Muon Tomography Simulations Using GEANT4 and the Cosmic Ray Shower Library on a Linux Cluster

Rafael Pena (rpena@fit.edu)

Florida Institute of Technology
Department of Computer Sciences
Melbourne, Florida USA

Abstract

This paper encapsulates the process of building a Geant4 application and running it on a Linux Cluster running Condor. Our application is designed to simulate cosmic ray muons traversing different materials. We use the Cosmic Ray Shower Library (CRY) to create our input cosmic ray muon at varying energies.

1. MuonApp

In the course of the last semester I was able to put together an application that simulated a cosmic ray muon traversing a lead box. Slowly we have modified this application to simulate more complex situations. To properly use Geant4 for our purposes we needed to implement CRY and install Geant4 on the network attached Storage (NAS). The CRY Implementation allows us to simulate cosmic ray muons. While installing Geant4 on the NAS will afford us the computing power of the cluster.

1.1. Essentials

I was able to build the MuonApp with the help of the Geant4 examples, the MOMO utility, and some online tutorials. Using these utilities I created a simple GEM detector which produced some output points. Using the detector we then feed the data into the POCA application implemented by R. Hoch. To get a properly working application with Geant4 we need a minimum of three classes G4VUserPrimaryGeneratorAction, G4VUserDetectorConstruction, and G4VUserPhysicsList.

These classes must be implemented in our source code. The respective classes we use to call these are MomoPrimaryGeneratorAction.cc, muonDetectorConstruction.cc, and muonAppPhysicsList.cc. These three classes do most of the work in our program. The main application muonApp.cc simply calls these are their appropriate times.

1.2. Primary Generator Action

The Primary Generator Action was the hardest class for me to implement as it had the most c++ intensive code. In our application MomoPrimaryGeneratorAction.cc is in charge of handling the initialization of the particles we need in our simulation. It enables all of the CRY parameters and generates all the random numbers used in the application. Here we also generate our input angle distribution from CRY.

To create this class I used MOMO which comes with Geant4. MOMO is a utility that helps create the classes required to build a Geant4 application. Although we don't use this anymore this turned out to be a very useful tool to understand the way Geant4 created and used its objects.

1.3. Detector Construction

The Detector Construction is the simplest class to implement as its parameters are very basic. We can create a variety of geometries and place them where ever we chose. This class produces our world, detector and different materials.

In Figure 1 we see how to initiate a new material and in Figure 2 we show how we make a box out of our material.

With only a few lines of code we can create a box

```
G4Material* Pb = new G4Material("Lead",
 82, 207.19*g/mole, 11.35*g/cm3);
```

Figure 1: Creating a new material

```
//Here we create 10 cm lead box
    G4double block_x = 5.0*cm;
    G4double block_y = 5.0*cm;
    G4double block_z = 5.0*cm;
    G4Box* lead_box = new G4Box("leadBox_box",
                                block_x,
                                block_y,
                                block_z);
    leadBox_log = new G4LogicalVolume(lead_box,
                                      Pb,
                                      "leadBox_log",
                                      0,0,0);
    G4double blockPos_x = 0.0*m;
    G4double blockPos_y = 0.0*m;
    G4double blockPos_z = 0.0*m;
    leadBox_phys = new G4PVPlacement(0,
                                     G4ThreeVector(blockPos_x,
                                                   blockPos_y,
                                                   blockPos_z),
                                     leadBox_log,
                                     "leadBox",
                                     experimentalHall_log,
                                     false,0);
```

Figure 2: Creating a box made of lead

give it a name, pass its dimensions, make it a certain material and place it anywhere inside our world. Notice that to make 10 cm box we actually have to set the dimensions to 5 cm. Geant4 creates objects from the center which took us a while to figure out.

1.4. Physics List

The physics list took quite some time to get working as MOMO would not help produce these without some technical knowledge of Geant4. When I first created Geant4 application I only added one physics process to the entire application. I was unsure as to how Geant4 added the physics at the proper time. The solution to this problem actually turned out to be very simple.

After attempting to enable only certain physics process I found an application which explained that it is best to set up all the particles to ensure we don't miss anything. Then setup the physics process for the particles we want to use. So in our case we setup the Leptons, Mesons, Baryons and for Bosons we only set up the photon. Once these particles are created we can then give them Physics processes. In Figure 3 we can see the physics processes that were given to the muon.

```
if (particleName == "mu-" || particleName == "mu+") {
  pmanager->AddProcess(new G4MultipleScattering, -1,1,1);
  pmanager->AddProcess(new G4MuIonisation(),-1,2,2);
  pmanager->AddProcess(new G4MuBremsstrahlung(),-1,3,3);
  pmanager->AddProcess(new G4MuPairProduction(),-1,4,4);
}
```

Figure 3: Physics Processes of a muon

1.5. Cosmetics

Although with these three Classes we can run a simple Geant4 application most – if not all – useful applications will require some cosmetics such as visualization, user interface, well defined process handling and some output. In our muonApp we have another six classes which work in conjunction with these. The additional classes handle parts of the random number generation process, visualization, and output of our data into text files. In the future we will need more classes to handle the AIDA interface.

2. CRY

To create our simulation we had to make use of CRY which has a built in energy and momentum direction distributions. CRY has an interface for Geant4 which allows us to specify parameters such as the number of particles, position of the created particles and height in the atmosphere of the particle. CRY comes with a simple example to combine it with Geant4. We used the example code with very few changes. The Geant4 application built to include CRY must have an interface written into "Primary Generator Messenger" class. This class will allow Geant4 to change parameters in the CRY implementation. The "Primary Generator Action" class must be completely rewritten to initiate CRY and insert the specified particle into Geant4.

One of the main changes we needed to make to CRY was our random number seed value. To get a random number we used the system time in microseconds and inserted the last 6 digits into the CLHEP random number generator. This allows us to get a unique seed every time we run CRY. Figure 4 shows this implementation. One of our goals when running our simulations was to verify that CRY was producing the correct input angle distribution. To get this distribution we simply output calculate the angle using

$$\theta_{rad} = \frac{-p_z}{\sqrt{p_x^2 + p_y^2 + p_z^2}}.$$

```
CLHEP::HepRandomEngine* MyRandomEngine =
        CLHEP::HepRandom::getTheEngine();
timeval tim;
gettimeofday(&tim, NULL);
double t1=tim.tv_sec+(tim.tv_usec)*100000;
if (t1 < 0) {
        t1=t1*(-1);
}
long seedValue = (long) t1;

MyRandomEngine->setSeed(seedValue,1);
RNGWrapper<CLHEP::HepRandomEngine>::
        set(MyRandomEngine,&CLHEP::
        HepRandomEngine::flat);
setup->setRandomFunction(
        RNGWrapper<CLHEP::
        HepRandomEngine>::rng);
InputState=0;
```

Figure 4: Getting Random Number From System Time

Which we then convert to degrees using

$$\theta_{deg} = \frac{180 \cdot \theta_{rad}}{4 \cdot \tan^{-1}(1)}.$$

Using this method we get the distribution in Figure 6. This is a problem as the distribution should have

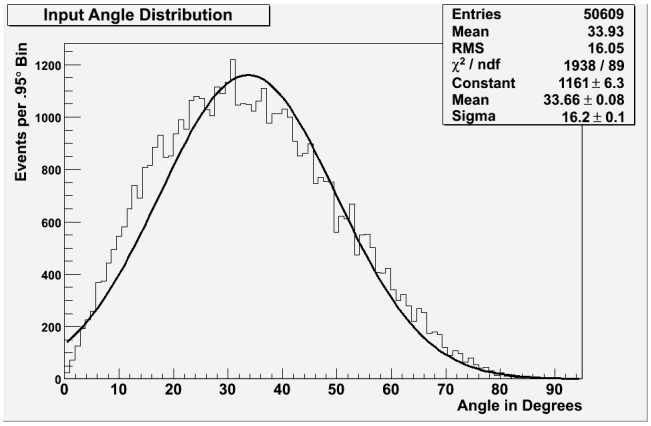

Figure 5: Input Angle Distribution

a maximum around 0 degrees.

3. NAS Installation

To Install Geant4 on the NAS we created a user called "geant4". This user will be accessible to everyone who wants to run Geant4 batch jobs on the cluster. This user home directory will exists in `/mnt/nas1/home/geant4` where the configured applications will be installed. All data will also exist in this directory. Placing all the data and the geant4 installation on the NAS allows for an efficient backup for if the frontend was to crash a machine could easily be inserted with no loss of data.

We needed to make some environmental variables available to condor to run Geant4 on the cluster. With Geant4 compiled in the geant4 home directory we added the lines in Figure 6 to the `/.bashrc` file:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
       /mnt/nas1/home/geant4/geant4.8.3/lib/Linux-g++:
       /mnt/nas1/home/geant4/CGCinstall/bin/lib
source /mnt/nas1/home/geant4/geant4.8.3/env.sh
source /mnt/nas1/home/geant4/cry_v1.2/setup
```

Figure 6: Environmental Variables

Additionally we need to add the lines in Figure 7 to get AIDA running properly:

```
export HEPRAPP=/mnt/nas1/home/geant4/HepRapp
alias HepRApp="java -jar $HEPRAPP/HepRApp.jar"

# JAVA and AIDA
export JDK_HOME=/usr/java/jdk1.5.0_07

#export JAIDA_HOME
export JAIDA_HOME=/mnt/nas1/home/geant4/CGCinstall/JAIDA-3.2.3
source $JAIDA_HOME/bin/aida-setup.sh

#export AIDAJNI_HOME
export AIDAJNI_HOME=
       /mnt/nas1/home/geant4/CGCinstall/AIDAJNI/AIDAJNI-3.2.3
source $AIDAJNI_HOME/bin/Linux-g++/aidajni-setup.sh
```

Figure 7: Additional Variables for AIDA

Notice that all the required scripts reside on the NAS.

3.1. Condor Jobs

To run Geant4 through condor we have to ensure condor has all the environmental variables to run the application. Therefore we created a simple script which would first initialize all the variables and then execute the Geant4 application. Figure 8 shows a script that can be submitted through condor. Notice that this figure runs `/bin/sh` which is then given an argument `script.sh`.

Figure 9 shows the contents of the `script.sh` file. The script first ensures that the environmental variables are available and the proceeds to run the application `exampleN01`.

```
Universe   = vanilla
Executable = /bin/sh
Arguments  = script.sh
Log        = exampleN01.log
Output     = exampleN01.out
Error      = exampleN01.error
Queue
```

Figure 8: Condor Script

```
#Source the .bash_profile
source ~/.bash_profile
#Running example1
/mnt/nas1/home/geant4/geant4/bin/Linux-g++/exampleN01
```

Figure 9: Condor Job

We are looking for a better solution to this as it is inconvenient to have a script that calls another script. This is a temporary hack.

To submit a job like this to run we would simply run

```
# condor_submit condorScript
```