**PROJECT #5 – Markov Chains & Discrete Events**

**Divyanshu Sahay  
USC ID: 6364938199  
(dsahay@usc.edu)**

Important Note:

Tool Used: **Python 3.7.4.**  
Editor Used: Visual Studio Code.

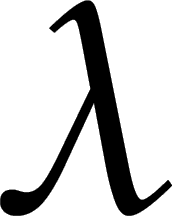
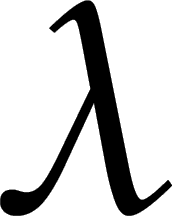
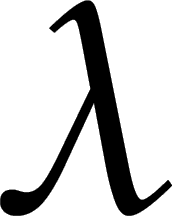
1. **Discrete Event Simulation – Non-Homogenous Poisson Process**

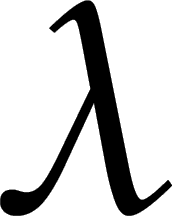
**Problem Statement:** A single server queue system is given where jobs (packets) arrive as a non-homogenous Poisson process. The initial rate of arrival is 4 jobs per second and which increases to 19 jobs per second and reduces back to 4 jobs per second. This happens linearly,  
 The service time provided by the server is distributed exponentially with the rate being 25 jobs per second. The server goes on a break if no jobs to service. The wait time has to be distributed uniformly on the interval (0,0.3).

* The total wait time in 100 hour of operation has to be calculated.

**Understanding the problem statement:**

The problem required simulating a non-homogenous arrival event. This can be done by using a thinning algorithm as following:

1. Initialize .
2. Generate ***U1 Unif(0,1)***
3. ******Set ***t = t – 1 / log U1***
4. Generate ***U2 Unif(0,1)*** independent of ***U1***
5. If ***U2 ≤ *** */ * then deliver ***t.***
6. Goto Step 1.

Here,  is selected as the maximum value of the rate i.e. 19 jobs per second. The problem needs handling of various variables to deal with different scenarios. The scenarios can be best if explained in comments while writing the program. The program written is a direct adaptation of the procedure provided by Professor Brandon in his notes.

**Program Written:**

import numpy as np

import random

import math

t = 0       #time variable

Na = 0      #number of arrival

Nd = 0      #number of departure

n = 0       #total jobs in the system

ta = 0      #time of arrival

i = 0

wait1 = 0

wait\_total1 = 0

wait2 = 0

wait\_total2 = 0

td = math.inf   #time of departure

T\_total = 100

Tp = 0

Ts = np.zeros(T\_total\*T\_total)

lamda\_max = 19  #maximum arrival rate

lamda = [4,7,10,13,16,19,16,13,10,7,4]  #list of rate of arrivals

while(Ts[i]<T\_total):

    if(ta <= td and ta < T\_total): #customer arrives and the queue is open for  customers to arrive

        t = ta

        Na += 1

        n += 1

        while(t<T\_total):

            u1 = np.random.rand()

            t = Ts[i] - np.log(u1)/lamda\_max

            u2 = np.random.rand()

            if (u2 <= lamda[int(np.mod(Ts[i],11))]/lamda\_max):

                ta = t

                break

        if(n==1):

            Y = random.expovariate(25)

            td = t + Y

    if(td < ta and td <= T\_total): #customer departs and the queue is open for  customers to arrive

        t = td

        n = n - 1

        Nd = Nd + 1

        if(n == 0): #if there is no one in the queue then the server goes to   
 sleep

            td = math.inf

            wait1 = random.uniform(0, 0.3)

            t = t + wait1 #the time has to be advanced by wait time

            wait\_total1 = wait\_total1 + wait1

        else:

            Y = random.expovariate(25)

            td = t + Y

    if(min(ta,td)>T\_total and n>0): #the queue is closed the already existing  customers remains queued

        t = td

        n = n-1

        Nd = Nd + 1

        if n==0:

            Y = random.expovariate(25)

            td = t + Y

            wait2 = random.uniform(0, 0.3)

            t = t + wait2

            wait\_total2 = wait\_total2 + wait2

    if(min(ta,td)>T\_total and n==0): #the queue is closed and there is no one in the queue, no new addition is alowed.

        Tp = max(t-T\_total,0)

    Ts[i+1] = t

    i += 1

print('server on break')

print(wait\_total1 + wait\_total2)

**Results and Observation:**

**Output Screen:**

**D:\SHIVAM\USC_STUDY\EE511\project5\p2.JPG**

The simulation was run several times and different results were obtained every time. The result shown here shows the maximum duration of the break obtained in several simulation attempts. So, the server went on break for a maximum of 20 hours.

1. **Head-of-List (HOL) blocking switch simulation:**

**Problem Statement:**  A 2X2 HOL switch has to be simulated where the inputs have a buffer to store the incoming packets. The main requirement of the problem is to simulate the system for multiple probability of arrival of the packets. Also, the probability of arrival is (***p***) which has to be changed for each simulation. The probability that a packet arriving at input (***i***)is switched to output (***j***)is given by (***rij***). All the packets in the buffer list should be serviced. Following are the requirements:

1. Assuming ***rij*** = 0.5 the distribution of mean of number of packets at the two buffers has to be plotted as a function of the arrival probability.
2. The mean of number of packets processed should be plotted as a function of the arrival probability.
3. A 95% confidence interval for the efficiency of the switch has to be estimated.
4. All the above has to be repeated for ***r1*** = 0.75 and ***r2*** = 0.25

**Understanding the problem statement:**

This Problem demands a careful evaluation of all the conditions of packet delivery to the output of a HOL switch for multiple probabilities of arrival and delivery. The program written explains in detail about the approach adopted to design the HOL switch.

The efficiency is given by the following expression:

Furthermore, the 95% confidence interval can be calculated using following expression:

D:\SHIVAM\USC_STUDY\EE511\project5\eq.JPG

0.95 =

Where, *σ* is the standard deviation and x̅ is the sample mean.

**Program Written:**

import random

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

import statistics

import math

p = 0 #probability of arrival, to be varied over time

r1 = 0.75 #probability that the a packet is switched to 1st port

r2 = 0.25 #probability that the a packet is switched to 1st port

T\_Total = 100 #number of times repeating the simulation

outlist1 = []

outlist2 = []

avg\_list = []

packet\_process\_list = []

efficiency\_list = []

packet\_process\_list\_mean = []

prob\_arrival = []

index1 = 0

index2 = 0

out\_1\_to\_2 = []

out\_1\_to\_1 = []

out\_2\_to\_2 = []

out\_2\_to\_1 = []

for j in range(T\_Total):

    p = np.random.rand()

    prob\_arrival.append(p)

    bufferlist1 = [] #input buffer for 1st input

    bufferlist2 = [] #input buffer for 2nd input

    index1 = 0

    index2 = 0

    packet\_process = 0

    prob1 = np.random.rand(T\_Total)

    prob2 = np.random.rand(T\_Total)

    for i in range(T\_Total):

        # ------- ANALYSING PACKET ARRIVAL AT INPUT 1------------------

        #checking if in the previous iteration the packet was not transferred from input 1

        #print('aya\_01')

        if(prob1[i] > p): #checking if the packet has arrived at input 1

            bufferlist1.insert(i,1) #updatinig the bufferlist if arrived

            transfer1 = np.random.rand()

            if(transfer1 < r1): #checking where the packet wants to go

                out\_1\_to\_2.insert(i,1)  #packet from 1 wants to go to 2

                out\_1\_to\_1.insert(i,0)

            else:

                out\_1\_to\_1.insert(i,1)  #packet from 1 wants to go to 1

                out\_1\_to\_2.insert(i,0)

        else:

            bufferlist1.insert(i,0) #updatinig the bufferlist if not arrived

            out\_1\_to\_1.insert(i,0)

            out\_1\_to\_2.insert(i,0)

        #----------- DOING THE SAME FOR INPUT 2 ---------------------

        #checking if in the previous iteration the packet was not transferred from input 2

        if(prob2[i] > p): #checking if the packet has arrived at input 1

            bufferlist2.insert(i,1) #updatinig the bufferlist if arrived

            transfer2 = np.random.rand()

            if(transfer2 < r2): #checking where the packet wants to go

                out\_2\_to\_1.insert(i,1)  #packet from 1 wants to go to 2

                out\_2\_to\_2.insert(i,0)

            else:

                out\_2\_to\_2.insert(i,1)  #packet from 1 wants to go to 1

                out\_2\_to\_1.insert(i,0)

        else:

            bufferlist2.insert(i,0) #updatinig the bufferlist if not arrived

            out\_2\_to\_1.insert(i,0)

            out\_2\_to\_2.insert(i,0)

    #print("aaa")

    while(index1 != T\_Total and index2 != T\_Total):

        #condition where both at 1 and 2 wishes to go to 2

        if bufferlist1[index1] == 0 and bufferlist2[index2] == 0:

            index1+=1

            index2+=1

        elif bufferlist1[index1] == 0 and bufferlist2[index2] == 1:

            if  (out\_2\_to\_1[index2] == 1):

                packet\_process +=1

                outlist1.append(1)

            elif (out\_2\_to\_1[index2] == 1):

                packet\_process +=1

                outlist2.append(1)

            index2 +=1

            index1 +=1

        elif bufferlist1[index1] == 1 and bufferlist2[index2] == 0:

            if  (out\_1\_to\_2[index1] == 1):

                packet\_process +=1

                outlist2.append(1)

            elif (out\_1\_to\_1[index1] == 1):

                packet\_process +=1

                outlist1.append(1)

            index2 +=1

            index1 +=1

        elif bufferlist1[index1] == 1 and bufferlist2[index2] == 1:

            if (out\_1\_to\_2[index1] == 1 and out\_2\_to\_2[index2] == 1):

                transfer3 = np.random.rand()

                if(transfer3 >= 0.5): #randomly decide which should go

                    index1 += 1

                    outlist2.append(1)

                else:

                    index2 += 1

                    outlist2.append(1)

                packet\_process += 1

            #condition where both at 1 and 2 wishes to go to 1

            elif (out\_1\_to\_1[index1] == 1 and out\_2\_to\_1[index2] == 1):  #condition where both wishes to go to same output port

                transfer4 = np.random.rand()

                if(transfer4 >= 0.5): #randomly decide which should go

                    index1 += 1

                    outlist1.append(1)

                else:

                    index2 += 1

                    outlist1.append(1)

                packet\_process += 1

            #condition where there is no conflict

            elif  (out\_1\_to\_2[index1] == 1 and out\_2\_to\_1[index2] == 1):

                index1 +=1

                index2 +=1

                packet\_process +=2

                outlist1.append(1)

                outlist2.append(1)

            #condition where there is no conflict

            elif (out\_1\_to\_1[index1] == 1 and out\_2\_to\_2[index2] == 1):

                index1 +=1

                index2 +=1

                packet\_process +=2

                outlist2.append(1)

                outlist1.append(1)

    #till here one of the list is emptied

    while(index1 != T\_Total): #checking if all list1 elements are tranferred

        if  (out\_1\_to\_2[index1] == 1):

            packet\_process +=1

            outlist2.append(1)

        elif (out\_1\_to\_1[index1] == 1):

            packet\_process +=1

            outlist1.append(1)

        index1 +=1

    while(index2 != T\_Total): #checking if all list1 elements are tranferred

        if  (out\_2\_to\_1[index2] == 1):

            packet\_process +=1

            outlist1.append(1)

        elif (out\_2\_to\_1[index2] == 1):

            packet\_process +=1

            outlist2.append(1)

        index2 +=1

    # print(packet\_process)

    packet\_process\_list.append(packet\_process)

    packet\_process\_list\_mean.append(packet\_process/T\_Total)

    efficiency = float(1/max(packet\_process\_list\_mean))

    efficiency\_list.append(efficiency)

    buff1 = sum(bufferlist1)

    buff2 = sum(bufferlist2)

    avg = (buff1 + buff2)/2

    avg\_list.append(avg)

    efficiency = max(packet\_process\_list\_mean)

# print(packet\_process\_list)

li = [i for i in range(1,T\_Total+1)]

fig0 = plt.figure()

plt.plot(li,prob\_arrival)

plt.show()

fig1 = plt.figure()

plt.bar(li,avg\_list)

plt.plot(prob\_arrival, 'r-')

plt.show()

fig2 = plt.figure()

plt.bar(li,packet\_process\_list\_mean)

plt.plot(prob\_arrival, 'r-')

plt.show()

#------- 95% CI -------

print(efficiency\_list)

eff\_sum = sum(efficiency\_list)

eff\_avg = eff\_sum / T\_Total

std\_dev = statistics.stdev(efficiency\_list)

moe = 1.96\*(std\_dev/math.sqrt(T\_Total))

CI\_l = eff\_avg - moe

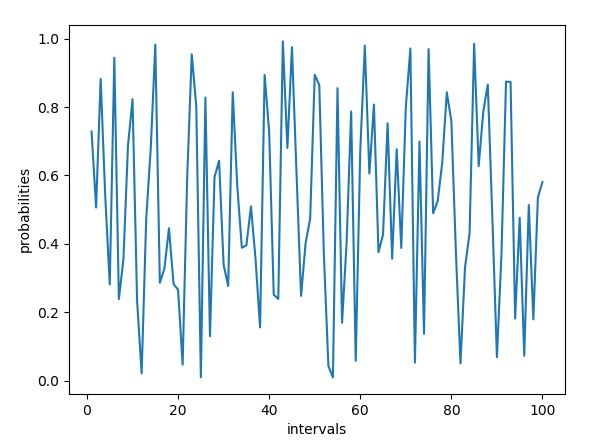
CI\_u = eff\_avg + moe

print(CI\_l)

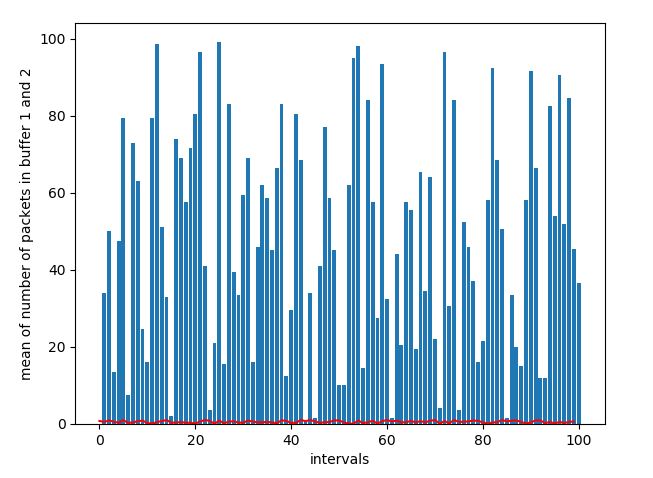
print(CI\_u)

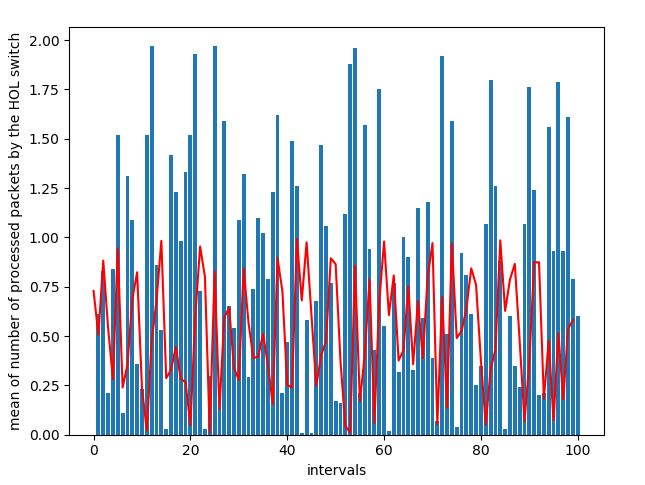
**Results and Observation:**

***rij*** = 0.5: Equal probability of going to either of the outputs.

  
**Fig.1. Distribution of Probabilities**

This is the distribution of probability with time. It can be observed that it is highly varied. This is the reason why the output observed were also observed considerably varied.

  
**Fig.2. The mean of number of packets plotted against probability (in red)**

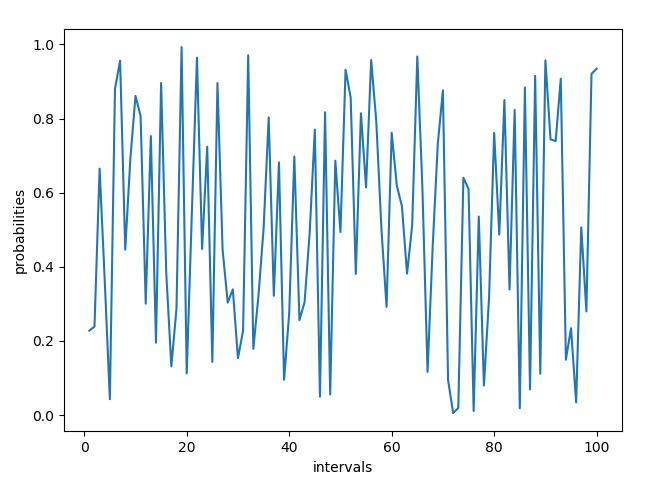
  
**Fig.3.The mean number of packet processed with the probability density (in red)**

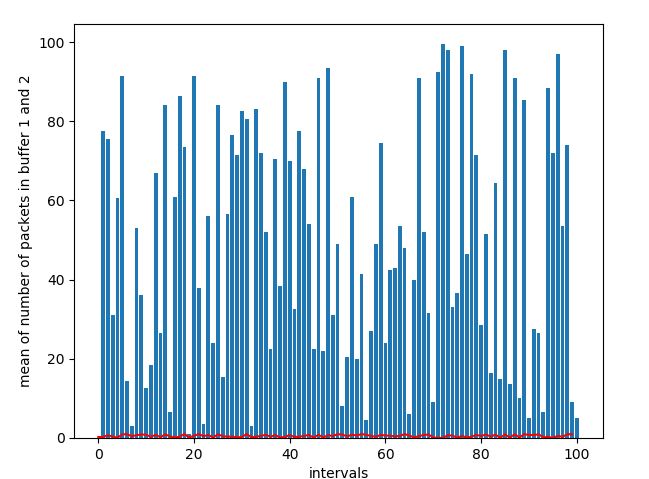
**Output Screen:**

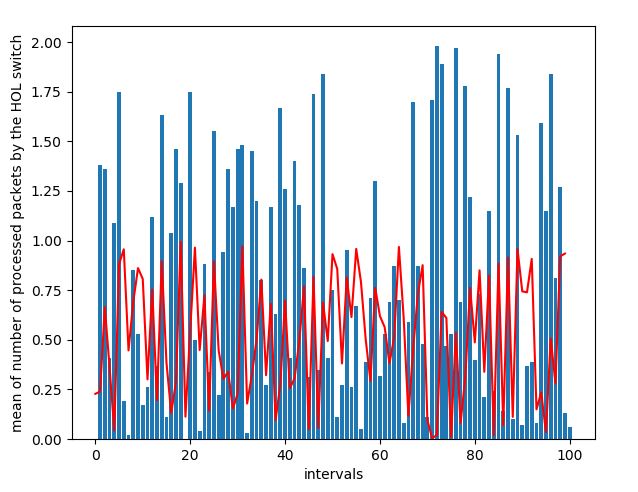
0.5178805146737651

0.5825672281162759

This is the 95% confidence interval obtained for the efficiency. The same is repeated for the values for ***r1*** = 0.75 and ***r2*** = 0.25.

**  
Fig. 4. Distribution of Probabilities (2nd Case)**

**  
Fig.5 The mean of number of packets plotted against probability (in red)**

**  
Fig. 6. The mean number of packet processed with the probability density (in red)**

**Output Screen:**

**95% CI:**

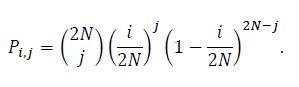
0.5427655615508811

0.5604328825171281

1. **The Wright-Fisher model Simulation: Markov-Chains**

**Problem Statement:**

It is given that a gene has two alleles i.e. A1 and A2. A population of N diploids are having 2N copies of these genes (each having two copies). The representation of state vector indicating the distribution these two allele at a given time is provided in the question. Also, it is mentioned that the density of genes evolves according to a binomial probability distribution, i.e. the state transition matrix is populated using binomial theorem as follows:



The Population genetic drift has to be simulated. Also, an explanation about the steady state genetic composition has to be made.

**Understanding the Problem:**

The problem is going to use a Markov Chain to obtain the subsequent states. Again, the source code provided by Porf. Brandon, were used directly with appropriate changes.

**Program Written** (a direct adaptation of the program provided by Professor)**:**

import numpy as np

from scipy.special import comb

N = 100 # number of individuals

a = np.array([0]\*(2\*N)) # initial distribution

input = np.insert(a,49,1)

print(input)

# transition matrix

P = np.zeros((2\*N+1, 2\*N+1))

for i in range(0,2\*N+1):

    for j in range(0,2\*N+1):

        P[i,j] = comb(2\*N,j, exact=True, repetition=False)\*(((i)/(2\*N))\*\*(j))\*((1-(i)/(2\*N))\*\*(2\*N-j))

#print(P)

n = 1600 # number of steps to take

output = np.zeros((n+1,2\*N+1)) # clear out any old values

t = np.arange(0,n)

output[0] = input  #generate first output value

for i in range(1,n):

    output[i,:] = np.dot(output[i-1,:],P)

    if np.allclose(output[i,:],output[i-1,:]):

        print('Convergence after '+str(i),' iterations')

        break

**Results and Observation:**

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

---- initial condition: 50% of each type in the population---

Convergence after 1643  iterations

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]

---- initial condition: at 199th location : 2N-1 of A1 and 1 of A2---

Convergence after 1421  iterations

It was observed that if we start with (50/50), i.e., each type of gene is equal in number, it will take us 1643 iterations to obtain convergence. It can be observed that it took a large amount of iterations for these pattern to converge. It seems to disobey the Perron-Frobenious and Makov Chain Ergodic Theorem as different initial conditions is resulting in different long term behaviour.