Mathematical Finance MSc Dissertation MTH778P, 2020/09

Winding angle distributions for Lorentz Lattice Gases

Nathalie Richards, ID 190564836

Supervisor: Prof Thomas Prellberg

A thesis presented for the degree of Master in Sciences in Financial Computing

School of Mathematical Sciences
Queen Mary University of London

Declaration of original work

This declaration is made on September 10th , 2020.

Student's Declaration: I, Nathalie Richards, hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text, nor has any part been written for me by another person. Referenced text has been flagged by:

1. using quotation marks "...", and
2. explicitly mentioning the source in the text

Abstract

The Lorentz Lattice Gas models a particle on a two dimensional square lattice whose trajectory is modified by scatterers at each lattice point. In this paper, a particular scattering rule known as the mirror model is examined where lattice points are populated with double sided mirrors. The trajectory of particles of N-step walks are probed at different probabilities of left and right mirrors by examining the winding angle distributions. It is confirmed that configurations with probabilities of left and right mirrors equal to one (referred to in this paper as mirror density of one) are compatible with known Gaussian behavior with variance growing asymptotically as C log N. The winding angle distributions where mirror density is equal to one are further probed to reveal interesting fine structure. Distributions are examined with varying steps, N, in order to confirm that behavior seen cannot be attributed to walk length alone. In addition, winding angle distributions of mirror density less than one are examined where Gaussian behavior breaks down. Winding angle distributions are examined by varying mirror probabilities by 0.0001 in order to achieve highly resolved results at interesting points of the sample space.

Contents

Appendix C is a full PDF of the commented code included in the .zip file submitted with program source code.

1 Introduction

1.1 Motivation for this work

1.1.1 Lorentz Gas

The Lorentz gas is a fundamental model arising in the theory of Hamiltonian dynamical systems whereby a particle makes mirror-like reflections from an extended collection of scatterers[1] whose dynamics can be likened to those of mathematical billiards[2]. It was proposed by Lorentz in 1905[3] as a model of electronic motion in a solid.

1.1.2 Lorentz Lattice Gas

Lorentz went one step further with this model by defining these dynamics on a two dimensional lattice. Lorentz modeled the particle as a free electron and its scatterers as fixed atoms at each lattice point. The trajectories of a single particle in a Lorentz Lattice gas are modeled as the particle moves from lattice site to lattice site. When the particle arrives at a lattice site, it encounters a scatterer that modifies its motion according to a given scattering rule [4]. Depending on the scattering rule, each scatterer can also have one of a number of orientations.

In this paper a particular scattering rule called the mirror model is looked at. The lattice is defined as a two dimensional square lattice populated with double-sided mirrors at the lattice sites. The mirrors have diagonal orientations with respect to the origin and are placed randomly at each lattice point reached by the particle with pre-determined probability:

$$P_{Left} + P_{Right} \leqq 1$$

In the notation above and for the remainder of this paper, the probabilities of left mirrors and right mirrors will be referred to as P_{Left} and P_{Right} respectively.

The trajectory of the particle is determined by the fixed positions of the mirrors and changes with the probabilities of right/left mirrors and unoccupied lattice sites (all three of which referred to in this paper as ‘mirror orientations’). The mirror orientations are determined as such by the first arrival of the particle at the lattice point and remain fixed for the remainder of the simulation of each particular walk. Details of how this was implemented are described in the programming section of this paper (*Section 2.1, Appendix A*).

1.1.3 Self Avoiding Trails

Whilst the particle trajectory may close in on itself, forming an orbit, a given bond will never be traversed in the opposite direction so the particle trajectories can be modeled as self-avoiding trails. In contrast to self-avoiding walks, which are site-avoiding lattice paths, self-avoiding trails are edge-avoiding, that is, they do not visit the same edge of the lattice twice[5]. Trails can visit the same lattice point more than once.

This behavior can be used to model the behavior of long chain polymers [6] which cements its relevance in fields such as chemical engineering and in the understanding of proteins and DNA. [7]

1.1.4 Winding Angle distributions

From the trajectories of particles across the lattice, the winding angle is determined. The winding angle distributions for lattice walks have received much theoretical attention since they are a way of measuring subtle changes in particle behavior due to their sensitivity to certain effects within the lattice. The winding angle of a two dimensional random walk which exhibits a Brownian path around a finite winding center has winding angle distributions showing slowing decays at long lengths. As proved by Spitzer,[8] the winding angle distributions of two dimensional random walks are Cauchy distributed. In contrast, simulations of interacting self-avoiding walks show that the winding angle distribution for N-step walks is compatible with the theoretical prediction of a Gaussian with a variance growing asymptotically as $C \log N$ [5] . The winding angle distributions are sensitive to the details of a model so are very good at probing the behavior of trajectories. In this paper, the winding angle distributions enabled probing of underlying structure that would not have been picked up using a different trajectory metric such as displacement from the origin.

1.2 Content of the thesis

1.2.1 Method

Figure 1 shows the probability space sampled with respect to the mirror orientation. The method of simulating particle trajectories (such as those seen in F*igures 2* and *3)* and determining winding angles are detailed in the method section of this thesis. The design and implementation of the program is detailed in *Appendix A* with well commented code in *Appendix B*.

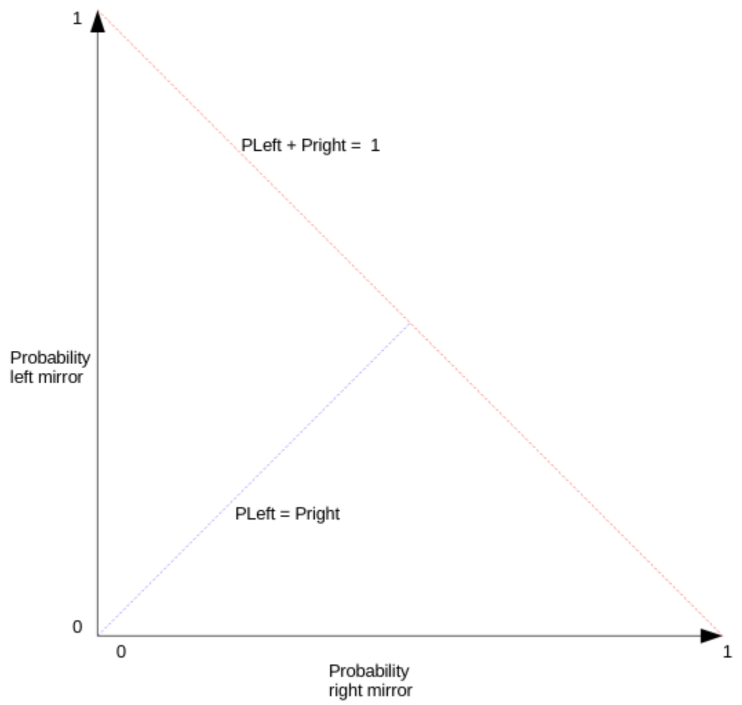

Figure 1: Sample space: The **r**ed line shows p $P_{Left} + P_{Right}$ equal to one. The blue line shows $P_{Left} = P_{right.}$

The winding angle is taken as the cumulative angle subtended by one end of the walk relative to the origin of the two dimensional lattice. For consistent results, step 1 should always be taken in the same direction and all winding angles should be measured with respect to this direction. In the following discussion, step 1 is always eastwards. The first step of the walk is from point (0,0) to (1,0) with no mirror being generated at the origin. The red mirror in *Figure 2* represents the mirror generated with pre-determined probability should the particle return to the origin. The winding angle is therefore calculated with respect to this initial step. *Figure 2* depicts a maroon dot at the point the particle has reached winding angle of 360 degrees and shows the final winding angle of the walk.

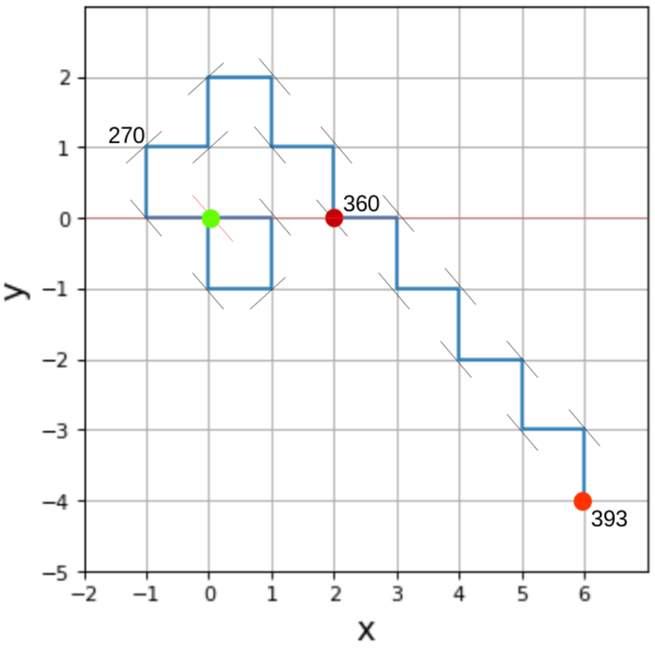

Figure 2: A 20 step walk generating right mirrors with probability 0.2 and left mirrors with probability 0.8, depicted as forward and backward slashes respectively to achieve a mirror density of 1. Notable winding angles are labeled including 360 degree winding angle marked by a maroon dot. The first step of the walk is from point (0,0) (denoted by a green dot) to point (1,0) with no mirror is generated at the origin, hence the red mirror indicates the mirror generated by return to the origin.

1.2.2 Results and discussion

The winding angles distributions for both mirror density equal to one and mirror density less than one are examined in depth. At mirror density of one, $P_{Right} + P_{Left} = 1$, prior research reports known Gaussian behavior[5,7,9-12]. This is confirmed in *Section 3.1* where the winding angle distributions of walks with mirror density equal to one are further probed to reveal interesting fine structure. In *Section 3.2,* walks with mirror density less than one (as demonstrated in *Figure 3*) are examined where there is strong numerical evidence Gaussian behavior breaks down. It is seen the winding angle distributions have more complex behavior than the theoretical Gaussian prediction.

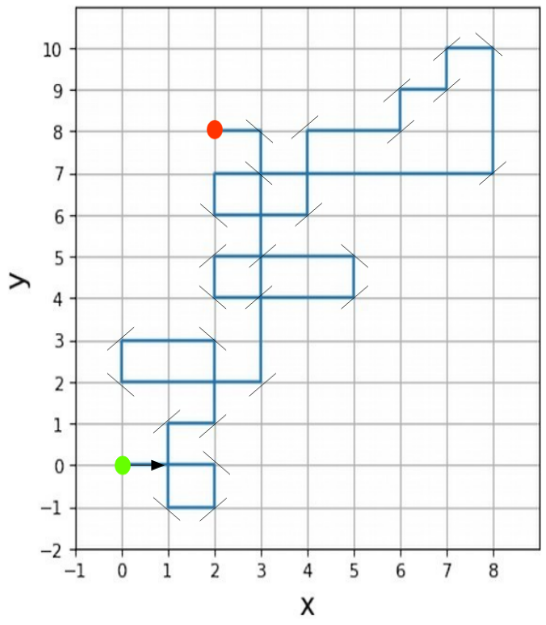

Figure 3: A 50 step walk generating right mirrors with probability 1/3 and left mirrors with probability 1/3 and designating a point having no mirrors with probability 1/3.

The winding angles at $P_{Right} + P_{Left} < 1$ are further investigated by varying P_{Right}, P_{Left} by increments as low as 0.0001. This allows probing of interesting distributions such as those seen at mirror probabilities equal to a third where an asymptotic value of kurtosis of around 3.8, which clearly deviates from the Gaussian value of 3[3] is seen. At very low P_{Right} and P_{Left}, near ballistic behavior is exhibited as the particle follows linear displacement with steps due to a lack of encounters with scatterers. In *Section 3.3,* kurtosis is used as a proxy to determine the probabilities at which unusual trajectories occur in the sample space and these are examined in depth.

2 Method

As aforementioned, the winding angle is taken as the cumulative angle subtended by the end of the walk relative to the origin of the two dimensional lattice. The first step of each walk is from point (0,0) to point (0,1) with a cumulative winding angle of zero. Note, the program implementation takes as its consistent first step a step “northwards” which is different from the prior discussion but since it is treated consistently this gives exactly the same results.

2.1 Program Flow of Execution

This section describes the flow of execution through the program in order to determine the Winding Angle.

The flow control of classes in this program is as follows:
{RandomWalk.ThreadMain()} → Grid → Direction
{RandomWalk.ThreadMain()} → Analytics

Further programming information such as; Design Considerations, Implementation and Learning Points along with other useful files such as; *RandomWalk.cpp, RandomWalkCommonHeader.h, RandomWalkUtilities.cpp, GridUtilities.cpp* are detailed in the Appendix. It is important to note in the body of this report that the program was designed to work effectively on a local computer with limited memory and processing power, in addition to the supercomputer, using a common code base; hence the flow of execution reflects this. All functionality was developed on the local computer before executing on the super computer.

The program takes 6 parameters in the command line. These parameters describe;

- The steps in each walk
- The number of configurations required

- The probability of a mirror facing left (as a percentage)
- The probability of a mirror facing right (as a percentage)
- The probability scale factor, which instructs the program the decimal points of probability which have to be taken into account .
- The action, which can be one of serialise, load , RAM (explained in detail in Appendix A)

The command line parameters collectively constitute one of two structures which control the execution of the program, the other control structure is a series of hash defines in *RandomWalkCommonHeader.h* which determine both which functionality is executed and which code compiles into the program. This second control structure reduces the need for processor costly evaluation of 'if' conditions, in order to maximize the number of configurations that can be determined in any given time frame.

The program determines whether it is running in single thread or multiple thread mode. For the rest of this method discussion it is assumed it is running in single thread mode.

The program uses the information passed on the command line to set up essential data elements for the subsequent executable, the most important of which is the 'GridMap'. (implementation and sizing described in *Appendix A*)

An instance of the Class *Grid* is created and is then initialized. During the initialization function the 'GridMap' is created and used to record the winding angle at every grid position. With the *Grid* initialized, the programs main function in *RandomWalk.cpp,* asks *Grid* to find a configuration and will keep doing so until the required number of configurations has been found. *Grid* never knows how many configurations it is required to find, it simple responds to the request to 'find a walk'.

Prior to each walk commencing, a number of parameters have to be reset. The most important of these is ‘GridMapTrace’, which records each arrival and departure destination for every grid point that is visited during the walk. Hence, at the start of each walk, every position is reset to “not visited”. Another key variable reset is the step count.

Every walk commences with a step from the origin (0,0) to point (1,0). This is treated in the code as a step “northwards” with respect to the Winding Angle.

While *Grid* does not know how many configurations it is required to find, it does know how many steps each configuration is comprised of. With the initialization complete, it asks the *Direction* Class to take one step. Each time *Grid* asks *Direction* to take one step, it invokes the ‘takeOneStep’ function on a subclass of *Direction (*the *North, East, South* or *West* subclass) but it is unaware of this.

Neither *Direction* nor its subclasses know the number of configurations or steps they are required to find. They simply take one step each time they are instructed to do so.

The majority of the work undertaken by the program is taken during the ‘takeOneStep’ function. First, the function updates its departure direction in ‘GridMapTrace’ for the grid point at which it is currently located. It then takes one step in the appropriate direction.

Having arrived at the new location, the function updates ‘GridMapTrace’ to indicate the direction from which it arrived. Next, the change in winding angle and new cumulative winding angle are calculated. For reasons explained in *Appendix A*, the process is as follows:

The change in winding angle is calculated by the difference between the winding angle stored on ‘GridMap’ for the current position and the winding angle stored on ‘GridMap’ for the previous position. *Note:* this can result in a positive or negative value. The cumulative winding angle is then updated by adding the change in winding angle. This automatically takes into account the direction of rotation and rotations greater than 360 degrees to an arbitrary size.

Assuming that the previous step did not involve passing through the center of the grid the program evaluates whether the current position is the center. If so, this is recorded with the direction of rotation and a further step is taken to determine whether the walk has entered a loop.

Assuming the new position is not the center. The new direction that the walk will take is determined. In the instance where the position is visited for the first time, an orientation for a new mirror is obtained according to the probabilities that have been passed on the command line; either right, left or straight on. In the instances where the position is visited for the second time, there is only one direction left to move in. *Note:* the exception to this is for the central grid position (0,0) since no mirror is generated at the origin at step 0.

The new direction is returned by the 'takeOneStep' function in the form of a pointer to the relevant subclass of *Direction* which the 'findWalk' function of *Grid* receives as a pointer to a *Direction* Class instance.

Returning now to 'findWalk' in *Grid*, *Grid* will keep invoking 'takeOneStep' until one of two conditions is satisfied; either the necessary number of steps have been taken, or direction has indicated it encountered a loop. In the former case, the walk is returned to the main function. In the second case, 'findWalk' resets all the relevant variables and starts again and keeps doing so until it has successfully completed a walk of the requisite number of steps.

At this point we have returned to the main calling function and we are in a position to process the walk according to the functionality enabled in *RandomWalkCommonHeader.h*.

Assuming ENABLE_OUTPUT_ANALYTICS() = TRUE, *RandomWalk.cpp* will pass each walk to the Analytics Class in order to generate a Histogram and calculate distribution metrics such as variance and kurtosis.

2.2 The Sample Space

Simulations were performed for walk lengths ranging from 10^2 to 10^6 steps using the algorithm detailed above. This algorithm determined the winding angles of each walk which were then passed to the *Analytics* class which allows production of Histogram data *in situ* with predetermined bin size and range. Winding angles were passed to the *Analytics* class upon generation and did therefore not need to be stored in RAM. Histograms were obtained for winding angle distributions at probabilities across the sample space. The bin size was altered in the range of 0.5 degrees to 22.5 degrees in order to pick up very fine structure and get a more general distribution shape.

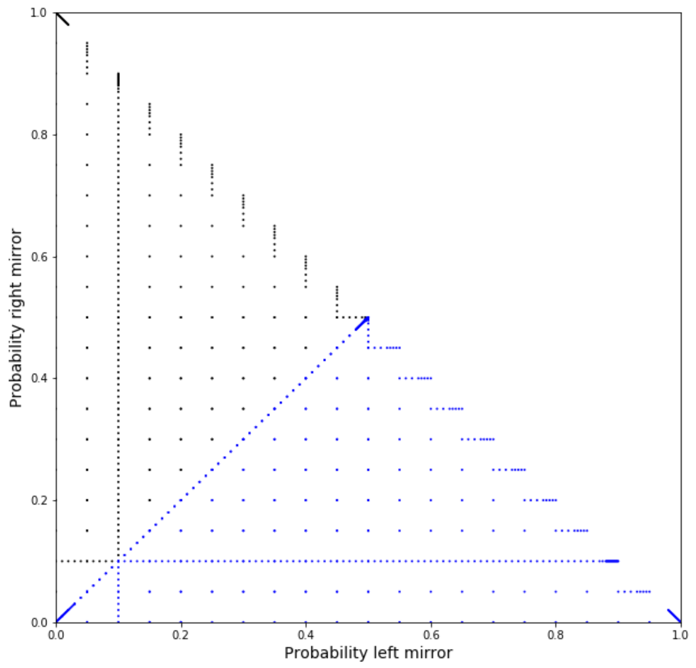

Figure 5: Points show the Left and Right mirror probabilities sampled at N = 1000 steps. The Blue points show actual sampled data that could then be used to extrapolate the black points after proving this behaves symmetrically.

Figure 5 shows the winding angles determined across a wide spectrum of probabilities, with interesting distributions examined in more depth. Note the increased sampling at very high and very low probabilities , where $P_{Right} = P_{Left}$ and as the mirror density $P_{Right} + P_{Left}$ approached one.

The maximum jump in probabilities sampled across the sample space was 0.05. However, at points of interest, probabilities were changed by 0.0001 in the range of 0.1 in order to achieve distribution data such as kurtosis at high resolution.

The *Analytics* class was also used in order to get key analytics data such as Kurtosis and Variance. This was done as follows in order to allow these metrics to be calculated without the program needing to store all winding angles.

```
Initially Mean and Variance and any intermediate values such as
Fractional_delta were set to zero.
```

With each new New_winding_angle pushed to the *Analytics* class, the Number_of_angles was incremented by one and its deviation from the mean was calculated.

```
Delta = New_winding_angle – Mean
Fractional_delta = Delta/ Number_of_angles
Mean = Mean + Fractional_delta
```

Using the terms Fractional_delta_squared and Term1 to calculate the Second_moment, variance could be returned.

```
Fractional_delta_squared = Fractional_delta * Fractional_delta
Term1 = (Number_of_angles – 1)* Delta * Fractional_delta
Second_moment = Second_moment + Term1
```

```
Variance = Second_moment/(Number_of_angles - 1)
```

One Analytics property deemed particularity useful when studying Winding Angles is Kurtosis, referred to as the fourth standardized central moment:

$\langle (X- u)^4\rangle / \langle (X-u)^2\rangle^2$, which indicates the amount of probability in the tails.

In order to calculate Kurtosis, Third_moment and Fourth_moment were determined with each new New_winding_angle pushed to the *Analytics* class.

```
Third_moment += Term1 * Fractional_delta  * (Number_of_angles - 2)  - 3
* Fractional_delta  * Second_moment;

Fourth_moment+= Term1 * Fractional_delta_squared *
(Number_of_angles*Number_of_angles - 3*Number_of_angles + 3) + 6 *
Fractional_delta_squared * Second_moment - 4 * Fractional_delta  *
Third_moment ;

Kurtosis = (Number_of_angles )* Fourth_moment/ ( Second_moment *
Second_moment )
```

Initial sampling across the sample space was done at 100 steps in order to probe for points of interest. A plot depicting the initial probing with respect to Kurtosis is shown in S*ection 3.3* of the results . The majority of the research done at points of interest was for steps of length 1000 or above. At very low mirror probabilities, 100 step walks were deemed insufficient for rigorous sampling as the finite size effect interfered with results. These effects became particularly prominent for mirror probabilities with the combined combination of left and right mirror of less than 0.01, where the vast majority of the walks generated of 100 steps contained no mirrors.

3 Results

3.1 Mirror Density of One

Gaussian behavior is seen with $P_{Right} + P_{Left} = 1$, also referred to in this paper as mirror density equal to one. The distributions below have been scaled to unit variance, using the mean and variance obtained by the *Analytics* class of the program. The standard deviation was obtained from the square root of the variance, allowing determination of the scaled winding angle by:

```
Scaled_winding_angle = (Winding_Angle – Mean)/Standard_deviation
```

This enabled the determination of the scaled density in the Histogram in order for the area under the curve to sum to one, allowing comparison with the Gaussian distribution.

```
Histogram['Scaled_angle_times_frequency'] = Scaled_Angle_Bin_Width
*Histogram['Frequency']

Histogram['Scaled_Density']= Histogram['Frequency']/
(abs( Histogram['Scaled_angle_times_frequency']).sum())
```

The histogram bin size was adjusted in order to show fine structure. At a bin size of 22.5 degrees, all of the below figures showed Gaussian distribution, as seen in *Figure 7a*. At bin size = 5 degrees, increasing amounts of fine structure is seen as the difference in P_{right} and P_{Left} increases.

Figures 6a and *6b* both show binning at 5 degrees in order to demonstrate that the fine structure of the distribution $P_{Right}=P_{Left}=0.5$ has perfect agreement with the Gaussian. Despite the distributions using 22.5 degree binning showing a kurtosis of 3 and perfect alignment with the Gaussian in *Figure 7a*, fine structure is seen at 5 degree binning. In addition, the fine structure of the distribution starts to loose symmetry and the mean shifts from 0.014921 for $P_{Right}=P_{Left}=0.5$ to -9.88333 for $P_{right}= 0.6$, $P_{left}= 0.4$.

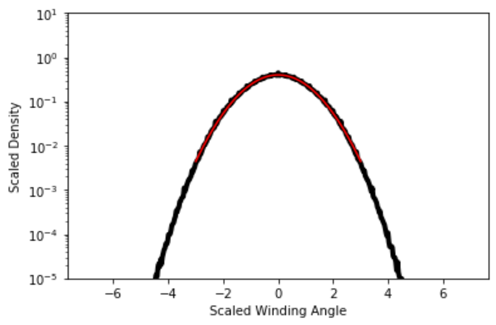

Figure 6a: 5 degree bins, $P_{Right}=P_{Left}=0.5$

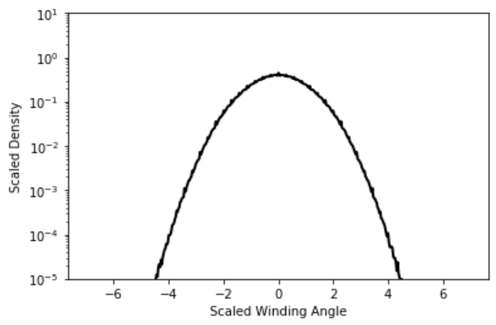

Figure 6b: 5 degree bins, $P_{Right}=P_{Left}=0.5$

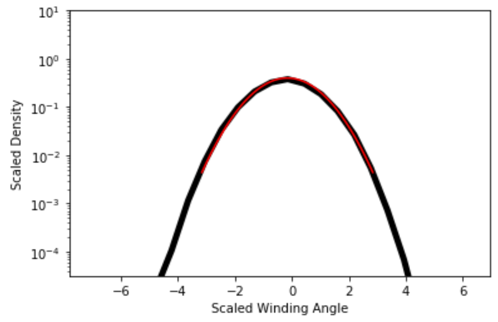

Figure 7a: 22.5 degree bins, $P_{Right}=0.6$ $P_{Left}=0.4$

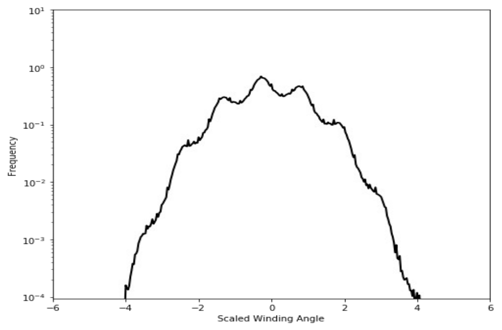

Figure 7b: 5 degree bins, $P_{Right}=0.6$ $P_{Left}=0.4$

Figures 6 **and 7:** Winding Angle Distribution from 10^8 configurations of 1000 step walks **. S**ince left and right mirror probabilities sum to 1, we see Gaussian distribution of winding angles. Figure 6 gives: Mean: 0.014921, Variance: 7.57432, Kurtosis: 3.00219. Figure 7 gives: Mean:-9.88333, Variance: 7.53568, Kurtosis: 3.04248. Figure 7a shows large bins in order to remove fine structure and demonstrate agreement with Gaussian, shown with a red curve. Figure 7b shows smaller binning of 5 degrees to include fine structure.

This fine structure becomes more prominent as the difference in mirror probabilities increases. The distributions plotted become increasingly modulating. The mean continues to move towards -45 degrees with $P_{Right} = 0.7$, $P_{Left} = 0.3$ and $P_{Right} = 0.8$, $P_{Left} = 0.2$, giving means of -19.5074 and -28.6580, respectively.

Figures [8-11] demonstrate the increasing modulation as P_{Right} is progressively increased by 0.1 and P_{Left} decreased by 0.1 maintaining $P_{Right}+P_{Left}=1$. The distributions generated with 22.5 degree bins have been omitted since they showed perfect Gaussian distribution as in *Figure 6a* and 7*a.* Instead the Gaussian is shown on the figures with a red curve. The fine structure seen below is surprising since analytics point to perfect alignment with the Gaussian at mirror density equal to one. Instead the fine structure reveals increasing modulation around the Gaussian with prominent sharp peaks showing at $P_{Right} = 0.9$, $P_{Left} = 0.1$.

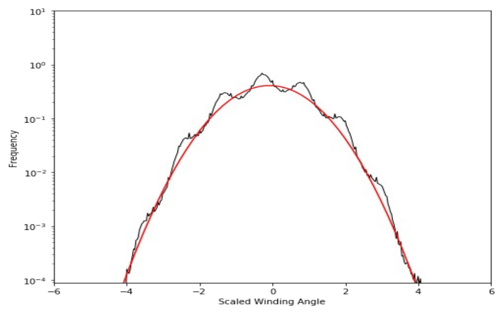

Figure 8: $P_{Right} = 0.6$, $P_{Left} = 0.4$

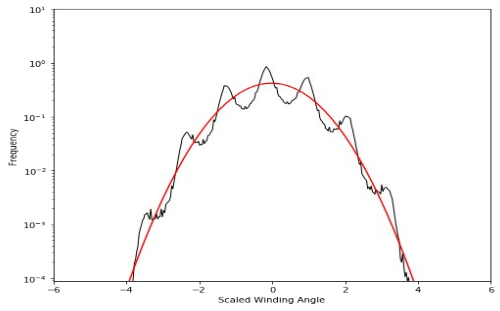

Figure 9: $P_{Right} = 0.7$, $P_{Left} = 0.3$

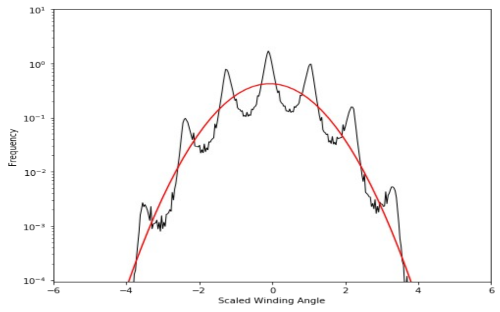

Figure 10: $P_{Right} = 0.8$, $P_{Left} = 0.2$

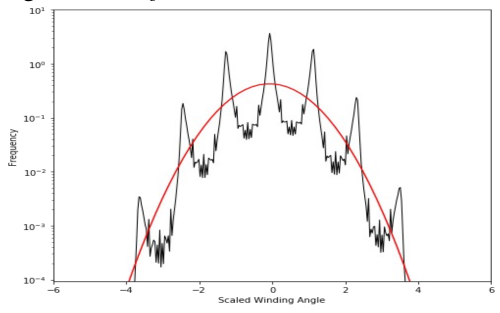

Figure 11: $P_{Right} = 0.9$, $P_{Left} = 0.1$

Figures 8, 9 ,10, 11 : Winding Angle Distribution from 10^8 configurations of 1000 step walks. Metrics for Figure 8 are described in the caption for Figure 7 on the previous page. Figure 9 has; Mean: -19.5074, Variance: 7.42298, Kurtosis: 3.03917. Figure 10 has; Mean: -28.658, Variance: 7.20924, Kurtosis: 3.03402. Figure 11 has; Mean: -37.1664, Variance: 6.79405, Kurtosis:3.02001. Gaussian is shown in red.

Figure 12 shows that as the number of steps increases to 10 000 steps, the fine structure becomes more prominent, as indicated by the blue line. This confirms that the fine structure seen is not noise and cannot be attributed to unusual behavior at a particular length walk.

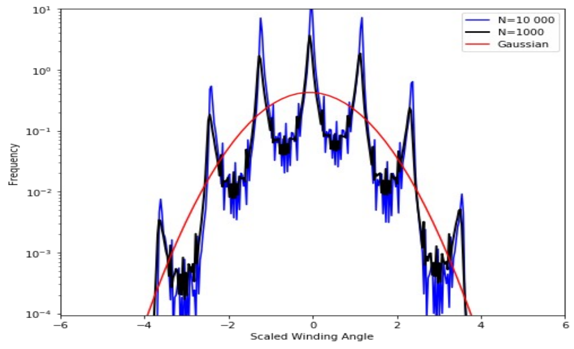

Figure 12: Winding Angle Distribution from 10^8 configurations generating right mirrors with probability 0.9 and left mirrors with probability 0.1. The distribution for 1000 step walks is shown in black with the distribution for 10 000 step walks shown in blue indicating increased structure.

Despite interesting fine structure shown in the distributions, all of the above distributions show mesokurtic distributions with a numerical kurtosis value of three and alignment with the Gaussian when 22.5 degree binning was used. An exception arose where mirrors of only one orientation were generated in order to achieve a mirror density of one, i.e. in the cases:

$P_{Right} = 1$ $P_{Left}=0$ and $P_{Right} = 0$ $P_{Left}=1$

Distributions with the probability of either the right or left mirror equal to one show a peak at -45 or 45 degrees, respectively, due to the formation of channels as demonstrated in *Figure 13*.

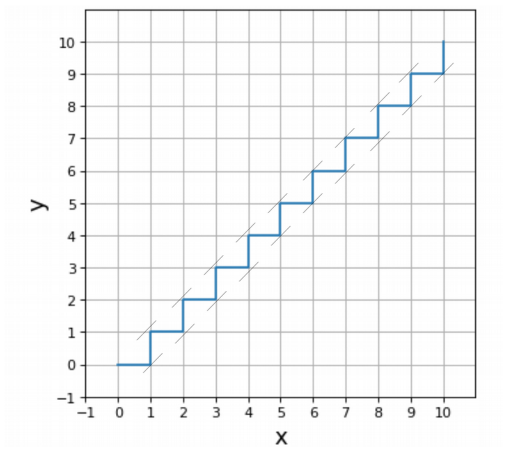

Figure 13: A 20 step walk generating right mirrors with probability 1 and left mirrors with probability 0 showing a final winding angle of - 45 degrees as the walk gets stuck in a channel.

3.2 Mirror Density Less than One

Whilst at mirror density equal to one, the walks have a winding angle distribution that is Gaussian, this breaks down with mirror densities less than one.

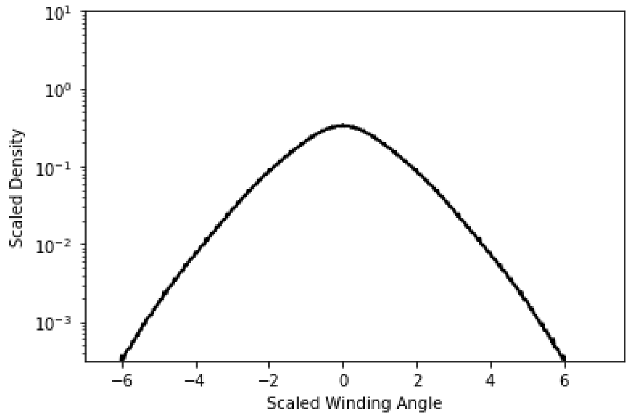

Figure 14: Winding Angle Distribution from 10^8 configurations of 1000 step walks generating right mirrors with probability 1/3 and left mirrors with probability 1/3. Mean: -0.0199307, Variance: 14.8567, Kurtosis: 3.83548.

Figure 14 shows the distribution generated from 10^8, 1000 step walks. The distribution clearly deviates from the Gaussian with Kurtosis of 3.835 seen. This value is in line with the value determined in prior research [3]. Due to a proportion of the mirrors being neither left nor right, a larger variance is observed than at mirror density equal to one with a broader more peaked distribution. Displacement from origin is also determined with maximum displacement returned using the program's analysis functionality in *RandomWalkCommonHeader.h*. Unsurprisingly, distributions with $P_{Right} + P_{Left} < 1$ have a higher displacement from origin than those generated with $P_{Right} + P_{Left} = 1$.

Moving focus to look at very low probabilities of mirrors generated a large displacement from the origin is seen since walks experience much less scattering. In addition, early scattering events significantly effect the winding angle. Altering the lengths of the walks sampled would greatly change the distributions at low mirror probabilities since at shorter walks such as those with 100 steps, scattering events become less and less likely. At a higher number of walks such as those with 10 000 steps, we would expect the below distribution to occur at lower mirror probabilities whereas with walks of 1000 steps at probability of 0.1 a mean free path of 100 leads on average to a scattering event every 100 steps. *Figures [15-24]* show the scaled winding angle for runs with 10^6 configurations of 1000 step walks for equal mirror probabilities decreasing under 0.01 for left and right mirrors. These distributions are symmetrical with a mean of around 0 degrees.

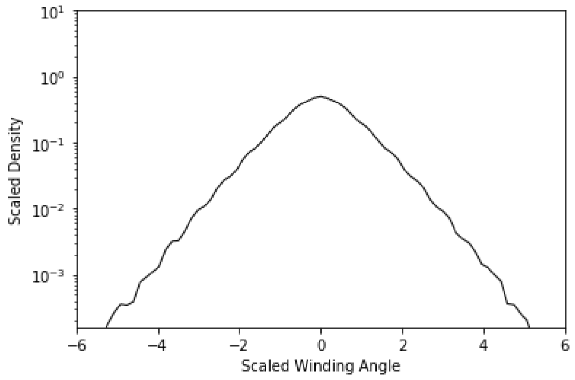

Figure 15: PRight = PLeft = 0.008, Bins = 12.25 degrees

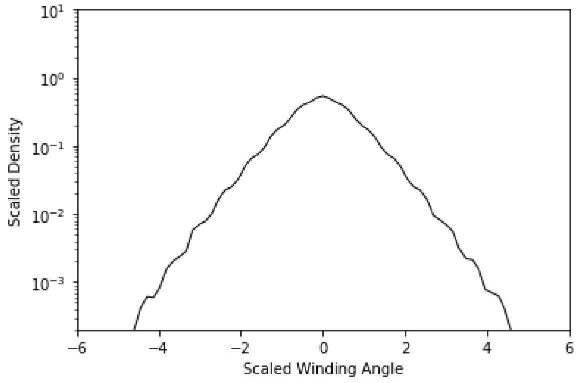

Figure 16 : PRight = PLeft = 0.006, Bins = 12.25 degrees

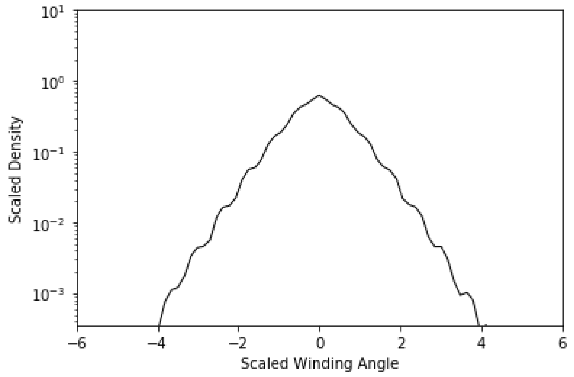

Figure 17: : PRight = PLeft = 0.004, Bins = 12.25 degrees

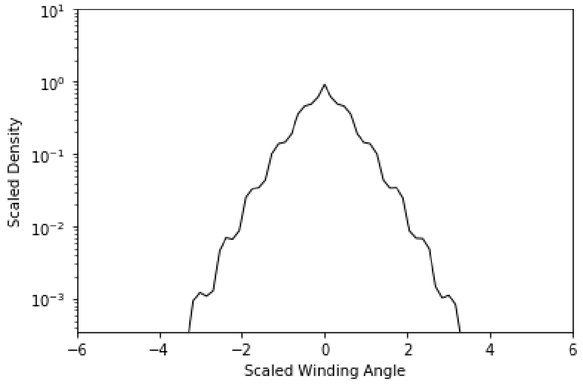

Figure 18 : PRight = PLeft = 0.002, Bins = 12.25 degrees

As P_{Right} and P_{Left} decreased, the bin size was made smaller to show the increased prominence of the central peak as more and more walks encounter no mirrors. Ridges appearing in the distribution show the effect of individual encounters with scatterers on the winding angle depending on where in the walk they are reached. Encounters with scatterers early on in the walk shift the winding angle by 45 degrees but this is decreased as the particle encounters a scatterer later.

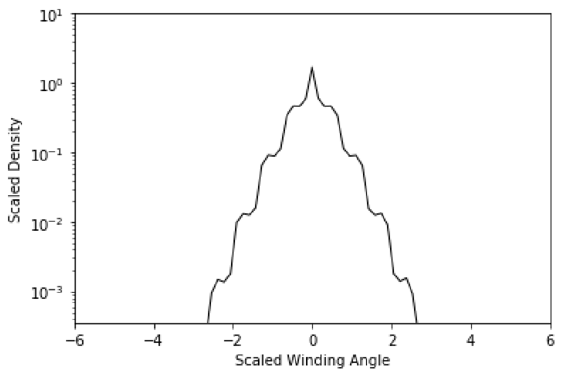

Figure 19: PRight = PLeft = 0.001, Bins = 12.25 degrees

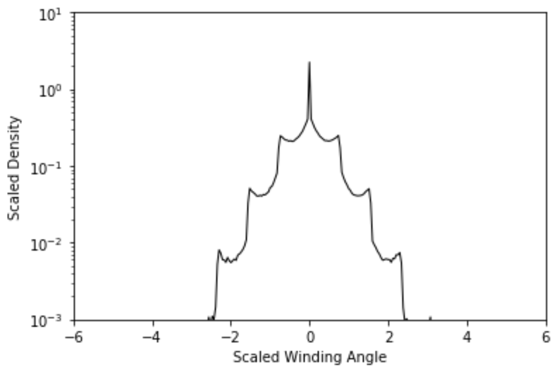

Figure 20 : PRight = PLeft = 0.0008, Bins = 5 degrees

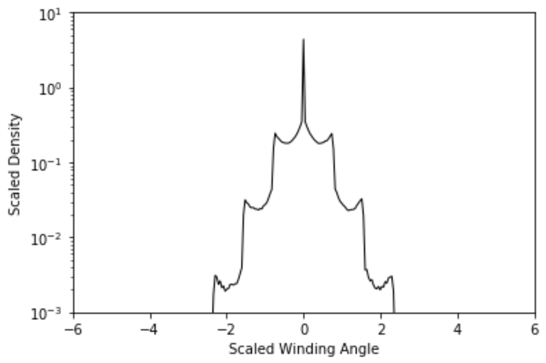

Figure 21: PRight = PLeft = 0.0006, Bins = 5 degrees

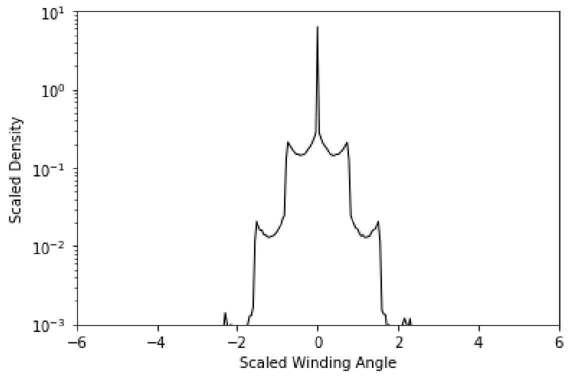

Figure 22: PRight = PLeft = 0.0004, Bins = 5 degrees

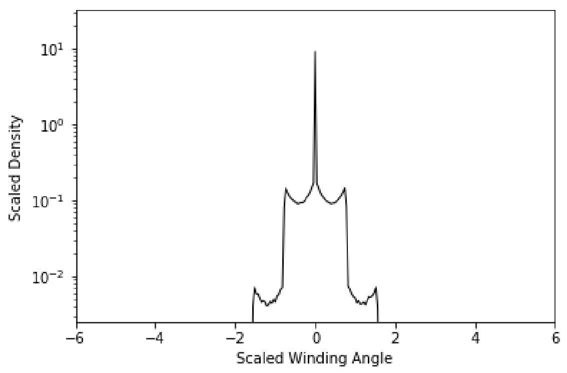

Figure 23: PRight = PLeft = 0.0002, Bins = 5 degrees

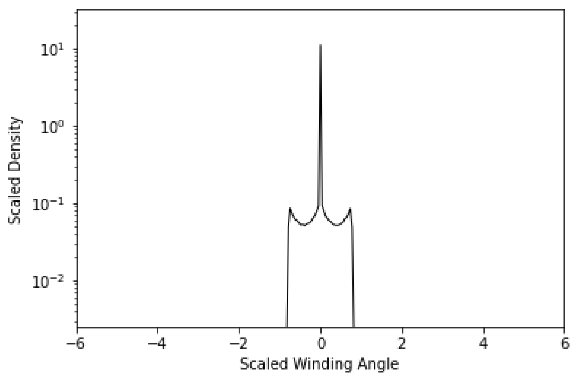

Figure 24: PRight = PLeft = 0.0001, Bins = 5 degrees

At the lowest probability sampled, *Figure 24* shows the vast majority of walks encounter no scatterers, the two ridges in the distribution indicate a small proportion of walks encounter either a left or right mirror which alter the winding angle by somewhere in the range of 0 to 45 degrees depending on the point in the walk where the scatterer is encountered.

3.3 Kurtosis

As mentioned above, normal distribution has kurtosis equal to to 3, this can also be referred to as a mesokurtic distribution. If the kurtosis is greater than 3, the distribution is leptokurtic and the data set has heavier tails than a normal distribution. An example of this is logistic distribution which resembles normal distribution but has heavier tails with kurtosis of approximately 4.2 [13]. If the kurtosis is less than 3, platykurtic distribution, then the dataset has lighter tails than a normal distribution.
Since kurtosis enables the probing of deviation of a random variable from a normally distributed one, it acts as a good proxy for determining where unusual behavior occurs on the sample space and hence where interesting distributions occur. *Figure 25* shows the sample space as a 3D scatter plot as sampled at 100 steps in order to probe for points of interest.

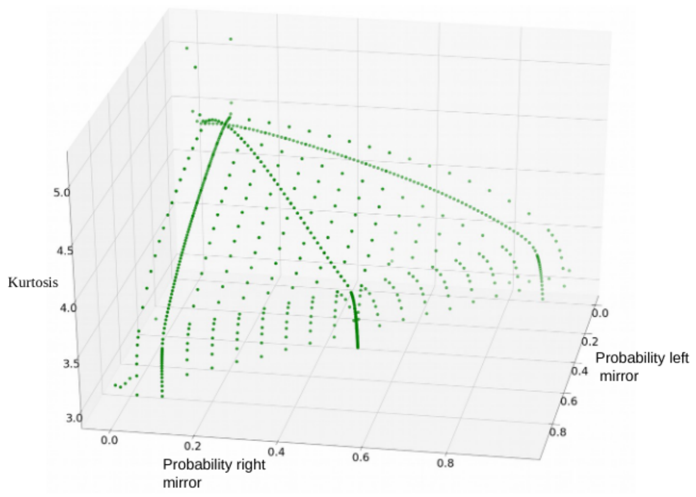

Figure 25: 3D scatter plot showing Kurtosis against probability of Left and Right mirrors for 100 step walk with analytics taken from 10^8 configurations

Variance was also looked at in order to confirm expected behavior. Several prior papers found the winding angle distribution having a variance that grows asymptotically as C log N [5,9]. *Figure 26* shows little noise and clear agreement with the theoretical straight line.

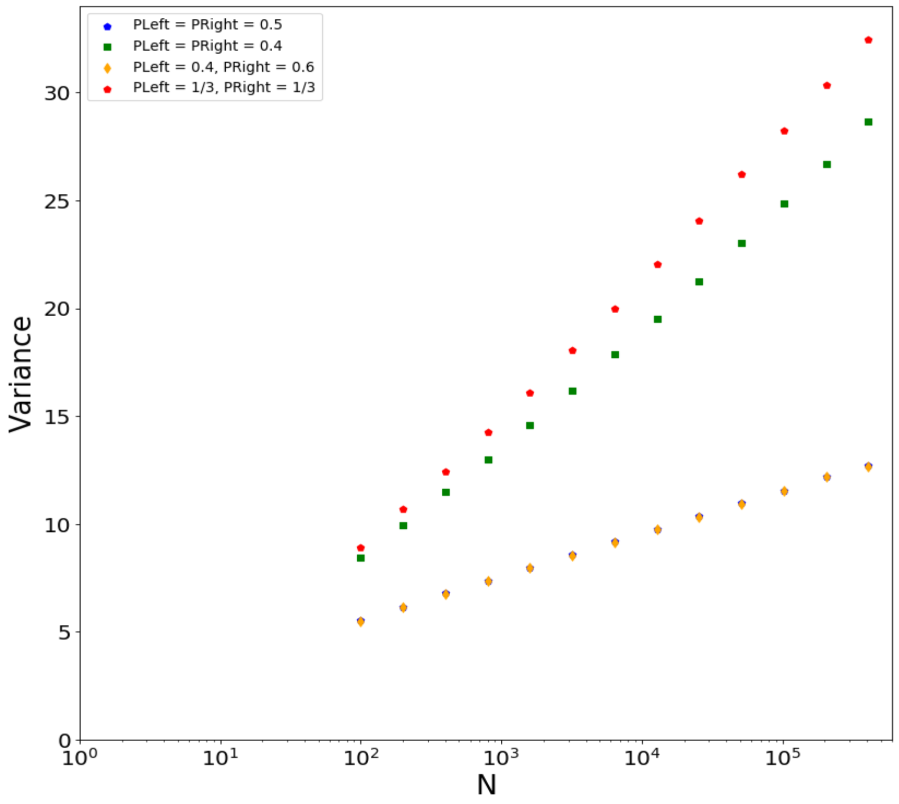

Figure 26 : Variance in radians against Log(N) with varying length walks where each variance point is taken from 10^8 configurations. Note overlap of variance with mirror density equal to one.

Almost identical variance was seen for both sets of probabilities resulting in a lattice mirror density of one. A disproportionate probability of left and right mirrors had very little effect on the variance as suspected from results found in *Section 3.1* where it was shown that regardless of the respective $P_{right,}$ P_{Left} , the winding angle distribution is Gaussian. As the mirror density was decreased from 1 ($P_{Right} = P_{Left} = 0.5$ and $P_{Right} = 0.6$, $P_{Left} = 0.4$) to 0.8 ($P_{Right} = 0.4$, $P_{Left} = 0.4$), the variance increased. This effect was seen as the density of mirrors on the lattice site was further decreased to $P_{Right} = 1/3$, $P_{Left} = 1/3$ and was more significant with higher N. In addition, *Figure 26* indicates slight curvature in the graph with mirror densities of less than one showing the variance grows faster than linearly in Log(N). Since the variance indicates that parameters change on the logarithmic scale, the Kurtosis was plotted as a function of 1/Log(N) keeping mirror probabilities fixed. The dotted lines seen in *Figure 27* are to guide the eye, not for extrapolation.

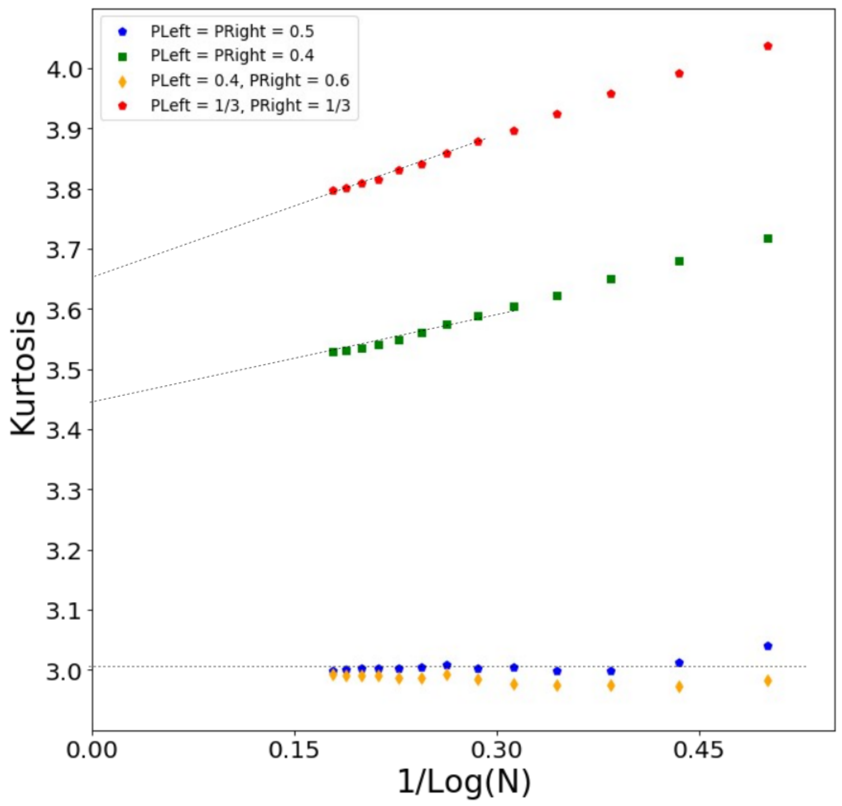

Figure 27: Kurtosis against 1/Log(N) with varying length walks where each kurtosis point is taken from 10^8 solutions. The green squares show Kurtosis at equal mirror probability = 0.4, whilst blue points and orange squares show kurtosis where mirror probability =1 with right mirror = left mirror and disproportionate mirror probabilities.

Whilst significant change in Kurtosis with 1/Log(N) is seen for $P_{Right} + P_{Left} < 1$, no notable change is seen for $P_{Right} + P_{Left} = 1$ where the value of the kurtosis in line with the Gaussian value of 3. Where mirror density is less than one, clear deviation from the Gaussian value is seen as winding angle distributions have significantly more probability in the tails. The red dots in *Figure 27* show $P_{Right} = P_{left} = 1/3$ and indicate an asymptotic value of Kurtosis between 3.65 and 3.8. This behavior does not indicate stretched exponential where much fatter tails are seen with a kurtosis value of 6[14]. The above findings clearly show the winding angle distributions move away from exhibiting Gaussian behavior as mirror probabilities are decreased from $P_{Right} + P_{Left} = 1$.

Figure 28 shows Kurtosis as this phenomenon was investigated further by decreasing P_{Right} and P_{Left} by equal increments of 0.001 from 0.5 to 0 in order to see this effect at high resolution.

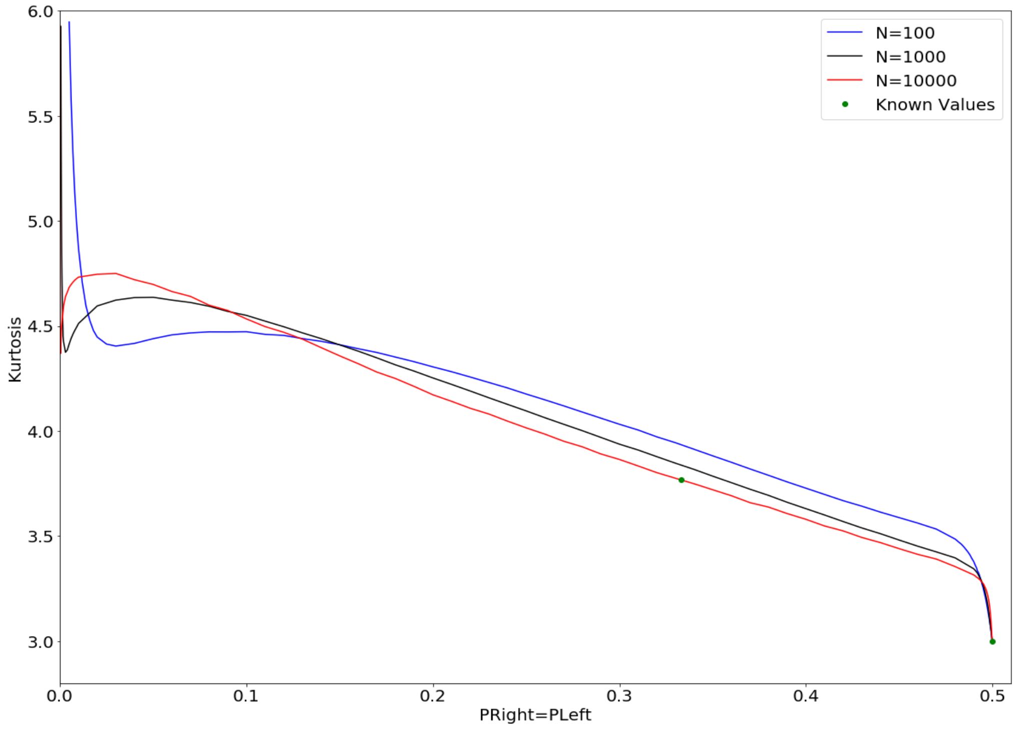

Figure 28: Values of kurtosis calculated from 10^8 configurations of 1000 step walks generating left and right mirrors of equal probability. The green dots indicate the known values of kurtosis at p=1/3 and p = ½ . This clearly shows that the numerical value of the kurtosis also approaches the Gaussian value of 3 . Values of kurtosis above 6 were cut as these were put down to the finite size effect.

Linear behavior is observed from approximately $P_{Right} = P_{left}$ 0.2 to 0.48. As in shown in *Figure 27*, kurtosis is seen to decrease as the number of steps is increased in this region. This changes at the extremities. Unusual behavior is seen as the Gaussian value of kurtosis as mirror density equal to one is approached where a sharp dip in kurtosis is observed.

At lower mirror density there is additional interesting behavior. In all walk lengths a dip is observed at low probabilities before the kurtosis dramatically increases. The sharp increase in kurtosis occurs at higher probabilities for lower N and can most likely be attributed to a finite size effect since at low mirror probabilities, the chances of a particle encountering a mirror on any given walk is smaller at smaller N.

Figure 29 shows the change in kurtosis at high resolution for $P_{Right}=P_{Left} < 0.1$. In all values of N, the dip is clearly defined and occurs at kurtosis of just under 4.4 for 1000 and 10 000 steps, showing the distributions have significantly more weight in the tails than the Gaussian distribution but cannot be attributed to a stretched exponential distribution of 6.

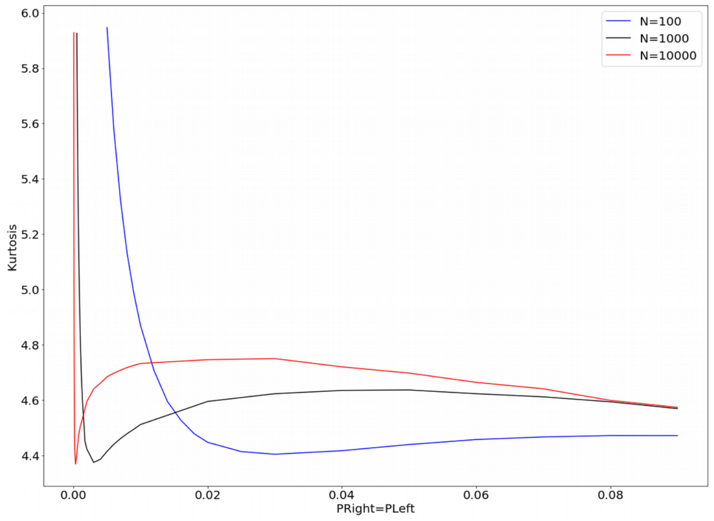

Figure 29: Values of Kurtosis calculated from 10^8 configurations generating left and right mirrors of very low equal probability. Values of kurtosis above 6 were cut as these were put down to the finite size effect.

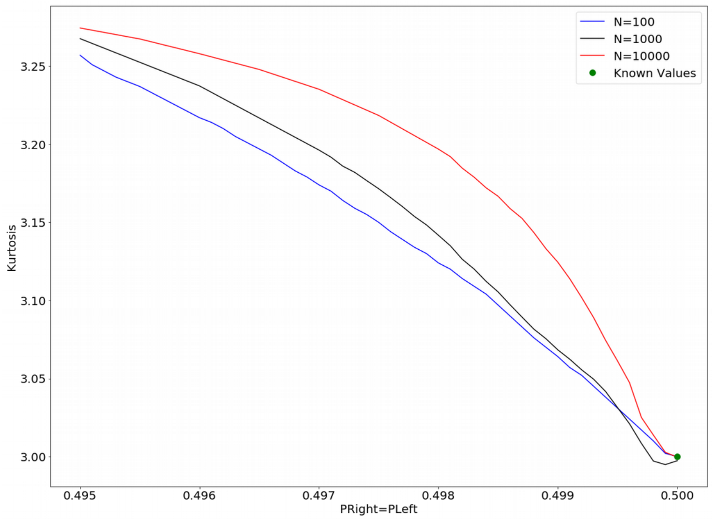

Figure 30: Values of Kurtosis calculated from 10^8 configurations generating left and right mirrors of equal probability approaching $p = 1/2$.

Figure 30 shows change in kurtosis as P_{Right} and P_{Left} are increased from 0.495 to 0.500. Surprising behavior is observed with N=100, N=1000 and N=10000 following no obvious trend. More sampling at 100 000 steps would be required in order to confirm the indication that the decrease in kurtosis becomes progressively less linear as N is increased.

A natural progression from observing the change in Kurtosis as the mirror density approaches one with equal probability of left and right mirrors is the probing of Kurtosis as the mirror density approaches one with unequal mirror probabilities.
Figure 31 was generated by altering P_{Left} in the range 0 to 0.9 keeping P_{Right} fixed at 0.1.

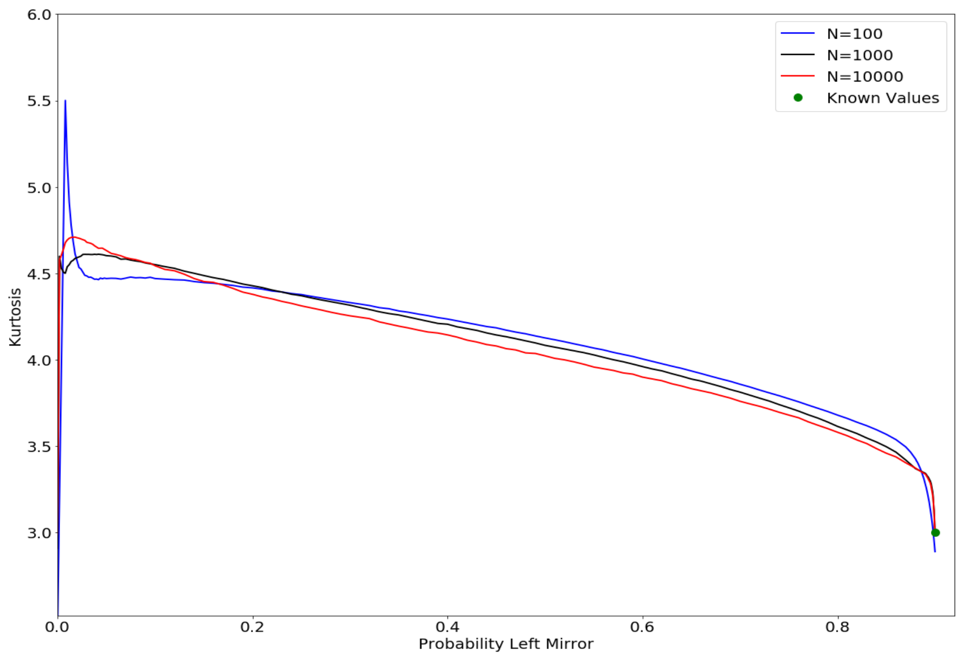

Figure 31: Values of Kurtosis calculated from 10^8 configurations generating left mirrors of varied probability and right mirror at fixed probability = 0.1.

Very similar behavior was seen as in *Figure 28*, with a large linear section showing theoretical agreement with the effect of step length on kurtosis in the section ranging from approximately P_{Left} = 0.2 to 0.7. The behavior differs at the extremities. Where P_{left} approaches 0, the kurtosis rapidly decreases at all values of N. Abnormal behavior is observed for N=100, which is mostly likely due to the step length since minute changes in probability are not reflected in the kurtosis at this resolution. At steps of length 100 and 1000, a dip is observed at approximately 4.4 but rounds off at 10 000 steps where the distribution reaches a kurtosis of 4.7. Clear deviation from the Gaussian distribution is again demonstrated with a kurtosis most closely attributed to logistic distribution which resembles normal distribution but has heavier tails and a kurtosis of 4.2.

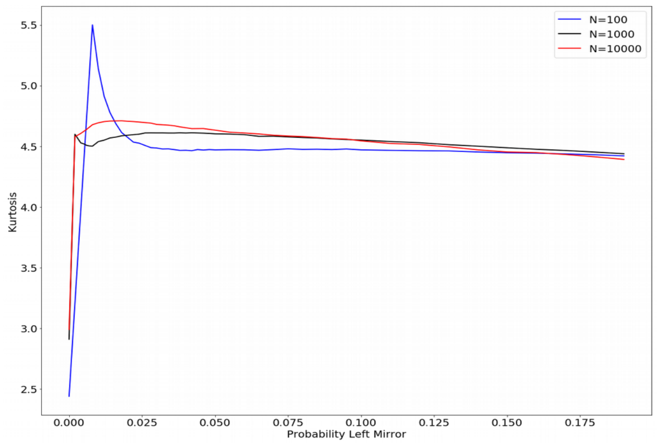

Figure 32: Values of Kurtosis calculated from 10^8 configurations generating left mirrors of varied low probability and right mirror at fixed probability = 0.1.

Figure 33 again indicates abnormal behavior for N=100 as kurtosis drops below the expected Gaussian value of 3 however since the drop is not significant this could be attributed to the finite size effect. Step lengths: 1000 and 10 000 behave consistently exhibiting very similar behavior with clear deviation from linearity as the mirror density approaches one.

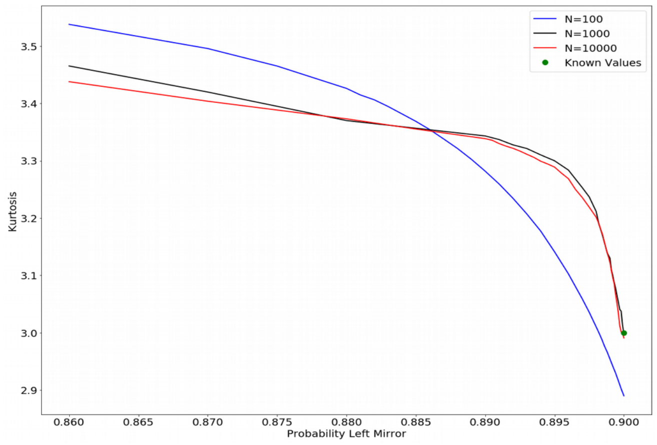

Figure 33: Values of Kurtosis calculated from 10^8 configurations generating left mirrors of varied high probability and right mirror at fixed probability = 0.1.

4 Conclusions

A program was designed in order to determine the winding angles for walks of up to 10^7 steps for up to 10^9 configurations which works quickly on a regular local machine. Known Gaussian behavior of the winding angle distributions were confirmed for lattices with a mirror density of one and deviation from the Gaussian for notable probabilities such as $P_{Right} = P_{Left} = 1/3$ were also seen in line with prior research.

Winding angles were studied across the parameter space at all possible probabilities, with kurtosis taken at probability increments of 0.05 for 100 step walks in order to probe for interesting behavior. Unusual fine structure was observed at a mirror density of one where the distributions modulates around the Gaussian. This is a new finding. It was proven that this could not be attributed to the finite size effect by demonstrating an increase in structure at high N.

The change in kurtosis as mirror probabilities moved towards one was examined in depth for equal and unequal probabilities of left and right mirrors. Again, unusual distributions that clearly deviate from the Gaussian were observed, both at very low mirror probabilities and as the lattice approached a mirror density of one. More research at higher step length is required in this area since from the variation of step lengths sampled it is unclear whether the correct asymptotic regime was reached.

References

[1] Dettmann, C, P.: Diffusion in the Lorentz Gas. Commun. Theor. Phys. **62**, 521–540 (2014)

[2] Chernov, N,I,. Markarian, R.: Chaotic Billiards, Amer.Math. Soc. Bookstore, Providence (2006)

[3] Lorentz, H,A.: Ergebnisse und Probleme der Elektronentheorie, Proc.Arnst.Acad. **7**, 438 (1905)

[4] Webb, B., Cohen, E.G.D.: Self-avoiding modes of motion in a deterministic Lorentz lattice gas. J. Phys. A Math. Theor. **47**, 315202 (2014)

[5] Narros, A,. Owczarek, A.L,. Prellberg, T.:Anomalous polymer collapse winding angle distributions. Phys. A: Math. Theor. **51,** 114001 (2018)

[6] Shapir, Y,. Oono, Y.: Walks, trials and polymers with loops. J. Phys. A Math. **17**, L39--L44 (1984)

[7] Drossel, B. Prellberg, T.: Winding angle distribution for two-dimensional polymers at the theta-point,' Physica A **249** 337-341 (1998)

[8] Spitzer, F .: Trans. Amer. Math. Soc. ,**87,** 187-197 (1958)

[9] Narros, A,. Owczarek, A.L,. Prellberg, T.: Winding angle distributions for two-dimensional collapsing polymers. J. Phys.: C. **686**, 012007 (2016)

[10] Owczarek, A.L,. Prellberg, T.: The collapse point of interacting trials in two directions from kinetic growth simulations. J .Stat. Phys **79**, 951-967 (1995)

[11] Owczarek, A.L,. Prellberg, T.: Collapse transition of self-avoiding trails on the square lattice. Physica A **373**, 433 (2006)

[12] Owczarek, A.L,. Prellberg, T.: Manhattan Lattice Theta-point Exponents from Kinetic Growth Walks and Exact Results from the Nienhuis O(n) Model, J. Phys. A: Math. Gen. **27** 1811-1826 (1994)

[13] Hanagal, D, D.: Handbook of Statistics, **37** 209-247 (2017)

[14] Zhang, J., Suo, S., Liu, G., Zhang, S., Zhao, Z., Xu, J., Wu, G.: Comparison of Monoexponential, Biexponential, Stretched-Exponential, and Kurtosis Models of Diffusion-Weighted Imaging in Differentiation of Renal Solid Masses. Korean journal of radiology, **20**, 791–800 (2019)

[15] Blackman, D., Vigna, S.: xoshiro** ,http://creativecommons.org/publicdomain/zero/1.0/

5 Appendix A

5.1 Design considerations and Decisions

The majority of the data for this paper was obtained by running this program on a super computer. However, in order i) avoid wasting costly supercomputer processing time and ii) make development more efficient, all functionality was first implemented and tested on the development machine before promotion to the supercomputer. Not surprisingly, there were significant differences between the capabilities of, and resources available to, the two computers. This decision to fully implement all functionality on both machines had significant design implications.

Initially hash tables were used, indexing on the x value for each bucket however this led to problems with speed since the evaluations at each new point was high due to the number of 'if' statements and 'push/pop' statements on the C++ container.

A decision was made to use a data construct that provided contiguous memory, hence vector was chosen since it also allows for direct access to its elements without having to use push and pop. In addition, a vector can be used to represent 2D array (Grid) with simple maths.

A key design decision was to minimise the number of 'if' statements that had to be evaluated. Each step could involve a dozen or so 'if' statements and thus a run with 1,000 steps and searching for 1,000,000 configuration might need to evaluate in excess of 10 billion 'if' statements. This key design decision was implemented using a three-fold approach. First, secondary functionality, such as printing verbose output or logging function execution times, as well as more fundamental functionality, such as the enabling of multiple threads and the saving of winding angles, were enabled or disabled using #defines. This enabled selected code to be removed from the compiled executable and with it the need to check at run time if it should be executed. For Verbose mode, in particular, this check would otherwise have had to be performed multiple times per step. Secondly, polymorphism was used so that the program does not need to evaluate the direction from which it arrives at a point. Finally, by encoding the “visit history” at each grid location, if a position is revisited the program does not need to evaluate what the previous transit through that point; straightforward subtraction can be used to determine the destination direction.

An early implementation of this approach used ‘ChangeInWindingAngles’ for every grid point relative to its four neighboring grid points. This had the benefit that knowing the direction one had reached a particular grid point from, one could immediately look up the change in angle and add it to the cumulative total. The only maths required was an addition. This implementation was faster - particularly if the change in winding angles were loaded from disk having been pre-calculated during a previous run. However, this approach used almost four times as much memory, so a compromise was taken to store winding angles themselves and not the changes.

For memory reasons floats are used rather than doubles despite implication for precision of winding angles over a very large number of steps.

Depending on the size of file written to, results were either written line by line or for files such as ‘GridMap’, in binary. Depending on the action passed on the command line, on execution this ‘GridMap’ is either, created in the current executable (Action = Serialise) and saved to disk, read from disk (Action = Load), or created in the current executable and not saved (Action = RAM).

A compression factor was used to reduce the size of GridMap created. A compression factor of 1 directs the program to create a vector with exactly the number of elements required to encode a grid of length (2 * Steps) + 1. Fully half of such a grid is never reachable by any walk. As the value of Steps increases, both the development computer and supercomputer run out of memory to capture the associated grid. In addition, the time required to create the grid rapidly increases. Further, the proportion of the full grid that can actually be reached drops off rapidly, particularly with mirror probabilities that create more winding. A lower compression factor such as 0.1, for example, would result in a grid allowing for a net movement of one tenth the number of steps, net, along either axis and a 100 fold reduction in required memory.

This program is well suited to a multi threaded approach. The benefits of using multiple threads both depends on, and has implications for, other design considerations. A multi threaded implementation which is materially faster was developed and testing indicates that it is functionally correct. It works by dividing up the solutions (configurations) between the number of threads which are available. However, it was found harder to debug and thus to be a more complicated solution to develop and maintain. Therefore, all the data collected for this project was was acquired by running the program in single thread mode in order to be confident about integrity of data.

5.2 Implementation

This section describes the implementation used to generate the data on which the mathematical analysis has taken place.

A key decision was to use a common header file, allowing definition of global constants and a sinmple way of controlling the program.

The program can be considered to be a simple main function and three classes. The flow control of classes in this program is as follows:

{RandomWalk.ThreadMain()} →   Grid → Direction

{RandomWalk.ThreadMain()} → Analytics

In the case of *Grid* some of its functionality is implemented via a utility file, *GridUtilities.* A number of utilities are also provided in *RandomWalk.cpp.*

Due to RAM limitations unsigned data types were used wherever possible and the shortest data type was used. This occasionally presented problems when variables were assigned values greater than originally foreseen. Particular instances were needing to replace unsigned short int with unsigned int and replacing unsigned int with unsigned long

In order to achieve both truer random distribution and faster random number generation, an implementation of the xoshiro** algorithm [15] was implemented with a bespoke initialization function.
The random generator function was modified to take the probability range of the generator which again reduced the amount of if conditions in processing each configuration (*see commented code in 6.8 Appendix B*)

Optional functionality includes the ability to log amount of time spent executing individual functions and he number of invocations of those functions, this was useful in the early days of development.

The program provides supporting functions which are not necessary for its mathematical purpose. These functions allow analysis of probabilities generated to confirm that they are correctly distributed,. In addition functions can return the extent that a set of walks reaches in the grid which allows for the compression factor to be adjusted and as small a grid as possible to be created for any given number of steps and desired configuration. Extensive structured output was available to support debugging through the ENABLE_VERBOSE_MODE().

5.3 Learning Points

I was pleased with, but under-estimated, the amount of work required for the program to generate useful info for debugging information and the subtlety of errors that were introduced through changes to code. In particular, whilst I am pleased I constantly sought to improve the design of the program I recognize that in doing so I created a significantly bigger regression testing need than I realized at the start.

I was conscious of wanting to write code to a set of standards from the outset, but did not do so rigorously initially. With hindsight, I would decide and follow my chosen standards from the outset. Not only would debugging, maintenance and refactoring be easier, but less will power would have been required towards the end of the project to make the changes which I knew were necessary but didn't want to contemplate because of the effort required and the risk of introducing defects.

I did not examine the requirements as rigorously as I should have before beginning design and development. Initially the program was designed to place the left right mirrors with respect to the direction of travel which led to completely different distributions and resulted in a less efficient program than had these requirements been better understood at the offset.

5.4 Experimental version

An experimental version has also been developed which was not used for this project but includes code that was at various times experimented with.

In addition to the submitted functionality, the experimental version supports; i) an alternative implementation of calculating grid coordinates that uses Direction's sub classes and has a lower processor overhead, ii) a low memory calculation of winding angles that replaces GridMap with a counter of complete revolutions which is added to the partial rotation angle and iii) a way of recording and replaying direction choices that enables regression testing and interoperability / comparison between different implementations.

It is believed that all functionality is correctly implemented. However, it has not been submitted as it was not the version used to obtain the data that is reported on above and which forms the basis of the findings of this research

6 Appendix B

This section provides notable source code files in the program.

It starts with RandomWalkCommonHeader.h which defines key global constants and is one of the two important control mechanisms.

This is followed by RandomWalk which contains main().

Next follow the implementation (.cpp) files of the three classes (and their subclasses, where applicable) in the order they are used within the program and discussed above.

A full PDF of commented source code, *Appendix 7* was submitted along with a README.txt in the source code .zip file.

6.1 RandomWalkCommonHeader.h

```
#ifndef RandomWalkCommonHeader_h
    #define RandomWalkCommonHeader_h

    #define CENTRE_INITIAL_STATE 0
    #define CLOCKWISE 0
    #define COUNTER_CLOCKWISE 1
    #define COMPRESSION_FACTOR 1 // Suggested values 1.0 for up to 10,000 steps, 0.1 for 100,000  steps
and 0.05 for 1 million steps
    #define EAST 1
    #define ENABLE_ANALYSIS_MODE() FALSE // It shows how much of Grid.Map has been used. Unused
    Grid.Map means COMPRESSION_FACTOR may be reduce. Aim for a "buffer" of 1. Introduces a time
    overhead!
    #define ENABLE_MULTIPLE_THREADS() TRUE // Splits solutions between available cores, each using its
    own grid map and creating instances of analysis class
    #define ENABLE_LOG_FUNCTION_EXECUTION_TIME() FALSE //:: This can be used to track the    relative
execution time of functions. See GridUtilities.cpp for more information.    Introduces an overhead!
    #define ENABLE_OUTPUT_ANALYTICS() TRUE //This writes winding angles to analytics class and reports
    mean and std dev to terminal
    #define ENABLE_OUTPUT_WINDING_ANGLES() FALSE // Enabling this functionality writes WindingAngles to
    the display. winding angles are not currently saved.
    #define ENABLE_SAVE_SUCCESSFUL_PATHS() FALSE // We do not need the paths once we have the winding
    angles. A value of FALSE therefore saves memory.
    #define ENABLE_SAVE_WINDING_ANGLES() TRUE // Enabling this function saves winding angle to csv
    file name as specified in RandomWalk.cpp
    #define ENABLE_VERBOSE_MODE() FALSE // Generates considerably more output. It is     advisable to
    only use this with small Steps and Solutions to avoid swamping the display with output.
    #define FALSE false
    #define FIRST_VISIT 1
    #define GO_LEFT 3
    #define GO_RIGHT 1
    #define GO_STRAIGHT_ON 0
    #define MAXIMUM_SOLUTION_STEPS 1000000000000
    #define MAXIMUM_STEPS 1000000
    #define NORMAL_TERMINATION 1
    #define NORTH 0
    #define PI 3.141592654
    /* Note that we have to give up one decimal point of precision due to resource constraints on the
    development machine. In other words we can multiple radian values between 0 and 2pi by 10 ^ 8
    and do signed arrithmatic without causing overflow errors. This is at the cost of precision. The
    cumulative effects should be calculated to see if they are significant. On a computer with
    greater memory we would have the option of using larger data types. */
    #define PRECISION_DECIMAL_POINTS 8
    #define SOUTH 2
    #define TAB1 "  "
    #define TAB2 "      "
    #define TAB3 "          "
    #define TRUE true
    #define UNVISITED 7
    #define VERSION "10.0.0 Final Project Code"
    #define VISITED_TWICE 6
    #define VERY_BIG 100000000
    #define WEST 3

    /* Below is the information requried for the statistic element of the program,
       The lower and upper bins are defined in winding angles and should be altered depending on the
    ratio of Left to Right. These #defines should be removed from RandonWalkCommonHeader and passed
    on the   Command Line.*/
    #define LOWEST_BIN -1097.5
    #define HIGHEST_BIN 1097.5
    #define NUMBER_BINS 439

    #include <math.h>
    #include <time.h>
    #include <omp.h>

    #include <cstdlib>
    #include <fstream>
    #include <iostream>
    #include <sstream>
    #include <string>
    #include <vector>

    using namespace std;

    typedef struct Log {
    clock_t TotalExecutionTime = 0;
    long Invocations = 0;
    std::string ID;
    std::string FunctionName;
    std::string RelativeIndentation = "";
    } Log;

    unsigned int nextRandomDirectionChange(unsigned int Range);
    void xoshiro256ss_init();

    inline void fatalError(const std::string& Message){
        std::cout<<"FATAL ERROR: "<<Message<<" Exiting..."<<std::endl;
        exit(EXIT_FAILURE);
        }
#endif
```

6.2 RandomWalk.cpp

```
#include "Analytics.h"
#include "Grid.h"
#include "RandomWalkUtilities.h"

#include <cstdlib>
#include <ctime>

int launchThreads(unsigned int& Steps, unsigned int& SolutionsWanted, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t& Timer,
std::string& Action, std::string& PRightAsText, std::string& PLeftAsText , std::vector<Analytics*>
&ProgramAnalytics);

int threadMain(unsigned int& Steps, unsigned int& SolutionShare, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t& Timer,
std::string& Action,unsigned short int ThisThread, std::string& PRightAsText, std::string& PLeftAsText,
Analytics* ProgramAnalytics);

int main(int argC, char** argV){
    clock_t Timer = clock();
    // Some information is useful even during VERBOSE_MODE
    std::cout<<std::endl<<"VERSION "<<VERSION<<" BEGINNING AT ("<<(float)(clock()-
    Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;

    float ProbabilityLeft;
    float ProbabilityRight;
    int CallSuccessful;
    std::string Action;
    std::string PLeftAsText;
    std::string PRightAsText;
    unsigned int SolutionsWanted;
    unsigned int Steps;
    /* 100 is multiplied by 10 raised to the power of the Probability Scale Factor to give the total
    Probability Points. This enables support for fractional probabilities for straight on, left and
    right. For example if we had straight on = 33.1, right = 33.5 and left = 33.4, we could set the
    scale factor to 1.This would give 1,000 probability points: 331 (straight on), 335 (right) and
    334 (left) = 1,000*/
    unsigned short int ProbabilityScaleFactor;

    // Some information is useful even during VERBOSE_MODE
    CallSuccessful = processCommandLine(argC, argV, *&Steps, *&SolutionsWanted, *&ProbabilityRight,
    *&ProbabilityLeft, *&ProbabilityScaleFactor, *&Action, *&PLeftAsText, *&PRightAsText);

    if(CallSuccessful == FALSE){
        std::cout<<TAB1<<"Problem with processing command line. Quitting..."<<std::endl;
        displayCommandLineInstructions();
        return -1;
        }

    // Seed random number generator for use in initialisation of xoshirostarstar implementation.
    srand(time(0));

    // Initialise custom xoshiro256** random number generator
    xoshiro256ss_init();

    // Initialise class for calculating Analytics
    std::vector<Analytics*> ProgramAnalytics;
    ProgramAnalytics.resize(omp_get_max_threads());

    // Ensure each thread has access to Analytics class, class uses the number of bins and bin range
    defined in RandomWalkCommonHeader.h
    for(unsigned short int i=0; i<omp_get_max_threads(); i++){
        ProgramAnalytics.at(i)=new Analytics(LOWEST_BIN, HIGHEST_BIN, NUMBER_BINS);
        }
        #if ENABLE_MULTIPLE_THREADS()
            launchThreads(Steps, SolutionsWanted, COMPRESSION_FACTOR, ProbabilityRight,ProbabilityLeft,
        ProbabilityScaleFactor, Timer, Action, PRightAsText,PLeftAsText,    ProgramAnalytics);
        #else
            threadMain(Steps,SolutionsWanted, COMPRESSION_FACTOR, ProbabilityRight,ProbabilityLeft,
        ProbabilityScaleFactor, Timer, Action, 0, PrightAsText, PLeftAsText,
    ProgramAnalytics.at(0));
        #endif

   return NORMAL_TERMINATION;
   }

// Intermediate function to execute main() across multiple threads.
int launchThreads(unsigned int& Steps, unsigned int& SolutionsWanted, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t& Timer,
std::string& Action, std::string& PRightAsText, std::string& PLeftAsText,  std::vector<Analytics*>
&ProgramAnalytics){
   unsigned int FirstSolution;
   unsigned int SolutionShare;
   unsigned short int ThisThread;
   unsigned short int Threads;

   #pragma omp parallel default (shared) private(ThisThread, Threads, SolutionShare, FirstSolution)
   {
      ThisThread = omp_get_thread_num();
      Threads = omp_get_num_threads();
      SolutionShare = SolutionsWanted / Threads;
      FirstSolution = ThisThread * SolutionShare;
      if(ThisThread == Threads-1){ SolutionShare = SolutionsWanted - FirstSolution; }

      threadMain(Steps, SolutionShare, Compression, ProbabilityRight, ProbabilityLeft,
    ProbabilityScaleFactor,Timer, Action, ThisThread, PRightAsText, PLeftAsText,
    ProgramAnalytics.at(ThisThread));
   }

   return NORMAL_TERMINATION;
   }

// Thread code split out of main
int threadMain(unsigned int& Steps, unsigned int& SolutionShare, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t& Timer,
std::string& Action,unsigned short int ThisThread, std::string& PRightAsText, std::string&
PLeftAsText ,Analytics* ProgramAnalytics){
   int CallSuccessful;

    // Create and resize grid and its associated vectors of winding angle differences.
    // Note we intentionally pass Steps rather than Steps + 1.
    Grid SolutionGrid(Steps, COMPRESSION_FACTOR, ProbabilityRight, ProbabilityLeft,
    ProbabilityScaleFactor, ThisThread);

    // Some information is useful even during VERBOSE_MODE.
    std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"  INITIALISING
GRID AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<")  seconds."<<std::endl;
    CallSuccessful = SolutionGrid.initialise(Timer, Action);

    if(CallSuccessful == FALSE) {std::cout<<TAB1<<"Problem with initialisation.
    Quitting..."<<std::endl; return -1;}

    unsigned int Solution = 0;
    unsigned int SolutionsFound;
    unsigned int SolutionsWanted = SolutionShare;

    #if ENABLE_SAVE_WINDING_ANGLES() || ENABLE_OUTPUT_WINDING_ANGLES()
```

```
        // Create a vector to hold Winding angles.
        std::vector<float> WindingAngles(SolutionsWanted);
    #endif

    /* Create vector of vectors to hold all solutions.
    Float because we shall be recording winding angles. With more available memory would use doubles.
    Add 1 to capture the centre position.Since no need for paths once we have the winding angles, there
    is an option to discard them as we are going along.*/
    #if ENABLE_SAVE_SUCCESSFUL_PATHS()
        std::vector<std::vector<signed long>> WalkCollection(SolutionsWanted, vector<signed
    long>(Steps+1));
    #else
        std::vector<std::vector<signed long>> WalkCollection(1, vector<signed   long>(Steps+1));
    #endif

    // Some information is useful even during VERBOSE_MODE.
    std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"  LOOKING FOR
SOLUTIONS AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
    // Repeatedly call the grid findWalk function until the require number of solutions have been
    found.
    for(SolutionsFound = 0; SolutionsFound < SolutionsWanted; SolutionsFound++){
        #if ENABLE_VERBOSE_MODE()
            std::cout<<std::endl<<std::endl<<TAB1<<"Looking for Solution
    "<<SolutionsFound+1<<"."<<std::endl;
        #endif

        #if ENABLE_SAVE_SUCCESSFUL_PATHS()
            Solution = SolutionsFound;
        #endif

        WalkCollection.at(Solution) = SolutionGrid.findWalk();

        #if ENABLE_SAVE_WINDING_ANGLES()
            WindingAngles.at(SolutionsFound) = (WalkCollection.at(Solution).at(Steps) /
    pow(10,PRECISION_DECIMAL_POINTS)) * (180 / PI);
        #endif

        #if ENABLE_OUTPUT_ANALYTICS()
            ProgramAnalytics->push((WalkCollection.at(Solution).at(Steps) /
    pow(10,PRECISION_DECIMAL_POINTS)) * (180 / PI));
        #endif

        #if ENABLE_OUTPUT_WINDING_ANGLES()
            std::cout<<std::endl<<"Solution: "<<SolutionsFound+1<<", Winding angle:
    "<<WindingAngles.at(SolutionsFound)<<"."<<std::endl;
        #endif
        }

    // Some information is useful even during VERBOSE_MODE.
    std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"  COMPLETED AT
("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Solutions found: "<<SolutionsFound<<"."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Returned to the centre: "<<(unsigned int)
    SolutionGrid.getReturnsToCentre()<<" times."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Loops encountered: "<<(unsigned int)
    SolutionGrid.getLoopsEncountered()<<"."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Average loop length: "<<(float)
    SolutionGrid.getAverageLoopLength()<<"."<<std::endl;

    #if ENABLE_ANALYSIS_MODE()
        /* Provide information about grid use coverage.
           The results can be used to determine whether we get away with a smaller grid.*/
        SolutionGrid.analyseResults();
        /* Investigation of behavioural variance with Professor's code
           Investigate random generation of changes of direction.*/
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
    VARIANCE ANALYSIS."<<std::endl;
        unsigned long TotalDirectionEvents =
SolutionGrid.GoStraightOn+SolutionGrid.TurnLeft+SolutionGrid.TurnRight+SolutionGrid.OtherDirectionChang
e;
        std::cout<<"Total Direction Events: "<<TotalDirectionEvents<<"."<<std::endl;
        std::cout<<TAB1<<"Turn Left: "<<SolutionGrid.TurnLeft<<" ("<<(float)
(100*SolutionGrid.TurnLeft)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"StraightOn: "<<SolutionGrid.GoStraightOn<<" ("<<(float)
(100*SolutionGrid.GoStraightOn)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"Turn Right: "<<SolutionGrid.TurnRight<<" ("<<(float)
(100*SolutionGrid.TurnRight)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"Other Direction Change: "<<SolutionGrid.OtherDirectionChange<<" ("<<(float)
(100*SolutionGrid.OtherDirectionChange)/TotalDirectionEvents<<"%) ."<<std::endl;
   #endif

   #if ENABLE_SAVE_WINDING_ANGLES()
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"  SAVING
"<<WindingAngles.size()<<" WINDING ANGLES AT ("<<(float)(clock()-   Timer)/CLOCKS_PER_SEC<<")
seconds."<<std::endl;
        std::string WindingAngleResults = getWindingAngleFileName(Steps, SolutionsWanted,
    PRightAsText, PLeftAsText, ThisThread);

        writeWindingAnglesToStream(WindingAngleResults, WindingAngles);
    #endif


    #if ENABLE_OUTPUT_ANALYTICS()
        // Some information is useful even during VERBOSE_MODE.
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"  SAVING
    HISTOGRAM AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;

        std::string HistogramResults = getHistogramFileName(Steps, SolutionsWanted,      PRightAsText,
PLeftAsText, ThisThread);
        std::string AnalyticResults = getAnalyticFileName(Steps, SolutionsWanted,    PRightAsText,
    PLeftAsText, ThisThread);

        ProgramAnalytics->writeHistogramToStream(HistogramResults);
        ProgramAnalytics->outputResults(AnalyticResults, Steps);

        // Some information is useful even during VERBOSE_MODE.
        std::cout<<std::endl<<"Analytics:"<<" THREAD: "<<SolutionGrid.getThread()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Mean Winding Angle: "<<ProgramAnalytics-
    >calculateMean()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Variance: "<<ProgramAnalytics-
    >calculateVariance()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Variance in Radians: "<<ProgramAnalytics-
    >calculateRadianVariance()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Kurtosis: "<<(ProgramAnalytics→calculateKurtosis()
    +3)<<std::endl<<std::endl;
    #endif

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        SolutionGrid.displayExecutionTimes();
    #endif

    return NORMAL_TERMINATION;
    }
```

6.3 Grid.cpp

```
#include "Grid.h"

/* PUBLIC
   Grid Constructor processes parameters passed to main() and then resizes its member vectors to the
appropriate size.*/
Grid::Grid(unsigned int Steps, float Compression, float ProbabilityRight, float ProbabilityLeft,
    unsigned short int ProbabilityScaleFactor, unsigned short int ThreadNumber){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for   initialiseLogger
and setUpProbabilities.
        clock_t LocalTimer = clock();
        initialiseLogger();
    #endif

    // First capture constructor parameters
    Thread = ThreadNumber;
    GridCompression = Compression;
    setUpProbabilities(ProbabilityRight, ProbabilityLeft, ProbabilityScaleFactor);

    /* We add 1 to the length because we want to represent the x = 0 and y = 0 axes withing the grid.We
    add a further 2 to the length to create a 1 grid point safe zone all around the  working grid.
    The objective is to avoid an out of bounds error when we create the East, North, South and West
    vectors.*/

    Length = (unsigned int) (ceil(2*Steps*GridCompression)+1);
    // Note that WalkSteps is 1 bigger than the Steps in RandomWalk::main(). This is so that we can
    capture the centre point.
    WalkSteps = Steps + 1;

    if((Length % 2) == 0) Length += 1;
    GridPoints = Length*Length;
    Centre = (GridPoints - 1)/2;

    Walk.resize(WalkSteps);
    WalkPositionsVisited.resize(WalkSteps);
    GridMapTrace.resize(GridPoints,UNVISITED);
    Reporter.resize(WalkSteps*2, "");

    CentreX = CentreY = (Length-1)/2;

    BoundaryTest = ((3/2)*PiAsUnsignedInt)+1;

    FacingEast->setStepSize(Length);
    FacingNorth->setStepSize(1);
    FacingSouth->setStepSize(-1);
    FacingWest->setStepSize(-Length);
    RetracingSteps->setStepSize(0);

    /* The first [] indicates the direction the walk went in to get to current point,
        The second [] indicates the the walk is leaving in from the current point */
    NewHeadings[0][0] = FacingNorth; // Facing North, straight on
    NewHeadings[1][1] = FacingEast; // Facing East, straight on
    NewHeadings[2][2] = FacingSouth; // Facing South, straight on
    NewHeadings[3][3] = FacingWest; // Facing West, straight on

    NewHeadings[0][1] = FacingEast; // Facing North, turn right (Clockwise)
    NewHeadings[1][2] = FacingSouth; // Facing East, turn right (Clockwise)
    NewHeadings[2][3] = FacingWest; // Facing South, turn right (Clockwise)
    NewHeadings[3][0] = FacingNorth; // Facing West, turn right (Clockwise)

    NewHeadings[0][3] = FacingWest; // Facing North, turn left (Counter clockwise)
    NewHeadings[1][0] = FacingNorth; // Facing East, turn left (Counter clockwise)
    NewHeadings[2][1] = FacingEast; // Facing South, turn left (Counter clockwise)
    NewHeadings[3][2] = FacingSouth; // Facing West, turn left (Counter clockwise)

    NewHeadings[0][2] = RetracingSteps; // Facing North, turning 180 degrees
    NewHeadings[1][3] = RetracingSteps; // Facing East, turning 180 degrees
    NewHeadings[2][0] = RetracingSteps; // Facing South, turning 180 degrees
    NewHeadings[3][1] = RetracingSteps; // Facing West, turning 180 degrees

    // Some information is useful even when ENABLE_VERBOSE_MODE() is set to FALSE.
    std::cout<<TAB2<<"Grid Length: "<<Length<<", Grid Points: "<<GridPoints<<", Walk Steps   including
Step 0 (the centre): "<<WalkSteps<<"."<<std::endl;
    std::cout<<TAB2<<"Centre position: "<<Centre<<", Centre coordinates:
    ("<<CentreX<<","<<CentreY<<")."<<std::endl;

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(0).Invocations += 1;
        Logger.at(0).TotalExecutionTime += (clock() - LocalTimer);
    #endif
    }

/* PUBLIC
   This critical function finds solutions and returns them to the calling function (main()). It does
    not know how many solutions are required. It uses many of Grid's private member functions.*/
std::vector<signed long> Grid::findWalk(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for initialiseWalk,
    reset and Direction::takeOneStep.
        clock_t LocalTimer = clock();
    #endif

    bool SolutionFound;
    bool WalkLoopEncountered;
    signed int CurrentX = 0; // Only used for displaying useful information.
    signed int CurrentY = 0; // Only used for displaying useful information.
    unsigned int LoopCounter = 0;
    unsigned int Position = 0;
    unsigned short int GoingInDirection = 0;
    unsigned short int NowFacing =0; // This represents a different grid point to GoingInDirection.
    Together they form a path across a grid point.

    Direction* FacingDirection;

    SolutionFound = FALSE;
    while(SolutionFound == FALSE){

        /* Set all Grid points to unvisited (0) apart from the centre (starting) position (0,0) which
    should be set to VISITED_TWICE (900). Note that we call this even for "Step 0, when the only affect
is to set the centre (starting) position. This reset will also be executed if we have to     restart
following a loop encountered event.*/
        reset(*&LoopCounter);
        Position = WalkPositionsVisited.at(0) = Centre;

        if(initialiseWalk(*&GoingInDirection, *&NowFacing, *&Position) == FALSE)
    fatalError("Grid::findWalk. Inititalise Walk failed.");

        FacingDirection = NewHeadings[GoingInDirection][NowFacing];

        #if ENABLE_VERBOSE_MODE()
            updateCoordinates(*&CurrentX, *&CurrentY, *&WalkPositionsVisited.at(0));
            std::string News = "     THREAD: " + to_string(Thread) + ". Step: 0. Centre
    (starting) position:" + to_string(WalkPositionsVisited.at(0));
            News = News + " (" + to_string(CurrentX) + "," + to_string(CurrentY) + ").
    Position status after initialisation: " +
    to_string(GridMapTrace.at(WalkPositionsVisited.at(0))) + ".\n";
```

```
            News = News + TAB3 + "Change in winding angle: " + to_string(Walk.at(0)) + ".
    Cumulative winding angle: " + to_string(Walk.at(0)) + ".\n";
            News = News + TAB3 + "Going " + getDirectionAsText(GoingInDirection) + ".";
            Reporter.at(0) = News;
        #endif

        LoopCounter = 1;
        WalkLoopEncountered = FALSE;
        while( (WalkLoopEncountered == FALSE) && (LoopCounter < WalkSteps) ){
            FacingDirection = FacingDirection->takeOneStep(LoopCounter, *&Position,
    Walk.at(LoopCounter-1), *&Walk.at(LoopCounter), *&WalkLoopEncountered);

            // We record the positions visited so that we can reset GridMapTrace after each
    solution is found.
            WalkPositionsVisited.at(LoopCounter) = Position;

            LoopCounter++;
            }

        /* We need to reset the grid following the successful identification of a solution       or on
encountering a loop.*/
        reset(*&LoopCounter);
        if(LoopCounter == WalkSteps) SolutionFound = TRUE;
        if(WalkLoopEncountered == TRUE) SolutionFound = FALSE;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(6).Invocations += 1;
        Logger.at(6).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    #if ENABLE_VERBOSE_MODE()
        for(int NewsItem = 0; NewsItem < Reporter.size(); NewsItem++)
    std::cout<<Reporter.at(NewsItem);
    #endif

    return Walk;
    }

/* PUBLIC
   This critical function determines the winding angle for each grid element.
   */
int Grid::initialise(clock_t& Timer, std::string Action){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for serialise and
loadWindingAngles.
        clock_t LocalTimer = clock();
    #endif

    signed int VectorSize;
    signed int XCoordinate;
    signed int YCoordinate;
    unsigned int AbsoluteX;
    unsigned int AbsoluteY;

    if((Action == "serialise")|(Action =="RAM")){ // The only difference between serialise and RAM is
that RAM doesnt output to file.
        std::cout<<TAB1<<"Creating angle change data at ("<<(float)(clock()-
    Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
        GridMap.resize(GridPoints);

        for(unsigned int LoopCounter = 0; LoopCounter < GridPoints; LoopCounter++){
            AbsoluteY = LoopCounter % Length;
            AbsoluteX = (LoopCounter - AbsoluteY) / Length;
            XCoordinate = AbsoluteX - CentreX;
            YCoordinate = AbsoluteY - CentreY;

            GridMap.at(LoopCounter) = floor((atan2(YCoordinate, XCoordinate)+PI) * pow(10,
    PRECISION_DECIMAL_POINTS));
            }

        // We want the centre (starting) point to have a value of 0.
        GridMap.at(Centre) = 0;

        if(Action == "serialise"){
            std::cout<<TAB1<<"Serealising grid map angles at ("<<(float)(clock()-
    Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
            serialise();
            }
        else{
            std::cout<<TAB1<<"Grid map angles (into RAM) at ("<<(float)(clock()-
    Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
            }
        }
    else if(Action == "load"){
        std::cout<<TAB1<<"Loading angle change data at ("<<(float)(clock()- Timer)/CLOCKS_PER_SEC<<")
seconds."<<std::endl;
        VectorSize = loadAngleChanges();
        if(VectorSize != GridPoints){std::cout<<TAB2<<"ERROR: Unexpected vector size
    ("<<VectorSize<<") vs. expected size("<<GridPoints<<")."<<std::endl;

        #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
            Logger.at(3).Invocations += 1;
            Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
        #endif

        return FALSE;}
        }
    else{
        std::cout<<TAB2<<"ERROR: Unrecognised initialisation       action:"<<Action<<"."<<std::endl;

        #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
            Logger.at(3).Invocations += 1;
            Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
        #endif

        return FALSE;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(3).Invocations += 1;
        Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return TRUE;
    }

/* PRIVATE
   We need to determine our initial direction of movement. We also need to ensure that the cumulative
    winding angle is 0 at the centre (Step 0).*/
int Grid::initialiseWalk(unsigned short int& GoingInDirection, unsigned short int& NowFacing, unsigned
int& Position){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    NowFacing = NORTH;
    GoingInDirection = NORTH;
    Walk.at(0) = (signed int) (0-GridMap.at(Position+1));

    FirstStep = NowFacing;
    GridMapTrace.at(Centre) = CENTRE_INITIAL_STATE;

   /* We need to ensure that the Direction Class static class variables which track rotation are reset. We
     can use the derived class chosen for Step 1.*/
    NewHeadings[GoingInDirection][NowFacing]->reset();

    Reporter.clear();
    Reporter.resize((WalkSteps*2));

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(7).Invocations += 1;
        Logger.at(7).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return TRUE;
    }

/* PRIVATE
    Load winding angle changes from file.
    Called from initialise()*/
signed int Grid::loadAngleChanges(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    long FileSize;
    size_t Index = 0;
    std::string FileName = "Grid_ST" + to_string(WalkSteps-1) + "CM" + to_string(COMPRESSION_FACTOR);

   while(TRUE){
      Index = FileName.find(".", Index);
      if(Index == std::string::npos) break;
      FileName.replace(Index, 1, "dot");
        }

    FileName = FileName + ".Map";

    ifstream GridMapIn(FileName, ios::binary);

    if(!GridMapIn.good()){
        fatalError("Grid::loadAngleChanges. Attempt to load file " + FileName + " failed.");
        }
    else{
        GridMapIn.seekg(0,ifstream::end);
        FileSize = GridMapIn.tellg();
        GridMapIn.seekg(0, ifstream::beg);
        GridMap.resize(FileSize / sizeof(unsigned int));
        GridMapIn.read((char*)&GridMap.at(0),FileSize);
        GridMapIn.close();
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(5).Invocations += 1;
        Logger.at(5).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return FileSize / sizeof(signed int);
    }

/* PRIVATE
```

```
   At the start of each attempt to find a solution we must ensure that each grid position is set to
    zero apart from the centre (starting) position.*/
void Grid::reset(unsigned int& StepsCompleted){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    // It is a great deal quicker to reset the grid positions visited during the previous solution /
    attempt than reset the entire grid each time.
    for(unsigned int LoopCounter = 0; LoopCounter < StepsCompleted; LoopCounter++){
        GridMapTrace.at(WalkPositionsVisited.at(LoopCounter)) = UNVISITED;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(8).Invocations += 1;
        Logger.at(8).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }

/* PRIVATE
   Writes all grid angles to file.
   Called from initialise() function when Action set to serialise.*/
void Grid::serialise(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    size_t Index = 0;
    std::string FileName = "Grid_ST" + to_string(WalkSteps-1) + "CM" + to_string(COMPRESSION_FACTOR);

   while(TRUE){
     Index = FileName.find(".", Index);
     if(Index == std::string::npos) break;
     FileName.replace(Index, 1, "dot");
        }

    FileName = FileName + ".Map";

    ofstream GridMapOut(FileName, ios::out | ios::binary);
    GridMapOut.write((const char*)&GridMap.at(0), GridMap.size() * sizeof(unsigned int));
    GridMapOut.close();

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(4).Invocations += 1;
        Logger.at(4).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }

/* PRIVATE
   By creating an array of probability weighted direction outcomes at the outset, we improve subsequent
efficiency during searcing for solutions. The scale factor lets us deal with non integer
probabilities.*/
void Grid::setUpProbabilities(float ProbabilityRight, float ProbabilityLeft, unsigned short int
ProbabilityScaleFactor){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    // To simplify the code and reduce typing later, we first create and calculate the value of a
    probality multiplier.
    unsigned int ProbabilityMultiplier = pow((unsigned short int)10, (unsigned short
    int)ProbabilityScaleFactor);

    // Now we scale ProbabilityPoints according to the ProbabilityMultiplier.
    ProbabilityPoints = ProbabilityPoints * ProbabilityMultiplier;

    // In case more decimal places have been used than are multiplied out by the scale factor, we use
    floor the left and right probabilities.
    DirectionChoices.resize(floor(ProbabilityRight*ProbabilityMultiplier), GO_RIGHT);
    DirectionChoices.resize(floor((ProbabilityLeft +   ProbabilityRight)*ProbabilityMultiplier),
GO_LEFT);
    DirectionChoices.resize((ProbabilityPoints), GO_STRAIGHT_ON);

    std::cout<<TAB2<<"Probability Points: "<<ProbabilityPoints<<"."<<std::endl;

    #if ENABLE_VERBOSE_MODE()
        std::cout<<TAB3<<" DirectionChoices.size():        "<<DirectionChoices.size()<<"."<<std::endl;
        std::cout<<TAB3<<" Direction[0]: "<<DirectionChoices.at(0)<<". ";

        if(ProbabilityRight > 0 && ProbabilityRight < 100){
            // Last Right is before position ProbabilityPoints -1.
            std::cout<<" Direction["<<(ProbabilityRight*ProbabilityMultiplier)-1<<"]:
    "<<DirectionChoices.at((ProbabilityRight*ProbabilityMultiplier)-1)<<". ";
            }

        if(ProbabilityRight > 0 && ProbabilityLeft > 0){
            // First Left is not position 0.
            std::cout<<" Direction["<<(ProbabilityRight*ProbabilityMultiplier)<<"]:
    "<<DirectionChoices.at(ProbabilityRight*ProbabilityMultiplier)<<". ";
        }

        if(ProbabilityLeft > 0 && (ProbabilityRight + ProbabilityLeft) < 100){
            // Last Left is before position ProbabilityPoints -1.
            std::cout<<"  Direction["<<((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)-
        1<<"]:"<<DirectionChoices.at(((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)-
    1)<<". ";
        }

        if((ProbabilityRight + ProbabilityLeft) > 0 && (ProbabilityRight + ProbabilityLeft) < 100){
            // First Straight On is not position 0.
            std::cout<<"  Direction["<<(ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier<<"]:
    "<<DirectionChoices.at((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)<<".     ";
            }

        std::cout<<" Direction["<<(ProbabilityPoints-1)<<"]:
    "<<DirectionChoices.at(ProbabilityPoints-1)<<"."<<std::endl;
    #endif

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(2).Invocations += 1;
        Logger.at(2).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }
```

6.4 Direction.cpp

```
#include "Direction.h"
#include "Grid.h"

// Define static member variables
std::vector<bool> Direction::JustLeftCentre;
signed int Direction::CentreTransitChanges[4][4][2] = { { {-ThreeSixty,ThreeSixty},{-Ninety,-Ninety},{-
OneEighty,OneEighty},{Ninety,Ninety} },
                                                         { {Ninety,Ninety},{-ThreeSixty,ThreeSixty},{-
Ninety,-Ninety},{-OneEighty,OneEighty} },
                                                         { {-OneEighty,OneEighty},{Ninety,Ninety},{-
ThreeSixty,ThreeSixty},{-Ninety,-Ninety} },
                                                         { {-Ninety,-Ninety},{-OneEighty,OneEighty},
{Ninety,Ninety},{-ThreeSixty,ThreeSixty} } };
std::vector<signed int> Direction::PartialRotationEnteringTransit;
std::vector<signed int> Direction::TransitChange;

/* PUBLIC
   Direction Constructor. By setting its member variables equal to passed parameters, its subclasses
get their differemt behaviours.*/
Direction::Direction(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int Adjustment){
    GoingInDirection = CompassPoint;
    CameFromDirection = (GoingInDirection + 2) % 4;
    SolutionGrid = ThisGrid;
    this->Adjustment=Adjustment;

    if(JustLeftCentre.size()!=omp_get_max_threads()) JustLeftCentre.resize(omp_get_max_threads(),
FALSE);
    if(PartialRotationEnteringTransit.size()!=omp_get_max_threads())
PartialRotationEnteringTransit.resize(omp_get_max_threads(), 0);
    if(TransitChange.size()!=omp_get_max_threads()) TransitChange.resize(omp_get_max_threads(), 0);
    }

/* PUBLIC
   Derived class constructors simply invoke the base class (Direction) constructor.*/
DirectionError::DirectionError(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int
Adjustment) : Direction(ThisGrid, CompassPoint, Adjustment){}
East::East(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
North::North(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
South::South(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
West::West(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}

/* PUBLIC
   This function is needed to reset the Direction classes static state variables which are shared by
all sub class instances.*/
void Direction::reset(){
    JustLeftCentre[SolutionGrid->Thread] = FALSE;
    PartialRotationEnteringTransit[SolutionGrid->Thread] = 0;
    TransitChange[SolutionGrid->Thread] = 0;
    }

/* PUBLIC
   This function is needed because Length isn't known when the subclasses of Direction are created and
the Direction constructor is called.*/
void Direction::setStepSize(unsigned int Length){
    StepSize = Length;
    }
```

```
/* PUBLIC
   This key function is not overriden by the derived classes East, North, South and West.
   However, the derived classes have different behaviour due the specific values of their member
attributes.*/
Direction* Direction::takeOneStep(unsigned int& Step, unsigned int& Position, signed long&
CurrentWindingAngle, signed long& UpdatedWindingAngle, bool& WalkLoopEncountered){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for initialiseWalk,
    getNewDirection navigate and reset.
        clock_t LocalTimer = clock();
    #endif

    #if ENABLE_VERBOSE_MODE() || ENABLE_ANALYSIS_MODE()
        std::string News;
        signed int CurrentX = 0; // Only used for displaying useful information.
        signed int CurrentY = 0; // Only used for displaying useful information.
        std::string ArrivalText = ". ERROR: Somehow this text has not been updated.          Position:
";
    #endif

    signed long ChangeInWindingAngle;

    // 1. Update GridMapTrace.at(Position) with GoingInDirection.
    SolutionGrid->GridMapTrace.at(Position) += GoingInDirection;
    #if ENABLE_VERBOSE_MODE()
        //std::cout<<" Position status on leaving: "<<SolutionGrid-
    >GridMapTrace.at(Position)<<"."<<std::endl;
        News="Postion status on leaving: " + to_string(SolutionGrid-    >GridMapTrace.at(Position)) +
"\n\n";
        SolutionGrid->Reporter.at((2*Step)-1)=News;
    #endif

    // 2. Take Step
    Position += StepSize;

    // 3. Evaluate the transit history of the new position.
    if(Position != SolutionGrid->Centre){
        // We are not back at the centre.
        if(SolutionGrid->GridMapTrace.at(Position) == UNVISITED){
            // 3.a Arriving at this off centre position for the first time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". New position: ";
            #endif

            // 3.a.1. Calculate the change in winding angle.
            // 3.a.1.i. Determine if this is the very first step.
            if(Step == 1){
                ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
    (signed long)SolutionGrid->GridMap.at(Position-StepSize);
                /* Ensure that future steps know we have not just left the centre.
                   Note that the following line is not really required since reset() sets
     JustLeftCentre to FALSE,as does the JustLeftCentre.resize() call in the
    constructor.*/
                JustLeftCentre[SolutionGrid->Thread] = FALSE;
                }
            // 3.a.1.ii. Determine if we have just passed back through the centre.
            else if (JustLeftCentre[SolutionGrid->Thread] == TRUE){
                ChangeInWindingAngle = TransitChange[SolutionGrid->Thread] +
    PartialRotationEnteringTransit[SolutionGrid->Thread];

                // Ensure that future steps know we have not just left the centre.
                JustLeftCentre[SolutionGrid->Thread] = FALSE;
                }
            // 3.a.1.iii. An ordinary step.
            else{
                ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
        (signed long)SolutionGrid->GridMap.at(Position-StepSize);

                // We need to adjust for any movement across the 0:2*Pi boundary
                if(ChangeInWindingAngle > SolutionGrid->BoundaryTest) ChangeInWindingAngle =
        ChangeInWindingAngle - (signed long)(2*SolutionGrid->PiAsUnsignedInt);
                else if(ChangeInWindingAngle < -SolutionGrid->BoundaryTest)
        ChangeInWindingAngle = ChangeInWindingAngle + (signed long)                 (2*SolutionGrid-
>PiAsUnsignedInt);
                }

            /* 3.a.2. Use CameFromDirection (equals GoingInDirection looking backwards).
               NOTE: This is plain =, not += . This is different from below.*/
            SolutionGrid->GridMapTrace.at(Position) = CameFromDirection;

            // 3.a.3. Change Direction with random selection.

            #if ENABLE_ANALYSIS_MODE()
                /* Investigation of behavioural variance with Professor's code.
                Investigate random generation of changes of direction.*/
                Choice =
        SolutionGrid→DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
        >ProbabilityPoints));
                switch (Choice){
                    case GO_STRAIGHT_ON: SolutionGrid->GoStraightOn += 1; break;
                    case GO_RIGHT: SolutionGrid->TurnRight += 1; break;
                    case GO_LEFT: SolutionGrid-> TurnLeft += 1;break;
                    default: SolutionGrid->OtherDirectionChange += 1;
                }

                if(Choice == GO_STRAIGHT_ON) NowFacing = GoingInDirection;
                else NowFacing = (GoingInDirection + Choice + Adjustment) % 4;
            #else
                Choice = SolutionGrid-
        >DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
        >ProbabilityPoints));
                // The normal case. No analysis. No testing. Just a standard run
                if(Choice == GO_STRAIGHT_ON) NowFacing =GoingInDirection;
                else NowFacing = (GoingInDirection + Choice +Adjustment) % 4;
            #endif

            }
        else{
            // 3.b. Arriving at this off centre position for the second time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". Previously visited off centre position: ";
            #endif

            // 3.b.1. Calculate the change in winding angle. Note we do not need to test for STEP == 0
            // 3.b.1.i. Determine if we have just passed back through the centre.
            if (JustLeftCentre[SolutionGrid->Thread] == TRUE){
                ChangeInWindingAngle = TransitChange[SolutionGrid->Thread] +
        PartialRotationEnteringTransit[SolutionGrid->Thread];

                // Ensure that future steps know we have not just left the centre.
                JustLeftCentre[SolutionGrid->Thread] = FALSE;
                }
            // 3.b.1.ii. An ordinary step.
            else{
                ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
        (signed long)SolutionGrid->GridMap.at(Position-StepSize);

                // We need to adjust for any movement across the 0:2*Pi boundary
            if(ChangeInWindingAngle > SolutionGrid->BoundaryTest) ChangeInWindingAngle =
    ChangeInWindingAngle - (signed long)(2*SolutionGrid->PiAsUnsignedInt);
            else if(ChangeInWindingAngle < -SolutionGrid->BoundaryTest)
    ChangeInWindingAngle = ChangeInWindingAngle + (signed long)(2*SolutionGrid-
    >PiAsUnsignedInt);
            }

        /* 3.b.2. Use CameFromDirection (equals GoingInDirection looking backwards).
           NOTE: This is +=, not plain = . This is different from above.*/
        SolutionGrid->GridMapTrace.at(Position) += CameFromDirection;

        // 3.b.3. Change Direction by subtraction from VISITED_TWICE.
        NowFacing = VISITED_TWICE - SolutionGrid->GridMapTrace.at(Position);
        }
    }
    else{
        // 3.c.1. For testing /analysis purposes we keep track of all the times we have  returned to
the Centre for the entire run.
        SolutionGrid->ReturnsToCentre += 1;

        // 3.c.2. Determine the change in winding angle. For the return to centre step, this is the
    partial or fractional loop rotation value.
        ChangeInWindingAngle = -(CurrentWindingAngle) % (2*SolutionGrid->PiAsUnsignedInt);

        /* 3.c.3. Use CameFromDirection (equals GoingInDirection looking backwards).
           NOTE: This is +=, not plain = . This is different from above.*/
        SolutionGrid->GridMapTrace.at(Position) += CameFromDirection;

        // 3.c.4. The normal case. No analysis. No testing. Just a standard run.
        Choice = SolutionGrid→DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
    >ProbabilityPoints));
        // The normal case. No analysis. No testing. Just a standard run
        if(Choice==GO_STRAIGHT_ON)NowFacing =GoingInDirection;
        else NowFacing = (GoingInDirection + Choice +Adjustment) % 4;

        // 3.c.5. Determine whether we are back for the first time (or the second and last time).
        if(SolutionGrid->GridMapTrace.at(Position) < VISITED_TWICE && (NowFacing != SolutionGrid-
    >FirstStep)){
            // 3.c.5.i. We are back at the centre for the first time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". Back at Centre (Starting) position for the first time: ";
            #endif

            // 3.c.5.i.2. We need to determine the angle of rotation at the point of reaching the
        centre.
            unsigned short int DirectionOfRotation;
            if(CurrentWindingAngle > 0) DirectionOfRotation = COUNTER_CLOCKWISE; else
    DirectionOfRotation = CLOCKWISE;

            // 3.c.5.i.3. We need to select the change of direction that must be supplied and store it
             where it can be obtained next step.
            TransitChange[SolutionGrid->Thread] = CentreTransitChanges[CameFromDirection]
    [NowFacing][DirectionOfRotation];

            // 3.c.5.i.4. We need to store the partial angle where it can be obtained next step.
            PartialRotationEnteringTransit[SolutionGrid->Thread] = (signed int)              -
ChangeInWindingAngle;

            // 3.c.5.i.5. Finally, we need to flag for the next step that we have just leftthe centre.
            JustLeftCentre[SolutionGrid->Thread] = TRUE;
            }
        else{
            // 3.c.5.ii. We are back at the centre for the second time.
            #if ENABLE_VERBOSE_MODE()
```

```
                ArrivalText = ". Back at Centre (Starting) position for the second time: ";
            #endif

            // 3.c.5.ii.1. In order for both the display of NowFacing and the return line to function
             properly, we must have a valid NowFacing.*/
            NowFacing = CameFromDirection;

            // 3.c.5.ii.2 We must set the LoopEncountered flag = TRUE.
            WalkLoopEncountered = TRUE;

            // 3.c.5.ii.3 Increment LoopsEncountered and, for interest, update the aggregate loop
        length so that we can later calcuate the average.
            SolutionGrid->LoopsEncountered += 1;
            SolutionGrid->AggregateLoopLength += Step;
            }
        }

    // 4. Update Winding Angle
    UpdatedWindingAngle = (signed long) (CurrentWindingAngle + ChangeInWindingAngle);

    // 5. Return New Direction
    #if ENABLE_VERBOSE_MODE()
        SolutionGrid->updateCoordinates(*&CurrentX, *&CurrentY, *&Position);
        News = "     THREAD: " + to_string(SolutionGrid->Thread) + ". Step: " +  to_string(Step) +
    ArrivalText + to_string(Position);
        News = News + " (" + to_string(CurrentX) + "," + to_string(CurrentY) + "). Came from the " +
    SolutionGrid->getDirectionAsText(CameFromDirection);
        News = News + ". Position status having arrived: " + to_string(SolutionGrid-
    >GridMapTrace.at(Position)) + ".\n";
        News = News + TAB3 + "Change in winding angle: " + to_string(ChangeInWindingAngle) + ".
    Cumulative winding angle: " + to_string(UpdatedWindingAngle) + " (";
        News = News + to_string((float)(UpdatedWindingAngle /   (pow(10,PRECISION_DECIMAL_POINTS)) *
    (180 / PI))) + ".\n";
        if(ArrivalText != ". Back at Centre (Starting) position for the second time: "){
            News = News + TAB3 + "Going " + SolutionGrid->getDirectionAsText(NowFacing) + ".";
            }
        SolutionGrid->Reporter.at(2*Step) = News;
    #endif

    #if ENABLE_ANALYSIS_MODE()
        SolutionGrid->updateCoordinates(*&CurrentX, *&CurrentY, *&Position);
        SolutionGrid->trackGridUse(Step, CurrentX, CurrentY);
    #endif

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        SolutionGrid->Logger.at(9).Invocations += 1;
        SolutionGrid->Logger.at(9).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return SolutionGrid->NewHeadings[GoingInDirection][NowFacing];
    }

/* PUBLIC
   We need to override the base class virtual function to call fatalError in this case. All other
derived classes use the base class virtual function.*/
Direction* DirectionError::takeOneStep(unsigned int& Step, unsigned int& Position, signed long&
CurrentWindingAngle, signed long& UpdatedWindingAngle, bool& WalkLoopEncountered){
    fatalError("DirectionError::takeOneStep. Requested to retrace steps.");
    return this;
    }
```

6.5 Analytics.cpp

```
#include "Analytics.h"

#include <cmath>
#include <iostream>

/* PUBLIC
   Analytics Constructor uses its parameters to resize BinData and set BindWidth.*/
Analytics::Analytics(double Left, double Right, unsigned int Bins) : LeftBoundary(Left),
RightBoundary(Right), NumberOfBins(Bins){
    clear();

    if(Right <= Left) fatalError("Histogram::Histogram(). Called with right <= left.");
    if(Bins == 0) fatalError("Histogram::Histogram(). Called with nBins == 0.");

    BinWidth = (Right-Left) / Bins;

    /* Histogram will contain nBins between left and right.
       There will also be 2 overspill bins, one for data < left, and one for data >= right.*/
    BinData.resize(Bins+2);
    }

/* PUBLIC
   Calculate Kurtosis of winding angles.*/
const double Analytics::calculateKurtosis(){
    return double ((NumberOfAngles)*M4 / (M2*M2)) - 3.0;
    }

/* PUBLIC
   Calculate Mean of winding angles.*/
const double Analytics::calculateMean(){
    return M1;
    }

/* PUBLIC
   Calculate Variance of winding angles in radians.*/
const double Analytics::calculateRadianVariance(){
    return radian_M2/(NumberOfAngles-1.0);
    }

/* PUBLIC
   Calculate Skewness of winding angles.*/
const double Analytics::calculateSkewness(){
    return sqrt(double(NumberOfAngles)) * M3 / pow(M2, 1.5);
    }

/* PUBLIC
   Calculate Standard Deviation of winding angles.*/
const double Analytics::calculateStandardDeviation(){
    return sqrt( calculateVariance() );
    }

/* PUBLIC
   Calculate Variance of winding angles.*/
const double Analytics::calculateVariance(){
    return M2/(NumberOfAngles-1.0);
    }

/* PUBLIC
   Clear intermediates and number of data values.*/
void Analytics::clear(){
    NumberOfAngles = 0;
    M1 = M2 = M3 = M4 = radian_M2 = 0.0;

    return;
    }

/* PUBLIC
   Format results in Histogram.*/
const void Analytics::getHistogramResults(vector<double>& binBoundaries, vector<long>& binData){
    binBoundaries.resize(NumberOfBins + 2);

    for (unsigned int i = 0; i <= NumberOfBins; ++i){
        binBoundaries[i] = LeftBoundary + static_cast<double>(i) * BinWidth;
        }

    binBoundaries[NumberOfBins + 1] = VERY_BIG;

    binData = BinData;    //  Copy the whole vector

    return;
    }

/* PUBLIC
   Output Analytics to Stream.*/
void Analytics::outputResults(std::string AnalyticResults, unsigned int Steps){
    ofstream AnalyticResultsOutput(AnalyticResults);

    std::cout<<TAB1<<AnalyticResults<<std::endl;

    if (!AnalyticResultsOutput){
        std::cout<<"Failed to open the file."<<std::endl;
        }
    else{
        AnalyticResultsOutput << "Excess_Kurtosis" << ", " << "Kurtosis" << ", " << "Variance" << ", "
<< "Radian_Variance" << ", " << "Mean" << ", " << "Steps" << std::endl;
        AnalyticResultsOutput << calculateKurtosis() << ", " << calculateKurtosis()+3 << ", " <<
calculateVariance() << ", " << calculateRadianVariance() << ", " << M1 << ", " << Steps << std::endl;
        AnalyticResultsOutput.close();
        }

    return;
    }

/* PUBLIC
   Calculate intermediate terms for analytics without storing individual winding angles.*/
void Analytics::push(float x){
    double delta, delta_n, delta_n2, term1, radian_term1;
    long long n1 = NumberOfAngles;
    NumberOfAngles++;
    delta = x - M1;
    delta_n = delta / NumberOfAngles;
    delta_n2 = delta_n * delta_n;
    term1 = delta * delta_n * n1;
    radian_term1 = (delta*(PI/180)) * (delta_n*(PI/180)) * n1;
    M1 += delta_n;
    M4 += term1 * delta_n2 * (NumberOfAngles*NumberOfAngles - 3*NumberOfAngles + 3) + 6 *    delta_n2 *
M2 - 4 * delta_n * M3;
    M3 += term1 * delta_n * (NumberOfAngles - 2) - 3 * delta_n * M2;
    M2 += term1;
    radian_M2 += radian_term1;

    //Histogram: Note that if x lies on a boundary, then it gets put in the bin to the   RIGHT.
    if ( x < LeftBoundary ){
        ++BinData[0];
    }
    else if ( x >= RightBoundary ){
        ++BinData[NumberOfBins + 1];
    }
    else{
        const int iBin = static_cast<int>((x - LeftBoundary) / BinWidth) + 1;
        ++BinData[iBin];
        }

    return;
    }

/* PUBLIC
   Return the number of data values pushed to Analytics Class.*/
const long long Analytics::returnNumberOfDataValues(){
    return NumberOfAngles;
    }

/* PUBLIC
   Output Histogram to Stream.*/
const void Analytics::writeHistogramToStream(std::string HistogramResults){
    ofstream HistogramResultsOutput(HistogramResults);

    std::cout<<TAB1<<HistogramResults<<std::endl;

    if(!HistogramResultsOutput){
        std::cout<<"Failed to open the file."<<std::endl;
        }
    else{
        vector<double> binBoundaries;
        vector<long> binData;
        getHistogramResults(binBoundaries, binData);

        vector<double>::const_iterator BoundaryIterator = binBoundaries.begin();
        vector<long>::const_iterator DataIterator = binData.begin();

        for( ; BoundaryIterator != binBoundaries.end(); ++BoundaryIterator, +
    +DataIterator ){
            HistogramResultsOutput << setw(10) << *BoundaryIterator << ", " <<
    setw(10) << *DataIterator << std::endl;
            }

        HistogramResultsOutput.close();
        }

    return;
    }
```