PHYS 488. Project. Individual Report

Lenka Pollachová (201007952)

Department of Physics, The University of Liverpool.

May 2016

Contents

1 Introduction

In the Standard Model, traditionally, lepton flavour is conserved at tree level. However, lepton flavour violation has been observed in neutrinos in the form of neutrino oscillations by the Super-Kamiokande observatory [5] in Japan and the Sudbury Neutrino Observatory [3] in Canada. It is therefore expected for lepton flavour violation to be observed in charged leptons as well. An experiment has been proposed by the Paul Scherrer Institut [6] in Switzerland that would attempt to take a closer look at the lepton flavour violating decay $\mu^+ \longrightarrow e^+e^-e^+$, which would theoretically only be viable via a process involving neutrino mixing:

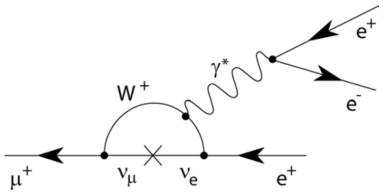


Figure 1: A Feynman diagram for the lepton flavour violating decay $\mu^+ \longrightarrow e^+e^-e^+$ involving a neutrino mixing loop [2].

The group I was a part of attempted to create a simulation of the events of a muon decaying into three electrons $\mu^+ \longrightarrow e^+e^-e^+$ and then take the results of this simulation as if they came from an actual experiment and analyse them accordingly. My contribution to the group involved creating a constructor class that would simulate individual detectors and calculate particle hits on these detectors. Once an instance corresponding to a given detector is created, the class takes the initial position and momentum of a particle and calculates the hit on a detector. This, together with the momentum with which the particle enters the detector material, can then be used further in the calculations of multiple Coulomb scattering and energy loss. I have also done major work on developing the main class that would string the different pieces of code together.

1.1 A Word on Geometry

The three particles are simulated to randomly appear somewhere in a well defined cylindrical space within the detector layers. They are then given randomised momenta values such that their respective momenta add up to zero (i.e. they are created from rest), and their energies add up to the muon mass. In this way these particles are assigned two four vectors, $\underline{\mathbf{X}} = (t, x, y, z)$ and $\underline{\mathbf{P}} = (E, p_x, p_y, p_z)$. The time component has not been implemented into the simulation, but in reality it plays an important role in distinguishing different decay events from one another. There is a uniform magnetic field $B = 1T$ along the z-axis in the positive direction. This uniform magnetic field causes curvature to the path of the charged particles travelling through it. This is due to the Lorentz force:

$$\vec{F_B} = q\vec{v} \times \vec{B}, \tag{1}$$

which only affects those momentum components that are parallel to the direction of the magnetic field. In our set up, this means that the path of the particle only gets affected in the x- and y- directions, while the movement of the particle in the z- direction is completely unaffected by the magnetic field (it is however affected by any Coulomb scattering within the material of the detector). This is schematically represented in figure 2.

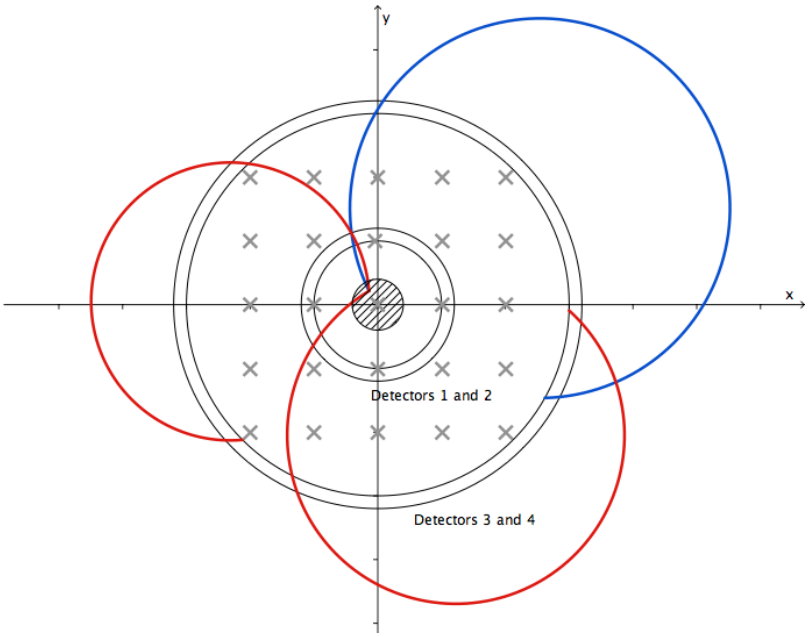


Figure 2: Schematic of the geometry of the simulation. Particles are created within the shaded region. The uniform magnetic field is directed into the page, causing any positively charged particles to curve anticlockwise and negatively charged particles to curve clockwise. Plot created using GeoGebra [4].

2 Motivation behind the code

The main assumption made in the following calculations is that the particle is travelling in perfectly circular paths when travelling between the individual detector layers. That is, that there is perfect vacuum and the effects of energy loss and Coulomb scattering are negligible.

2.1 Hit position

A particle with charge Q, originating at (x_0, y_0) with momentum $\vec{p} = (p_x, p_y, p_z)$ will, given a uniform magnetic field $\vec{B}$ and perfect vacuum, follow a path described by

$$(x - x_0 - |r|\sin\phi + Q\frac{\pi}{2})^2 + (y - y_0 - |r|\cos\phi + Q\frac{\pi}{2})^2 = r^2, \tag{2}$$

where $r = \frac{|p_\perp|}{0.3|\vec{B}|}$ is the radius of the curvature of the particle, and $\phi = \arctan\frac{p_y}{p_x}$ [1]. The detector is simulated as a circular body with its origin at $(x, y, z) = (0, 0, 0)$ and radius R. As such, the detector can be described by the equation

$$x^2 + y^2 = R^2. \tag{3}$$

This situation is graphically represented in figure (3).

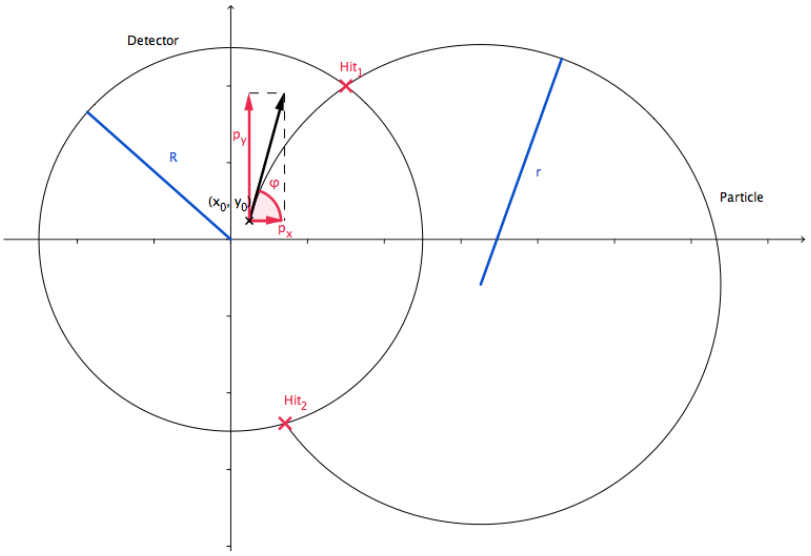


Figure 3: Graphical representation of the problem at hand. Magnetic field is alongside the z-axis and as such only affects the trajectory of the particle in the x-y plane.

This essentially simplifies to the problem of finding the intersection of two circles. Rewriting (2) as

$$(x + C_1)^2 + (y + C_2)^2 = r^2$$

with $C_1 = -x_0 - |r| \sin\phi + Q\frac{\pi}{2}$ and $C_2 = -y_0 - |r| \cos\phi + Q\frac{\pi}{2}$, one can expand the brackets and subtract equation (3) to obtain a new equation linear in x and y:

$$y = -\frac{C_1}{C_2}x + \frac{r^2 - C_1^2 - C_2^2 - R^2}{2C_2} \tag{4}$$

Again, the equation can be simplified by introducing new variables m and C where $m = -\frac{C_1}{C_2}$ and $C = \frac{r^2 - C_1^2 - C_2^2 - R^2}{2C_2}$ (appendix A: line 116). That way equation (4) can be rewritten as $y = mx + C$ and substituted into equation (3) to give an equation quadratic in x:

$$x^2 + (mx + C)^2 = R^2. \tag{5}$$

Expanding, this can be rewritten as

$$(m^2 + 1)x^2 + (2Cm)x + (C^2 - R^2) = 0, \tag{6}$$

a quadratic equation with coefficients $a = m^2 + 1$, $b = 2Cm$, and $c = C^2 - R^2$ (appendix A: line 124). This quadratic equation can be solved using the determinant (appendix A: line 50) to give two solutions as follows:

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{7}$$

Substituting these back into equation (4) one obtains the two sets of coordinates (x_1, y_1) and (x_2, y_2) for the two intersection points of the two circles. The outlook of this is presented in figure 4 below:

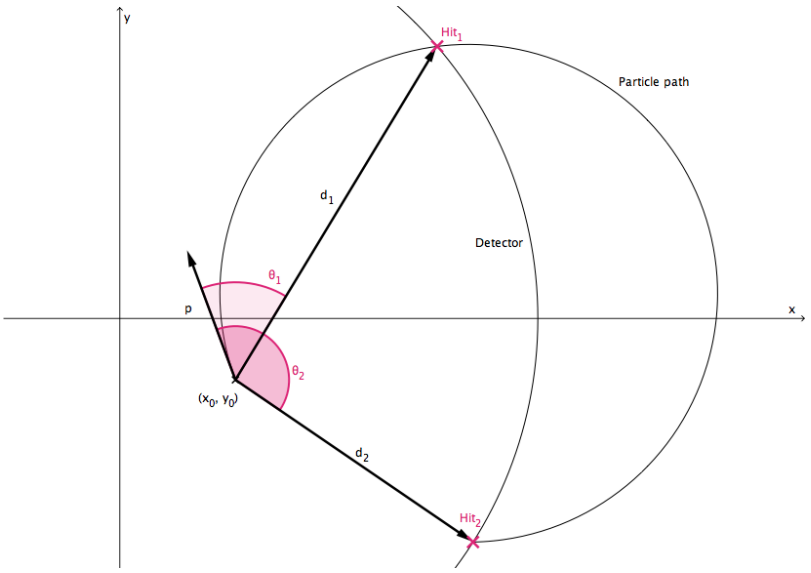


Figure 4: This shows the two calculated hit coordinates - Hit_1 and Hit_2.

Now a problem arises from the fact that when this function is called, a user only desires to know the one solution corresponding to Hit_1. When a particle hits the detector, it will then undergo multiple coulomb scattering and energy loss, resulting in the need for its path to be recalculated and a new value for Hit_2 to be obtained. This problem was solved by considering (1.) the distance from the particle starting position to the particle hit position, (2.) the angles between the initial $\vec{p}$ vector and the vector from the starting position to the hit position, which I shall call vector $\vec{x}$, and (3.) comparing the actual hit position value with the start position value to ensure they are not identical. There are numerous possible combinations of starting conditions and hit conditions, hence all of these need to be considered in some combination.

1. First, the distance from the particle's start position (x_0, y_0) to the two possible hits is found by simple distance formula
$$d_{1/2} = \sqrt{(x_0 - x_{hit,1/2})^2 + (y_0 - y_{hit,1/2})^2}$$
(appendix A: line 173). Then, using a basic *if* condition, the hit coordinates corresponding to the shortest distance are selected. This condition is sufficient to discriminate between solutions once the particle is already in between the detector layers, however it fails when the particle is just created and the very first detector hit needs to be determined. Consider for example the case pictured in figure 4. The distance to the Hit_1 is clearly greater than that to Hit_2. This is where the need for the second condition arises.

2. Secondly then, we consider the two vectors $\vec{x}$ and $\vec{p}$ introduced above, and the angle between them. The angle is found from the Euclidean definition of a vector scalar product:
$$\vec{p} \cdot \vec{x} = |\vec{p}||\vec{x}| \cos\theta,$$
rearranging:
$$\theta = \arccos \frac{\vec{p} \cdot \vec{x}}{|\vec{p}||\vec{x}|}.$$
This way two angles can be obtained corresponding to the two hit positions (appendix A: line 159). Then it is simply a matter of comparing the two, where the smaller angle corresponds to the solution desired.

3. Finally, to make sure that when a particle leaves the last detector to be later recorded on that same detector the program doesn't simply return that same starting position, a condition of kind *if* $(x_1 == x_0)$ *return* x_2 had to be included.

2.2 Momentum of hit

The momentum of the hit is found by first finding a vector from the centre of the particle path to the point where the particle hits the detector, and then finding a vector perpendicular to it. If the hit coordinates are (x_{hit}, y_{hit})

and the coordinates of the centre of the particle path are $(-C_1, -C_2)$, then the vector connecting these is

$$(x_{hit} + C_1, y_{hit} + C_2)$$

and a vector perpendicular to it is given by

$$\vec{v} = (-(y_{hit} + C_2), x_{hit} + C_1).$$

Scaling this vector to a unit vector and multiplying by the magnitude of the momentum vector perpendicular to the direction of the magnetic field yields the desired result:

$$\vec{p_{hit}} = \begin{bmatrix} p_x \\ p_y \end{bmatrix}_{hit} = \frac{|\vec{p_\perp}|}{|\vec{v}|} \times \begin{bmatrix} -(y_{hit} + C_2) \\ (x_{hit} + C_1) \end{bmatrix}. \tag{8}$$

The vector also has to be scaled by -1 (or $+1$) depending on the charge of the particle (appendix A: line 250).

3 Output

The class as such does not simulate anything, it simply processes input and returns hit position and momentum based on this input. I did however write another class that simulates a single particle and sends it through a number of detectors to follow its path exactly. This class does not take into account any scattering or energy loss that occur within the detector, rather, it simply tests whether my main constructor class behaves as it should. This set up is illustrated in figure 5 below:

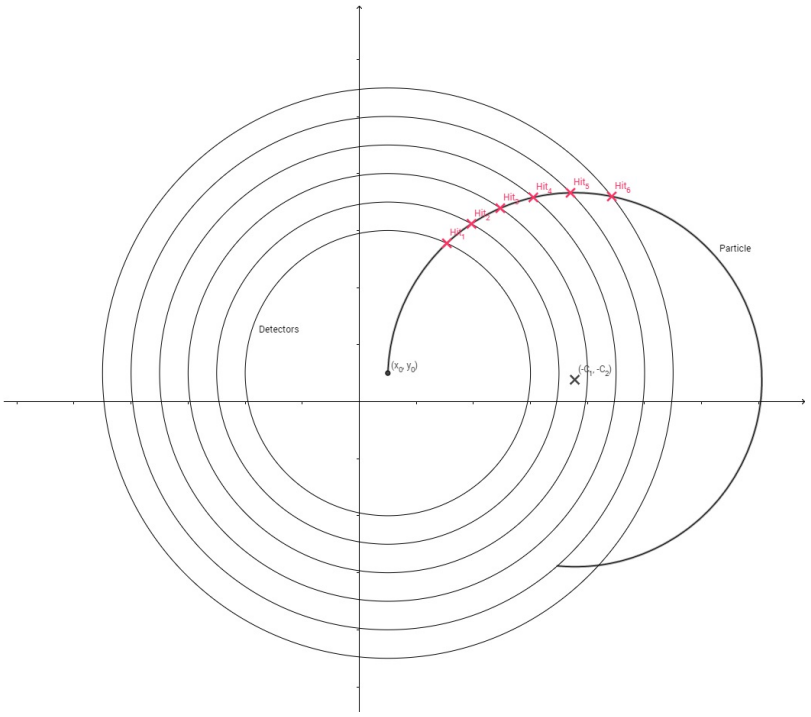


Figure 5: Detector set up in order to debug.

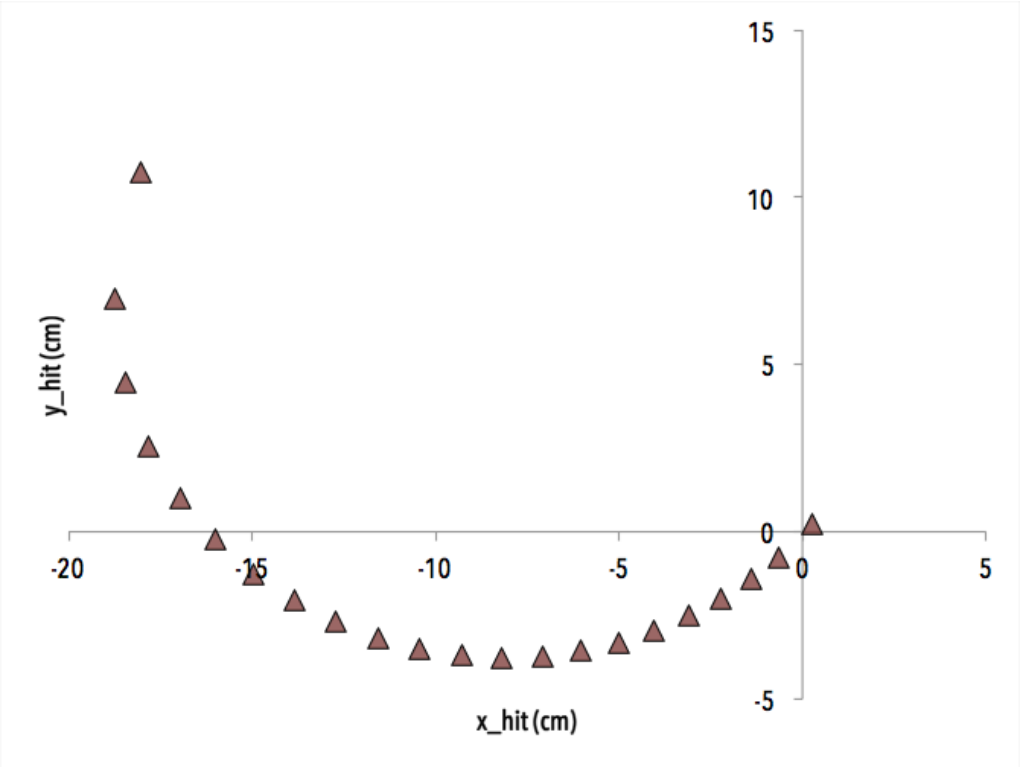
Using this other class that actually simulates a particle, I could obtain plots as shown in figure 7. I also printed the initial position, momentum, and charge to screen for comparison shown in figure 6.

```
Position: (x, y) = (0.25, 0.25)
Momentum:  p_x = -20.0, p_y = -25.0
Charge: -1.0

Please type in the file name
particlePath
 Data written to disk in file particlePath.csv
```

Figure 6: Screen prints from BlueJ.

x	y
0.25	0.25
-0.6605004	-0.7508257
-1.4139414	-1.4144857
-2.2300073	-2.0067555
-3.1043281	-2.522528
-4.032308	-2.9564323
-5.0091229	-3.3028303
-6.0296532	-3.5557393
-7.0883951	-3.7087268
-8.1793437	-3.7547752
-9.2958403	-3.6861026
-10.430365	-3.4939208
-11.574246	-3.1680949
-12.717236	-2.696649
-13.846867	-2.065013
-14.947422	-1.2548187
-15.998172	-0.2418398
-16.970099	1.00783588
-17.819061	2.5457967
-18.468855	4.46109686
-18.755167	6.94576824
-18.048586	10.7353875

(a) Java output in the form of a .csv file.



(b) Plot using the data from the .csv file.

Figure 7: An example of the output obtained for a negatively charged particle with initial momenta $P_x = -20MeV$ and $P_y = -25MeV$. Magnetic field oriented into the plane of the page, causes negatively charged particles to curve clockwise.

4 Troubleshooting

When working on the program, the part that calculates the hit position was working correctly on the first try. When coding a program that would calculate the hit momentum, however, I took two completely different approaches before arriving at the final one. At the very beginning, I considered momentum change $\Delta\vec{p}$ in small time increments, and kept on adding these increments onto the current momentum $\vec{p_{new}} = \vec{p_{old}} + \Delta\vec{p}$ in a loop until arriving at the momentum at the hit position. This was however deemed inefficient as in order to minimise inaccuracies, the individual time increments had to be small, which in turn meant taking a great number of steps, resulting in a high computing time.

The second approach I took was using trigonometric identities, congruent triangles, and other such geometrical constructions. I have illustrated this approach in figure 8 below:

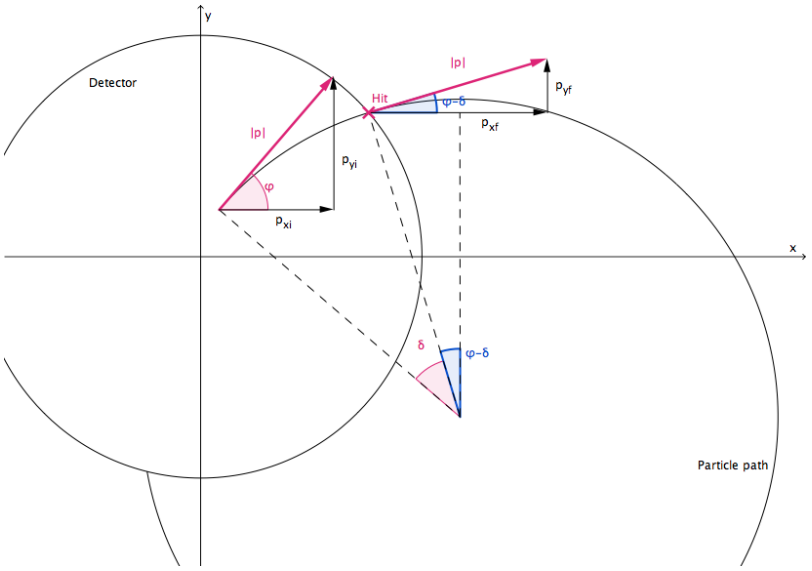


Figure 8: Illustration of the relations between individual momentum components.

While this worked perfectly well in theory, in practice Java returns any arccosines and arcsines in the range $0 < \theta < \frac{\pi}{2}$ and so the code called for a large number of *if* statements that would check the initial momentum and position and the hit position and add π or 2π or any other needed value to the calculated angle such that it would make physical sense. This method, if executed properly, would have worked, but I did not have enough time to go through every possible combination of starting coordinates, momentum, and hit coordinates individually and check that the returned angle is physical.

In the final version of the class, I still have not managed to account for all different scenarios. In some instances, positively charged particles curve as if they were negatively charged and vice versa. For these cases, I have hard coded into the main method of the simulation a condition of the kind

```
if (p_x > 0 && p_y > 0)
    {z = - Q;}
```

and for the calculation of hit position and hit momentum used this new variable z rather than Q. While this solution is not ideal, it does work and produces the right trajectories.

5 Individual Study

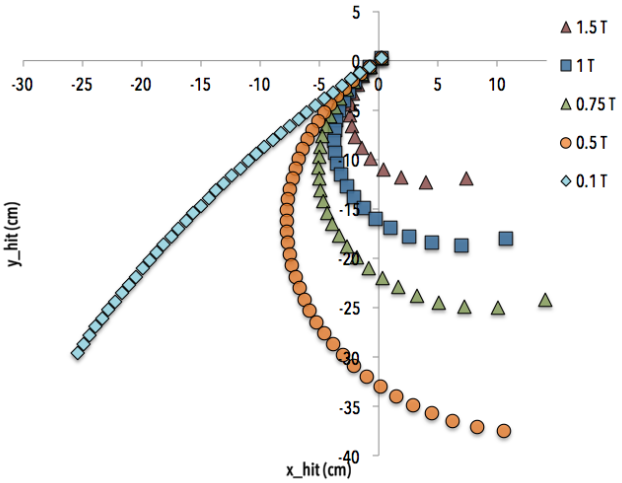
5.1 Varying Magnetic Field Strength

Taking the expression giving Lorentz force (1) and the expression giving the relationship between centripetal force and radius of curvature $F = \frac{mv^2}{r}$, the two can be solved to find r:

$$r = \frac{p}{QB}, \tag{9}$$

1.5 T		1 T		.75 T		.5 T		.1 T	
x	y	x	y	x	y	x	y	x	y
0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
-0.7216838	-0.6922229	-0.7508257	-0.6605004	-0.7648498	-0.6442087	-0.7785012	-0.6276431	-0.7995543	-0.6005939
-1.3189707	-1.5034348	-1.4144857	-1.4139414	-1.4598201	-1.3670865	-1.5035054	-1.3188903	-1.5699102	-1.2391052
-1.7998399	-2.4001201	-2.0067555	-2.2300073	-2.1035623	-2.1389309	-2.1958746	-2.0440486	-2.3340915	-1.8846795
-2.152247	-3.3716217	-2.522528	-3.1043281	-2.693289	-2.9573966	-2.8544519	-2.8021607	-3.0921909	-2.5373915
-2.3630832	-4.4063406	-2.9564323	-4.032308	-3.226065	-3.8200137	-3.4779685	-3.592177	-3.8441859	-3.197223
-2.4177123	-5.4913266	-3.3028303	-5.0091229	-3.698864	-4.7242359	-4.0651246	-4.4130219	-4.5900348	-3.8641404
-2.2992295	-6.6116219	-3.5557393	-6.0296532	-4.1085566	-5.66743	-4.6145972	-5.2636007	-5.3296902	-4.5381056
-1.9873006	-7.7492346	-3.7087268	-7.0883951	-4.4518872	-6.6468565	-5.1250389	-6.1427987	-6.0631029	-5.2190788
-1.4562096	-8.8814106	-3.7547752	-8.1793437	-4.725445	-7.6596455	-5.5950746	-7.049478	-6.7902223	-5.9070196
-0.6712549	-9.9774454	-3.6861026	-9.2958403	-4.92563	-8.7027679	-6.023297	-7.9824741	-7.5109979	-6.6018869
0.4187758	-10.992026	-3.4939208	-10.430365	-5.0486109	-9.773	-6.4082615	-8.940592	-8.2253783	-7.3036397
1.90044761	-11.848557	-3.1680949	-11.574246	-5.0902749	-10.866881	-6.7484808	-9.9226008	-8.9333122	-8.0122364
3.96012489	-12.382141	-2.696649	-12.717236	-5.0461647	-11.98066	-7.0424184	-10.927229	-9.6347483	-8.7276358
7.39444023	-11.887904	-2.065013	-13.846867	-4.9113992	-13.110231	-7.2884817	-11.95316	-10.329635	-9.4497961
0	0	-1.2548187	-14.947422	-4.6805718	-14.251044	-7.485014	-12.999022	-11.01792	-10.178676
0	0	-0.2418398	-15.998172	-4.3476164	-15.397994	-7.6302852	-14.063383	-11.699552	-10.914233
0	0	1.00783588	-16.970099	-3.9056286	-16.545273	-7.722482	-15.144744	-12.374479	-11.656426
0	0	2.5457967	-17.819061	-3.3466181	-17.686157	-7.7596955	-16.241525	-13.042649	-12.405213
0	0	4.46109686	-18.468855	-2.6611593	-18.812715	-7.7399076	-17.352056	-13.70401	-13.160552
0	0	6.94576824	-18.755167	-1.8378785	-19.915376	-7.6609746	-18.474563	-14.358509	-13.9224
0	0	10.7353875	-18.048586	-0.8626715	-20.982273	-7.5206083	-19.607153	-15.006094	-14.690716
0	0	0	0	0.2825516	-21.998185	-7.3163529	-20.747795	-15.646713	-15.465457
0	0	0	0	1.62200792	-22.942735	-7.0455579	-21.894294	-16.280313	-16.246581
0	0	0	0	3.19000795	-23.787052	-6.7053442	-23.04427	-16.906841	-17.034046
0	0	0	0	5.03915871	-24.486872	-6.2925636	-24.195116	-17.526245	-17.827808
0	0	0	0	7.26033705	-24.965727	-5.8037479	-25.343964	-18.138471	-18.627825
0	0	0	0	10.0464441	-25.061304	-5.2350444	-26.487626	-18.743466	-19.434055
0	0	0	0	14.0683169	-24.20914	-4.5821348	-27.622528	-19.341176	-20.246454
0	0	0	0	0	0	-3.8401285	-28.744624	-19.931549	-21.064979
0	0	0	0	0	0	-3.003423	-29.849279	-20.514531	-21.889587
0	0	0	0	0	0	-2.0655149	-30.931111	-21.090067	-22.720235
0	0	0	0	0	0	-1.01874	-31.98378	-21.658104	-23.556878
0	0	0	0	0	0	0.14609638	-32.999677	-22.218588	-24.399474
0	0	0	0	0	0	1.44025889	-33.969481	-22.771464	-25.247979
0	0	0	0	0	0	2.87790418	-34.88148	-23.316677	-26.102348
0	0	0	0	0	0	4.47739056	-35.720484	-23.854172	-26.962538
0	0	0	0	0	0	6.26349295	-36.465993	-24.383896	-27.828504
0	0	0	0	0	0	8.27144453	-37.088856	-24.905792	-28.700201
0	0	0	0	0	0	10.5551111	-37.544502	-25.419804	-29.577585

(a) Java output in the form of a .csv file.



(b) Plot using the data from the .csv file.

Figure 9: Investigation of the effects of varying magnetic field strength on the trajectory of a positively charged particle.

i.e. the radius of curvature is inversely proportional to the strength of the magnetic field. Varying the strength of magnetic field passed into the constructor class I created, this is exactly the effect that could be observed: The detectors in figure 9 are spaced equally in $1cm$ increments from $1cm$ to $40cm$. It is evident (refer to figure 9a) that as the magnetic field strength gets stronger, the particles don't ever reach some of the detectors. In other words, the spacing and positioning of the detectors has to be chosen carefully and relative to the strength of the magnetic field. The radius of the curvature is further also proportional to the momentum in the x-y plane of the particle, and even within our final simulation we found that sometimes one of the three decay product particles is created such that its momentum is too low to reach the third and fourth detector. These points then had to be discarded from analysis.

While the strength of the magnetic field and spacing of detectors is discussed and taken into account in the PSI proposal, I found no mention of the second effect, which leads me to think that the way we assigned the momenta to individual particles is not entirely physical.

5.2 Using Data from all Detectors to Improve the Accuracy of Analysis

Another aspect of our simulation I chose to investigate is how the accuracy of our analysis could be improved. Specifically, how could the calculation of the centre of trajectory be improved and from there the determination of the vertex of the event. Presently, the analysis relies on first finding the centre of the trajectory by method illustrated in figure 10 below:

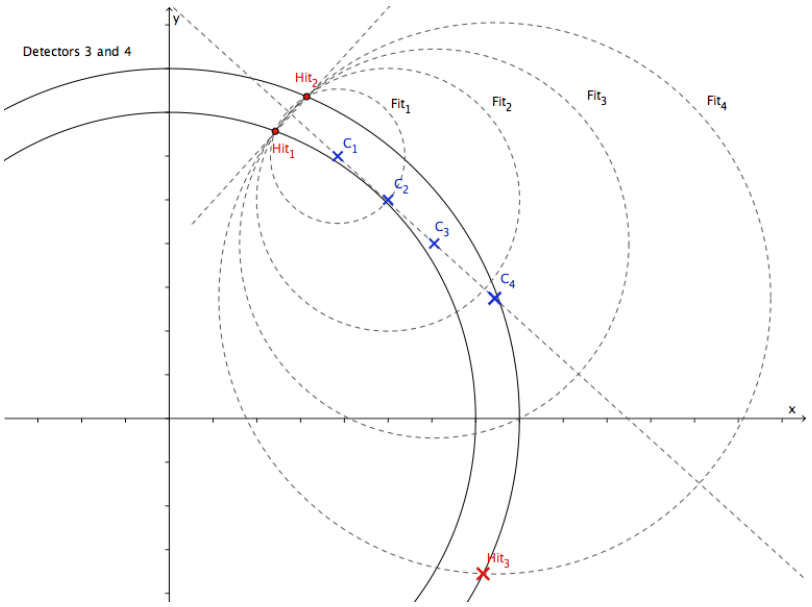


Figure 10: Method used to obtain the momentum of a particle from its hits on the outer set of detectors.

A bisector line is drawn in between the hits on the outer set of detectors and a circle is fitted from a number of points along the bisector, until it matches up with the hit obtained after the particle's recoil. The problem with this method is that once multiple coulomb scattering, energy loss, and the limitations due to the detector resolution are taken into account, it quickly becomes very inaccurate. Using a similar idea and using the hits on all the detectors would improve the accuracy of the analysis greatly. What I propose is to construct perpendicular bisectors between each subsequent hit, as well as between the hit on detector four, the recoil hit and detector one, and evaluate the intersections of each set of bisectors. This way, there are $\binom{5}{2}$ $(= 10)$ sets of coordinates obtained, which can easily be averaged out to obtain a set of coordinates corresponding to the centre of that particle's trajectory.

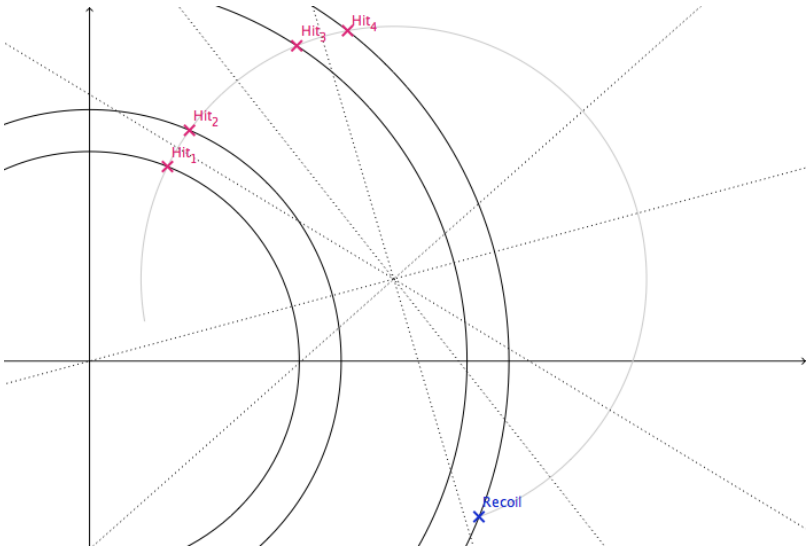


Figure 11: Proposed method for centrepoint calculation.

This way, the effects of coulomb scattering and energy loss would be averaged out in the process and would not skew the data as much. The coordinates would be simply given as

$$\bar{x} = \frac{\sum_{n=1}^{10} x_n}{10} \qquad \text{and} \qquad \bar{y} = \frac{\sum_{n=1}^{10} y_n}{10},$$

with statistical uncertainties given by the standard error on mean:

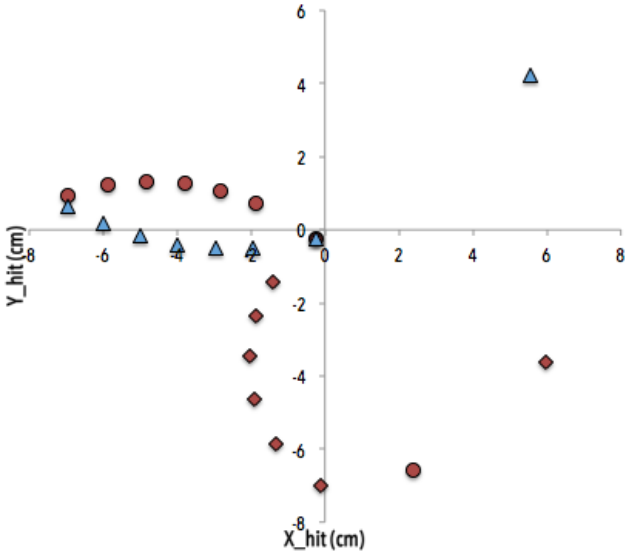
$$SE_x = \frac{\sqrt{\sum_{n=1}^{10}(x_n - \bar{x})^2}}{10} \qquad \text{and} \qquad SE_y = \frac{\sqrt{\sum_{n=1}^{10}(y_n - \bar{y})^2}}{10}.$$

From there, it is easy to show how adding another more detectors could improve the accuracy of the calculated position. Adding another detector, constructing perpendicular bisectors as before, and calculating their intersections would result in $\binom{6}{2}$ $(= 15)$ points of intersection. These averaged out would give a more precise result as the error on mean decreases proportionally to the inverse of number of intersection points. The class calculating the vertex of the decay (i.e. the starting position of the three particles) uses the three calculated centrepoints and as such, the error on centrepoint coordinates propagates into the calculations of the vertex coordinates. Due to time constraints, I was not able to properly carry out all the steps described above, however, I have attempted to.

Using the final simulation, I first suppressed the randomly thrown starting position and momenta to make sure that the case analysed would work properly and to be able to replicate it if needed. The java output together with the plotted .csv file are shown in figure **??**. I have then rewritten the analysis class; first loading up the .csv file

```
Detector 0: radial position: 2.0
Detector 1: radial position: 3.0
Detector 2: radial position: 4.0
Detector 3: radial position: 5.0
Detector 4: radial position: 6.0
Detector 5: radial position: 7.0

Position: (x, y) = (-0.25, -0.25)
Particle 1:
Momentum:  p_x = 20.0, p_y = -25.0
Charge: 1.0

Particle 2:
Momentum:  p_x = -10.0, p_y = -15.0
Charge: 1.0

Particle 3:
Momentum:  p_x = 10.0, p_y = 40.0
Charge: -1.0


Please type in the file name
muonDecay
--> Data written to disk in file muonDecay.csv
```

(a) Java print-out.



(b) Plot using the data from the .csv file.

Figure 12: Simulating an event.

(appendix B: line 41), then using the points to calculate central points of the three particles' trajectories (appendix B: line 101). The output of this analysis class is shown below:

```
Particle 1 (x,y) = (-7.8240743, -9.675926)
Particle 2 (x,y) = (3.2870371, -5.9722223)
Particle 3 (x,y) = (-4.1203704, 14.398149)
```

Figure 13: Calculated average centrepoints of the particles' trajectories.

I have then simulated the same event, taking into account the limitations due to the resolution of the detectors. I did not manage to incorporate the calculations of standard error on mean into my analysis class due to time constraints (although I've attempted, see appendix B: line 133), but with just a little more time this could be implemented and also effects of scattering and energy loss could be added back into the simulation and their effects on the accuracy of the analysis properly investigated.

6 Conclusion

Java, as an object oriented programming language, was ideal for this project as different pieces of code needed to be reused multiple times. Within the program that I wrote, I used different class methods that serve a single

purpose and can be used throughout the program. An example of such a class method is one named *dotProduct* which as the name suggests, when one passes two vectors into this method, it returns the dot product of the two vectors. Another example would be a class method titled *getLength*, which takes two numbers, say a and b and returns $length = \sqrt{a^2 + b^2}$. I used such methods heavily throughout my code, as certain tasks needed to be performed over and over again.

From the perspective of the whole group project, i.e., the grand simulation itself, my program also acts as a single reusable unit whose sole purpose is to return the hit position and momentum when called. In this way, the whole simulation could be put together despite each person's different coding style. Additionally, classes written in weeks prior to the start of the project, such as the classes calculating energy loss, multiple coulomb scattering, or smearing of hits due to detector resolution could be easily reused.

In the end, as a group, we managed to simulate the individual decay events, calculate hits, shakily incorporate the classes written in weeks prior, and finally analyse the individual hits. There is surely plenty room for improvement - background decays could be simulated, the whole process could be automated a bit more, the crude hard-coded instances could be rewritten - but given that none of the members of our group worked with Java prior to taking this module, the simulation came together rather well.

References

[1] *A Simple Model of the ATLAS Upgrade Tracker*. Mar. 2016.

[2] A. Blondel et al. "Research Proposal for an Experiment to Search for the Decay μ eee". URL: `https://www.psi.ch/mu3e/DocumentsEN/ResearchProposal.pdf`.

[3] The Sudbury Neutrino Observatory Institute. *The SNO*. URL: `http://www.sno.phy.queensu.ca/` (visited on 04/29/2016).

[4] International GeoGebra Institute. *GeoGebra*. Version 4.2.56.0. Mar. 16, 2016. URL: `http://www.geogebra.org/`.

[5] Univ. of Tokyo Kamioka Observatory ICRR. *Super-Kamiokande*. URL: `http://www-sk.icrr.u-tokyo.ac.jp/sk/index-e.html` (visited on 04/29/2016).

[6] Prof. Dr. Joël Mesot. *Paul Scherrer Institut*. URL: `https://www.psi.ch/` (visited on 04/29/2016).

A Detector Class Code

```
import java.io.*;
/**
 * Given a magnetic field strength and radial position of a detector, this class returns the hit
    on the detector.
 *
 * @ Lenka Pollachova
 * @ March 2016
 */
public class Hit
{
    static PrintWriter screen = new PrintWriter( System.out, true);

    // instance variables - replace the example below with your own
    private double B, R;

    protected double m = .511; // MeV
    protected double pi = Math.PI;

    /**
     * Constructor for objects of class Curvature
     */
    public Hit(double strength, double rad)
    {
        // initialise instance variables
        B = strength; // in Tesla, uniform magnetic field
        R = rad; // in cm
    }

    public double getRadius(double x, double y)
    {
        double r = getLength(x, y) / (3 * B); // B in T, P in MeV, r in cm
        return r;
    }

    public double getLength(double x, double y)
    {
        double length = Math.sqrt(x*x + y*y);
        return length;
    }

    public double dotProduct(double [] x, double [] y)
    {
        double sum = 0;
        for (int i = 0; i < x.length; i++)
        {
            sum = sum + x[i] * y[i];
        }
        return sum;
    }

    public double Root1(double a, double b, double c)
    {
        double sol1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
        return sol1;
    }

    public double Root2(double a, double b, double c)
    {
        //screen.println("test root " + (b*b - 4*a*c));
        double sol2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);
        return sol2;
    }

    public double [] findCenter(double x0, double y0, double r, double phi, double Q, double []Pi
    )
    {
        double C1, C2;
        if (Q == 1)
```

```
67          {
68              C1 = - x0 - r * Math.sin(phi + Q*pi/2);
69              C2 = - y0 - r * Math.cos(phi + Q*pi/2);
70
71          }
72          else
73          {
74              C1 = - x0 - r * Math.sin(phi + Q*pi/2);
75              C2 = - y0 - r * Math.cos(phi + Q*pi/2);
76          }
77          double [] C = {C1, C2};
78          return C;
79      }
80
81      public double [] getPosition(double []Xi, double []Pi, double Q)
82      {
83          double x = 0, y = 0;
84
85          /**
86           * particle starting at (x0, y0) will describe the trajectory
87           * (x - x0 - r*sin(phi + Q pi/2))^2 + (y - y0 - r*cos(phi + Q pi/2))^2 = r^2
88           * let me summarise the constant values into C1 and C2
89           */
90
91          double x0 = Xi[1];
92          double y0 = Xi[2];
93          double phi;
94          phi = Math.atan2(Pi[2], Pi[1]); // radians
95          double r = getRadius(Pi[1], Pi[2]); // cm
96
97          //screen.println("test pos " + Xi[1]+ ", "+ Xi[2] );
98          //screen.println("test mom "+ Pi[1]+ ", "+ Pi[2]);
99          //screen.println(" test r "+ r);
100
101         double [] Centre = findCenter(x0, y0, r, phi, Q, Pi);
102         double C1 = Centre[0];
103         double C2 = Centre[1];
104
105         //screen.println( " c1 = " +C1+ ", c2= "+C2);
106         double [] X = new double [Xi.length];
107
108         if (2*r < R) {return X;}
109
110         /**
111          * eq I: (x+C1)^2 + (y+C2)^2 = r^2  (circle that particle will follow)
112          * eq II: x^2 + y^2 = R^2 (circle of the detector)
113          * take I - II to obtain an equation linear in both x and y and rearrange to the form y =
     mx + C
114          */
115
116         double m = - C1 / C2;
117         double C = (r*r - R*R - C1*C1 - C2*C2) / (2*C2);
118
119         /**
120          * substitute values back into one of original equations to obtain a quadratic in x in
    the form
121          * a x^2 + b x + c = 0
122          */
123
124         double a = m*m + 1;
125         double b = 2*C*m;
126         double c = C*C - R*R;
127
128         /**
129          * solve quadratic using the determinant class method to get two solutions corresponding
    to the two intercepts
130          */
131
132         double x1 = Root1(a, b, c);
133         double x2 = Root2(a, b, c);
```

```
134
135         /**
136          * substitute this back into y = mx + C
137          * to obtain root y1 and y2 corresponding to x1 and x2 (aka two intersecting points)
138          */
139
140         double y1 = m*x1 + C;
141         double y2 = m*x2 + C;
142
143         /**
144          * now we have two roots (x1, y1) and (x2, y2)
145          * we now have to check which root makes sense
146          * to do this we find vector v1 = (x1 + C1; y1 + C2) and vector v2 = (x2+C1, y2+C2)
147          * and find the angle between this vector and the initial Pt vector
148          * the one that gives the smaller angle is the desired solution
149          */
150
151         double [] v1 = {x1 - x0, y1 - y0};
152         double [] v2 = {x2 - x0, y2 - y0};
153         double [] pt = {Pi[1], Pi[2]};
154
155         double v1Length = getLength(v1[0], v1[1]);
156         double v2Length = getLength(v2[0], v2[1]);
157         double ptLength = getLength(pt[0], pt[1]);
158
159         double theta1 = Math.acos(dotProduct(v1, pt) / (v1Length * ptLength));
160         double theta2 = Math.acos(dotProduct(v2, pt) / (v2Length * ptLength));
161         /*
162         if (Math.abs(theta1) < Math.abs(theta2))
163         {
164         X[1] = x1;
165         X[2] = y1;
166         }
167         else
168         {
169         X[1] = x2;
170         X[2] = y2;
171         }
172          */
173         double d1=Math.sqrt((x1-x0)*(x1-x0)+(y1-y0)*(y1-y0));
174         double d2=Math.sqrt((x2-x0)*(x2-x0)+(y2-y0)*(y2-y0));
175         if (d1<d2)
176         {
177             if (Math.abs(theta1) < Math.abs(theta2))
178             {
179                 if ( (float)x0 == (float) x1 && (float)y0 == (float)y1)
180                 {
181                     X[1] = x2;
182                     X[2] = y2;
183                 }
184                 else
185                 {
186                     X[1] = x1;
187                     X[2] = y1;
188                 }
189             }
190             else
191             {
192                 if ( (float)x0 == (float)x2 && (float)y0 == (float)y2)
193                 {
194                     X[1] = x1;
195                     X[2] = y1;
196                 }
197                 else
198                 {
199                     X[1] = x2;
200                     X[2] = y2;
201                 }
202             }
203         }
```

```
        else
        {
            if ((float) x0 ==(float) x2 && (float)y0 == (float)y2)
                {
                    X[1] = x1;
                    X[2] = y1;
                }
                else
                {
                    X[1] = x2;
                    X[2] = y2;
                }
        }
        //screen.println("test "+X[1]+ ", "+ X[2] );

        X[0] = Xi[0];
        X[3] = Xi[3];
        return X;
    }

    public double [] getMomentum(double []Xi, double []Pi, double Q)
    {
        double [] Xf = getPosition(Xi, Pi, Q);

        /**
         * particle starting at (x0, y0) will describe the trajectory
         * (x - x0 - r*sin(phi + Q pi/2))^2 + (y - y0 - r*cos(phi + Q pi/2))^2 = r^2
         * let me summarise the constant values into C1 and C2
         */

        double x0 = Xi[1];
        double y0 = Xi[2];

        double phi = Math.atan2(Pi[2], Pi[1]); // radians
        double r = getRadius(Pi[1], Pi[2]); // cm

        double [] Centre = findCenter(x0, y0, r, phi, Q, Pi);
        double C1 = Centre[0];
        double C2 = Centre[1];
        /**
         * equation becomes: (x+C1)^2 + (y+C2)^2 = r^2  (circle that particle will follow)
         * centre of circle at (-C1, -C2)
         * vector from origin (O) to hit position (P) OP = (xf + C1 , yf + C2)
         * vector perpendicular to that: vp = (xf + C1 , - yf - C2)
         */

        double [] v = new double [2];
        if ( Q == 1)
        {
            v[0] = -(Xf[1] + C1);
            v[1] = (Xf[2] + C2);
        }
        else
        {
            v[0] = (Xf[1] + C1);
            v[1] = -(Xf[2] + C2);
        }
        double vSize = getLength(v[0], v[1]);
        double pSize = getLength(Pi[1], Pi[2]);
        double [] p = new double [v.length];

        /**
         * now to make it to a unit vector we divide each component through by vSize
         * we then multiply by pSize to scale it back up to obtain the momentum components
         */

        for (int i = 0; i < v.length; i++)
        {
            p[i] = v[i] / vSize * pSize;
        }
```

```
274
275         if (Q == 1)
276         {
277             if (Math.abs(Math.atan2(p[1],p[0])-Math.atan2(Pi[2],Pi[1]))>pi/2.)
278             {
279                 p[0]=-p[0];
280                 p[1]=-p[1];
281             }
282         }
283
284         double E = Pi[0];
285         double pz = Pi[3];
286
287         double [] P = {E, p[0], p[1], pz};
288         return P;
289     }
290
291 }
```

B Analysis Class Code

```
import java.io.*;
import java.util.Scanner;
/**
 * Analyses data.
 *
 * @ Lenka Pollachov
 * @ May 2016
 */
public class Analysis
{
    static PrintWriter screen = new PrintWriter( System.out, true);
    static BufferedReader keyboard = new BufferedReader (new InputStreamReader(System.in)) ;

    public static void main(String [] args) throws IOException
    {
        /**
         * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         * Scanning .csv file into array
         * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         */
        int[] yearNum;
        double[][] hits;

        File file = new File("muonDecay.csv");
        Scanner scan = new Scanner(file);

        //Read the int on the next line to allocate arrays
        final int n = 8;

        //Allocate arrays with length n
        hits = new double[n][];

        //Read in the header line of years, parse and copy into yearNum
        String[] yearHeaders = scan.nextLine().split(",");
        final int q = yearHeaders.length;

        //Now read until we run out of lines - put the first in country names and the rest in the
     table
        int c = 0;
        while(scan.hasNext())
        {
            String[] inputArr = scan.nextLine().split(",");
            hits[c] = new double[q];
            for(int i = 0; i < q; i++)
            {
                hits[c][i] = Double.parseDouble(inputArr[i]);
                //screen.println(hits[c][i]);
            }
            c++;
        }
        scan.close();

        /**
         * Find midpoints between hits
         * Find bisectors:
         * If we have two points A and B then the \vec{AB} is defined as ((xB - xA), (yB - yA))
         * vector perpendicular to it is given by (-(yB - yA), (xB - xA))
         */
        double [][] midpoints = new double [7][6];
        double [][] bis = new double [7][6];
        for (int j = 0; j < 6; j++) // loop over particles (columns)
        {
            for (int i = 0; i < 6; i++) // loop over detectors (rows)
            {
                //screen.println(hits[i][j]);
                midpoints[i][j] = (hits[i][j] + hits[i+1][j])/2;
                bis[i][0] = -(hits[i+1][1] - hits[i][1]);
                bis[i][1] = hits[i+1][0] - hits[i][0];
```

```
68
69                  bis[i][2] = -(hits[i+1][3] - hits[i][3]);
70                  bis[i][3] = hits[i+1][2] - hits[i][2];
71
72                  bis[i][4] = -(hits[i+1][5] - hits[i][5]);
73                  bis[i][5] = hits[i+1][4] - hits[i][4];
74              }
75              midpoints[6][j] = (hits[6][j] + hits[0][j])/2;
76
77              bis[6][0] = -(hits[0][1] - hits[6][1]);
78              bis[6][1] = hits[0][0] - hits[6][0];
79
80              bis[6][2] = -(hits[0][3] - hits[6][3]);
81              bis[6][3] = hits[0][2] - hits[6][2];
82
83              bis[6][4] = -(hits[0][5] - hits[6][5]);
84              bis[6][5] = hits[0][4] - hits[6][4];
85          }
86          /**
87           * Find intersections between individual hits
88           * y=mx+C
89           */
90          double [][] m = new double [7][3];
91          double [][] C = new double [7][3];
92          for (int i = 0; i < 6; i++) // loop over detectors
93          {
94              for (int j = 0; j < 3; j++) // loop over particles
95              {
96                  m[i][j] = bis[i][2*j+1] / bis[i][2*j];
97                  C[i][j] = - m[i][j] * midpoints[i][2*j] + midpoints[i][2*j+1];
98              }
99          }
100
101         double [][] Xsol = new double [6][3];
102         double [][] Ysol = new double [6][3];
103         for (int i = 0; i < 6; i++) // loop over detectors
104         {
105             for (int j = 0; j < 3; j++) // loop over particles
106             {
107                 Xsol[i][j] = (C[i+1][j]-C[i][j])/(m[i][j]-m[i+1][j]);
108                 Ysol[i][j] = m[i][j]*Xsol[i][j] + C[i][j];
109             }
110         }
111         /**
112          * Add up solutions and average them out
113          */
114         double [] sumX = {0, 0, 0};
115         double [] sumY = {0, 0, 0};
116         for (int j = 0; j < 3; j++) // part
117         {
118             for (int i = 0; i < 6; i++) // det
119             {
120                 sumX[j] = sumX[j] + Xsol[i][j];
121                 sumY[j] = sumY[j] + Ysol[i][j];
122             }
123         }
124
125         double [] avgX = new double [3];
126         double [] avgY = new double [3];
127         for (int i = 0; i < 3; i++)
128         {
129             avgX[i] = sumX[i]/sumX.length;
130             avgY[i] = sumY[i]/sumY.length;
131         }
132         /**
133          * Calculate standard error on mean
134          */
135         double [] SEx = new double [3];
136         double [] SEy = new double [3];
137         double [] SDsumX = {0, 0, 0};
```

```
        double [] SDsumY = {0, 0, 0};
        for (int j = 0; j < 3; j++) // part
        {
            for (int i = 0; i < 6; i++) // det
            {
                SDsumX[j] = SDsumX[j] + (Xsol[i][j]-avgX[j])*(Xsol[i][j]-avgX[j]);
                screen.println((Xsol[i][j]) + "  " + avgX[j]);
                SDsumY[j] = SDsumY[j] + (Ysol[i][j]-avgY[j])*(Ysol[i][j]-avgY[j]);
            }
            SEx[j] = SDsumX[j]/6;
            SEy[j] = SDsumY[j]/6;

            //screen.println("Particle " + (j+1));
            //screen.println("x_centre = " + (float)avgX[j] + " \u00B1 " + (float)SEx[j]);
            //screen.println("y_centre = " + (float)avgY[j] + " \u00B1 " + (float)SEy[j]);
        }
    }
}
```