Documentation of the project: ISR jet tagging

Author:
Andrés Felipe García Albarracín

Advisor:
Juan Carlos Sanabria, Ph.D.

June 18, 2015

Contents

Chapter 1

Introduction

During the last semester of 2014, I made my Undergraduate Thesis Project entitled "*Design of algorithms to identify high momentum Initial State Radiation (ISR) Jets in proton – proton collision events*", under the supervision of Juan Carlos Sanabria, Ph.D.. As the name suggests, the project consisted in the proposal of an algorithm to identify ISR jets. Due to the promising results, I was employed during the first semester of 2015 under the charge "Joven Investigador" of COLCIENCIAS in order to improve the initially obtained results. Throughout this time, several codes and programs were developed. To encourage the continuation of this project, this report has been written with a summary of all the technical work done so far.

In practical matters, one of the main drawbacks of Quantum Field Theory (QFD) is the inherent difficulty of its calculations. Feynman diagrams are not easy to solve and specially when high orders are involved. Consequently, the usage of algorithms and computer simulations have played an important role in the prediction of numerical results thanks to the great calculation power of modern computers. Several programs have been written with this purpose and today there exists a machinery which combines QFD, statistical models and Monte Carlo methods to reproduce High Energy Physics experiments.

In this project, three of those programs were used: MadGraph 5.2 (MadEvent) [1], Pythia 8.2 [2] [3] and Delphes 3.2 [4] with the aim of simulating proton - proton collision events. The description of those programs and their particular purposes in the project are described in chapter 2. In addition, chapter 2 includes the explanation of the codes and the scripts that were developed both to integrate those programs, and to run the simulations under specific conditions.

In despite of the fact that those simulations demanded a huge amount of computational time, they just served as inputs of the algorithms written throughout the project, which contain the main proposed analysis and ideas. Altogether, four algorithms were elaborated. Each of them are explained in chapter 3, where their documentation and an overall description are presented.

Finally, chapter four includes a brief description of some software tools that were introduced to the project. Specifically, this project used C++ codes which included root libraries instead of root macros. This transition reduced the execution time of the algorithms six times. Additionally, the development environment *Eclipse* was also introduced, which made easier the programming process. Overall, these tools dramatically improved the technical work of the project.

Chapter 2

Simulation chain

"*Divide et impera*",
"Divide and conquer"

Philip II of Macedon

At first glance, it is not clear why it is necessary to use three programs at the simulation stage instead of just one. The answer is quite simple: each one of those programs has been developed to run a specific task in the simulation process, and therefore, each one has been optimized to do so as accurate and fast as possible. While MadGraph and Pythia are responsible for the simulation of high energy collision's Physics, Delphes takes the final state particles produced by the former programs, and determines what would be the corresponding response of a detector. This scheme is useful as it maintains the detector apart from the main calculations of the simulation. Additionally, it makes the change of experiment parameters as simple as modifying Delphes execution specifications.

As presented before, MadGraph and Pythia handle the Physics of the collision. Again, there is more than a single program for this task, and now the reason to use two programs lies on the limits of the theoretical models. At the very first moment of the collision when the Energy Density of the System is high enough, perturbative Quantum Chromo-Dynamics (pQCD), Quantum Electro-Dynamics (QED) and ElectroWeak Theory are the most accurate models known so far. MadGraph, and specifically MadEvent, use them to calculate the transverse sections of a particular channel defined by the user. From this calculation and the Monte Carlo models, it randomly establishes the kinematic variables of the resulting particles of the collision.

Once the energy density of the collision has been reduced significantly, the models used by MadGraph are not valid, and then Pythia appears in the scene. The particles resulting from MadGraph are taken by Pythia, which makes the evolution to a multi-hadronic final state [2]. The task run by Pythia involves the usage of Monte Carlo techniques to simulate hadronization, decays and showers. Finally, the particles obtained at the end of the Pythia simulation are the inputs for the Delphes simulation.

Although the usage of several programs for the simulation means better results, it also implies the challenge of connecting them. This task has already been done inside the MadGraph package, which connects MadEvent + Pythia 6 + Delphes / PGS[1]. However, the version of Pythia included there (v.6) is old and does not offer the possibility of controlling ISR emissions as the last one (v.8) does. As ISR emissions were the main focus of the project, it was convenient to use Pythia 8 instead of Pythia 6 and therefore to develop the integration of MadGraph 5.2 with Pythia 8.2 and Delphes 3.2.

Throughout this chapter, the codes and scripts written to achieve the simulation will be explained. One section is devoted to each program and another one presents the script that connects the three programs. Finally, the last section of this chapter presents the procedure known as Matching between MadGraph and Pythia, which ensures that the physics calculations made by each program correspond to the Energy scale that each one of them should handle.

2.1 Usage of MadGraph 5.2

The most basic procedure to simulate collision events using MadGraph is by means of its executable program. Follow the next steps to run a set of simulations of the channel $p\ p\ \rightarrow t\ \tilde{t}$. It is important that MadGraph has

[1] *Pretty Good Simulation*, PGS, is another program for detector simulation

been correctly installed [2].

1. In the folder where MadGraph has been installed, type:
   `./bin/mg5_aMC`
2. Once MadGraph has been initialized, import the Standard Model parameters:
   `import model sm`
3. Generate the event $p\,p\ \rightarrow t\ \tilde{t}$:
   `generate p p > t t~`
4. Create an output folder where all the simulation files will be saved, in this case test_t_tbar:
   `output test_t_tbar`
5. Launch the Feynman diagrams production:
   `launch -m`
   and select the number of cores you want to use for the simulation
6. Turn off Pythia and other programs[3]. You can switch off and on by typing the number before the program (type 1 to toggle pythia, for instance). Then, press enter.
7. Modify the `run_card.dat` file by typing 2. Write `:32` and press enter to go to line 32, then type `i` and press enter to modify the file. Change the number of events from 10000 to 1000. Press `Esc` and write `:wq` to write and quit.
8. Press enter to run the simulation

Although simple, the latter approach is not the best as it requires the user interaction several times to configure the simulation, which is not desirable when more than a single simulation will be performed. In such situations,

[2] A full set of instructions to install MadGraph and other High Energy Physics programs can be found at `http://goo.gl/vigBdj`

[3] This project uses the last version of Pythia (8.2) instead of the sixth version that uses MadGraph

all the configuration parameters can be defined trough an input file. For the previous example, the input file would be:

```
import model sm
generate p p > t t~
output test_t_tbar -f
launch -m
2
pythia=OFF
Template/LO/Cards/run_card.dat
models/sm_v4/param_card.dat
```

where 2 corresponds to the number of cores used in the simulation, `run_card.dat` is the default file of MadGraph and `param_card.dat` contains the Standard Model parameters and values. Here, these two files correspond to the default ones that MadGraph provide. In order to use another set of configuration parameters, the files should be copied to another location and modified according to desired simulation conditions.

The input file may be saved as `mg5_input.mg5` and the simulation can be executed as:

`./bin/mg5_aMC -f mg5_input.mg5` [4]

As a result of the simulation by MadGraph, the output folder contains several folders with all the information related to the simulation. The folder `Cards` for instance, contains some parameter cards used in the simulation, while the folder `HTML`, and specially the file `info.html` present the Feynman diagrams created by MadGraph. The events resulting from the simulation are found in the folder `Events/run_01` in the form of two files: a root file called `unweighted_events.root` and a compressed Les Houches Event file with name `unweighted_events.lhe`.

[4] Observe that it is supposed that `mg5_input.mg5` is localed at the MadGraph folder and that the command is run from the same directory. If not, the execution instruction and the input file should contain the full path accordingly.

2.2 Usage of Pythia 8.2

The simulation carried out by MadGraph is now passed to Pythia, which takes the file `unweighted_events.lhe` as input. Pythia uses the information contained in such file to develop the hadronization, and produces another file with the kinematic variables of the resulting particles. The task performed by Pythia can be summarized in the Black Box of Fig. 2.1, where in addition to the file produced by MadGraph, a plain text file with extension `.cmnd` is passed by parameter to configure the simulation.

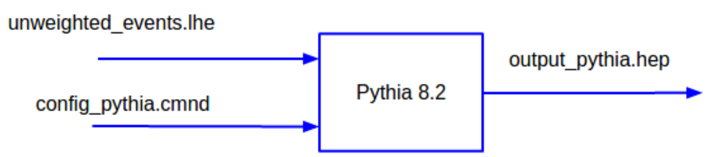

Figure 2.1

The functionality of the black box of 2.1 is done by a program written in C++, which is based on the examples provided by Pythia developers [3]. The code is called `hadronization02.cc`, was written in C++ and can be found at Appendix A. It performs specific requirements for this project that will be mentioned soon. Before presenting the operations performed by the program, it is convenient to describe how this code should be compiled and used.

2.2.1 Code Usage

To use `hadronization02.cc`, it is necessary to have installed Pythia[5] and StdHep[6] [5]. Once installed, go to the `examples` folder located at the Pythia directory [7]. Inside such folder, copy the code `hadronization02.cc` and

[5]Again, information to install Pythia 8.2 and HepMC can be found at `http://goo.gl/vigBdj`

[6]StdHep can be downloaded from `http://cepa.fnal.gov/psm/stdhep/getStdHep.shtml`. It is enough to type `make` to install it

[7]If `examples` is not exactly there, it may be in `share/Pythia8`

then modify the Makefile in order to compile it. It is enough to insert the following lines at the beginning of the Makefile:

```
1 # Include STDHEP libraries. The following 5 lines were sent by Mrenna.
2 STDHEP_DIR = <STDHEP Directory>
3 MCFIO_DIR = $(STDHEP_DIR)
4 SINC=$(STDHEP_DIR)/src/inc
5 INCS = -I$(SINC) -I$(STDHEP_DIR)/mcfio/src
6 LOCAL = -L$(STDHEP_DIR)/lib  -lstdhepC -lFmcfio -lstdhep -lm
```

changing <STDHEP Directory> in line 2 by the local installation directory of StdHep. Furthermore, these other lines should be included at the end of the Makefile:

```
# Hadronization. (To compile files that read .lhe files and produce stdhep files)
# No further modifications are needed to compile the class UserHooks
hadronization% : hadronization%.cc $(PREFIX_LIB)/libpythia8.a
	$(CXX) $^ -o $@ $(CXX_COMMON) $(INCS) $(LOCAL) -L$(PREFIX_LIB) -Wl,-rpath $(PREFIX_LIB) -lpythia8
```

After doing so, the code is compiled by typing on terminal:

```
make hadronization02
```

As a result, the executable file hadronization02 is created in the current folder. It may be copied and used in other directory. The instruction to run this program is:

```
./hadronization02 input.cmnd [output.hep]
```

where input.cmnd is the full name (with the path) of the configuration file, and output.hep is an optional parameter that corresponds to the name of the output file.

Continuing with the $t\,\tilde{t}$ production example of the previous section, the following file may be saved as `input.cmnd` and used as input of the Pythia simulation:

```
! Hadronization from a .lhe file
! This file contains commands to be read on a Pythia8 run.
! Lines not beginning with a letter or digit are comments.

// Specify statistics parameters.
Main:numberOfEvents    = 1000  ! number of events generated (It needs to be <= Number of events generated in MG)
Init:showChangedParticleData = off ! not useful info
Next:numberShowInfo    = 1  ! 1 to show info, 0 to not
Next:numberShowEvent   = 0  ! Especify the number of events that will be listed as output

// Read .lhe file
Beams:frameType = 4 ! To take a MG file as input
Beams:LHEF = unweighted_events.lhe  ! MG .lhe file

! Hadronization:
PartonLevel:FSR = off ! switch final state radiation
PartonLevel:ISR = on ! switch initial state radiation
PartonLevel:MPI = off ! switch off multiparton interactions
Random:setSeed = on ! For random seed
Random:seed = 1 ! any number between 1 and 900,000,000
```

Each line of this file is a different command, each of which is described after the exclamation mark character '!'. As it can be seen, 1000 events are hadronized, the file `unweighted_events.lhe` from MadGraph is read, and only ISR emissions are allowed.

2.2.2 The code

Having explained the procedure to compile and use the hadronization program, this subsection presents the code and what it does. As stated before, the code can be found in the Appendix A and also, in the repository of the project: `https://github.com/andresfgarcia150/ISR_tagging_project`, at the folder `/Codes/Simulation/Pythia_Codes/`, where the modified Makefile is also included.

Overall, the code can be described in terms of two procedures: the configuration and the execution of the simulation. The first of them, that corresponds to lines 76 - 106 in Appendix A, establishes all the parameters needed for the simulation. It starts with the definition of some Strings to be used by the StdHep methods, and an object of class `Pythia` in line 82. Then, in lines 84-93, the names of the input file (`.cmnd` file) and the output file are read from the execution instruction by means of `**argv`. Next, lines 95-98 define some variables to control the hadronization: `nEvent` corresponds to the number of events to be hadronized, while `nAbort` and `iAbort` are the maximum and current numbers of allowed events that present an error. Finally, the simulation configuration ends with some necessary functions to handle StdHep files (lines 100-102) and with the definition of an object of the class `MyUserHooks`.

The latter definition is extremely important for this project as it contains the restriction on the ISR emission. The object defined in line 105 belongs to the class `MyUserHooks`, which is written at the beginning of the code (lines 37-67). This class, in turn, inherits from `UserHooks` and just two of its methods are re-written: `canVetoISREmission()` and `doVetoISREmission()`. Each time an ISR emission is produced during the simulation of an event, the first of those methods stops the simulation and executes the second one, which counts the number of ISR partons produced so far and veto all the emissions in case that already exits one. This way, only one (or zero) ISR parton is produced in each event.

With the definition of the pointer `myUserHooks` and its inclusion in the object `pythia`, the configuration stage finishes. Then, the execution starts by initializing the simulation at line 109. Basically, the simulation consists of the *for* loop of lines 111-125, where each iteration corresponds to the generation of a new event through the call of method `pythia.next()`. Observe that if the latter method returns `false`, either pythia has reached the end of the input file (from MadGraph), or an error has happened and the execution should stop if the maximum number of errors is reached. Once this has been verified, each cycle ends by writing the event in the output `.hep` file.

After the simulation has been completed, the StdHep file is closed in line 127, some statistics of the simulation are published (line 128) and the pointer `MyUserHooks` is deleted. These lines conclude the code that develops the hadronization process.

2.2.3 Pythia ntuple generation

Although the file produced by the latter code is passed directly to Delphes, it cannot be read by ROOT. Therefore, it is necessary to develop a conversion from `.hep` to `.root`, which is performed by `ExRootAnalysis`. After having it properly installed, go to the installation directory and run the executable file `ExRootSTDHEPConverter` by typing:

```
./ExRootSTDHEPConverter output_pythia.hep output_pythia.root
```

where `output_pythia.hep` is the full path name of the file produced by the hadronization code and `output_pythia.root` is the output `ntuple`. This procedure makes possible the reading of the pythia simulation when executing C++ codes with Root libraries.

***

To summarize, it has been shown how to carry out simulations with MadGraph and Pythia 8.2. As a result of the simulation of MadGraph, the file `unweighted_events.lhe` is produced. Pythia receives that file as parameter and creates the file `output_pythia.hep`. To complete the simulation process, the next section will introduce Delphes, that takes the file generated by Pythia and performs the detector simulation.

2.3 Usage of Delphes 3.2

Because High Energy Experiments such as the Compact Muon Solenoid (CMS) and A Toroidal LHC ApparatuS (ATLAS) are already created and there is not much we can do to modify them, the simulation of those detectors is a simple task. To use Delphes, for instance, it is enough to have installed it and use the existent cards.

For the CMS simulation of the $t\,\tilde{t}$ production example that has been used throughout this chapter, go to the Delphes installation directory and use the execution file `DelphesSTDHEP`. To do so, type on the terminal:

```
./DelphesSTDHEP cards/delphes_card_CMS.tcl delphes_output.root
output_pythia.root
```

taking care that each one of the parameters should be replaced by the full path name of each file. With this instruction, `delphes_output.root` is generated and the files: `output_pythia.root` from the Pythia simulation, and `delphes_card_CMS.tcl` with CMS experiment specs are taken as inputs.

Appendices

Appendix A

Pythia code: hadronization02.cc

```
// Copyright (C) 2015 Torbjorn Sjostrand.
// PYTHIA is licenced under the GNU GPL version 2, see
   COPYING for details.
// Please respect the MCnet Guidelines, see GUIDELINES
   for details.

/*
-------     Universidad de los Andes      -------
-------      Departamento de Fisica       -------
-------   Proyecto Joven Investigador     -------
-------  Andres Felipe Garcia Albarracin  -------
-------    Juan Carlos Sanabria Arenas    -------

This code develops pythia hadronization. Takes as
parameter a .cmnd where a .lhe file from MadGraph
and other parameters are specified. Then the code
produces .hep files after making the hadronization

Obs: The class MyUserHooks is written in order to
veto all the ISR emissions produced after the
first ISR parton. It is an extension of the code
hadronization01
```

```

run as ./hadronization02 input.cmnd [output.hep]

The MakeFile has been also modified to compile
this file
*/

#include "Pythia8/Pythia.h"
#include "stdhep.h"
#include "stdcnt.h"
#include "stdhep_mcfio.h"
#include <string.h>

using namespace Pythia8;
void fill_stdhep(int i, Event &e);

// Write own derived UserHooks class.

class MyUserHooks : public UserHooks {

public:

        // Constructor.
        MyUserHooks() { }

        // Destructor.
        ~MyUserHooks() { }

        // Allow a veto of ISR emissions
        virtual bool canVetoISREmission(){
                return true;    // Interrupts the initial shower emission after each emission
                                // and allow the emission to be vetoed by the next method.
        }

        // Analize each emissionand asks for the number of the ISR emissions so far, in order
```

```
        // to allow just 1 ISR parton per event
        virtual bool doVetoISREmission(int sizeOld, const Event& event, int iSys){
                // counts the number of ISR partons (i.e. the numer of particles with status 43)
                int ISR_part = 0;
                for( int i = 0; i < event.size(); i++){
                        if (event[i].status() == 43 || event[i].status() == -43)
                                ISR_part ++;
                }
                if (ISR_part > 1)
                        return true;
                else
                        return false;
        }
};

//==================================================


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

        // Interface for conversion from Pythia8::Event to HepMC event.
        char fileout[500], title[100];
        strcpy(title,"output_pythia8\0");

        // Set up generation.
        // Declare Pythia object
        Pythia pythia;

        // Set simulation configurations. Read the file as parameter. If none, it reads hadro_input.cmnd
        if (argc > 1 ) pythia.readFile(argv[1]);
        else {
                cout << "ERROR:␣\n␣No␣␣parameters␣file␣has␣passed␣as␣parameter.␣Abort␣" <<
```

```
                    endl;
                return 1;
        }

        // Specify the name of the output file
        if (argc > 2 ) strcpy(fileout,argv[2]);
        else strcpy(fileout,"output_pythia8.hep\0");

        // Especify the number of events
        int nEvent = pythia.mode("Main:numberOfEvents"); // For reading only
        int nAbort = 10; // Maximum number of failures accepted
        int iAbort = 0; // Abortions counter

        // Necessary stdhep functions
        int istr(0);
        int ierr = StdHepXdrWriteOpen(fileout, title, nEvent, istr);

        // Set up to do a user veto and send it in.
        MyUserHooks* myUserHooks = new MyUserHooks();
        pythia.setUserHooksPtr( myUserHooks);

        // Initialize simulation
        pythia.init();

        // Begin event loop; generate until none left in input file.
        for (int iEvent = 0; iEvent < nEvent ; ++iEvent) {
                // Generate events, and check whether generation failed.
                if (!pythia.next()) {
                        // If failure because reached end of file then exit event loop.
                        if (pythia.info.atEndOfFile()) break;
                        // First few failures write off
```

```
                            as "acceptable" errors, then quit.
                         if (++iAbort < nAbort) continue;
                         break;
                 }

                 // Fill stdhep file
                 fill_stdhep(iEvent+1,pythia.event);
                 ierr = StdHepXdrWrite(1,istr);
        }

        StdHepXdrEnd(istr);
        pythia.stat();
        cout << ierr;
        delete myUserHooks;
        return 0;

}

// This functions writes in stdhep format. It was written by Steve Mrenna
void fill_stdhep(int i, Event &e)
{
   int num = e.size();
   hepevt_.nevhep = i;
   hepevt_.nhep = num;
   for (int j = 0; j < num; j++) {
      hepevt_.idhep[j] = e[j].id();
      hepevt_.isthep[j] = e[j].statusHepMC();
      hepevt_.jmohep[j][0] = (e[j].mother1()>0) ? e[j].mother1()+1 : 0;
      hepevt_.jmohep[j][1] = (e[j].mother2()>0) ? e[j].mother2()+1 : 0;
      hepevt_.jdahep[j][0] = (e[j].daughter1()>0) ? e[j].daughter1()+1 : 0;
      hepevt_.jdahep[j][1] = (e[j].daughter2()>0) ? e[j].daughter2()+1 : 0;
      hepevt_.phep[j][0] = e[j].px();
      hepevt_.phep[j][1] = e[j].py();
      hepevt_.phep[j][2] = e[j].pz();
```

```
        hepevt_.phep[j][3] = e[j].e();
        hepevt_.phep[j][4] = e[j].m();
        hepevt_.vhep[j][0] = e[j].xProd();
        hepevt_.vhep[j][1] = e[j].yProd();
        hepevt_.vhep[j][2] = e[j].zProd();
        hepevt_.vhep[j][3] = e[j].tProd();
    }
}
```

Bibliography

[1] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, et al. The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. *JHEP*, 1407:079, 2014.

[2] Torbjörn Sjöstrand, Stefan Ask, Jesper R. Christiansen, Richard Corke, Nishita Desai, et al. An Introduction to PYTHIA 8.2. *Comput.Phys.Commun.*, 191:159–177, 2015.

[3] Torbjorn Sjostrand, Stephen Mrenna, and Peter Skands. Pythia 6.4 physics and manual. *JHEP*, 05:026, 2006.

[4] J. de Favereau et al. DELPHES 3, A modular framework for fast simulation of a generic collider experiment. *JHEP*, 1402:057, 2014.

[5] Lynn Garren. *StdHep 5.06.01 Monte Carlo Standardization at FNAL Fortran and C Implementation.* Fermilab, 11 2006. Available at http://cepa.fnal.gov/psm/stdhep/.