].eta());
energyjet->Fill(pythia.event[n].e());

       }

pythia.stat();

pTgamma->Draw();
pzgamma->Draw();
etagamma->Draw();
energygamma->Draw();
pTjet->Draw();
pzjet->Draw();
etajet->Draw();
energyjet->Draw();

std::cout << "\nDouble click on the histogram window to quit.\n";
gPad->WaitPrimitive();

// Save histogram on file and close file.
pTgamma->Write();
pzgamma->Write();
etagamma->Write();
energygamma->Write();
pTjet->Write();
pzjet->Write();
etajet->Write();
energyjet->Write();
delete outFile;
```

```
// Done
  return 0;
}
```

This program will take two incoming proton beams at a center of mass energy of 7TeV and will generate the hard process

$$q + g \rightarrow \gamma + jet$$

as well as the complete event.

The output file contains a listing of no., id, name, status, mothers, daughters, colours, p_x, p_y, p_z, e and m for all the particles for both the hard process and the complete event. A pTHatMin cut of 20 GeV is applied which throws out all the final particle with pT less than 20 GeV. Since photons are stable and all the jets decays out in π^+ and π^- so this cut is effective for photons only. And the condition for maximum pT value for each event is applied so that only gamma and jet with maximum pT from the various gammas and jets produced in the event has been selected and plotted.

4 Histograms

The histograms that result from executing above program are as shown:

1. For the energy distribution of gamma and jet: shown in Figure 1 and 2.

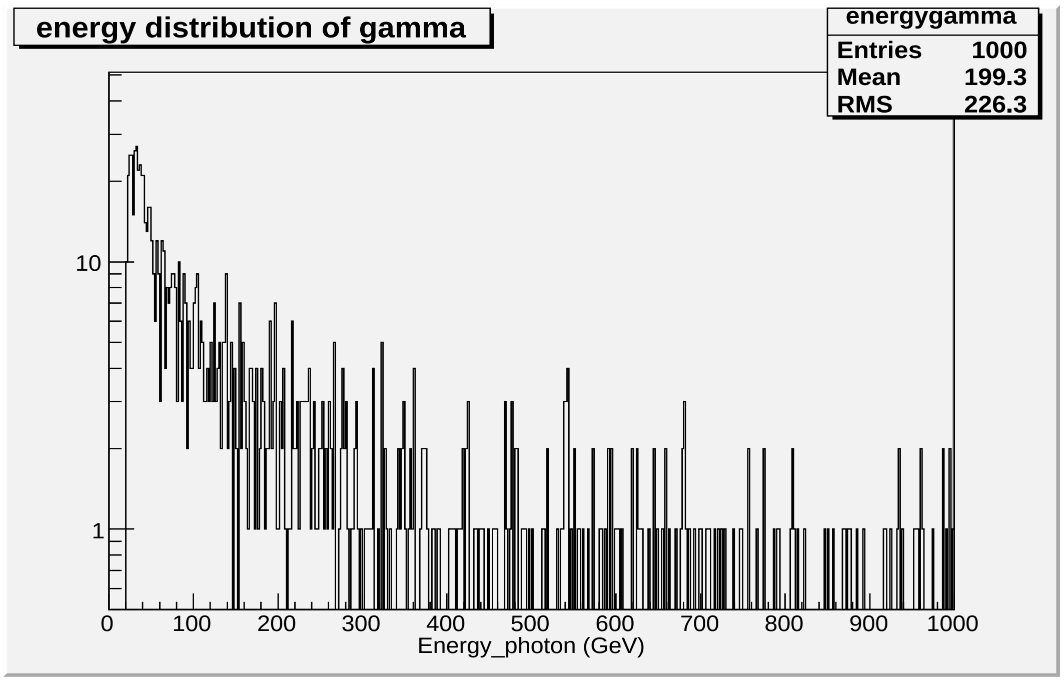

Figure 1: The energy distribution for γ

2. Pseudorapidity of gamma and jet: shown in Figure 3 and 4.

3. Transverse momentum of gamma and jet: shown in Figure 5 and 6.

4. Longitudinal momentum of gamma and jet: shown in Figure 7 and8.

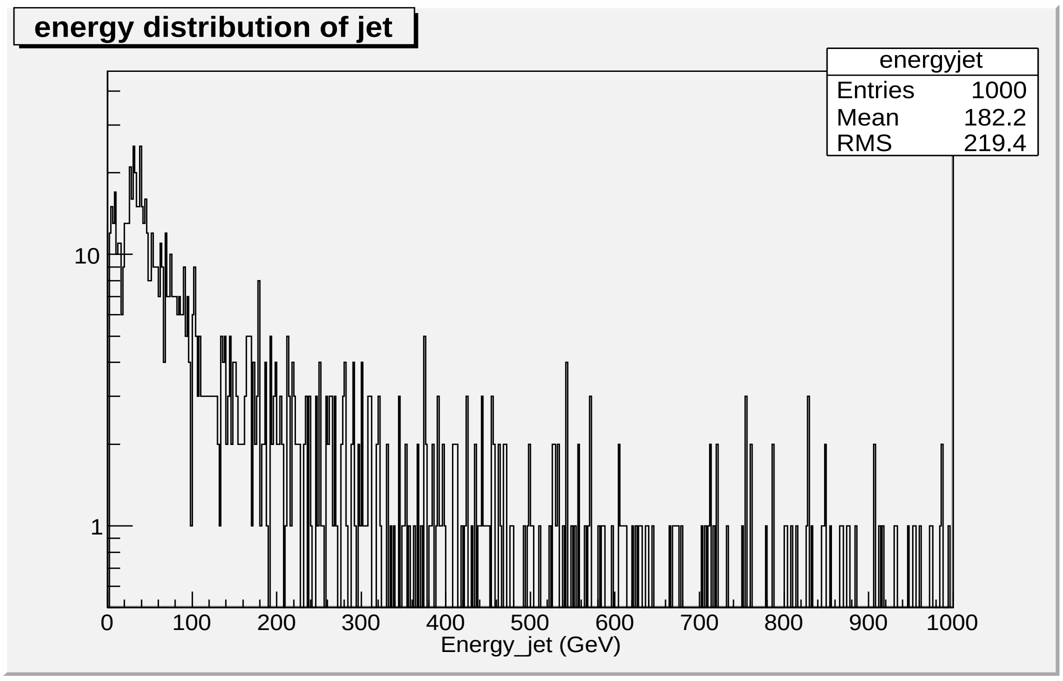

Figure 2: The energy distribution for jet

5 Conclusions

- The graphs for energy distribution for gamma and jet shows that most of the particles have small energy and particles with large energy are comparetively small.
- The pseudorapidity is defined as

$$\eta = -ln\left[\tan\frac{\theta}{2}\right]$$

  where θ is the angle between particle's momentum and the beam axis. So there is an inverse relation between η and θ. A large value of η signifies a small value for θ. So the graphs for pseudorapidity distribution shows that the particles are scattered in almost all the directions.
- The transverse and longitudinal dirstributions shows that both gamma as well as jet have small values of the transverse momentum but comparatively large value for the longitudinal momentum which means that the particles suffering hard scattering in the collision are very less and most of the particles suffer single collisions. Since only very small fraction of the particles has significant transverse momentum which implies that the fraction of hard scattering is small.

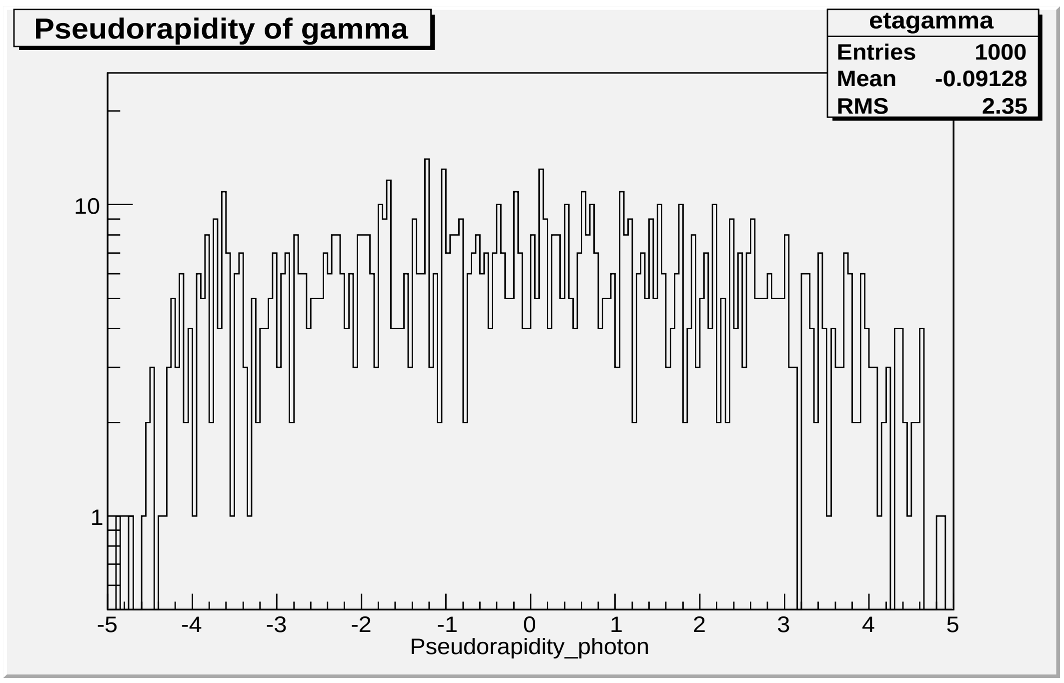

Figure 3: Pseudorapidity distribution for γ

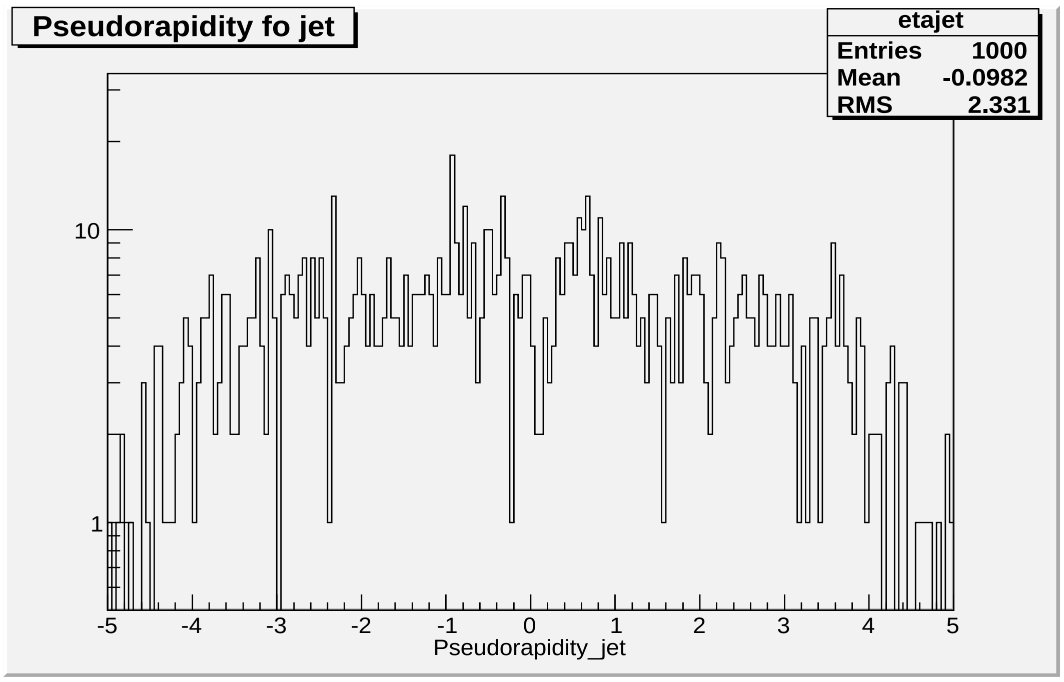

Figure 4: Pseudorapidity distribution for jet

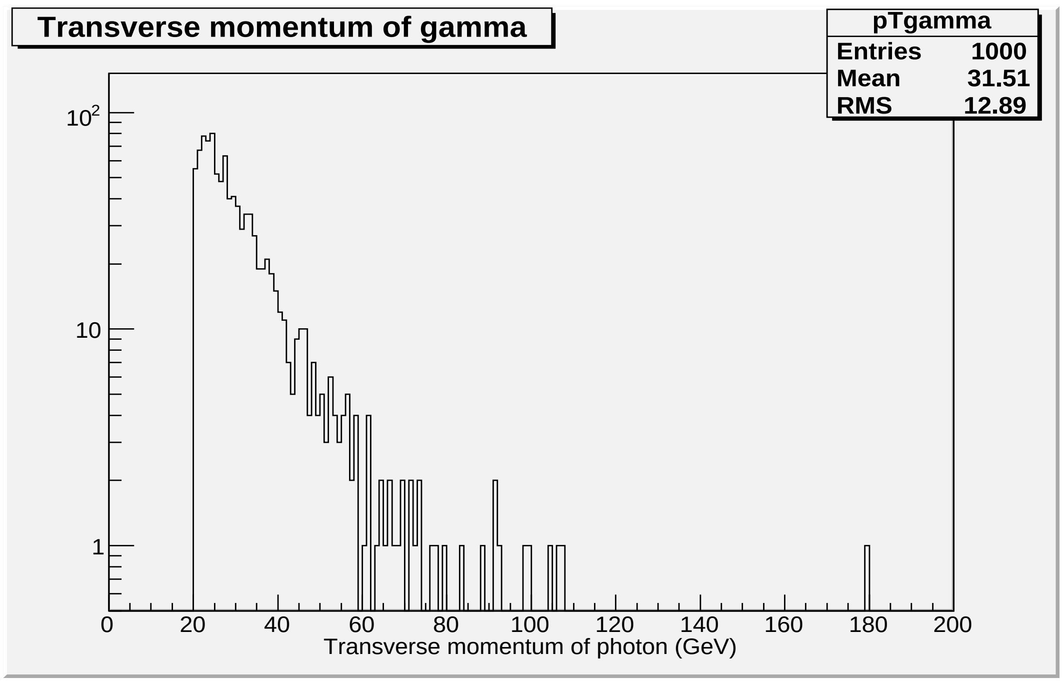

Figure 5: Transverse momentum distribution for γ

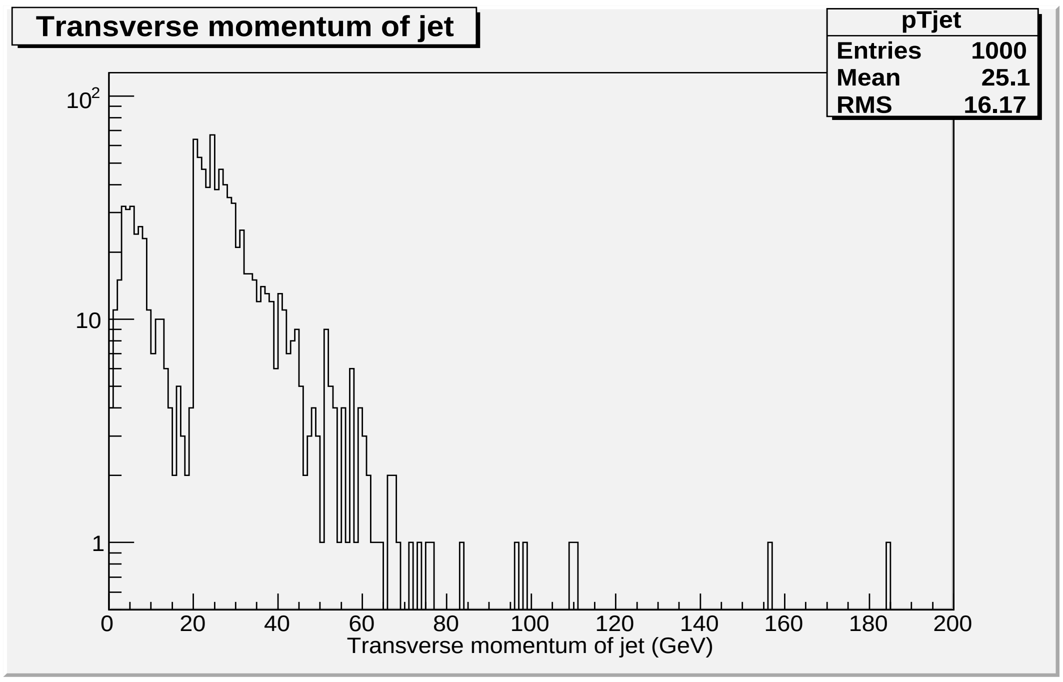

Figure 6: Transverse momentum distribution for jet

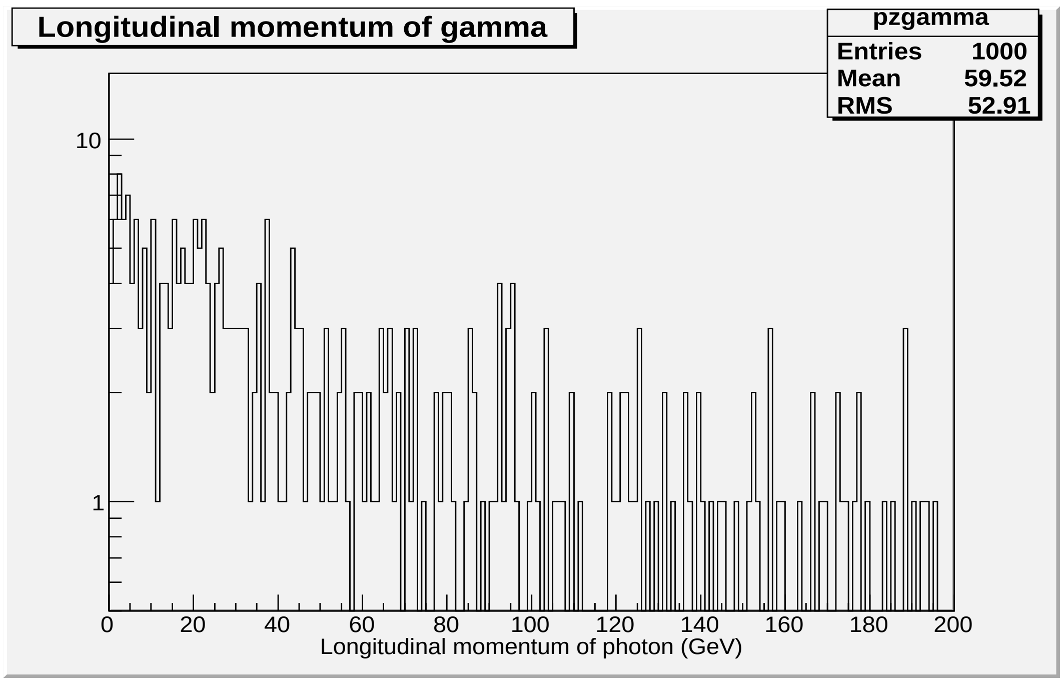

Figure 7: Longitudinal momentum distribution for γ

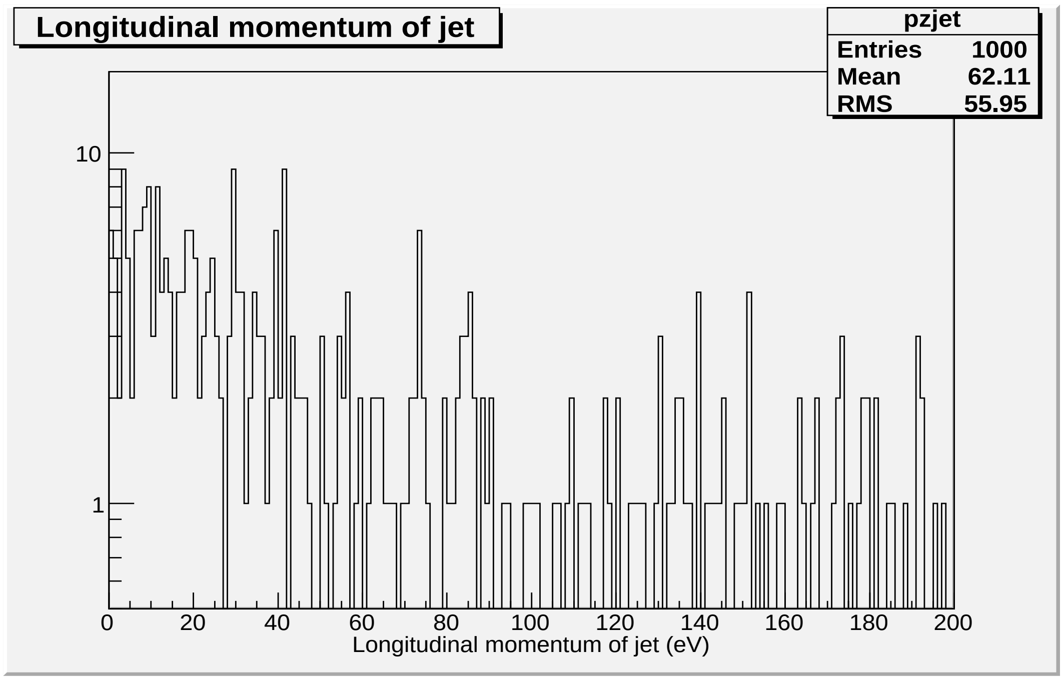

Figure 8: Longitudinal momentum distribution for jet