Vinh Vu

Dr. Chassiakos

EE 381

April 22nd, 2019

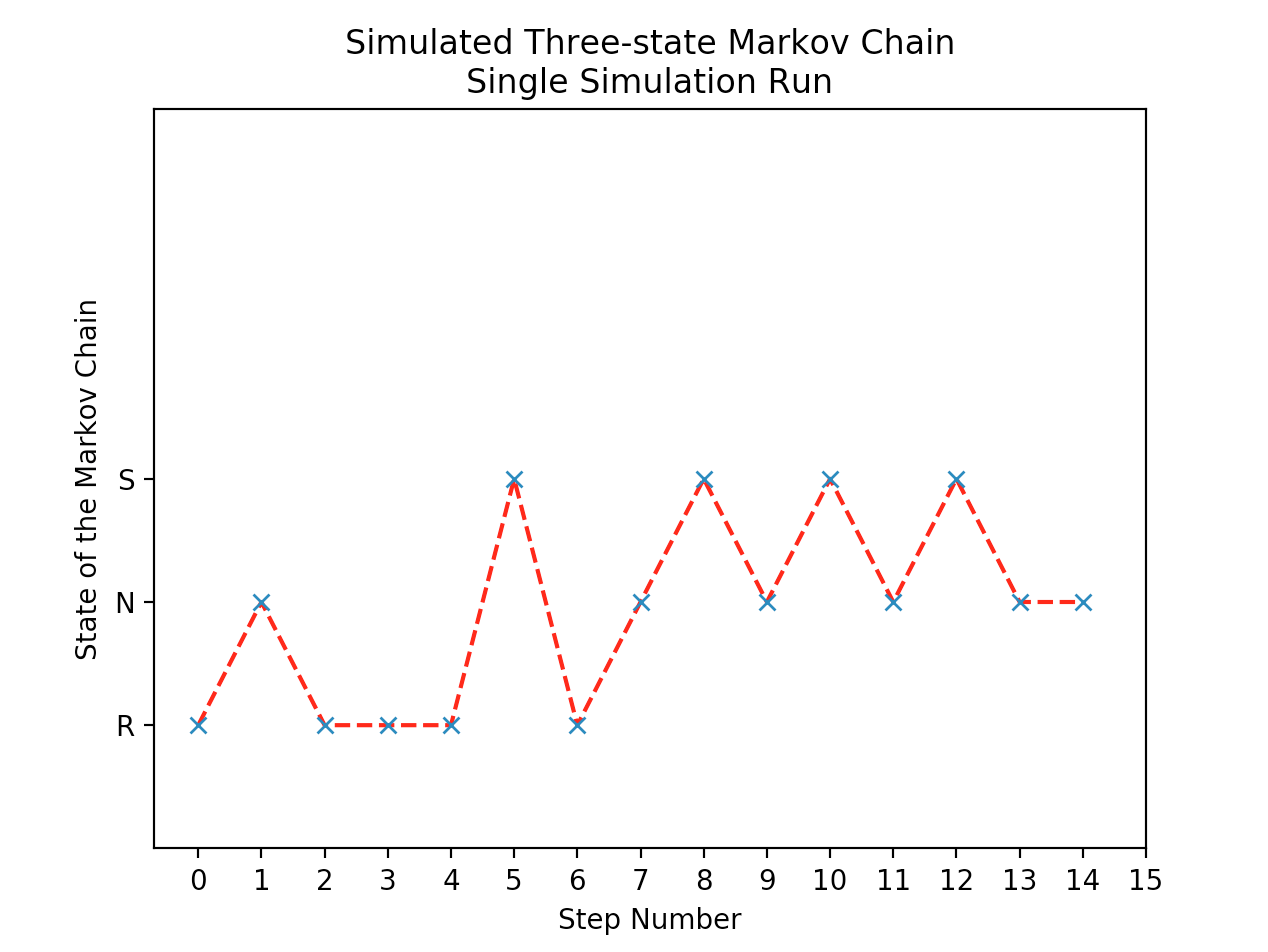
Project 6: Markov Chains

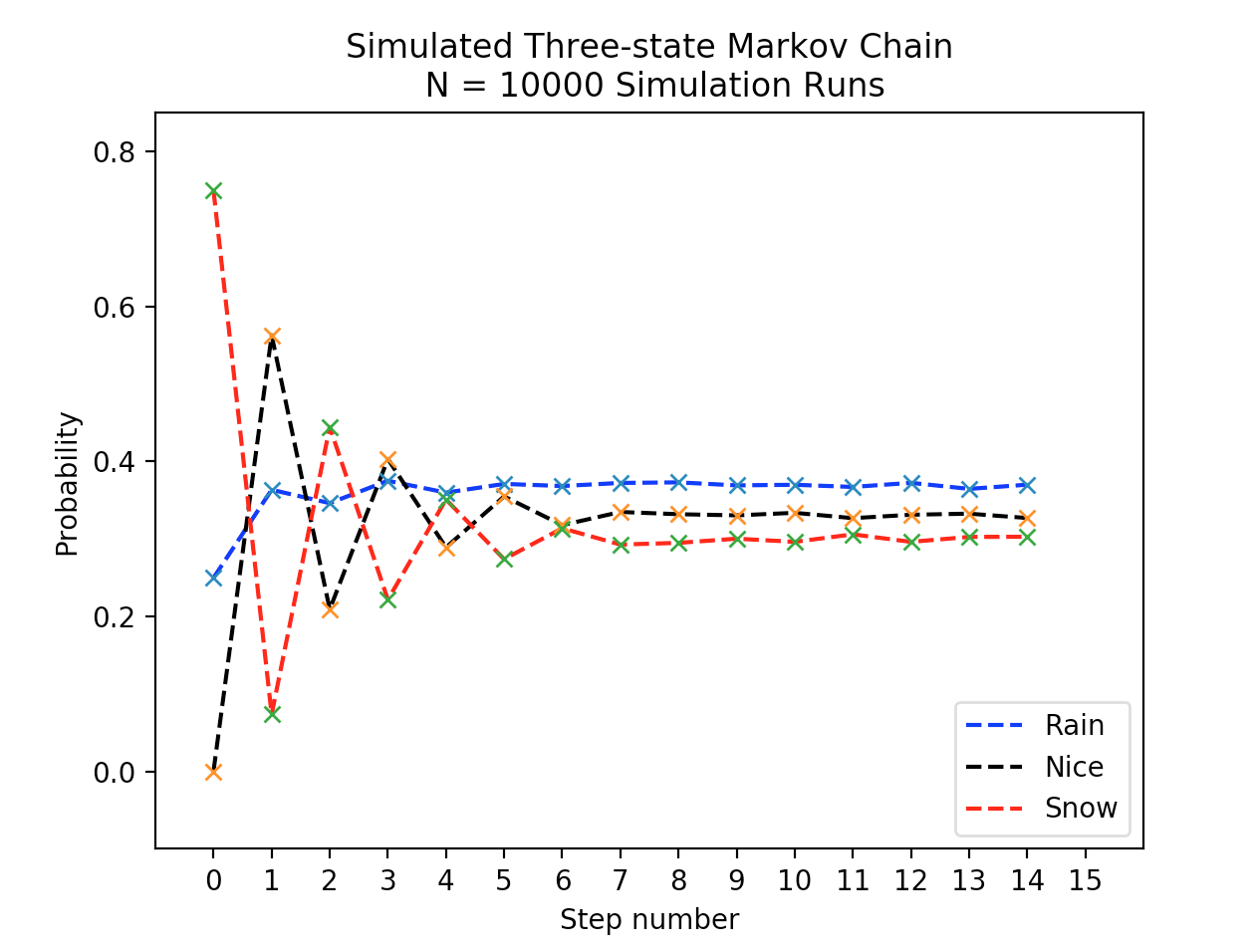
**Problem 1: A three-state Markov Chain**

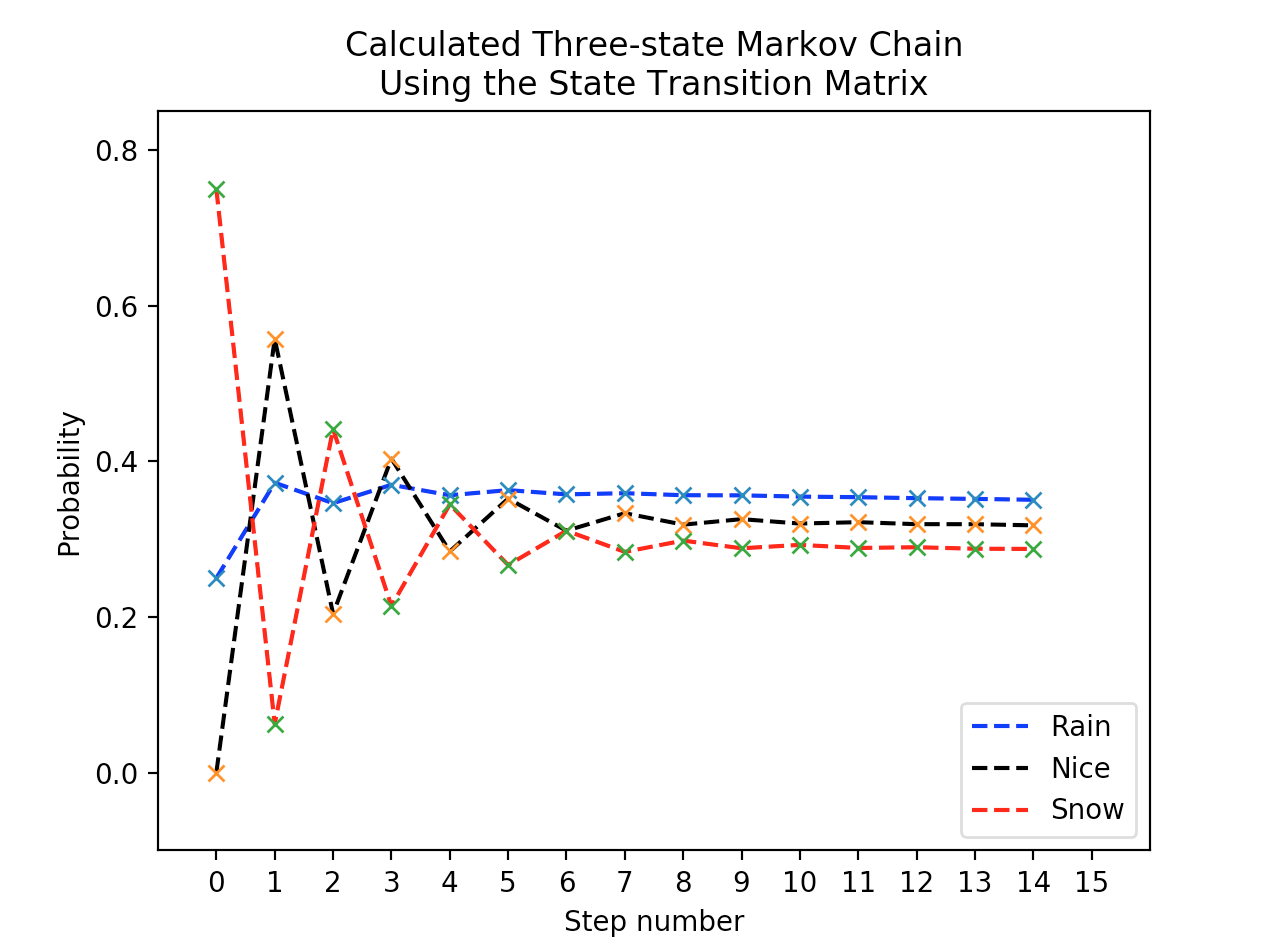
**Introduction:** We are given a state transition matrix and initial probability vector for the weather. There are three states for the weather: R (rain), N (nice), and S (snow). We are asked to use this state transition matrix and initial probability vector to generate the Markov Chain that will predict the weather for the next 15 days. In order to generate the initial weather state, an initial probability vector is used. After generating the initial weather state, the state transition matrix is utilized to determine the next day’s weather. There are a few graphs to be generated: single run over 15 days, N=10000 experiments with 15 days each, state transition matrix approach. All results are being compared at the end.

**Methodology:** In order to create the Markov Chains for the three weather conditions, a function is created with its parameters being three empty vectors for R (rain), N (nice), and S (snow) and the two probability vectors of state transition and initial. The function generateChain() with its parameters generates the three-state Markov chain for each weather condition is randomly generate with a uniform probability between 0 and 1. The probability is then compared to the state transition matrix to decide the next day’s weather. Based on the new weather, the probability for the new state is updated accordingly. After generating the three-state Markov chains, they are being graphed as the single-simulation run for 15 days. The next experiment consists of the same logical steps but is repeated 10,000 times. The averages for the weather chains are calculated and graphed. The next experiment repeats the simulation 10,000 times with 15 days but this time using the matrix. The methodology for the matrix includes multiplying the probabilities for every new weather condition for 15 days. The lists which store the probabilities are graphed and compared to the simulated probability graph from experiment 2.

**Results and Conclusion:**







**Appendix:**

import numpy as np

import random

import matplotlib.pyplot as plt

def generateChain(init,stateTrans, chain):

steps = list(range(0,15))

for step in steps:

x = np.random.uniform(0,1)

if x <= init[0]:

chain[step] = 'R'

nextState = stateTrans[0]

if x > init[0] and x <= init[0]+init[1]:

chain[step] = 'N'

nextState = stateTrans[1]

if x > init[0]+init[1]:

chain[step] = 'S'

nextState = stateTrans[2]

init = nextState

return chain, steps

def generateChains(init,stateTrans,chainR, chainN, chainS):

steps = list(range(0,15))

for step in steps:

x = np.random.uniform(0,1)

if x <= init[0]:

chainR[step] += 1

nextState = stateTrans[0]

if x > init[0] and x <= init[0]+init[1]:

chainN[step] += 1

nextState = stateTrans[1]

if x > init[0]+init[1]:

chainS[step] += 1

nextState = stateTrans[2]

init = nextState

return chainR, chainN, chainS, steps

def generateChainUsingMatrix(init,stateTrans,chainR, chainN, chainS):

p = stateTrans

curr = init

for x in range(0,15):

chainR[x] = curr[0]

chainN[x] = curr[1]

chainS[x] = curr[2]

curr = init@p

p = p@stateTrans

return chainR, chainN, chainS

def graphChain(chain,steps):

plt.plot(steps,chain,'r--')

plt.plot(steps,chain,'x')

plt.title("Simulated Three-state Markov Chain\nSingle Simulation Run")

plt.ylabel("State of the Markov Chain")

plt.xlabel("Step Number")

plt.ylim(-1,5)

plt.xticks(np.arange(0, 16, 1))

plt.show()

def graphChains10000(chainR,chainN,chainS,steps):

maxNum = max(max(chainR),max(chainN),max(chainS))

minNum = min(min(chainR),min(chainN),min(chainS))

plt.plot(steps,chainR,'b--',label="Rain")

plt.plot(steps,chainN,'k--',label="Nice")

plt.plot(steps,chainS,'r--',label="Snow")

plt.plot(steps,chainR,'x')

plt.plot(steps,chainN,'x')

plt.plot(steps,chainS,'x')

plt.title("Simulated Three-state Markov Chain\n N = 10000 Simulation Runs")

plt.ylabel("Probability")

plt.xlabel("Step number")

axes = plt.gca()

axes.set\_xlim([-1,16])

axes.set\_ylim([minNum-0.1,maxNum + 0.1])

plt.xticks(np.arange(0, 16, 1.0))

plt.legend(loc='lower right')

plt.show()

def graphChainsMatrix(chainR,chainN,chainS,steps):

maxNum = max(max(chainR),max(chainN),max(chainS))

minNum = min(min(chainR),min(chainN),min(chainS))

plt.plot(steps,chainR,'b--',label="Rain")

plt.plot(steps,chainN,'k--',label="Nice")

plt.plot(steps,chainS,'r--',label="Snow")

plt.plot(steps,chainR,'x')

plt.plot(steps,chainN,'x')

plt.plot(steps,chainS,'x')

plt.title("Calculated Three-state Markov Chain\nUsing the State Transition Matrix")

plt.ylabel("Probability")

plt.xlabel("Step number")

axes = plt.gca()

axes.set\_xlim([-1,16])

axes.set\_ylim([minNum-0.1,maxNum + 0.1])

plt.xticks(np.arange(0, 16, 1.0))

plt.legend(loc='lower right')

plt.show()

# Given state transition and initial probabilities

stateTrans = np.array([[0.5,0.25,0.25],[0.25,0.125,0.625],[0.33,0.66,0]])

init = np.array([0.25,0,0.75])

# Create empty lists for each weather condition with 0 filled 15 elements

R = [0]\*15

N = [0]\*15

S = [0]\*15

chain = [0]\*15

# Generate three-state Markov chain for single run

chains, steps = generateChain(init, stateTrans, chain)

graphChain(chains, steps)

# Generate three-state Markov chain for 10,000 runs

experiments = 10000

for x in range(0,experiments):

chainR, chainN, chainS, steps = generateChains(init, stateTrans,R, N, S)

chainLength = 15

for y in range(0,chainLength):

chainR[y] = chainR[y]/experiments

chainN[y] = chainN[y]/experiments

chainS[y] = chainS[y]/experiments

graphChains10000(chainR, chainN, chainS, steps)

# Generate three-state Markov chain using Matrix

chainR, chainN, chainS = generateChainUsingMatrix(init, stateTrans,R, N, S)

graphChainsMatrix(chainR, chainN, chainS, steps)

**Problem 2: The Google PageRank Algorithm**

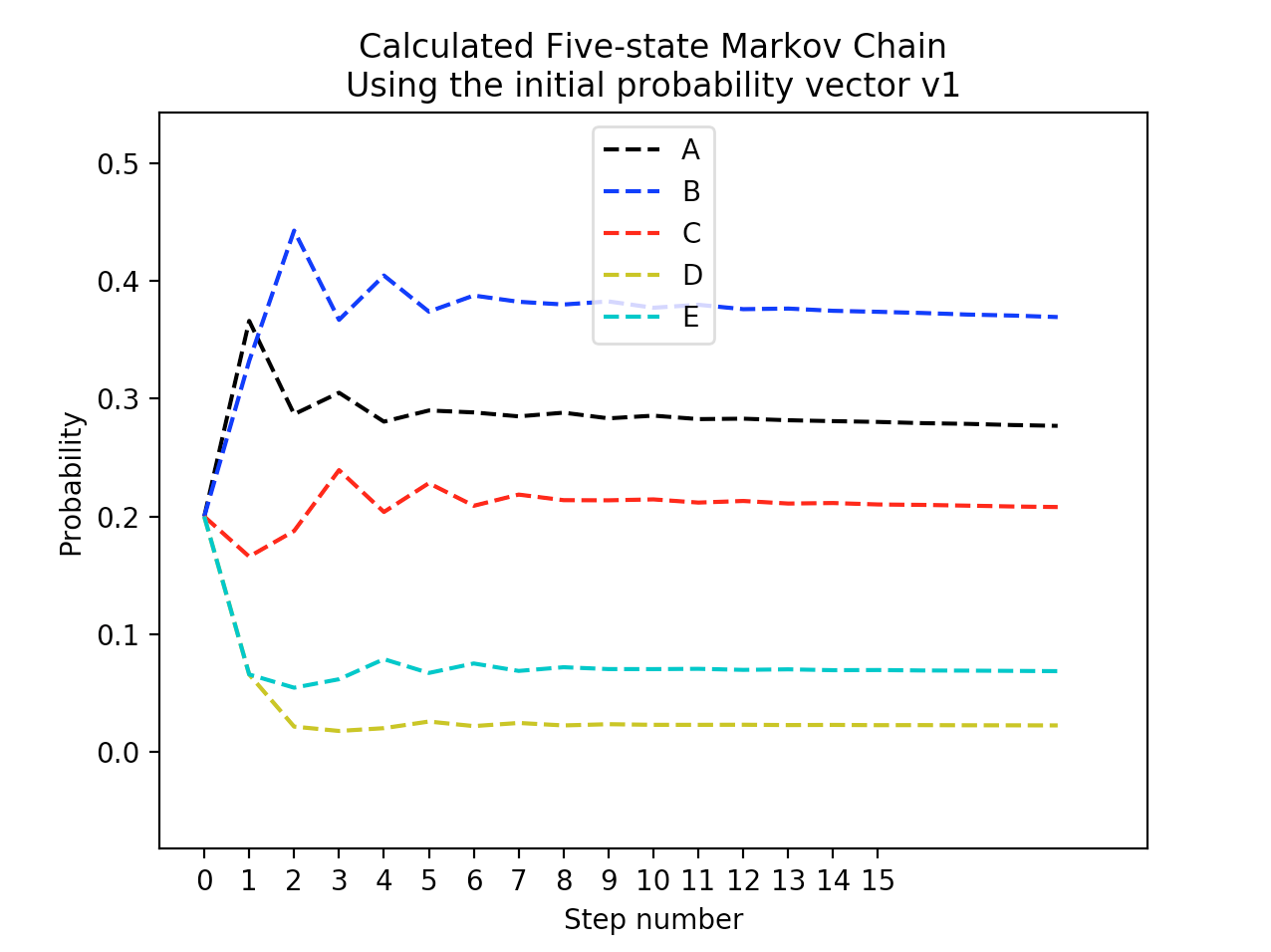
**Introduction:** This problem presents an introduction to the algorithm used for ranking web pages for searching purposes. Google uses much-advanced PageRank algorithm to rise to the top of all web search engines. Initially, all pages are equally likely to be visited with the initial state probability distribution vector being v1 = [A B C D E] = [0.2 0.2 0.2 0.2 0.2]. We are calculating the probability vectors for 20 steps only using the state transition matrix approach, not the simulation of the chain. After yielding calculations from all the web pages, we rank the pages {A, B, C, D, E} in order of importance. Once we have the calculated chains, we graph the chain for each of the five states versus the number of steps for n=1, 2,…20. The second experiment is repeated with the initial state probability distribution vector changed to v2 = [A B C D E] = [0 0 0 0 1].

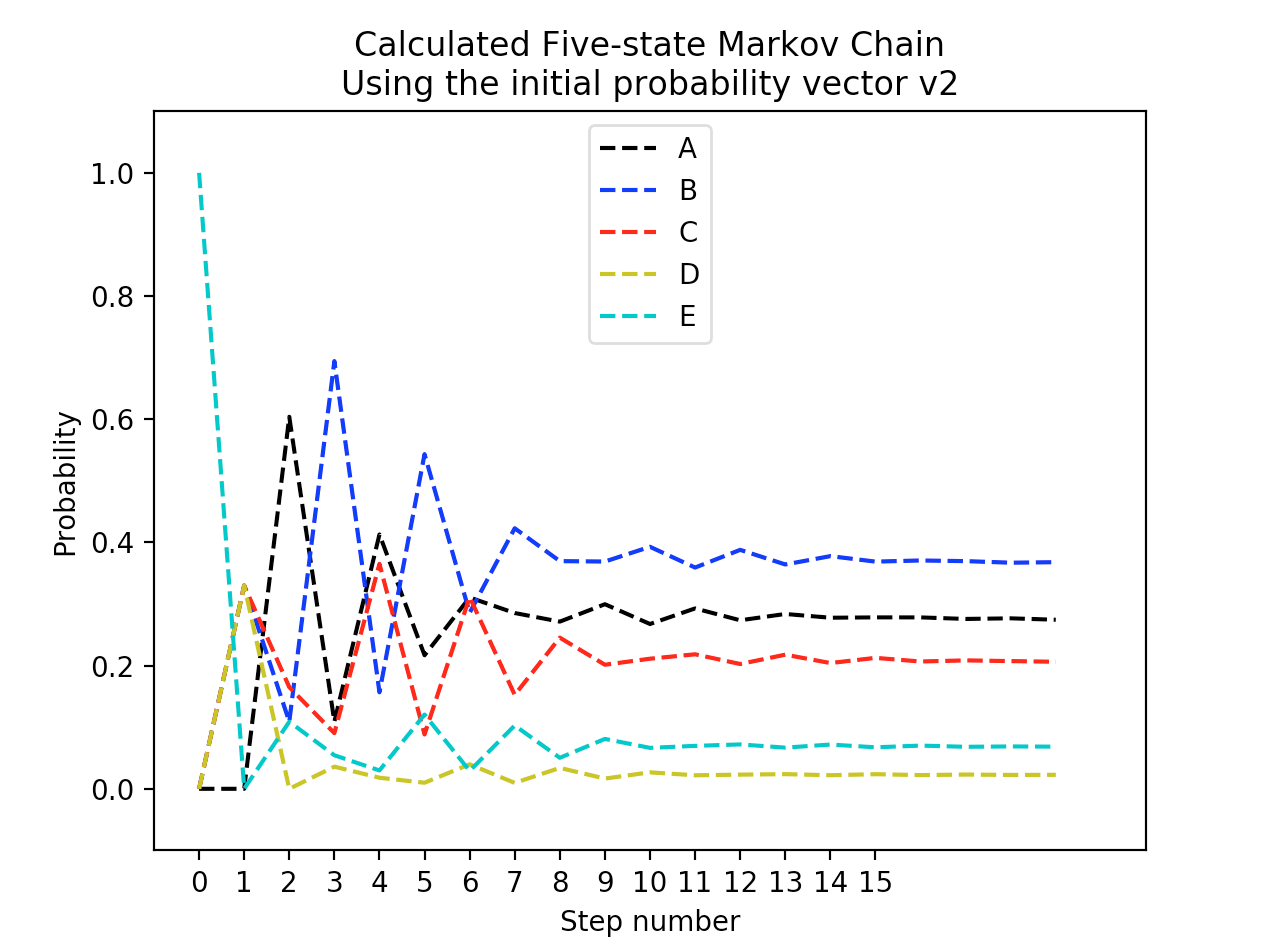
**Methodology:** We are generating the Markov Chain in the manner as Problem 1 but with step number being 20 and five states instead of three. After generating the full chain using the state transition matrix calculation, we rank the most likely to be visited webpage based on its final probability. We graph the chain for each state with the same methodology for comparison. For the second experiment, the same process is repeated with the second initial probability vector.

**Results and Conclusion:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **State Transition Matrix** | | | | | |
| **Page** | **A** | **B** | **C** | **D** | **E** |
| **A** | 0 | 1 | 0 | 0 | 0 |
| **B** | 0.5 | 0 | 0.5 | 0 | 0 |
| **C** | 0.333 | 0.333 | 0 | 0 | 0.333 |
| **D** | 1 | 0 | 0 | 0 | 0 |
| **E** | 0 | 0.333 | 0.333 | 0.333 | 0 |

|  |  |  |
| --- | --- | --- |
| Initial probability vector: v1 | | |
| Rank | Page | Probability |
|  |  |  |
| 1 | B | 0.3692 |
| 2 | A | 0.2769 |
| 3 | C | 0.2080 |
| 4 | E | 0.0688 |
| 5 | D | 0.0278 |





|  |  |  |
| --- | --- | --- |
| Initial probability vector: v2 | | |
| Rank | Page | Probability |
|  |  |  |
| 1 | B | 0.3678 |
| 2 | A | 0.2744 |
| 3 | C | 0.2062 |
| 4 | E | 0.0684 |
| 5 | D | 0.0227 |

**Appendix:**

import numpy as np

import random

import matplotlib.pyplot as plt

def generateChainUsingMatrix2(init,stateTrans,chainA, chainB, chainC, chainD, chainE, chainLength):

p = stateTrans

curr = init

for x in range(0,chainLength):

chainA[x] = curr[0]

chainB[x] = curr[1]

chainC[x] = curr[2]

chainD[x] = curr[3]

chainE[x] = curr[4]

curr = init@p

p = p@stateTrans

return chainA, chainB, chainC, chainD, chainE

def graphChainsMatrix(chainA,chainB,chainC,chainD,chainE, steps, chainLength,vector):

maxNum = max(max(chainA),max(chainB),max(chainC),max(chainD),max(chainE))

minNum = min(min(chainA),min(chainB),min(chainC),min(chainD),min(chainE))

plt.plot(steps,chainA,'k--',label="A")

plt.plot(steps,chainB,'b--',label="B")

plt.plot(steps,chainC,'r--',label="C")

plt.plot(steps,chainD,'y--',label="D")

plt.plot(steps,chainE,'c--',label="E")

plt.title("Calculated Five-state Markov Chain\nUsing the initial probability vector v" + str(vector))

plt.ylabel("Probability")

plt.xlabel("Step number")

axes = plt.gca()

axes.set\_xlim([-1,chainLength + 1])

axes.set\_ylim([minNum-0.1,maxNum + 0.1])

plt.xticks(np.arange(0, 16, 1.0))

plt.legend(loc='upper center')

plt.show()

# Generate first experiment with equal initial probability

steps = list(range(0,20))

stateTrans = np.array([[0,1,0,0,0],[0.5,0,0.5,0,0],[0.33,0.33,0,0,0.33],[1,0,0,0,0],[0,0.33,0.33,0.33,0]])

init = np.array([0.2,0.2,0.2,0.2,0.2])

chainLength = 20

A = [0]\*chainLength

B = [0]\*chainLength

C = [0]\*chainLength

D = [0]\*chainLength

E = [0]\*chainLength

vector = 1

chainA, chainB, chainC, chainD, chainE = generateChainUsingMatrix2(init, stateTrans,A, B, C, D, E, chainLength)

graphChainsMatrix(chainA, chainB, chainC, chainD, chainE, steps, chainLength,vector)

print("prob of Page A: ",chainA[len(chainA)-1])

print("prob of Page B: ",chainB[len(chainB)-1])

print("prob of Page C: ",chainC[len(chainC)-1])

print("prob of Page D: ",chainD[len(chainD)-1])

print("prob of Page E: ",chainE[len(chainE)-1])

# Generate second experiment with E being initial webpage

vector = 2

init = np.array([0,0,0,0,1])

chainA, chainB, chainC, chainD, chainE = generateChainUsingMatrix2(init, stateTrans,A, B, C, D, E, chainLength)

graphChainsMatrix(chainA, chainB, chainC, chainD, chainE, steps, chainLength,vector)

print("prob of Page A: ",chainA[len(chainA)-1])

print("prob of Page B: ",chainB[len(chainB)-1])

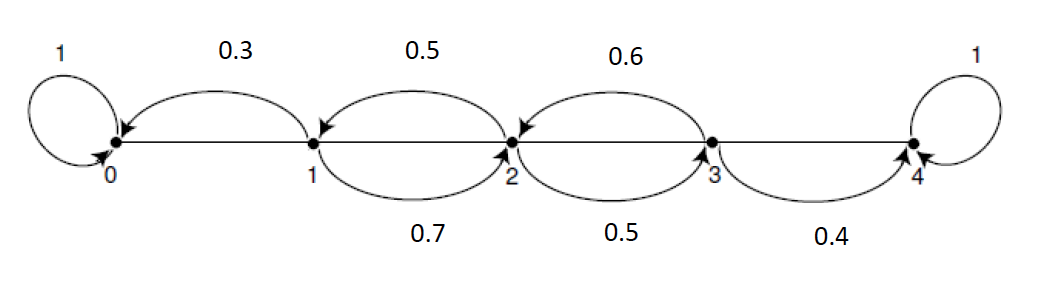
print("prob of Page C: ",chainC[len(chainC)-1])

print("prob of Page D: ",chainD[len(chainD)-1])

print("prob of Page E: ",chainE[len(chainE)-1])

**Problem 3: Simulate a five-state absorbing Markov Chain**

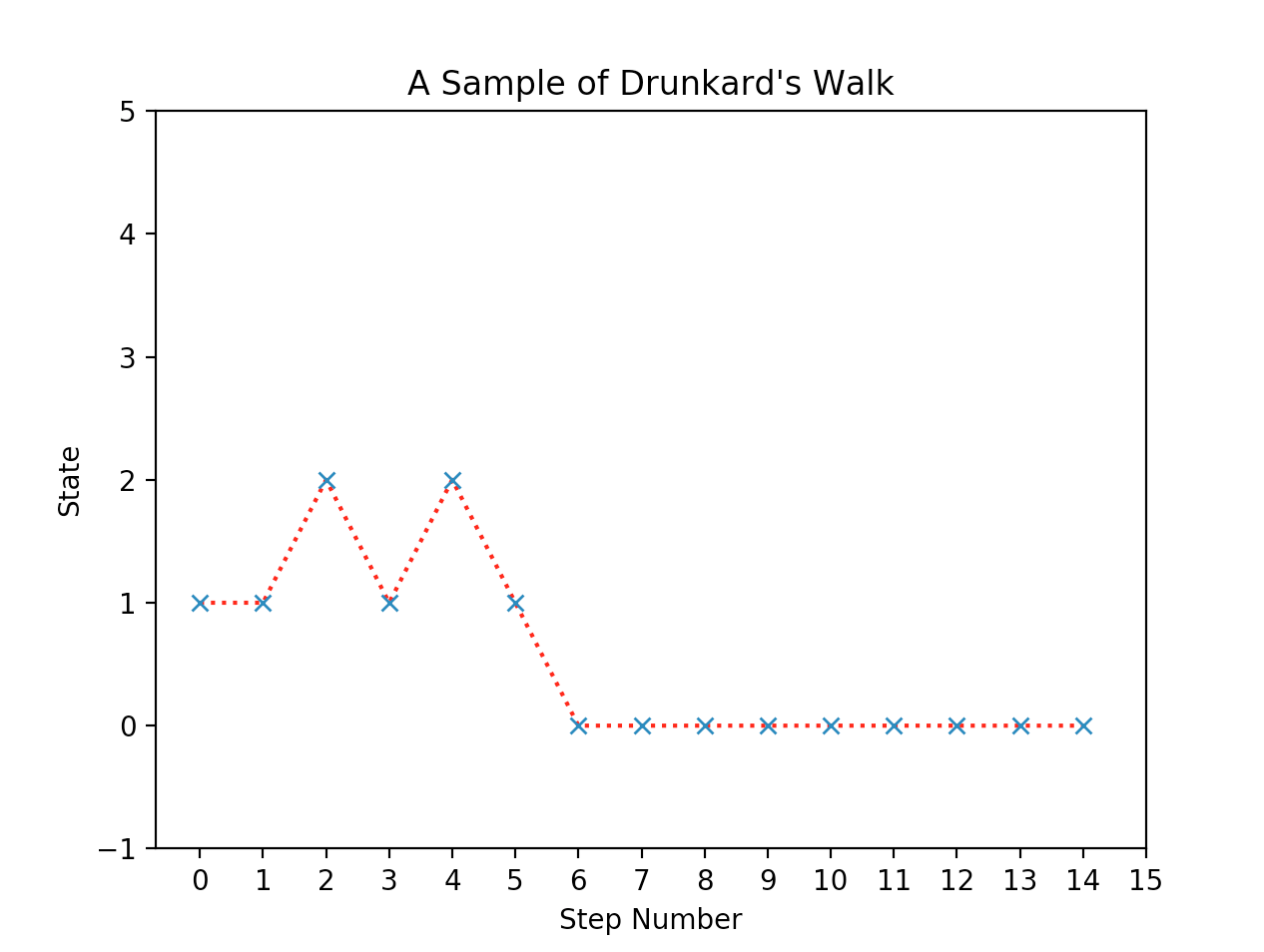
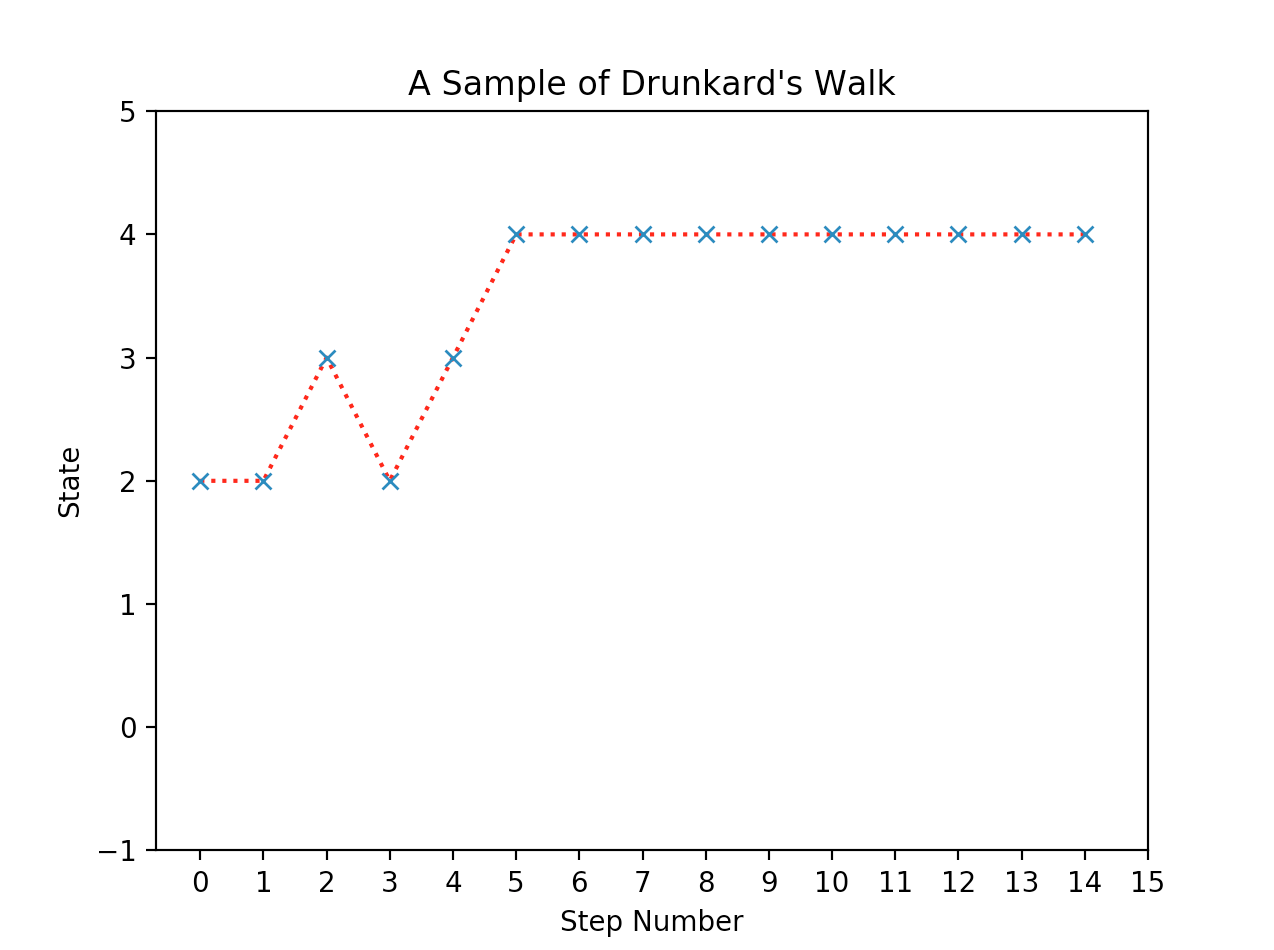
**Introduction:** This experiment requires us to simulate a Markov chain based on the “Drunkard’s Walk.” Drunkard’s Walk is an experiment where one state can only transition to its neighborhood states. There is the concept of absorbing state, which means that the chain is stuck at a certain state forever. For this particular experiment, the absorbing states are state 0 and state 4. Below is the state transition illustration. We are to define the State Transition Matrix and simulate two runs: absorbed by state 0, absorbed by state 4.



**Methodology:** The initial state can start at a random transient state, meaning either state 1, 2, or 3. Those states can move in both directions of forward and backward. Once the initial state is simulated, based on the state transition matrix probabilities, the next 15 steps are generated uniformly random. When reaching a new state, the probability vector gets updated every step. In order to update the vector accurately, multiple conditions are used. Each state has different probabilities of moving forward and backward. For example, if the current state is 1, the randomly generated chance gets split into two decision: if less than or equal to 0.3 then it moves to state 0, if it is greater than 0.3 then it moves to state 2. The same logic applies to all the state for decision. If either state 0 or 4 is reached, the Markov Chain gets absorbed and stays there for the rest of the 15 steps. After generating the single-run simulation, the result is graph in a properly labeled figure.

**Results and Conclusion:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **State Transition Matrix** | | | | | |
| **State** | **0** | **1** | **2** | **3** | **4** |
| **0** | 1 | 0 | 0 | 0 | 0 |
| **1** | 0.3 | 0 | 0.7 | 0 | 0 |
| **2** | 0 | 0.5 | 0 | 0.5 | 0 |
| **3** | 0 | 0 | 0.6 | 0 | 0.4 |
| **4** | 0 | 0 | 0 | 0 | 1 |



**Appendix:**

import numpy as np

import random

import matplotlib.pyplot as plt

def compareLists(listA,listB):

for i in range(len(listA)):

if listA[i] != listB[i]:

return False

return True

def generateChain(init,stateTrans, chain):

steps = list(range(1,15))

x = np.random.uniform(0,1)

if x <= 0.33:

chain[0] = 1

init = stateTrans[1]

elif x<= 0.66 and x>0.33:

chain[0] = 2

init = stateTrans[2]

elif x > 0.66:

chain[0] = 3

init = stateTrans[3]

for step in steps:

x = np.random.uniform(0,1)

if (compareLists(init,stateTrans[0])):

chain[step] = 0

if x <= 1:

nextState = stateTrans[0]

if (compareLists(init,stateTrans[1])):

chain[step] = 1

if x <= 0.3:

nextState = stateTrans[0]

else:

nextState = stateTrans[2]

if (compareLists(init,stateTrans[2])):

chain[step] = 2

if x <= 0.5:

nextState = stateTrans[1]

else:

nextState = stateTrans[3]

if (compareLists(init,stateTrans[3])):

chain[step]=3

if x <= 0.6:

nextState = stateTrans[2]

else:

nextState = stateTrans[4]

if (compareLists(init,stateTrans[4])):

chain[step] = 4

if x <= 1:

nextState = stateTrans[4]

init = nextState

return chain, steps

def graphChains(chain,steps):

plt.plot(steps,chain,'r:')

plt.plot(steps,chain,'x')

plt.title("A Sample of Drunkard's Walk")

plt.ylabel("State")

plt.xlabel("Step Number")

plt.ylim(-1,5)

plt.xticks(np.arange(0, 16, 1))

plt.show()

stateTrans = np.array([[1,0,0,0,0],[0.3,0,0.7,0,0],[0,0.5,0,0.5,0],[0,0,0.6,0,0.4],[0,0,0,0,1]])

init=[0,0.33,0.33,0.33,0]

chain = [0]\*15

chain, steps = generateChain(init, stateTrans, chain)

steps.insert(0,0)

graphChains(chain,steps)

**Problem 4: Compute the Probability of absorption using the simulated chain**

**Introduction:** For this experiment, we are simulating the same five-state absorbing Markov Chain. The main difference is the initial state is always state 2 with the initial probability vector being [0 0 1 0 0]. The same simulation is repeated 10,000 times. We are to keep track of how often the chain ended at state 0 and state 4. Based on the counts, the probabilities of absorption of b20 and b24 is calculated.

**Methodology:** The same code and procedure are utilized in Problem 3. An additional feature is a for-loop to generate 10,000 five-state absorption Markov chains. Two variables are used to keep track of state 0 and state 4 absorption by checking the last element of the chains. The probabilities are recorded at the end of the simulation.

**Results and Conclusions:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Absorption Probabilities**  (via simulations) | | | |
| b20 | 0.3976 | b24 | 0.5302 |

**Appendix:**

import numpy as np

import random

import matplotlib.pyplot as plt

def compareLists(listA,listB):

for i in range(len(listA)):

if listA[i] != listB[i]:

return False

return True

def generateChain(init,stateTrans, chain):

steps = list(range(1,15))

x = np.random.uniform(0,1)

chain[0] = 2

init = stateTrans[2]

for step in steps:

x = np.random.uniform(0,1)

if (compareLists(init,stateTrans[0])):

chain[step] = 0

if x <= 1:

nextState = stateTrans[0]

if (compareLists(init,stateTrans[1])):

chain[step] = 1

if x <= 0.3:

nextState = stateTrans[0]

else:

nextState = stateTrans[2]

if (compareLists(init,stateTrans[2])):

chain[step] = 2

if x <= 0.5:

nextState = stateTrans[1]

else:

nextState = stateTrans[3]

if (compareLists(init,stateTrans[3])):

chain[step]=3

if x <= 0.6:

nextState = stateTrans[2]

else:

nextState = stateTrans[4]

if (compareLists(init,stateTrans[4])):

chain[step] = 4

if x <= 1:

nextState = stateTrans[4]

init = nextState

return chain, steps

def graphChains(chain,steps):

plt.plot(steps,chain,'r:')

plt.plot(steps,chain,'x')

plt.title("A Sample of Drunkard's Walk")

plt.ylabel("State")

plt.xlabel("Step Number")

plt.ylim(-1,5)

plt.xticks(np.arange(0, 16, 1))

plt.show()

stateTrans = np.array([[1,0,0,0,0],[0.3,0,0.7,0,0],[0,0.5,0,0.5,0],[0,0,0.6,0,0.4],[0,0,0,0,1]])

init=[0,0,1,0,0]

absorb\_4 = 0

absorb\_0 = 0

for i in range(10000):

chain = [0]\*15

chain, steps = generateChain(init, stateTrans, chain)

steps.insert(0,0)

if chain[14] == 4:

absorb\_4+=1

if chain[14] == 0:

absorb\_0+=1

chance\_0 = absorb\_0/10000

chance\_4 = absorb\_4/10000

print("Absorbtion Probabilities for state 0:",chance\_0)

print("Absorbtion Probabilities for state 4:",chance\_4)