# Random Walk (Project Report)

**Course:** Introduction to Probability and Random Variables – Spring 2020

**Instructor:**
Dr. Abdul Samad

**Group members:**
Swaleha Saleem (sm05152)
Kuldeep Dileep Lohana (kl04008)

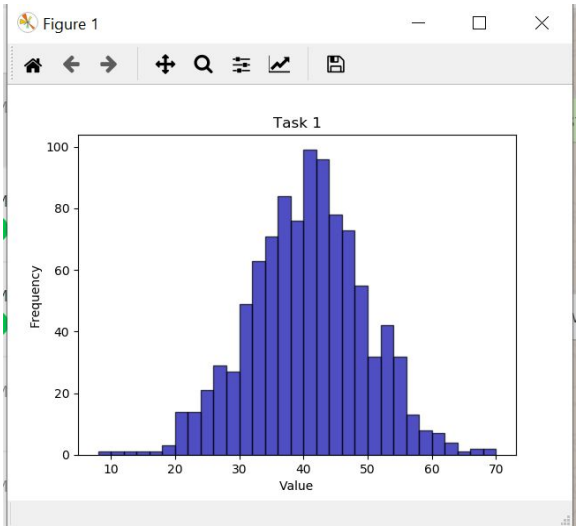
**Task 1:**
In order to simulate a 1-D random walk model, we have created a function named path_finder, that would randomly move the node either towards right or towards left, depending upon the probabilities specified by the user input.

```
def path_finder(steps, prob, path):
    pos = 0
    for i in range(steps):
        # Selects 1 & -1 with the corresponding probabilties specified in
prob list.
        choice = random.choices([1, -1], prob)
        pos = path[-1] + choice[0]
        path.append(pos)
    return path
```

The randomly generated step is then added to the previous/current location of the person and the path list is updated. Until all the steps are taken we receive a list of the path taken by the person in 1D. The rest of the code asks for the user input (probability to move right, no. of steps and starting position) and runs the experiment multiple times to find out the expected value and creates a histogram of the obtained values. The freqn function finds the expected distance from the starting point which has the maximum count in the list and displays height and value associated with it.

**Sample Input:**
Enter the probability to move right (p): 0.7
Enter the number of steps taken: 100
Enter the starting position: 0

**Sample Output:**
Expected distance from starting point:  Rectangle(xy=(40, 0), width=2, height=99, angle=0)




**Task 2:**

In this task, after taking the inputs (probabilities and initial positions of both the nodes), we keep simulating the random walk of both the nodes until they meet and run the experiment for 1000 times. The result is a list containing all the time values taken by the two nodes to meet each other. Then these values are plotted on a histogram to see the expected value which will be the one with maximum count in the list which is obtained using the freqn function defined.

**Necessary Assumption:**

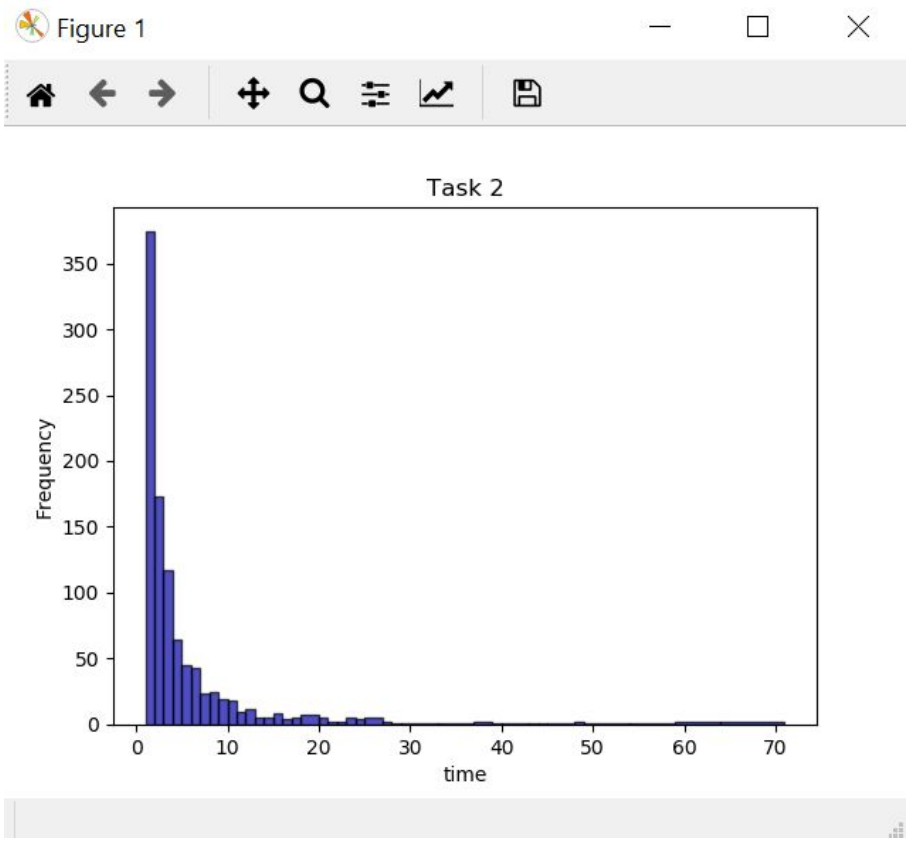
1. We have assumed the value of the time variable to be equivalent to real-time. For instance, if time is equal to 10 then the program is assumed to have run for 10 seconds in reality it would have taken a few ms.
2. In case the two nodes never meet we have set a time constraint that the program shall run for 100 second maximum and if the two nodes don't meet within 100 seconds they are assumed to never meet.

**Sample Input:**

Enter the right probability of person 1: 0.5
Enter the right probability of person 2: 0.7
Enter the starting position of person 1: 0
Enter the starting position of person 2: 2

**Sample output:**

Expected time taken:  Rectangle(xy=(100, 0), width=0, height=572, angle=0)



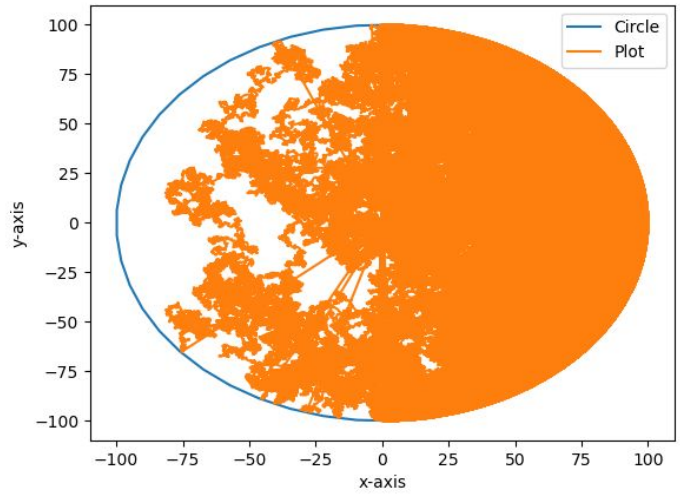

**Task 3:**

In this task we were to simulate 2D random walk within a circular region of radius 100 units. In order to accomplish this we have initially created a circular region of radius 100 units and then we have made sure that the random walk pattern stays within the set boundary. For random walk, at each step a discrete theta and step are chosen from $[0,2\pi]$ and {0,0.5,1} respectively. Using theta and radius(step) we calculated x and y components of the step taken and used it to make sure that the random walk is within the set boundary. Program continuously checks if the point is within the set circular boundary. To do so we have divided the circular region into two regions. If the point is either in 1st or 4th quadrant and outside the boundary the extra 'x' from the circumference to the point will be subtracted and will be reflected over y-axis by subtracting it from the point on circumference to bring it back into the region where the value of 'y' will stay the same. Likewise, if the point is in 2nd or 3rd quadrant and outside the region then we'll subtract the extra 'x' from the circumference and will add it in the point on the circumference keeping the 'y' component the same. And then we append all the points and plot it at the end.

**Sample input:**

Steps = 1000000

**Sample output:**




**Task 4:**

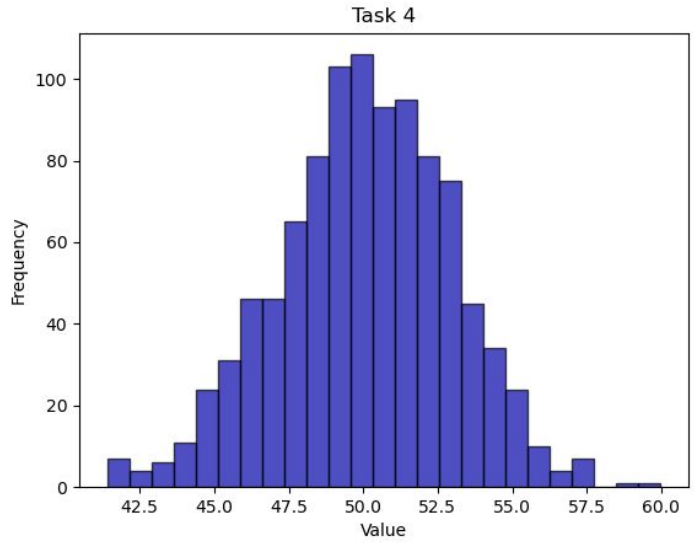
In this task we were to repeat task 1 but this time we have to assume that step size is a continuous uniform random variable between 0 − 1. To achieve this we used python's built-in continuous random variable function (uniform.rvs()) from Scipy library. The remaining implementation is the same as in task 1.

**Sample input:**

Enter the probability to move right (p): 0.1
Enter the number of steps taken: 100
Enter the starting position: 2

**Sample output:**

Expected distance from starting point:  Rectangle(xy=(49.5762, 0), width=0.743532, height=106, angle=0)



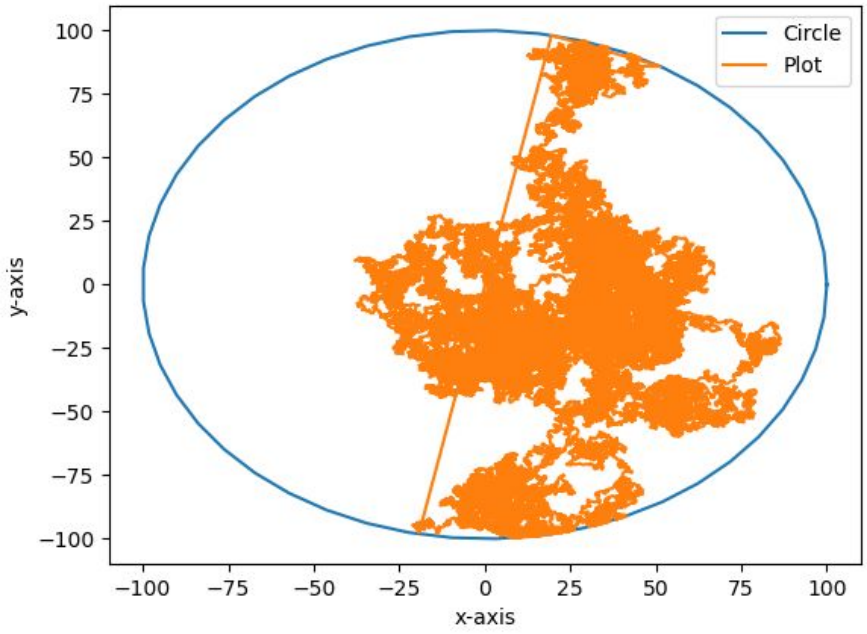
**Task 5:**

In this task we have to repeat task 3 but this time we have to assume that step size and the orientation are both continuous random variables between $0 - 1$ and $0 - 2\pi$. To achieve this we used python's built-in continuous random variable function (uniform.rvs()) from Scipy library. The remaining implementation is the same as in task 3.

**Sample input:**

steps = 200000

**Sample output:**



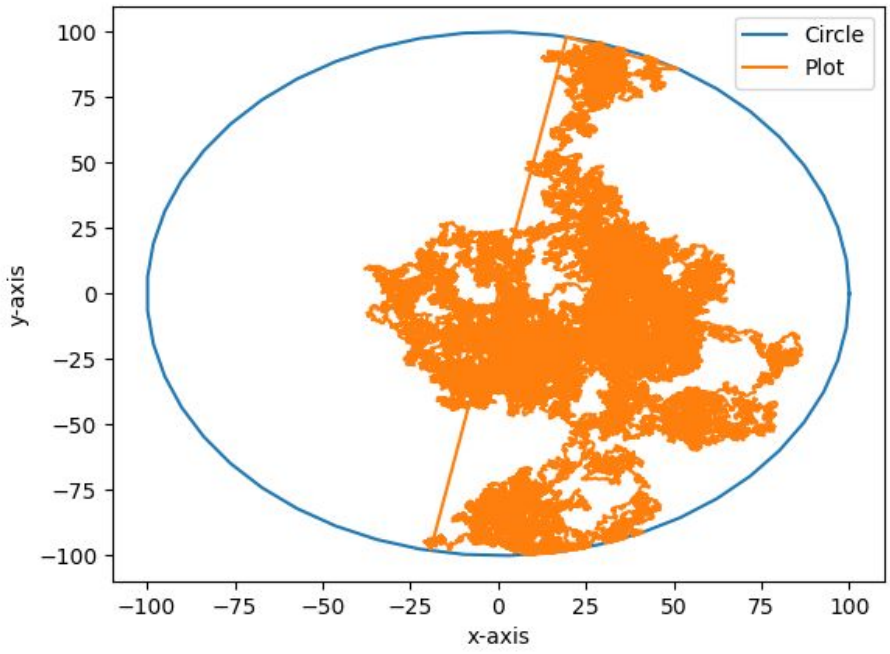

**Task 7:**

In this task we have to repeat task 3 by assuming that the step size is a discrete random variable and the orientation is a continuous random variables between $0 - 2\pi$. To achieve this we used python's built-in continuous random variable function (uniform.rvs()) only on orientation. The remaining implementation is the same as in task 3.

**Sample input:**

steps = 100000

**Sample output:**




**Task 8:**

In order to make the code efficient and scalable to enable the working in task 8, we have modularized the code by making separate classes to simulate Node and their steps. In addition to that, another function has been developed to model the task multiple times in order to reach an expected value.

**Class Node:**

The Node class takes the initial x and y values of the node respectively and these are the only attributes of the class. Amongst the methods of the class, the most important method is the change_path method which randomly changes the position of the node and it is the same method that was employed in Task 5. The rest of the methods are just getters or the helper functions for the change_path method.

**Class Steps:**

The Steps class is the one that actually performs this task. It's only attribute is the number of steps between the two nodes which is initialized to zero.

```
    class Steps:
        def __init__(self):
            self.steps = 0
```

The most important method of this class is the calculate_steps method which keeps changing the path of both the nodes and calculates no. of steps until the distance reduces to less or equal to 1 between them or when the step exceeds 10,000.

```
    def calculate_steps(self, node: Node, node1: Node):
            '''
            calculate no. of steps until the distance reduces to less or
equal to 1 or when the step
            exceed 10,000
            '''

            while True:

                if self.check_distance(node, node1):
                    return self.steps
                else:
                    node.change_path()
                    node1.change_path()
                    if self.steps < 10000:
                        self.steps += 1
                    else:
                        return 10000
```

The other method is just a helper function for the above function which checks the distance between both the nodes and returns True if the distance is less than or equal to 1 and False otherwise.

```
def check_distance(self, node: Node, node1: Node):
        '''
        checks is the distance is less than one or not
        '''

        pos_x, pos_y = node.get_pathx()[-1], node.get_pathy()[-1]
        pos_x1, pos_y1 = node1.get_pathx()[-1], node1.get_pathy()[-1]

        distance = math.sqrt((pos_x-pos_x1)**2 + (pos_y-pos_y1)**2)
        return distance <= 1
```

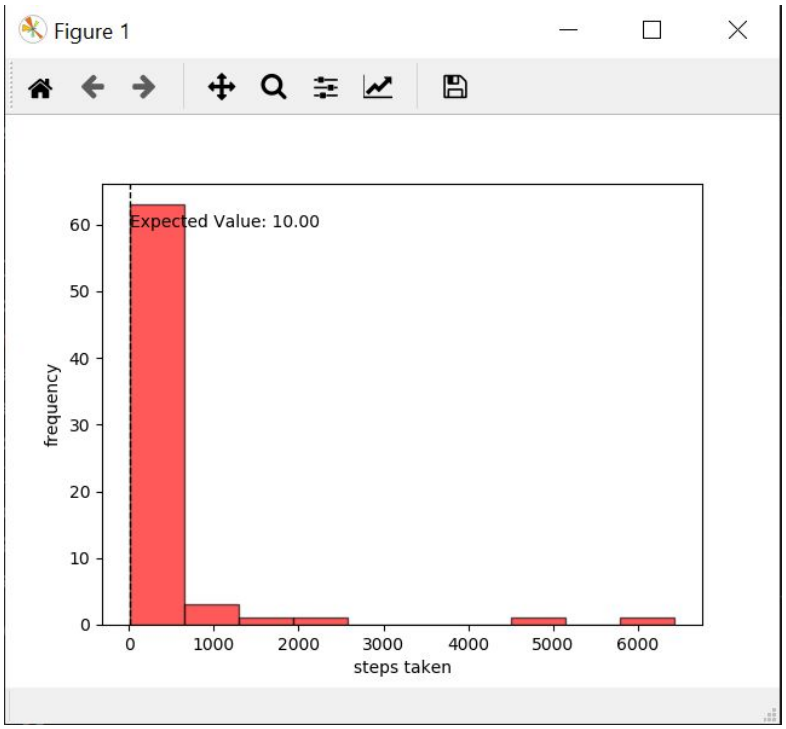
**Calculating Expected Value:**

Finally, in order to calculate the expected value, a global helper function is created which simulates the task between the similar nodes multiple times and saves the values in a list which do not exceed 10,000. Then a histogram is generated to model those values and show the expected value.

**Sample Input:**

Initial value of Node 1 = (0,0)
Initial value of Node 2 = (2,2)
No. of simulations to run = 100

**Sample Output:**




**Task 9 (Bonus):**

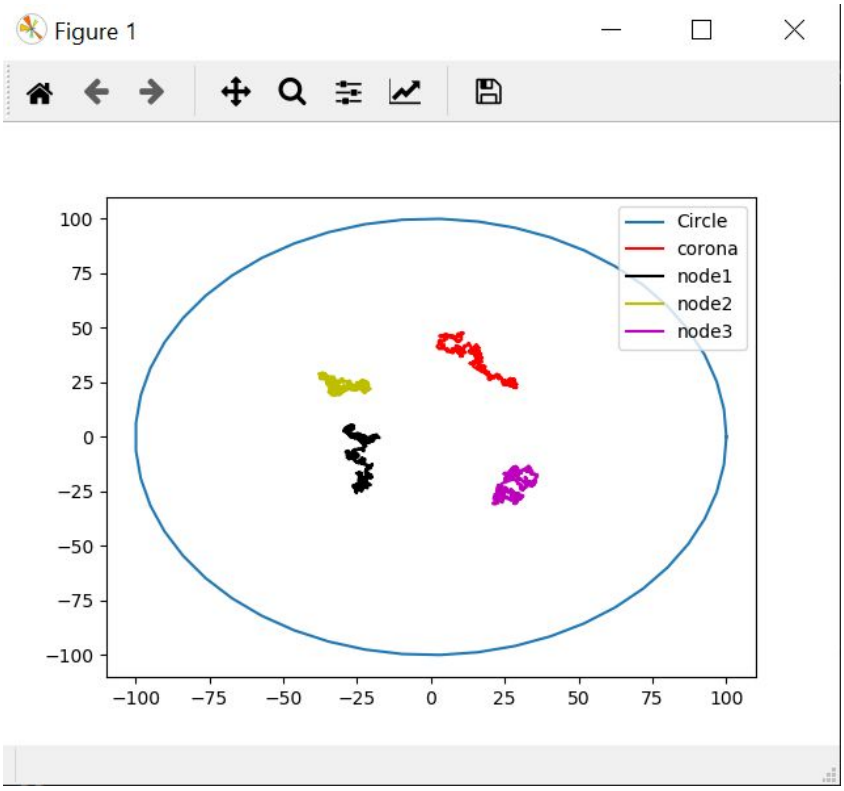
There were a number of applications employed using random walk which we found out through literature review. Our application shows the importance of social distancing and has the following features:

- It shows two simulations containing similar nodes, starting off at different positions.
- Most of those nodes represent healthy people, while the rest (shown in red) represent corona infected people.
- All of the nodes will be simulated using random walks.

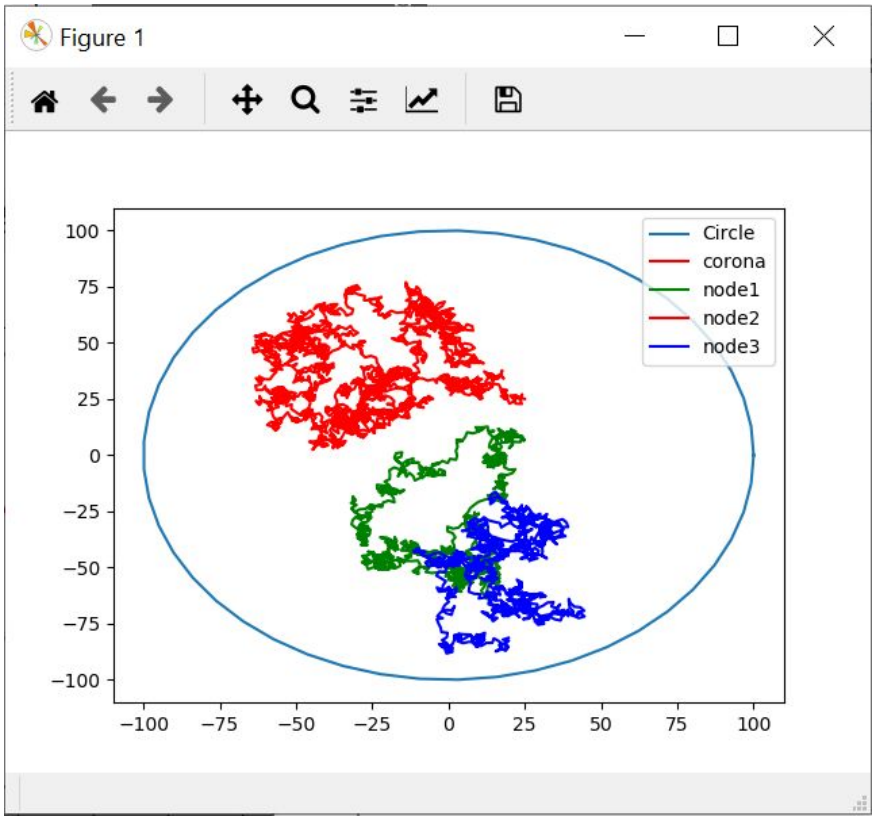
- If a healthy person comes in the vicinity of a corona infected person (<=6 distance between them), he also becomes corona infected and its color also changes to red.
- One of the simulations will show random walks with very small step size simulating social distancing (the person stays in their home or does not go beyond his vicinity).
- The other simulation will show random walks with a fairly bigger step size simulating that social distancing is not being practiced (the person goes far and wide).
- After a specified number of step sizes, we will compare both the simulations. The simulation which shows lack of social distancing would have more red nodes (indicating more infected people) than the simulation with a lesser number of red nodes.
- This is how our simulation will be able to demonstrate the importance of social distancing during the COVID-19 pandemic.

**Sample input:**
No. of Steps: 1000

**Sample output:**
With social distancing:



Without social distancing:




**Technical details:**

The code is similar to the one in Task 8 with the following changes:

- The Node class has two additional properties: corona and distance demonstrating whether a person is corona infected or not and whether they are practicing social distancing or not, respectively.
- If the person is following social distancing then his step size will be initialized to 0.5, otherwise the step size would be 2.
- The Node class has an additional method called plot, which plots the corona infected person in red while the rest of them in random colors.
- The Step class does not have a calculate_steps method since we do not need the number of steps. There is just one method which checks the distance between two nodes, if it is  less than or equal to 6 and one of the nodes is corona infected, the other node's property of corona would become true as well indicating that now that person is corona infected too.
- Another global function called draw_circle, which draws the circle just like it was drawing in previous tasks.
- Instead of simulating a histogram like the previous task, this would simulate the actual random walk of both the nodes where their colors would represent if they are corona infected or not.