Homework1

October 15, 2022

Anubhav Shankar (01951462)

Monte - Carlo Integration Pseudo Code

```
double random_number(){


    //Generate uniform random number

    return (double)rand()/((double)RAND_MAX+1);

    //We need to cast the rand() to double as it is an int by default.


   This is particularly required in this case as we expect our output to be a double.


 }


double func(double x){


//This function will be used to pass the integrand that we need to approximate


}


double monteCarlo(double lower, double upper, double N){

      //General function to execute the Monte Carlo Approximation
      //Can be used for any integrand and any bound

     double apx = 0.0;
     //This will store the approximate results of the integrand for every iteration
     //within our defined bounds
     double sum = 0.0;
     //This stores the incremented value for every iteration.
     //This is basically V of the MonteCarlo integration


    for(int i = 0; i<N; i++){

           //Generate a random number within the limits of the integration

             double rand_num = random_number();

             double func_value = func(rand_num);

             sum+=func_value;
    }


  return apx = (upper - lower) * (sum/N);
  // N is the number of iterations for this function. It can take any value

}


int main(){


   call the Monte Carlo integration function here


   return 0;


}
```

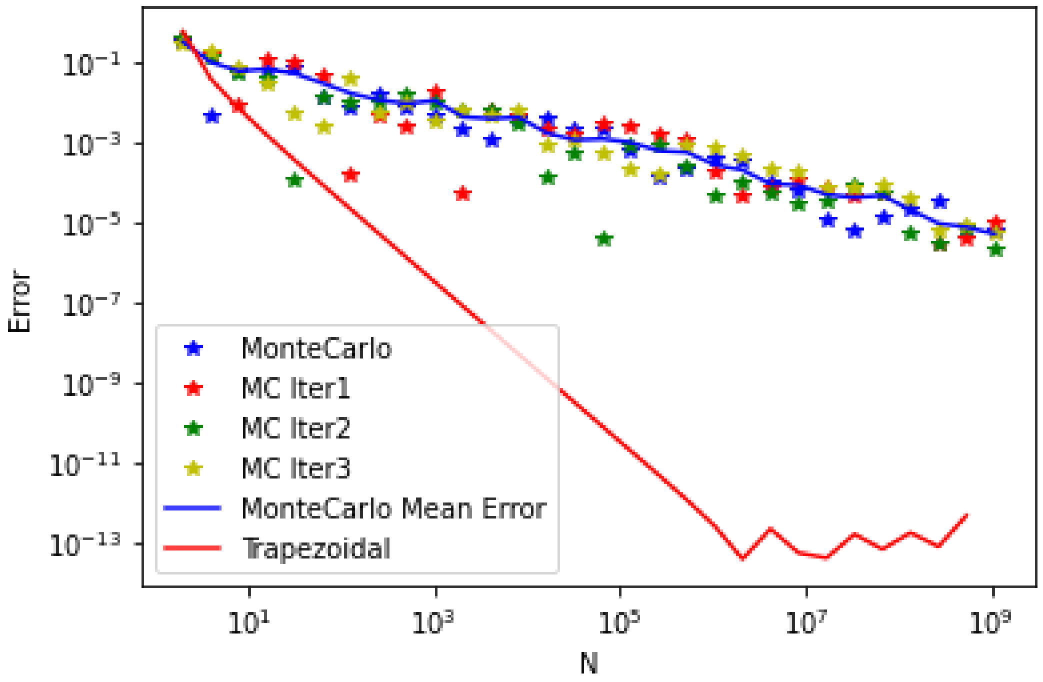
[17]:
```python
# Load required packages

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn.linear_model import LinearRegression
import pandas as pd
```

[18]:
```python
# Read in the .dat files
montecarlo = np.loadtxt("MonteCarlo.dat")
montecarlo1 = np.loadtxt("MonteCarlo1.dat")
montecarlo2 = np.loadtxt("MonteCarlo2.dat")
```

```
montecarlo3 = np.loadtxt("MonteCarlo3.dat")
montecarlo4 = np.loadtxt("MonteCarlo4.dat")
trap = np.loadtxt("mydata.txt")

# Create two atomic vectors/variables to store the dat file contents
x = montecarlo[:,0]
y = montecarlo[:,2]

x1 = montecarlo1[:,0]
y1 = montecarlo1[:,2]

x2 = montecarlo1[:,0]
y2 = montecarlo2[:,2]

x3 = montecarlo3[:,0]
y3 = montecarlo3[:,2]

x4 = montecarlo4[:,0]
y4 = montecarlo4[:,2]

xt = trap[0:29,0]
yt = trap[0:29,1]
```

[19]:
```
# Smoothing out the Monte Carlo error

y_add = y + y1 + y2 + y3 + y4

y_mean = y_add/5 #Divide it by the number of iterations and not the number of␣
 ↪trials
```

[20]:
```
# Create a log-log plot

plt.loglog(x, y,'b*',label='MonteCarlo')
plt.loglog(x1, y1,'r*',label='MC Iter1')
plt.loglog(x2, y2,'g*',label='MC Iter2')
plt.loglog(x3, y3,'y*',label='MC Iter3')
plt.loglog(x, y_mean,'b',label='MonteCarlo Mean Error')
plt.loglog(xt, yt,'r',label='Trapezoidal')
plt.legend(loc=3)
plt.xlabel("N")
plt.ylabel("Error")

plt.savefig("plot.png")
```



From the above plot we see that the Trapezoidal function's error decomposes at a much faster rate than the Monte-Carlo approximation. This shows that the Trapezoidal rule is a higher order method. However, Trapezoidal rule is not suitable for higher dimensional problem solving where the associated features and complexity increase manyfold.

Now, we see that the integrand approximation follows a power law. To fit this non-linear form into a linear function we'd need to take the logarithm of the Monte-Carlo Mean Error. The error finction then behaves like the equation below:

```
log(E) = B + A * log(N)                    -(1)
```

(1) looks like a linear regression equation where, 'B' is the intercept and 'A' is the coefficient of an independent variable/slope of the equation. In this case, the independent variable is 'N' which is the number of iterations.

Let us now run a linear regression model to with the mean error (E) as the dependent variable and the iterations (N) as the independent variable.

[21]:
```
model = LinearRegression().fit(np.log(x.reshape(len(x),1)), np.log(y_mean.
 ↪reshape(len(y_mean),1)))

# Sklearn expects the input to be a 2-D array. However, in our case, we are␣
 ↪using the number of iterations i.e. N and
# the error values i.e. E as 1-D arrays. Hence, we need to reshape our input to␣
 ↪be between the length of our 1-D array and
```

```
# scale it to 1.
```

[22]:
```
model.coef_
```

[22]:
```
array([[-0.50702882]])
```

[23]:
```
model.intercept_
```

[23]:
```
array([-1.34251826])
```

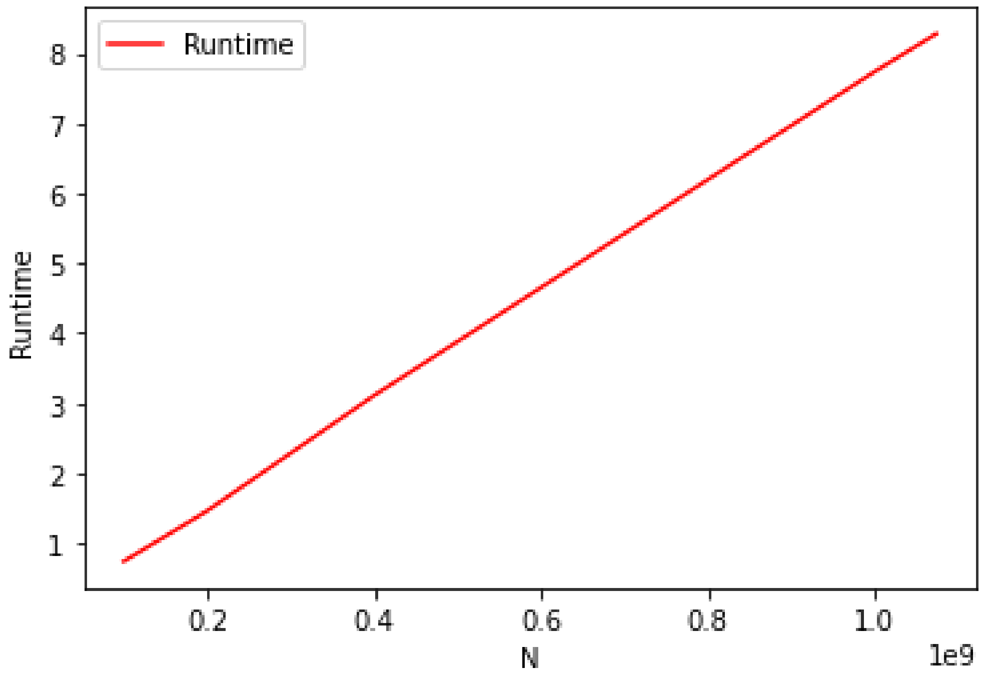
By running the Linear Regression, we see that the value of **B = -1.34251826 and A = -0.50702882**. Substituting the values in (1) we get the following final equation -

```
log(E) = -1.34251826 + -0.50702882 * log(N)
```

Let us now time our program for different values of N and see the trend.

[24]:
```
run_time = np.loadtxt("mc_runtime.dat")
n = run_time[:,0]
t = run_time[:,1]
```

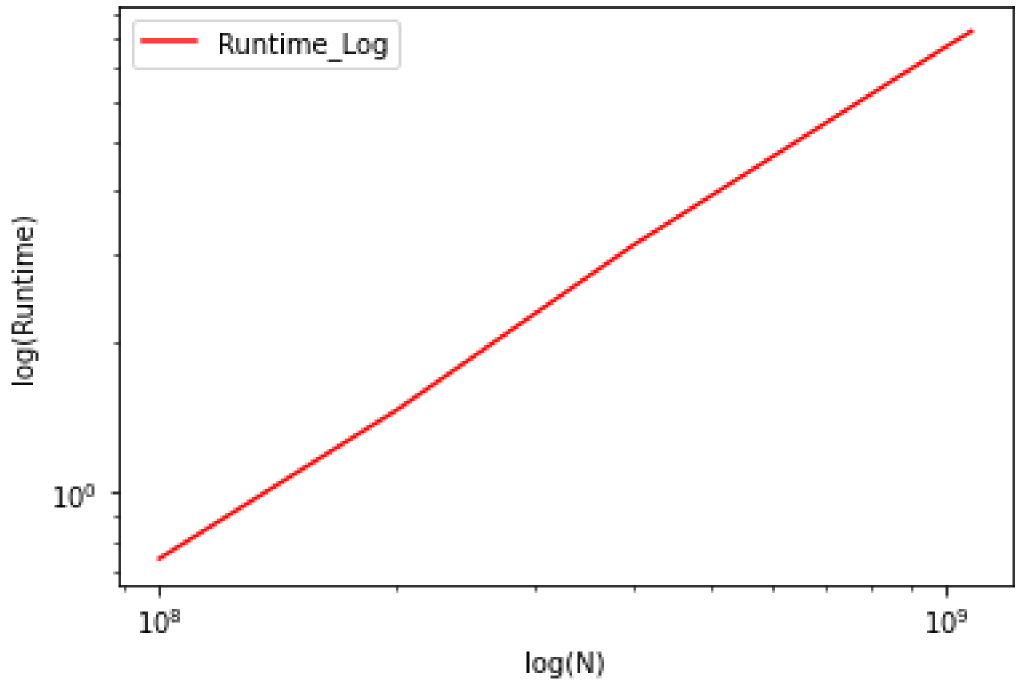
[25]:
```
plt.plot(n, t,'r',label='Runtime')
plt.legend(loc=0)
plt.xlabel("N")
plt.ylabel("Runtime")

plt.savefig("N vs. Runtime.png")
```



From the above plot we see that there is a **linear relationship between the number of points (N) and the runtime**. However, let us take a lgarithmic view for the same arguments and see if it holds true.

[26]:
```
plt.loglog(n, t,'r',label='Runtime_Log')
plt.legend(loc=0)
plt.xlabel("log(N)")
plt.ylabel("log(Runtime)")
```

[26]:
```
Text(0, 0.5, 'log(Runtime)')
```



We get the same linear trend. **As the program took ~9s to run for 2 raised to the power 30 iterations and given the linear trend of the program runtime, I'd estimate that, for 2 raised to the power 32 iterations, it'd take somewhere between 10-11s.**

The trend seems right to me as in our program, we only have one for-loop and there is no nesting perse which by default makes this program of the order **O(n)** given that it takes constant time to generate a random number and the main iteration is the calculation of the integrand once the random number is passed.

[27]:
```
############################################ SNOWFALL ESTIMATION␣
↪##############################################################
```

[28]:
```
# Load the snowfall data

sf = np.loadtxt("Snowfall.dat")
print(sf)
```

```
[[2.00000000e+00 7.06062000e-01 2.93938000e-01]
 [4.00000000e+00 1.22884000e+00 2.28840000e-01]
 [8.00000000e+00 8.59505000e-01 1.40495000e-01]
 [1.60000000e+01 1.03734900e+00 3.73490000e-02]
 [3.20000000e+01 1.15995300e+00 1.59953000e-01]
 [6.40000000e+01 9.56636000e-01 4.33640000e-02]
 [1.28000000e+02 8.19549000e-01 1.80451000e-01]
 [2.56000000e+02 9.49911000e-01 5.00890000e-02]
 [5.12000000e+02 9.84584000e-01 1.54160000e-02]
 [1.02400000e+03 9.97503000e-01 2.49700000e-03]
 [2.04800000e+03 9.82209000e-01 1.77910000e-02]
 [4.09600000e+03 9.63956000e-01 3.60440000e-02]
 [8.19200000e+03 9.64252000e-01 3.57480000e-02]
 [1.63840000e+04 9.88701000e-01 1.12990000e-02]
 [3.27680000e+04 9.98651000e-01 1.34900000e-03]
 [6.55360000e+04 1.00016800e+00 1.68000000e-04]
 [1.31072000e+05 9.96391000e-01 3.60900000e-03]
 [2.62144000e+05 9.96647000e-01 3.35300000e-03]
 [5.24288000e+05 9.98165000e-01 1.83500000e-03]
 [1.04857600e+06 9.97879000e-01 2.12100000e-03]
 [2.09715200e+06 9.98326000e-01 1.67400000e-03]
 [4.19430400e+06 9.99986000e-01 1.40000000e-05]
 [8.38860800e+06 1.00008700e+00 8.70000000e-05]
 [1.67772160e+07 1.00031400e+00 3.14000000e-04]
 [3.35544320e+07 9.99843000e-01 1.57000000e-04]
 [6.71088640e+07 9.99966000e-01 3.40000000e-05]
 [1.34217728e+08 1.00012500e+00 1.25000000e-04]
 [2.68435456e+08 1.00003700e+00 3.70000000e-05]
 [5.36870912e+08 9.99956000e-01 4.40000000e-05]
 [1.07374182e+09 9.99925000e-01 7.50000000e-05]]
```

[29]:
```
# Convert to a pandas dataframe for ease of use

sf_df = pd.DataFrame(sf,columns = ['N', 'Estimate', 'Error'])

print(sf_df)
```

```
               N  Estimate     Error
0   2.000000e+00  0.706062  0.293938
1   4.000000e+00  1.228840  0.228840
2   8.000000e+00  0.859505  0.140495
3   1.600000e+01  1.037349  0.037349
4   3.200000e+01  1.159953  0.159953
```

```
5   6.400000e+01  0.956636  0.043364
6   1.280000e+02  0.819549  0.180451
7   2.560000e+02  0.949911  0.050089
8   5.120000e+02  0.984584  0.015416
9   1.024000e+03  0.997503  0.002497
10  2.048000e+03  0.982209  0.017791
11  4.096000e+03  0.963956  0.036044
12  8.192000e+03  0.964252  0.035748
13  1.638400e+04  0.988701  0.011299
14  3.276800e+04  0.998651  0.001349
15  6.553600e+04  1.000168  0.000168
16  1.310720e+05  0.996391  0.003609
17  2.621440e+05  0.996647  0.003353
18  5.242880e+05  0.998165  0.001835
19  1.048576e+06  0.997879  0.002121
20  2.097152e+06  0.998326  0.001674
21  4.194304e+06  0.999986  0.000014
22  8.388608e+06  1.000087  0.000087
23  1.677722e+07  1.000314  0.000314
24  3.355443e+07  0.999843  0.000157
25  6.710886e+07  0.999966  0.000034
26  1.342177e+08  1.000125  0.000125
27  2.684355e+08  1.000037  0.000037
28  5.368709e+08  0.999956  0.000044
29  1.073742e+09  0.999925  0.000075
```

[30]:
```
np.mean(sf_df['Estimate'])
```

[30]:
```
0.9861825333333333
```

[31]:
```
np.mean(sf_df['Error'])
```

[31]:
```
0.04227566666666666
```

[32]:
```
np.median(sf_df['Estimate'])
```

[32]:
```
0.9982455
```

The snowfall estimation model given by Eq.(3) in the problem statement should converge to one (1) for the bounds [0,10]. While we get the average snowfall of ~1 inch from the above execution, a better way to judge the model is to run the integrand over multiple function values and if the 'Estimate' converges to ~1 that should be a good indicator of the model's effectiveness.

Mean, while a useful measure of central tendency, happens to be more volatile. Hence, we should look at the median of our observations which is also ~1.

So, by the virtue of the above observations we can say that the model has a **high degree of reliability**.