**Homework 3**

**Han Liu**

**20608469**

The assignment questions are solved using Python. The modules needed to repeat following results are:

import math  
import random  
import pickle  
import warnings  
import numpy as np  
import matplotlib.pyplot as plt

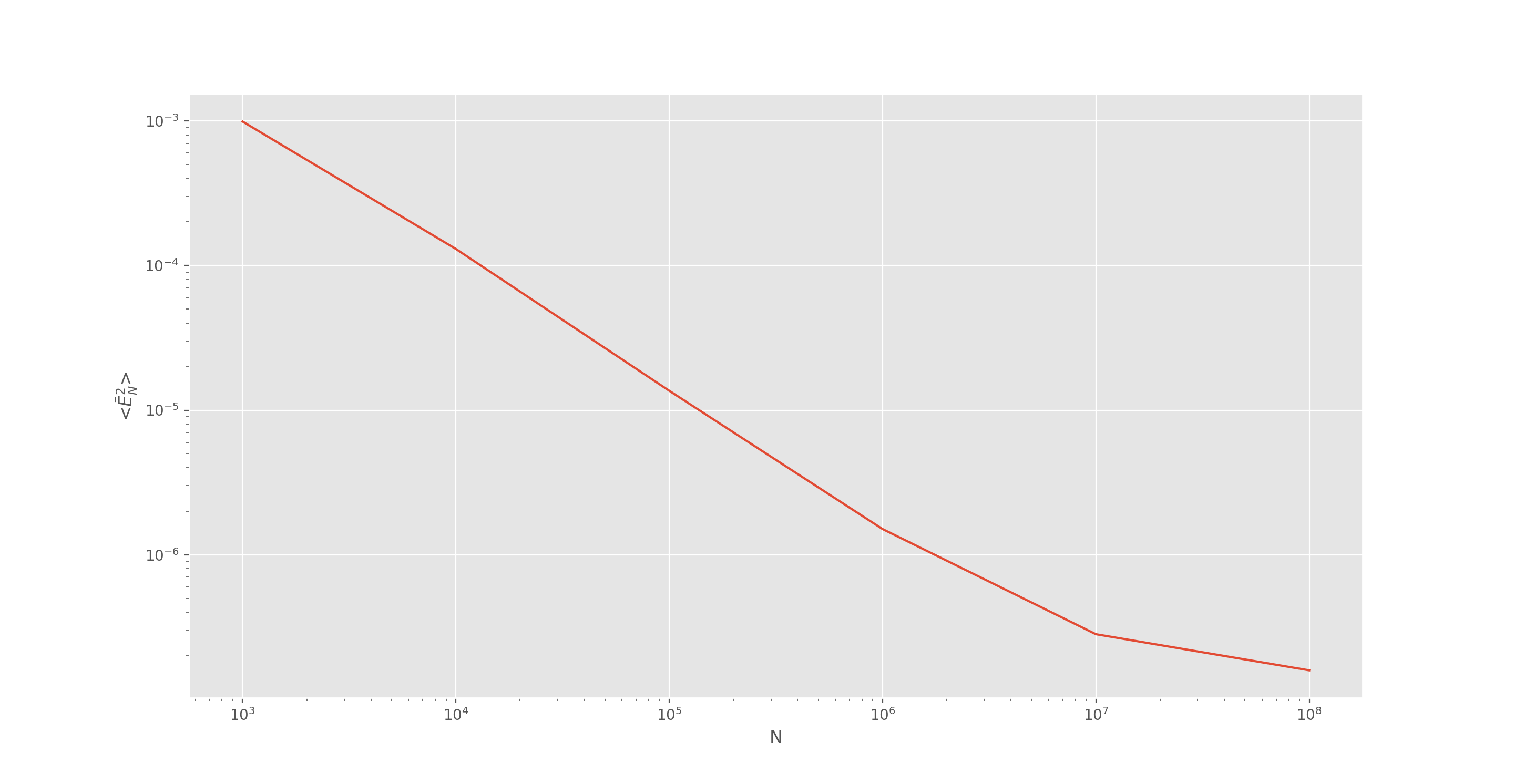
***4-10***

1. Based on equation 4.24, all the possible values that an (xi, yi) pair can take on range from ([-1,1], [-1,1]), which occupies a square centered at the origin. The point is only considered “accepted” when its distance to the origin is smaller or equal to 1, which occupies a circle with a radius of 1 centered at the origin. Assuming the random number generator is at least pseudo-random, this implies that every single location in the domain has an equal probability of being generated. Asymptotically, as more and more pairs are generated, the probability of landing within the circle equals to the area of the circle divided by the total probability, which is the area of the square. The area of the circle is and the area of the square is . Therefore, the acceptance ratio approaches asymptotically.
2. As shown in the graph below, log <EN2> and log N has linear relationship in the double-log plot. Therefore, <EN2> is proportional to 1/N.

A picture containing wall

Description automatically generated

1. As shown in the graph below, similar to (b), <EN2> initially scales with 1/N. However, the tail of <EN2> eventually levels off likely due to the long-term predictable behavior of the bad random number generator.



The Python code responsible for Problem 4-10 is attached below:

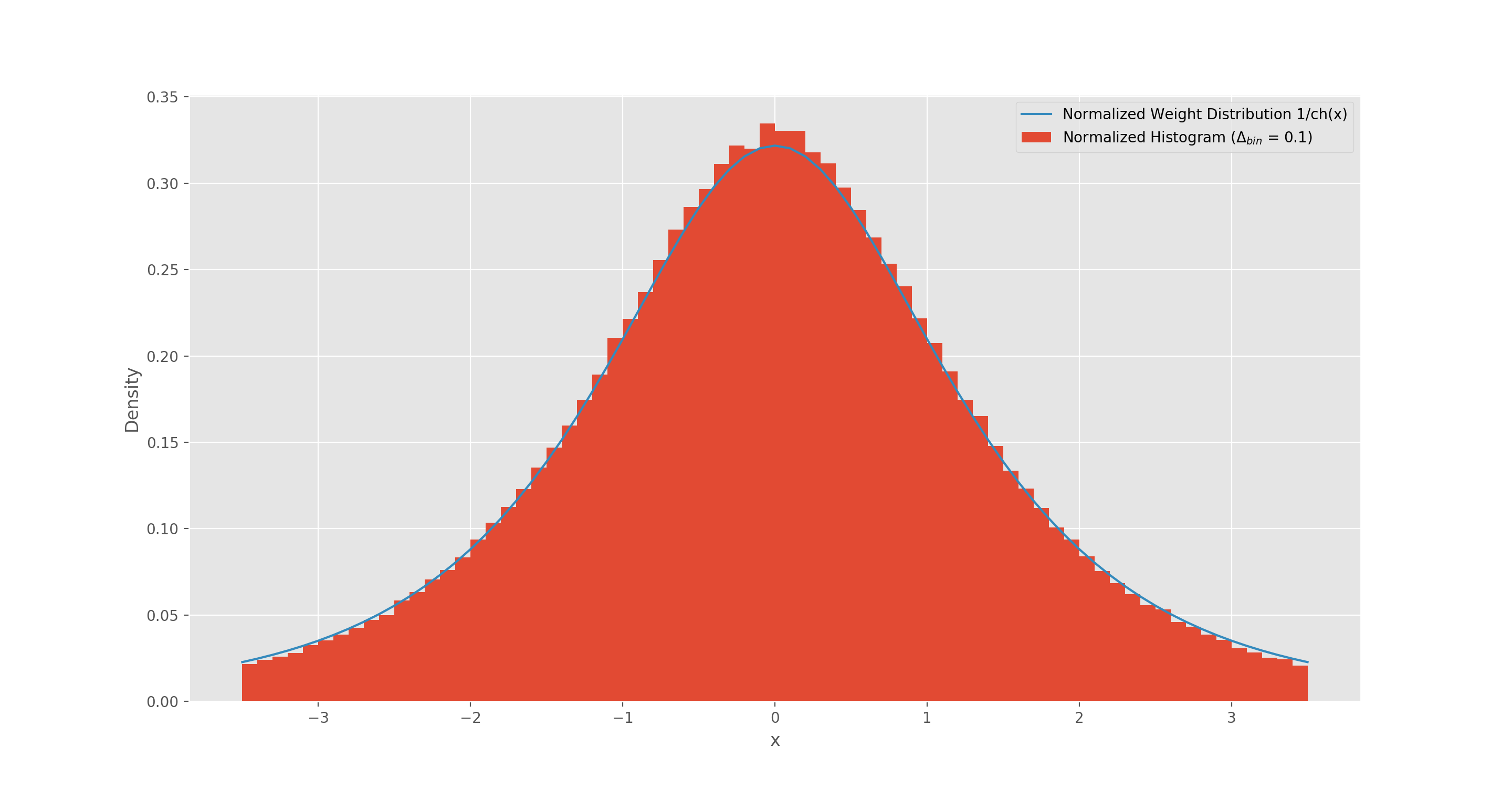
import random  
import math  
import matplotlib.pyplot as plt  
  
class BadRandomNumberGenerator(object):  
 def \_\_init\_\_(self, a=1277, c=0, m=131072):  
 self.a=a  
 self.c=c  
 self.m=m  
 self.S=1  
 def getNumber(self):  
 self.S=((self.a\*self.S)+self.c)%self.m  
 return (self.S/self.m)  
  
#Problem 4-10  
#(b)  
  
graph\_y=[0]  
l=[]  
n\_accepted=0  
for i in range(1,(10\*\*8)+1):  
 x=2\*random.random()-1  
 y=2\*random.random()-1  
 r2=(x\*\*2)+(y\*\*2)  
 if r2<=1:  
 n\_accepted+=1  
 accepted=n\_accepted/i  
 E\_N2=(accepted-math.pi/4)\*\*2  
 l.append(E\_N2)  
 if i in [10\*\*3,10\*\*4,10\*\*5,10\*\*6,10\*\*7,10\*\*8]:  
 y=(sum(l)+((i/10)\*graph\_y[-1]))/i  
 print(y)  
 graph\_y.append(y)  
 l=[]  
  
del graph\_y[0]  
plt.style.use('ggplot')  
fig = plt.figure()  
ax1 = fig.add\_subplot(111)  
ax1.loglog([10\*\*3,10\*\*4,10\*\*5,10\*\*6,10\*\*7,10\*\*8],graph\_y)  
  
ax1.set\_xlabel("N")  
ax1.set\_ylabel(r"<$\bar E\_{N}^2$>")  
plt.show()  
  
#(c)  
  
graph\_y=[0]  
l=[]  
n\_accepted=0  
num=BadRandomNumberGenerator()  
for i in range(1,(10\*\*8)+1):  
 x=2\*num.getNumber()-1  
 y=2\*num.getNumber()-1  
  
 r2=(x\*\*2)+(y\*\*2)  
 if r2<=1:  
 n\_accepted+=1  
 accepted=n\_accepted/i  
 E\_N2=(accepted-math.pi/4)\*\*2  
 l.append(E\_N2)  
 if i in [10\*\*3,10\*\*4,10\*\*5,10\*\*6,10\*\*7,10\*\*8]:  
 y=(sum(l)+((i/10)\*graph\_y[-1]))/i  
 print(y)  
 graph\_y.append(y)  
 l=[]  
  
del graph\_y[0]  
plt.style.use('ggplot')  
fig = plt.figure()  
ax1 = fig.add\_subplot(111)  
ax1.loglog([10\*\*3,10\*\*4,10\*\*5,10\*\*6,10\*\*7,10\*\*8],graph\_y)  
  
ax1.set\_xlabel("N")  
ax1.set\_ylabel(r"<$\bar E\_{N}^2$>")  
plt.show()

***4-11***

1. . Note is not a function of , which is randomly sampled between 0 and . In addition, the terms containing cancel each other out on the numerator and denominator. Here we can see that the weight function is . If we let , then and . Therefore, . Note here the interval over which the integral is evaluated is changed. Therefore, algorithm (b) is correct and the analytical solution is .
2. Algorithm (a) essentially, this sampling method does not take into account the Jacobian. . The equation has a non-zero analytical solution of .

***4-12***

The weight follows the distribution, whose integral from -∞ to +∞ is π. Therefore has a normalized distribution. Because has its peak at x=0 as shown in the plot below, therefore the initial value from the Metropolis algorithm is selected to be near 0. In addition, the step size δ is adjusted for a trial run of size of so that the overall acceptance rate is between 40% and 60%. A warning feature has been implemented to ensure that δ meets such requirement. Based on trial and error, when δ is around 3, the rate of acceptance is around 55%. After generating 106 data points randomly following the weight distribution, the normalized histogram of the set is compared to the normalized weight function as shown in the graph attached below.



The Python code responsible for Problem 4-12 is attached below:

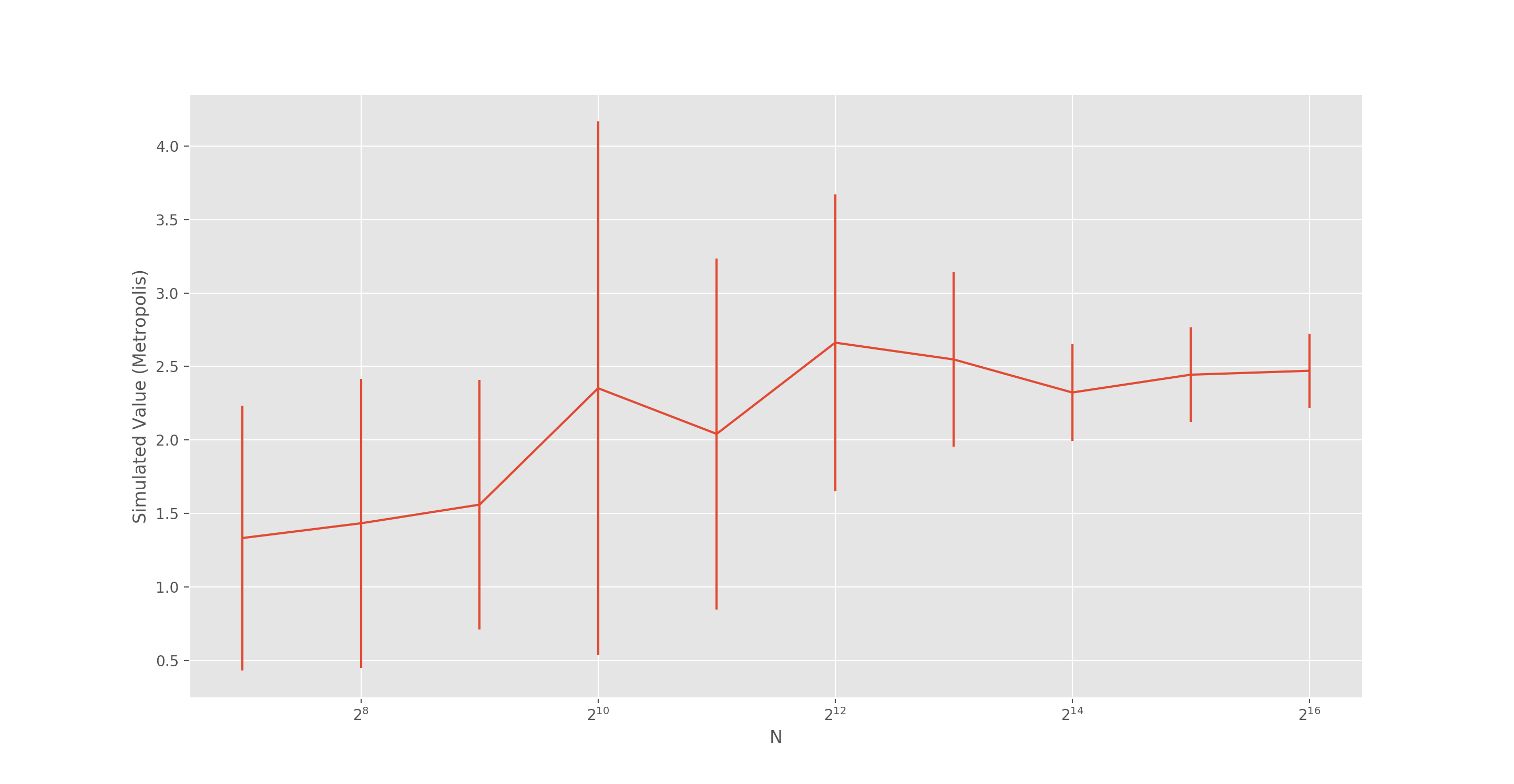
#Problem 4-12  
  
delta=3  
x0=0  
N=10\*\*6  
data=[x0]  
num\_reject=0  
  
def weightDistribution(x):  
 return 1/(math.cosh(x))  
  
def normalizedWeightDistribution(x):  
 return 1 / (math.cosh(x)\*math.pi)  
  
for i in range(N-1):  
 new\_x=data[-1]+delta\*(1-(random.random()\*2))  
 w=weightDistribution(data[-1])  
 new\_w=weightDistribution(new\_x)  
  
 if i==0.01\*N:  
 acceptance\_rate=(1-(num\_reject/(0.01\*N)))  
 print(acceptance\_rate)  
 if acceptance\_rate < 0.4 or acceptance\_rate > 0.6:  
 warnings.warn("The accpetance rate is not around 50%, please adjust step size.")  
 if (new\_w/w)>=1:  
 data.append(new\_x)  
 else:  
 r=random.random()  
 if (new\_w/w)>r:  
 data.append(new\_x)  
 else:  
 data.append(data[-1])  
 num\_reject+=1  
  
normalized=[normalizedWeightDistribution(i) for i in np.linspace(-3.5,3.5,71)]  
  
plt.style.use('ggplot')  
fig = plt.figure()  
ax1 = fig.add\_subplot(111)  
ax1.hist(data,bins=np.linspace(-3.5,3.5,71),density=True, label=r"Normalized Histogram ($\Delta\_{bin}$ = 0.1)")  
ax1.plot(np.linspace(-3.5,3.5,71), normalized,label=r"Normalized Weight Distribution 1/ch(x)")  
ax1.set\_xlabel("x")  
ax1.set\_ylabel("Density")  
ax1.legend()  
plt.show()

***4-13***

The weight follows the distribution. This question asks to evaluate the integral of using the total number of sampling points with Metropolis method where .

Because the weight function has a distribution centered about 0, the initial point x0 starts at 0 and delta is adjusted so that the acceptance rate is around 50% based on the trial population, which is defined to be the first 10% of the samples. If trial population is smaller than 10 (when ), then it is concluded that the sample size is too size to calculate the rate of acceptance. The simulated output log is shown below.

To investigate the convergence behavior of the simulated result, the step size is set to 0.5 and the Metropolis simulation is placed in a “while” loop until 20 stimulated results have been generated, each with an acceptance rate between 40% and 60%. The mean and standard deviation for each N from 26 to 216 is plotted to show that as N increases the simulated result converges to the analytical result 2.47 as shown below.



A screenshot of text

Description automatically generated

A screenshot of a cell phone

Description automatically generated

The Python code responsible for Problem 4-13 is attached below:

#Problem 4-13  
  
def weightDistribution(x):  
 return 1/(math.cosh(x))  
  
def f(x):  
 return x\*\*2  
  
def Metropolis(N, weight, function, delta=0.5, x0=0, test\_tolerance=0.1,limit=[10\*\*(-4),10\*\*4]):  
  
 data=[x0]  
 num\_reject=0  
 for i in range(1, N + 1):  
 new\_x=data[-1]+delta\*(1-(random.random()\*2))  
 w=weight(data[-1])  
 new\_w=weight(new\_x)  
  
 if int(test\_tolerance\*N)>=10:  
 if i==int(test\_tolerance\*N):  
 acceptance\_rate=(1-(num\_reject/(test\_tolerance\*N)))  
 # print(acceptance\_rate)  
 if acceptance\_rate < 0.4 or acceptance\_rate > 0.6:  
 warnings.warn("The accpetance rate is not around 50%, please adjust step size.")  
 else:  
 acceptance\_rate = None  
  
 if new\_x<limit[0] or new\_x>limit[1]:  
 data.append(data[-1])  
 num\_reject+=1  
  
 if (new\_w/w)>=1:  
 data.append(new\_x)  
 else:  
 r=random.random()  
 if (new\_w/w)>r:  
 data.append(new\_x)  
 else:  
 data.append(data[-1])  
 num\_reject+=1  
  
 f = [function(i) for i in data]  
 I = sum(f)/float(len(f))  
  
 if acceptance\_rate is None:  
 print("Using ", str(N), " sample points, given step size ", str(delta), ", the integral evaluates to be ", str(I), "; \nan acceptance rate is not calculated as the total number of sample points collected is too small (<10).")  
 else:  
 print("Using ", str(N), " sample points, given step size ", str(delta), ", the integral evaluates to be ", str(I), " with an acceptance rate of ", str(acceptance\_rate\*100), "%.")  
  
 return I, acceptance\_rate  
  
  
I = Metropolis(N=2\*\*12, weight=weightDistribution, function=f)  
  
# l=np.array([7,8,9,10,11,12,13,14,15,16])  
# d=dict()  
# for i in l:  
# d[i]=[]  
# N=2\*\*i  
# while len(d[i])<20:  
# result,rate = Metropolis(N,weightDistribution,function=f)  
# if abs(rate - 0.5) < 0.1:  
# d[i].append(result)  
#  
# with open('Metropolis Method.pickle','wb') as handle:  
# pickle.dump(d,handle)  
  
d = pickle.load(open('Metropolis Method.pickle', 'rb'))  
  
avg=[]  
std=[]  
  
l=[7,8,9,10,11,12,13,14,15,16]  
N=[]  
for i in l:  
 N.append(2\*\*i)  
  
for i in l:  
 avg.append(np.mean(d[i]))  
 std.append(np.std(d[i]))  
  
plt.style.use('ggplot')  
fig = plt.figure()  
ax1 = fig.add\_subplot(111)  
ax1.errorbar(N,avg,yerr=std)  
ax1.set\_xscale('log',basex=2)  
ax1.set\_xlabel("N")  
ax1.set\_ylabel("Simulated Value (Metropolis)")  
plt.show()

***4-14***

The weight follows the distribution of , which has its peak at . Therefore, the initial vector for the multidimensional Metropolis method is 0, 0, …, 0 for x1, x2, …, xD. In addition, the integral of from -∞ to +∞ is . Therefore, its normalized version can be used to verify the computer-generated histogram in the end.

The step size vector δ is assumed to be uniform across all dimensions and adjusted for a trial run of size of so that the overall acceptance rate lies between 40% and 60%. Based on trial and error, as dimension D increases, the acceptance rate decreases if the step size is kept constant. For example, the acceptance rate is 58% when with δ = 1. In comparison, the acceptance is 43% when with δ =1. A warning feature has been implemented to ensure that δ meets such requirement. After generating 107 data points randomly following the weight distribution, the normalized histogram of the set is shown below. It is expected that r spreads more as the dimensionality increases because collectively “r” is more likely to increase as the new point x’ is more likely to move away from the previous point in at least one of the D dimensions.

A screenshot of a social media post

Description automatically generated

The Python code responsible for Problem 4-14 is attached below:

#Problem 4-14  
  
def weightDistribution(r\_sqrt):  
 return (math.e\*\*(-1\*r\_sqrt))  
  
def normalizedWeightDistribution(r\_sqrt):  
 return (math.e\*\*(-1\*r\_sqrt))/(math.pi\*\*0.5)  
  
def distributionGenerator(D,delta,N=10\*\*7):  
 x0=np.array([0]\*D)  
 data=[x0]  
 num\_reject=0  
  
 for i in range(N-1):  
 new\_x=np.array([ i+delta\*(1-(random.random()\*2))for i in data[-1]])  
 new\_r\_sqrt=(new\_x\*\*2).sum()  
 w=weightDistribution((np.array(data[-1])\*\*2).sum())  
 new\_w=weightDistribution(new\_r\_sqrt)  
  
 if i==0.01\*N:  
 acceptance\_rate=(1-(num\_reject/(0.01\*N)))  
 print(acceptance\_rate)  
 if acceptance\_rate < 0.4 or acceptance\_rate > 0.6:  
 warnings.warn("The accpetance rate is not around 50%, please adjust step size.")  
 if (new\_w/w)>=1:  
 data.append(new\_x)  
 else:  
 r=random.random()  
 if (new\_w/w)>r:  
 data.append(new\_x)  
 else:  
 data.append(data[-1])  
 num\_reject+=1  
 return data  
  
# D=1  
# data=distributionGenerator(D, delta=2)  
# data=[ (i\*\*2).sum() for i in data]  
# with open('D1.pickle','wb') as handle:  
# pickle.dump(data,handle)  
  
# D=2  
# data=distributionGenerator(D, delta=1)  
# data=[ (i\*\*2).sum() for i in data]  
# with open('D2.pickle','wb') as handle:  
# pickle.dump(data,handle)  
  
# D=3  
# data=distributionGenerator(D, delta=1)  
# data=[ (i\*\*2).sum() for i in data]  
# with open('D3.pickle','wb') as handle:  
# pickle.dump(data,handle)  
  
# D=4  
# data=distributionGenerator(D, delta=1)  
# data=[ (i\*\*2).sum() for i in data]  
# with open('D4.pickle','wb') as handle:  
# pickle.dump(data,handle)  
  
# D=5  
# data=distributionGenerator(D, delta=0.75)  
# data=[ (i\*\*2).sum() for i in data]  
# with open('D5.pickle','wb') as handle:  
# pickle.dump(data,handle)  
  
D1 = pickle.load(open('D1.pickle', 'rb'))  
D2 = pickle.load(open('D2.pickle', 'rb'))  
D3 = pickle.load(open('D3.pickle', 'rb'))  
D4 = pickle.load(open('D4.pickle', 'rb'))  
D5 = pickle.load(open('D5.pickle', 'rb'))  
  
plt.style.use('ggplot')  
fig = plt.figure()  
ax1 = fig.add\_subplot(321)  
ax2 = fig.add\_subplot(322)  
ax3 = fig.add\_subplot(323)  
ax4 = fig.add\_subplot(324)  
ax5 = fig.add\_subplot(325)  
  
weights=np.ones(len(D1))/len(D1)  
ax1.hist(D1,bins=np.linspace(0,5,51),density=True, label=r"Normalized Histogram for D = 1 ($\Delta\_{bin}$ = 0.1)")  
ax1.set\_xlabel("r")  
ax1.set\_ylabel("Density")  
ax1.legend()  
  
ax2.hist(D2,bins=np.linspace(0,5,51),density=True, label=r"Normalized Histogram for D = 2 ($\Delta\_{bin}$ = 0.1)")  
ax2.set\_xlabel("r")  
ax2.set\_ylabel("Density")  
ax2.set\_ylim(0,1)  
ax2.legend()  
  
ax3.hist(D3,bins=np.linspace(0,5,51),density=True, label=r"Normalized Histogram for D = 3 ($\Delta\_{bin}$ = 0.1)")  
ax3.set\_xlabel("r")  
ax3.set\_ylabel("Density")  
ax3.set\_ylim(0,1)  
ax3.legend()  
  
ax4.hist(D4,bins=np.linspace(0,5,51),density=True, label=r"Normalized Histogram for D = 4 ($\Delta\_{bin}$ = 0.1)")  
ax4.set\_xlabel("r")  
ax4.set\_ylabel("Density")  
ax4.set\_ylim(0,1)  
ax4.legend()  
  
ax5.hist(D5,bins=np.linspace(0,5,51),density=True, label=r"Normalized Histogram for D = 5 ($\Delta\_{bin}$ = 0.1)")  
ax5.set\_xlabel("r")  
ax5.set\_ylabel("Density")  
ax5.set\_ylim(0,1)  
ax5.legend()  
  
plt.show()