MATH 371 : Assignment -2

-Chaitanya Garg

2021248

Question 1: Markov Chain

We model the writing of paper as a Markov Process having 4 discrete states. Those being read(r), write(w), email(e) and surf(s).

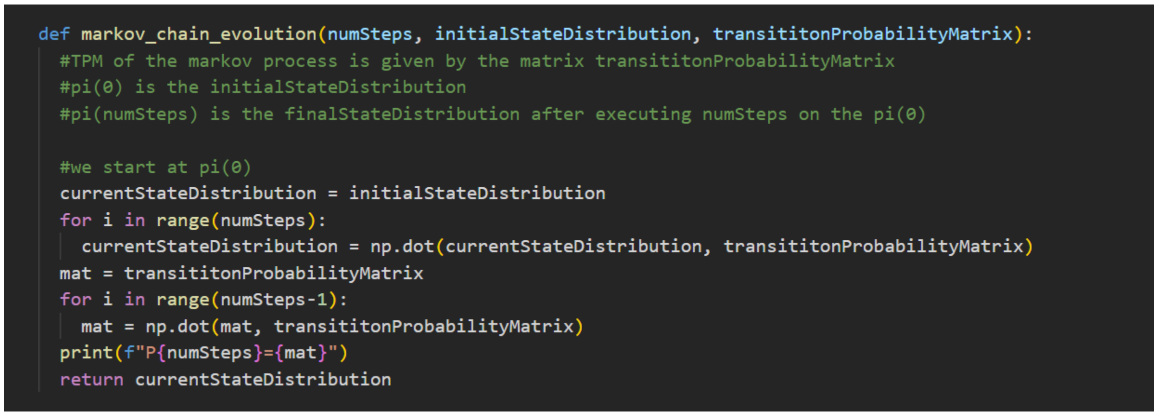
The Transition Probability Matrix of the Markov Process is given as:

$$P = \begin{array}{c c} & \begin{array}{cccc} r & w & e & s \end{array} \\ \begin{array}{c} r \\ w \\ e \\ s \end{array} & \left(\begin{array}{cccc} 0.5 & 0.3 & 0 & 0.2 \\ 0.2 & 0.5 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.3 & 0.3 \\ 0 & 0.2 & 0.3 & 0.5 \end{array} \right) \end{array}$$

In a Markov process P(Xj=xj|Xi=xi,Xi-1=xi-1,...,X1=x1)=P(Xj=xj|Xi=xi) where Xj,Xi,...X1 are random variables used to denote the output of the process at the time instant t for Xt and Xj is the event happening right after Xi.

If P is the TPM and pi(0) is the initial state distribution
Also, pi(t) = pi(0) * P^t (by property of Markov Chains)

To monitor Markov Chain Evolution, the following function is used-



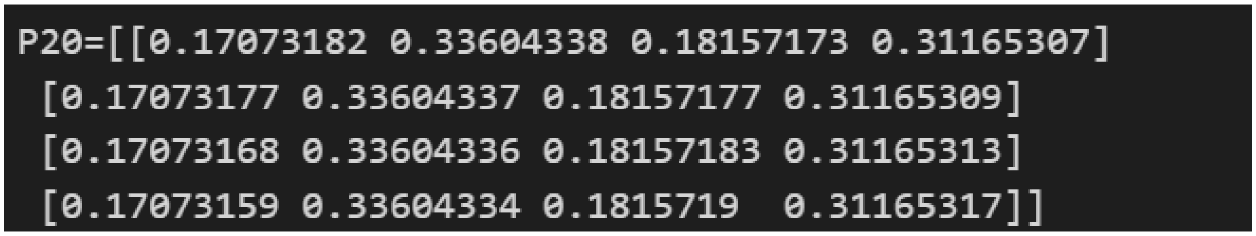
```python
def markov_chain_evolution(numSteps, initialStateDistribution, transititonProbabilityMatrix):
    #TPM of the markov process is given by the matrix transititonProbabilityMatrix
    #pi(0) is the initialStateDistribution
    #pi(numSteps) is the finalStateDistribution after executing numSteps on the pi(0)

    #we start at pi(0)
    currentStateDistribution = initialStateDistribution
    for i in range(numSteps):
        currentStateDistribution = np.dot(currentStateDistribution, transititonProbabilityMatrix)
    mat = transititonProbabilityMatrix
    for i in range(numSteps-1):
        mat = np.dot(mat, transititonProbabilityMatrix)
    print(f"P{numSteps}={mat}")
    return currentStateDistribution
```

a) To find markov evolution after 20 steps and P(X20=s|X0=r)

Markov Evolution done by using the markov_chain_evolution() fn.

pi(20) = pi(0)*P^20

P^20 is obtained as:



```
P20=[[0.17073182 0.33604338 0.18157173 0.31165307]
 [0.17073177 0.33604337 0.18157177 0.31165309]
 [0.17073168 0.33604336 0.18157183 0.31165313]
 [0.17073159 0.33604334 0.1815719  0.31165317]]
```

Taking pi(0) = [0.25 0.25 0.25 0.25]

pi(20) is [0.17073172 0.33604336 0.18157181 0.31165311]

To calculate P(X20=s|X0=r):

Taking pi(0)=[1 0 0 0]

Then obtain pi(20) = pi(0)*P^20 and check pi(20) fourth element which is 0.3116530652582661.

So, P(X20=s|X0=r) = 0.3116530652582661

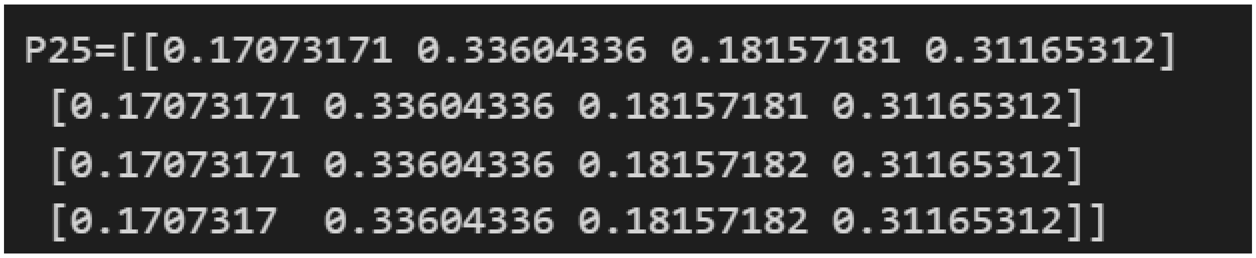
While pi(20) = [0.17073182 0.33604338 0.18157173 0.31165307]

Multiplying pi(0) with P^10 is essentially the same as checking the 1st row 4th column value of P^20

b) To find markov evolution after 25 steps and P(X25=s|X20=s)

Markov Evolution done by using the markov_chain_evolution() fn.

pi(25) = pi(0)*P^25
P^25 is obtained as:



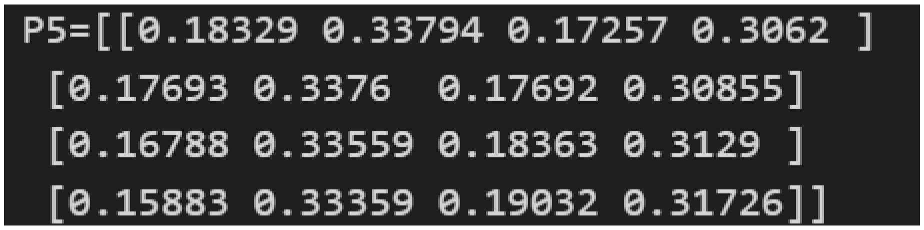
```
P25=[[0.17073171 0.33604336 0.18157181 0.31165312]
 [0.17073171 0.33604336 0.18157181 0.31165312]
 [0.17073171 0.33604336 0.18157182 0.31165312]
 [0.1707317  0.33604336 0.18157182 0.31165312]]
```

Taking pi(0) = [0.25 0.25 0.25 0.25]
pi(25) is [0.17073171 0.33604336 0.18157182 0.31165312]

To calculate P(X25=s|X20=s):
Taking pi(0)=[0 0 0 1]
We see P(X25=s|X20=s)=P(X5=s|X0=s) (by stat. increment)
Then obtain pi(5) = pi(0)*P^5 and check pi(5)'s fourth element which is 0.31726.
So, P(X25=s|X20=s)=P(X5=s|X0=s) = 0.31726
While pi(5) = [0.15883 0.33359 0.19032 0.31726]

Multiplying pi(0) with P^5 is essentially the same as checking the 4th row 4th column value of P^5

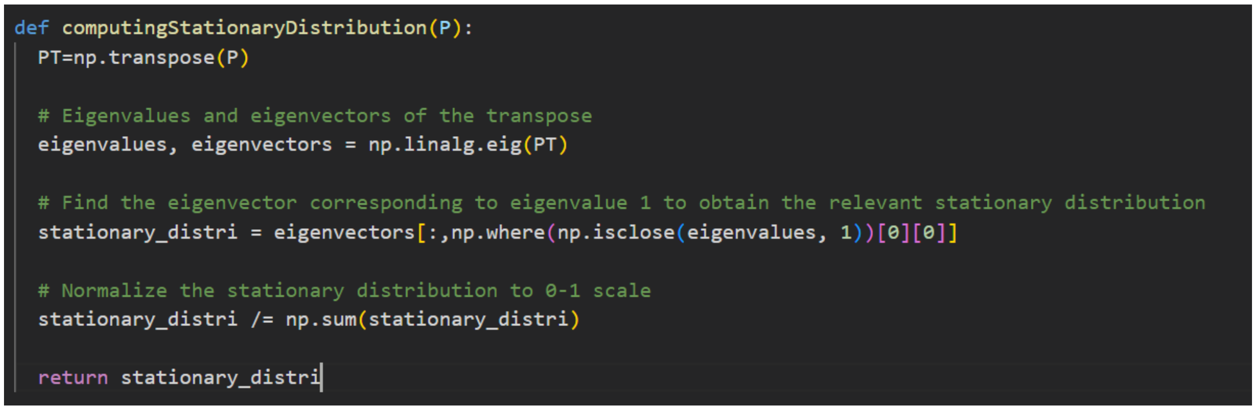
Where P^5 is:



```
P5=[[0.18329 0.33794 0.17257 0.3062 ]
 [0.17693 0.3376  0.17692 0.30855]
 [0.16788 0.33559 0.18363 0.3129 ]
 [0.15883 0.33359 0.19032 0.31726]]
```

c) We now make a new fn named as computingStationaryDistribution() which takes the TPM whose stationary distribution is to be found and then calculates the distribution.

Code is as:



```
def computingStationaryDistribution(P):
  PT=np.transpose(P)

  # Eigenvalues and eigenvectors of the transpose
  eigenvalues, eigenvectors = np.linalg.eig(PT)

  # Find the eigenvector corresponding to eigenvalue 1 to obtain the relevant stationary distribution
  stationary_distri = eigenvectors[:,np.where(np.isclose(eigenvalues, 1))[0][0]]

  # Normalize the stationary distribution to 0-1 scale
  stationary_distri /= np.sum(stationary_distri)

  return stationary_distri
```

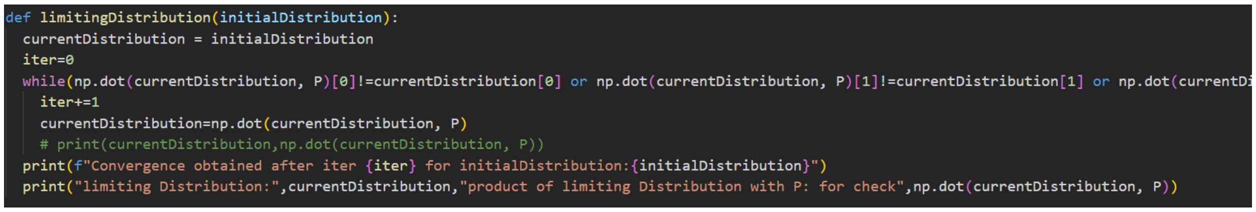
Steps followed:

1. P is the TPM taken as input argument
2. PT is the transpose of the TPM P
3. Using np.linalg.eig the corresponding eigenvalues and eigenvectors of PT are found out
4. We now take the eigenvector for which the corresponding eigenvalue is 1, the reason being
   Solving pi*P=pi
   We have 1 as eigenvalue of the above equation
5. Next we normalize the corresponding eigenvector for eigenvalue=1 and make it a unit vector as the sum of probabilities of a stateDistribution is always 1. This is ensured by normalization.

We apply the fn to our problem and observe that a stationary distribution exists and is given by
[0.17073171 0.33604336 0.18157182 0.31165312]
Yes, stationary distribution exists.

d) Now, to check for a limiting distribution we make a function with a while loop with works till convergence of the distribution

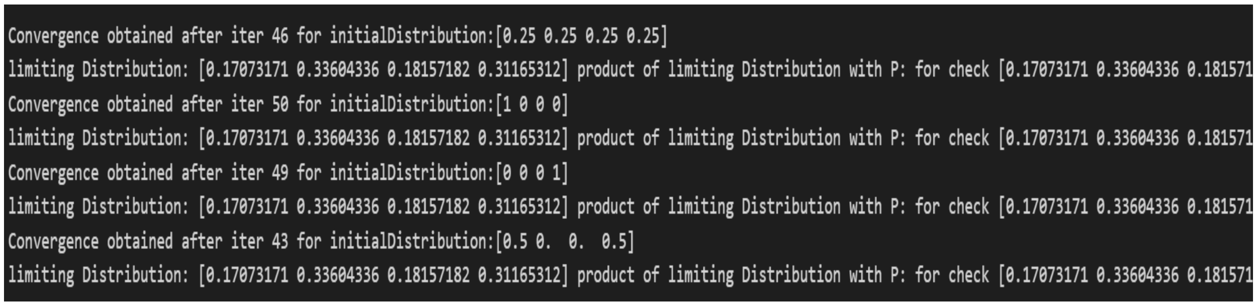
The code is as:



```
def limitingDistribution(initialDistribution):
  currentDistribution = initialDistribution
  iter=0
  while(np.dot(currentDistribution, P)[0]!=currentDistribution[0] or np.dot(currentDistribution, P)[1]!=currentDistribution[1] or np.dot(currentDi
    iter+=1
    currentDistribution=np.dot(currentDistribution, P)
    # print(currentDistribution,np.dot(currentDistribution, P))
  print(f"Convergence obtained after iter {iter} for initialDistribution:{initialDistribution}")
  print("limiting Distribution:",currentDistribution,"product of limiting Distribution with P: for check",np.dot(currentDistribution, P))
```

The function takes in the initial distribution as the argument and finds the limiting distribution along with the number of steps/iterations taking place for the same.

So, we see yes a limiting distribution does exist and check it for multiple inputs as given below:



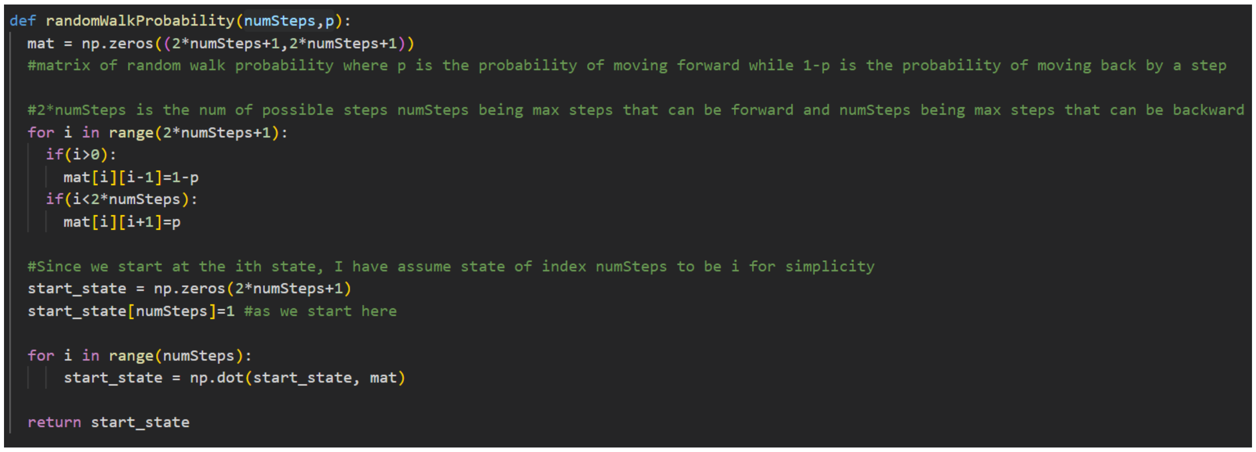
```
Convergence obtained after iter 46 for initialDistribution:[0.25 0.25 0.25 0.25]
limiting Distribution: [0.17073171 0.33604336 0.18157182 0.31165312] product of limiting Distribution with P: for check [0.17073171 0.33604336 0.181571
Convergence obtained after iter 50 for initialDistribution:[1 0 0 0]
limiting Distribution: [0.17073171 0.33604336 0.18157182 0.31165312] product of limiting Distribution with P: for check [0.17073171 0.33604336 0.181571
Convergence obtained after iter 49 for initialDistribution:[0 0 0 1]
limiting Distribution: [0.17073171 0.33604336 0.18157182 0.31165312] product of limiting Distribution with P: for check [0.17073171 0.33604336 0.181571
Convergence obtained after iter 43 for initialDistribution:[0.5 0.  0.  0.5]
limiting Distribution: [0.17073171 0.33604336 0.18157182 0.31165312] product of limiting Distribution with P: for check [0.17073171 0.33604336 0.181571
```

Question 2: 1-D Random Walk

Two functions are implemented randomWalkProbability(numSteps,p) and RandomWalkSimulator(numSteps,p)

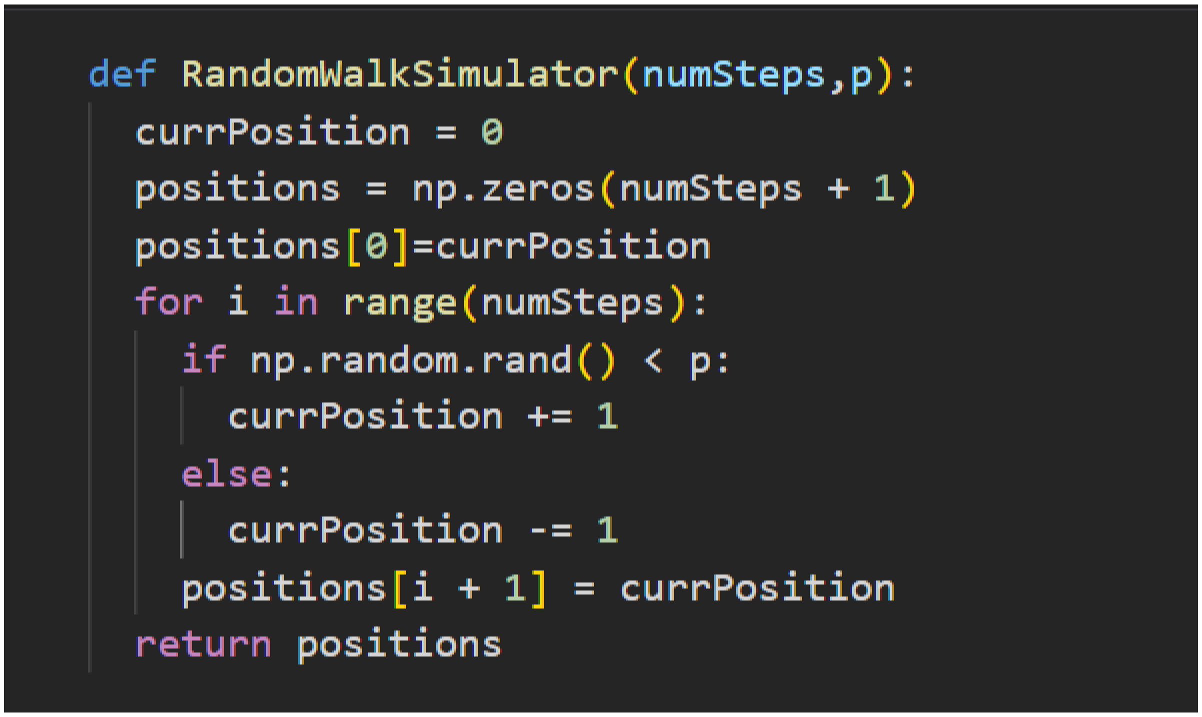
randomWalkProbability(numSteps,p) takes in as input as the numSteps and p(probability of moving forward)

Now a TPM, P is constructed of dimensions (2*numSteps+1,2*numSteps+1) and parameters are set accordingly P[i][i-1]=1-p and P[i][i+1]=p to handle the corresponding movements P^numSteps gives the mathematical probability estimate of reaching a location. For the purpose of simplicity I have assumed, instead of starting at step i, we are at step 0 and backward movement means along the negative axis and vice-versa.



```python
def randomWalkProbability(numSteps,p):
  mat = np.zeros((2*numSteps+1,2*numSteps+1))
  #matrix of random walk probability where p is the probability of moving forward while 1-p is the probability of moving back by a step

  #2*numSteps is the num of possible steps numSteps being max steps that can be forward and numSteps being max steps that can be backward
  for i in range(2*numSteps+1):
    if(i>0):
      mat[i][i-1]=1-p
    if(i<2*numSteps):
      mat[i][i+1]=p

  #Since we start at the ith state, I have assume state of index numSteps to be i for simplicity
  start_state = np.zeros(2*numSteps+1)
  start_state[numSteps]=1 #as we start here

  for i in range(numSteps):
      start_state = np.dot(start_state, mat)

  return start_state
```

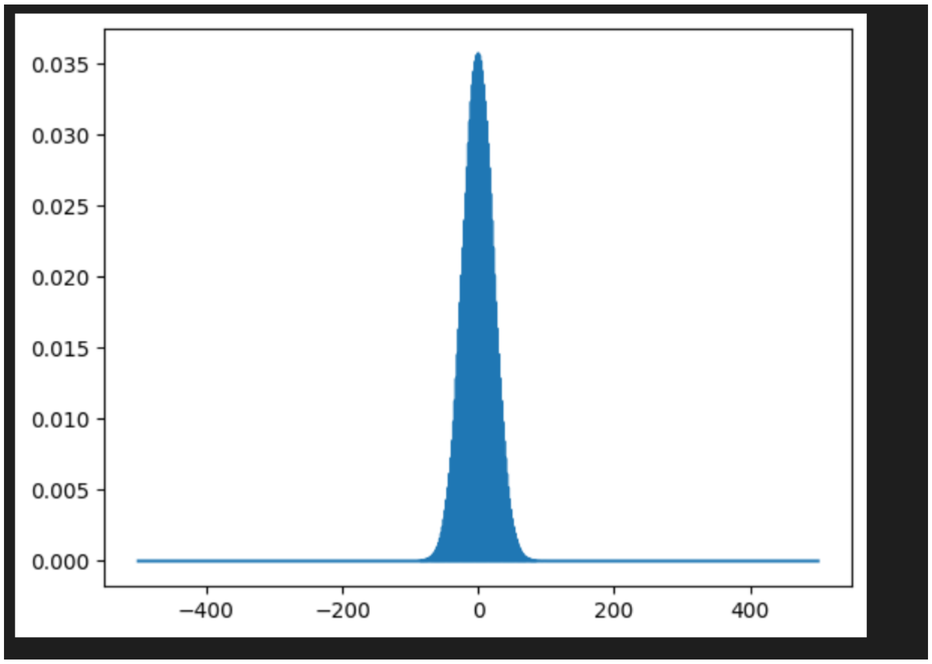
Next, is the implementation of RandomWalkSimulator(numSteps,p) which takes in the numSteps(number of Steps) and p(probability of moving forward) and executes the corresponding random walk and displays it path



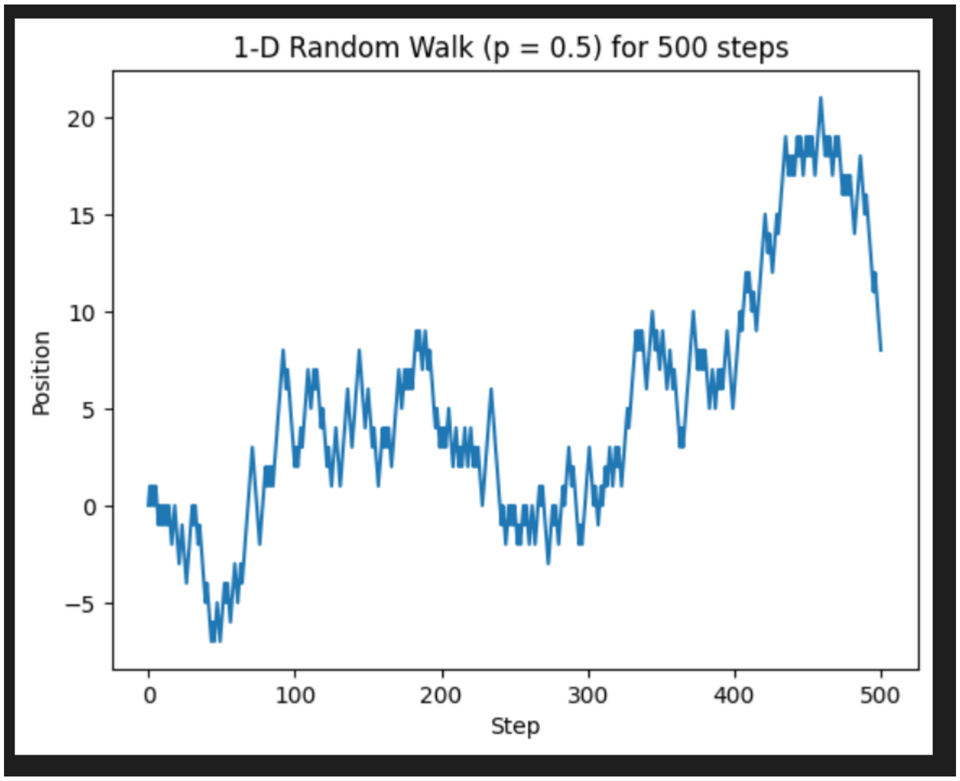
```python
def RandomWalkSimulator(numSteps,p):
  currPosition = 0
  positions = np.zeros(numSteps + 1)
  positions[0]=currPosition
  for i in range(numSteps):
    if np.random.rand() < p:
      currPosition += 1
    else:
      currPosition -= 1
    positions[i + 1] = currPosition
  return positions
```

a) numSteps = 500 and p=0.5

Mathematical probability plot of random walk as found as the function randomWalkProbability(numSteps,p)

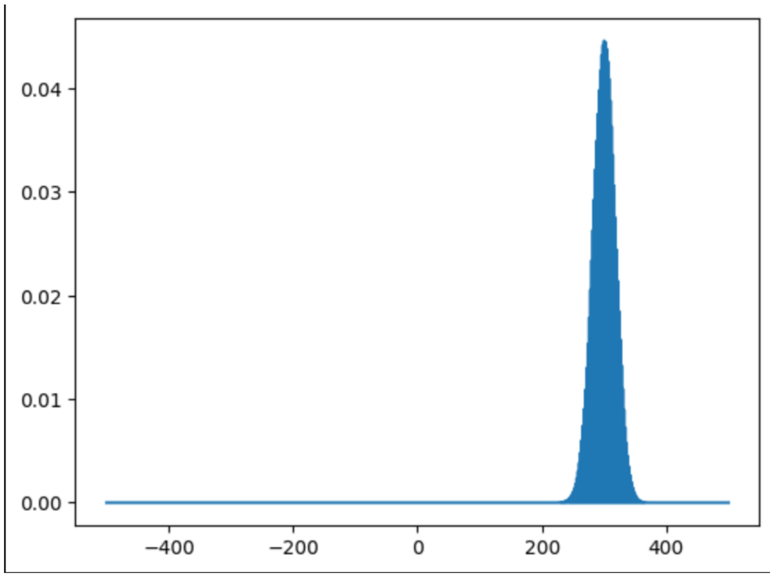


Next one random walk is executed by using RandomWalkSimulator(numSteps,p)

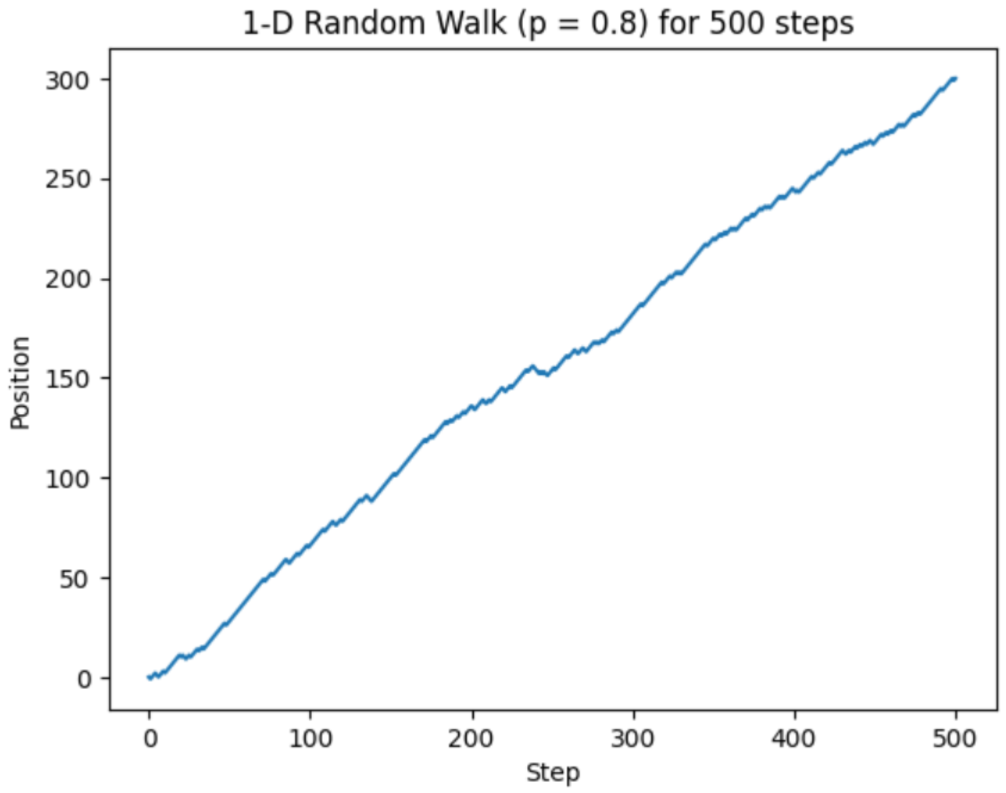


b)numSteps = 500 and p=0.8

Mathematical probability plot of random walk as found as the function randomWalkProbability(numSteps,p)



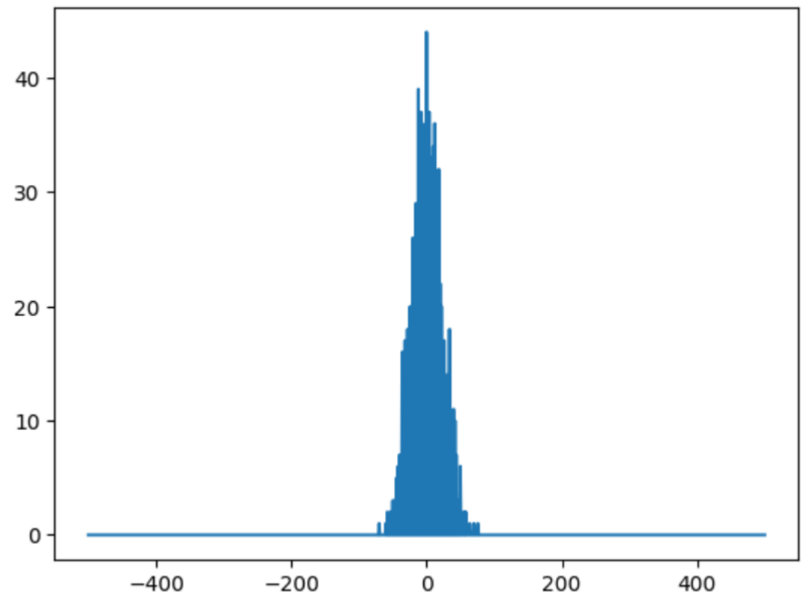
Next one random walk is executed by using RandomWalkSimulator(numSteps,p)



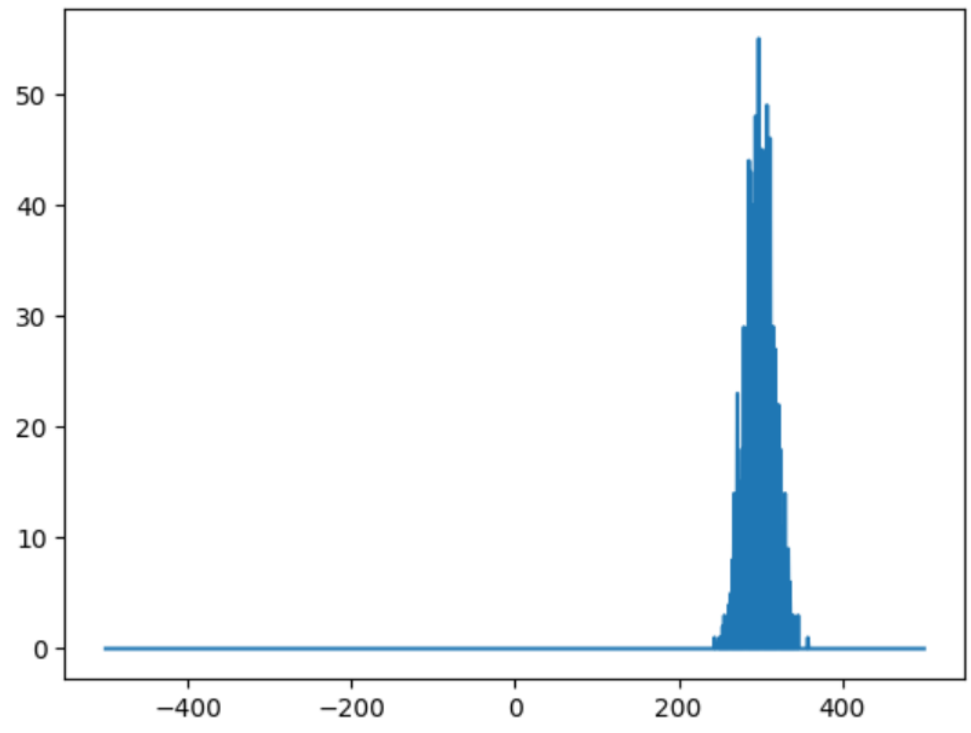
Comparisons:
i) Mathematical Distribution shifts to the right as there is more probability to move forward then back in b)
ii) Both mathematical are gaussian curves since, movement is done in a uniform way
iii) Path of b) is forward going which is again to be expected as there is more probability to move forward then back in b)

c) Simulating the process for 1000 times for p1=0.5 and p2=0.8 each composing of 500 steps we obtain the following

For p=0.5:

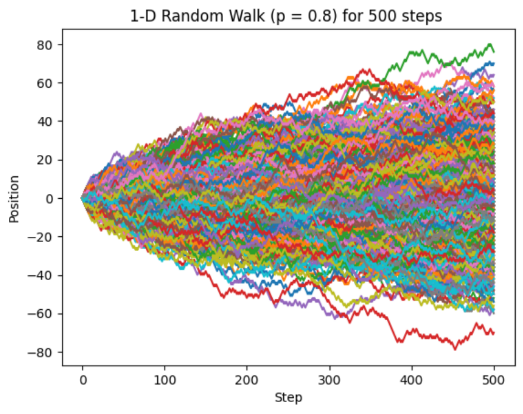


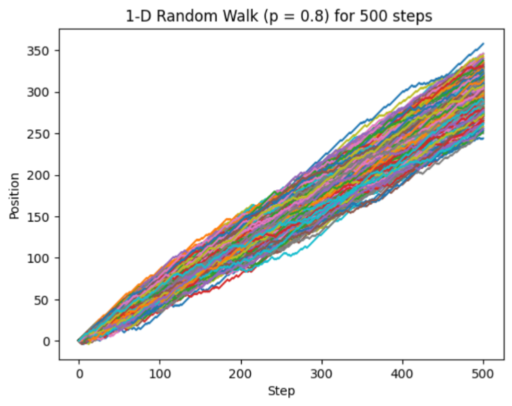
For p=0.8:



In the above graph after each random walk is executed, it’s finals points location is increment by 1, then after a 1000 iterations the final plots are plotted to test out how similar they are to the mathematical model

Comparisons:
i) The graphs obtained closely resemble the mathematically modeled graphs of a) and b) which were gaussian and so are these.
ii) As, these graphs are gaussian, a random walk with probability p is utterly close to that being done with a mathematical model
iii) To see the path variations (pics attached below), we see for a fair process the paths are spread out while a bias one, concentrated at or near a location





This file is the original work of Chaitanya Garg (2021248).
Discussed with:
Tanishq Jain and Raghav Sakuja