The Statistics of Random Walks

Habib Rehman

June/July 2015

Contents

1 Exposition **1**
1.1 Aims and Objectives 1
1.2 Context 2
1.3 Selection of approach 2

2 Abstract **3**

3 Introduction **3**
3.1 Simple Random Walk on a $\mathbb{Z}$ lattice. 4
3.2 Loop Erased Random Walks (LERW) 5

4 Methodology **5**
4.1 Random Walk Implementation 6
4.2 Collecting Data 10

5 Results and Analysis **11**

6 Conclusion **15**
6.1 Findings 15
6.2 Implications 16

A Appendices **16**
A.1 Imports 16
A.2 System Definition code 16
A.3 Random Walk generation code 17
A.4 Loop Erased Random Walk code 17
A.5 Random Walk plot code (using the matplotlib API) 18
A.6 Loop Erased Random Walk plot code 18
A.7 Histogram plot (and parse from raw2bin.c) code 18

B Bibliography **19**

C Acknowledgements **20**

1 Exposition

This section act as an exposition to the research project.

1.1 Aims and Objectives

The main aims of this research project were to:

1. Identify the correspondence between the return distribution of a 1D random walk and a 2D random walk
2. Consider the correspondence between the arc length distribution of a random walker returning

The objectives for both aims were the following:

1. Explore the physical phenomena (i.e. a stochastic process) that implicates a random walk
2. Devise a mathematical model that models the physical phenomena
3. Precisely delineate the properties of the mathematical model and all aspects regarding it
4. Translate the mathematical model into `Python` code to implement the model
5. Collect data by running the code with nuanced configurations of the mathematical model
6. Analyse the data and draw conclusions to obtain results

1.2 Context

Random walks are applied in a range of fields such as physics, financial economics, population genetics and computer science. Random walks are essential mathematical models used to model stochastic processes which are ubiquitously present in many fields. Any natural process such as diffusion is a stochastic process as it occurs randomly and is thought to be non-deterministic (unpredictable). However, modelling the process as a random walks enables us to deduce the statistics of certain behaviour of the process and thus predict the outcome (to a certain extent) to of an event which implicates the process. The ability to predict this for stochastic processes is of great importance as it provides actionable information that can be crucial.

For instance, in finance, random walks are used to model stock prices, by considering the change of the stock prices with respect to time. Where the stock prices and the time are the independent random variables that form a 2D random walk.

In physics, random walks are essential in predicting how fast one gas will diffuse into another, how fast heat will spread in a solid, how big fluctuations

in pressure will be in a small container, and many other statistical phenomena. They were used by Albert Einstein to find the size of atoms from the Brownian motion exhibited by atoms.

Additionally, in computer science, random walks are used to improve the scalability networks such as Tor [2] by allowing efficient allocation of resources which prevents network latency and prevent network failure. This example has very real impacts on the users of the network as it dictates the time that they have to invest in order to receive a response to their request.

1.3 Selection of approach

The following are the proposed approaches initially considered out which the most effective approach was selected.

Approaches:

1. Model the stock price change (with respect to time) data for a particular stock as a random walk, erase loops and then identify the correspondence between the return and arc length distributions. All implementation in `Python`
2. Model the Tor network bandwidth usage (with respect to time) data as a random walk and then identify the correspondence between the return and arc length distributions. All implementation in `Python`
3. Model pseudorandomly generated sequence(s) of independent variables as a random walk and then identify the correspondence between the return and arc length distributions. All implementation `C++`
4. Model a pseudorandomly generated sequence(s) of independent variables as a random walk and then identify the correspondence between the return and arc length distributions. All implementation in `Python`

Selection I selected approach 4. because it was the most practical, feasible and reproducible approach. While approach 4. doesn't use true random data like approach 1. or 2. , it does provide reproducibility as the seed used to generate pseudorandom sequence always yields the same sequence for the particular seed and thus the results can be verified by a third-party. Data obtained from approach 1. could be subject to manipulation (stock manipulation) and thus yield in unreliable results and selecting a suitable stock would pose non-problem oriented challenges. While approach 2. addresses the problems with approach 1. by definitively yield true random data [1], it is impractical to gather the required amount of data in the time available for the research and it would also pose additional non-problem oriented challenges. Finally, approach 4. was selected over approach 3. because of its use of the `Python` programming language instead

[1] As the Tor network is fully decentralised and extensive enough in size that it is infeasible for any single entity to cause significant disturbances in the network

of the `C++` programming language. This is because `Python`, being a high-level language, already provided much of functionality (more extensive standard library) that we needed where as in `C++` we would have had to implement most of that needed functionality. So, `Python` enabled us to stay problem oriented and accomplish our objectives faster.

2 Abstract

The purpose of this research has been to identify the precise correspondence between the return distribution of a one-dimensional random walk and a two-dimensional random walk. This research also considers the correspondence between the arc length distribution of a random walker returning and the arc length distribution of the loops erased.

3 Introduction

A random walk is a stochastic process formed by the repeated summation of identically distributed (independent) random variables distributed random variables and regarded to be a is a V-valued Markov chain on a graph [7]. This research considers random walks on a integer lattice $\mathbb{Z}^d$ as the discrete space, practically, allows for more analysis (operations) to be carried out with a feasible computational power. The research is also concerned with loop erased random walks in the second dimension (i.e. $d = 2$) as it is below the critical dimension for random walks $d = 4$ at which the process converges to Brownian motion [5] and scaling limits exist (i.e. the limit as the lattice spacing approaches zero). It becomes trivial in the regard that self-intersection becomes highly improbable. The random walks studied correspond to increment distributions with the properties of having zero mean and finite variance. These properties are necessary for normal convergence in the increments of the distributions to occur. They also allow for the succinct expression of the main result quickly. Firstly, I will delineate the statistical behaviour of a simple random walk in what follows. Thereafter, I will provide the appropriate theorems and definition for random walks and loop erased random walks.

3.1 Simple Random Walk on a $\mathbb{Z}$ lattice.

The simplest non-trivial case is to let $X_1, X_2, \ldots, X_{\mathrm{n}}$ represent the outcomes of independent experiments and is in the first dimension. Suppose that these experiments are tosses of a fair coin and there is a gain of $+1$ for each head loss -1 on each tail. The outcome of a flip of the coin is equally likely to be heads or tails, so the walk is clearly unbiased, in that there is no preference for gain or loss. Then S_{n} represents his cumulative gain or loss on the first n plays. Then the sequence $(S_{\mathrm{n}})_{n=0}^{\infty}$ would be described as a simple random walk (for this scenario its called the *gamblerś ruin estimate*) on the $\mathbb{Z}$ lattice. When the

behaviour of such a simple process is analysed in detail, it becomes apparent that it more intriguing than it appears to be. What we can deduce is that:

1. The walk returns to its starting position (i.e. 0) at time $2n$ with probability

$$u_{2n} := P(S_{2n} = 0) = \frac{1}{2^{2n}} \binom{2n}{n}.$$

   By *Stirling's approximation*, it is

$$u_{2n} \sim \frac{1}{\sqrt{\pi n}} \qquad (n \to \infty)$$

   [3]

2. The walk eventually reaches each integer within $\mathbb{Z}$ lattice with certainty and so traverses it infinitely often. Suppose $S_0 := 0$ and $T := \inf\{n : S_n = +1\}$ then $P(T < \infty) = 1$ and $P(T = \infty) = 0$

3. The distribution of T is

$$P(T = 2n - 1) \;=\; (-1)_{n-1}\tbinom{\frac{1}{2}}{n}$$

A Simple Random Walk generally refers to random walk in the first dimension. However, this research is concerned with random walks in 2 dimensions. But interestingly, a 2 dimensional random walk is essentially composed of two Simple Random Walks in either plane as the vertices in either plane selected are independently selected.

3.2 Loop Erased Random Walks (LERW)

A Loop erased random walk on the integer lattice is a process obtained from Random Walk by erasing the loops chronologically. In LERW, the process is expected to have a continuum limit that is conformal, and nonrigorous conformal, field theory gives an exact prediction of the exponent as a simple rational number. In three dimensions there is no reason to believe that the exponent takes on a rational value. Once the loops are erased from the random walk, it essentially becomes a uniform spanning tree (UST), which is a spanning tree chosen uniformly at random, and so the algorithms for generating LERWs are used in application to produce uniform spanning trees.

Loop Erased Random Walk definition. If $\omega = [\omega_0, \ldots, \omega_n]$ is a path, then the *(chronological) loop-erasure* function $LE(\omega)$ would be defined as follows[4]
Let $\lambda_0 = \max\{j \ \leq \ n : \omega_j = 0\}$.
Set $\beta_0 = \omega_0 = \omega_{\lambda_0}$.
Suppose $\lambda_i < n$. Let $\lambda_{i+1} = \max\{j \ \leq n : \omega_j = \omega_{\lambda_i+1}\}$.
Set $\beta_{i+1} = \omega_{\lambda_{i+1}} = \omega_{\lambda_i+1}$.
If $i_\omega = \min\{i : \lambda_i = n\} = \min\{i : \beta_i = \omega_n\}$,
then $LE(\omega) = [\beta_0, \ldots, \beta_i]$

4 Methodology

In this section, I will explain the theory behind the code that was written to produce the resultant data. The code was mainly[2] written in `Python` as it enabled us to stay problem-oriented and allowed us to implement the LERW rapidly as well as readily provided us much greater useful functionality through its extensive library. We used the Python `matplotlib` library to generate the required statistical plots from the data as it integrated seamlessly with our workflow. We also used the `numpy` library for efficiently creating and manipulating the large matrices for the system instances.

Code distribution was a major issue as sharing code via an email did not only make it hard to keep track of it but also aroused great inconsistencies and miss communication. So, I proposed the usage of the version control systems called *git*. The usage of this system enabled us to stay in sync with the latest code version of the project, resolve any conflicts in the code and update the code. So our workflow consisted simply of fetching the latest code and pushing the changes made [3], each with a single command [4].

Overall, these changes made the tasks easier to perform and made the workflow more efficient saving us an enormous amount of time.

4.1 Random Walk Implementation

The Random Walk was theoretically modelled to transverse on the 2 dimensional surface of a 3 dimensional cylinder (i.e. the system) with two vertical open boundaries and to two horizontal periodic. At the open boundaries, the Random Walk is initiated from starting boundary and ceased at the ending boundary. Due to the curvilinear geometry of a cylinder, the system had the property of allowing the random walk to transverse an infinite number of steps in any vertical direction given that the random walk did not reach the ending boundary. The system was implemented as a 2 dimensional $\mathbb{Z}$ lattice of a particular size (length by circumference) which represented the 2 dimensional

[2] The *raw2binned.c* script in `C++` was used for the sake of efficiency and was originally written by Francesc, later modified by Gunnar Pruessner and then adapted by me for this project

[3] The changes made were committed and pushed to *GitHub* (a centralised project hosting platform that uses the *git* version control systems) on which the project was hosted.

[4] In the shell/terminal/command-line

surface of the cylinder and, to implement the previously stated property of the original system, periodic boundaries where inaugurated. The function of the periodic boundaries was ensure that the random walk continues to transverse from the boundary iteratively until it took a valid step (i.e. a step within the system boundaries) when the random walk surpassed a system boundary (see Figure 1). The following is a formal definition of the system.

System definition. Given that $\mathbf{S} = \langle i, j \rangle$ is the system of size (L, C), where i and j are the respective vertical and horizontal components of the system (refer to Appendix A.2 for the actual system definition)

1. The system generation can be expressed as

$$\mathbf{S} = \left\{ (\sum_{k=0}^{L} i_k, \sum_{k=0}^{C} j_k) \;\middle|\; i_k j_k \in \mathbb{Z} \right\} 1$$

2. The length and the circumference (L, C) of the system are of equal size which implies that the lattice was of equal length (for this research)

3. $\mathbf{S}_{\mathrm{i}\,=\,0}$ is the starting boundary and $\mathbf{S}_{\mathrm{i}\,=\,\mathrm{L}}$, where L represents the length of the system, is the ending boundary.

4. $(\mathbf{S}_{\mathrm{i}\,=\,0}, \mathbf{S}_{\mathrm{j}\,=\,\mathrm{y}})$ is the starting point, where y is the first pseudorandom number in the sequence generated by lib function using a given seed.

5. Periodic boundaries exist at $(\mathbf{S}_{\mathrm{j}\,=\,0}, \mathbf{S}_{\mathrm{j}\,=\,\mathrm{C}})$, where C is the circumference of the system, to allow j to be of an indefinite size

Random Walk generation. A Random Walk, S_n, was generated using the generation function

$$\omega_n = \sum_{k=0}^{n} X_k$$

where $\{X_k\}$ are independent and identically distributed random variables. Implementing the periodic boundaries as well, resulted in the generation function

$$\omega_n = \left\{ \sum_{k=0}^{n} X_k \;\middle|\; X_k \neg\, (0 \vee L \vee C) \right\}$$

I translated this into `Python` code which is viewable in Appendix A.3. In the code, the *trajectory* list[5] containing the running set of vertices corresponding to the pseudorandom steps that the random walk had talking was a `numpy` list because of extensive functionality (i.e functions/methods) that were required to perform the non trivial manipulations quickly as well as staying problem-oriented.

When Implementing the algorithm, I had to duplicate the trajectory list and perform some operation to it. However, the problem was that by default python only performed a so called *shallow copy* of the list which is creates a new list

[5] Collectively, the *path* taken (by the random walk)

that only references the data that the old list links to [1]. Hence, when the new list is changed the old list is changed correspondingly as it references the same data. To resolve this issue, I made a *deep copy* of the list which instead of creating a duplicate list that just references the old memory location it actually duplicates the memory location (by recursively copying the properties/methods) and creates a duplicate list with a reference to. This makes any modification to the new list independent from the old list [1].

While originally planned to only use `matplotlib` for plotting the analytical graphs (i.e histograms), I figured out a way to use the `matplotlib` API to plot of the actual trajectory that a random walk makes. This offered us a qualitatively insight into the structure and behaviour of a random walk with a particular system. Refer to the following figure (Figure 1) to see a sample plot. Refer to Appendix A.5 for the plot code that uses the matplotlib API.

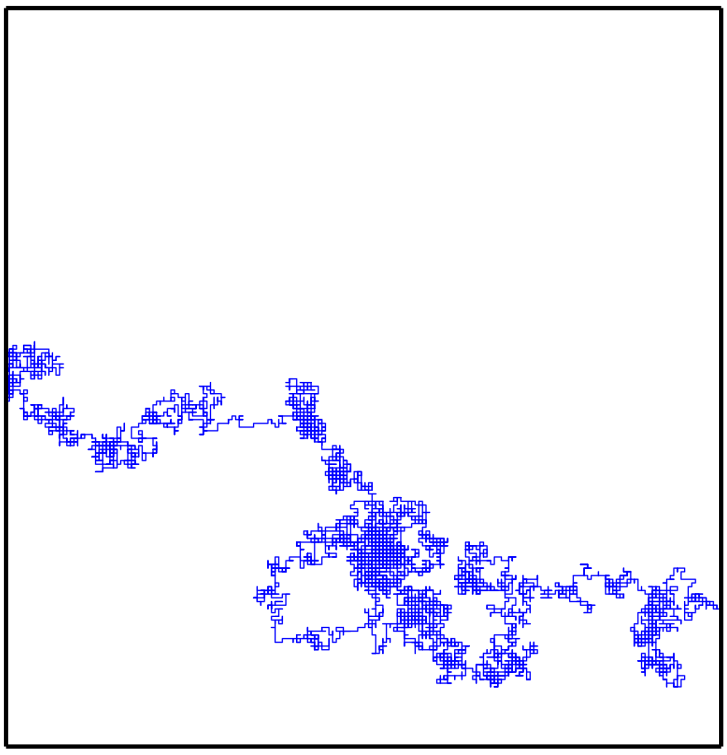

Figure 1: Random walk on a $\mathbb{Z}^2$ lattice (system size: 200; seed: 7)

Random Walk Loop Erasure. A loop is simply the result of the random walk intersecting itself. Considering that the random walk is transversing on an

integer lattice, for a loop to occur, a vertex on the lattice has to be transversed at least twice - at the start of the loop and at the end of the loop. So to erase the loop, an algorithm was devised which removed every duplicate of a given list of vertices (i.e. the random walk trajectory) from the first occurrence of the vertex onwards to (and including) the last occurrence in the random walk trajectory list iteratively until all duplicates (i.e. loops) are removed. The $LE(\omega)$ algorithm defined in Loop Erased Random Walk definition in § 3.2, is a condensed form of the algorithm that was used to generate the LERWs to generate the data. When implementing the LERW and testing, the qualitatively insight gained from plotting the actual LERW trajectory using `matplotlib` was instrumental in evaluating the effectiveness of the code and identifying any logic errors[6]. While the `matplotlib` API sufficed for the random walk implementation, it did not achieve the expected results for the loop erased random walk because its inability to handle discontinuity that resulted from the erasure. This is evident in Figure 2 for the output from which we can qualitatively concluded that the output of the `matplotlib` API plot code (Appendix A.5) is incorrect. So, I had to write algorithm that accounted for this by discreetly plotting the continuous sequences and then joining them. Figure 3 is the output of the final loop erased random plot code (Appendix A.6). Refer to Appendix A.4 for the implementation of the LERW.

[6] "A logic error (or logical error) is a mistake in a program's source code that results in incorrect or unexpected behaviour" from *techterms.com/definition/logic_error*

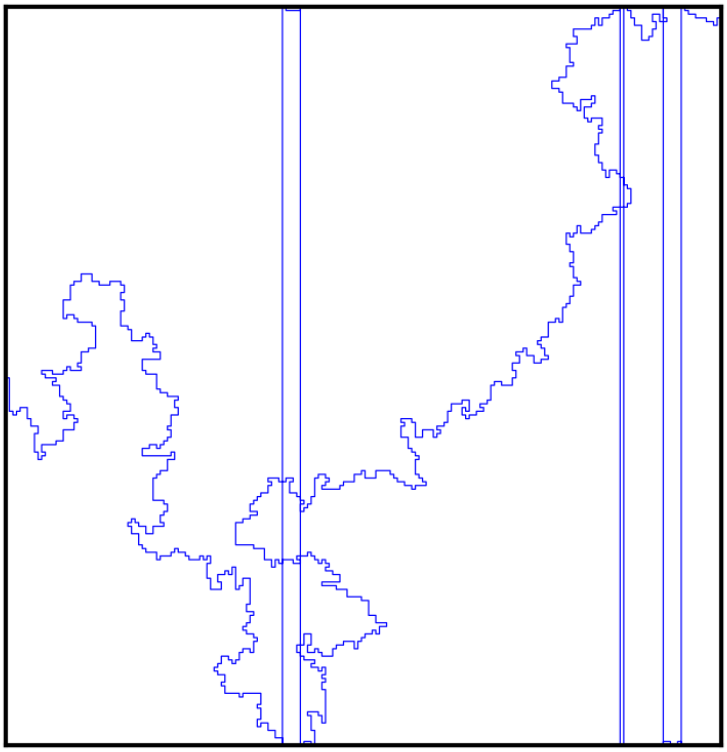

Figure 2: Loop erased random walk plot using `matplotlib` API (system size: 200; seed: 9)

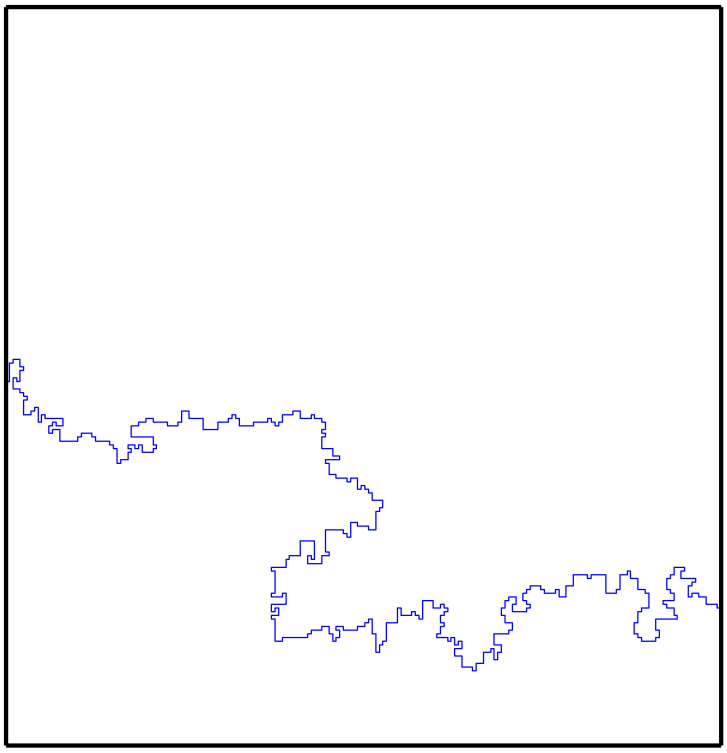

Figure 3: Loop erased random walk on a $\mathbb{Z}^2$ lattice (system size: 200; seed: 7)

4.2 Collecting Data

All data collected for analysis was primary data generated by self-written code and systematic randomness. Data collection was an automatic procedure carried out by the code that I had written when run. A significant amount data had to collected for different system in order to draw a reliable and a more decisive conclusion. As discussed before, the main implication of the presence of periodic boundaries was that the size of the vertical component was indefinite and thus the size of the result was indefinite. Since the code could not be optimised for multithreading, the task for the execution of the code could not utilise more than one core of the multi-core computational infrastructure. Also considering that the computational infrastructure was not dedicated for the task (i.e other OS-related task have had a higher priority), it was apparent that the code had to run for a prolonged period of time in order to collect the necessary. Data was collected for different systems, namely, one dimensional random walk,
Fortunately, the computational resource available at disposal for this project were vast. In particular, High Performance Computing (HPC). The source code

was run on the HPC as a series of jobs over the period of a whole day (24 hours) . Bash code was written to instruct the HPC to output the generated the data by the python interpreter as a series of UTF-8 encoded sequential .dat files (in the format required for analysis). Some metadata (such as the system size) was prepended to the files to distinguish and identify the files easier in analysis stage.
Data was collected for the following systems:

- 2 dimensional Random Walks of system sizes of 50, 100, 200 and 400
- 1 dimensional Random Walks of system sizes of 50, 100, 200 and 400
- 2 dimensional Loop Erased Random Walks of system sizes of 50, 100, 200 and 400

On the first run on the cluster, we received a runtime error due to the incorrect configuration of the cluster but this was quickly fixed once the system was reconfigured. Once the files containing the data were generated, they were inspected manually to ensure that the generated data was valid (i.e. conformed to the expected data) and no logical errors where present in the code. Due to time constraints for each job of 15 hours on the cluster and considering that only one core of the cluster was being utilised, the code for generating the 2 dimensional Random Walks of system sizes of 400 and the code for generating 2 dimensional Loop Erased Random Walks of system sizes of 200, 400 was unable to completely execute. This incomplete execution resulted in the files not being generated (files were not completely written and so any associated temporary was discarded). Due to scarcity of time, I made the decision to just work with the data that we had obtained for the analysis instead of spending more time on generating data and produce less analysis. We agreed that this was the most productive course of action. Hence, *in this report we will only analyse the data by a system*[7] *where* $|\mathbf{S}| \leq 200$.

5 Results and Analysis

After the output .dat files were received, the data such as the return distributions needed to be plot as a histogram. The problem I encountered here was grouping the raw data in suitably sized bins in order to construct the histogram. Since the data was very large (up to 100MB), I we considered using a script written in *C*. However, since this script was in *C* and I was working in *Python* so there was no programmatic way of exchanging the data (variables in one language could not be used in the other). Instead of rewriting this script in *Python* and compromising performance, I figured out a way to exchange the data by modifying the script to output the data in a particular serialised format in *C* and then parsing the output in *Python* (refer to Appendix A.7 for the

[7] refer to § 4.1 for definition

code). Once parsed, the data was available in *Pythonś* memory space and it could proceed to generate the histogram. The initial histogram depicted a very high positive skew as only a few significant values dominated resulting in an ambiguous trend. Discussing this, I found that this was due to great variation not being represented well due to the linear scales and so he advised me to use a double logarithmic scale. The double logarithmic histograms (refer to, Figure 5 - 7) were used to qualitatively compare and evalulate the return times distributions for 2 dimensional discrete ($\mathbb{Z}^2$) LERWs, 2 dimensional discrete ($\mathbb{Z}^2$) Random Walks and 1 dimensional discrete ($\mathbb{Z}$) Random Walks.

What my research found was that was that return times distribution for $\mathbb{Z}^2$ random walk and return times distribution for $\mathbb{Z}$ random walk were very similar. What my research also found that there is a difference in the distribution of the return times of a random walker (1D and 2D) and the distribution of the loops erased in LERW.

Return distribution of 1D and 2D Random Walks. Comparing Figure 6 and Figure 7), I found that the return times distribution for $\mathbb{Z}^2$ random walk and return times distribution for $\mathbb{Z}$ random walk were very similar. For a random walk in one dimension starting at n, the probability that the random walk eventually returns to n equals one [6] $\implies \Pr(S_n = n) = 1$, $n \in \mathbb{Z}$. However, the time required to return to n, averaged over all possible vertices in the system, is infinite. We conclude that this observation is due to the survival probability $S(t)$ ultimately decaying to zero for a random walk in two dimensions. This means that a random walk is recurrent - it is certain to eventually return to its starting point, and indeed visit any vertex of an infinite lattice. This is the 2. deduced characteristic stated in §1.1. This is also because of the fact that a random walk has no memory (random variables independently distributed) and so it transits to a new state (i.e. resets itś state) every time a specific lattice vertex is transversed. Hence, recurrence also implies that every site is visited infinitely often.

The arcsin law, which is concerned with the statistics of returns to the origin, also applies here as our random walk is an unrestricted random walk. From the arcsine law, we can infer that the most probable outcome is that the walk always remains entirely on the positive or on the negative axis. This is against the natural expectation of approximately one-half of the total time being spent on the positive axis and the remaining one-half of the time on the negative axis for a random walk which starts at x = 0. Surprisingly, the natural expectation is the least probable outcome.

Difference in arc length distribution of returns and loops erased. As depicted in Figure 4, there is a correlation between the distribution of the loops erased in a random walk and the probability of return to (an arbitrary) vertex in the lattice for both systems of size 100 and 200. In fact, the combined

distribution is positively skewed which implies that as the loops erased increase, the probability of returning to a vertex in the lattice decreases. The fact that the data is skewed, indicates that there is a difference arc length distribution of a random walker returning and arc length distribution of the loops erased.

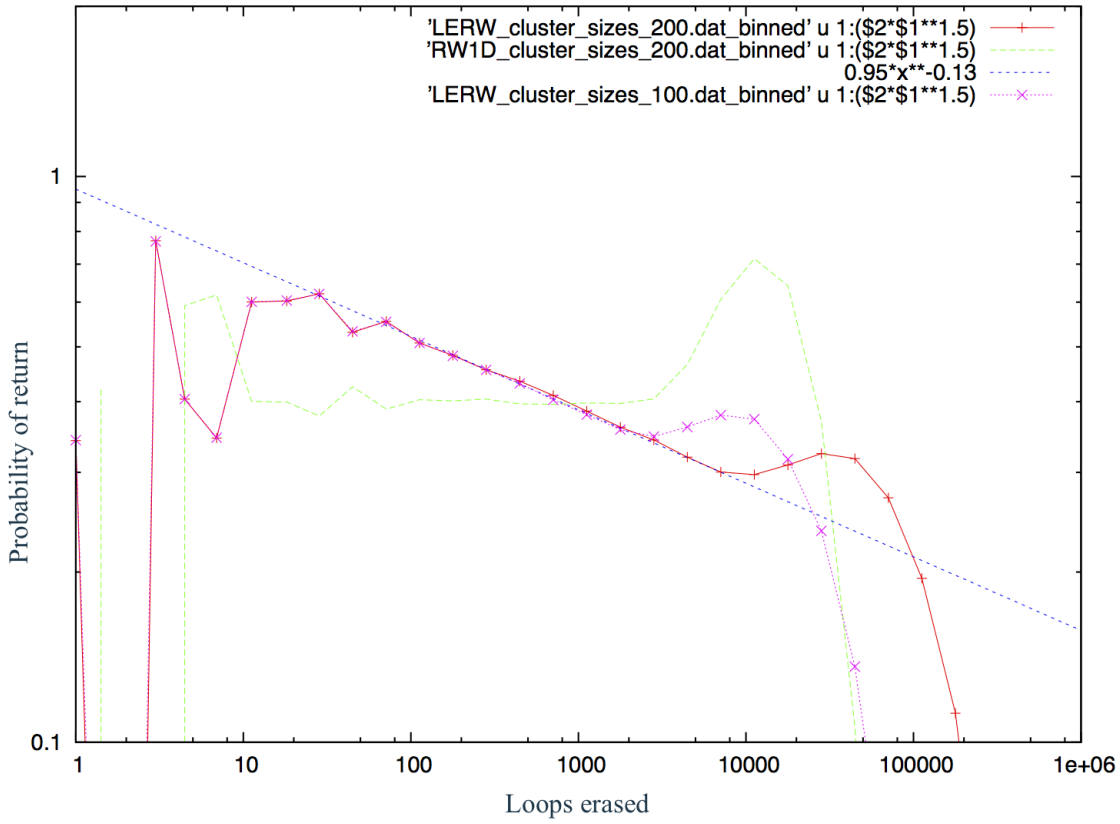

Figure 4: Probability of return and loops erased distributions for $\mathbb{Z}^2$ LERWs (system sizes: 100, 200; seed:10) and $\mathbb{Z}$ Random Walk (system size:200; seed:10)

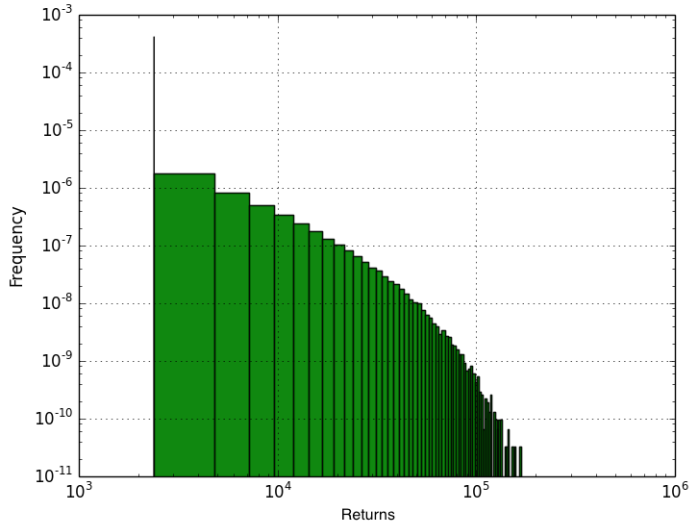

Figure 5: Histogram for the distribution of the Loop Erased for Random Walks on $\mathbb{Z}^2$ lattice (system size:200; seed:10)

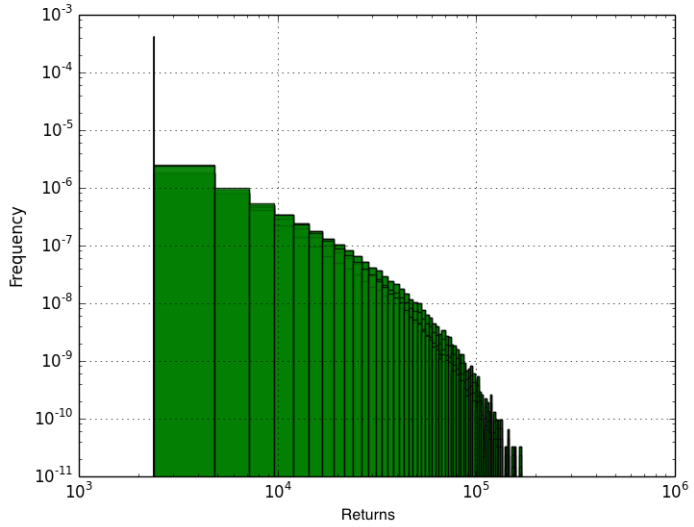

Figure 6: Histogram for the returns distribution of Random Walks on $\mathbb{Z}$ lattice (system size:200; seed:10)

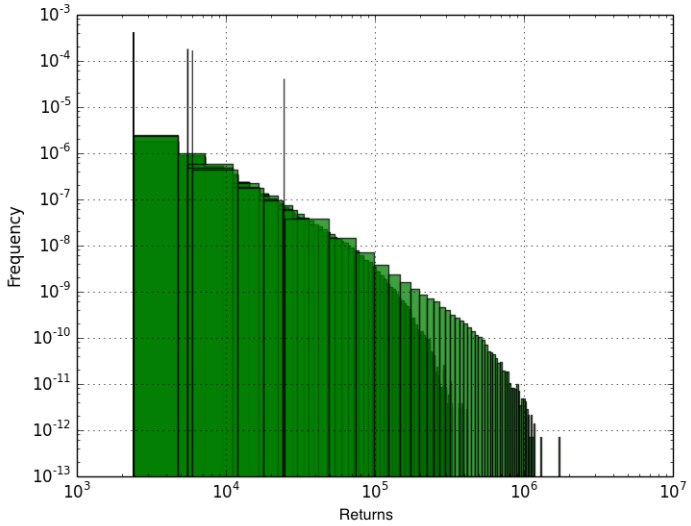

Figure 7: Histogram for the returns distribution of Random Walks on $\mathbb{Z}^2$ lattice (system size; seed:10)

6 Conclusion

6.1 Findings

The research has shown that the return distribution of a one-dimensional random walk and a two-dimensional random walk is almost identical. It can be concluded that the similarity in the return distribution of the one-dimensional random walk and the two-dimensional random walk is due to the survival probability of the two-dimensional random walk eventually being zero. Since the return probability of a one-dimensional random walk is one, this means that return probability of a two-dimensional random walk is very close to one. We can also conclude that there is a difference in the distribution of the return times of a random walker and the distribution of the loops erased in LERW.

To further develop this research, I would incorporate the usage of Self Avoiding Random Walks (SARW) which are random walks that do not self-intersect during their transversal and are in a different universality class from LERW. I would also consider the relationships in the statistics of enclosed area between bidirectional LERWs, the statistics of the intersections and the variance of the number of intersections. A more abstract area that could be explored is the topography of LERWs, in particular, the average eccentricity of the loop erased random walk trajectory.

6.2 Implications

I believe that the results that I have obtained could have an impact on many stochastic processes and on the people whose careers are associated with them. The return probability of a two-dimensional and a one-dimensional random walk being almost identical implies that a stochastic process modelled using two essential (independent) random variables will inevitable return to its current position. What this means in the real world, for instance in stock market is that the price of a stock on the stock market with respect to time (where the stock price and time are the two random variables) will eventually return to its current stock price (over an indefinite time period). In this application case, this discovery may be beneficial to shareholders as it allows them to anticipate a future market position and make actionable decision with that information. The cumulative effect of this could dictate the the future of the market both short-term and long-term. Similarly, when applied in a network, this discovery asserts that the current bandwidth usage in network will eventually be reached again over an indefinite time period. This could prompt network administrators to anticipate peak bandwidth usage by deducing the return to peak bandwidth usage time and allocate the resources across the network more efficiently.

A Appendices

Please note that any variables/constants declared are accessible throughout (globally) the (8.x) code sections respectively of their scope. Also note that this is not all the code produced for the research project but only parts that this report explores. Appendix A.1 declares the libraries used globally and their respective identifiers used to reference them in the subsequent code.

A.1 Imports

```
# Import statements
import random
import numpy as np
import matplotlib.pyplot as plt
import subprocess
```

A.2 System Definition code

```
# System definition & initialization
seed = 10  # random seed
Length = 200  # length of the cyclinder
Circ = 200  # circumference of cyclinder
x = 0   # x coordinate of starting location
# y coordinate of starting location. Origin is at centre of square
y = Circ / 2
s = 0  # Step number.
```

```
trajectory = []    # List of the x coordinates of all points visited.
# (Length x Circ) 2D array of zeros
lattice = np.zeros((Length, Circ), dtype=int)
random.seed(seed)
```

A.3 Random Walk generation code

```
# Random walk generation
while True:
 s += 1
 if (bool(random.getrandbits(1))):
   if (bool(random.getrandbits(1))):
     x += 1
   else:
     x -= 1
 else:
   if (bool(random.getrandbits(1))):
     y += 1
   else:
     y -= 1

 if (x >= Length):
   break
 elif (x < 0):
   x = 0
 if (y >= Circ):
   y -= Circ
 elif (y < 0):
   y += Circ
 lattice[x][y] += 1
 trajectory.append((x, y))
```

A.4 Loop Erased Random Walk code

```
# Random walk loop erasure
x0, y0, pos = None, None, 0
while pos < len(trajectory):
        x, y = trajectory[pos]
        if lattice[x][y] > 1 and (not x0):
                x0, y0 = x, y
                pos0 = pos
        elif (x == x0) and (y == y0) and (lattice[x][y] == 1):
                del trajectory[pos0:pos+1] # erase loop
                x0, y0 = None, None
                pos = pos0
        lattice[x][y] -= 1
        pos += 1
```

A.5 Random Walk plot code (using the matplotlib API)

```
# Plot random walk
dpi = 300 # plot resolution
fig, ax = plt.subplots()
fig.set_size_inches(3, Circ * 3. / Length)
ax.set_xlim(0, Length - 1) # set x axis scale
ax.set_ylim(0, Circ - 1) # set x axis scale
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.plot(*zip(*trajectory), linewidth=0.3)
fig_name = __file__[:-3] # plot name
plt.savefig(fig_name+".png", bbox_inches="tight", dpi=dpi)
```

A.6 Loop Erased Random Walk plot code

```
# LERW plot
fig, ax = plt.subplots()
fig.set_size_inches(3, self.Circ * 3. / self.Length)
ax.set_xlim(0, self.Length - 1)
ax.set_ylim(0, self.Circ - 1)
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

Plots = []
prevInd = 0

# generate an (discrete) array of continuous sequences
for pos in range(len(LERW)):
        x, y = LERW[pos]
        if (x == Length) or (x == 0) or (y == Circ) or (y == Circ) or (y == 0):
                Plots.append(LERW[prevInd:pos])
                prevInd = pos
        pos += 1

# plot each sequences and connect them to form a continuous sequences
for plot in Plots:
        plt.plot(*zip(*plot), color=c, linewidth=0.2)

plt.savefig(figname, bbox_inches='tight', dpi=300)
```

A.7 Histogram plot (and parse from raw2bin.c) code

```
# Groups the raw data into bins using raw2bin.c
# Plots a double logarithmic histogram using the bins
```

```
dpath = "data/qsub/" # .dat files dir
spath = "plots/hist/dbins/" # histogram plot output dir
p = subprocess.Popen(['ls', '-v', dpath], stdout=subprocess.PIPE)
fnames = p.communicate()[0].split('\n')[:-1]
data = []

# Parse & process the output from raw2bin.c
for fname in fnames:
        for line in open(dpath+fname):
                if (line.strip()).isdigit():
                        data.append(int(line.strip()))

# Plot the return distribution double log histogram
x = np.array(data)
n, bins, patches = plt.hist(x, 70, normed=1, facecolor='green', alpha=0.75)

plt.yscale('log', nonposy='clip')
plt.xscale('log', nonposy='clip')
plt.xlabel('Cluster size (sys size: {})'.format(fname[-6:-4]))
plt.ylabel('Frequency')
plt.title(r'$\mathrm{'+fname[:-4].replace('_', '\ ')+':}\ $')
plt.grid(True)

plt.savefig(spath+fname[:-4]+"_histogram.png", bbox_inches="tight")
```

B Bibliography

References

[1] Docs.python.org, *8.17. copy shallow and deep copy operations python 2.7.11 documentation*, 2016

[2] Aaron Doll, *Applications of random walks in tor*.

[3] Rick Durrett, *Probability: Theory and Examples Rick Durrett January 29, 2010*, vol. 49, 2010.

[4] J. Klafter and I.M. Sokolov, *First steps in random walks: From tools to applications*, OUP Oxford, 2011.

[5] G.F. Lawler and V. Limic, *Random walk: A modern introduction*, Cambridge Studies in Advanced Mathematics, Cambridge University Press, 2010.

[6] Sidney Redner and J. R. Dorfman, *A Guide to First-Passage Processes*, vol. 70, 2002.

[7] Jason Schweinsberg, *The loop-erased random walk and the uniform spanning tree on the four-dimensional discrete torus*, Probability Theory and Related Fields **144** (2009), no. 3-4, 319–370.

C Acknowledgements

I would like to thank Professor Gunnar Pruessner of Imperial College London for supervising the research project and the Nuffield Foundation for funding this research. I would also like to thank Imperial College London for allowing us to use the High Performance Computing (HPC) mainframes at Huxley Building in ICL.

<> *with* <3 *by* Habib Rehman