Fatmanur Yaman-2019402204-IE203-HW2

Matrix Generation

I generate a function called generate_transition_prob_matrix to obtain the probability matrices that the values in the rows equal to 1.

In [1]:
```
import numpy as np
from collections import Counter
```

In [2]:
```
def generate_transition_prob_matrix(n):
    matrix = np.random.rand(n+1,n+1)
    for i in range(n+1):
        matrix[i] = matrix[i]/np.sum(matrix[i])
    print('The probability matrix is: ')
    print(matrix)
```

In [5]:
```
generate_transition_prob_matrix(5)
```

```
The probability matrix is:
[[0.24699403 0.05254914 0.25392523 0.22384289 0.1839045  0.03878422]
 [0.1183883  0.0703641  0.06445878 0.279344   0.26067078 0.20677404]
 [0.1377907  0.27593526 0.18412503 0.23003671 0.03725343 0.13485888]
 [0.03767324 0.20028249 0.0900469  0.15891705 0.1299921  0.38308821]
 [0.17214468 0.0029967  0.21903951 0.15086425 0.19727163 0.25768324]
 [0.28746794 0.26103512 0.26165862 0.0075681  0.15189018 0.03038003]]
```

In [6]:
```
generate_transition_prob_matrix(25)
```

```
The probability matrix is:
[[0.05593127 0.04214442 0.03438998 0.05576025 0.04372927 0.06423386
  0.01711114 0.00764319 0.02101092 0.05215566 0.02068882 0.05353539
  0.00665825 0.06142522 0.0623578  0.02335227 0.06262842 0.06129817
  0.00239422 0.0633603  0.05708739 0.01904671 0.00717194 0.01986463
  0.04526127 0.03975924]
 [0.02101037 0.01220568 0.01480443 0.05765004 0.00719957 0.005823
  0.05007034 0.05240681 0.04592991 0.03281498 0.03359753 0.02371419
  0.06177195 0.03909503 0.0080294  0.06199072 0.05013118 0.05775721
  0.06361152 0.05041519 0.00395421 0.06675399 0.03229853 0.05857886
  0.03271046 0.0556749 ]
 [0.03956733 0.04419134 0.04707905 0.0486105  0.03814573 0.00627706
  0.03339389 0.06823158 0.02486799 0.04355367 0.04167049 0.05935119
  0.0741948  0.00854914 0.04186359 0.01744359 0.03338676 0.02902329
  0.03000645 0.02547389 0.00186459 0.03189498 0.07605644 0.06161809
  0.01429744 0.05938715]
 [0.04584264 0.04193133 0.02440795 0.03612505 0.07022841 0.00165212
  0.001409   0.00336748 0.06698524 0.06619732 0.02453789 0.05705831
  0.01971116 0.03976616 0.03444769 0.03109931 0.06036691 0.03183449
```

In [7]:
```
generate_transition_prob_matrix(50)
```

```
The probability matrix is:
[[0.01462797 0.0343296  0.02255115 ... 0.01681337 0.02740019 0.0151155 ]
 [0.01127875 0.03196809 0.00212428 ... 0.00079318 0.01883036 0.00358736]
 [0.00171946 0.02933281 0.0395913  ... 0.02308419 0.01547672 0.04277437]
 ...
 [0.00270769 0.01839346 0.0016344  ... 0.01900756 0.00239439 0.02559295]
 [0.02513162 0.00659106 0.00924132 ... 0.03141635 0.02584217 0.00969428]
 [0.02720191 0.01745541 0.02913828 ... 0.02125263 0.03814802 0.01962977]]
```

Monte Carlo Simulation

I defined a function called monte_carlo_simulation that takes n and M and returns the steady state distribution. First, it finds the upper limits of the states by adding up the probability values. Then, it looks for the new state where the value of the current state is in. It updates the current state and stores which states that the algorithm visits in the list called X_list. The algorithm stops when it visits the states M times. It counts the number of times that each step is observed in X_list. Then it divides the number of counts to the total length of the list and stores them in the list called Pi_list. Pi_list gives the steady state distribution.

In [3]:
```
def monte_carlo_simulation(n,M):
    matrix = np.random.rand(n+1,n+1)
    for i in range(n+1):
        matrix[i] = matrix[i]/np.sum(matrix[i])
    print('The probability matrix is: ')
    print(matrix)
    current_state = np.random.randint(n+1)
    X_list = [current_state]
    for b in range(M):
        r = np.random.rand(1)
        CDF_list = []
        value = 0
        for k in range(n+1):
            value += matrix[current_state][k]
            CDF_list.append(value)
        for m in range(n):
            if r <= CDF_list[m+1] and r >= CDF_list[m]:
                current_state = m
                X_list.append(m)
    Pi_list = []
    count = Counter()
    for j in range(n+1):
        Pi_list.append(X_list.count(j)/M)
    print('The steady state distribution is: ')
    print(Pi_list)
```

In [8]:
```
monte_carlo_simulation(5,2000)
```

```
The probability matrix is:
[[0.16552578 0.19096872 0.15648476 0.14564331 0.16320194 0.17817549]
 [0.06852791 0.23687741 0.01510642 0.43400165 0.24152068 0.00396593]
 [0.12433635 0.16849497 0.23650678 0.26744038 0.12854172 0.0746798 ]
 [0.14429291 0.39870122 0.02897698 0.34192538 0.01669289 0.06941062]
 [0.14584765 0.25203428 0.13024997 0.15223483 0.22506511 0.09456817]
 [0.11498189 0.25563251 0.12733638 0.1123317  0.10421542 0.28550209]]
The steady state distribution is:
[0.2465, 0.1335, 0.2485, 0.146, 0.1, 0.0]
```

In [9]:
```
monte_carlo_simulation(25,2000)
```

```
 7.42324124e-02 2.68772290e-02]
[4.03659939e-02 5.41965312e-02 1.65578629e-02 3.64607250e-02
 3.47699042e-02 3.08424073e-02 7.33085356e-02 5.14188952e-02
 4.11729494e-02 3.21436997e-02 1.23576019e-02 8.01963641e-02
 7.72994186e-03 2.94783531e-02 5.28820617e-03 4.39620795e-02
 6.74282027e-02 9.16048119e-02 1.22182833e-02 5.75532109e-02
 8.39452364e-02 9.14182410e-03 3.81399331e-02 7.41264366e-03
 3.39706738e-02 8.33512905e-03]
[4.00010306e-02 1.14396615e-02 3.78540359e-02 1.06322009e-02
 2.69743424e-02 8.47206555e-02 5.36443274e-02 2.82936593e-02
 3.83554247e-02 3.06644104e-02 1.27077622e-03 1.49460023e-02
 8.56552313e-02 4.77479837e-02 5.28445892e-02 9.93945311e-03
 4.25719289e-02 3.83206309e-02 2.75830617e-02 7.44051567e-02
 1.09240401e-02 7.83744601e-02 2.46484709e-02 3.01541031e-02
 1.92206018e-02 7.88137615e-02]]

The steady state distribution is:
[0.02, 0.033, 0.029, 0.039, 0.034, 0.043, 0.043, 0.0455, 0.044, 0.0435, 0.045
5, 0.037, 0.0275, 0.0385, 0.044, 0.031, 0.041, 0.0335, 0.039, 0.042, 0.04, 0.
0405, 0.0405, 0.053, 0.042, 0.0]
```

In [10]:
```
monte_carlo_simulation(50,2000)
```

```
The probability matrix is:
[[0.01726091 0.0303174  0.02783993 ... 0.01601308 0.0047771  0.0179838 ]
 [0.01519795 0.02528434 0.01265616 ... 0.02186775 0.0256309  0.02470171]
 [0.02638526 0.00527387 0.00440257 ... 0.00231437 0.02618944 0.01850053]
 ...
 [0.01198153 0.00815675 0.03331573 ... 0.01403559 0.00220798 0.01269429]
 [0.03069806 0.03298096 0.03077869 ... 0.03336268 0.00870492 0.0145693 ]
 [0.00696845 0.01048581 0.03748246 ... 0.01583755 0.01835186 0.00167087]]
The steady state distribution is:
[0.024, 0.0225, 0.02, 0.0185, 0.0255, 0.0195, 0.0175, 0.0185, 0.019, 0.0155, 0.
0165, 0.0185, 0.02, 0.0185, 0.02, 0.0185, 0.0165, 0.0195, 0.0235, 0.0255, 0.01
7, 0.0145, 0.0235, 0.019, 0.015, 0.0175, 0.019, 0.0255, 0.018, 0.0165, 0.0185,
0.0215, 0.022, 0.0205, 0.029, 0.016, 0.0205, 0.02, 0.017, 0.0225, 0.019, 0.018,
0.02, 0.0145, 0.015, 0.0205, 0.0205, 0.019, 0.0235, 0.0235, 0.0]
```

Monte Carlo Simulation with Absorbing State

In [18]:
```
n = 5
M = 2000
epsilon = 0.0005
matrix = np.random.rand(n,n+1)
for i in range(n-1):
    matrix[i] = matrix[i]/np.sum(matrix[i])
matrix = np.vstack((matrix,[0,0,0,0,0,1]))
print('The probability matrix is: ')
print(matrix)
current_state = np.random.randint(n+1)
X_list = [current_state]
for b in range(M):
    r = np.random.rand(1)
    CDF_list = []
    value = 0
    for k in range(n+1):
        value += matrix[current_state][k]
        CDF_list.append(value)
    for m in range(n):
        if r <= CDF_list[m+1] and r >= CDF_list[m]:
            current_state = m
            X_list.append(m)
Pi_list = []
count = Counter()
for j in range(n+1):
    Pi_list.append(X_list.count(j)/M)
print('The steady state distribution is: ')
print(Pi_list)
```

```
The probability matrix is:
[[0.00873133 0.20615599 0.12227159 0.19498119 0.24821932 0.21964058]
 [0.11290846 0.0322857  0.21803586 0.23844503 0.13346398 0.26486096]
 [0.12266871 0.21310154 0.24696812 0.06749518 0.13372379 0.21604265]
 [0.2266326  0.29024472 0.01775534 0.16051284 0.21842294 0.08643156]
 [0.1365415  0.83557838 0.02296452 0.47689888 0.70370049 0.45370927]
 [0.         0.         0.         0.         0.         1.        ]]
The steady state distribution is:
[0.3175, 0.104, 0.1345, 0.1745, 0.16, 0.0]
```

Matrix Multiplication

I defined a function called matrix_multiplication that takes n and epsilon value and returns the steady state distribution. It multiplicates the matrices until the values in the rows converge. The convergence is being controlled by looking at the sum of the difference between the average values of the columns and the values in a randomly chosen row. If the difference is smaller than the given epsilon value, the steady state distribution is obtained.

In [4]:
```
def  matrix_multiplication(n,epsilon):
    k = 0
    matrix = np.random.rand(n+1,n+1)
    for i in range(n+1):
        matrix[i] = matrix[i]/np.sum(matrix[i])
    while(True):
        k += 1
        matrix = np.dot(matrix,matrix)
        average = matrix.mean(axis=0)
        r = np.random.randint(n)
        difference = matrix[r]-average
        average_list = []
        avr_difference = 0
        for j in range(n):
            avr_difference += difference[j]
        avr_difference = avr_difference / (n+1)
        if (abs(avr_difference) < epsilon):
            break
    print('The steady state distirbution is: ')
    print(matrix)
```

In [20]:
```
matrix_multiplication(5,0.0005)
```

```
The steady state distirbution is:
[[0.14366437 0.22307677 0.14152114 0.1879527  0.16813892 0.1356461 ]
 [0.14282037 0.22236061 0.13976576 0.19010911 0.1683845  0.13655965]
 [0.14563034 0.22384991 0.14170274 0.18421579 0.17013044 0.13447079]
 [0.14394287 0.22444138 0.14063688 0.18738581 0.16825181 0.13484125]
 [0.14398218 0.2210211  0.14268339 0.18668058 0.16936707 0.13626568]
 [0.142246   0.22129972 0.14142394 0.19041355 0.16747706 0.13713973]]
```

Comments

If there is an absorbing state, there cannot be a steady state. It can be understood that the algorithms focus only on the absorbing states instead of the whole matrix. Thus, the result contains 1's and 0's.