Introduction to Particle Physics Methods - IPP Summer Student Report

Patrick (Zhi Gu) Li[1,*]

[1] *Department of Physics, University of Alberta, Edmonton, AB, Canada T6G 2G7*

The main objective of this summer project was to familiarize myself with key methods used in the particle physics research community and obtain some hands-on experience with the procedures itself. In summary, the summer was largely an introduction to the object-orientated program ROOT, of which the purpose was originally intended for particle physics data analysis. By using ROOT, I was able to implement many advanced plotting and data sorting algorithms for a given problem. Additionally, I worked under a Linux environment and went through the typical procedures of building programs – a practice that yields useful skills since the use of various Linux operating systems is very wide spread in the field of science. Over the summer, I was required to invoke my problem-solving skills on a series on subsequent tasks that I was assigned for, where I was given the freedom to accomplish them by any means necessary.

I. BUILDING GODDESS - A GEANT4 EXTENSION FOR MODELING OF OPTICAL DETECTOR COMPONENTS

A. Introduction

The Geant4 Objects for Detailed Detectors with Scintillators and SiPMs, or GODDeSS for short, is an extension of Geant4 which allows for easy modeling of optical detector components. Due to the extensiveness and flexibility of Geant4, specifically its geometric parameters, creating a simulation that models physical processes and sensitive detector components would require tedious calculations from the user's end. With increase in complexity of the simulation space, these calculations can become rather laborious and is one of the main issues that GODDeSS attempts to solve. GODDeSS essentially limits the complexity and reduces the flexibility of the Geant4 program to specialize in simulating optical detector components, thus minimizing the required efforts, reduces user errors, and lowering the necessary skill levels required for a Geant4 simulation. [1]

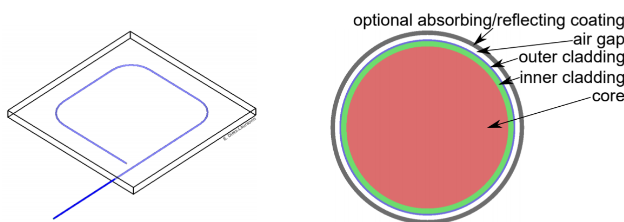

FIG. 1: Two examples of fairly straightforward optical components setup that would be considered "complex" within the Geant4 simulation space, however under the framework of GODDeSS, this simulation can be setup with roughly 8 lines of code. [1]

*Electronic address: `zhigu@ualberta.ca`

Our objective of this task is to build and compile the GODDeSS extension program and its dependant programs. Our original plan was to study and simulate the the scintillator tile detectors for muon detection that are proposed to be installed over ATLAS. Although this plan fell through, the notes I kept from my successful attempt at building GODDeSS still holds merit as this information is fairly scarce with minimal online documentations to provide any guidance.

B. Building Process

The official GODDeSS repository and its installation instructions is publicly available, however the provided instructions can be rather vague and easily misinterpreted by someone who is not technologically literate [2].

With no explicit requirement of a certain operating system, I originally attempted to install GODDeSS on Windows 10. While all the dependencies installed flawlessly, the GODDeSS package ran into an unsolvable error between Visual Studio 2019's C++ Linker and Boost. Following many failed attempts at solving this issue, I resorted to using a distribution of Linux instead, which is what ultimately worked. The GODDeSS package was built successfully with Ubuntu 16.04.6 LTS with gcc 5.4.0, and it should work with most versions of Ubuntu and even most Linux distributions with gcc 5.4.0 or higher. Ubuntu and gcc versions can be verified with the following commands:

```
$ lsb_release -a
$ gcc --version
```

The prerequisite programs for GODDeSS include: Geant4, Boost version 1.59, CMake, make, and zlib, and while it is not explicitly stated, a version of OpenGL and X11 must installed. Installing QT is also highly recommended.

Prior to installing Geant4, it is recommended to first installing OpenGL, X11 and QT since we wish to build GEANT4 with OpenGL, X11 and QT support. Additionally, a version of make and CMake must also be

present during the installing of Geant4. I used CMake version 3.17.3 which was **not** obtained from apt-get (since that version is too old to be used), but from CMake's official repository. The installation instructions can be publicly found at [3] (Note: *make* must be installed beforehand using apt-get). The instructions provided for CMake is rather straightforward, and I found zero ambiguities while following it.

While GODDeSS supports many versions of Geant4, I found that the combination of Geant4 version 10.4 Patch 4 with GODDeSS version 4.3 worked successfully. All previous versions and releases of Geant4 can be found publicly at [4], and the installation instructions at [5]. The installation instructions are originally written for Geant4 version 10.3.1, but I found that it worked flawlessly with our version of Geant4 10.4.4. The installation instructions are well written so I will not be going into too much detail, except for the fact that when you reach the build configuration stage, it is important to turn on the flags: install data, QT interface, and X11 OpenGL drivers. This can be done at the build configuration stage with the following commands:

```
$ cmake -DGEANT4_INSTALL_DATA=ON .
$ cmake -DGEANT4_USE_OPENGL_X11=ON .
$ cmake -DGEANT4_USE_QT=ON .
```

Keep in mind that, if one wishes to change the build configurations of Geant4 after the program have already been built, one can easily do so by changing one's terminal directory to the Geant4 build folder and execute build configuration commands similar to those listed above. After one has set their desired build configurations, simply execute (Note that sometimes with "$ make install", one must have "sudo" in front to grant administrator privileges):

```
$ make -j
$ make install
```

After, one have completed the installation steps, one can test if Geant4 is functioning properly using ExampleB1 shown in the instructions [5]. One should see a pop-up visual window with multiple objects in the simulation space. If this window does not pop up, then it is highly possible that OpenGL X11 has not been configured correctly. Also if one finds that the only way to pan the view is through terminal commands and not through a mouse interface, then QT4 has not been configured correctly.

Our next order of operations is to install Boost. As far as I know, GODDeSS does require Boost libraries to be built, since it utilizes the Boost.Regex library. The GODDeSS wiki suggests Boost version 1.59, which can be publicly found on Boost's official repository [6]. Originally, I built Boost in a desktop folder, however I ran into several errors with GODDeSS and CMake failing to locate Boost, so instead, I build boost in the suggested default location "/usr/local". However, one might run into permission issues when building programs in this location so when following the official instructions shown from Boost's website, one should make sure to change the permissions of the folders when added to "/usr/local" such that they are writable. The build process for Boost is well documented and there are no specific configuration options needed for GODDeSS.

Lastly, with all the GODDeSS dependencies installed, the final step is to build GODDeSS itself. Prior to downloading from the GODDeSS repository [2], make sure Git Large File Storage is installed, otherwise, the repository would not download properly onto the computer. The instructions for installing the GODDeSS example simulation found on the GODDeSS wiki worked mostly, but a few problems arose on the last step when executing the command: "$ make -j install". Depending on the administrative permissions, one might need to add "sudo" before the *make* command. Additionally, an error (specifically "Error 2") might occur following the execution of the *make* command. Taking a look at the error messages shows that the "isnan" function or other functions are undefined/declared. This issue is due to the difference in versions of C++ between of which GODDeSS was written in and the user's version. This issue can be solved easily by locating the line of code in each file where this error occurred and add "std::" before any functions that can result in the declaration error. Additionally, another possible error could occur when trying to run the example simulation with default settings. Following the "./RunSimulation" command, an error regarding a missing file "TwinTilerReflector_Aluminum" will pop up. This is because this file was not included in GODDeSS version 4.3, however it is included in GODDeSS version 2.0 and one can overcome this problem by copy and pasting it into version 4.3. With everything done correctly, a simulation window should open with a interactive GUI showing the GODDeSS simulation space.

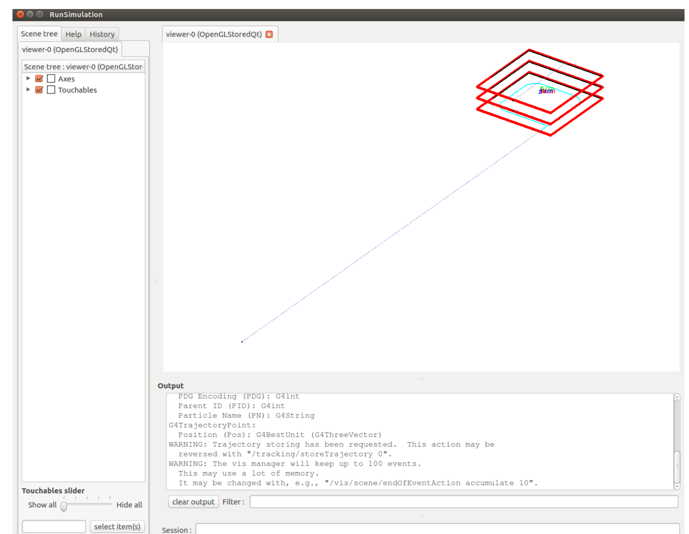

FIG. 2: The example GODDeSS simulation window with QT interface

While it can seem trivial to many individuals familiar with C++ and Linux systems, for someone who was completely inexperienced with Linux and the process of building applications, building GODDeSS successfully is not an easy accomplishment. With very little documentation on this process, using trial and error of different combinations of GODDeSS versions, Geant4 versions, operating systems, and build configurations can be very frustrating and time consuming. However, with the aid of these notes, I hope it could bring simplification to others who are also trying to build GODDeSS.

II. ROOT TASKS

A. Preface

The data taken in experimental particle physics can be overwhelming and abundant. With experiments like ATLAS at the Large Hadron Collider (LHC) having the potential to export terabytes of data or beyond within just a few seconds, these scenarios establish the importance of data analysis and the enormous role it plays in determining accurate results of these large scale experiments. ROOT was originally designed for the purpose of particle physics data analysis and in the follow sections, I will be completing various tasks to familiarize myself with important particle physics data method and ROOT, and implement various self-written algorithms to accomplish the given tasks.

B. Histograms

The first task I was given was to construct simple two-dimensional histograms. Histograms play an important role in particle data analysis as it simplifies the data into bins and visualizes the relevant physics. It graphically summarizes the distribution and variation of broad data sets which are fundamental to Monte Carlo (MC) methods, a popular approach used in particle physics.

In this task, I was given a set of detector data files where my objective was to plot them on a two-dimensional histogram and observe the locations where particles triggered the detector. Additionally, knowing the actual number of events, we can easily compute the efficiency of these detectors. The code written by me to complete this task can be found publicly at [7]. This algorithm is rather simple, as the main purpose of this task was an introduction to ROOT and its object-oriented work flow.

The algorithm first scans a specific folder containing all the comma-separated values (CSV) data files, and parses them through ROOT. For each file, its data is stored into vectors which are then used for further calculations and plotting. Again, this particular example is not meant to be hard, but a step towards familiarity in commonly used basic operations. Essentially, the code plots a 2-D histogram with the information given by files and calculates the detector's efficiency. The efficiency is calculated by counting the number of total entries found in our data file and comparing it with the number of actual events that took place. This process is then cycled until ROOT has parsed through all the files inside the set folder. Figure 3 is an example of a plot that would be produced by the macro. These example results are obtained from a simulation of a Nuclear Track Detector (NTD) for a monopole with mass 100 GeV, magnetic charge of 1 g_D.

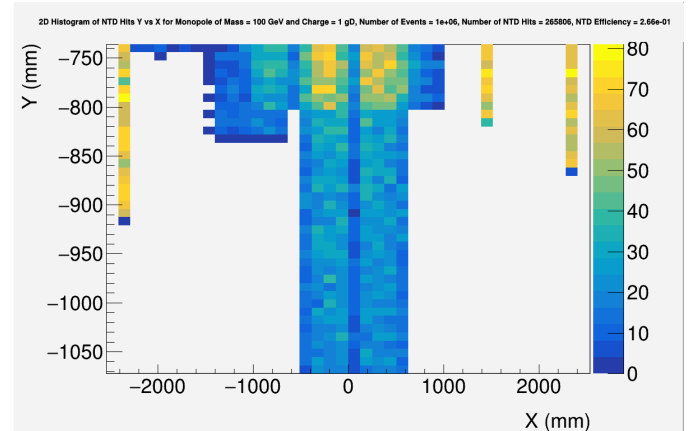

FIG. 3: 2D Histogram of the position of events for a Nuclear Track Detector. The efficiency is calculated by comparing events detected and the total events.

C. Mass Limits

In particle physics (and many other areas of physics), when it comes to searching for something new or unknown, we often set limits of various parameters based on what we *are* able to observe. For the particle physics case, in the attempts to predicting a new hypothetical particle, we often set limits on its mass. For instance, if we have not observed a certain predicted particle from consistently repeated collider experiments, we can then assume that the mass of the hypothetical particle that *would have* been produced from these repeated collider experiments is invalid, thus setting a limit on the particle's mass. Setting limits on various parameters are of an extreme importance to particle physicists as it allows us to narrow down our search in terms of where to look for these hypothetical particles.

This next task involves plotting and finding intersection points of the curves. Mass (in GeV) is plotted on the X-axis, while the cross-section is on the Y-axis. There are two curves on this plot, one being the cross-section limit versus mass, and the other being the scaled cross-section versus mass. The point where these two curves intersect will determine the value of our mass limit. With our given data, not every cross-section limit and scaled cross-

section pair will intersect on the graph, which means that those pairs will provide no use for us. Our objective for this task now becomes not only data plotting and calculations, but also data sorting.

This code will parse the CSV file data into ROOT and make plot(s) with 3-6 curve pairs on each plot. The data pairs with no intersections will not be plotted, and the pairs that do have intersections will have their intersection point (mass limit) exported into another CSV file. The majority of this code is dedicated to data sorting and intersection finding. There are already other existing intersection detection algorithms available, but I chose to write my own as it is a good practice to test my problem solving skills. This algorithm behaves as follows for any sets of data for two curves:

1. Assuming both curves are one to one functions, sort both curves by ascending x-values.

2. Starting from the first point of curve 1, using the (i)-th and (i+1)-th point, interpolate a linear curve using the point-slope formula. Do the same with the (j)-th and (j+1)-th point on curve 2, also starting from the first point.

3. Now with the linear equations of both curves, find the x-value of where these linearly interpolated curves intersect.

4. If this intersection x-value satisfy both conditions:

$$x_i \leq x < x_{i+1} \quad \text{and} \quad x_j \leq x < x <_{j+1} \tag{1}$$

   then it is counted as an intersection, and the corresponding y-value is calculated.

5. The x-y intersection value pair is then exported into a CSV file for the corresponding cross-section limit and scaled cross-section pair. If an intersection does not exist, this step is skipped.

6. Repeat step 1 to 5 using $i = i + 1$ and once all the points on curve 1 is cycled through, use $j = j+1$ for curve 2 and reset i back to the first point of curve 1. The intersection algorithm is complete once curve 2 has been cycled through.

Given our data structure, only one possible intersection can exist, so it could be quite beneficial to stop the algorithm after the detection of one intersection, however, I left it in as a precautionary measure since the accidental results of two intersections is obviously wrong and easy to detect.

The intersection algorithm made in-house was the highlight of this code, and the rest consists of tedious data sorting procedures. The particle we are plotting for has 3 parameters: spin, magnetic charge, and electrical charge. The plotting algorithm must sort these curves by ascending order and ascending priorities respectively (e.g spin 0, magnetic charge 0, electrical charge 0 - spin 0, magnetic charge 0, electric charge 1). Additionally, the code also makes sure that the plots are not too cluttered by keeping a good balance of 3-6 curve pairs per plot. The code and its instructions can be found publicly at [8]. Figure 4 is an example plot constructed using this code, and the particle in question here is a ”Dyon”.

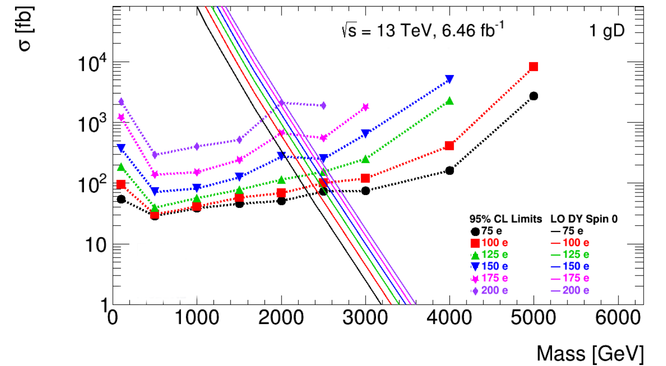

FIG. 4: Example of a mass limit plot. The dotted lines represent the data for cross-section limits, and the solid lines are scaled cross-sections. The point where each respective curves intersect is their mass limits.

D. Concave Hull

This next task I was handed with is less general in the applications of particle physics. Regardless, it still targets my problem-solving abilities and allows me to become more familiar with ROOT, which was the main focus of these tasks. Our next task here uses data calculated from a theory regarding the light inflaton field and the chaotic inflationary model [9]. While I do not fully understand the theory, the objective was clear. The calculated data consists of numerous scatter points, and the objective was to sort the data in a way such that only a smooth concave hull remained on the plot. If the theory is true, this hull would represent 95% or more confidence level that one is certain to see a particle in the detector. The calculated data plotted on a log-log plot takes on the geometric form shown in Figure 5.

I found this task to be rather challenging since all the existing concave hull algorithms are quite complicated with multiple hard-to-determine wrapping parameters. Convex hulls, however are much simpler and can be applied to this scenario under the right conditions. Another challenge arose due to the geometric structure of our data set. It looks quite simple and easy to grasp when plotted on a logarithmic axis, but linearly, the relative locations of these points were difficult to determine. Due to these unique constraints, I struggled to write a general algorithm that would allow me to extrapolate concave hulls for any given data set.

To tackle this problem, I divided up our data set into three regions. On the left and right regions, before and after the dip at 1 GeV, I applied a simple convex hull

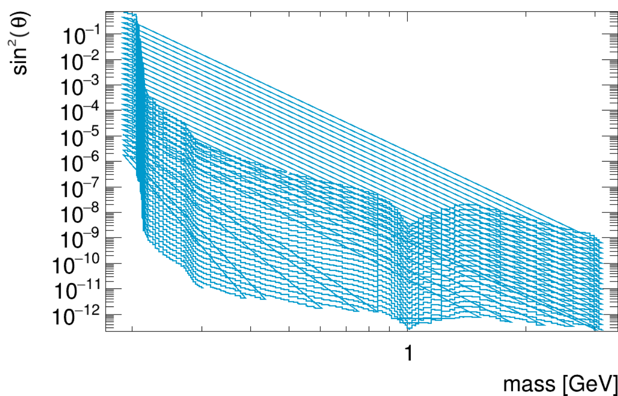

FIG. 5: Data calculated from a theoretical model found in [9]. Our objective here is the plot a smooth concave hull derived of this data set.

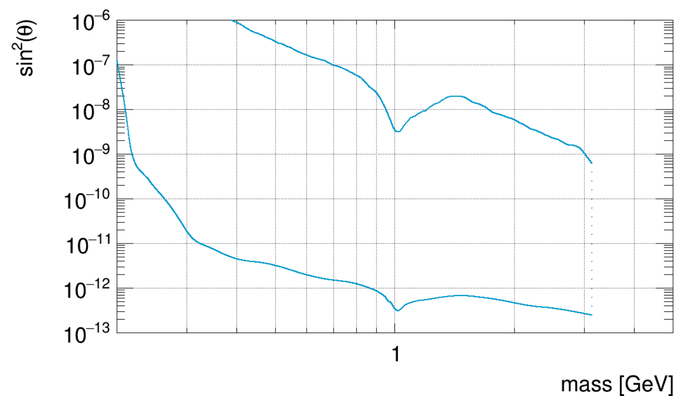

FIG. 6: The concave hull is obtained from the data calculated in Figure 5. The hull was then made more smooth using multiple cycles of averaging and interpolation until it became acceptable. For the y-range, we were only interested up to a value of 1e-6.

algorithm called the "Gift Wrapping Algorithm". This algorithm essentially starts on the most left (right) and upper point and scans in a (counter) clockwise manner. The first point it detects after scanning some angle θ now becomes the next pivoting point, and the scan now starts at θ. This is done until we return back to the first pivoting point, which by then θ should have spanned 2π. The reason why this problem is split into three regions is due to the fact that most, if not all convex hull algorithms would skip right over the dip at 1 GeV. For the middle region at 1 GeV, I had no choice but to manually sort out the upper and lower points. In the end, I ended up with with a nice hull where regions 1 and 3 was obtain using Gift Wrapping, and the middle section was done manually.

After obtaining the hull points, it was time to smooth out the curve. Due to the abrupt ends on the bottom few curves of Figure 5 (where the lines connect the last point of each curve back to the top), they create very rough edges on the contour which we do not want. First I took the weighted average (since not all hull points are spaced the same) of each point with their surrounding points. I tried taking various numbers of points to average each point with until I found a parameter that worked the best; enough to blur the edges while not losing too much information from the original data. Afterwards, I interpolated the graph using ROOT's own built-in Akima Spline algorithm which creates some artificial bumps and divots. This averaging and interpolating processes was repeated a couple timed until it was smooth enough to become acceptable. Lastly I repopulated the points by finishing off with a linear interpolation of the entire hull. Figure 6 shows the final results after multiple averaging and interpolation cycles.

E. Calculating Detector Efficiency

In a typical LHC experiment (or any applicable experiments), it is crucial to understand the efficiency of our detectors. Not only does it illustrate the performance of our detectors but also allows us to extrapolate what is happening in real life based on our detector readings. While we did some surface level efficiency calculation in task B: Histograms, this method will be a lot more involved. This final task will be done in Python since I struggled to make ROOT's 3D visualization to work properly.

While most detector efficiencies are computed using Monte Carlo (MC) methods, there are times where MC simulations are not adequate. In some scenarios, the efficiency is so low such that it would literally require more than hundred billions of events or so just to estimate the efficiency sufficiently. In our case, we will be calculating our efficiency using analytical expressions and numerical integration. The efficiency of our detector is given by the analytical expression:

$$\frac{d\epsilon}{dV} = \frac{1}{2\pi r^2 c\tau} \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{1}{2}(\frac{\eta-\mu}{\sigma})^2) \int_0^1 \exp(\frac{-r}{c\tau\beta\gamma}) d\beta \tag{2}$$

Where ϵ is the efficiency of the detector, and if we assume the particle travels approximately at the speed of light then $c\tau$ is the average distance the particle would travel before decaying. This equation is written using pseudorapidity as spatial coordinate, so this means that r is the radius away from the beam line, η is the pseudorapidity - an angle describing the particle's position relative to the beam axis, and ϕ is the azimuthal angle which in our case is disregarded due to azimuthal symmetry of the beam line. μ and σ are the mean and variance of the normal distribution respectively. The example detector we will be integrating our efficiency over is given by the following 8

vertices in Cartesian coordinates: [3.27,3,-52.83],[3.27,-2,-52.83],[4.798,3,-71.91],[4.798,-2,-71.91],[12.24,3,-33.63],[12.24,-2,-33.63],[21.13,3,-37.4],[21.13,-2,-37.4].

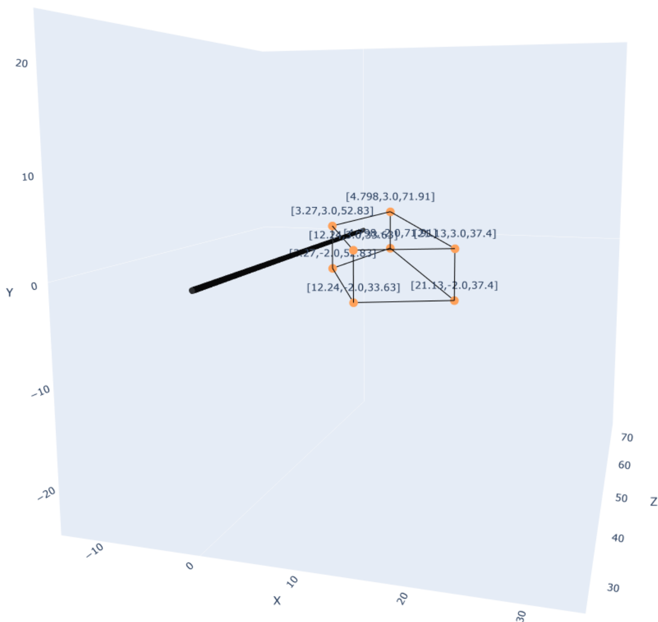

FIG. 7: 3D visualization of the detector volume with respect to the beam line.

Since our detector coordinates are given in Cartesian, we must convert it to spherical and pseudorapidity using the following relations:

$$r = \sqrt{x^2 + y^2 + z^2} \tag{3}$$

$$\theta = \mathbf{arccos}(\frac{z}{r}) \tag{4}$$

$$\eta = -\ln\left(\tan(\frac{\theta}{2})\right) \tag{5}$$

Since we are integrating this volume numerically, we can take on the approach of integrating in Cartesian and converting it into spherical coordinates using the above relationships as we integrate. Since we chose to integrate in Cartesian, we must calculate our integration bounds by finding the equations of planes responsible for each side of the surface.

To find a plane, for example the plane defined by: [12.24,3,33.63], [12.24, -2, 33.63], [3.27, 3, 52.83], [3.27, -2, 52.83] we must find two vectors that lay on the plane. Two obvious vectors that lay on the plane is:

$$\vec{v_1} =< 3.27-12.24, 3-3, 52.83-33.63 >=< -8.97, 0, 19.2 > \tag{6}$$

$$\vec{v_2} =< 12.24-12.24, 3-(-2), 33.63-33.63 >=< 0, 5, 0 > \tag{7}$$

Therefore a vector normal to the plane can be found by:

$$\vec{n} = \vec{v1} \times \vec{v2} =< -96, 0, -44.85 > \tag{8}$$

Thus our equation for this plane is simply:

$$96x + 44.85z = 2683.3455 \tag{9}$$

Later we will be integrating this surface with respect to z so this equation in terms of x is (we will call this x_2):

$$x_2 = \frac{2683.3455 - 44.85z}{96} \tag{10}$$

Similarly, we can find the equation of the planes for the rest of the surfaces:

$$x_3 = \frac{6700.0655 - 81.66z}{172.55} \tag{11}$$

$$x_4 = \frac{7.64z - 91.6632}{95.4} \tag{12}$$

$$y_2 = 3 \tag{13}$$

$$y_1 = -2 \tag{14}$$

Now with our planes calculated, we can compute the efficiency of our detector by integrating Eq 2 over our detector volume:

$$\epsilon = \int_{-2}^{3}\int_{33.63}^{37.4}\int_{x_2}^{x_1} \frac{d\epsilon}{dV} dxdzdy + \int_{-2}^{3}\int_{37.4}^{52.83}\int_{x_2}^{x_3} \frac{d\epsilon}{dV} dxdzdy + \int_{-2}^{3}\int_{52.83}^{71.91}\int_{x_4}^{x_3} \frac{d\epsilon}{dV} dxdzdy \tag{15}$$

III. CONCLUSIONS

Over the course of this summer I was able to familiarize myself with important practices used in experimental particle physics. I was able to accomplish a series of tasks that not only improved my problem solving skills but also allowed me to become familiar with the object-oriented program ROOT. ROOT is often seen as a standafrd program in the particle physics community, and understanding the program can be advantageous as it invites the opportunity for collaboration among other particle physics

members. Additionally, I was able to get some field experience in building programs under a Linux environment - a task that is done regularly in the scientific field. In conclusion, at the end of this summer, I learned multiple numerical algorithms, became familiar with Linux and the process of building programs, and above all, I got a new programming language under my belt.

IV. ACKNOWLEDGEMENTS

First and foremost, I would like to thank Ameir Shaa Bin Akber Ali, whom spend a great deal of time patiently answering all my questions and assisting me in my tasks. Additionally I would also like to extend my sincere gratitude to Dr. James Lewis Pinfold for supervising me on this summer research project and for helping me through all the application process of IPP Summer Student Fellowship and NSERC USRA. I would also like to thank Dr. Peggy White, Dr. Michael Roney, and Dr. Steven Robertson for offering me a position in this 2020 IPP fellowship program and facilitating the whole process.

[1] E. Dietz-Laursonn *et al.*, *GODDeSS: a Geant4 extension for easy modelling of optical detector components*, JINST 12 P04026 2017
[2] The GODDeSS GitHub Repository, https://git.rwth-aachen.de/3pia/forge/goddess-package
[3] CMake Installation Instructions, https://cmake.org/install/
[4] Geant4 download site, https://geant4.web.cern.ch/support/download_archive?page=1
[5] Instructions for installing Geant4.10.3.1, https://indico.cern.ch/event/679723/contributions/2792554/attachments/1559217/2453759/Geant4InstallationGuide.pdf
[6] Official Boost Repository, https://www.boost.org/users/history/
[7] 2D Histogram Plotting Algorithm, https://github.com/MustyHickory106/2D_MMT_NTD_Histogram
[8] Mass Limit Plotting ALgorithm, https://github.com/MustyHickory106/limit_plots
[9] F. Bezrukov, D. Gorbunov, *Light inflaton hunter's guide.* J. High Energ. Phys. 2010, 10 (2010).