Simulation of random lattice walk of a particle in one and two dimensions.

[153071]
PHY2027, University of Exeter, U.K.

1 introduction

The concept of randomness has deep historical roots, with early discussions found in ancient Greek philosophy with Aristotle pondering the role of chance in natural events. The Renaissance saw renewed interest in probability when mathematicians like Cardano first formalised a theory of randomness[3] to aid his gambling, as well as Fermat, and Pascal, blazing the way towards a probability theory in the 17th century. But the study of randomness exploded in the 20th century with the advent of Quantum Mechanics, which stripped the deterministic laws of Newton away as it was accepted that chance ruled our lives at the very smallest of scales.

A random walk is a powerful mathematical tool that models the motion of a particle or body. They have many useful applications in physics and mathematics, such as describing the random motion of a particle through a lattice. In this case at each step the particle jumps to another neighboring site according to some given probability distribution, this forms a lattice path.

This project aimed to created a random lattice walk in 1 and 2 Dimensions, the results would then be examined upon changing the given probabilities of movement and the number of steps. This was accomplished using the C programming language and visualised in Python. The results were then analysed to see how on average the initial displacement of the particle varies as the number of steps changes. In the literature it is shown that a random lattice walk in 1 or 2 Dimensions should return to its starting position, this phenomena is known as the drunkards walk, a full mathematical proof of this can be found[1].

2 Construction

2.1 1-Dimensional solution

Within the code the concept of divide and conquer was used in order to generate a solution. The first problem that was identified was generating probabilities given by some constants from the diffusion equation.

$$P = D\frac{\Delta T}{(\Delta x)^2} \tag{1}$$

In order to get my code to run as predictably as possible I set all parameters to be as simple as possible, I label;ed these as the **initial conditions**: the **time** and **space** steps to be 1; a **diffusion constant** as 0.4 and **initial position** of 0. This resulted in probabilities to move = 0.4 a piece, and remaining in one place as 0.2. These constants were placed in a **structure, constants** outside of **main**. This improved the readability of the code and made it easier to change these in the future as well as increasing the memory optimisation.

This given probability-based relationship was placed into the function **random walk** which took these constants as parameters to the function. In addition I also defined a number of steps that would be the number of iterations that the walk would perform, this was first set to 100. With this I established a hierarchy of functions, with the **main** calling **random walk**. This then iterated through the number of steps defined, and each iteration called **random step** which was passed the probabilities.

The **random step** function ensures that the sum of passed probabilities equals 1, even though previous functions have already established this. This redundancy aids in easier unit testing and enhances stability for future expansions. Within this function, a double-precision random value is generated using **rand()** seeded with time. This ensures the division is performed with floating-point precision, vital for accurately simulating various possibilities, especially when dealing with small probability values.

2.2 2-Dimensional Solution

In order to make the solution suitable to 2Dimensions I opted to use the same **random walk** and **random step** functions and expand their capabilities. I first opted for defining a number of other variables relating to the y dimension (Different probabilities, position histories and directions). However, this severely inflated the size of the random walk function which became cumbersome and inefficient. To stop this I introduced a **structure walker**. This structure contained values for position, probability to move and probability to remain. Instances X and Y were created; the initial position set to be 0; the probability to move made equal to the initial conditions, informed by the diffusion equation 1. This drastically increased the readability of the code and increased cache optimisation.

All values were reset to the standard form so step sizes of 1 were generated. The 1-Dimensional array **position history** was updated to be A 2-Dimensional array which held the x and y coordinates of every step. **Random step** was now called for both x and y for each step. The direction for each step was then passed to position history. These values were printed to screen for every iteration.

I implemented a condition in main that delta t and diffusion constant must be greater than 0 as in both cases the systems would not make sense as the systems would not progress. This allowed me to remove the ceck in **random step**. I decided not to implement the same with X and Y as it would allow me to target these capabilities for unit testing.

2.3 Plotting

The data in **position history** was then written to an external text file, after checking the file had been opened correctly in write mode, the code then iterated through the whole array and uploaded the positions. A Jupyter notebook was written which opened up the text file and extracted the data. First I plotted a graph that showed the motion of X and Y as 2 line graphs, I then combined the two and showed how the coordinates of a particle changed throughout the iteration 1.

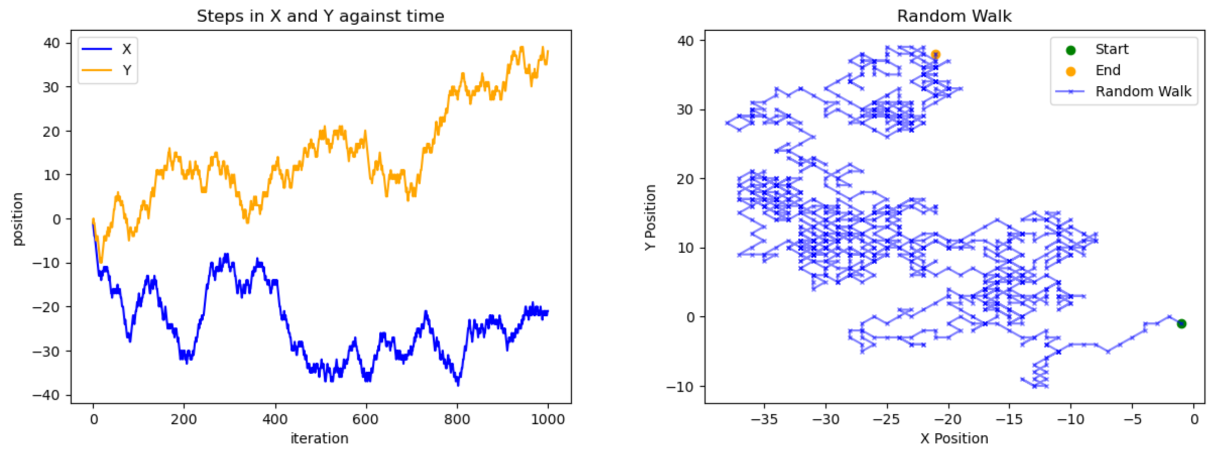

Figure 1: a) Line graph of how X and Y vary independently in position over 1000 steps. b) This plot shows the output of a 2D simulation of a random walk with the initial conditions over 1000 steps.

3 Optimisation and Refinements

3.1 memory requirements

I decided that steps must be an integer as it is an iterator. The time interval however needs to be a float of high precision as this is going to vary based on the system being modeled. X and Y step size are much the same and whilst it would make sense in a discrete model of a lattice to confine them as ints, there are plenty of examples that require a different level of precision, so I decided to use floats, therefore I did the same for position. The diffusion constant is a scalar so it was set to double. All probabilities had to be set to doubles for high accuracy.

Currently in the code I am using a fixed length 2-Dimensional array of length steps, if steps becomes too large this could become an issue as the memory allocated is the same every time. To fix this I opted to use dynamic memory allocation to allocate the correct size given the number of steps. This was done with the use of the **malloc** function. This converts position history into a pointer to an array of integers. After this I then had to ensure that this manual memory allocation was successful, if not then an error statement would be printed to screen. Then the dynamic memory was freed and the pointer to position history was set to NULL.

3.2 Run time optimisation

When breaking the code down to look for run time optimisation it is best to employ a bottom up approach. This philosophy means that to make significant improvements to the run-time of my code I should aim to optimise random step as it is the most called function. I rermoved the check in **random step** and placed it in **random walk**

The **rand** function is a performance bottleneck for my code as it is run for every step it is included by the ¡std.io¿ package in C this is a very old implementation and a more modern function would be far better, which does not require a type cast as well as being faster. An approach that would be counted on to produce randoms for a large simulation length is the Mersenne Twister [5].

Parallelisation would greatly increase the speed of this code as it would allow the random walks in X and Y to occur simultaneously as they are independent of each other, this can be done by including the key word **pragma** before the step loop in random walk. It is also beneficial to use tools such as 'gprof' or 'perf' to identify performance bottlenecks.

3.3 Limitations

There are a few issues that still need to be addressed in the code. Firstly the structure of the code, whilst it is clear and logical it could be enhanced by further subdivision of functions. This would help the maintainability of the code, this is most clear with writing position history to a file. This file would also benefit from being another type of file such as a CSV or JSON for speed and ease of access. Another function could also be beneficial for carrying out all appropriate checks on passed data which would increase the efficiency of all other functions. There is also a danger of the precision that has been used to check for valid probabilities, as they are saved as doubles but in some extreme cases this could cause issues and a safer alternative would be to use **long longs**.

Another key limitation is the error codes that are returned it would be better to denote certain return values in functions to be specific values that inform the user what error has occurred. The code is also very inflexible the checks are hard coded to be the specific conditions of this implementation, the code is going to be hard to adapt to investigate results.

3.4 Future Improvements

An expansion into 3 Dimensions would be easy to implement in the code, but from a mathematical perspective it greatly changes the problem. As has already been evidenced a 2-Dimensional is recurrent, the same is not true of a 3-Dimensional walk as it is transient[6]. This could be useful not only modelling insects and birds but in more fundamental scientific systems: gases and fluid dynamics as well as larger projects such as galaxies spin [4].

Another key improvement could be user input, this would simply allow a user to input the values for the diffusion equation and demonstrate to them how this would change the final displacement. The best way to do this that would require a large amount of work is to convert the code into a library that could be accessed through python, this would require lots of optimisation and debugging to make the code indestructible. This would allow the user to plot directly in python.

Something that would also be interesting to model is the effect of other walkers being present in a system. This could be done in many ways and represent many physical systems, whether it be for gases in a box or a novel examination of disease spreading around a predefined lattice representing some geography[2].

4 Investigation

4.1 Approach

I wanted to investigate how changing the number of steps and diffusion constant would change the program over many runs. For a 1D approach I removed all the print statements and references to the y dimension. I then defined the int num walks, which I set to 100000. This would call **random walk** for each iteration and the final position would be **append** to the array position history. This array was exported to Python and plotted firstly into a histogram22g. After this, I calculated both the final squared position(FSP) and average final absolute position(AFAP). This was originally done for the**initial conditions**, changing the steps from 5-2500 and diffusion constant from 0.01 to 0.5.

To see these results in 2D I reinitialised and analysed the results. Position histories in 1 and 2D were plotted for all varied parameters 2.

Secondly, I wanted to asses what would happen to the walk on average if the walk was weighted more towards one direction than another. To do this I changed the variable to move left and right to be independent in the **struct definition**, then hard coded these in the initialisation of X and Y. I changed **random step** to take in 3 probabilities and changed the if condition to allow for that. However, I decided to remove the probability to remain as it allowed me to maintain the relationship constant and focus on how changing the probability of moving in 1 direction effects the total displacement. Since I wanted to analyse how a difference in weighting effected the final position, I created a loop like in the **multiple walks** implementations. I plotted these results. All are represented in 2

4.2 Analysis

As can be seen from figure 2g the mean value of displacement is 0. This is to be expected due to the symmetry of the random walk, as each individual step has equal probability. Therefore, every step left should be counteracted by a step to the right and, since the same is true over many iterations, the average displacement is 0.

I will be seeing how Final Square Position (FSP) and Average Final Absolute Position (AFAP) vary across different executions of these modified functions. FSP 2 takes all the final values and squares the results. Thereby removing negative elements, the mean of those is then found. AFAP instead finds the average position and looks at the the absolute value before finding the mean. Both of these

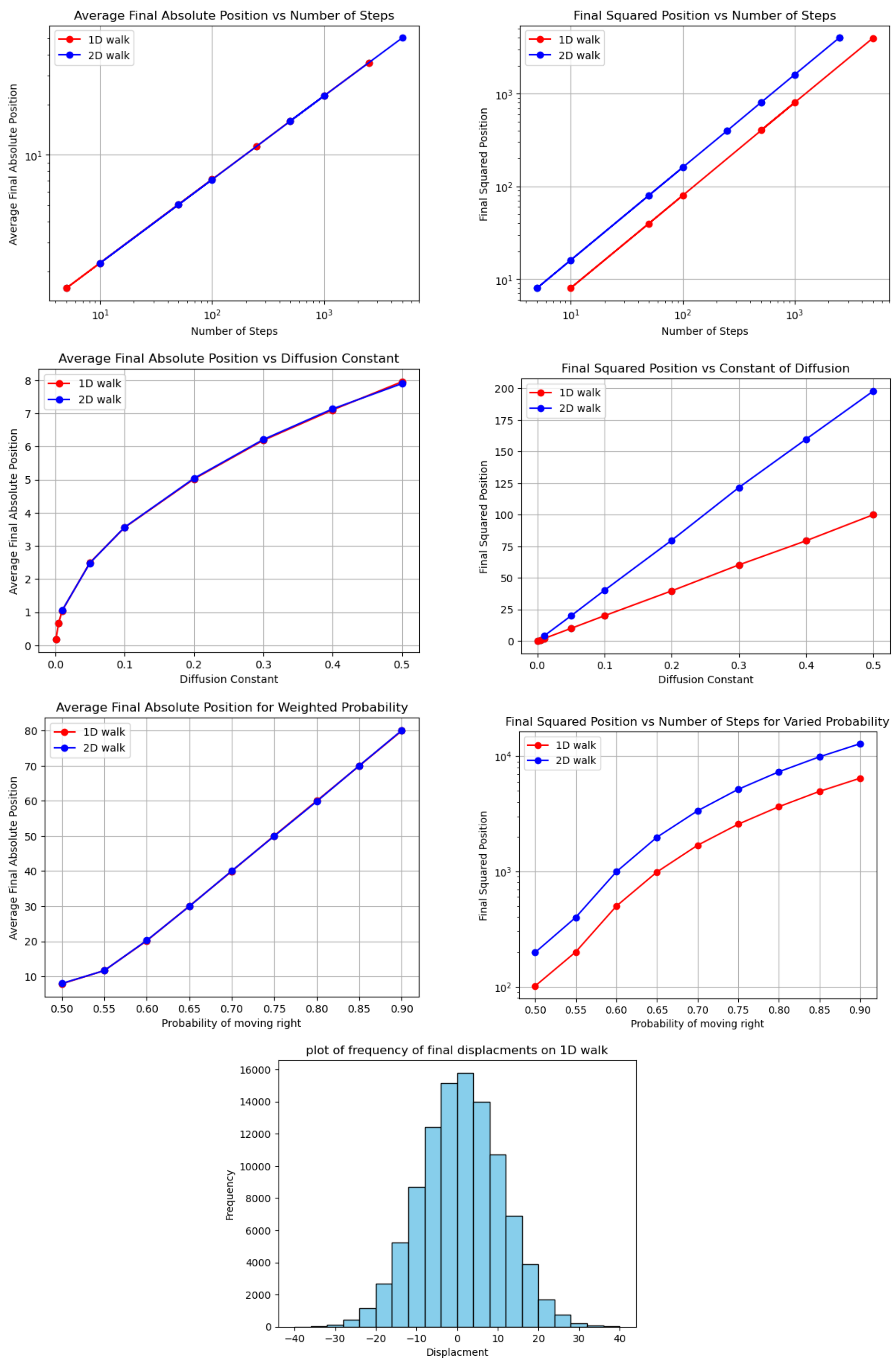

Figure 2: (a,b) graphs showing AFAP, FSP as simulation time changes on a log scale (c,d) AFAP and FSP changes as Diffusion Constant changes at 100 steps and initial condition(e,f) AFAP and FSP for a weighted probability walk with p = 0.5 over 100 steps (g)The resultant plot of average displacement in 1-Dimension from multiple walks when run 250,000 times.

approaches worked by finding the mean of 100000 simulations endpoints. Whilst the mean/mode value returned by the function is 0, these functions will not return 0, as they are a measure of spread.

I varied the total number of steps from 5 to 2500 resulting in figures 2a,b. Observing FSP there was an exponential growth with the increasing number of steps. This is because the sum or average of these steps tends to follow a normal distribution according to the Central Limit Theorem. In this walk, the variance of the final coordinates increases linearly with the number of steps taken, meaning a greater distance from the mean can be reached. I then examined how AFAP varied as run-time increased. This also exhibited exponential growth as the variance grew linearly.

Whilst there are clear differences between the simulations in 1 and 2 dimensions it becomes clear when looking at 2 that the AFAP does not change in 2D. This is because the simulation in both directions is the same. Therefore, it is just averaging the final absolute coordinates. by contrast, the FSP exhibits changes when extended to 2 dimensions. This is because the result is the sum of the two mean squared positions, meaning that the total displacement is higher as the shortest route to the origin from any coordinate is along the hypotenuse. For a simple random walk on a one-dimensional lattice, the expected values for both AFAP and FMSD grow with the square root of the number of steps due to the diffusion-like behavior of the random walk. This is a consequence of the central limit theorem, which states that the sum of a large number of independent, identically distributed random variables tends to follow a normal distribution.

I then varied the diffusion constant from 0.5 (guaranteed movement) to 0.01 (Very unlikely movement) for 100 steps. This is shown in figure 2c,d. In 1d this resulted in a linear in increase in FSP.

AFAP behaves in much the same way across both 1 and 2 dimensional results. This behaviour does not change as the diffusion constant varies, as all the diffusion constant is doing is changing the likelihood of a move being made. When it is lower there will be less movements made, so the probability of staying is less. Fewer moves means a lower distance travelled across the simulation time.

Varying the probability weighting the FSP starts small but increases nearly exponentially. This is because as one side is favoured the average displacement from 0 increases, meaning as the constant increases the particle moves further from the origin on average per run. This is also exhibited in the AFAP graph which shows a near linear increase the more weighted the chances. This is the same in 2D. FSP again exhibits that the 2D is greater because it is summed from 2 terms.

Insum increasing the number of steps in the simulation increase the mean distance from the origin, irrespective of the measure of spread. Adjusting the constant of diffusion to allow for more or less motion also increases the distance from the origin. Weighting the motion in 1 direction or another also increases this spread.

[1] Alin Bostan, Irina Kurkova, and Kilian Raschel. A human proof of gessel's lattice path conjecture. *Transactions of the American Mathematical Society*, 369(2):1365–1393, 2017.

[2] Mohadeseh Feshanjerdi and Abbas Ali Saberi. Universality class of epidemic percolation transitions driven by random walks. *Physical Review E*, 104(6):064125, 2021.

[3] Prakash Gorroochurn. Some laws and problems of classical probability and how cardano anticipated them. *Chance*, 25(4):13–20, 2012.

[4] Masanori Iye, Masafumi Yagi, and Hideya Fukumoto. Spin parity of spiral galaxies. iii. dipole analysis of the distribution of sdss spirals with 3d random walk simulations. *The Astrophysical Journal*, 907(2):123, 2021.

[5] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[6] Elliott W Montroll. Random walks on lattices. In *Proc. Symp. Appl. Math*, volume 16, pages 193–220, 1964.