Hrishikesh Hippalgaonkar

[hippalga@usc.edu](mailto:hippalga@usc.edu)

**EE 511 Simulation Methods for Stochastic Systems**

**Project #2**

1. **Problem 1**

**[Networking]**

**Given n = 50 people in a social network. Suppose any given unordered pair of two people are connected at random and independently with probability p**

1. **Generate and plot three network samples for each value of p in {0.02, 0.09, 0.12}**

**Briefly discuss the structure of these sample graphs.**

1. **Count the number of connections each vertex or node on a sample from the previous question. This statistic is the *degree* of the vertex. Plot a histogram of the vertex degrees for each of your sample graphs.**
2. **Vertex degree is supposedly binomially distributed for small network sizes. Generate a network with (n, p) = (100, 0.6) and plot a histogram of the vertex degree. Does your histogram support the binomial distribution assumption?**

**Approach:** Using the inverse transform method, generate a Bernoulli trial for every pair of nodes in the network. This can be implemented in the following way:

1. Generate uniform random variable from In python, this can be done using the *random.random* command.
2. If  *,* then ; else

, which means that edge doesn’t exist; in this case *p* will {0.02, 0.09, 0.12}.

In a network of n = 50 people, the total number of possible edges is equal to

**N =**

And using the inverse transform method we must generate 1225 Bernoulli trials and determine whether edge exist between pair of nodes

**Code:**

import numpy as np

import random as rand

import matplotlib.pyplot as plt

import networkx as nx

def bernoulliTrialsForEdges(n, p):

edgeExist = [] # Empty list

G = nx.Graph()

for i in range(1, n):

for j in range(i + 1, n + 1):

# Return random floats in the half-open interval [0.0, 1.0).

# Results are from the “continuous uniform” distribution over the stated interval

x = rand.random()

if(x <= p):

result = 1

edgeExist.append([i, j, result])

G.add\_edge(i, j)

else:

result = 0

edgeExist.append([i, j, result])

G.add\_node(i)

G.add\_node(j)

return G

def main():

G = nx.Graph()

n = 100 # Number of people in the network

p = 0.06 # Probability of edge existing between two nodes

degree = np.empty([n + 1])

G = bernoulliTrialsForEdges(n, p)

for i in range(1, n + 1):

degree[i] = G.degree[i]

print("Total # of edges",G.number\_of\_edges())

plt.figure(1)

plt.hist(degree, bins ='auto', alpha = 0.9, hatch ='/')

plt.title("Histogram of degree of each node(p = 0.06)", alpha = 0.75)

plt.xlabel('# of edges each node has in the network')

plt.ylabel('Count')

plt.grid(True)

plt.show()

plt.figure(2)

plt.title("Graph for network of n = 50 , p = 0.02", alpha = 0.75)

options = {

'with\_labels': True,

'node\_color': 'red',

'node\_size': 150,

'width': 3

}

#nx.draw\_circular(G, \*\*options)

#nx.draw\_spectral(G, \*\*options)

nx.draw\_random(G, \*\*options)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Result:**

|  |
| --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphCircular_1.png |
| 1. Graph represented in a Circular structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphRandom_1.png |
| 1. Graph represented in a Random structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphRandom_11.png |
| 1. Graph represented in a Random structure |
| **Figure 1. Graphs for n = 50 and p = 0.02** |

It can be observed that since *p* is very small (p *= 0.02),* the number of edges is very low. Furthermore, it can be seen that a few nodes are not connected to any other node. After running the program a few times, it was observed that the total number of edges was around 20 – 40 which is very low as compared to 1225. But this is to be expected because the low probability.

|  |
| --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphCircular_2.png |
| 1. Graph represented in a Circular structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphRandom_2.png |
| 1. Graph represented in Random structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphSpectral_2.png |
| 1. Graph represented in Spectral structure |
| **Figure 2. Graphs for n = 50 and p = 0.09** |

In this case, the p is slightly greater than the case before. Thus, it can be observed that the number of edges in this case is greater than the previous case. Furthermore, almost all the nodes in the above graphs are connected to at least one other node. After running the program a few times, it was observed that the total number of edges was around 100 – 120 which is again very low as compared to 1225. But is to be expected because of the low probability.

|  |
| --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphCircular_3.png |
| 1. Graph represented in a Circular structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphRandom_3.png |
| 1. Graph represented in a Circular structure |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GraphSpectral_3.png |
| 1. Graph represented in a Circular structure |
| **Figure 3. Graphs for n = 50 and p = 0.12** |

In this case, the p is slightly greater than the case before. Thus, it can be observed that the number of edges in this case is greater than the previous case and the graph looks very dense. Furthermore, almost all the nodes in the above graphs are connected to at least one other node. After running the program a few times, it was observed that the total number of edges was around 130 – 170 which is again very low as compared to 1225. But this is to be expected because of the low probability.

|  |  |
| --- | --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Vertex_degree(0.02).png | C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Vertex_degree(0.09).png |
| 1. Histogram for degree of vertices for   n = 50 and p = 0.02 | 1. Histogram for degree of vertices for   n = 50 and p = 0.09 |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Vertex_degree(0.12).png | C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Vertex_degree100(0.06).png |
| 1. Histogram for degree of vertices for   n = 50 and p = 0.12 | 1. Histogram for degree of vertices for   n = 100 and p = 0.06 |
| **Figure 4. Histograms for degree of vertices** | |

Figure 4. (a), (b) and (c) represent histogram of the degree of the vertices of the graphs for different values of p. Figure (d) represents the histogram for degree of vertices of graph with n = 100 and p = 0.06

Vertex degree is supposedly binomial distribution and it can be seen in the above histograms too. Consider the histogram (d). This histogram can be approximated by a Gaussian curve with mean around 6. And, according to the Central Limit Theorem, binomial distribution can be approximated with a Gaussian curve with mean equal to *np =* 100\*0.06 = 6. Thus, the above histogram for degree of vertex can supports the binomial distribution assumption.

1. **Problem 2**

**[Counting successes]**

* **Use the inverse CDF method to generate 1000 independent samples, of the exponential random variable with an average waiting time of 0.2 time units. Evaluate the quality of your RNG with goodness of fit tests.**
* **Each exponential random sample represents the waiting time until an event occurs. Implement a routine to count the number of exponentially-distributed time intervals that occur in 1 time unit. Generate such counts for 1000 separate unit time intervals. How are these counts distributed? Justify your answer**

**Approach:** In the first part of the question, using the inverse transform method, we must generate samples from the exponential distribution. This can be implemented in the following way:

1. Generate uniform random variable from In python, this can be done using the *random.random* command.
2. The distribution function for an exponential random variable is given by,

Then, if we let ,

Hence, we can generate a random uniform variable U and then generate exponential random variable X by setting

Once the samples are generated, the quality of samples is determined using ‘*chi-square* *goodness of fit test’.* To test for **goodness of fit** means that we wish to test that a certain function is the distribution function of a distribution from which we have a sample

Then we test whether the **sample distribution function** defined by

*Sum of the relative frequencies of all sample values not exceeding x*

fits “sufficiently well.” If this is so, we shall accept the hypothesis that is the

distribution function of the population; if not, we shall reject the hypothesis.

The test is justified by the fact that if the hypothesis is true, then is an observed value of a random variable whose distribution function approaches that of the chi-square distribution with degrees of freedom (or degrees of freedom if *r*

parameters are estimated) as *n* approaches infinity. The requirement is that at least five

sample values lie in each interval

In the second part of the question, we have to count the number of exponentially-distributed waiting time intervals that occur in 1 unit time interval. This can be implemented by generating random exponentially distributed samples using the above method, and adding them until their addition is greater than 1

**Code: Question Part (a)**

import numpy as np

import random as rand

from math import \*

import matplotlib.pyplot as plt

from scipy.stats import expon

from scipy.stats import chisquare

def exponentialRNG(numSamples, avgTime):

# This function creates random numbers with exponential

# distribution using inverse transform method

# numSamples: # of samples to be generated

# avgTime: average waiting time(lambda parameter)

expSample = np.empty([numSamples])

for i in range(0, numSamples):

x = rand.random()

expSample[i] = (-1) \*avgTime \* log(1 - x)

return expSample

def main():

avgTime = 0.2 # avgTime: average waiting time(lambda parameter)

numSamples = 1000 # numSamples: # of samples to be generated

numBins = 10 # nunBins: This is the # of bins for the

bins = [0, 0.044, 0.1021, 0.1832, 0.3219, 1 ]

expSample = exponentialRNG(numSamples, avgTime)

e = expon.rvs(size = numSamples, scale = avgTime)

# Generates random numbers from an exponential continuous random variable.

plt.figure(1)

plt.hist(expSample, numBins, color = 'blue', hatch = '/')

plt.title("Histogram of 1000 samples of exponential random variables(using Inverse transform method)", alpha = 0.75)

plt.xlabel('Waiting Time')

plt.ylabel('# Samples')

plt.grid(True)

plt.savefig("Observed\_Histogram.png")

plt.show()

plt.figure(2)

plt.hist(e, numBins, color = 'red', hatch = '/')

plt.title("Histogram of 1000 samples of exponential random variables(using inbuilt function) ", alpha = 0.75)

plt.xlabel('Waiting Time')

plt.ylabel('# Samples')

plt.grid(True)

plt.savefig("Expected\_Histogram.png")

plt.show()

(observedValues, bins, patches) = plt.hist(expSample, bins, color = 'blue', hatch = '/')

print("Observed Values: ",observedValues,"\n")

(expectedValues, bins, patches) = plt.hist(e, bins, color = 'red', hatch = '/')

print("Expexted Values: ",expectedValues,"\n")

(chi) = chisquare(f\_obs = observedValues, f\_exp = expectedValues)

print(chi)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Question part (b)**

import numpy as np

import random as rand

from math import \*

import matplotlib.pyplot as plt

def exponentialRNG(numSamples, avgTime):

# This function creates random numbers with exponential

# distribution using inverse transform method

# numSamples: # of samples to be generated

# avgTime: average waiting time(lambda parameter)

expSample = np.empty([numSamples])

for i in range(0, numSamples):

x = rand.random()

expSample[i] = (-1) \*avgTime \* log(1 - x)

return expSample

def numTimeIntervals(numSamples, avgTime):

# This function counts the number of exponentially-distributed time

# intervals that occur in 1 time unit

# numSamples: # of unit time intervals

expSample = np.empty([numSamples])

actualSample = 1

while(actualSample <= numSamples - 1):

countTimeIntervals = 0

time = 0

while(time <= 1):

temp = exponentialRNG(1, avgTime)

time = time + temp

countTimeIntervals = countTimeIntervals+ 1

actualSample = actualSample + 1

expSample[actualSample-1] = countTimeIntervals

return expSample

def main():

avgTime = 0.2 # avgTime: average waiting time(lambda parameter)

numSamples = 1000 # numSamples: # of samples to be generated

expSample = numTimeIntervals(numSamples, avgTime)

plt.figure(1)

plt.hist(expSample, bins = 'auto', color = 'blue', hatch = '/')

plt.title("Histogram of counts of 1000 separate unit time intervals ", alpha = 0.75)

plt.xlabel('# of time intervals in 1 unit time interval')

plt.ylabel('Count')

plt.grid(True)

plt.savefig("Observed\_Histogram\_Intervals.png")

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Result:**

|  |  |
| --- | --- |
| **C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Observed_Histogram.png** | **C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Expected_Histogram.png** |
| 1. Histogram of exponential samples generated using inverse transform method | 1. Histogram of exponential samples generated using inbuilt *expon.rvs* function |
| **Figure 5. Histogram of the exponential samples** | |

It can be seen that the histogram of samples generated using inverse transform method and the histogram of samples generated using the inbuilt Python function for exponential PDF are the same. Thus, we can say that the samples generated using inverse transform method do belong to the exponential distribution.

The samples generated using the inverse transform method are then divided into five bins based on percentiles. The first bin contains the first 25% of the total samples, second bin contains next 25% samples and so on.

|  |
| --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\ChiSquare1.jpg |
| 1. Chi – Square output |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\ChiSquare2.jpg |
| 1. Chi – Square output |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\ChiSquare3.jpg |
| 1. Chi- Square output |
| **Figure 6. Chi- Square Output** |

|  |
| --- |
|  |
| **Figure 7. Chi -Square table** |

From figure 6, it can be observed that the chi-square values for the samples we generated are between 2 and 7, and from the table in figure 7, it can be observed that the chi – square values that we obtain are is less than the threshold value 9.49 , which is the chi-square value for 95% confidence interval and degree of freedom 4 (since number of bins is m = 5, then degree of freedom = m – 1)

Thus, we can accept our hypothesis with a 95% confidence level that the samples we generated are from the exponential distribution

|  |
| --- |
| C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Observed_Histogram_Intervals.png |
| **Figure 8. Histogram of number of exponential time intervals in 1 unit time** |

The number of time intervals in a unit time is given by Poisson distribution. Poisson distribution expresses the probability of a given number of events occurring in a fixed interval of time with a known constant rate and independently of the time since the last event. The above histogram can be approximated by a Poisson distribution and this justifies the assumption that the number of time intervals follow a Poisson distribution.

* **Generating Discrete Random Variables:**
* **The Rejection Method:**

Suppose we wish to generate a continuous probability density function . The basic idea is we must find another probability density function such that this function is *close* to . We assume that the ratio is bounded by a constant *c* given by . Thus, we can generate samples from by generating from and then accepting this generated sample with a probability proportional to )

1. **Problem 3**

**[Double Rejection]**

**The random variable X has a bimodal distribution made up of an equally weighted, convex summation of a beta and triangle distribution:**

**Implement rejection sampling routines for X. Generate 1000 samples of the random variable using each envelope. Track the rejection rate of your rejection sampling RNGs. The rejection rate is the average number of rejected candidates per sample. This is a measure of the efficiency of your RNG**

**Approach:**

1. Generate from the uniform distribution where *a = 0* and *b = 6*
2. Find . For the above problem, c = 1.4625
3. Sample from the uniform distribution
4. Accept the sample if

**Code:**

import numpy as np

import random as rand

import matplotlib.pyplot as plt

def acceptReject(numSamples):

expSample = np.empty([numSamples])

rejectCount = 0

actualSamples = 0

acceptSamples = 1

while(acceptSamples <= numSamples):

c = 1.4625

flag = 1

while(flag == 1):

actualSamples = actualSamples + 1

u1 = -6 \* rand.random() + 6

u2 = rand.random()

if(u1 <= 1):

f = 396 \* pow((1 - u1), 4) \* pow(u1, 7)

elif(u1 > 4 and u1 <= 5):

f = 0.5 \* (u1 - 4)

elif(u1 > 5 and u1 <=6):

f = -0.5 \* (u1 - 6)

else:

f = 0

if(u2 <= (f/c)):

expSample[acceptSamples - 1] = u1

flag = 0

acceptSamples = acceptSamples + 1

else:

flag = 1

rejectCount = rejectCount + 1

rejectionRate = (rejectCount/actualSamples) \* 100

print("\nTotal # of Samples: ", actualSamples)

print("\n# of Rejected Samples: ", rejectCount)

print("# of Accepted Samples: ", numSamples)

print("\nRejection Rate: ", rejectionRate, " %")

return (expSample)

def main():

numSamples = 1000

expSample = acceptReject(numSamples)

plt.figure(1)

plt.hist(expSample, bins = 'auto', hatch = '/')

plt.title("Histogram of samples of the bimodal random variable", alpha = 0.75)

plt.xlabel('x ->')

plt.ylabel('# of Samples')

plt.grid(True)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Result:**

|  |
| --- |
| **C:\Users\Hrishikesh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Rejection_Rate_Histogram.png** |
| 1. Histogram of the bimodal random variable |
|  |
| 1. Rejection rate for the samples generated |
| **Figure 9. Samples generated for the bimodal distribution using rejection method** |

As we can observe, the rejection rate is almost 90%, which is quite inefficient. For generating just 1000 samples belonging to the distribution, around 13500 samples were rejected. Again, this kind of rejection ratio was to be expected because we don’t have any leeway in this problem for changing the value of ‘c’. With trial and error, we can choose a value of ‘c’ such that it not only gives samples in the desired distribution, but also reduces the rejection ratio.