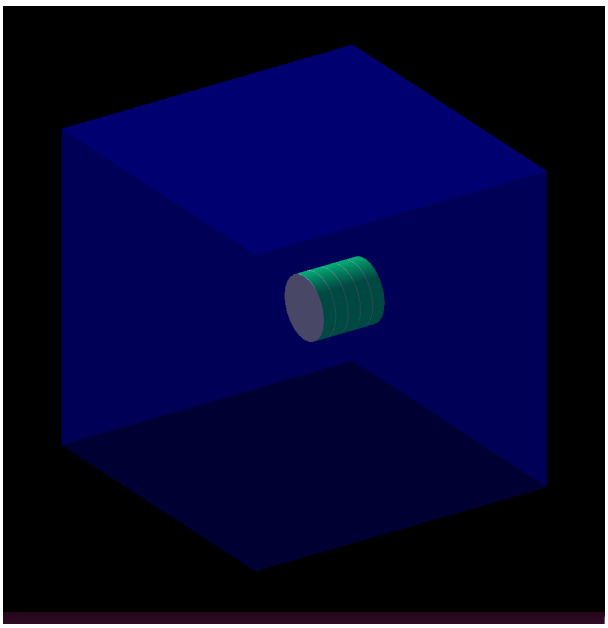
PDAML Geant4 Exercises 2

Ben Wynne (bwynne@cern.ch)

Please submit all modified C++ code, all Python code that you create, and any other outputs requested. Python may be presented in a Jupyter notebook

1 Detector development

As usual, I've provided an example program to get you started: download `GeantExample2Part1.tgz` from the course LEARN page. Decompress the file, compile and run the code just like before, and take a look at your new prototype of a sampling calorimeter:



Note the thin layers of lead (grey), and larger sensitive volumes of liquid argon (green). This structure is defined in `src/DetectorConstruction.cpp` as usual, using a `for` loop to create the repeating pattern. Rather than have all the data go into separate files like last time, I've changed the code a little to put the information from each layer into columns of the same ntuple. To do this meant changing `src/EnergyCounter.cpp` to use the columns in this shared ntuple. I've also had to add a new action, `src/EventAction.cpp`, that moves to a new row in the ntuple after every event.

Try running the simulation for 20 events or so (`/run/beamOn 20`) and have a look at the output files. You should find an ntuple containing raw data, and a histogram in both `.ps` and `.csv` formats. The histogram shows how energy is distributed through our detector: each histogram bin is a different layer of the sampling calorimeter. You can see how hadronic and electromagnetic showers differ by adjusting which particles are fired into the simulation. Electromagnetic showers will not travel far into the detector, so most of the energy should be deposited in the first layers. Hadronic showers travel much further, and increasing the beam energy will push either kind of shower further through the detector as well.

Show how the energy distribution histogram changes for electrons and neutrons of the same initial energy. [1 mark]

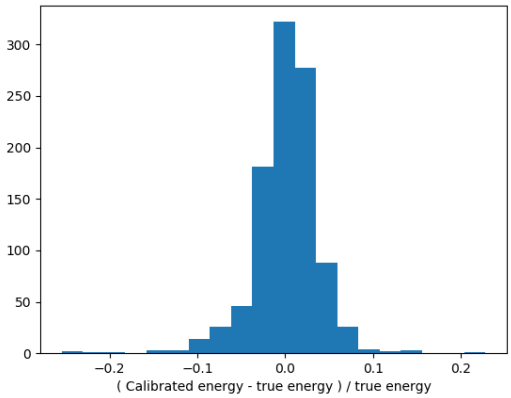
The raw data ntuple has a simple format: the first column is the beam energy, and the other columns are the energies deposited in each detector layer. Remember that this is a sampling calorimeter, and so a large fraction of the total energy will be absorbed by the lead plates and not detected. We need to understand our detector performance by calibrating for the energy lost in the absorbers and calculating the energy resolution. This is probably best to do in Python, using the tools you are already familiar with. Generate a reasonably large ntuple (a few hundred events) to reduce statistical errors.

Write a Python program to load the ntuple. If you want to use `pandas` (you don't have to!) then you'll need to specify that the first few lines of the input are comments:

```
import pandas as pd

inputData = pd.read_csv( 'myNtuple.csv', comment='#', \
  names=['TrueEnergy', 'Layer1', 'Layer2', 'Layer3', 'Layer4', 'Layer5'] )
```

Since we know that all of the energy we detect has come from the original particle beam, and we control exactly what that was, we can now calibrate the detector. Let's use a very simple calibration: the value of $E_{true}/E_{detected}$ averaged over all events. Assume this is constant, and multiply the detected energy in an event by this constant to retrieve the calibrated value. The detector resolution is the standard deviation of $(E_{calibrated} - E_{true})/E_{true}$ for all events.

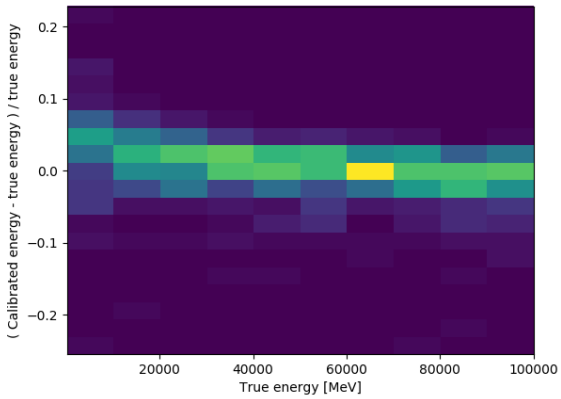


Write a Python program to calibrate your energy data. Plot a 1D histogram of the calibrated energy as shown, and find the resolution by taking the standard deviation of this data. [1 mark]

As you've already seen, the energy distribution in our detector will vary depending on the incident particle type and energy. This could also affect the calibration and resolution of the detector. We can examine the trend with energy by creating a 2D histogram. As we increase the true beam energy along the x-axis, each bin essentially contains a whole 1D histogram along the y-axis for the calibrated energy.

```
import matplotlib.pylab as pl

pl.hist2d( x=trueEnergy, y=calibrationQuality, bins=(10, 20) )
```



We're going to need a large number of events to create a plot like this. At this point, the user interface and visualisation tools of Geant begin to get in our way. Although they are helpful for debugging our simulation, once it's working we just want to run it quickly and efficiently. We can disable these components by commenting out these four lines in the `src/main.cpp` file:

```
G4VisManager* visManager = new G4VisExecutive();
visManager->Initialize();

...

ui->SessionStart();

...

delete visManager;
```

Now you control the simulation entirely through `run.mac`. Set up the beam as you need it, and make sure there's a `/run/beamOn` command at the end of the file. Your code will just simulate those events quickly, and quit. The raw data ntuple is produced just like before.

You can manually change the true beam energy several times, and load each ntuple separately, but this is quite tedious. Remember that each `/run/beamOn` command starts a new run, which will overwrite your existing data. So, if you want to automate changing the beam energy, you need to do this inside the run. The easiest way to do this is to add some code to `src/GeneratorAction.cpp` that will increase the beam energy each time a particle is fired.

Make a 2D histogram showing how your calibrated energy varies with beam energy. Describe what beam energy values you used. [2 marks]

NB: For the plot above I used 1000 events, with 100 MeV energy increases between them. It took about 1 minute to simulate. It might be more interesting to cover a larger range.

2 Finding the truth

Now set up `GeantExample2Part2.tgz` from the course LEARN page. I've added another action — `src/SteppingAction.cpp` — in order to examine what's really happening in the simulation. Each step of the simulation is an opportunity for something to change, and this code can examine every step. My example summarises information from all the steps in a single event. The "once per event" action actually does the calculations, and the step action is only used to collect information. The actions are set up in `src/ActionInitialization.cpp` like this:

```
// As before
this->SetUserAction( new GeneratorAction() );
this->SetUserAction( new RunAction() );

// Keep the event action pointer to use below
EventAction* myEvent = new EventAction();
this->SetUserAction( myEvent );

// Stepping action needs a pointer to the event action
this->SetUserAction( new SteppingAction( myEvent ) );
```

Take a look at `src/SteppingAction.cpp`: all it does is send the step information away to be analysed. At the bottom of `src/EventAction.cpp` you can see where the analysis is done. For example, this is how to find what type of particle defines the track on which this step occurs:

```
G4Track * track = step->GetTrack();

// Get the type of particle that made this track
std::string particle = track->GetParticleDefinition()->GetParticleName();
```

At the moment, my code counts how many photons are produced through electron Bremsstrahlung. It also counts the total number of particles in the event, and the relationships between particles: which created which. Remember that one particle track can have many steps, so I use a safety feature to avoid double-counting particles. Each track has a unique ID number, and I add the IDs to C++ data structures called `set`, which can only contain unique entries. The number of unique particles is thus the size of the `set`.

You can experiment with the existing code, and look up the Geant4 classes like `G4Track` online, to retrieve more event information. My code makes a little text summary of each event, displayed on the command line, after the event is finished. It might be helpful to start using the visualisation and command line interface again, if you removed them.

Count the number of positrons produced by photon conversion in an event. [1 mark]

Remember that we can retrieve the energy deposited in a given step. We can use this approach to examine particle energies in the event action. Conversely, all the information that we have available in the event action can also be accessed in the detector class `src/EnergyCounter.cpp`, through the `G4Step` object. So, we can modify our detectors to record separate information when struck by a particular kind of particle, even if that doesn't make physical sense.

Calculate the fraction of the total *detected* energy in an event that was deposited by electrons. [1 mark]

Use any truth information that you like to produce your own visualisation of an event. For example, in the lecture I showed a network graph of which particles created what, made using PyGraphViz. [0 marks, but best answer gets a Mars bar or something. Optional, submit by email before Feb 14th.]