Using Delphes

Michael Hutcheon

Summer 2016

1 Introduction

Delphes is a detector response simulation framework that takes as input a monte-carlo event file. It provides some basic simulation modules that allow for detector simulation out of the box as well as providing the framework to add new simulation modules in order to carry out more advanced studies. In this document I outline what I have learnt from using Delphes as part of a study for the CMS trigger upgrade, hopefully someone will find this useful.

2 Installing Delphes

In order to begin using Delphes it must be built from source. I recommend using the GitHub version of the code as it is updated more regularly than the version available on the website (a permalink to the version I am using is available in the appendix). The most recent version can be obtained using the following command, which should create a directory "delphes" in the current working directory:

```
git clone https://github.com/delphes/delphes.git
```

The next step is to configure the environment for ROOT and the gcc compiler. For lxplus users this can be done by running the following commands:

```
source /afs/cern.ch/sw/lcg/external/gcc/4.9.3/x86_64-slc6/setup.sh
source /afs/cern.ch/sw/lcg/app/releases/ROOT/6.06.00/x86_64-slc6-gcc49-opt/root/bin/thisroot.sh
```

Note that I use scientific linux 6 and ROOT version 6.06.00 (as can be seen from the above directories). It should now be possible to build delphes using:

```
cd delphes
make -j 8
```

You should now have a working installation of Delphes, but note that you will not be able to do anything with it until you have the necessary input (a monte-carlo event file). A quick check of the installation is to attempt to run delphes without an event file, by running the command:

```
./DelphesHepMC
```

This should produce an error message with the arguments that need to be passed to Delphes, in the order that they should be entered (a useful hint if you have forgotten this order).

3 Delphes input and output

Delphes uses a monte-carlo event file and a configuration card (.tcl) as input and, by default, produces a ROOT output file. Delphes is compatible with input files from many event generators (I use PYTHIA 8, with the HepMC format). The input file can be of one of the following formats: HepMC, STDHEP(XDR) or LHEF; each file format has it's own corresponding executable in the /delphes directory. Below are some example uses of each:

```
./DelphesHepMC card.tcl output.root input.hepmc
./DelphesSTDHEP card.tcl output.root input.hep
./DelphesLHEF card.tcl output.root input.lhef
```

Delphes can also be run with input files stored in CASTOR, or with files accessible via HTTP:

```
rfcat /castor/cern.ch/user/d/demine/test.hepmc.gz | gunzip | ./DelphesHepMC card.tcl output.root
curl http://cp3.irmp.ucl.ac.be/~demin/test.hepmc.gz | gunzip | ./DelphesHepMC card.tcl output.root
```

The locations of the input and output files are specified relative to the /delphes directory; for example, to run delphes with one of the default configuration cards provided with Delphes (specifically delphes_card_CMS.tcl) one would use the following syntax:

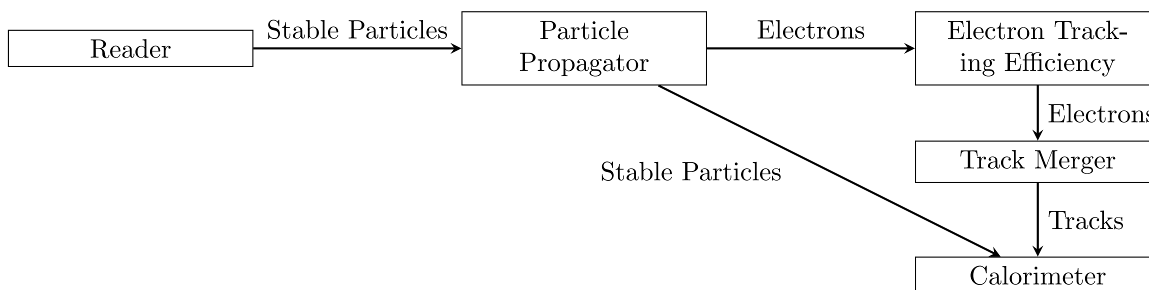
```
./DelphesHepMC cards/delphes_card_CMS.tcl output.root input.hepmc
./DelphesSTDHEP cards/delphes_card_CMS.tcl delphes_output.root input.hep
./DelphesLHEF cards/delphes_card_CMS.tcl delphes_output.root input.lhef
```

These cards are the most common (and easiest) method to spedicfy/modify the operation of delphes. The next section explains how delphes operates and how card files are used to control it's operation.

4 Delphes Operation

4.1 Modules

Delphes is based on a modular system, whereby the input file is read and the information within is passed though a series of modules which make alterations to the passing data. The data is passed between modules in the form of an array of 'candidates' (a delphes class type). These candidates are designed to represent any object that may be useful for detector simulation; for example a candidate may represent a particle, track, jet or some other object depending on context. An example of a section of this modular structre is given below:



This example takes stable particles from the reader module (which read the particles from the input file) and propagates them to the outer edge of the detector with the 'particle propagator' module; this will then passes stable particles to the 'calorimeter' module, which will record their energy deposits. However, a tracking efficiency formula is applied to the electrons in the 'electron tracking efficiency' module, which will discard some of the electrons; the remaining electrons are then passed into a track merger which will take the input electrons and convert them into tracks in the detector, which are then passed to the calorimeter.

Each of the arrows in this diagram represents one of the aforementioned arrays of candidates. If we wanted to, we could pass any of the outputs from one module into any of the inputs of another (although it might not make sense to do so); this modular structure is specified in the card file (see next section).

The operation of a module is specified in C++ code and is compiled as part of Delphes; this code is located within the /modules directory of the installation. Several modules come with Delphes; these constitute it's out-of-the-box operation. It is possible to write your own modules and recompile Delphes with the new functionality that they may bring; see the section on adding modules later.

4.2 The Delphes Card File

The card file is one of the inputs to Delphes, it is written in the tcl scripting language (although knowledge of this language is not necessary to be able to use the card files, they are very human-readable). The card file tells Delphes which order to execute modules and specifies the data flow between modules. Some example card files are included in the /cards directory of the Delphes installation; it's worth having a look at some of these to familiarise yourself with the layout. The card files allow for the moddeling of a wide variety of detector simulations; cards are provided in order to model several particle physics experements (CMS, ATLAS etc...), as well as to run them with various settings (e.g with Pileup, or without Jet Reconstruction). The general structure for a card file is something like:

```
set ExecutionPath {
    moduleName1
    moduleName2
    ...
}

module moduleClass1 moduleName1 {
    set/add variable1 value
```

```
        set/add variable2 value
        ...
    }

    module moduleClass2 moduleName2 {
        set/add variable1 value
        set/add variable2 value
        ...
    }

    ...
```

The "ExecutionPath" entry (at the top of the example cards) tells Delphes what order to execute modules; the module names within this entry refer to instances of module classes that are also declared within the card file following the pattern above.

Delphes modules may require the user to specify parameters of operation as well as links to other modules in the form of input and output arrays. A more complete example of the declaration of a module (specifically the Electron tracking efficiency module from earlier) is shown below:

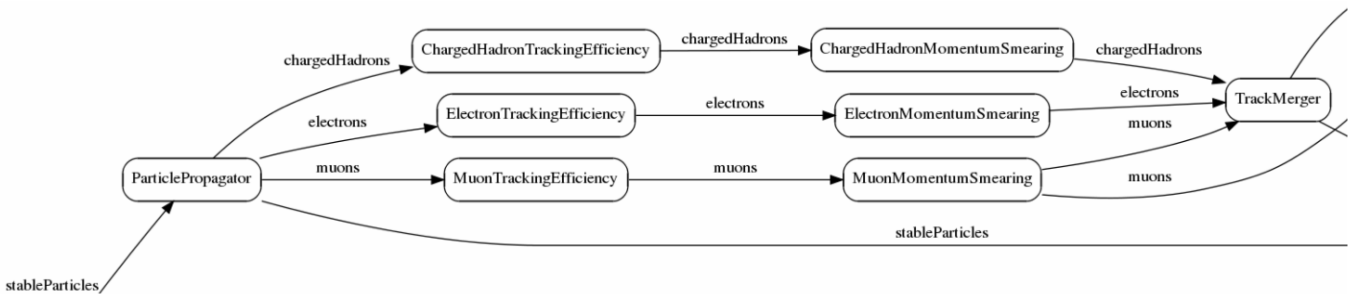
```
module Efficiency ElectronTrackingEfficiency {
    set InputArray ParticlePropagator/electrons
    set OutputArray electrons
    set EfficiencyFormula {0.8}
}
```

This module will take an input array called "electrons" from the ParticlePropagator module, apply an efficiency formula to remove some of these electrons from the array and then output the modified array under the name "electrons". As before, these arrays will contain 'Candidates' (see the start of the modules section above).

A useful tool when looking at card files is a data flow diagram. This diagram can be generated using a script included within delphes by running the command:

```
./doc/data_flow.sh cards/cardName.tcl
```

From the Delphes installation directory; this should create a file called data_flow.png, also in the installation directory. Below is an example of part of the data flow diagram generated for the delphes_card_CMS.tcl card file:



The data flow for some cards can be quite complex, so being able to see it like this is extremely useful.

4.3 Specifying Delphes output

A particularly important module is the TreeWriter module, which appears in all of the example cards (at the bottom). This module takes arrays of candidates from other modules and writes them to the ROOT output file. If you need access to data from any stage of Delphes operation, this is where you can get it. For example if you need access to the tracks that the TrackMerger module produces, you might have in your card file:

```
module TreeWriter TreeWriter {
    add Branch TrackMerger/tracks MergedTracks Track
}
```

This specifies a new Branch in the ROOT file, called "MergedTracks", containing all of the Candidates from the TrackMerger/tracks array. It also specifies that these candidates should be typecast as members of the 'Track' class. The general syntax is:

```
add Branch ModuleName/ArrayName BranchName ClassName
```

4.4 Using Delphes Output

The output ROOT file from Delphes may be accessed in several ways. One of the most convenient ways is using a ROOT macro (C code interpreted within ROOT). Several examples of how to access data within the ROOT output are included under /examples in the Delphes installation directory; these examples use several custom classes included within Delphes for reading data from the ROOT file. Below is the basic scheme for reading the output file:

```
//------------------------------------------
// ExampleReader.C
//------------------------------------------
#ifdef __CLING__
R__LOAD_LIBRARY(libDelphes)
#include "classes/DelphesClasses.h"
#include "external/ExRootAnalysis/ExRootTreeReader.h"
#endif

void ExampleReader(const char *ROOTfile) // This function must have the same name as the file
{
    // Load the Delphes shared libraries
    gSystem->Load("libDelphes");

    // Load the data from the root file
    TChain chain("Delphes");
    chain.Add(ROOTfile);
    ExRootTreeReader *treeReader = new ExRootTreeReader(&chain);

    // Get a branch (a specific data series) from the root tree
    TClonesArray *branch = treeReader->UseBranch("Branch Name");

    // Loop over all events within the ROOT file
    int eventCount = treeReader->GetEntries();
    for(int event = 0; event < eventCount; ++event)
    {
        // Load branches with data from this event
        treeReader->ReadEntry(event);

        // Run over each entry in the branch
        int branchEntryCount = branch->GetEntries();
        for (int branchEntry = 0; branchEntry < branchEntryCount; ++branchEntry)
        {
            // Obtain the objects within the branch as both candidates
            // or as some DelphesType (Jet, Track, GenParticle etc...)
            Candidate *Cand = static_cast<Candidate*>(branch->At(branchEntry));
            DelphesType *Type = static_cast<DelphesType*>(branch->At(branchEntry));
        }
    }
}
```

Note the typecasts to the Candidate type and to some DelphesType. These types are intended to represent objects within the detector. Below is an example snippet of how to output the particle ID's of the particles that are associated with tracks within Delphes:

```
    for (int n=0,ntest=trackBranch->GetEntries(); n<ntest; ++n)
    {
        // Get track and associated particle
        Track *t = static_cast<Track*>(bTrack->At(n));
        GenParticle *p = static_cast<GenParticle*>(t->Particle.GetObject());
        cout << p->PID << "\n";
    }
```

This code relies on the trackBranch TClonesArray* to be initialized as in the previous example:

```
    TClonesArray *trackBranch = treeReader->UseBranch("Track");
```

You should now be able to use the default features of Delphes, the following sections deal with more advanced topics if this default functionality is insufficient.

5 Running Delphes with pile-up

In order to run an event with pile-up interactions, Delphes needs both an input file for the hard event (for example a t-tbar production) as well as a sample of events to use as pile-up (often a minimum bias sample). The hard event can be in any of the standard formats that Delphes can use (HepMC, STDHEP(XDR) or LHEF). However, the pile-up input must be converted into Delphes own .pileup format. A pre-converted file 'MinBias.pileup' is included with Delphes (it contains 1000 13 TeV minimum bias p-p collisions) or you can convert your own sample from hepmc, stdhep or root format by running one of:

```
./stdhep2pileup MinBias.pileup MinBias.hep
./hepmc2pileup MinBias.pileup MinBias.hepmc
./root2pileup MinBias.pileup MinBias.root
```

From the Delphes installation directory (this will overwrite the previous MinBias.pileup). Note that in the conversion to .pileup format some event information is lost so it is worth keeping a copy of the original. To get Delphes to run with pile-up we must use a card that specifies using pile-up interactions. Some are provided, and can be run in the normal way (with your choice of hard-event format):

```
./DelphesSTDHEP cards/delphes_card_CMS_PileUp.tcl output.root input.hep
./DelphesSTDHEP cards/delphes_card_ATLAS_PileUp.tcl output.root input.hep
```

These will read the file 'MinBias.pileup' from the Delphes installation directory as well as the hard event 'input.hep'.

6 Adding Modules to Delphes

In order to extend the operation of Delphes it may be necessary to write new modules and specify where they fit into things in the card file. All delphes modules inherit from the class DelphesModule and must implement three functions theirin: Init(), Process() and Finish(). These are called by Delphes to run the module. Init is called when the module is first loaded, process is called as each event gets passed through to the module and Finish is called when there are no more events.

To implement a new module we must create two files in the /modules directory: a header file and a c++ source file. Below is an example of how these files should look:

```
//------------------------------------------
// modules/exampleModule.h
//------------------------------------------
#ifndef exampleModule_h
#define exampleModule_h
#include "classes/DelphesModule.h"

class TObjArray; // Forward decleration
class exampleModule: public DelphesModule
{
public:
    exampleModule();
    ~exampleModule();
    void Init();
    void Process();
    void Finish();
private:
    const TObjArray *fInputArray;
    TObjArray *fOutputArray;
    ClassDef(exampleModule, 1)
};
#endif

//------------------------------------------
// modules/exampleModule.cpp
//------------------------------------------
#include "modules/exampleModule.h"

exampleModule::exampleModule(){} // Class constructor
exampleModule::~exampleModule(){} // Class destructor
```

```
void exampleModule::Init(){
    // Get the input array and create the output array
    fInputArray = ImportArray(GetString("InputArray", "InputModuleName/InputArrayName"));
    fOutputArray = ExportArray(GetString("OutputArray", "OutputArrayName"));
}

void exampleModule::Process(){} // Process each event in this function
void exampleModule::Finish(){} // Called once on finish
```

The next step is to add the new modules to the modules/ModulesLinkDef.h file by inserting the following lines:

```
#include "modules/IsolationLEP.h"
#pragma link C++ class IsolationLEP+;
```

It should be clear where they go based on the modules already listed. The last step is to reconfigure and rebuild Delphes by returning to the installation directory and running the following commands:

```
./configure
make -j 8
```

7 Useful Links

Delphes workbook:

https://cp3.irmp.ucl.ac.be/projects/delphes/wiki/WorkBook

The Delphes ticket system (Has a lot of troubleshooting information):

https://cp3.irmp.ucl.ac.be/projects/delphes/report

A tutorial:

https://indico.cern.ch/event/315979/

8 Appendix

Version of delphes I used at the time of writing:

https://github.com/delphes/delphes/tree/659c7b6c6b8deb696fe0eb5ab84bf3c546f4e8fb