Methods and Concepts in Experimental Particle Physics

Connor Carrico, Alexander Lowery, Alexander Mattern, Emily Samples, Alex Serati, and Brandon Speegle

College of Science, Virginia Tech
Professor Leo E. Piilonen
5 May, 2025

Abstract :

The modern experimentalist in particle physics relies on a toolbox of equations, theory, software, data processing strategies, programming languages, and physical instruments. It takes years of practice and study to understand and apply all of the tools in that toolbox, so well-structured education and documentation are essential to that end. This report highlights the progress made by our undergraduate research group throughout the spring 2025 semester in learning important lessons related to particle physics experimentation. Existing math, physics, and coding skills were refined, and followed by the introduction of new software and data analysis skills unique to physics research. The group successfully progressed through the stages of the project, culminating in the analysis of data previously theorized, visualized, generated, simulated, and reconstructed across the semester. This project was effective in supporting comprehension through each step of the process, and—perhaps more importantly—creating an understanding of the process as a whole.

1 Introduction

Our learning began with some basic particle physics concepts and mathematics, which we practiced in a short problem set. That practice was then applied to create visual simulations of particle interactions in Python, which helped us learn about Python, Visual Python, randomness, generalization of a problem, vectors, and spherical coordinates.

Once we had obtained a decent grasp on the physics concepts, visuals, and math, we moved on to creating data. This required us to gain a basic understanding of the Linux OS, and to connect to and utilize Tesla, a powerful remote computer owned by the physics department. We also learned a bit about modern data analysis and simulation software like Geant4, EVTGEN, ROOT, and basf2. We utilized that software, along with some pre-existing examples and code to generate our own simulated data on Tesla, which we could then copy to our own computers for later analysis.

The final step was to analyze our data in ROOT, which started with learning how to navigate in the browser, customize graphs, and understand units and data. More importantly, we moved on to creating best fit curves, learning what they mean, how to judge them, and how to modify them for a better fit.

The remainder of this report will delve into the specifics of each step we took this semester, highlighting the key insights we gained along the way.

2 Particle Interaction Exercises

To solve various mechanical problems within particle physics, various laws and mathematical tools can be used in order to solve them. These include, but are not limited to the following:

1. The Mass Invariant

$$M = \sqrt{E_{tot}^2 + p_{tot}^2}$$

2. Four Vectors

$$\vec{P} = (E, p_x, p_y, p_z)$$

3. Lorentz Transform Matrix

$$\begin{bmatrix} \gamma & 0 & 0 & \beta\gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta\gamma & 0 & 0 & \gamma \end{bmatrix}$$

By combining all of these tools along with the conservation of linear momentum, which is relativistically correct, five particle interaction problems were discussed and solved to determine either the mass of daughter particles, their momentum, or the direction of said momentum. An example of two particles of equal mass (P_1 and P_2) and differing momenta colliding is solved using four vectors as follows:

$$\vec{P_1} = \left(\sqrt{p_1^2 + m^2}, 0, 0, p_1\right) \tag{1}$$

$$\vec{P_2} = \left(\sqrt{p_2^2 + m^2}, 0, 0, -p_2\right) \tag{2}$$

$$\vec{P_f} = \left(\sqrt{p_f^2 + M^2}, 0, 0, p_f\right) \tag{3}$$

$$\vec{P_f} = \vec{P_1} + \vec{P_2} \tag{4}$$

Using the identity established in Eq. 4, we get the following:

$$\sqrt{(p_1 - p_2)^2 + M^2} = \sqrt{p_1^2 + m^2} + \sqrt{p_2^2 + m^2} \tag{5}$$

This expression can then be solved for M to get the following expression:

$$M = \sqrt{2m^2 + 2p_1p_2 + 2\sqrt{(p_1^2 + m^2)(p_2^2 + m^2)}} \tag{6}$$

In this example, the mass invariant and the four vector were used in order to determine the mass of a daughter particle generated due to a collision. In order to apply the Lorentz transform matrix, different frames of reference need to be used. Algebraically it follows the same process as the example problem, with the main difference being the application of the matrix:

$$\vec{P_i} = \vec{P_f}L \tag{7}$$

Where L is the Lorentz transform matrix, and P_i, P_f are four vectors.

3 Python Simulations

Following our calculations from the particle interactions questions, we used our results to create a visual demonstration of them. Using Python and the Visual Python package, our task revolved around simulating particle collisions and decays described in the original questions. To begin, we defined a class for the four momentum vector. This worked as a generalized set of parameters for each particle, and it allowed us to define the particles as spheres. This helped to create a number of characteristics for the particles, such as their mass, size, and color, for the sake of differentiating them.

We were able to represent the particle interaction for questions 1 and 2 simultaneously, since they both involved the collision of an electron and a positron. We began by creating a "while" loop, which updated particle positions with each frame of the animation, and inserted the generalized version of the physics in questions 1 and 2, ensuring it had a limit on the length of the visual demonstration. We can see in Figure 1 the written code for this section.

```
def problem1_2(particle1, particle2):
    particle1.sphere.visible = True
    particle2.sphere.visible = True
    time.sleep(1)
    while True:
        vp.rate(30)
        particle1.sphere.pos += particle1.velocity * particle1.dt
        particle2.sphere.pos += particle2.velocity * particle2.dt
        avg = (particle1.pos + particle2.pos)/2
        totalmom = particle1.momentum + particle2.momentum
        if (particle1.sphere.pos - particle2.sphere.pos).mag <= 0.5:
            particle1.sphere.visible = False
            particle2.sphere.visible = False
            newParticleMass = math.sqrt((particle1.energy + particle2.energy)**2 - (particle1.momentum + particle2.momentum).mag**2)
            newParticle = fourMomentum(newParticleMass, totalmom.x, totalmom.y, totalmom.z, avg.x, avg.y, avg.z, vp.color.purple)
            newParticle.sphere.visible = True

            i = 0
            while i <= 150:
                vp.rate(30)
                newParticle.sphere.pos = newParticle.sphere.pos + newParticle.velocity * newParticle.dt
                i += 1
            newParticle.sphere.visible = False
            print("New particle momentum:", newParticle.momentum.mag, 'GeV/c')
            print("New particle mass:", newParticleMass, 'GeV/c^2')
            print()
            break
    return None
```

Figure 1. A screenshot from Python for questions 1 and 2 of the particle interactions sheet.

Similar to questions 1 and 2, we easily combined questions 3 and 4 into a new definition function in Python. After plugging in the results from the questions, we defined a transition to polar coordinates. For this section, we wanted to reproduce the decays of the particles along many different axes with a uniform distribution. This required us to slightly modify our code, since we originally programmed for a "randomness" that unintentionally resulted in a larger distribution around the poles of the axes. After several derivations and drawings in spherical coordinates, we were able to tweak the "rand.uniform" function to account for a completely random distribution on all axes. Figure 2 demonstrates the written code for questions 3 and 4, including the adjusted "rand.uniform" function that changes the "Φ" axis distribution.

To finish setting up for the particle interaction simulations, we needed to define a scene in which we would observe our results. Using visual Python, we coded the "vp.canvas" function, which allowed a space for us to run our results.

Finally, we listed all of the values for each parameter, separating by question, in order to program the visual interactions. Figure 3 depicts these lines of code that told the visual Python how to operate, for it included the four momenta, the masses, the velocities, and other parameters.

This Python demonstration helped to better visualize our understanding of the particle interactions referenced in section 2. Not only was this a test to our programming abilities, but it also provided a more comprehensive learning experience with a "hands-on" element.

```
def problem3_4(parentMass, daughterMass1, daughterMass2):
    parent = fourMomentum(parentMass, 0, 0, 0, 0, 0, 0, vp.color.purple)
    parent.sphere.visible = True
    p = math.sqrt(((parentMass**2 + daughterMass1**2 - daughterMass2**2) / (2 * parentMass))**2 - daughterMass1**2)
    theta = rand.uniform(0, math.pi*2)
    phi = math.acos(1 - rand.uniform(0, 2))
    x = p*math.cos(theta)*math.sin(phi)
    y = p*math.sin(theta)*math.sin(phi)
    z = p*math.cos(phi)
    daughter1 = fourMomentum(daughterMass1, x, y, z, 0, 0, 0, vp.color.red)
    daughter2 = fourMomentum(daughterMass2, -1*x, -1*y, -1*z, 0, 0, 0, vp.color.blue)
    time.sleep(1)
    daughter1.sphere.visible = True
    daughter2.sphere.visible = True
    parent.sphere.visible = False
    i=0
    while i <= 150:
        vp.rate(30)
        i += 1
        daughter1.sphere.pos = daughter1.sphere.pos + daughter1.velocity * daughter1.dt
        daughter2.sphere.pos = daughter2.sphere.pos + daughter2.velocity * daughter2.dt
    daughter1.sphere.visible = False
    daughter2.sphere.visible = False
    print("The momentum of either daughter is:", daughter1.momentum.mag)
    print()
    return None
```

Figure 2. A screenshot from Python for questions 3 and 4 of the particle interactions sheet.

```
problem1_2(fourMomentum(5.11 * 10**(-4), 5, 0, 0, -10, 0, 0, vp.color.red), fourMomentum(5.11 * 10**(-4), -5, 0, 0, 10, 0, 0, vp.color.blue))
#problem 1
problem1_2(fourMomentum(5.11 * 10**(-4), 7, 0, 0, -10, 0, 0, vp.color.red), fourMomentum(5.11 * 10**(-4), -4, 0, 0, 10, 0, 0, vp.color.blue))
#problem 2
problem3_4(5.27966, 0.13957, 0.13597)
#problem 3
problem3_4(5.27966, 0.11343, 0)
#problem 4
```

Figure 3. A screenshot from Python for our established values of the individual interactions.

4 Navigating Linux and Tesla

Over the course of this semester, we downloaded and utilized PuTTY, an open-source terminal emulator, to access Tesla and copy files from Dr. Piilonen's directory for use in various softwares, and export what we needed into our own directories to open and view in ROOT. Each person downloaded the PuTTY software on their own, along with Linux, the operating system that Tesla uses, and were provided with individual Tesla accounts, which were made accessible through the software. Tesla runs in the Linux operating system, so the download was necessary in order to continue. The login passwords were customizable and the usernames were provided in the form 'username@tesla.phys.vt.edu'.

In Tesla, we learned a variety of commands for navigation. The 'cd' command was used to navigate between several directories. The first day we experimented with this, we used the command to navigate into several preexisting directories. Using the command 'cd /home/piilonen/' we navigated into Dr. Piilonen's directory, and by using the 'ls' command, we were able to generate a list of all items within this directory. Within the list, we could see what we could either 'cd' into or open as a file. Blue text indicated a directory in the list that could be 'cd'ed into, while green text indicated executable files, and red indicated compressed files (zipped files). As such, many '.py' and '.sh' files were listed in green many directories and subdirectories were listed in blue.

We also used Tesla to copy things into our own directories. To do this, it was first necessary to make our own individual directories using the 'mkdir xxx' command, where 'xxx' represents the name chosen for the individual research directories we generated. Once in our own directories, we could use the 'cp' or copy command to copy the necessary files into our own directories. An example of how we used this command was in our generation of output root files utilizing the Tesla 'minions'. After copying the basf2 scripts necessary into our directories, we used the 'qsub'

command to assign tasks to minions based on the basf files. The minions generated Root files, which were then exported to the laptop either using the 'scp' command on Tesla for those on a mac, or using a different software, WinSCP on Windows systems.

5 Software: Geant4, EVTGEN, & basf2

After acquainting ourselves with the Linux operating system and the Tesla computer, we could start using the various softwares available to us in order to handle large simulations, calculations, and datasets. EVTGEN is the software that generates collision/decay events based on known interactions and their probabilities. Geant4 is the software that takes those events and simulates the precipitating interactions in a simulated version of the Belle II detector, keeping track of every particle, atom, and material, and therefore noting what the detector will detect.

5.1 Application of Software

We started by learning about EVTGEN, first searching through a master decay file containing all known heavy-flavor physics, and finding the few decays that fell within the group of interactions we were looking for. We then copied those into our own decay file, but this was simply an exercise before really applying EVTGEN.

We were able to make use of EVTGEN by running EVTGEN files through a more complicated program: basf2. This is Belle II's software that handles the generation of simulation data and processing of that data. We created our own folder in Tesla, and used the following command to enter the Belle II software environment which would allow us to use basf2:

source ~piilonen/belle2/tools/b2setup release-06-00-03

We then copied two EVTGEN files and two basf2 Python scripts. We ran the basf2 scripts using instructions fed to the compute nodes ("minions") on Tesla to take the computing burden of forming those initial events into full simulated data, which was packaged in the form of a ROOT file. The data we created would be the basis for all of our later analysis in ROOT.

5.2 Details of Simulation Software

It is important also to note how basf2 works, and how Geant4 comes into this process. The Belle II software, basf2, follows a multi step process to turn EVTGEN decays/interactions into a large set of organized data. First, the initial EVTGEN decay files are used to generate collision events. Second, those events are simulated in Geant4 to record data of how this collision would play out within the Belle II detector, and record detector hits within the simulation. Finally, those simulated detector measurements are reconstructed into particles and their four-momenta. This full process results in the data that we see in ROOT.

6 Data Analysis in ROOT

Once we got the proper files from Tesla, we ran ROOT on our computers. Some of us had minor difficulties with this, as the process for Macs and Windows are different, and a few of us had issues giving our computers permission to run ROOT. We managed to get around most of the issues we had, and successfully run TBrowser. With TBrowser, we could view our data and try to map it with a line of best fit. We found various ways to represent our data, either as a 2D graph, or as a 3D Lego chart. TBrowser seems very useful and versatile, and with more practice we can see it as being practical for our data analysis.

The final stretch of our research this semester was learning data analysis tools on ROOT to get a line of best fit of the data we generated from basf2. The data we generated from basf2 were based off of simulation using probabilistic physics leading to different data sets for different users. This data was then put into ROOT files that we could analyze and make functions of best fit from.

Because of the differences in the "randomly" generated data, we all wound up with different Chi2 values and thus different Chi2 to NDF (number of degrees of freedom) ratios when adjusting our parameters the same way. An important aspect of the data analysis portion of our research was being able to distinguish between signal and background noise—particularly so in the deltaE graph which had two gaussians. Figure 8 shows the two gaussians in question, with the broader curve on left being part of the background, whereas the more narrow peak toward the right is the signal we are concerned with.

To create functions that would be able to fit our data in ROOT, we had to type the following code into the command section of our ROOT browser:

```
root [1] TF1 *myfunc = new TF1("MyFitFunction", "gaus(0) + pol4(3)", 5.2, 5.3)
(TF1 *) 0x10a0fd6c0
root [1] myfunc->SetNpx(10000)
root [1] myfunc->SetParNames("Constant", "Mean", "Sigma", "p0", "p1", "p2", "p3", "p4")
root [1] myfunc->SetParameters(426.0, 5.28, 0.0044, 60.0, 0.0, 0.0, 0.0, 0.0)
root [1] Mbc->Fit("MyFitFunction", "", "", 5.21, 5.29)
```

Figure 4. Fitting MBC

```
root [1] TF1 *myfunc2 = new TF1("MyFitFunction2", "gaus(0) + gaus(3)", -2.0, 2.0)
(TF1 *) 0x125409dc0
root [1] myfunc2->SetNpx(10000)
root [1] myfunc2->SetParNames("Constant", "Mean", "Sigma", "Constant2", "Mean2", "Sigma2")
root [1] myfunc2->SetParameters(2000.0, 0.0, 0.05, 300.0, 0.0, 0.3)
root [1] deltaE->Fit("MyFitFunction2", "", "", -1.0, 1.0)
```

Figure 5. Fitting deltaE

The first line of code described what the function would be made out of, in the case of MBC it was a gaussian and a 4th degree polynomial (pol4). The arguments of the functions gaus and pol4 labelled what parameters aligned to what function, which were defined in the third line of code. The gaussian function took in a "Constant" (the height of the gaussian), a "Mean" (the mean value of the gaussian), and a "Sigma" (the standard deviation or width of the gaussian). The 4th degree polynomial took in 5 parameters—p0, p1, p2, p3, p4—which all made up the coefficients of the polynomial such that:

$$p_4x^4 + p_3x^3 + p_2x^2 + p_1x + p_0 \tag{8}$$

The next line of code sets the values of each parameter in the function. In our first round of doing this, we set the parameter values for the polynomial to all be 0, and allowed the function to determine which values worked best in terms of lowering the Chi2. This could lead to issues, such as getting stuck in a local minimum of a best fit when the real best fit could be elsewhere if the parameters

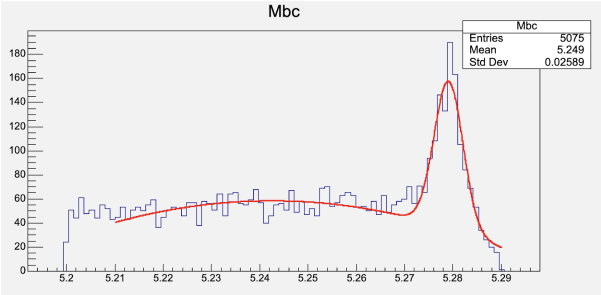

Figure 6. Graph of MBC w/ curve of best fit

```
Minimizer is Minuit2 / Migrad
Chi2                      =      100.502
NDf                       =           67
Edm                       =  8.39736e-07
NCalls                    =          538
Constant                  =      122.183   +/-   6.57827
Mean                      =      5.27907   +/-   0.000152629
Sigma                     =   0.00299147   +/-   0.000194769
p0                        =      -134449   +/-   13458.2
p1                        =      12846.8   +/-   1281.21
p2                        =      4891.93   +/-   488.664
p3                        =      465.609   +/-   46.5882
p4                        =     -177.898   +/-   17.7409
(TFitResultPtr) <nullptr TFitResult>
```

Figure 7. Values of MBC's function of best fit

had been changed to a different starting point. The final line gives the function a starting and ending point.

We began to play around with ways and methods of modifying the fit to make it better—the closer the Chi2 to NDF ratio is to 1, the better the fit. Adjusting our parameters allowed us to lower our Chi2 and bring it closer to the NDF value; we found we could do this by adjusting where the mean or standard deviation was. Another method of lowering the Chi2 was finding a part of the function that wasn't mapping right onto the data, and cutting that section off (the tail ends). The closer we got to what seemed to be the genuine values of the mean, standard deviations, constants, and coefficients, the better the function was able to fit the data.

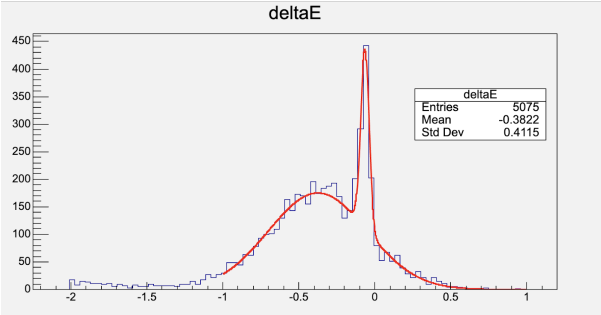

Figure 8. Graph of deltaE w/ curve of best fit

7 Conclusion

This semester-long process gave us a well-rounded look at many facets of particle physics practices, and allowed us to build our skills gradually over the course of the semester. We notably enhanced our problem solving, data analysis, and programming proficiency. We also got our hands onto an operating system, software, and realistic datasets that were completely new to us, and were uniquely tied to the particle physics world. Looking back, we accomplished quite a bit, and developed a better understanding of many different topics while also getting the birds-eye view of how it all fits together.

```
Minimizer is Minuit2 / Migrad
Chi2                      =      79.9169
NDf                       =           45
Edm                       =  2.50384e-07
NCalls                    =          313
Constant                  =      324.702   +/-   21.4976
Mean                      =   -0.0657734   +/-   0.00172254
Sigma                     =    0.0274465   +/-   0.00177147
Constant2                 =      175.057   +/-   3.85119
Mean2                     =    -0.375038   +/-   0.00606478
Sigma2                    =     0.325477   +/-   0.00512372
(TFitResultPtr) <nullptr TFitResult>
```

Figure 9. Values of deltaE's function of best fit

Acknowledgements

This research received support during the PHYS 2994 course, instructed by Professor Piilonen in the College of Science at Virginia Polytechnic Institute and State University.