|  |  |
| --- | --- |
| The Random Walk | Abstract  This report summarises code written using Python to simulate various “Random Walks”. The following paper will show that particles traversing a random walk undergo diffusive motion.  Scerbo, Yahya Robert  PX3017 – Research and Computing Skills |

**Contents Page Number**

Introduction Pg. 2-3

Method Pg. 3-5

Results Pg. 6-7

Conclusion Pg. 7

Extra – The First Passage Problem Pg. 8

Bibliography Pg. 8

**Introduction**

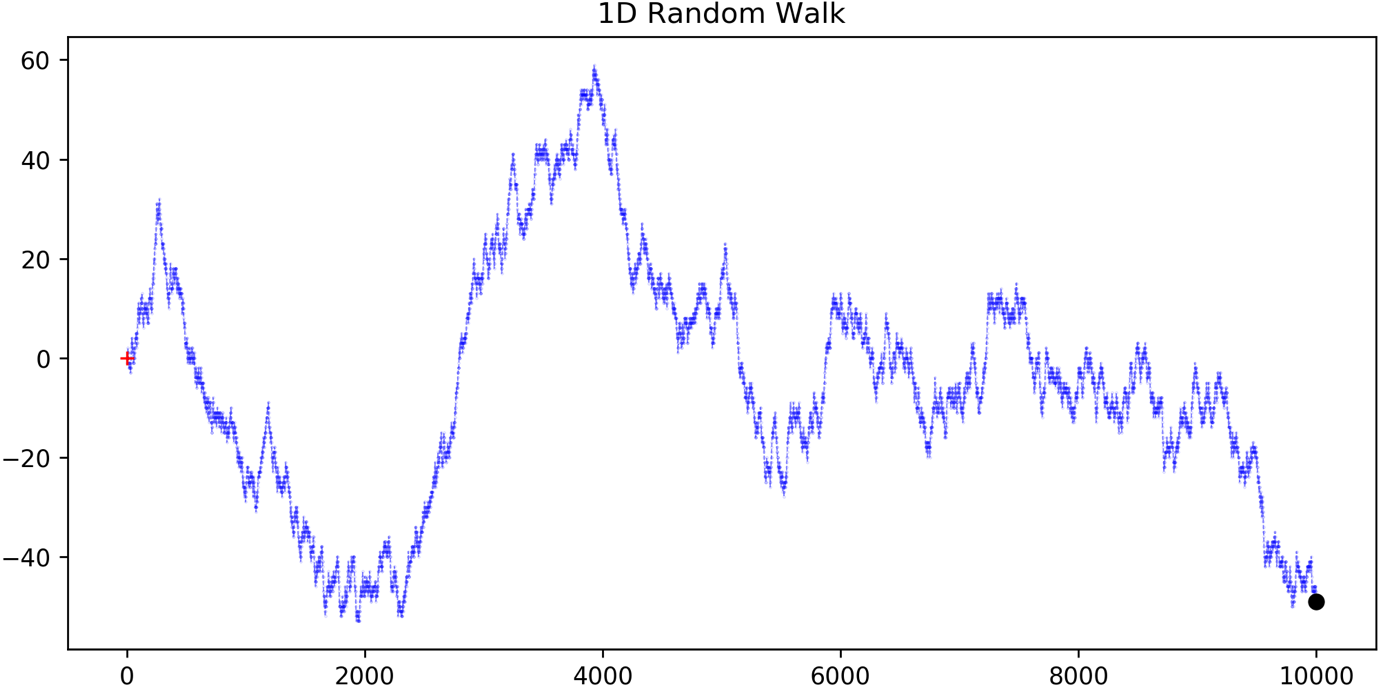
I have chosen to complete the random walk computer programming assignment.

The term “random walk” or “drunkard’s walk” refers to a process by which a trackable object or particle moves through space in a step by step sequence. At each step, the direction of movement of the particle changes randomly.

The particle begins at a starting position (0), (0,0) or (0,0,0) whether the system is in 1,2 or 3 dimensions. Using a coordinate system allows us to track the constantly changing path of a particle, we can plot these paths; they often produce interesting fuzzy images.

The situation being considered in this random walk assignment is a dust particle moving through space. Individual molecules such as these do not move in a straight trajectory, instead they constantly bump into other molecules. These molecules deflecting each other all the time causes the direction of travel of each to be constantly changing. Each particle is therefore performing a random walk, this is the motion by which a gas typically diffuses.

The one-dimensional random walk. A good example of a one-dimensional random walk can be performed on the integer number line, i.e. (…-2,-1,0,1,2…). If we take ‘0’ as the starting position, find a regular coin, assign “heads” to taking a step forward along the number line and “tails” to taking a step backward. At each time step the coin will be flipped and the particle/concentration will move either forward or backward. After one-coin flip; the particle will either be at -1 or 1. The flipping of the coin randomises the direction of the particle after each step. Graphing this system over time delivers a graph similar to a stock price:



The random walks that I simulate and discuss come under the category of two-dimensional random walk. Again, at each time step the direction of travel changes but now the path is tracked in terms of x and y. I will further discuss my own two-dimensional random walks and how they work throughout this report.

Random walks can be applied to various real-world situations: the movements of an animal foraging for food in the wilderness, the path tracked by a molecule as it moves through a liquid or gas (this would be included in diffusion), the price of a stock as it moves up and down, the path of a drunkard wandering through the streets of Aberdeen and even the financial status of a gambler at a roulette wheel. A couple of the examples listed are situations that we intuitively associate with a reasonable process but in fact can be modelled with a stochastic system such as the random walk.

‘Spyder’ is the programming environment used for the random walk simulations in this project, Spyder is written in the Python programming language.

The aim of my project is to simulate the random walk of a particle and to eventually show experimentally that the distance travelled by a particle is proportional to the square root of the time taken, meaning that the particle undergoes diffusive motion.

My final results match quite well with the aim stated above, and throughout the report I will explain how I achieved these results.

**Method**

The very first step before any real code gets written is to import a few important python packages. Importing the ‘NumPy’, ‘math’, ‘PyLab’ and ‘random’ packages is necessary for the code written later on; especially NumPy as it forms much of the underlying numerical foundation that everything in the code will rely on.

The initial position of the particle can be labelled as in (x, y) coordinates and the dynamics of the system can be described by the equation , where is a random angle between 0 and 2π, i.e. creating a completely random direction at each time step.

I create a function called “oneRW(t)”, this function has a single input (t) which represents the number of time steps to be taken during a single random walk. When the function is called it will simulate a single random walk.

At the beginning of my function I assign initial values to:

* (x) – the coordinate on the x axis at a time (t)
* (y) – the coordinate on the y axis at a time (t)
* (r) – the resultant of the x and y coordinates at a time (t), therefore the distance from the origin (0,0) at a given time.
* (dis\_sq) – the distance from the origin at a given time (t) squared

I assign all of the variables listed on the previous page to an array of zeros with length (t).

Then I introduce a ‘for loop’; “for i in range(t):”. (i) will index through the range (t) over each loop. At the beginning of each loop a random angle is calculated, this is the angle that will be used to change the direction of the particle.

The angle is created using a function ‘random.random()’, this function when called will deliver a random number between 0 and 1. Multiplying this function by 2π will give us our random angle.

The next part of the loop will take this random angle and use it to find the next value of x and y:

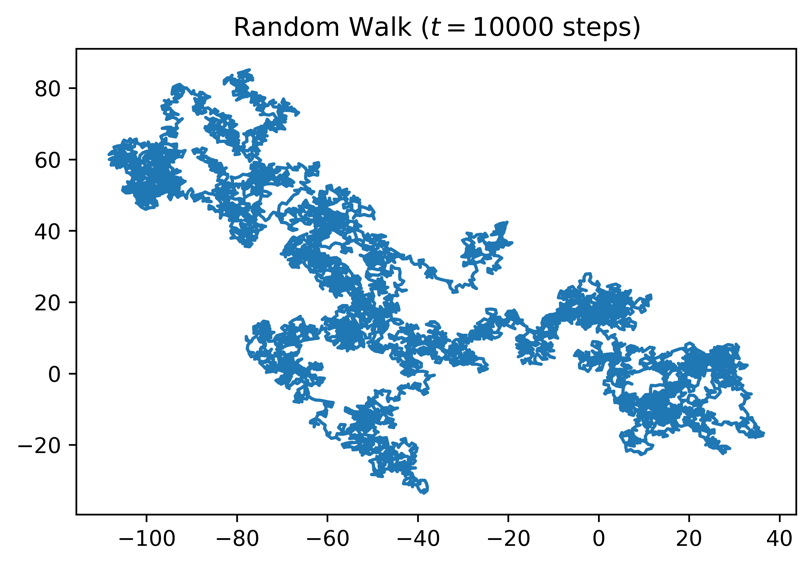
The (x) value is equal to the ( (x) value plus the cosine of the randomly generated angle. The same process occurs for (y) but using sine of the angle instead. To find the new value, the programme will take the previous value and add on the sine/cosine of the randomly generated angle, so as the loop indexes through the range (t); each number in the arrays of x and y are being changed one after the other.

The way in which each (r) value is found is easier to understand. For each new (x) and (y), the following formula is used to find (r) at each time step:

, so now for each random walk there is an array of values for (r).

Finally, with an array of values for (r) we can create an array of values for (dis\_sq). This is done with a very simple formula:

The array of values of square distance from the origin will be used later in the code for a different stage of the project. The function works, and I ran it a few times with plotting code to show the paths of the random walks, the plots create very pretty, interesting looks pictures. An example random walk of 10000 steps is below:



The total number of steps chosen for the random walk previously shown is 10,000. This will deliver us more data later on, which will help us in understanding what is going on once we begin to take averages of these random walks. Ideally a higher number of steps would be even more useful, but this will cause the programme to take a very long time to loop through all of the code.

At the very end, the array of (dis\_sq) is returned in the output of the function. At this point in a new cell I can value to (t) and a new variable (n). This new variable represents the number of random walk simulations that I wish to run.

I create a new variable (allSqDist) and assign to it an all zero matrix of shape (n) rows and (t) columns. Next, I introduce a new ‘for loop’ that takes the (dis\_sq) array delivered from the function and assigns this to the rows of the matrix. (i) will then index the range (n) and create a “list” of random walk results in the matrix. The matrix would now be of the form:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Every column represents a time step (t) | | |
| Each row represents a new random walk from range (n) | … | … | … |
| … | … | … |
| … | … | … |

For each row of the matrix we now wish to find the ‘mean square distance’. Again, we assign a list of all zeros to a new variable (MSD), and using a loop we will change the value of each zero to the (MSD) of each random walk.

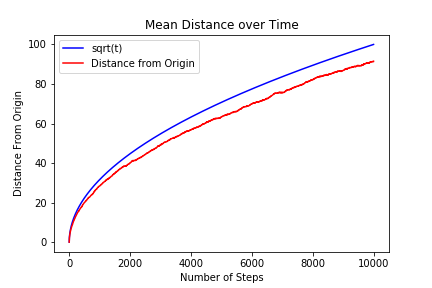
The (MSD) values can then be plotted against the time steps, if the code has worked and our predictions are correct then the graph should show a linear correlation between (MSD) and time. This would mean there is a correlation between Distance and the square root of the time, this is diffusive motion.

I do a final line to print the gradient of the curve in my graph of (MSD) versus time elapsed. The gradient should be equal to the spatial unit chosen earlier in the task. The spatial unit chosen was: 1.

Results

The diffusion of a gas can be thought of as; many particles simultaneously traversing a random walk. With the code I have written, I have simulated the random walks of 1000 different gas particles over 10,000-time steps. If a particle undergoes diffusive motion, then the distance travelled should be proportional to the square-root of the time elapsed.

The graph below shows: the average distance from the origin versus time.

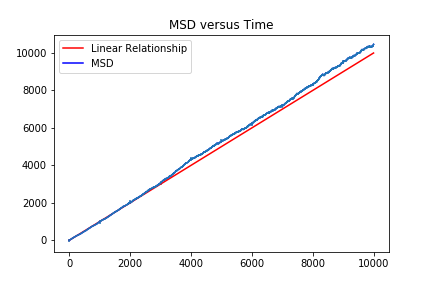


The blue line represents the function . This line has been plotted to make it easier to see how closely the particles undergo diffusive motion.

The red line represents the average distance from the origin of 1000 different random walks versus time elapsed.

The task requests that the mean square distance is found, in the next section of the results I will display my findings of the mean square distance. I chose to plot the mean distance versus the time as well to make the theory easier to understand.

The graph below shows: the mean square distance versus the time elapsed.



If the average distance travelled from the origin is proportional to the square root of the time elapsed; then the average square distance should be proportional to the time.

The red line represents the simple function y = x. This line has been added to better emphasise linear correlation.

The blue line represents the mean square distance of 1000 random walks versus the time elapsed.

Conclusion

My results showed that the MSD is proportional to the square root of the time elapsed.

The gradient of the graph was calculated was 1.05547825. This number is fairly close to 1, so we can say that the results have shown diffusive motion.

Bibliography

King, P. (2019). *What is Random Walk Theory? - Magoosh Statistics Blog*. [online] Magoosh Statistics Blog. Available at: https://magoosh.com/statistics/what-is-random-walk-theory/ [Accessed 23 Nov. 2019].

Slater, J. (2019). *A Brief Introduction to SciPy, NumPy, Matplotlib, and PyLab*. [online] Inside the Ivory Tower. Available at: http://josephcslater.github.io/scipy-numpy-matplotlib-pylab.html [Accessed 23 Nov. 2019].