Computer Modelling of a Vertex Locator

By Mia Boulter

Abstract

The production of a computer model Vertex Locator in Python is explained. It follows the model used by the LHCb, though its shape can be adapted as desired. It has the capability to model the hits of any particle starting at the origin, including hit efficiency and pixel resolution. It can reconstruct the particle track given the hits, which can be generated here by the program itself or externally, and includes the ability to plot the sensors on a 3D axis so the track can be seen passing through them. In investigating the effect of hit efficiency on pseudorapidity it is found that, with the current model, particles with a pseudorapidity of around -2 are most effected by hit efficiency.

1 Introduction

Particle detectors measure the passage of various particles using what are effectively sub-detectors. One such sub-detector is the vertex locator (VELO) of the LHCb detector of the Large Hadron Collider (LHC). It is positioned close to the collision point and offers a high resolution for tracking the resultant particles. The latest upgrade to the VELO as of January 2020 replaces the sensors within to be pixel detectors rather than strip detectors. The goal of this investigation is to produce a computer model of the new VELO which is able to model various complicating effects such as hit efficiency and resolution, and to be able to reconstruct a particle track from the hits it produces. The Python code produced is explained in section 3, and in section 4 some products of the code are shown as well as discussing some effects it can show.

2 Background

The VELO is close to the interaction point so that it can make precise measurements of particle tracks. There are L-shaped silicon pixel sensors that are 0.2mm thick, surrounded by radio frequency (RF) foil. The RF foil protects the sensors while affecting the particles passing through as little as possible. The radio foil makes a large contribution to multiple scattering [3], where particles scatter from atoms in the materials in their path, making it such that tracks are not straight lines.

The right and left sensors are rotationally symmetric around the z-axis, which is defined as the beam axis, and the two sides can separate while the beam is being stabilised. The particles produced will vary in momentum magnitude, angle of azimuth, and pseudorapidity. The pseudorapidity of a particle is defined in terms of the angle, theta, from the z-axis (also sometimes referred to as the longitudinal axis) as

$$\eta \equiv -\ln\left(\tan\left(\frac{\theta}{2}\right)\right)$$

The sensors are positioned on the z-axis in the positive direction to cover a range of pseudorapidity values of 2 to 5 [2].

3 Code Structure and Algorithms

3.1 Modelling a VELO

When modelling a VELO Sensor, the main functionality it needs to have is the ability to check if a particle is within its three-dimensional volume. For this I took a bottom up approach for creating a prism.

Within the shapes file I started by creating class called *Point*, this was to provide the position of a particle in Euclidean space. There are also some methods to allow various operations as necessary for manipulation of points, though they are easily converted to NumPy arrays when necessary since they inherit from *list*. After that I started looking at creating the face of the prism in the x-y plane. There is the *Rectangle* class; a basic shape that can be used to build a *CompositeShape*. A *Rectangle* has a *__contains__* method implemented so that one can check whether a *Point* is within its boundaries using the *in* keyword. A *CompositeShape* can be constructed from any number of shapes and its *__contains__* method will return *True* if the *Point* is within any of its sub-shapes. From here I created a Prism class which introduces the z-dimension. It is composed by an *xy_face*, which can be any two-dimensional shape, and some z-limits. If a *Point* is within both the *xy_face* and the z limits, then it is within the *Prism*.

The *VELOSensor* class inherits from *Prism*. It has two class methods which automatically construct sensors of the left or right specifications.

The z-limits of the sensor are 0.1mm either side of the sensor's z-position.

There is also a static method that generates a sensor identifier (*sid*) that is helpful in that gives a name identifier to a sensor.

A VELO is composed of velo sensors at specific positions. To allow versatile positioning of sensors I allowed the user to pass in the z-positions of left and right sensors to the constructor or use the class method *standard* to build it to the actual VELO specifications automatically.

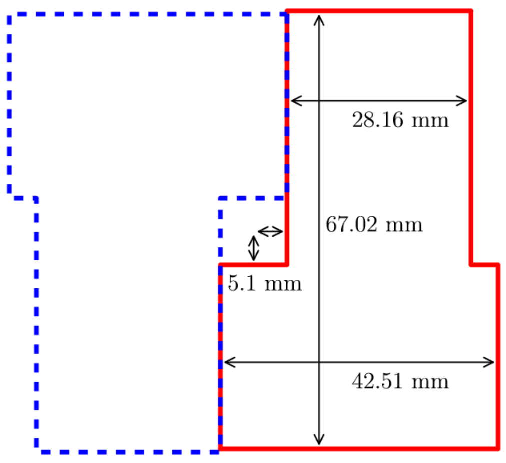


Figure 1 - Dimensions of a VELO Sensor

3.2 Predicting the Hits of a Particle

For this I created a Particle class. When initialising a Particle, you pass in the momentum four vector and initial position. It also has both a *point* attribute and an *init_point* so that it can be moved in space while keeping the initial point stored separately.

Here I encountered the challenge of how to approach seeing whether the particle hits the sensor. I came up with three approaches:

3.2.1 Method 1 – Advance by small increments

From the start position of the particle I advanced the particle by small distance increments, where the proportion of x, y and z was dependent on the proportion of the particle's momentum. After each advancement I check if the particle is within a sensor. I defined an overall bounding box of the velo and limited the particle to starting within the bounding box. This way if the particle moves out of the bounding box you know for certain that it will not re-enter and can break from the advancement loop. After the loop was broken, since the distance increment can theoretically me made infinitely small, I took an average of the times the particle was inside a particular sensor in order to get a single value to be counted as a hit.

Though this method made it very simple to start the particle from any position within the bounding box of the VELO, it proved to be far too slow since you check every sensor after every increment. For purposes that required many particles to be processed it did not suffice – a much faster algorithm was necessary.

3.2.2 Method 2 – Jumping between sensor z-positions

To vastly increase the speed, I had to avoid as many unnecessary calculations as possible. I decided to calculate how far to advance the particle in the z-direction to move from sensor to sensor with nothing in-between. Using this z-distance, one can calculate the corresponding x-distance and y-distance using momentum proportions of the particle.

This method required the user to take care so as not to make the particle move backwards in time from its initial position, but it proved to be extremely fast as expected by how many fewer calculations had to be done. But here I noticed a problem: if a particle passes through the sensor, but it's incoming angle is such that it isn't within the sensor when it is at the central z-position of the sensor, then a hit is not recorded. It can miss possible hits.

3.2.3 Method 3 – Jumping between sensor z-limits

To address the problems of method two I noted that the particle is extremely unlikely to not pass through both the z-minimum and the z-maximum of the sensor, and so if I instead check for hits at those points then almost all possible hits will be recorded. This would only have to do twice as many checks as method 2, and only when it was necessary since the second check could be skipped if the first was successful.

3.2.4 Decision and Implementation

Since method 1 was far too slow, I decided to test the difference between method 2 and 3 to see if method 3 was worth the small extra processing time.

The following is part of a figure that will be discussed later, but I am showing it here to validate my choice of method.

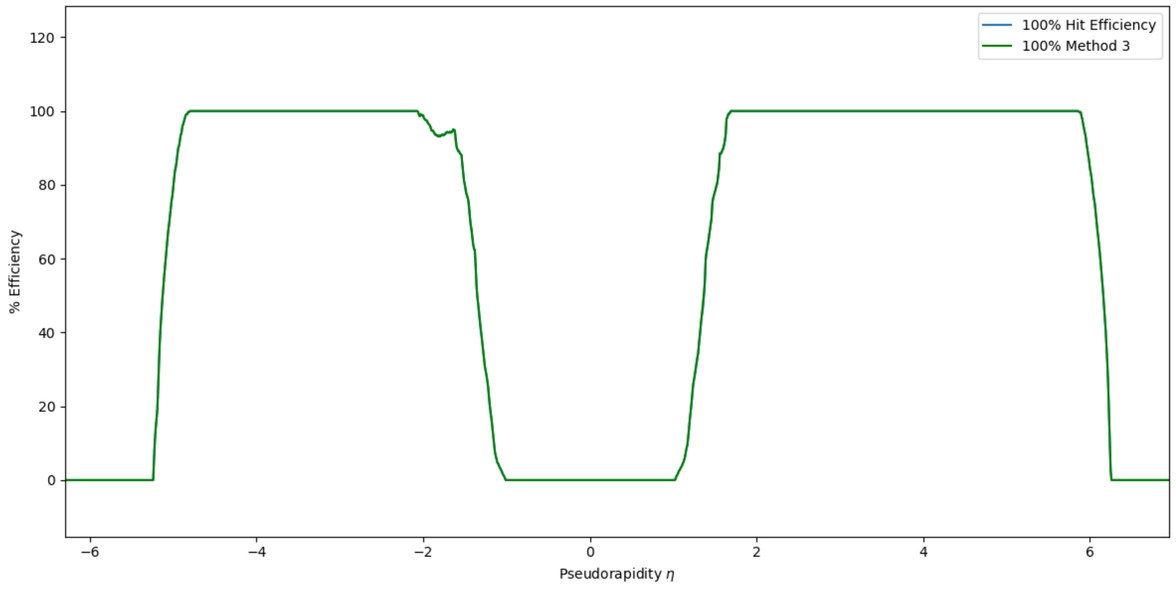


Figure 2 - Comparison of Methods 2 (blue) and 3 (green) showing that their difference is negligible.

The difference between the blue line of method 2 and the green of method 3 is so insignificant that one has to zoom in very close to see the difference, and that difference only occurs at the not-flat sections.

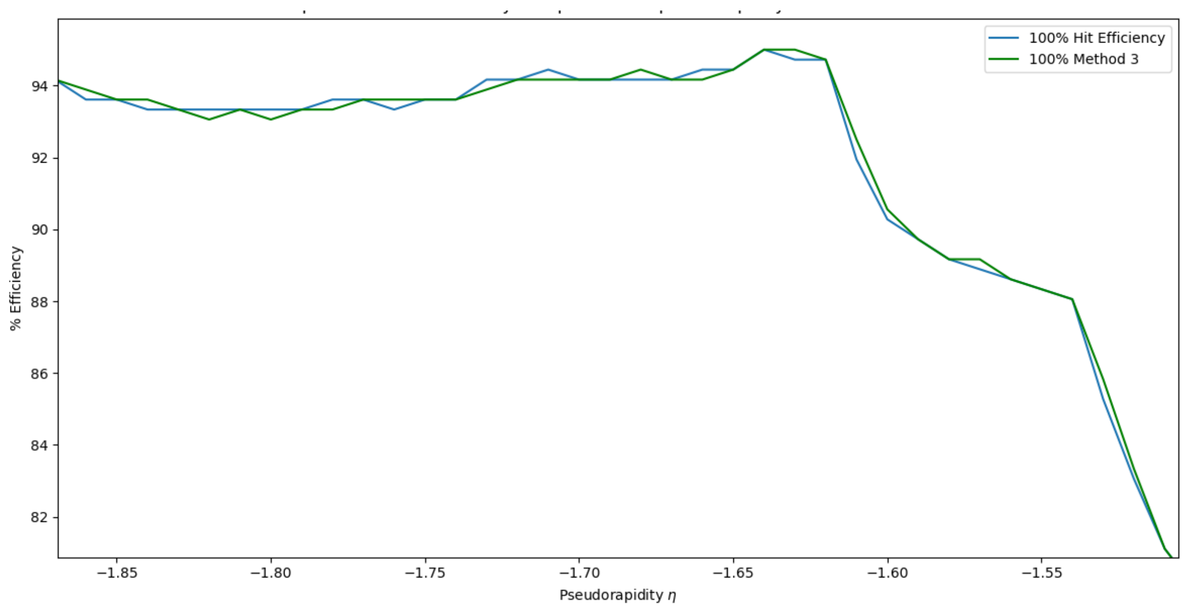


Figure 3 - Close up version of the previous figure showing that there is a small difference.

Method 3 offers negligible benefit and so I will use Method 2 in calculations moving forward.

3.3 Finding How Pseudorapidity Influences Efficiency

The reconstruction efficiency of a detector is the ratio of reconstructed particles to the number of particles that pass through the detector in total. Here reconstructed means that there are clusters associated to it on at least three sensors.

But to truly model efficiency there are of course more factors. To the *VELO hits* method I also added an option for hit efficiency. This allows the user to model a percentage hit efficiency, where only a certain percentage of theoretical hits are actually detected by the sensor. In reality this hit efficiency is around 98%.

The first step in finding the relationship to pseudorapidity was to be able to generate a particle from its pseudorapidity. Since the current model uses cartesian coordinates, the class method of *FourVector* called *from_pseudo* implements this by the formulae which link pseudorapidity η and angle of azimuth ϕ to cartesian coordinates.

$$\boldsymbol{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = p_T \begin{pmatrix} \cos\phi \\ \sin\phi \\ \sinh\eta \end{pmatrix}$$

$$p_T = \frac{|\boldsymbol{p}|}{\cosh\eta}$$

Ultimately the magnitude of the momentum doesn't matter when modelling efficiency since it doesn't have an impact on whether the particle is detected or not.

To find the relationship between pseudorapidity and efficiency I would need to make a method that calculates the efficiency for a particular pseudorapidity, then use that for a range of pseudorapidities and plot the result.

For a given pseudorapidity, the method of *VELO* called *efficiency_from_pseudorapidity* conducts a trial for a range of angle of azimuth, something the user would likely have as the default full 360 degrees cycle. It returns the calculated efficiency for that value. *efficiency_by_pseudorapidity* uses this method for a range of pseudorapidities and returns all values ready for comparisons to be plotted.

3.4 Simulating Hit Resolution and Reconstructing the Particle Track

The VELO has a certain resolution on hits. The model can simulate this by randomly smearing the hits according to a Gaussian distribution. This can be done using NumPy's *random.normal* which takes a random sample according to Gaussian with the standard deviation you pass to it around the point you pass to it. The figure to the right was generated using it and confirms that it creates the desired distribution. The hits are stored as a *Hits* instance, where *Hits* has convenience methods for applying hits efficiency and random smearing to all hits, with the individual smearing of any particular hit being done by a method of the *Point* class.

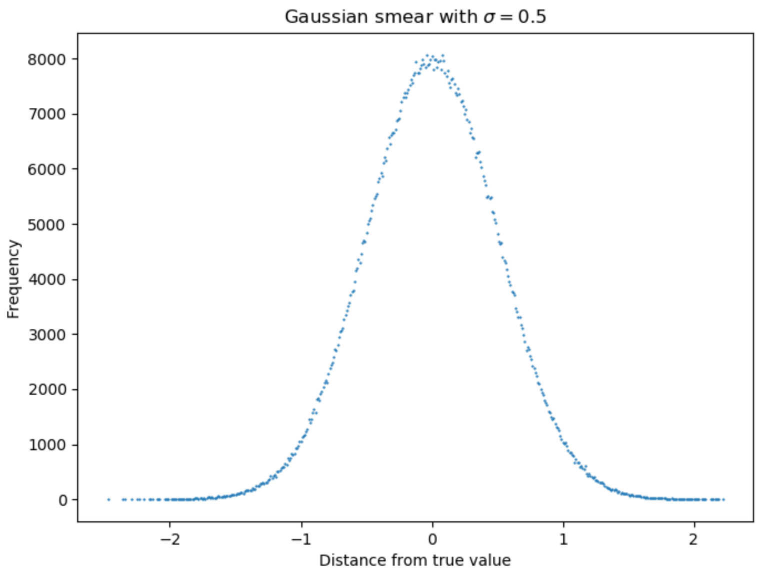


Figure 4 - Demonstration of functionality of NumPy's random.normal. Points are rounded to two decimal places and their frequencies summed.

Once the randomly smeared hits are obtained one may wish to fit them to a line. This can be done using the method *fit* of *Hits*. What this does is centralise the data around the mean and performs a single value decomposition on the data, and the first row of the returned parameter *vh* holds the optimal fit unit vector of the line. The mean of the data is a point on the line, so you have everything you need to reconstruct the particle track. If the total momentum is known, the individual momenta can be obtained by scaling the unit vector appropriately.

To plot the track on a 3D axis is trivial and adding the sensors to the plot can be done using a method included in the *VELOSensor* class called *draw*. You simply pass the 3D axis you are plotting on to the sensor and it will be added to the plot.

4 Demonstration and Results

4.1 Demonstration of Fitting and Plotting of a Particle Track

To the right is an example plot of a fit that is possible using the *fit* method of *Hits* and the *draw* method of *VELOSensor*. The input particle had the momenta (in which only the ratio of momenta matters, and so behaves like a unit vector) of:

$$\boldsymbol{p} = \begin{pmatrix} 0.9 \\ 1.1 \\ 8 \end{pmatrix}$$

Only the sensors that were hit are shown because it becomes cluttered and hard to see with all of them on there. The blue points are the gaussian smeared hit points. The blue line is the best fit line, seeming to go straight through all the points because of how small the smearing is. The origin is marked with a red point.

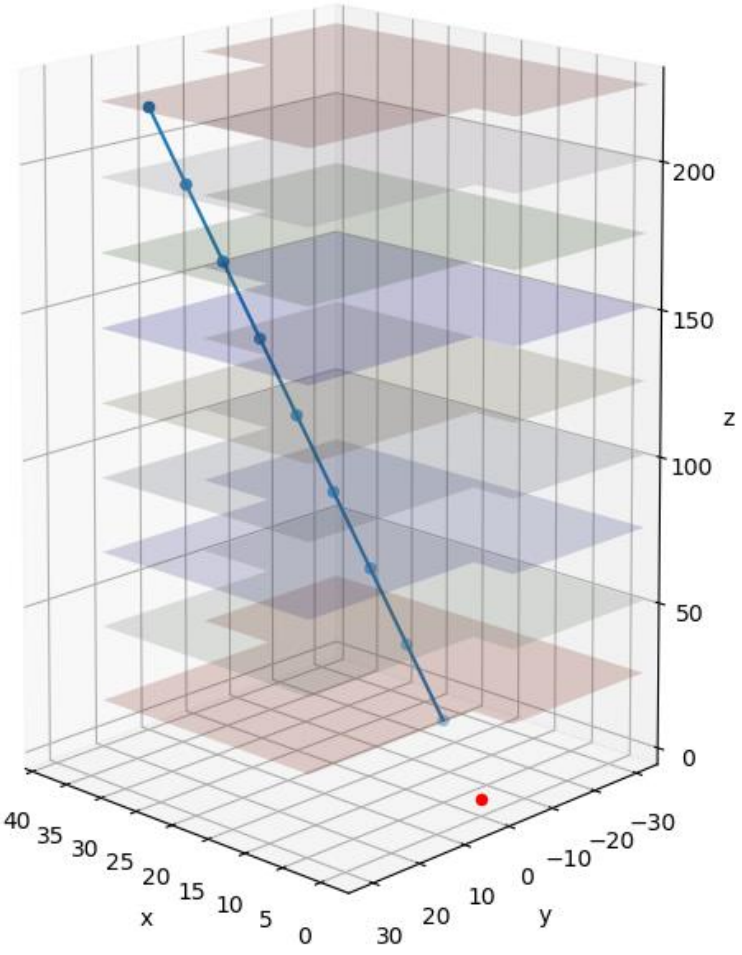


Figure 5 - 3D plot of sensors and hits for a particle with a best fit line.

The values the fit unit vector obtained are

$$\hat{\boldsymbol{u}} = \begin{pmatrix} 0.11074875 \\ 0.13556703 \\ 0.98455893 \end{pmatrix}$$

Where a point on the line, the data mean, is

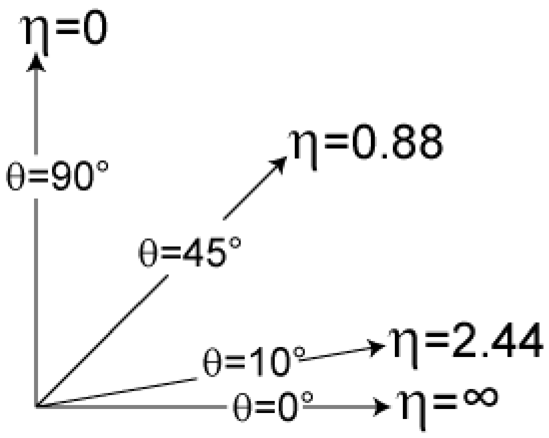
$$\boldsymbol{\mu} = \begin{pmatrix} 14.16937008 \\ 17.32589782 \\ 126. \end{pmatrix}$$

Note that these values will of course be different each time the calculation is done due to the random nature of the gaussian smearing of points. If we scale the momentum to a unit vector, we get

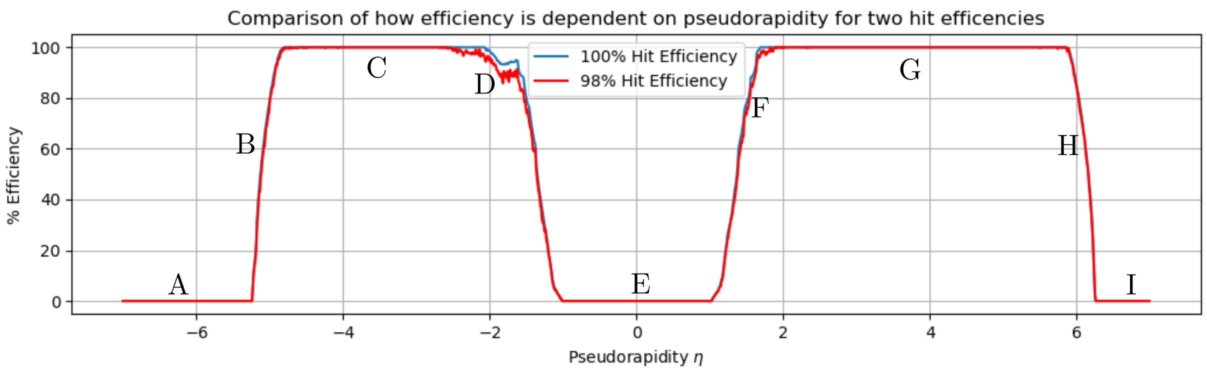
$$\hat{\boldsymbol{p}} = \begin{pmatrix} 0.11076556 \\ 0.13538012 \\ 0.98458275 \end{pmatrix}$$

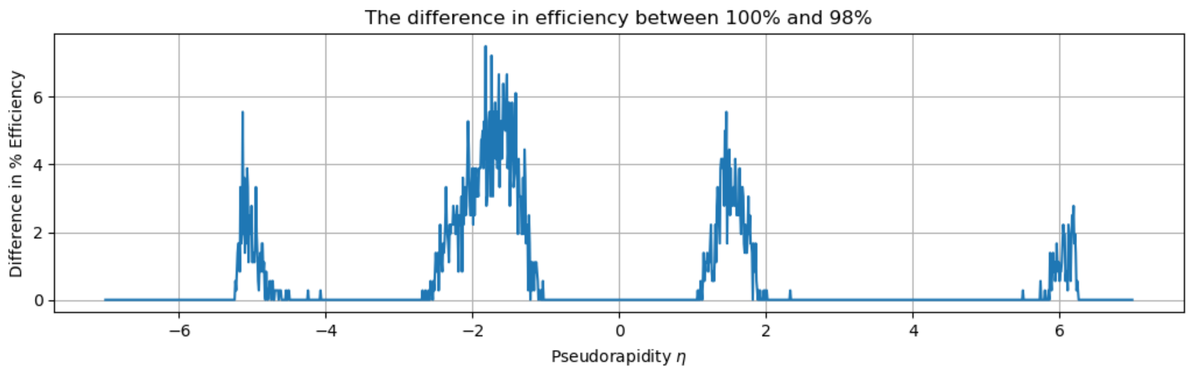
4.2 How Efficiency Varies with Pseudorapidity

Pseudorapidity increases exponentially as the angle theta from the z-axis tends to zero.



After a bit of testing, a pseudorapidity range from -7 to 7 proved suitable to show the characteristics of the relationship. And here is that relationship obtained:





At A and I there are minimum efficiency flat sections where particles move through the hole in the middle of the detectors on the negative-z and positive-z side respectively due to large magnitude longitudinal momenta in comparison to the transverse momenta. Here hit efficiency has no effect because no sensors are being hit An increase in efficiency occurrs around B that gets less fast the further one goes, likely due to the nature of how pseudorapidity varies with theta.

The maximum efficiency sections C and G are where particles pass through most of the detectors and so the chance of getting 3 hits is near 100% – the effect of hit efficiency is negligible. In the positive-z direction around G the range of pseudorapidities if around 2-5 confirming the model behaves in accordance with the design.

Around D is the area which seems most affected by the decreased hit efficiency there must be many instances where just three sensors are hit, this makes it so that the reduced hit efficiency has a high chance to make the particle not reconstructible.

In the middle there is a flat section which is where the particle passes out of the side of the detector because of a large transverse momentum, and so no hits are recorded. Then there is F which is similar to D but less effected by hit efficiency. The negative end is more affected by hit efficiency, and the areas where the particle passes near through the middle of sensors is less affected.

H is not symmetric with the B. It occurrs around 6, while B occurrs around 5. This is because there are sensors up to a distance much further away in the positive direction and so the particle must be much closer to the z-axis, i.e. have a smaller theta value, in order to not hit three sensors.

5 Conclusion

The detector is clearly designed with a focus on particles travelling in the positive z-direction. But in fact, the reason that the negative-z sensors aren't as far away as one may want them to be is due to a limitation in the size of the RF box [1 (Page 22)].

This model is functioning well, but it is still missing many complicating factors. There is multiple scattering which could be implemented by randomly adjusting the momentum after each hit by a small amount, though including the RF foil would make this rather complicated. Another effect that may be considered and there is also bremsstrahlung, which is where a photon can be released when the charged particles accelerate upon interacting with other charged particles they encounter as they pass through the VELO. Adding those effects would take the model much closer to reality.

References

[1] The LHCb Collaboration, *LHCb VELO Upgrade Technical Design Report*, November 29 2013, CERN/LHCC 2013-021 LHCb TDR 13

[2] T. Head, *The silicon vertex locator for the LHCb upgrade*; *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 765*, 2014, Pages 244-246, ISSN 0168-9002

[3] S. E. Richards, *Characterisation of silicon detectors for the LHCb Vertex Locator Upgrade*, University of Bristol, December 2017, https://cds.cern.ch/record/2626889