Home Work 01: EAS 520

Sayem Khan

Tuesday, October 8, 2019

Problem a

Specialize the general Monte Carlo integration method.

Solution

The problem is,

$$I_1 = \int_{-1}^{1} \frac{1}{1+x^2} dx \tag{1}$$

We need to write equation (1) as 1-D Monte Carlo integration formula, that is,

$$\hat{I}_N = V \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1+x_i^2} = 2 \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1+x_i^2} \tag{2}$$

where, $N \to \infty$, each x_i is randomly and uniformly drawn from the interval $[-1, 1]$ and V is the *volume*, that is in this case,

$$V = \int_{-1}^{1} dx = x \mid_{-1}^{1} = 1 - (-1) = 2$$

Problem b

The Pseudocode to compute the integration of equation 2.

Solution

```
def monteCarloIntigration(N):  # N = Number of random point

    random()  # Random Number Initilization

    I = 0.0  # intergral initialization

    # summation calculation
    for i in range(N):
        # Randmon number from the interval
        x = draw a random number from [-1, 1]
        f = 1/(1+x*x)
        I = I + f(x)
    end

    #final calculation
    I = 2 * (1 / N) * I  # Volume, V = 2

    # Error calculation
    abserrorCal()

end
```

Listing 1: Pseudocode for equation 2

Problem c

Solution

The implementation of equation 2 in C++ given below:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

double fqn(double x);          // function declearation
double error(double itegral); //Error calculation

int main(int argc, char **argv)
{

    //Declearation and Initialization of the variables
    int N = atoi(argv[1]); // Number of mesh
    double I_1 = 0.0;      //Approximate integration
    double a = -1.0;       //lower bound
    double b = 1.0;        // upper bound
    double V = b - a;      // volume of the intergral

    srand(time(NULL));

    for (int i = 0; i < N; i++)
    {
        int randNum = rand();
        double x = ((double)randNum / (double)RAND_MAX);
        cout << x << endl;
        I_1 = I_1 + fqn(x);
    }

    //Computitation
    I_1 = V * (1.0 / N) * I_1;
    //cout << "Value of Integration = " << I_1 << endl;
    cout << N << " " << error(I_1) << endl;
    return 0;
}

double fqn(double x)
{

    return (1.00) / (1 + x * x);
}

double error(double itegral)
{
    const double pi = 3.141592653589793;
    double absolute_error = fabs(itegral - (pi / 2.0));
    return absolute_error;
}
```

Listing 2: **monteCarlo.cpp**: C++ implementation for equation (2) with error calculation.

The bash script for Problem d, e, f given below:

```bash
#!/bin/bash
#g++ monteCarlo.cpp -o monteCarlo
g++ -Wall -o monteCarlo monteCarlo.cpp
gcc -Wall -o trap trap.c

file_1="data.dat"
file_2="trapData.dat"
file_3="avgErrdata.dat"

if [ -f $file_1 ] ; then
    rm $file_1
fi

i=1
while [[ i -le 30 ]]
do
    ./monteCarlo $((2**i)) >> data.dat
    ((i = i + 1))
done

#trap data
if [ -f $file_2 ] ; then
    rm $file_2
fi

i=1
while [[ i -le 30 ]]
do
    ./trap $((2**i)) >> trapData.dat
    ((i = i + 1))
done


# Script for Rerun

if [ -f $file_3 ] ; then
    rm $file_3
fi


for (( i=1; i<=30; i++ ))
do
  for (( j=1; j<=200; j++ ))
  do
    ./monteCarlo $((2**i)) >> avgErrdata.dat
  done
done

python3 plot.py
```

Listing 3: Bash script **dataScript.sh** for execution and data generation in Problem d, e, f

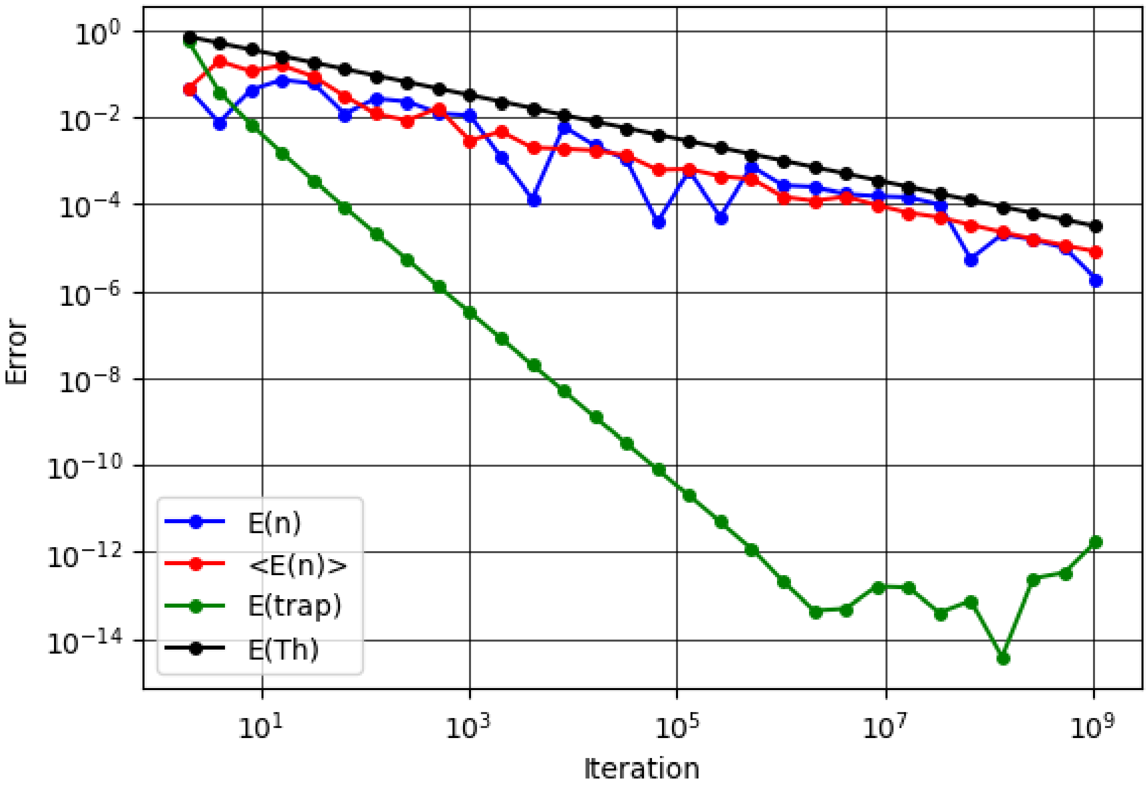

Figure 1: Numerical Integration Err vs Iteration: Problem d, e, f

Problem d, e, f

Solution

Here in the figure 1, the Error vs Iteration is being plotted. The sequence of $N = 2^i$ for integers $i = 1, 2, 3, \cdots, 30$ used as input. The theoretical error of the Monte Carlo integration is $E \propto N^{-1/2}$.
Here, the experiment run 200 times and average error is being calculated.

Problem e Solution: Here the error behaves like:

$$logE(N) = B + A \times log(N)$$

If we assume $B = 0$ (since it is a constant), using the linear regression analysis, it is found that:

$$A = -0.6049353724290725$$

But, it should be equal to -0.5. Since, several errors like rounding error and random number generation contributes error here.

Problem g

Solution

The execution time taken by **time.sh**.

```
#!/bin/bash
g++ monteCarlo_time.cpp -o monteCarlo_time
#$ TIMEFORMAT=%e
file_2="time.dat"
if [ -f $file_2 ] ; then
    rm $file_2
```

```
fi

for (( i=1; i<=30; i++ ))
do
  /usr/bin/time -f "%e"  ./monteCarlo_time $((2**i)) &>> time.dat
     #( time (./monteCarlo_time $((2**i)))) &>> time.dat
done
python3 timePlot.py
```

Listing 4: **time.sh**: For execution and data generation in Problem g

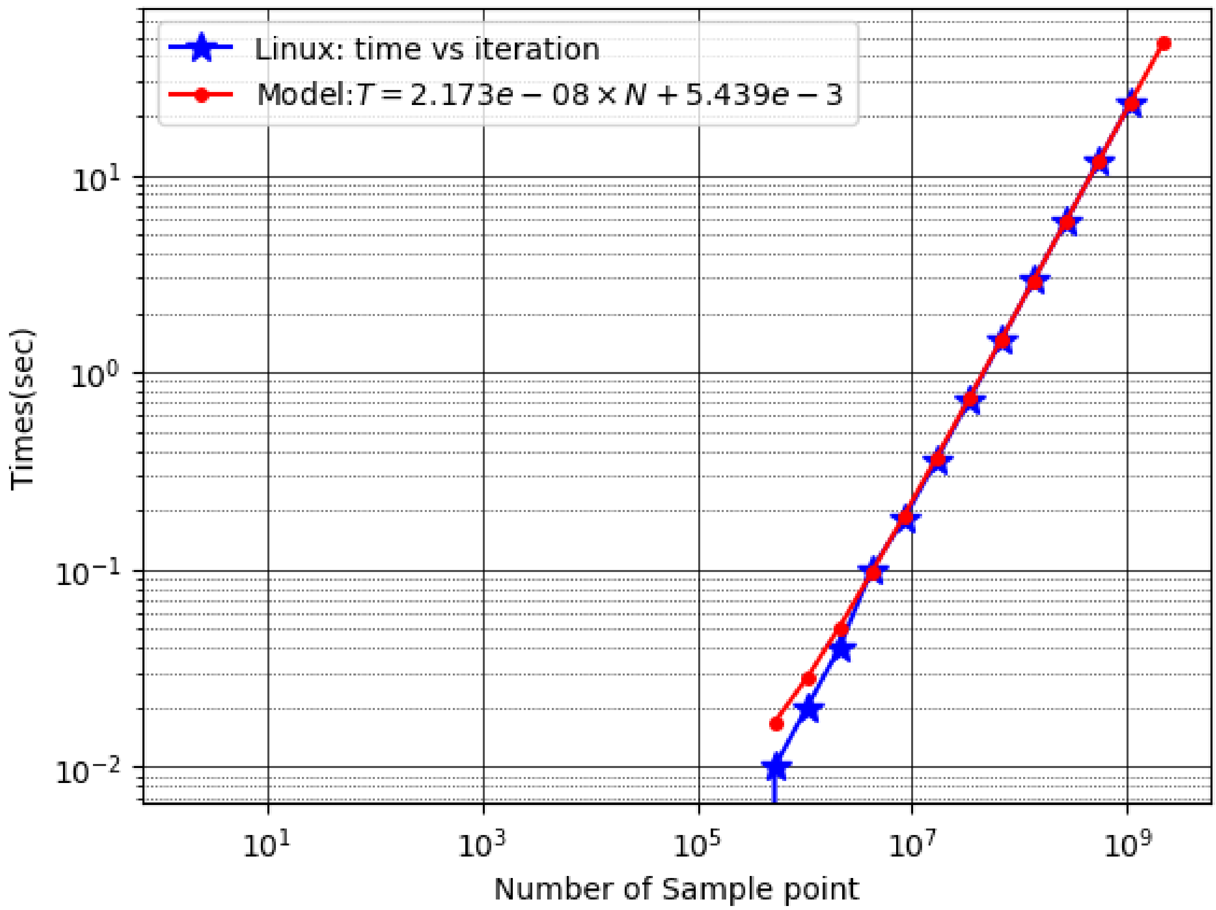

Figure 2: Time vs Iteration: Problem g

The run time is calculated by the Linux *time* command. For first 18 iteration, we got zero (0). Perhaps, CPU take some time in 'nano' second scale. So, *time* command ignored that. The time data is plotted in 2. We have seen that the relation between iteration and time is linear. From the data, the best fitted model for time is:

$$T = 2.17309336 \times 10^{-8} \times N + 5.4391834667749 \times 10^{-3}$$

where, T is time in second and N is number of iteration. Using this formula as well as using linear regression, it is found that time for $N = 2^{32}$ is 93.339 seconds.