STA6106 – Homework1

By: Anmol Suresh Kumar Panchal; UID: 4446829

Q.1:

a] Estimate I using basic Monte Carlo Integration for function (1/sqrt(8*pi)) *e^(-x^2/8) N(0,4)

b] Estimate I using Importance Sampling of same above equation

c] Which is a better method?

Solution:

In [29]:
```
import math
import pandas
import numpy as np
import scipy.stats
from scipy.stats import uniform
import math
from scipy.stats import norm
import matplotlib
import matplotlib.pyplot as plt
```

In [37]:
```
N=1000
x = np.random.uniform(0,4,N)
y = (0.1994582095*2.178*np.exp(-x*np.exp(2)/8))
m = np.mean(y)
s = np.var(y)
z = (norm.pdf(y)/N)
a= np.mean(z)
b= np.var(z)
print(m,s)
print(a,b)
plt.plot(z,norm.pdf(y))
print(z)
```

```
0.11403173901673128 0.011659064946209937
0.0003941297627505069 5.883052733705017e-11
```

Importance Sampling is a better method as Importance sampling speeds up Monte Carlo procedures for rare events (a “Monte Carlo procedure” is sampling based on random walks). As it speeds up the process, it’s sometimes referred to as “fast simulation using importance sampling.” It’s also called a “forced Monte Carlo procedure” because it’s forcing the Monte Carlo procedure to behave somewhat abnormally. If you’re using Monte Carlo procedures, you’re more than likely using software because of the large number

of computations. And also Importance Sampling yields more accurate approximations and estimations of the integral functions.

Q.5.03:

In [1]:
```
import numpy as np
import scipy.stats
from scipy.stats import uniform
from scipy.stats import norm
from scipy.stats import beta
import math
import pandas as pd
```

In [65]:
```
N = 1000
e =  2.718281
def estimate():
    def g(y):
        return e**(-y/2)*1/2
    xs = g(np.random.uniform(size = N))
    theta_hat = np.mean(xs)
    var = st.variance(xs)/N
#     df = pd.DataFrame(theta_hat, var)
    return (theta_hat, var)
```

In [57]:
```
def estimate_unif_max():
    def g(y):
        return e**(-y)
    xs = g(np.random.uniform(0, 0.5, N))
    var = st.variance(xs)/N
    theta_hat = np.mean(xs)*1/2
#     df = pd.DataFrame(theta_hat, var)
    return theta_hat, var
```

```
In [80]: def estimate_exp():
             def g(y):
                 return 1/y
             y = np.random.exponential(scale = 1, size = N)
             var = st.variance(y)/N
             theta_hat = np.mean(y)
         #     df = pd.DataFrame(theta_hat, var)
             return theta_hat, var
```

```
In [66]: estimate()

Out[66]: (0.39394949186205164, 3.314779337454527e-06)
```

```
In [58]: estimate_unif_max()

Out[58]: (0.3934063376631093, 1.2184812703294514e-05)
```

```
In [81]: estimate_exp()

Out[81]: (1.0421580799088643, 0.0010682387333829412)
```

The variance of exponential distribution is less than by normal sampling variance as we know Sampling Uniform Distribution the pdf values are uniformly distributed across [a,b] whereas in Exponential Distribution the pdf for distribution across [a,b] is exponentially distributed which is also close to binomial distribution. Hence, if λ < 1 the curve starts lower and flatter than for the standard exponential. The asymptotic limit is the x-axis. Therefore, we get lower variance the uniform distribution.

Q.5.04:

```
In [1]: n = 10000
        a = 3
        b = 3

        # checking density
        xs = rbeta(n, shape1 = a, shape2 = b)
        plot(density(xs))

        mc.cdf.beta = function (p, shape1, shape2) {
          if (p <= 0 || p >= 1) return(0)
          us = runif(n)
          return(mean(dbeta(p*us, a, b)) * p)
        }

        xs = seq(0, 1, 0.1)

        round(rbind(sapply(xs, function (x) mc.cdf.beta(x, a, b)), sapply(xs, function(x) pbeta(x, a, b))), 3)
```

0	0.009	0.059	0.163	0.319	0.5	0.677	0.841	0.938	0.981	0
0	0.009	0.058	0.163	0.317	0.5	0.683	0.837	0.942	0.991	1

```
#Q.5.4
import math
import pandas
import numpy as np
import scipy.stats
from scipy.stats import uniform
import math
from scipy.stats import norm
import matplotlib
import matplotlib.pyplot as plt
a=3
b=3
x=np.arange(0,1,0.1)
y=scipy.stats.beta.pdf(x,a,b)
z=scipy.stats.beta.cdf(x,a,b)
print(y)
print(z)
plt.plot(x,y)
```

```
[0.    0.243 0.768 1.323 1.728 1.875 1.728 1.323 0.768 0.243]
[0.      0.00856 0.05792 0.16308 0.31744 0.5     0.68256 0.83692 0.94208
 0.99144]
```

The beta values we get in second figure varies with pbeta values we obtained in R function as shown in first figure.

Q.10:

In [15]:
```
n<-10000
CL<- 0.95
g<- function(x)
    (1-0)*exp((2.178^-x)/(1+x^2))
x<- runif(n,0,1)
u<-c(x,1-x)
u<-(1-0)*u+1
y<-g(u)
mu3<-mean(y)
mu4<-var(y)
mu3
mu4
```

```
1.11664111603407
```

```
0.00360269889387697
```

In [11]:
```
se3<-sqrt(var(g(x) + g(1-x))/(2*n))
CI <- c(mu3-qnorm((1+CL)/2)*se3,mu3+qnorm((1+CL)/2*se3))
CI
```

```
1.11400904044103  -1.93590776469459
```

When we are using Monte Carlo averages of quantities f(Xi) then the randomness in the algorithm leads to some error cancellation. In antithetic sampling we try to get even more cancellation. An antithetic sample is one that somehow gives the opposite value of f(x), being low when f(x) is high and vice versa. Ordinarily we get an opposite f by sampling at a point xe that is somehow

opposite to x.

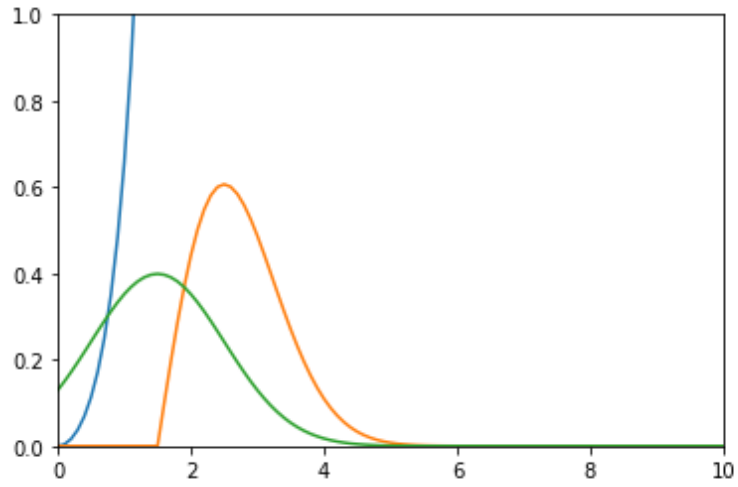
Q.5.13

5.13

```
from scipy.stats import rayleigh
def g(x):
    return (x**2)/(np.sqrt(2*math.pi)*np.exp(-x**2/2))

xs = np.arange(0,10.1,0.1)
ys_g = g(xs)
ys_rayleigh = rayleigh.pdf(xs, 1.5)
ys_norm = norm.pdf(xs, 1.5)
lim =  max(np.r_[ys_g, ys_rayleigh, ys_norm])
```

```
import matplotlib.pyplot as plt
plt.plot(xs, ys_g)
plt.plot(xs, ys_rayleigh)
plt.plot(xs, ys_norm)
plt.ylim(ymin = 0, ymax=1)
plt.xlim(xmin = 0, xmax = 10)
plt.show()
```



The two importance sampling functions are rayleigh and normal distribution functions.

def is_rayleigh() performs better than def is_norm(). As normal distribution has variance of 1 whereas rayleigh distribution has variance of 0.429 and the precision of this estimate depends on the variance of X in importance sampling.

Q.5.14

```
def g(x):
    res = x**2/np.sqrt(2*math.pi)*np.exp(-x**2/2)
    return res

sigma_rayleigh = 1.5
mean = 1.5
n = 10000

def f1(x):
    return rayleigh.pdf(x, sigma_rayleigh)

def f2(x):
    return norm.pdf(x, mean)

def rf1():
    return rayleigh.rvs(sigma_rayleigh, size = n)

def rf2():
    return norm.rvs(mean, size = n)

def is_rayleigh():
    xs = rf1()
    a = g(xs)/f1(xs)
    return np.mean(a)
```

```
def is_norm():
    xs = rf2()
    a = g(xs)/f2(xs)
    return np.mean(a)
```

```
theta1 = is_rayleigh()
theta2 = is_norm()
print("Theta1:", theta1, " Theta2:", theta2)
```

```
Theta1: 0.2577996357738749  Theta2: 0.8501566201196435
```