

Pythia8 to HepMC to Rivet

Ewen Gillies
University of Edinburgh

1 Pythia8

1.1 An event record of $gg \rightarrow t\bar{t}$ based events

In the supplied documents, the Pythia8 worksheet [5] provides a quick guide to producing a “Hello World” program. This program creates one event from a proton-proton collision at LHC energies (7 TeV). The example adds levels of complexity as the worksheet continues. Below summarizes the first few steps, then goes on to write a correlation example, not explicitly covered in examples.

1.1.1 Writing, Compiling, and Running

Basic Event Record for $gg \rightarrow t\bar{t}$ Events

```
#include "Pythia.h"
2 using namespace Pythia8;

4 // Set up generation
int main() {
6   Pythia pythia; //Declare object
   pythia.readString("Top:gg2ttbar□=□on"); //Switch on process
8   pythia.init( 2212 , 2212 , 7000.); //Initialise pp beam at 7 TeV

10  // Show settings
   pythia.settings.listChanged(); // Show changed settings
12  pythia.particleData.listChanged(); // Show changed particle data

14  // Generate event(s).
   pythia.next(); // Generate an(other) event. Fill event record.
16  pythia.event.list(); // Print contents of event record.
   return 0;
18} // End main program with error-free return.
```

Write

- Start by opening a new file in VIM or any other such editor called `mymain.cc` in the same directory as the supplied Pythia8 examples, i.e. in `/pythia8***/examples`. This is important for compilation purposes. Begin the program by including the standard Pythia header, and use the namespace `Pythia8`.
- Inside the main, initialize a Pythia object named `pythia`. All functionality is derived from this object. Use the function `pythia.readString` to turn on the $gg \rightarrow t\bar{t}$ channel.¹ Initialise the beam using `pythia.init`. The first two arguments of this function are particle identification codes², and the last is the beam energy in GeV. Note this is formatted as a double.
- Next, require that all settings that have been changed from the default are printed, as well as all particle data that is changed, as below.

¹A full list of input arguments for this function can be found here [6], under ‘Setup Run Tasks’.

²These can be found in section 4.2

- Finally, initialize the “next” event using `pythia.next()`. This command initializes all events, making it essential to `pythia`. Usually this command is in a loop, hence the use of the word “next.” The last command prints the event record, which is a step by step evolution from low multiplicity partonic states to high multiplicity hadronic states.

Compile

```
# Create an executable file for one of the normal text programs
main00 main01 main02 ... main08 main09 main10 \
...
main44 main71 main72 : \
```

- Noting that we are still working in the `/pythia8***/examples` directory, open the Makefile located there in a text editor. Scroll down to around line 30 to see the above code.
- Append this code, after `main72`, with `mymain`, or whatever name was chosen for your program. Exit the editor, and enter `make mymain` to create the desired `mymain.exe` file.

Running

- Assuming no compilation errors, running the `mymain.exe` will print the output to the terminal. To store it in a file instead, direct the output to an output file with the following:
`./mymain.exe > mymain.dat.`
- The contents of the output file are self-explanatory. The bulk of this sample output is the event record, or event listing. For a more complete discussion of this output, see Appendix A in [5].

1.1.2 Utilizing .cmd files

- In the previous example, all process settings are set in the beginning of the `.cc` file. This means that any changes to the settings would require recompilation, which is cumbersome for larger simulations. To aide this, Pythia8 utilizes `.cmd` “card” files, which effectively replaces lines 7-8 in our short example. To use this functionality, change lines 3-9 as follows:

Using a Card File

```
int main (int argc, char* argv[]) {
4 //Set up generation
  Pythia pythia; //Declare pythia object
6
  //Read in command file
8 pythia.readFile(argv[1]);
  pythia.init(); //initialise as per .cmd file
```

- The main function now calls an argument, which is fed into the `pythia.readFile` function. The initialization conditions, as well as all processes and settings, will now be defined in a “card” file. Note that `pythia.init()` is still called, and that any arguments in here will take precedent over the settings in the card file.
- To write the card file, create a new file named `mymain.cmd`. Enter the following into the new file:

Card File

```
# t tbar production at the LHC
Beams:idA = 2212      #incoming protons
Beams:idB = 2212      #incoming protons
Beams:eCM= 7000      #collision energy in CoM frame
Top:qqbar2ttbar = on  #q qbar -> t tbar channel is on
Top:gg2ttbar = on    #g g-> t tbar channel is on
```

- The format for these files is simple. Generally, there is one command per line with the following syntax:

`variable = value`

- In the above example, the first three lines replace the `pythia.init()` arguments, the fourth line turns on the quark fusion channel, as before, while the last line also adds the gluon fusion channel (optional). Note that any non-alphanumeric characters (such as `!`, `$`, `#`) are used to start a comment.
- Recompile `mymain.cc`. It is now set up to read in `mymain.cmnd` file. To bring it all together, enter the following in the terminal:

```
./mymain.exe mymain.cmnd > mymain.dat
```

- The output file should be as before, with the new gluon fusion channel added. The full power of card files is not covered here. Variables like the top mass, to the number of events, to random number seeds can be set in these files. Refer to the online Pythia 8 Manual for a full list of arguments [6].

This concludes the summary of the example in [5]. In the supplied document, the program goes on to demonstrate how to pick out the “final” top particle in an event record, as well as demonstrate the built in histogramming function of Pythia8.

1.2 A Correlation Example

The previous example is covered in tutorials online. The following is aimed to provide an example of writing a quick correlation study, as well as demonstrate some of the powers of coding Pythia8 in C++. The aim of this will be to determine the correlation coefficients for consecutive symmetric pseudo-rapidity bins.

What we are aiming for in this example is to determine correlation coefficients for the multiplicity of charged final state particles in forward and backwards rapidity bins. For symmetric distributions, the coefficient can be calculated using the following formula:

$$\rho_{FB}(\Delta y) = \frac{\langle n_F n_B \rangle - \langle n_F \rangle^2}{\langle n_F^2 \rangle - \langle n_F \rangle^2}$$

Initialization

- Start by opening a new C++ file called `correlationExample.cc`. The beginning of this code is just as before. The object here is to run many events for different rapidity windows. We will define some variables for use in later calculations. Since we want to run this over several rapidity windows, open a loop over rapidity windows:

Correlation Initialization

```
#include "Pythia.h"
2 using namespace Pythia8;

4 int main (int argc, char* argv[]) {
    //Set up generation
6   Pythia pythia; //Declare pythia object

8   //Read in command file
    pythia.readFile(argv[1]);
10  pythia.init(); //initialise as per .cmdn file

12  // Show settings
    pythia.settings.listChanged(); // Show changed settings
14  pythia.particleData.listChanged(); // Show changed particle data

16  //Define variables
    int nEvents = pythia.mode("Main:numberOfEvents");
18  //Number of events as defined in .cmdn card file
    int nF = 0; //Number in the forward bin
20  int nB = 0; //Number in the backward bin

22  double nFAvg = 0.; //Averages
    double nBAvg = 0.;
24  double nFAvg2 = 0.; //Average of squared value
    double nBAvg2 = 0.;
26  double nBnFAvg = 0.; //Average of product

28  // Loop over rapidity windows
    for (int delY=0; delY < 5; ++delY){
```

- We are now going to look at the result of many events, therefore we define the number of events in `nEvents` on line 17. To ensure this value is flexible, pythia allows this value to be defined in the `.cmdn` file and accessed here. This will be revisited later on in this section.
- Now two very important loops in pythia coding will be introduced. The first is the called the **event loop** and the second is called the **particle loop**.

Event and Particle Loops

```
30  //Generate event(s)
    //Event Loop
32  for (int iEvent = 0; iEvent < nEvents; iEvent++){
        pythia.next(); //Generate an(other) event. Fill record.
34
        //Reset Variables
36        nF = 0;
        nB = 0;
38
        //Particle Loop
40        for (int i = 0; i < pythia.event.size(); ++i){
            //Check if it is charged and final
42            if (pythia.event[i].isCharged() && pythia.event[i].isFinal()){
                //Check if it is in the forward of backwards bin
44                if(pythia.event[i].y()>=delY && pythia.event[i].y() <= delY+0.5) ++nF;
                if(pythia.event[i].y()<=-delY && pythia.event[i].y()>=-delY-0.5) ++nB;
46            }
        }
48        //Running sums for averages
        nFAvg += nF;
50        nBAvg += nB;
        nFAvg2 += nF*nF;
52        nBAvg2 += nB*nB;
        nBnFAvg += nB*nF;
54    }
```

Event Loop

- Any commands in the event loop will be done for each event. The only necessary command is the `pythia.next()` command, which generates the next event. This particular event loop will count the number of charged final state particles in each event. After the particle loop, it keeps running sums for all expectation values.

Particle Loop

- The particle loop can access the event constituents, i.e. initial, intermediate, and final state particles. This loop is determined by the `pythia.event.size()`, which returns the number of particles in the event record.
- Each particle in the record is accessed with `pythia.event[i]`, which returns the i^{th} particle as an object. To further access properties, such as particle identity code, functions like `pythia.event[i].id()` can be called.
- In this case, we will use `pythia.event[i].y()`, which returns the pseudorapidity of the i^{th} particle. To ensure this is a charged final state particle, `pythia.event[i].isCharged()` and `pythia.event[i].isFinal()` are both called, which return booleans.

Final Calculations and Printing

- All that is left now is to calculate the averages, and hence to correlation coefficient. We will then ask the program to output these coefficients, close the pseudorapidity loop, and exit quietly.

Calculations and Output

```
56 // Determine true averages
   nFAvg2 = nFAvg2/nEvents;
58 nBAvg2 = nBAvg2/nEvents;
   nBnFAvg = nBnFAvg/nEvents;
60 nFAvg = nFAvg/nEvents;
   nBAvg = nBAvg/nEvents;
62
   //Determine forward backward coefficient
64 double rhoFB = ((nBnFAvg - nFAvg*nFAvg) / (nFAvg2 - nFAvg*nFAvg));
   cout << "The forward backwards correlation is" << rhoFB
66    << " for the forward rapidity window" << delY << " to" << delY+0.5 << endl;
68     }
   return 0;
70
}
```

Card File

- The card file for this example is much like the last one, with the addition of all hard and soft QCD processes. More or less processes can be selected using the options listed here in the Process Selection section of the pythia 8 online manual [6].

Card File

```
# Correlation Example for pythia8
Beams:idA = 2212 #incoming protons
Beams:idB = 2212 #incoming protons
Beams:eCM= 7000 #collision energy in CoM frame
HardQCD:all = on #turn on all hard processes
SoftQCD:all = on #turn on all soft processes
Main:numberOfEvents = 1000 #number of events to generate
```

All that is left now is to compile the program, and run it as before. As a sanity check, ensure that the correlation coefficients printed are between -1 and 1.

2 HepMC Format

HepMC files are self described as “an object oriented event record written in C++ for High Energy Physics Monte Carlo Generators” [4]. They provide a convenient and universal format for storing all information from MC event generators. Since this is the only file type that Rivet accepts, the following section covers how to create HepMC file types from various MC generators.

2.1 Pythia8 and Pythia6

Pythia8 and Pythia6 are able to output HepMC files directly. To use RIVET with other generators, see Section 3.2, which covers Rivet’s built in steering program: AGILE. Pythia8 provides example code that outputs a HepMC file in the `\examples` subdirectory under `main41.cc` and `main42.cc`.

The following sample program outputs a simple histogram using the built in histogramming tool in Pythia8. This code is mostly from the previous example. Instead of calculating a correlation coefficient, it demonstrates how to add all events to a HepMC record. In order to generate this output file, there are three key pieces of code. Note that some reside in the HepMC namespace (and are hence prefixed with `HepMC::`). No changes are needed in the card file.

Linking the HepMC library

- In order to properly output to the a `.hepmc` file type, the HepMC libraries must be properly linked to pythia8. If this was done during installation, check that no errors were made by compiling and running `main41.cc` or `main42.cc` in the `\examples` subdirectory.
- If these libraries were no linked, move to the `\pythia81XX` directory and run the following:

```
./configure --with-hepmc=path    path refers to the install directory of HepMC libraries
make                               this may take a minute or two depending on your computer
source config.sh                   run config.csh for tcsh or csh shells
```

- Once pythia8 has been reconfigured with HepMC, check that this has been successful by running `main41.cc` or `main42.cc`, as before.

Outputting to HepMC

Including and Initializing ToHepMC

```
#include "Pythia.h"
2#include "HepMCInterface.h" //Interface HepMC to Pythia8
#include "HepMC/GenEvent.h" //Generate HepMC event
4#include "HepMC/IO_GenEvent.h" //IO stream for HepMC event

6using namespace Pythia8;

8int main (int argc, char* argv[]) {

10 //Set up pythia to hepmc object
   HepMC::I_Pythia8 ToHepMC;
12 HepMC::IO_GenEvent ascii_io("test.hepmc", std::ios::out);

14 //Set up generation (using card file)
```

```

Pythia pythia; //Declare pythia object
16 pythia.readFile(argv[1]);
pythia.init();
18 int nEvents = pythia.mode("Main:numberOfEvents");
//Number of events as defined in .cmdd card file

```

- To print to HepMC, additional libraries must be included in the header, seen in lines 2-4. This includes the functions used to set up and write into the new `.hepmc` file.
- Before the Pythia object is initialized, two lines of code are needed to declare objects that will fill the event record in the `.hepmc` format, and the output stream to which this result will be printed.
- Note that the output file will be whatever string is the first argument of the function in the second line. In this code, I have chosen `test.hepmc`.

Printing to HepMC

```

20 // Set up histogram
22 Hist mul ("Charged_Multiplicity", 100, -1, 399);
int iMul = 0;
24 // Generate event(s) and HepMC event (for output file)
26 for (int iEvent = 0; iEvent < nEvents; iEvent++) {
28     pythia.next();
    iMul = 0;
30     for (int i = 4; i < pythia.event.size(); ++i){
32         if (pythia.event[i].isCharged() && pythia.event[i].isFinal()) ++iMul;
    }
34     //Fill the file
36     HepMC::GenEvent* hepmcevt= new HepMC::GenEvent();
    ToHepMC.fill_next_event( pythia , hepmcevt );
38     ascii_io << hepmcevt;
    delete hepmcevt;
40     //Fill the histogram
42     mul.fill( iMul );
}
44 //Print histogram, exit
cout << mul;
46 return 0;
}

```

- In the event loop, after all calculations and accessing, the HepMC file is filled in four steps. First an empty HepMC event is declared. Next, it is filled with the current event from the `pythia` object. The resulting format is printed to the output stream that was previously defined. After it is printed, the HepMC event is then deleted.
- Ensure that `hepmcOut.cc` is added to the `Makefile` in the appropriate location, i.e. where the other files that link to HepMC libraries are stored. Searching this file for “main41” should point out this location quickly.
- Note: This program also sets up a basic histogram in `pythia8` that counts the charged final state multiplicity. This is not needed for HepMC output files, but can be used as a quick check in many scenarios. To set up the histogram, declare the `Hist` object as in line 22, fill the histogram in the event loop, line 42, and print it to the standard output, line 45.

Card File

- Since the above calls for a card file, create a file `testHepMC.cmd` and fill it with the following general QCD, or with your own settings:

Card File

```
# Testing HepMC output in pythia8
Beams:idA = 2212      #incoming protons
Beams:idB = 2212      #incoming protons
Beams:eCM= 7000      #collision energy in CoM frame
HardQCD:all = on      #turn on all hard processes
SoftQCD:all = on      #turn on all soft processes
```

Running this program will print the normal pythia output, as well as its very basic histogram, and a HepMC file named `test.hepmc`.

3 Rivet

Now that we have written our first few pythia8 scripts, let us move on to using Rivet. Rivet is “a C++ class library, which provides the ... calculation tools for simulation-level analyses ... enabling physicists to validate event generator models with minimal effort and maximum portability.” This analysis tool reads final state particle information from Monte Carlo events to allow for easy validation and comparison to real data through publicly available results. The full manual can be found here [3].

There are several tutorials online, linked in section 4.1, all of which are powerpoint presentations. This section aims to streamline what is covered in these tutorials, as well as cover writing a very basic analysis from scratch.

3.1 Using Rivet with HepMC files

Rivet comes with many analyses built-in. The aim is that as more analysis scripts are written and used, they are added to the distribution. This way, if a similar analysis is needed, the existing code need only be altered to suit the new parameters or set up. The available of analyses are covered in detail in the Standard Rivet Analyses section of the manual, here [3].

Test Installation

- To test that Rivet has been installed correctly, enter the following command. It should list the built-in analyses available.

```
rivet --list-analyses
```

- A long list of analyses, followed by a very brief description, should appear. For a more detailed description, the `--show-analyses` option can be used. To list all of the ALICE analyses in detail, enter the following:

```
rivet --show-analyses ALICE
```

First Rivet Run

- Continuing with the Pythia8 example used before, we should have a `.hepmc` file ready for Rivet to analyze. A good tester analysis is provided under `MC_GENERIC`. This, and all other source codes for analyses, can be found under `$HEP/build/Rivet-1.8.3/src/Analyses`.
- While this example is more in-depth than this tutorial will go into, take a look at the source code before running. To run this analysis, simply enter:

```
rivet --analysis=MC_GENERIC filename.hepmc
```

- This should start with an initialization print out, and then start reading out the processing status, something like this:

```
Event 100 (0 s elapsed)
```

```
Event 200 (1 s elapsed)
```

```
Event 300 (2 s elapsed)
```

- It should then finish, and the result is a new `Rivet.aida` (default name) file. To generate some plots from this file type, enter

```
rivet-mkhtml Rivet.aida
```

- A similar message should appear, starting with:

```
Making 17 plots
```

```
Plotting ./plots/MC_GENERIC/E.dat (16 remaining)
```

```
Plotting ./plots/MC_GENERIC/ECh.dat (15 remaining) ...
```

- A new directory has been made, if it was not already there, called `/plots/MC_GENERIC`. Inside this directory, there should be a collection of `.pdf`, `.png`, and `.dat` files. The `.dat` files are ASCII data files of the corresponding plots formatted in `.pdf` and `.png`.
- Note: `rivet-mkhtml` is a good command for a quick check the the analysis works. For more in-depth plotting options, see the `make-plots` command, which has documentation linked in section 4.1.

3.2 Using Rivet with AGILE

The authors of Rivet offer the package AGILE (A Generator Interface Library and executable) to help steer event generators directly into a Rivet analysis. This allows for events to be analyzed as they are generated through a filesystem pipe (`.fifo`) connection. This program also allows for quick an easy analysis with one or two lines of code. Additionally, for generators that do not provide their own main program with the HepMC output, the `agile-runmc` command acts as an interface to Rivet. Note that Pythia8 has its own steering program called Sacrifice, whose documentation can be found here [2].

Test Installation

- To test that AGILe has been installed correctly, enter the following command.

```
agile-runmc --help
```

- A help menu should appear, which describes the basic functionality of `agile-runmc`.

```
rivet --list-gens
```

- This will list the generators available to `agile-runmc`. If you are running on a machine with the CERN LCG AFS area mounted, then `agile-runmc` will automatically detect and use the generators packaged by the LCG Genser team. If not, then you must build your own mirror of the LCG generators. Since this process is evolving, see appropriate documentation here [1].

First AGILe Run

- In order to save disk space and streamline the analysis process, create a filesystem pipe. Note that this must NOT be in an AFS directory.

```
mkfifo $PWD/hepmc.fifo
```

- In this example, we are using Pythia6 to simulate 2000 events at the LHC, and attaching the output to our `.fifo`. See `agile-run -help` for details on the argument structure.

```
agile-runmc Pythia6:426 --beams=LHC:8000 -n 2000 -o $PWD/hepmc.fifo &
```

- Now we attach rivet to the other end of the pipe:

```
rivet -a MC_GENERIC $PWD/hepmc.fifo
```

- The result should be a Rivet.aida file as before. Proceed with the `rivet-mkhtml` as outlined in the pervious section.
- If you want to do this with Pythia8, install Sacrafice, and use the following instead of `agile-runmc`, and continue as before.

```
pythia -n 2000 -c HardQCD:all=on $PWD/hepmc.fifo &
```

3.3 Writing an analysis

The authors of Rivet stress that the existing library of analyses should be comprehensive enough to cover most tasks. With that said, writing a basic analysis highlights some of the key features of Rivet. To open a template for a new analysis code, enter the following command:

```
rivet-mkanalysis HELLO_WORLD
```

This will create a `.info`, a `.plot` and most importantly, a `.cc` file. Open the `.cc` file in a text editor. This template outlines the three main parts common to any analysis file, listed below.

- `init()`: Projections are registered and histograms are booked.
- `analyze()`: Projections are applied and histograms are filled.

- `finalize()`: Histograms are normalized and/or scaled.

While one can define functions to use in the analysis section, most of the actual legwork in Rivet is done in the libraries, namely in the Projection libraries. For more on projections, see the Rivet manual [3]. The basic idea is that the relevant information is “projected” out of the event from the HepMC file and filled into the designated histogram files. Most of this relevant information is about the final state particles so that the simulated results match up with the experimental data.

This example is the most basic analysis structure. It will only look at charged final state (CFS) particles, and only cut by pseudorapidity.

Include Projection Libraries

Header

```

// -*- C++ -*-
2 #include "Rivet/Analysis.hh"
  #include "Rivet/RivetAIDA.hh"
4 #include "Rivet/Tools/Logging.hh"
  #include "Rivet/Projections/FinalState.hh"
6 #include "Rivet/Projections/ChargedFinalState.hh"

8 namespace Rivet {
10
12   class HELLO_WORLD : public Analysis {
13   public:
14     /// @name Constructors etc.
15     ///@{
16
17     /// Constructor
18     HELLO_WORLD()
19       : Analysis("HELLO_WORLD")
20     {   }
21
22     ///@}
23
24   public:
25
26     /// @name Analysis methods
27     ///@{
28

```

- The only change here to the template is that we have included the CFS projection. For more complicated analysis, jet analysis for example, more libraries are needed.
- No additions needed to be made to the constructor, and no additional functions are needed for the analysis. The next section to be edited is inside the `init()` function.
- Note: If any changes are made to the file name, ensure these changes are made to all occurrences of the file name in the code, including the class name, the constructor, the Analysis argument, above, and the `DECLARE_RIVET_PLUGIN` function at the end of the code.

Three Main Parts of Analysis

Initialize

```

/// Book histograms and initialise projections before the run
30 void init() {
32

```

```

//Define ChargedFinalState projection objects
34 const ChargedFinalState cfs2(-0.2,0.2);
const ChargedFinalState cfs4(-0.4,0.4);
36 const ChargedFinalState cfs6(-0.6,0.6);
const ChargedFinalState cfs8(-0.8,0.8);
38
/*
40 Add these projections. This function must be called. This
structure has to do with cacheing information from
42 within or between events
*/
44 addProjection(cfs2, "CFS2");
addProjection(cfs4, "CFS4");
46 addProjection(cfs6, "CFS6");
addProjection(cfs8, "CFS8");
48
//Book the histograms to be filled
50 _hist_cfs2 = bookHistogram1D("ChargeFS2", 100, -0.5, 199.5);
_hist_cfs4 = bookHistogram1D("ChargeFS4", 100, -0.5, 199.5);
52 _hist_cfs6 = bookHistogram1D("ChargeFS6", 100, -0.5, 199.5);
_hist_cfs8 = bookHistogram1D("ChargeFS8", 100, -0.5, 199.5);
54 }

```

- Four CFS objects are the first things that are defined. These objects have several constructors. The easiest way to look these up is in the source code ³. See section 4.1 for links to the source code.
- The constructor used cuts by pseudorapidity bins, where the first argument is a lower limit, and the second is an upper limit. Here, we have designated four such bins symmetrically around zero.
- The next block of code adds these projections, naming them in the process.
- Finally, we book the histograms that will be filled. In the constructor, they are named, the number of bins is specified, and the range of values is specified.

Analysis

```

/// Perform the per-event analysis
56 void analyze(const Event& event) {
const double weight = event.weight();
58
const ChargedFinalState& charged_02 =
60 applyProjection<ChargedFinalState>(event, "CFS2");
62 const ChargedFinalState& charged_04 =
applyProjection<ChargedFinalState>(event, "CFS4");
64
const ChargedFinalState& charged_06 =
66 applyProjection<ChargedFinalState>(event, "CFS6");
68 const ChargedFinalState& charged_08 =
applyProjection<ChargedFinalState>(event, "CFS8");
70
_hist_cfs2->fill(charged_02.size(), weight);
72 _hist_cfs4->fill(charged_04.size(), weight);
_hist_cfs6->fill(charged_06.size(), weight);
74 _hist_cfs8->fill(charged_08.size(), weight);
}

```

³This can be browsed here: http://rivet.hepforge.org/code/dev/a00491_source.html

- The `analyze` takes the current event as an argument, and applies the desired CFS projection. It then fills the histogram with the size of the cut, with the correct weight. Note that this weight is specified per event, at the beginning of the function.

Finalize

```

76  /// Normalise histograms etc., after the run
    void finalize() {
78
        normalize(_hist_cfs2);
80    normalize(_hist_cfs4);
        normalize(_hist_cfs6);
82    normalize(_hist_cfs8);
    }

```

- The `normalize(Histo1DPtr histo)` function normalizes the area of the histogram to 1 by default. Likewise, the function `scale(Histo1DPtr histo, double scale)` multiplicatively scales the given histogram by the given scale factor. These are the two functions typically used in the `finalize()` section.

Add Histograms

Closer

```

84    }
86    ///  

88    private:
89
90    // Data members like post-cuts event weight counters go here
91
92    private:
93
94    /// @name Histograms
95    ///  

96    AIDA::IHistogram1D *_hist_cfs2;
97    AIDA::IHistogram1D *_hist_cfs4;
98    AIDA::IHistogram1D *_hist_cfs6;
99    AIDA::IHistogram1D *_hist_cfs8;
100   ///  

101
102   };
103
104   // The hook for the plugin system
105   DECLARE_RIVET_PLUGIN(HELLO_WORLD);
106
107 }
108 }

```

- All that is left to edit is to add the histograms to the AIDA namespace, as in lines 96-99.

Compile and Run

- To compile the analysis script, use the following rivet command:

```

rivet-buildplugin RivetHELLO_WORLD.so HELLO_WORLD.cc

```

- After compilation, add the directory of the new analysis to RIVET_ANALYSIS_PATH using the following:

```
export RIVET_ANALYSIS_PATH=$PWD
```

- We can now run the HELLO_WORLD analysis on our HepMC file, generated from Pythia8 using the following:

```
rivet --analysis=HELLO_WORLD <path_to_hepmc_file>/filename.hepmc
```

- This will execute the analysis, as before, and generate the Rivet.aida file. Use the rivet-mkhtml command to generate plots, as before.

4 Resources

4.1 Useful Links

Pythia8

- Pythia Homepage, including all documentation and tutorials:
<http://home.thep.lu.se/~torbjorn/Pythia.html>
- Pythia Online Manual:
<http://home.thep.lu.se/~torbjorn/pythia81html/Welcome.html>
- Pythia Worksheet :
<http://hep.ps.uci.edu/~arajaram/worksheet.pdf>

Rivet

- Rivet Homepage:
<http://rivet.hepforge.org/>
- Rivet Tutorials and Manual:
<https://rivet.hepforge.org/doc>
- Rivet Source Code Browser:
http://rivet.hepforge.org/code/dev/dir_bf09416337d2e74436d2f8f2113abf3b.html
- Rivet make-plots command :
<http://rivet.hepforge.org/make-plots.html>

4.2 Particle ID Codes

Here are some of the most common particle ID codes, as in [5]. For all codes, see http://www.physics.ox.ac.uk/CDF/Mphys/old/notes/pythia_codeListing.html

1	d	11	e^-	21	g	211	π^+	111	π^0	213	ρ^+	2112	n
2	u	12	ν_e	22	γ	311	K^0	221	η	313	K^{*0}	2212	p
3	s	13	μ^-	23	Z^0	321	K^+	331	η'	323	K^{*+}	3122	Λ^0
4	c	14	ν_μ	24	W^+	411	D^+	130	K_L^0	113	ρ^0	3112	Σ^-
5	b	15	τ^-	25	H^0	421	D^0	310	K_S^0	223	ω	3212	Σ^0
6	t	16	ν_τ			431	D_s^+			333	ϕ	3222	Σ^+

References

- [1] Andy Buckley. Mirror documentation. [<http://rivet.hepforge.org/trac/wiki/GenserMirror>].
- [2] Andy Buckley. Sacrifice documentation. [<https://agile.hepforge.org/trac/wiki/Sacrifice>].
- [3] Andy Buckley, Jonathan Butterworth, Leif Lonnblad, Hendrik Hoeth, James Monk, et al. Rivet user manual. 2010.
- [4] Matt Dobbs. Hep mc 2. [http://lcgapp.cern.ch/project/simu/HepMC/20400/HepMC2_user_manual.pdf].
- [5] T. Sjöstrand R. Corke. Pythia 8 worksheet. [<http://home.thep.lu.se/~torbjorn/pythia8/mergingworksheet8160.pdf>], March 2012.
- [6] S. Mrenna T. Sjöstrand and P. Skands. Jhep05 (2006) 026, comput. phys. comm. 178 (2008) 852. [<http://home.thep.lu.se/~torbjorn/pythia81php/Welcome.php>].