university of groningen / van swinderen institute

Bachelor Thesis

Development of a Modified RETINA Algorithm for Particle Tracking

July 2019

Author: M. Kami
Supervisor: dr. ir. C.J.G. Onderwater
Second Examiner: dr. J.G. Messchendorp

Abstract

Building upon the concept of a retina algorithm, a modified retina algorithm was developed for efficient particle tracking. A modified retina algorithm was implemented and the performance was tested by conducting simulations. Simulations were first conducted for a single true track and then increased to ten tracks. Beginning from the most simple simulation of a single track with no hit errors and noise, the simulation parameters were altered to test the algorithm. The number of tracks tested in these simulations imply that a modified retina algorithm could be more efficient.

Contents

Chapter 1

Introduction

Particle tracking plays a crucial role in analysing results from high energy particle physics' experiments. For example, the LHCb (Large Hadron Collider beauty) experiment collides protons and the resulting secondary particles are studied in order to measure CP violations in the interactions of B mesons [3]. The LHCb experiments could potentially reveal new physics and challenge the Standard Model.
In such large scale experiments, a great number of particles are produced. If the particle has enough momentum, it will travel through the detector and form a trajectory. The detector will not be able to follow the whole trajectory, instead it will record a spatial coordinate at a certain time on a detector layer. Therefore, it is required to connect a recorded signal for one detector layer to a signal in another layer which belongs to the same particle to reconstruct a particle track. Reconstruction for hundreds of particles is a computationally expensive task. The retina algorithm was developed in order to make particle tracking more efficient. However, a retina algorithm may still have room for improvement as it relies upon a brute force method. An optimised retina algorithm could be developed in order to improve computational speeds. The optimised retina algorithm will be implemented using C++ and the validity of the algorithm will be tested by simulations. Thus, the research question will be can a retina algorithm be modified to increase the efficiency of particle tracking?
The paper will start with a general explanation of particle tracking and important definitions. As the goal is to improve the retina algorithm, the original concept of the algorithm will be explained first and then the theory of the modified retina algorithm will follow. Next, an explanation of the implementation of the retina algorithm and implementation results will be provided and discussed.

Chapter 2

Particle tracking

In this section, the LHCb detector will be used as an example of a particle tracking detector to explain the working principle of a particle tracking detector. Also, important definitions and assumptions taken in this report will be explained.

The LHCb experiment aims to identify particles resulting from proton collisions, more specifically b meson particles which contain b and anti-b quarks and the decay products. At the production vertex, the protons collide and the resulting particles propagate in line with the beam pipe of the detector which is 20 meters long. There are a series of sub-detectors within the LHCb detector with different roles. The sub-detectors collects information such as the trajectory, momentum and energy to identify the produced particles [2]. Figure 2.1 shows particle tracks of an event seen from above. The detector includes a powerful magnet which causes the charged particles trajectories to deflect so the charge can be distinguished. The momentum can also be determined by the curvature of the trajectory which is also required for particle identification.

The curvature of the particle trajectory must be taken into account when reconstructing the particle tracks. However, in this report the simulation model will be based on the VELO (VErtex LOcator) detector which is sufficiently far from the magnetic field so that straight tracks can be assumed. Figure 2.2 shows a close up view around the collision vertex where the tracks are not yet fully deflected by the magnetic field so the use of a simpler model for straight particle tracks is possible.

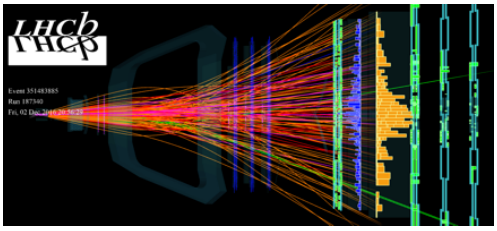

Figure 2.1: Particle tracks of an event

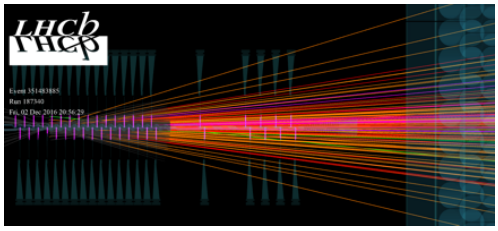

Figure 2.2: A close up view around the collision vertex

Particle tracking detectors consists of layers of detectors which records a spatial coordinate when a particle passes through it. This spatial coordinate is defined

as a hit. Figure 2.3 shows the hits on the detector layers. A sequence of hits that belongs to the same particle is called a track as shown in Figure 2.4. Generally, a minimum of five consecutive hits is considered a valid track worth investigating. If there are less than five, the particle probably did not have enough momentum which is an uninteresting case. The detector layer also includes noise which can complicate the reconstruction process.

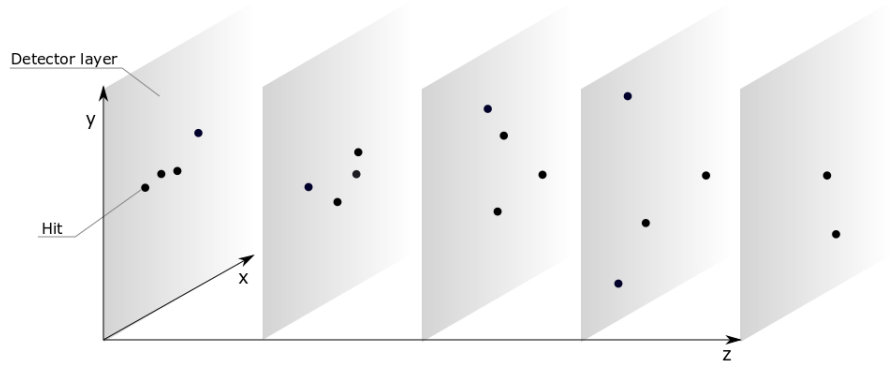

Figure 2.3: Particle tracking detector

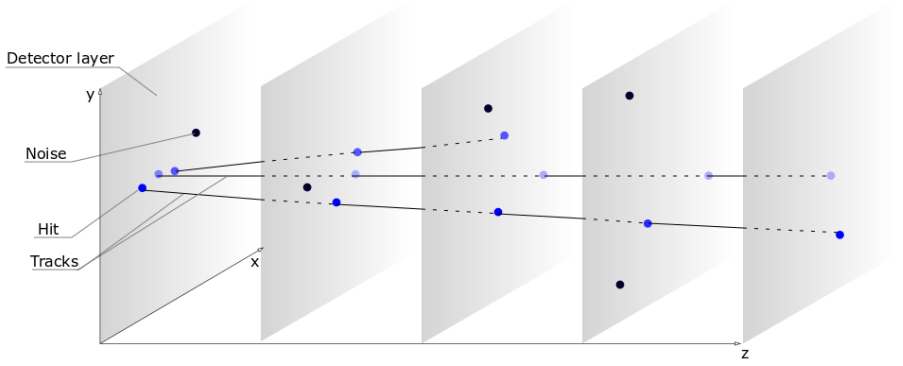

Figure 2.4: Particle tracking detector with reconstructed tracks

Only a collection of hits per layer is available as data, which is insufficient to determine the trajectories of the particles. Further analysis is required to connect the hits in each layer that belong to the same particle. This could be a simple task for a few particles but when there are more particles, the reconstruction complexity increases. For example, the LHCb VELO detector records hundreds of particles per layer from the proton collision. The reconstruction process is a computationally heavy task and due to limitations in data storage, an efficient algorithm is needed.

Chapter 3

Retina Algorithm

A retina algorithm was inspired by the mechanism of a mammals retina. Neurons react to stimuli of certain shapes on the retina and the response is proportional to how close the stimulus is to that shape [1]. Therefore, the retina exhibits properties of pattern matching as the neurons which is associated with a pattern is matched with the stimulus shape.

A retina algorithm can be easily explained with a simple 2D case, where there is no magnetic field so we can assume straight tracks. Since the tracks are straight they can be described by a linear function with two parameters,

$$x = \alpha z + \beta.$$

From this, a parameter space of α and β can be defined where each point in the parameter space corresponds to a possible track in detector space. This parameter space can then be discretized to allow for a pattern matching method since a finite number of possible tracks are created. These possible tracks will be called test tracks for the rest of this report. Figure 3.1 illustrates the parameter space and the two tracks in the detector space is represented in the parameter space as two red dots. The algorithm will go through each of the points and then compute a response function. The discretized points in parameter space of α and β is indexed by i and j respectively. Then the response function is defined to be,

$$R_{ij} = \sum_{d,n} \exp\left(-(x_{d,n} - \langle x_d^{ij} \rangle)^2 / 2\sigma^2 \right). \tag{3.1}$$

where n denotes the n-th hit in detector layer d and $\langle x_d^{ij} \rangle = \alpha_i z_d + \beta_j$. The standard deviation σ comes from the errors in the hits and detector parameters [1].

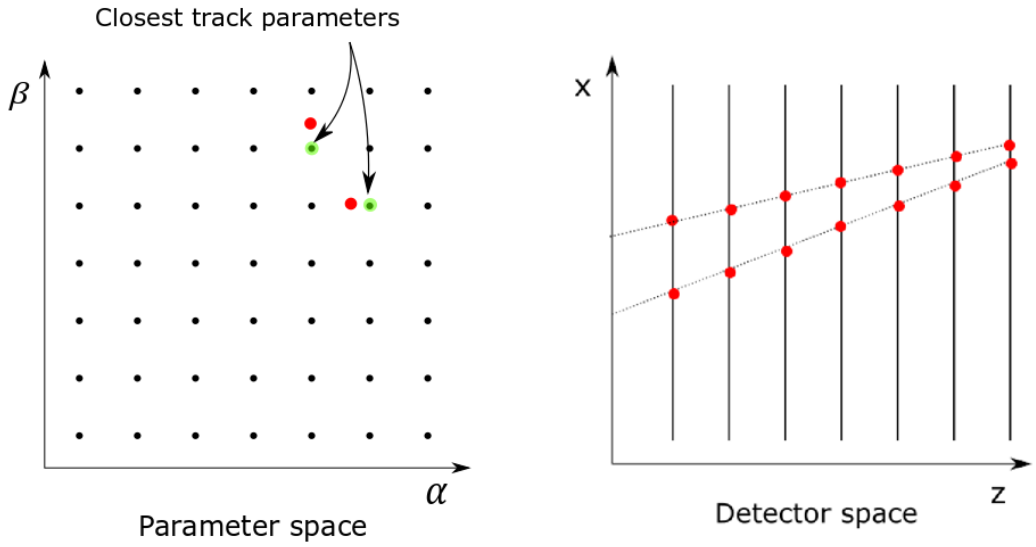

Figure 3.1: Discretized parameter space

By computing the response function for all the discretized points in the parameter space, the algorithm will be able to pick out the track parameters which are the closest to the true track parameters. Then a fit can be performed with the hits to give a better estimation of the track parameters.

However, the number of points in this parameter space must be large in order to give an accurate prediction of the track parameters. This leads to increased computations which decreases the efficiency of the algorithm. There is no need to compute the response function for all the discretized points in parameter space if the area where the hits are concentrated is known. This is especially relevant for the LHCb experiment as the particles produced from the proton collisions come from a production vertex which means that the hits are not spread out in the entire parameter space but concentrated in one area.

Chapter 4

Modified Retina Algorithm

In a retina algorithm, you are required to check through all the discretized points in parameter space and compute the response function. This algorithm quickly becomes inefficient in high energy particle physics experiments where the hit density is very high. When the hit density is high, a higher number of discretized points in parameter space are required. This will rapidly increase the number of response calculations which leads to more calculations in areas where there are no hits.

A modified retina algorithm is an attempt to increase the efficiency by not checking the entire parameter space and only focusing on hit dense regions thus decreasing the number of unnecessary computations.
A 'search cone' is defined to investigate hit dense regions. The search cone is defined around a certain test track by giving the parameters a distribution which would be the width of the prediction. This implies that $x(z)$ will also exhibit a normal distribution defining the search cone. A 2D representation of a search cone is defined as:

$$p(x) = f(\alpha)\ z + g(\beta).$$

with $f(\alpha)$ and $g(\beta)$ uncorrelated normal distributions:

$$f(\alpha) = \exp\Big(-(\alpha-\alpha_0)^2/2\sigma_\alpha^2\Big),$$

$$g(\beta) = \exp\Big(-(\beta-\beta_0)^2/2\sigma_\beta^2\Big),$$

thus *p(x)* is also normally distributed:

$$p(x) = \exp\Big(-(x-x_0)^2/2\sigma_x^2\Big) \tag{4.1}$$

where:

$$x_0(z) = \alpha_0\ z + \beta_0, \quad \sigma_x(z) = \sqrt{(\sigma_\alpha z)^2 + \sigma_\beta^2}$$

In order to quantify how hit dense it is within a search cone, a weight function is computed as:

$$W(x,z) = \sum_{d,n} \exp\left(-\frac{S(x_{d,n}, z_d)^2}{2\sigma_x^2}\right)$$

$$S(x_{d,n}, z_d) = x_{d,n} - x_0(z_d)$$

where n denotes the n-th hit on detector layer d.

When an n-th hit is exactly on the test track the n-th term in of the weight function will compute a 1 as the exponent will be zero. The search cone is defined to be large at first from a certain position so that it looks at large chunks of the detector space. This is illustrated in the parameter space shown in Figure 4.1. The position of the first test track is adjustable, however, it seems reasonable to start in the middle of the parameter space which is set to be (α,β) = (0,0) in the following implementation.

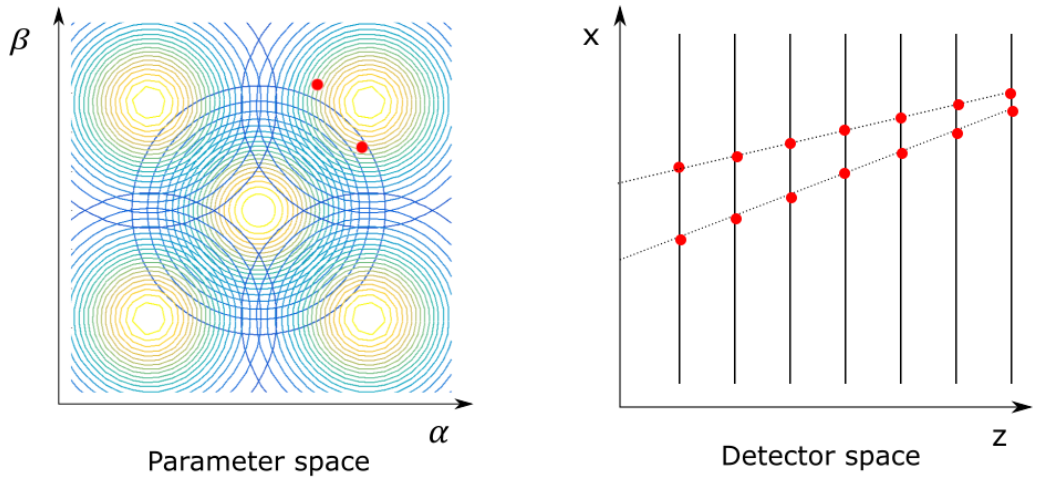

Figure 4.1: Search cones in parameter space

For each search cone a weight will be calculated to determine if the area is hit dense. Then by recursively halving the standard deviation σ_x which is referred to as sigma for the rest of this report, only the hit dense areas are further investigated. The weight function will be computed with a new sigma and new test tracks which comes from shifting the original test track. In theory as a search cone is shifted closer to a true track, the weight will become closer to the value of the number of detector layers as the weight will be 1 for when a hit is exactly on the true track which is summed over the detector layers.

The test tracks are shifted by a step size Δ which will scale the new sigma $\sigma^{'}$ which is half the previous sigma.

$$\sigma^{'} = \frac{\sigma}{2}$$
$$\Delta = k\,2\sigma^{'} \tag{4.2}$$

where k is an adjustable constant. The new test track parameters will be

$$\beta_0^{'} = \beta_0 \pm \Delta,$$

$$\alpha_0^{'} = \alpha_0 \pm \Delta.$$

Figure 4.2 shows a 1D representation of how a search cone is shifted by a step size, depending on k. The optimal k parameter for the simulations in this report will be determined by comparing the results for different k parameters, shown later in the report.

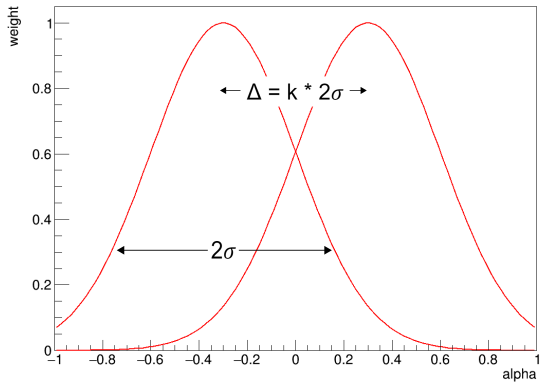

Figure 4.2: 1D representation of a shifted search cone

Chapter 5

Implementation

In this section, an explanation of the implementation of a modified retina algorithm is provided. First, the toy event generation is explained along with some assumptions taken for the simulations. In the next pattern recognition section, the pseudo-code of the algorithm is provided to explain how a modified retina algorithm is implemented. Finally, an explanation is given on how the performance of the algorithm was evaluated in this paper.

5.1 Toy event generation

The simulation model will be based on a simplified version of LHCb VELO detector. In reality, the particle tracking is conducted in 3D which means there would be five parameters to define a track. However, in this implementation 2D tracks are generated for simplicity where only two parameters are required. If the algorithm is successful in 2D it may be extended to a 3D version.

First, a toy event is generated to simulate secondary particle trajectories resulting from proton collisions. Parameters α and β are randomly selected between a certain range of values to create a set of tracks from a production vertex. These tracks will be called the true tracks. Then the hits can be generated for each layer based on these true tracks. In the simulations, the hits are produced exactly on the true track although in reality, the hits may not be detected along an exact straight line due to the fact that the pixel size limits the accuracy. In high energy particle physics experiments, an event consist of hundreds of tracks. The signals from these tracks can include noise and some particles may not have enough momentum to travel through all the detector layers. Even if the particles goes through the detector layer it may not record a signal. Many situations that deviate the hits from the true track are possible and may influence the tracking quality significantly. At first, a simple case was simulated with one track with no noise. Then, a simulation with ten true tracks was conducted to check if the program can reconstruct multiple particle tracks. Then, more complications will be added to analyse the limitations of the algorithm.

5.2 Pattern recognition

The pattern recognition step is implemented first by searching for the candidate tracks. Pseudo-code for the search is presented in algorithm 1 and the code for the algorithm is in the appendix.

Algorithm 1: A Modified Retina Algorithm

```
1  Function Search test tracks(F):
2      while sigma > sigma threshold do
3          foreach Number of test tracks qualified do
4              foreach stepsize direction do
                   move test track position to stepsize direction
                   Perform weight calculation
5              end
6              if weight > weight threshold then
                   Update qualified test tracks
7              end
8          end
           sigma = sigma * 0.5
           stepsize = k*2sigma
9      end
10 End Function
```

The minimum sigma of the search cone is set to be the pixel size for the one track simulation with no hit errors. This sigma is an important parameter since it can affect the reconstruction quality. A sigma that is too high imposes a risk of merging two nearby tracks and if it is too low several tracks may end up reconstructing one true track.

There will be a number of qualified test tracks per sigma meaning that they meet the weight threshold. For each of those test tracks, another set of test tracks will be generated by going through the step size direction loop with half of the previous sigma and a smaller step size as defined in the algorithm. Which is to mean a step is taken in four directions from a qualified test track and plus the same test track with a smaller sigma than before.

In the real program, this is implemented by using a queue container to store the qualified test tracks temporarily. The container will include the qualified test tracks and it will pop out a track to calculate the weight function and add new tracks to the container.

5.3 Performance evaluation

After the algorithm has selected the candidate tracks, they will be filtered out further according to weight and sigma restrictions. Then the difference between

the final test track parameters and the true track parameters will be taken and if the difference is smaller than the reconstruction criteria it will be considered reconstructed. For simulations which involves a set of many true tracks, the reconstruction rate will be defined as

$$\epsilon_{recon} = \frac{T_{recon}}{T_{true}}$$

Where T_{recon} is the number of reconstructed tracks and T_{true} is the total amount of true tracks. The reconstruction rate differ for different simulations as the tracks are generated randomly so an average of ten simulations will be taken in order to obtain the reconstruction rate.

By looking at the number of total test tracks the algorithm checked, the efficiency of the algorithm in a simulation can be compared with another simulation. For every test track, a weight calculation is conducted and since the weight calculations are the most computationally heavy part of the algorithm, it will be an indicator of the efficiency of the algorithm. The number of entries will be compared between simulations with different k parameters and weight thresholds. A low number of entries will be considered more efficient compared to the other, only if the reconstruction rate is approximately the same.

The robustness of the algorithm will also be tested by adding noise hits and errors in the hits along the true track. If the reconstruction rate does not change under these circumstances, it will be considered robust against the added noise.

Chapter 6

Implementation result

6.1 Single track reconstruction

Before introducing many true tracks to the simulation, the algorithm is tested for only one true track and with no hit errors so the hits are exactly on the true track. The true track is generated according to the pixel size. Also no noise were introduced. The maximum weight for one track is 10 since the maximum weight per layer is 1 and there are ten layers. Since a track is defined as at least five consecutive hits, the minimum weight threshold is 5. The initial sigma value was set to be 1. The detector length is 0.8 m and the pixel size is 10^{-4} which is common throughout the simulations.

The track was reconstructed in all the simulations where the k parameter and the weight threshold was varied. The track was reconstructed with an accuracy of 10^{-3}. Figure 6.1 and Figure 6.2 shows the true tracks and the test tracks plotted with the true track for different k parameters and weight thresholds.

Initially, all the test tracks that were higher than the weight threshold were admitted to the queue for the next test tracks to be investigated. However, there were many duplicates of the test tracks with the same sigma value. This could be due to the fact that different search cones are competing too close in to the true tracks. For example, a test track could be in the middle of two search cones and they will both take the same size steps towards the tracks. So it is possible that one search cone goes past the true track and end up in the same position as the other search cone. Also, the weight function has a circular symmetry so if you take the same size steps towards the true track and past it will compute the same weight.

This will happen more if the weight threshold is lower as there will be more accepted test tracks around the true track. Also, there seems to be a negative correlation between the amount of duplicates and the k parameter. A lower k parameter means that the search cones will overlap more, hence a higher possibility that the true tracks is sitting in the overlapping region. Needless to say, these duplicates will only end in redundant weight computations so the algorithm was revised to check for duplicates and omit them. This was done by first putting the alpha and beta parameters in a pair container which associates

the track parameters to each other as both of the values must match another set of track parameters. The pairs of track parameters were then added to a set container which only allows for unique entries to check if there are any duplicates.

The initial position of the test track, set in the simulation to be $\alpha = 0$ and $\beta = 0$, also had an influence on the results at this point. The true tracks that were generated close to the initial position of the test track were reconstructed very accurately. However, if you move the true track away from the initial position enough, the reconstruction was not achieved. As you move further away from the initial position of the track, the initial weight will be very low since there is only one track. If this happens the weight threshold must be low enough such that the search cone that contains the true track is selected for further investigation.

A low weight threshold may be beneficial in the beginning of the search for the above reasons. Yet, as the sigma of the search cone becomes smaller and if the weight is low, that is an indication that the test track is not an ideal candidate for a true track. As an attempt to remedy this situation, the weight threshold was set low in the beginning and after each sigma round the weight threshold was increased. However, this was not successful. Another attempt was to increase the initial sigma such that the initial weights are high enough to pass the weight threshold. In this way, the weight threshold can be consistent throughout the process. In reality there is no telling where the production vertex will be so it would be important to estimate the possible range of the tracks well and set the sigma such that not only does it cover the whole parameter space but cover the space at a sigma where it will give a weight above the threshold even at the edge of the parameter space where the possible tracks could be. After the initial sigma was changed from 1 to 1.5 then the track was reconstructed again.

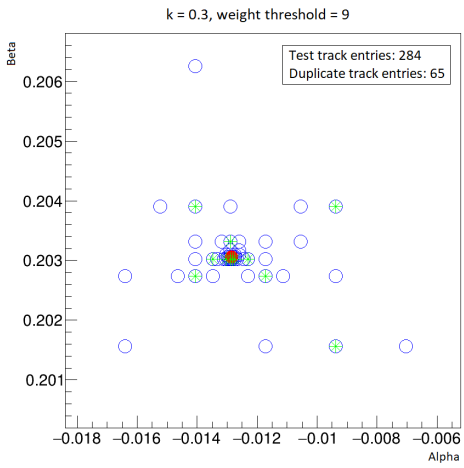

Figure 6.1

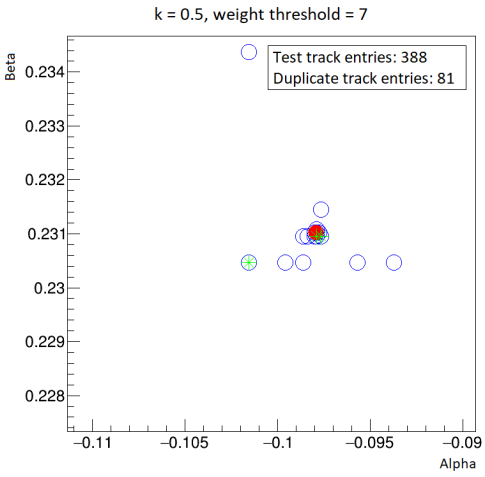

Figure 6.2

After all the adjustments were made the simulation was conducted and Figure 6.3 shows the histogram of the test tracks. The yellow area indicates where the algorithm is searching the most intensely which is indeed where the true track is in the histogram. This histogram indicates that the algorithm is searching only in the region of the true track according to theory. In a retina algorithm,

regardless of whether one track or one hundred tracks needs to be reconstructed, the number of test tracks produced would not change. However, in a modified retina algorithm, the efficiency of the algorithm will depend on other factors such as the number of true tracks. In a retina algorithm, the detector size is 0.8 meters so the dimensions of the parameter space would be 0.8 meters by 0.8 meters and that would result in 640,000 tracks to test if the distance between the tracks are 0.001 m apart. Figure 6.3 shows that the entries are 1243 so there is a huge reduction in the number of tracks being tested.

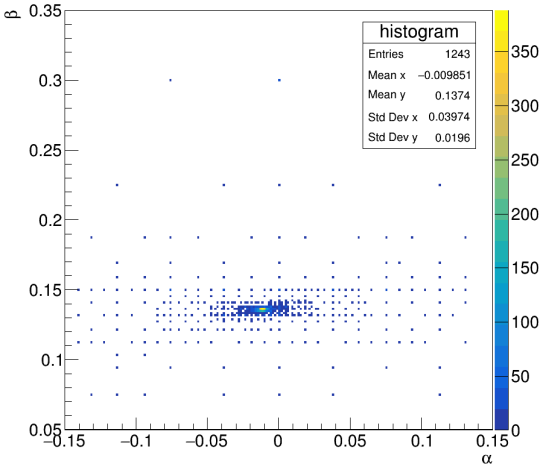

Figure 6.3: Histogram of test tracks

6.2 Multiple track reconstruction

Multiple tracks must also be tested as in reality there are hundreds of tracks to be reconstructed. For multiple tracks the weight value is high at first, for example if you generate 10 tracks the highest possible weight will be 100. In theory, as the sigmas becomes smaller and as more test tracks are generated near a true track, the weight values will become close to 10 since the maximum weight for one track is 10. Therefore, instead of taking the test track with the highest weight to be the candidate track in the case of a single track reconstruction, the test tracks with a weight between 9.9 and 10.1 will be selected. The sigma was also restricted since test tracks that have a high sigma with weights in this range will not be suitable candidates. In all the simulations, the test tracks with a sigma lower than 0.002 was selected.These tracks will be called the candidate tracks where the reconstructed tracks will be selected from.

The weight threshold was set to be 5 which is the minimum weight for one track throughout the process. Since, for this weight threshold, the computations took too long for tracks more than 10, a more hit dense case was simulated by generating the true tracks in a smaller interval.

At first 10 true tracks were uniformly generated with an interval of 0.2 for both α and β where the reconstruction criteria was 10^{-3}. Figure 6.4 shows the optimal k value for this situation. The duplicate count is increased when a track is reconstructed more than once. For k parameters between 0.1 to 0.2, all the true tracks

are reconstructed, however, when increased further than 0.2 the reconstruction number drops. Also, the number of candidates and duplicates decreases sharply as the k parameter increases from 0.1 to 0.2. So for this simulation, the optimal k parameter is 0.2.

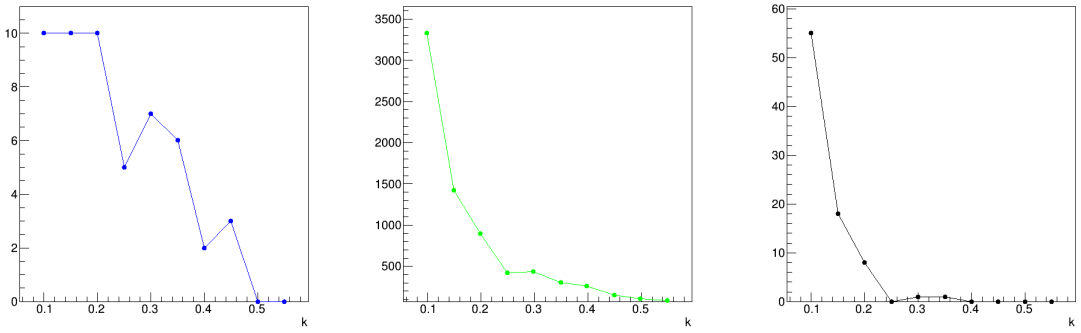

Figure 6.4: k parameter comparison: From the Left, number of reconstructed tracks, number of candidate tracks, and number of duplicate tracks

A histogram was also generated from all the test tracks found in the $k = 0.2$ simulation shown in Figure 6.5. The figure shows that the test tracks are concentrated around the true tracks meaning that the algorithm is not testing further around the less hit dense areas according to the theory. The reconstruction rate was 100 percent and around 48,000 test tracks were produced. As mentioned in the previous single track section, 640,000 tracks would be generated for a retina algorithm which is significantly higher so a higher efficiency for reconstruction may be possible. Figure 6.6 shows the results for a simulation for $k = 0.4$. The number of entries are less than half of the simulation for $k = 0.2$ and computation time is short but the reconstruction rate is 85 percent so the reliability of reconstruction is lower. The rest of the simulations below are conducted with $k = 0.2$.

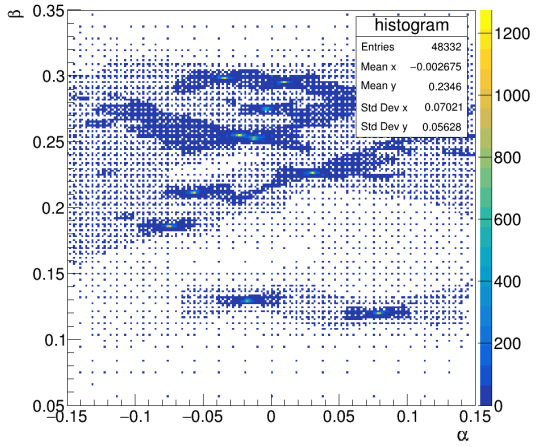

Figure 6.5: Histogram of test tracks for interval 0.2

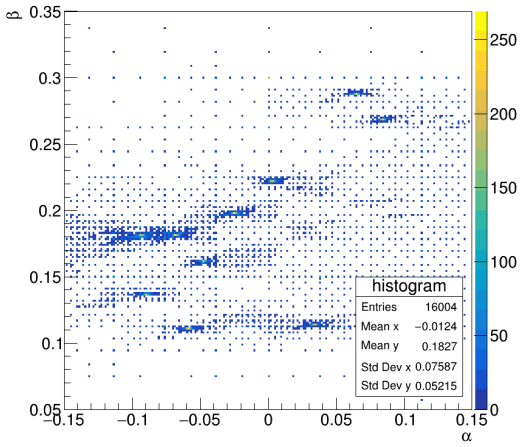

Figure 6.6: Histogram of test tracks for interval 0.4

Next, simulations were conducted for an interval of 0.02. The reconstruction criteria was decreased to 10^{-4} for this interval. The reconstruction rate was

20 percent so a significant decrease was observed for a smaller interval and the number of test tracks in the histogram increased significantly. The algorithm is ignoring large areas of the parameter space, however, there is a pattern appearing where the test tracks are concentrated as can be seen in Figure 6.7. This may be due to the fact that all the true tracks are within one search cone from the beginning of the refining loop in the algorithm such that all the test tracks are qualified and further investigated until a later stage in the loop. The reconstruction criteria was lowered to 10^{-3} which then the reconstruction rate increased to 88 percent. Although the reconstruction rate was lower than the simulation for an interval of 0.2, further analysis such as parameter estimation may be applied to reconstruct the true tracks to a better accuracy.

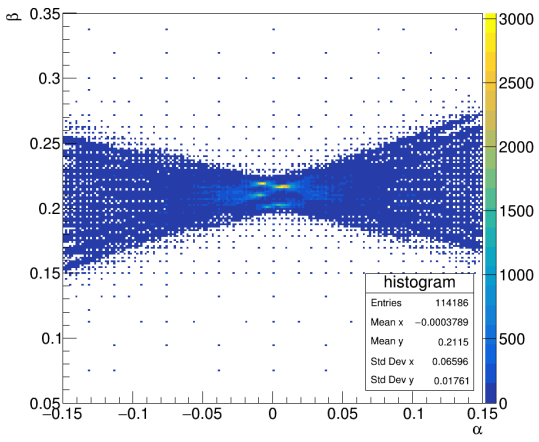

Figure 6.7: Histogram of test tracks for interval 0.02

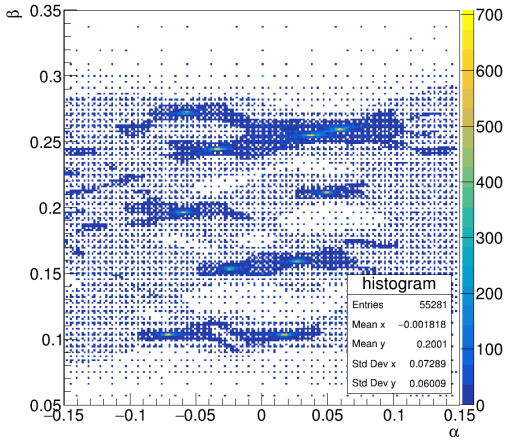

Figure 6.8: Histogram of test tracks for interval 0.2 including noise

In the next simulation, noise and errors in the hits were introduced to the simulations where $k = 0.2$ and the true tracks were produced in an interval of 0.2. First, 10 percent noise was introduced by adding a randomly generated hit in each layer. The reconstruction rate was 97 percent which is a small decrease so the algorithm seems to be robust against noise and hit errors. Figure 6.8 shows that there is a slight increase in the number of entries so the algorithm is testing for more tracks. This is expected as the noise contributes to the weight values and increases the number of qualified test tracks to go through the refining loop. However, when errors in the hits were introduced on top of the noise, the reconstruction rate dropped to 76 percent. Again, this percentage may increase if a fit is performed to better approximate the true track.

A weight threshold of 5 throughout the reconstruction process is guaranteed to include all the test tracks so that they are further investigated. Therefore, if the weight threshold is higher the reconstruction rate would decrease since the search cones containing a true track may be omitted from the refining loop. Figure 6.9 shows a simulation result of when the weight threshold was initially set to 50 and recursively decreased by 5 until it reached a minimum of threshold 5. As expected, the reconstruction rate dropped to 84 percent and number of test tracks were decreased. Thus to ensure that the tracks are reconstructed, the

weight threshold must be set to the minimum.

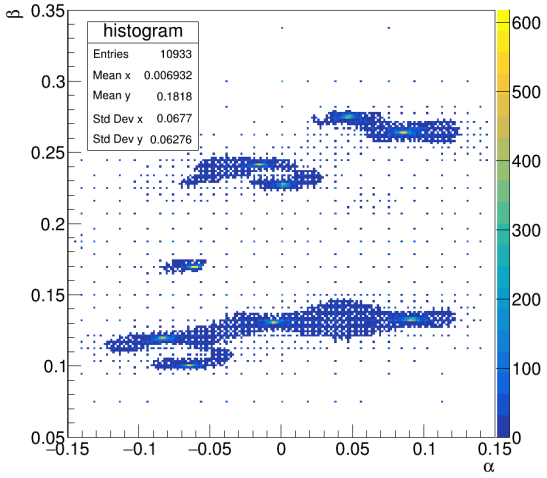

Figure 6.9: Histogram of test tracks for interval 0.2 with modified weight threshold

Chapter 7

Conclusion and Discussion

7.1 Conclusion

The implementation results have shown that the algorithm has narrowed down the number of test tracks, searching only in areas where the tracks were generated. The simulations of a single true track and ten true tracks revealed insights on how to use this algorithm. As long as an optimal k parameter is chosen the number of candidate and duplicate tracks are minimized. For a ten track simulation where the true track parameters were produced within an interval of 0.2, the optimal k parameter was 0.2 and the reconstruction rate was 100 percent. The number of test tracks for a simulation was 48,000 which is far less than the number of test tracks required for a retina algorithm. This implies that a modified retina algorithm could increase the efficiency of a retina algorithm for a 10 track simulation. There were limitations such as a lower reconstruction rate and increased amount of test tracks for when the true tracks were generated in a smaller interval of 0.02 compared to 0.2. The algorithm was robust against noise as the simulations including 10 percent noise resulted in a reconstruction rate of 97 percent. However, error in the hits decreased the reconstruction rate to 76 percent.

7.2 Discussion

This report was limited to simulations of ten true tracks. As the number of true tracks increased, the computation time increased as well. The algorithm slows down when it reaches low sigmas, this is most likely due to there being an increased amount of qualified test tracks which are being added to the queue. One reason for this is that the algorithm calculates the weight and checks if the weight is higher than the threshold and adds the qualified test tracks in a container in one loop. This process could be made more efficient if the tasks are broken down and the weight calculation and the check could be done in parallel. For further research, this could be implemented by applying multi-threading where more than one CPU core can be used to execute tasks. Then, the algorithm could be sped up and more than ten true tracks could be generated and a more realistic LHCb simulation may be possible.

Bibliography

[1] A. Abba and etc, "Simulation and performance of an artificial retina for 40 mhz track reconstruction," *JINST*, vol. 10, no. 03, Sep. 2014. DOI: 10.1088/1748-0221/10/03/C03008.

[2] CERN. (Jun. 2019). The lhcb detector, [Online]. Available: https://lhcb-public.web.cern.ch/lhcb-public/en/Detector/Detector-en.html.

[3] R. M. van der Eijk, "Track reconstruction in the lhcb experiment," PhD thesis, University of Amsterdam, Sep. 2002.

Appendix

```
#include "TRandom3.h"
#include <queue>
#include <memory>
#include <algorithm>
#include <iterator>
#include <cmath>
#include <functional>
using namespace std;
typedef pair<double, double> paird;


class myPaird: public paird
{
    using paird::paird;
    public:
    bool operator<(const myPaird &comp)
    {
        if (this->first < comp.first) return true;
        else if ( (this->first == comp.first) && (this->second <
          ↪  comp.second) ) return true;
        else return false;
    }
};


/*************************************************************/

TRandom3 rnd;

//Detector parameters etc.
int nlayers = 10;             // Number of layers
```

```
31 double detLength = 0.8;        // Detector length [m]
32 int nTracks = 10;              // Number of tracks
33 double stdp = 2;               // Pixel size [m]
34 double pixel = 100e-6;         //Standard deviation-gaussian error
    ↪  in pixels [pixels]
35 double sep = detLength/nlayers;
36
37 /*****************************************************************/
38
39 //function prototypes
40 void run();
41 double calcWeight(TNtupleD *hits, double meana, double meanb,
    ↪  double siga, double sigb);
42 void search(TNtupleD *hits, TNtupleD *testTracks,TNtupleD *dups);
43 void genHits(TNtupleD *tracks, TNtupleD *hits);
44 void genTracks(TNtupleD *tracks);
45 void calcParam(TNtupleD *hits, double& meana, double& meanb);
46 void add(myPaird &x, set<myPaird> &s,queue<myPaird> &que, TNtupleD
    ↪  *dups, double dw);
47 void recon(TNtupleD *testTracks, TNtupleD* tracks, TNtupleD*
    ↪  hits);
48
49 /*****************************************************************/
50
51 void run()
52 {
53     rnd.SetSeed(0);
54
55     TNtupleD *tracks = new TNtupleD("tracks","tracks","a0:b0");
56     TNtupleD *hits = new TNtupleD("hits","hits","x:z");
57     TNtupleD *testTracks = new
        ↪  TNtupleD("testTracks","testTracks","testa:testb:testsiga:testsigb:w");
58     TNtupleD *dups = new TNtupleD("dups","dups","da:db:dw");
59
60     genTracks(tracks);
61     genHits(tracks,hits);
62     search(hits,testTracks,dups);
63     recon(testTracks, tracks,hits);
64
65     TFile* file = new TFile("nodupretinaoneout.root","recreate");
```

```
66      tracks->SetDirectory(file);
67      hits->SetDirectory(file);
68      testTracks->SetDirectory(file);
69      dups->SetDirectory(file);
70      file->Write();
71      file->Close();
72  }
73
74  /*****************************************************************/
75
76  //reconstruction
77  void recon(TNtupleD *testTracks, TNtupleD* tracks,TNtupleD* hits)
78  {
79
80      int entries = testTracks->GetEntries();
81      int trueentries = tracks->GetEntries();
82      double adiff, bdiff, testa, testb, testsiga, w, a0, b0;
83      testTracks->SetBranchAddress("testa",&testa);
84      testTracks->SetBranchAddress("testb",&testb);
85      testTracks->SetBranchAddress("testsiga",&testsiga);
86      testTracks->SetBranchAddress("w",&w);
87      tracks->SetBranchAddress("a0",&a0);
88      tracks->SetBranchAddress("b0",&b0);
89
90      int recon = 0;
91      int cand = 0;
92      int dupl = 0;
93      int count = 0;
94
95      for (int idx2 = 0; idx2 < trueentries; ++idx2)
96      {
97          tracks->GetEntry(idx2);
98          for (int idx = 0; idx < entries; ++idx)
99          {
100             testTracks->GetEntry(idx);
101             if (w < 10.1 && w > 9.9 && testsiga < 0.002)
102             {
103                 ++cand;
104                 adiff = abs(a0-testa);
105                 bdiff = abs(b0-testb);
```

```
106
107                 if (adiff <= 0.001 && bdiff <= 0.001)
108                 {
109                     ++count;
110                 }
111             }
112         }
113
114         if (count > 0)
115         {
116             dupl += count - 1;
117             ++recon;
118         }
119         count = 0;
120     }
121 }
122
123 /*****************************************************************/
124
125 //refinement loop
126 void search(TNtupleD *hits, TNtupleD *testTracks, TNtupleD *dups)
127 {
128     //initial parameters
129     double testsiga = 1.5;          //initial search cone sigma
130     double testsigb = 1.5;
131     myPaird testab(0,0);            //initial test track position
132     double k = 0.2;                 //k parameter
133     double wThreshold = 5;          //Weight threshold
134     double sigThreshold = pixel;    //Sigma threshold
135
136     double stepa = k*2*testsiga;
137     double stepb = k*2*testsigb;
138     double nTestTrack = 5;
139     int entries = 1;
140     double w,testa,testb;
141
142     queue<myPaird> que;
143     que.push(testab);
144     set<myPaird> s;
145     myPaird currentab;
```

```
146     myPaird nextab;
147
148     while (true)
149     {
150         for (int idx = 0; idx < entries; ++idx)
151         {
152             for (int idx2 = 0; idx2 < nTestTrack; ++idx2)
153             {
154                 currentab = que.front();
155                 testa = currentab.first;
156                 testb = currentab.second;
157
158                 switch (idx2)
159                 {
160                     case 1:
161                     testa += stepa;
162                     break;
163
164                     case 2:
165                     testa -= stepa;
166                     break;
167
168                     case 3:
169                     testb -= stepb;
170                     break;
171
172                     case 4:
173                     testb += stepb;
174                     break;
175
176                     default:
177                     break;
178                 }
179
180                 w =
                      ↪  calcWeight(hits,testa,testb,testsiga,testsigb);
181
182                 if ( w >= wThreshold )
183                 {
```

```
184
                        ↪ testTracks->Fill(testa,testb,testsiga,testsigb,w);
185
186                     nextab = {testa,testb};
187                     add(nextab, s, que, dups, w);
188                 }
189             }
190             que.pop();
191         }
192         //if (wThreshold > 5)   wThreshold -= 5 ;
193         entries = que.size();
194         testsiga *= 0.5;        //Halve the sigma
195         testsigb *= 0.5;
196         stepa = k*2*testsiga;  //Change the step size
197         stepb = k*2*testsiga;
198         if (testsigb < sigThreshold) break;
199     }
200 }
201
202 /*****************************************************************/
203
204 //function to filter out duplicate test tracks
205 void add(myPaird &x, set<myPaird> &s,queue<myPaird> &que, TNtupleD
    ↪  *dups, double dw)
206 {
207     double da, db;
208
209     if (s.find(x) == s.end())
210     {
211         que.push(x);
212         s.insert(que.back());
213     }
214     else
215     {
216         da = x.first;
217         db = x.second;
218         dups->Fill(da,db,dw);
219     }
220 }
221
```

```
222 /*****************************************************************/
223
224 //Weight calculation
225 double calcWeight(TNtupleD *hits, double meana, double meanb,
      ↪  double siga, double sigb)
226 {
227     double xm,x,z,sigx;
228     double w = 0;
229     hits->SetBranchAddress("x",&x);
230     hits->SetBranchAddress("z",&z);
231
232     for (int nhit = 0; nhit < hits->GetEntries(); ++nhit)
233     {
234         hits->GetEntry(nhit);
235         sigx = sqrt(pow(siga*z,2) + pow(sigb,2));
236         xm = meana*z + meanb;
237         w += exp(-0.5*pow((x-xm)/(sigx),2));
238     }
239     return w;
240 }
241
242
243 /*****************************************************************/
244
245 //Hit generation
246 void genHits(TNtupleD *tracks, TNtupleD *hits)
247 {
248     double a0,b0,xm,x;
249     tracks->SetBranchAddress("a0",&a0);
250     tracks->SetBranchAddress("b0",&b0);
251     int nTracks = tracks->GetEntries();
252
253     for (int track = 0; track < nTracks; ++track)
254     {
255         tracks->GetEntry(track);
256
257         for (double z = -detLength/2; z <= detLength/2; z += sep)
258         {
259             xm = a0*z + b0;
```

```
260             x = (round(rnd.Gaus(xm/pixel,stdp)))*pixel;    // Hit
                ↪  error
261             x = a0*z + b0;                                 //
                ↪  Without hit error
262             hits->Fill(x,z);
263         }
264     }
265
266     Noise generation
267
268     int nNoise = 1;     // adjustment of amount of noise
269     for (int idx = 0; idx < nNoise; ++idx)
270     {
271         for (double z = 0; z < detLength-sep; z += sep)
272         {
273             x = rnd.Uniform(0.05,0.2);
274             hits->Fill(x,z);
275         }
276     }
277 }
278
279 /*****************************************************************/
280
281 // True track generation
282 void genTracks(TNtupleD *tracks)
283 {
284     for (int idx = 0; idx < nTracks; ++idx)
285     {
286         double a0 = rnd.Uniform(-0.1,0.1);
287         double b0 = rnd.Uniform(0.1,0.3);
288         tracks->Fill(a0,b0);
289     }
290 }
```