**PROJECT #4 – INTEGRALS & INTERVALS**

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

Important Note:

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

1. **Approximate Integrals using Monte Carlo Simulation:**

**Problem Statement:** It is asked to calculate following definite integrals using the Monte Carlo Simulation Method:

**Understanding the problem statement:**

The problem asks to evaluate the definite integrals using Monte Carlo Simulation. This relies on the Law of Large numbers to approximate the Population Mean. Let’s try to understand in very simple terms the process to be carried out. The average value of a function, can be written as follows:



The ( ) term will be the sample mean for samples drawn from the [a,b]. However, from Law of Large Numbers we know, for large N it will approach the population mean. Hence, this approximation can help us find the definite integral given in Q.1.

The aforementioned approach will work for proper integral. However, things become more interesting for improper integrals, as given in problem 2 and 3. Let’s get going:

Say,

So we can write,

**(i)**

Now, we can do a substitution for *x* and *y* using spherical coordinates, as follows,

Substituting this in eq.(i) we get,

Now this becomes simpler to work with. We do following substitution,

We get following,

This can be solved with the similar procedure, as we did for 1st. Finally, the last question can also be solved using the same approach. We can write,



**Program Written (1a)**:

import random

import math

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

a = -2

b = 2

N  = 1000

result = 0

law = 0

final\_result = []

def func(X):

        return math.exp(X + X\*X) #THE ACTUAL FUNCTION

for i in range(N):

    total\_samples = np.zeros(N)

    law = 0

    for i in range (len(total\_samples)):

        total\_samples[i] = random.uniform(a,b) #GENERATING RANDOM VALUES BETWEEN -2 and 2

    for i in range(N):

        law = law + func(total\_samples[i])

    result = (b-a)\*(law/N)

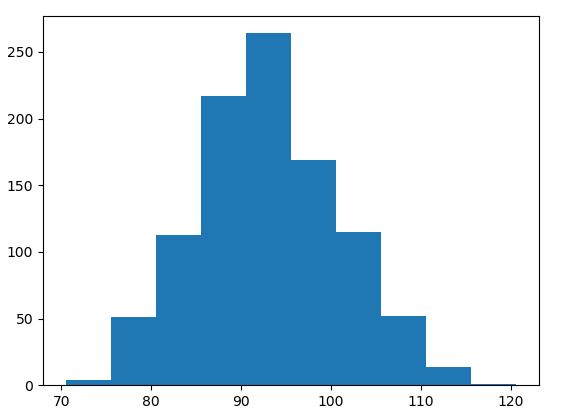
    final\_result.append(result)

fig = plt.figure()

plt.hist(final\_result)

plt.show()

**Results and Observation (1a)**:

  
Fig.1: Histogram showing results for Q1 (**X-axis: Result Obtained Y-axis: Frequency**)

It can be observed in the code that the experiment was repeated was repeated for 1000 samples. The theoretical value of the integral was calculated using an online calculator and it was observed as 93.16. It can be seen in the plot the mean value is close to 93. Hence it can be concluded that the Monte Carlo method for integration is sufficiently efficient.

**Program Written (1b)**:

a = 1

b = 0

N  = 1000

result = 0

law = 0

final\_result = []

def func(X):

    k1 = -((1-X)/X)\*((1-X)/X)

    k2 = math.exp(k1)\*(X-1)

    k3 = k2/(X\*X\*X)

    return k3

for i in range(N):

    total\_samples = np.zeros(N)

    law = 0

    for i in range (len(total\_samples)):

        total\_samples[i] = random.uniform(a,b)

     for i in range(N):

        law = law + func(total\_samples[i])

    result = (b-a)\*(law/N)

    final\_result.append(math.sqrt(2\*math.pi\*result))

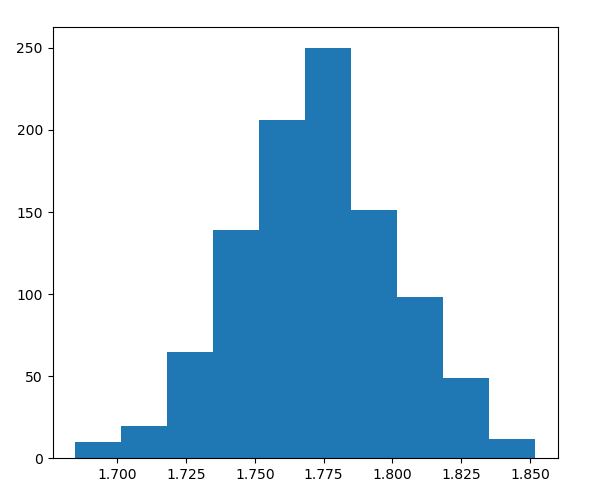
fig = plt.figure()

plt.hist(final\_result)

plt.show()

It is evident that the code is almost the same. Only difference is the function. The approach to obtain this function is explained above. As the function was complex it had to be divided into k1, k2 and k3.

**Results and Observation (1b)**:

  
Fig.2: Histogram showing results for Q2 (**X-axis: Result Obtained Y-axis: Frequency**)

The theoretical value of the integral was calculated using an online calculator and it was observed as 1.74 i.e. . It can be seen in the plot the mean value is close to 1.74.

**Program Written (1c)**:

import random

import math

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

a = 0

b = 1

N  = 1000

result = 0

law = 0

final\_result = []

def func(X,Y):

    k = (X+Y)\*(X+Y)

    p = - k

    return math.exp(p)

for i in range(N):

    total\_samplesX = np.zeros(N)

    total\_samplesY = np.zeros(N)

    law = 0

    for i in range (len(total\_samplesX)):

        total\_samplesX[i] = random.uniform(a,b)

        total\_samplesY[i] = random.uniform(a,b)

    for i in range(N):

        law = law + func(total\_samplesX[i],total\_samplesY[i])

    result = (b-a)\*(b-a)\*(law/N)

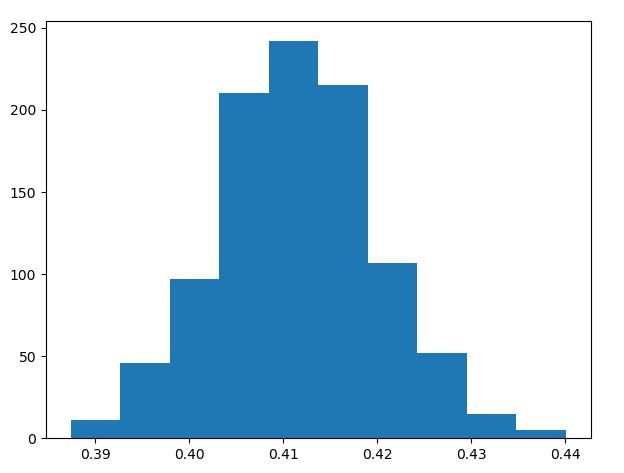
    final\_result.append(result)

fig = plt.figure()

plt.hist(final\_result)

plt.show()

**Results and Observation (1c)**:

  
Fig.3: Histogram showing results for Q3 (**X-axis: Result Obtained Y-axis: Frequency**)

The theoretical value of the integral was calculated using an online calculator and it was observed as 0.411. It can be seen in the plot the mean value is close to 0.41.

1. **Empirical and Theoretical Distribution:**

**Problem Statement:** A random variable ***X*** is given as where is Normally Distributed. ***X***  is Chi-Square random variable (i.e. X ~ ). Following are the requirement:

* Plot Empirical Distribution and overlay Theoretical Distribution () for 10 samples.
* Estimate Lower Bound
* Find 25th, 50th and 90th percentiles for empirical and theoretical distribution
* Repeating for 100 and 1000 samples.

**Understanding the problem statement:**

We had been generating samples given the distribution of a random variable. In this problem we need to plot the distribution. We can generate **empirical distribution function**which is simply a step function that jumps up by 1/n at each of the n samples. The empirical distribution function for samples {x1, x2, ⋯, xn} is defined by:

where n is the number of samples and x is just a dummy variable of the function. According to Glivenko-Cantelli theorem, the empirical distribution function  converges to the cumulative distribution function *(cdf)*  with probability one. That is, the estimation gets better and better as we increase the number of samples n.

We can use ***stats.chi2.cdf****,* the inbuilt function in Python to create a theoretical distribution. The samples are taken as N. This is varied from 10, 100 and 1000.

**Program Written**:

import random

import numpy as np

import matplotlib.pyplot as plt

import scipy.stats as stats

from statsmodels.distributions.empirical\_distribution import ECDF

X\_samples = []

N= 10

mu = 0

sigma  = 1

for i in range(N):

    z1 = np.random.normal(mu, sigma)

    z2 = np.random.normal(mu, sigma)

    z3 = np.random.normal(mu, sigma)

    z4 = np.random.normal(mu, sigma)

    X = z1\*z1 + z2\*z2 + z3\*z3 + z4\*z4

    X\_samples.append(X)

X\_samples.sort()

print(len(X\_samples))

print(X\_samples)

y = np.arange(0,1,1/N)

x = np.arange(0,12,10\*\*(-4))

plt.plot(x, stats.chi2.cdf(x, df=4))

plt.step(X\_samples,y,label='Empirical')

plt.show()

diff = 0

max\_diff = 0

print('-- the difference---')

for i in range(len(X\_samples)):

    diff = abs(i\*(1/N) - stats.chi2.cdf(X\_samples[i], df=4))

    if(diff > max\_diff):

        max\_diff = diff

print(max\_diff)

print('----25th percentile : Empirical ----')

print(np.percentile(X\_samples,25))

print('----50th percentile : Empirical----')

print(np.percentile(X\_samples,50))

print('----90th percentile : Empirical----')

print(np.percentile(X\_samples,90))

print('----25th percentile : Theoretical ----')

print(np.percentile(x,25))

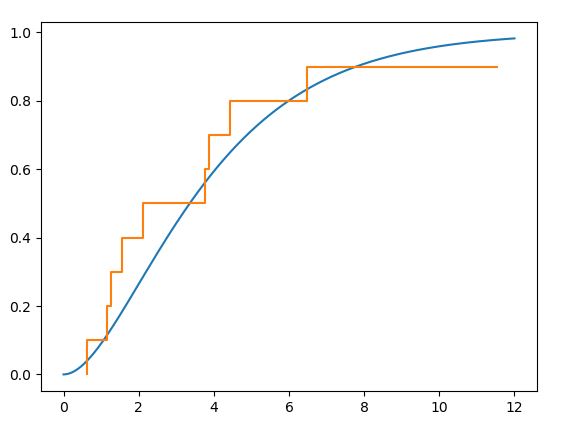
print('----50th percentile : Theoretical ----')

print(np.percentile(x,50))

print('----90th percentile : Theoretical ----')

print(np.percentile(x,90))

**Results and Observation**:

  
Fig.4: Empirical Distribution & Theoretical Distribution: N = 10 **(X-axis: Samples Y-axis: CDF)**

It can be seen that the theoretical distribution when overlayed with the empirical distribution, does not fit perfectly. This is in congruence with our expectation that, with lesser samples the estimation is not that good.

**Output Screen**

-- the maximum difference---

0.13635384707458237

----25th percentile : Empirical ----

1.8409978176756092

----50th percentile : Empirical----

2.6602375170577384

----90th percentile : Empirical----

7.290258576264269

----25th percentile : Theoretical ----

2.999975

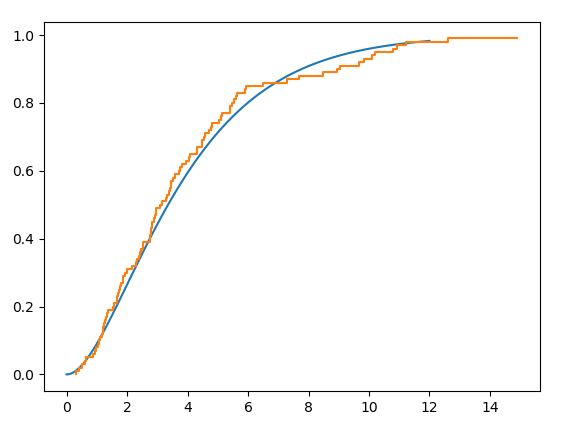
----50th percentile : Theoretical ----

5.99995

----90th percentile : Theoretical ----

10.79991

The maximum difference obtained is the largest of all the. Additionally, the percentile values are also considerable different. All this is the indicator that the data samples taken are not enough to give a good estimate of the theoretical distribution.

  
Fig.5: Empirical Distribution & Theoretical Distribution: N = 100 **(X-axis: Samples Y-axis: CDF)**

**Output Screen:**

-- the maximum difference---

0.0488149810027666

----25th percentile : Empirical ----

1.8569223549374586

----50th percentile : Empirical----

3.1348390170081943

----90th percentile : Empirical----

8.945884200306843

----25th percentile : Theoretical ----

2.999975

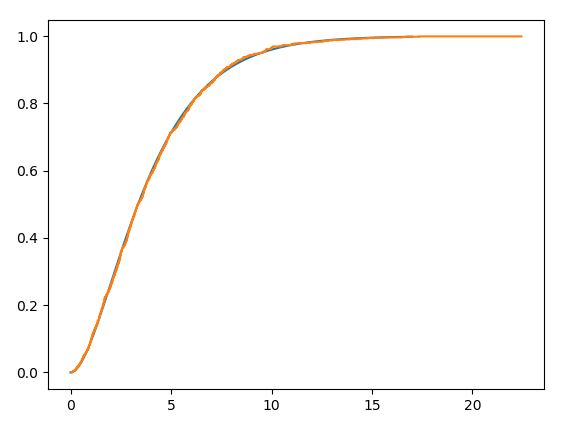
----50th percentile : Theoretical ----

5.99995

----90th percentile : Theoretical ----

10.79991

As the samples are increased to 100 it is clearly seen that the fitting has become better. Similarly, the percentile values have also shown some resemblance with the theoretical values.

****  
Fig.6: Empirical Distribution & Theoretical Distribution: N = 1000 **(X-axis: Samples Y-axis: CDF)**

Finally, when the sample values are increased to 1000. We get a perfect fit. It can also be seen that the maximum difference is reduced considerably. The results for the percentiles produced below were obtained after several iterations. The range of number is different from previous numbers because I carried out different iterations with different axes parameters to attain best possible match. However, it can be confidently concluded that that as the number of samples increases, the estimation becomes as good as the theoretical distribution.

**Output Screen**

-- the maximum difference---

0.026543263066655637

----25th percentile : Empirical ----

3.596625601874808

----50th percentile : Empirical----

6.762928728489089

----90th percentile : Empirical----

12.546158728055048

----25th percentile : Theoretical ----

3.749975

----50th percentile : Theoretical ----

7.49995

----90th percentile : Theoretical ----

13.499910000000002

1. **Statistical and Bootstrap Confidence Interval:**

**Problem Statement:** In this problem a data file has been provided which contains the waiting times and the duration for eruption of a geyser. Following are the requirements:

* For 15 samples compute statistical and bootstrap confidence interval of 95%
* For all the data samples compute the same. Compare the two readings.

**Understanding the problem statement:**

We had calculated bootstrap confidence intervals in previous projects also. Similarly, in this project the same steps have been adopted. Bootstrap confidence interval is simply sampling with replacement. So after the data is sampled with replacement, its sorted and desired levels are truncated. The statistical confidence interval is the usual "estimate plus-or-minus a margin of error". To calculate this, the inbuilt function **sem** is used which is *standard error of measurement.* The confidence interval runs from xbar - zα/2\*σ/sqrt(n) to  xbar + zα/2\*σ/sqrt(n). The same is done by sem (taken from Piazza).

**Program Written**:

import random

import numpy as np

from scipy.stats import sem

with open('DATA.dat') as f:

    li = [line.split()[2] for line in f]

sample\_stat = []

for i in range(15):

    sample\_stat.append(int(li[i]))

print(sample\_stat)

print('------BOOTSTRAP SAMPLING------')

mean\_bootstrap\_list = []

for i in range(100):

    bootstrap\_sample = np.random.choice(sample\_stat, 100)

    mean\_bootstrap = (sum(bootstrap\_sample)/len(bootstrap\_sample))

    mean\_bootstrap\_list.append(mean\_bootstrap)

# print(mean\_bootstrap\_list)

lower\_bound\_bootstrap\_percentile = np.percentile(mean\_bootstrap\_list, 2.5)

upper\_bound\_bootstrap\_percentile = np.percentile(mean\_bootstrap\_list, 97.5)

print(lower\_bound\_bootstrap\_percentile)

print(upper\_bound\_bootstrap\_percentile)

print('------STATISTICAL SAMPLING------')

mean\_stat = (sum(sample\_stat)/len(sample\_stat))

mean\_stat\_95ci = 1.96 \* sem(sample\_stat)

lower\_bound\_stat\_percentile = mean\_stat - mean\_stat\_95ci

upper\_bound\_stat\_percentile = mean\_stat + mean\_stat\_95ci

print(lower\_bound\_stat\_percentile)

print(upper\_bound\_stat\_percentile)

**Results and Observation**:

**Output Screen: (for 15 samples)**

------BOOTSTRAP SAMPLING------

68.344

74.12575

------STATISTICAL SAMPLING------

63.278770830413194

78.58789583625348

**Output Screen: (for all samples)**

------BOOTSTRAP SAMPLING------

68.32849999999999

73.28875

------STATISTICAL SAMPLING------

69.28139874542947

72.51271890162934

The program written above was used both for 15 and all the samples. It can be clearly seen that bootstrap sampling performs better with respect to statistical sampling with respect to value of confidence interval obtained. The values of confidence interval hardly changed for two experiment. However, for statistical confidence interval a great change is seen in the two cases. Earlier the interval was (63.27,78.58) ad it became (69.28,72.51). This is a huge change.