Monte Carlo Simulation Of Background Radiation in Jefferson National Laboratory Hall C for Experiment E12-11-009 (G_{EN})

Daniel Wilbern
Dr. William Tireman

April 24, 2015
Northern Michigan University
CS480 – Senior Project in Computer Science

Table Of Contents

Overview

EXPERIMENT BACKGROUND:

At Jefferson National Laboratory in Newport News, VA, experiment E12-11-009, informally known as G_{EN} or C-G_{EN}, is currently scheduled to take place in Experiment Hall C in 2020. The electric form factor of the neutron, G^n_E, is a fundamental quantity that is related to the spatial distribution of quarks in the neutron. Its dependence on Q^2 reflects the distribution of charge in the neutron. This experiment aims to measure G^n_E through the ratio of two scattering asymmetries associated with the positive and negative precessions of the neutron polarization vector. JLab experiment E93-038 previously measured this quantity for Q^2 values of 0.45, 1.13, and 1.45 $(GeV/c)^2$; E12-11-009 will extend these measurements to Q^2 = 3.95, 5.22, and 6.88 $(GeV/c)^2$.[1] Fig. 1 below shows data from E93-038 compared to data from other G^n_E measurements made in this range of Q^2.

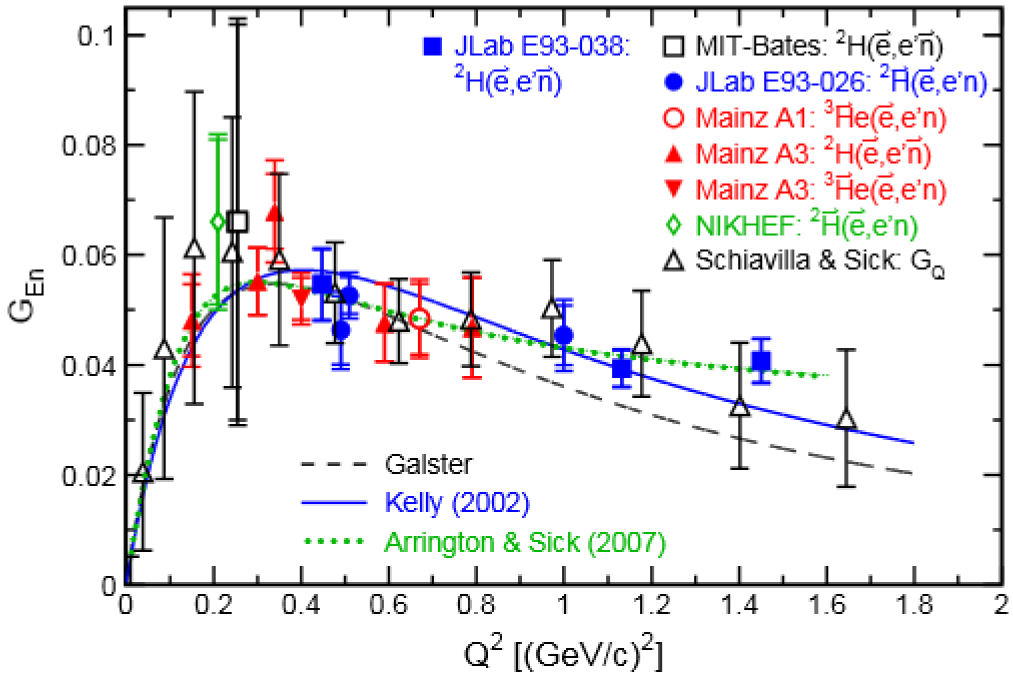

Figure 1: G_{EN} measurements made at a lower energy range than E12-11-009's proposed energy range. (from E12-11-009 proposal)

The G_{EN} experiment will take place in Jefferson Lab's Experimental Hall C. Fig. 2 provides an overview of Hall C. Each object of interest to this simulation program and its function will be discussed briefly.

Hall C is concrete cylinder with a radius of about 30 meters with a concrete dome roof partially buried in the ground at JLab. JLab's accelerator is capable of delivering a beam of electrons of desired spin-polarization characteristics with energy up to about 11 GeV to Hall C in an 80 μA beam with a diameter less than that of a human hair. A long sealed aluminum pipe known as the beam-line encloses the electrons and keeps them in a vacuum. Near the middle of the hall is the scattering chamber with the target inside. The half of the beam-line extending from the point where electrons enter the hall to the scattering chamber is known as the upstream beam-line. The half extending from the scattering chamber to the beam-dump is known as the downstream beam-line. The upstream beam-line is a cylindrical pipe with constant diameter, while the diameter of the

1 B.D. Anderson et. al. (2010). *The Neutron Electric Form Factor at Q^2 up to 7 $(GeV/c)^2$ from the Reaction* $^2H(\vec{e},e'\vec{n})^1H$ *via Recoil Polarimetry*. Retrieved from https://wiki.jlab.org/E12-11-009/

downstream beam-line increases with distance from the scattering chamber. The hole in the wall in Fig. 2 is where the downstream beam-line connects to the beam-dump. The beam-dump's purpose is to stop electrons that did not scatter in the target and dissipate their energy to minimize the amount of background radiation in the hall.

The scattering chamber is a cylindrical aluminum fixture about 7 meters from the center of the hall. The scattering chamber houses the target, which for C-G_{EN} is an aluminum can with diameter 4 cm and length 20 cm filled with liquid deuterium (^{2}H). Inside the scattering chamber, electrons in the electron beam interact with deuterons inside the target can and scatter. Many electrons scatter at shallow angles and continue down the beam-line to the beam-dump. Some interactions result in large angle scattering of an electron and other particles such as neutrons or pions ejected out of the target. The interaction C-G_{EN} is concerned with is $n(\vec{e}, e'\vec{n})$, or a collision of a spin-polarized electron (from the electron beam) and an unpolarized neutron from the deuterium target resulting in an electron and a spin-polarized neutron. The scattered electron is detected by the SHMS (Super High Resolution Spectrometer) or the HMS (High Resolution Spectrometer), and the polarized neutron is detected by NPOL.

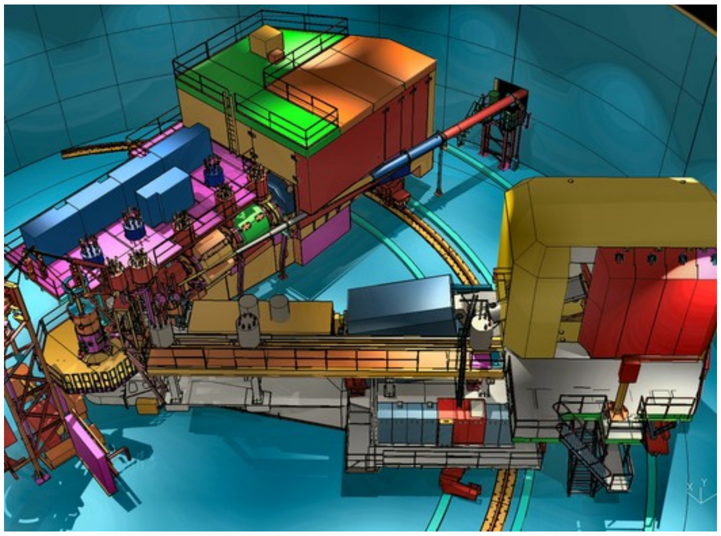
Figure 2: A model of Hall C's HMS (in the foreground) and SHMS (in the background). The scattering chamber is the small vertical cylinder on the left. The downstream beamline extends from the scattering chamber to the hole in the wall, which leads to the beam dump. (from https://www.jlab.org/Hall-C/)

NPOL is C-G_{EN}'s recoil polarimeter. Its proposed design is pictured in Fig. 3. The SHMS and HMS are on rails and can be rotated around the scattering chamber. During the experiment NPOL will be located roughly where the HMS is in Fig. 2, and the HMS will be moved to be opposite the SHMS with respect to the scattering chamber. Each detector is a plastic scintillator detector fitted with PMTs on either end capable of outputting voltage proportional to the amount of energy deposited by ionizing radiation. Spin-polarized neutrons originating from interactions of the electron beam with the deuterium target will strike the front detectors of NPOL (magenta in Fig. 3). The neutrons have about a 10% probability of interacting with a proton (hydrogen nucleus) in the plastic of each front detector, resulting in the proton scattering out of the detector and into one of the detectors on the top or bottom (blue in Fig. 3). Which direction and therefore which set of detectors the proton scatters into depends on the spin-polarization of the neutron. The data collected by this polarimeter will be used in the calculations for G^n_E.

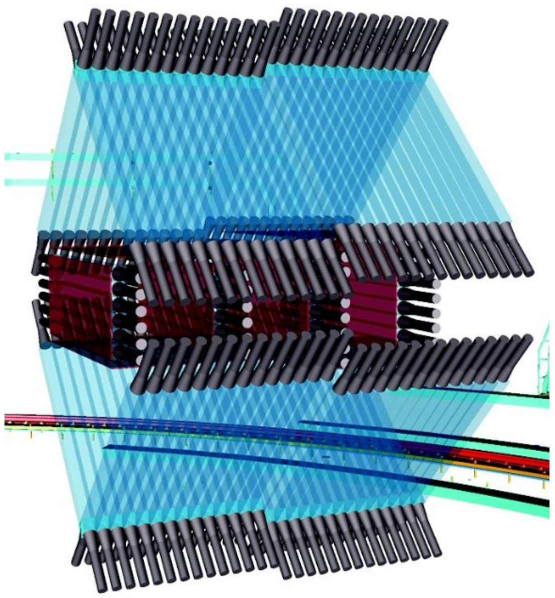
Figure 3: NPOL's proposed design (Geant4 Simulation for C-GeN, W. Tireman)

SIMULATION PROGRAM INTRODUCTION:

There is much that needs to be done before C-G_{EN} gets beam time in 2020. One thing on the list is this computer simulation to produce information about the effects of background radiation in the experiment hall on NPOL. The scintillator detectors are triggered by any ionizing radiation that strikes them, not only the scattered protons relevant to the experiment. Wherever possible, measures should be taken to reduce the number of false strikes. These measures include adding additional shielding around the detectors or perhaps repositioning troublesome equipment. But even where these measures are not possible, a simulation of background radiation can be useful in making decisions about which sets of data are "good" data and which sets are tainted by high levels of noise and should be disregarded.

This simulation program is not and was never intended to be a precise analysis of every variable that could possibly contribute to background radiation. It is intended to be a tool that can be used to draw quick conclusions about components of the experiment hall suspected to be significant contributors to the background noise in NPOL. These conclusions can either be examined more carefully by taking time to improve the precision of the simulated components or through other, more sophisticated means.

The development of this simulation program consists of three tasks: building the simulated experiment hall, tracking particles as the simulation runs and storing relevant data about them, and getting the stored data into usable formats. Discussions of each of these tasks will coincide with discussions of the structure of the simulation program.

TECHNOLOGIES USED:

The simulation program was written in C++ using some libraries and toolkits designed for particle physics simulations.

The simulation program was built with a toolkit by the name of Geant4, which simulates the passage of particles through matter. Geant4 was developed by an experimental collaboration in the mid 1990's to be a successor to GEANT3, first released in 1982 for experiments at CERN and written in FORTRAN. Geant4 was designed as a rewrite of GEANT3 in C++ with modern object-oriented design. Currently, Geant4 is maintained by the international Geant4 collaboration. The current stable version of Geant4, the version this simulation uses, is version 10.0, released in December 2013.

ROOT is a set of C++ libraries designed for particle physics data analysis. ROOT was written by a collaboration at CERN in 2003 and has since replaced its old FORTRAN analysis libraries. The output of the Geant4 simulation program is a file in a format understood by ROOT, and as such ROOT is what will be used to analyze the output of the simulation.

Implementation Details

SIMULATION PROGRAM STRUCTURE:

The simulation program's class definitions each belong to one of these categories: geometry definition, user action, or utility.

The geometry definition classes each describe the geometry of an object in the simulated experiment hall. The following classes belong to this category: NpolBDump, NpolBeamLineDown, NpolBeamLineUpper, NpolDipole1, NpolDipole2, NpolHallShell, NpolHBender, NpolPolarimeter, NpolScatteringChamber, NpolShieldHut, and NpolWorld. These are abstract factory classes that describe the construction and placement of the objects in the simulated experiment hall.

The user action classes are subclasses of Geant4's abstract user action classes that Geant4's kernel calls upon whenever the action happens. For example, NpolSteppingAction defines what happens whenever a step happens. The classes in this category are: NpolActionInitialization, NpolDetectorConstruction, NpolEventAction, NpolPrimaryGeneratorAction, NpolRunAction, and NpolSteppingAction.

The classes NpolAnalysisManager, NpolMaterials, and NpolDetectorFactory provide consistent and easy-to-use interfaces to things such as the materials table and Geant4 analysis manager.

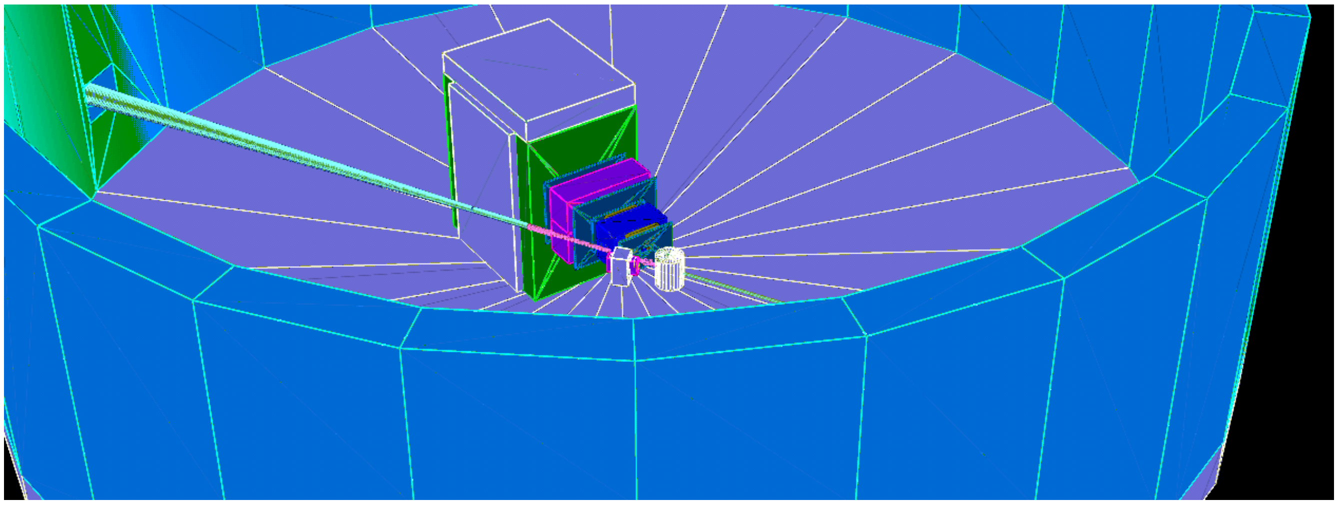
Figure 4: The simulated experiment hall. White vertical cylinder: scattering chamber with target inside. White box: SHMS's horizontal bender magnet. Blue and pink boxes: dipole magnets. Large box: shield hut with NPOL inside. The beamline extends from one end of the hall to the other, intersecting the scattering chamber. The hole in the wall on the left leads to the beam dump.

SIMULATED HALL:

Fig. 4 shows the simulated experiment hall, much more empty than the real experiment hall. The pink section of the beamline immediately downstream of the scattering chamber was modeled in more detail than the rest of the beamline. Fig. 5 shows the simulated NPOL inside the shield hut.

The only objects necessary in the simulated hall are NPOL, objects near and immediately downstream of the scattering chamber, and objects that stand between NPOL and the scattering chamber. One of the main background radiation concerns are particles interacting with large metal fixtures immediately downstream of the scattering chamber near the beamline. The other main concern the experiment collaboration has at the moment is stray particles getting caught in the dipole magnets' fringe fields. Objects far from the scattering chamber and far from NPOL, especially objects upstream of the beamline (like the HMS), are not expected to contribute to the background noise in NPOL in any significant way and it is therefore not necessary to include them in the simulation. The SHMS was not modeled, except for its horizontal bender magnet, because even though it is downstream of the scattering chamber, only the horizontal bender magnet is close enough to the scattering chamber to be expected to contribute at all to the background radiation detected by NPOL.

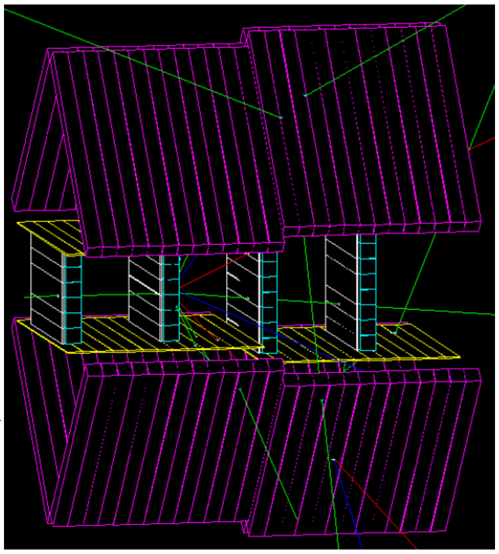
Figure 5: NPOL in the simulated experiment hall.

SIMULATED HALL IMPLEMENTATION:

Geant4 allows one to construct objects in a simulation (called volumes) if the attributes (shape, material, location) are specified.

A G4VSolid object describes the shape of a volume. Geant4 includes many useful 3D shapes that can be used to define detector geometries. These shapes include G4Box (rectangular prism), G4Tubs (hollow cylinder), G4Sphere, G4Torus, G4Cons (cone), G4EllipticalTube, and so on. A custom shape can be defined with G4TesselatedSolid if the vertexes of the shape are specified. A more complicated shape can be defined as a G4IntersectionSolid, G4SubtractionSolid, or G4UnionSolid, which results in a solid that is the intersection, subtraction, or union of two other solids.

One anomaly encountered while defining solids is that G4Tubs and G4Cons produce very low-polygon approximations to a hollow cylinder or a cone. See the blue hall wall in Fig. 3 for an example, this volume's shape is defined as a G4Tubs. For some volumes such as the deuterium target can (because it is very small) or the hall walls (which are far away from the scattering chamber), this will not likely cause any significant difference in the simulated background radiation in NPOL. However, the downstream beamline was defined as a tessellated solid with more polygons instead of a G4Cons because the low polygon approximation of the downstream beamline, which gets awfully close to NPOL, may cause significant inaccuracies in the background radiation detected by NPOL.

Once the shape of a volume is defined as a G4VSolid, A G4LogicalVolume can be defined which holds the volume's name, shape, visualization attributes, and some other things including the material of the volume (air, vacuum, aluminum, scintillator, etc.). The NpolMaterials class was defined to centralize the definitions of materials, which are often shared between many volumes. NpolMaterials is a singleton initialized with the lazy initialization pattern. The factory objects that construct the objects in the hall can get references to the materials they needs to define their volumes without needing to know specific information about the definitions of the materials (Were they materials built into Geant4? Were they custom-defined? How were they defined?).

Finally, once a volume's shape, material, and other attributes are defined, it must be placed at a location in the simulation. This is done by creating a G4VPhysicalVolume, which requires specification of the logical volume to be placed, its mother logical volume, and the coordinates (in the mother volume's coordinate system) to place the volume at. Geant4 requires one volume to be defined as the world volume which has no mother volume. Every other volume can have its mother specified as the world volume or another volume. Any overlaps in geometry between a mother volume and its daughter result in the overlapping space belonging to the daughter volume. Overlaps between any volumes except volumes that have a mother-daughter relationship are not allowed and cause errors in the simulation when a particle enters the overlapping region. Because of this, great care must be taken when placing volumes to ensure these errors do not arise. In particular, situations where there may be imprecision in the calculation of the boundaries of a solid (e.g. G4Tubs or G4Cons) are the most troublesome and many volumes defined by these shapes were placed with a few millimeters of “buffer” space between them and nearby volumes to ensure that they do not overlap.

Each object in the simulated experiment hall has an associated abstract factory class that knows the name of the object, how to construct the object, and how to place the object given a mother volume. Each factory class is a subclass of the abstract class NpolDetectorFactory, which requires definition of a GetName method and a Place method, which takes the mother volume as a parameter. Each factory is to construct its constituent G4LogicalVolumes in its constructor, and place them inside the provided mother volume in its Place method. The user action class NpolDetectorConstruction, which is called upon by the Geant4 kernel to construct volumes, acts as a builder that calls upon the abstract factories to construct and place their volumes. NpolDetectorConstruction creates and stores the factories in a set (from the standard template library), and then calls the Place method of each factory (with the world as the mother volume) in the set later when the Geant4 kernel calls for the detector geometry to be defined. The one factory that is not constructed and placed in this way is the NpolTarget factory, which is constructed in the constructor of NpolScatteringChamber and placed in NpolScatteringChamber's Place method, with the inside vacuum volume of the scattering chamber as the mother volume.

DATA STORAGE AND OUTPUT:

Geant4's analysis manager, accessed through the class G4AnalysisManager, is used to write the data collected during the running of the simulation to a file readable by the ROOT data analysis framework. The two types of data structures it writes are histograms and Ntuples. Currently the simulation program outputs one histogram for each detector in NPOL, filled with one entry per primary particle event that corresponds to the energy deposited in the detector for that event. The simulation also outputs an Ntuple, which is like a table, with some information about what occurred during each simulation step (well, not EVERY step but that will be discussed a little later).

The primary motivation for the NpolAnalysisManager class, a singleton that coordinates the setup of G4AnalysisManager's data structures and filling them with data, is that G4AnalysisManager is based on a system of integer ID numbers that correspond to histograms and Ntuple columns. This is not the most convenient system to work with, so NpolAnalysisManager keeps two data structures to deal with the integer IDs.

The first structure is a map (from the standard template library) that maps physical volume pointers to the integer histogram IDs. One of the things NPOL's factory class does is have one histogram created for every scintillator detector volume placed. It calls on NpolAnalyisManager with a pointer to the newly placed volume (along with some other things such as the axes scaling information), so that it can be mapped to its corresponding histogram ID (actually, a struct containing the ID and that other information is mapped to the volume pointers). If in the future the experiment collaboration wants energy deposition histograms for more volumes than just NPOL's detectors, all that has to be done is have the object's factory class pass the physical volume pointers and some other information G4AnalysisManager requires to NpolAnalysisManager, it will take care of the rest.

The other structure is a simple struct that stores one integer per quantity being stored in the Ntuple. It could have even just been an array, but a struct was used to give the elements meaningful names such as xPosColID. When an Ntuple column is created, the column's integer ID number is stored in the struct in the appropriate spot. When the ID number is needed later, it can be fetched out of the struct.

There is one more structure that must be kept in NpolAnalysisManager: a map (again, from the standard template library) that maps physical volume pointers to unique integer IDs. The reason for this map is that it is necessary to store something that can identify which detector the step occurred in for every Ntuple row. In order for the ROOT analysis routines to be able to associate these integer IDs with the volumes, a file is written out at the end of the simulation that lists the name of every volume with its integer ID for that run. This file can be read in a ROOT script or manually by a human to translate the ID numbers back to volumes. Note that these IDs are different from the histogram IDs because these IDs are assigned to every volume in the simulation and not just the ones that have energy deposition histograms.

PARTICLE TRACKING:

Geant4 simulates the passage of particles through matter. A user action class must be defined and made a subclass of G4VUserPrimaryGeneratorAction. This class will define the characteristics of the initial particle (particle type, initial position, initial momentum, etc.). Geant4 generates the specified primary particles and simulates its passage though the world. Each step of the simulation, Geant4 looks up the defined physics processes for the particle and the calculates the probability of each process happening in the step, then chooses a process at random to occur. This is what makes a Geant4 simulation a Monte Carlo simulation. A step is defined by the point location of the particle before the step, the location after the step, and the changes in the particle that occurred during the step. Some things that can happen during a step are ionization (loss of energy), creation of daughter particles (perhaps due to a collision), decay of the tracked particle, or perhaps nothing at all except a change in position. All of this information can be obtained by creating a user action class that is a subclass of G4UserSteppingAction. This class must override the UserSteppingAction method, which takes a G4Step object as an argument. The Geant4 kernel will call this method after every step with a G4Step object that holds the information about what occurred during the step.

After the primary particle and all of the daughter particles created by its interaction with the world, if any, have decayed or lost enough energy to fall below a defined threshold, Geant4 will terminate the tracking. An “event” refers to the tracking of one primary particle. The user may create a user action class as a subclass of G4UserEventAction and implement the BeginOfEventAction and EndOfEventAction methods to get information about what transpired over the course of a whole event. The Geant4 kernel will call these methods and pass a G4Event object at the beginning and end of each event.

All of the work in collecting and outputting data from the simulation is done in user action classes which subclass Geant4's abstract user action classes. A discussion of this simulation's implementation of the user action classes and how the data collected in them will explain the organization of the simulation's output file which is analyzed in a ROOT script.

USER ACTION IMPLEMENTATIONS:

The NpolPrimaryGeneratorAction user action class defines the initial characteristics of the primary particles. It's possible to do fancy things in this class like generate the particles in random locations in a distribution or generate them with random momenta in a specified range, but this simulation keeps it simple since the electron beam in the real Hall C is thinner than a human hair and the particles are all accelerated to have the same momentum.

NpolRunAction implements methods that are called at the beginning and end of each “run” of the simulation. This user action class is responsible for a seeing that a few things are done such as the initialization of the analysis manager and its data structures. The simulation would not output anything if it ran without the data structures being defined and initialized. This user action class must also see that the output files are written at the end of a run. This is a pretty boring user action class.

NpolSteppingAction is more interesting. This user action class implements a method that is called by Geant4 after every simulation step, and is passed a G4Step object with contains anything that could be desired to be known about the step and more. Data is extracted out of the G4Step and is used to fill an Ntuple row by passing the data to NpolAnalysisManager. The energy deposition histograms are not filled until the end of the event, but running totals of energy deposited in each detector are kept in NpolAnalysisManager's map of histogram information. If the pointer to the volume the step occurred in found in the map, then the energy deposited in the step is added to the running total. It should also be noted that all of this stuff is only done for the primary particle and its first few daughter particles. Daughter particles of daughter particles of daughter particles of daughter particles have so little energy compared to the primaries and immediate daughters that it isn't worth keeping track of them. In fact, particles that are more than a few generations down on the tree are killed because it's a waste of time to compute the physics processes for them and any children they might produce.

NpolEventAction implements methods called at the beginning and end of each primary particle event. In this simulation, the only things that are kept at an event scale are the NPOL energy deposition histograms. Running totals of the energies are kept by NpolSteppingAction during the event while steps are happening, so by the time NpolEventAction's EndOfEventAction method is called it can simply ask NpolAnalysisManager to fill the histogram of each detector in its map with the total energy deposited.

MULTI-THREADING:

In Decemer 2013, Geant4 version 10.0 was released. The main new feature was support for running simulations in multi-threaded mode. When Geant4 runs in multi-threaded mode, multiple events run simultaneously in separate threads (called “worker” threads) while one thread (called the “master” thread) holds on to data invariant among the worker threads such as the detector geometry and the physics processes. The number of desired worker threads can be specified to Geant4 at run time.

Not much work was needed to get the simulation running in multiple threads. The geometry definitions are done in the master thread before the worker threads are even created, so nothing needed to be changed there. The user action classes are all thread-local and as such hold thread-local variables, so nothing needed to be changed there either.

The only troublesome class that needed a fix was NpolAnalysisManager since it is a singleton that holds data accessed by multiple threads. All accesses to the instance variables of NpolAnalysisManager are protected by mutex locks, which incurs a performance hit because threads are forced to wait to access this data, but the performance hit is not too bad compared to the amount of computing time required to calculate and carry out the physics processes. It is not necessary to consider the possibility of multi-threading to accidentally create more than one instance of the singleton NpolAnalysisManager even though it is initialized with the lazy initialization pattern because it is always initialized during the program's setup before the worker threads are started. The GetInstance method, however, needs to protect access to the pointer to the singleton instance, however, because that variable is shared by all threads.

When the simulation is run with six worker threads, it runs noticeably faster. Not quite six times faster, but there is no question that the efficiency of the simulation is positively affected by the multi-threading. The electron beam delivered to Hall C delivers an 80 μA beam of electrons. Each electron has a charge of $1.6*10^{-19}$ C. This means that when the experiment runs in the real world, about 10^{14} or 100 trillion electrons will enter the hall every second. Currently it takes the simulation program several hours running on the physics department's 3.4 GHz hex-core machine to simulate 1-10 million events. Clearly the simulation runs several orders of magnitude slower than the experiment will in the real world, even with multi-threaded mode.

Future Development

OUTPUT AND ANALYSIS CONSIDERATIONS:

At the moment, the simulation program runs and produces output that can be analyzed with ROOT. A ROOT Ntuple is produced by the simulation which holds information such as position, momentum, particle type, and vertex energy for every step that occurred in the simulation. The data output by the simulation in the energy deposition histograms and the Ntuple can currently be viewed quickly with ROOT's Tbrowser. The data output by the simulation as it is at the moment is reasonable and can be explained physically.

The experiment collaboration does not yet have any specific requests for analysis of simulation data, but this simulation program was written to be easy to make changes to the elements the collaboration is likely to want to investigate, such as the addition or modification of objects in the simulated experiment hall, or the inclusion or exclusion of the tracking of certain particles.

Currently many objects in the simulated experiment hall are crude approximations to their real-life versions. They are made of simple shapes and a lack of documentation on certain parts at this early stage of the experiment resulted in guesswork in some size and spacing measurements. Some of these objects don't exist in real life yet or do not even have finalized designs, so their specifications may change as more things are worked out by the engineers or if funding changes plans. It was decided to keep it simple at this early phase when designing the objects as the the simulated representations can always be changed or made more accurate later if the changes are expected to contribute significantly to the background radiation.

As more analysis is done on the background radiation and the factors that contribute to it, the collaboration may want to change certain parameters in the simulation and investigate the results of these changes in the output. If an object needs to be added to the experiment hall, a factory class needs to be implemented and added to the builder NpolDetectorConstructor's list of things to build. If more event-level energy deposition histograms are needed, the factory class for the detector should have NpolAnalysisManager create the histogram by passing a pointer to the volume and the other required information (name, axes scaling, etc.). If another quantity needs to be output in the Ntuple, all that needs to be done is to add a column to the Ntuple (and an integer to the struct that holds the column IDs) and have NpolSteppingAction pass that quantity to NpolAnalysisManager's FillNtupleColumn method. These are all additions and changes that will likely be made to the simulation as more background radiation study is done, so this simulation program was designed to easily accommodate them.

ADDITIONS AND IMPROVEMENTS:

The simulation program is performing well at this stage of its development. When more specific requests for additions or changes are made, implementation of these changes can be made easily without disturbing the rest of the program. Some of these possible additions or improvements will be discussed here.

Geant4's electromagnetic physics processes can take into account a magnetic field. Currently there is no magnetic field in the simulated experiment hall, even though there will be magnetic fields present during the running of the experiment. To define a magnetic field, create a field object such as G4UniformMagField and use G4FieldManager to add it to the simulation. Currently we are waiting for field maps to be calculated for the real magnets to be used in the experiment. These field maps are files that describe the magnetic field at many points in space, which are read in and used to create the field object. When the magnetic fields are implemented into the simulation, decisions will have to be made about how accurately Geant4 will calculate the particles' propagation through them. This is done by shortening the distance Geant4 chooses for steps' lengths, which will slow down the simulation quite a bit if this is overdone.

It may also be possible to speed up the simulation a little bit by coming up with schemes to minimize the amount of time the worker threads spend in code sections protected by mutex locks in NpolAnalysisManager. One way this might be possible is to create multiple instances of NpolAnalysisManager, one per thread, instead of making it a singleton. This would incur a cost in required memory because each thread would hold its own copy of the same data (NpolAnalysisManager's instance variables), but this would pay off in speed because threads would no longer have to wait for mutexes to come unlocked while other threads are accessing NpolAnalysisManager's instance variables. The majority of CPU time is spent by Geant4 calculating the physics processes, so this improvement would only have a slight effect on the speed of the simulation's execution. For this reason, any improvements made solely to improve the running speed of the simulation, like this improvement, have a low priory compared to other things.

It will also eventually be necessary to write analysis scripts using the ROOT data analysis libraries to analyze the Ntuple and energy deposition histograms and produce formatted output that demonstrate the results of the simulation to the physicists studying the background radiation in the hall.

CONCLUSION:

This interdisciplinary project provided me with an excellent opportunity to demonstrate my skills as a programmer while learning about the process of setting up and running a particle physics experiment. Using object-oriented design patterns such as the singleton pattern and the abstract factory pattern with the builder pattern allowed resulted in a well-structured program with independent components that can accommodate the changes that are likely to be made to it.