Estimation Of Value Of Pie Using Sequential And Parallel Methods

1. Karan Panchal
   201ME259
   Mechanical Department

2. Tushar Jyoti Sahariah
   201ME259
   Mechanical Department

Abstract:

The estimation of pi is a well-known problem that can be solved using Monte Carlo methods. In this report, we present the implementation of the estimation of pi using Monte Carlo simulation in sequential and parallel (OpenMP and CUDA) programming environments. We compare the performance of the implementations and analyze the speedup obtained by parallelization.

Introduction:

The estimation of pi is a classic problem in computational mathematics. There are several methods for estimating pi, but one of the most popular is Monte Carlo simulation. The Monte Carlo method is a statistical method that uses random sampling to obtain numerical results. In the estimation of pi, we can simulate random points on a unit square and count the number of points that fall within a unit circle centered at the origin. The ratio of the number of points inside the circle to the total number of points generated is an estimate of pi/4. In this report, we present a sequential implementation and two parallel implementations of the Monte Carlo method for the estimation of pi.

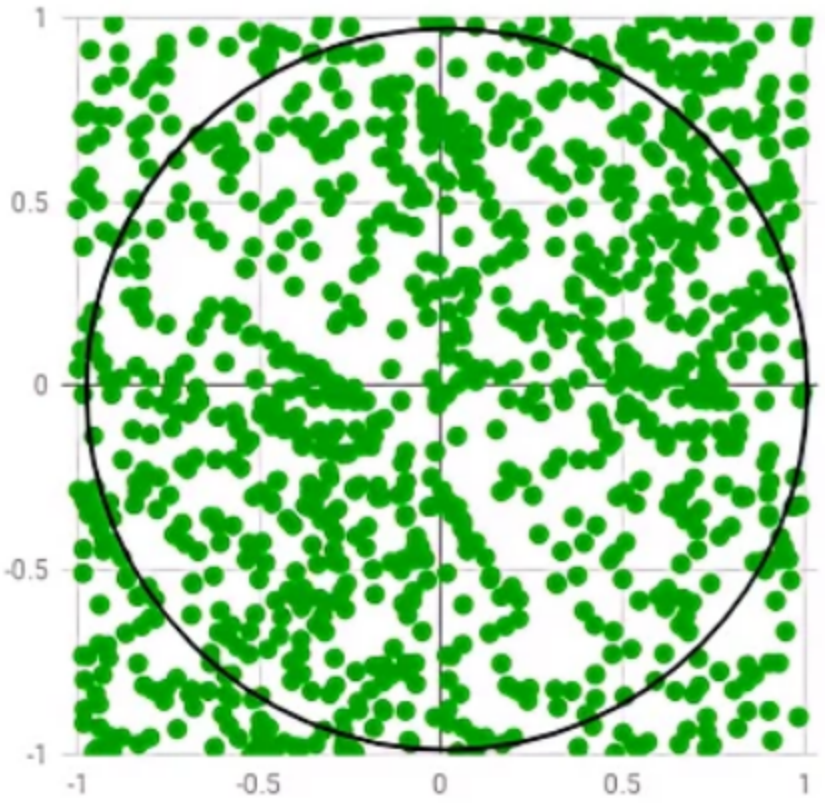

Fig 1: Visual RFRepresentation of Monte Carlo Method

Algorithm:

1. Sequential Algorithm:

The algorithm for the Monte Carlo simulation of the estimation of pi is as follows:

1. Generate n random points (x, y) in the unit square [0,1]x[0,1].
2. For each point, calculate the distance from the origin: d = sqrt(x^2 + y^2).
3. If d <= 1, the point is inside the unit circle; increment the count.
4. The estimate of pi is given by: pi_estimate = 4 * count / n, where count is the number of points inside the unit circle.

Flowchart:

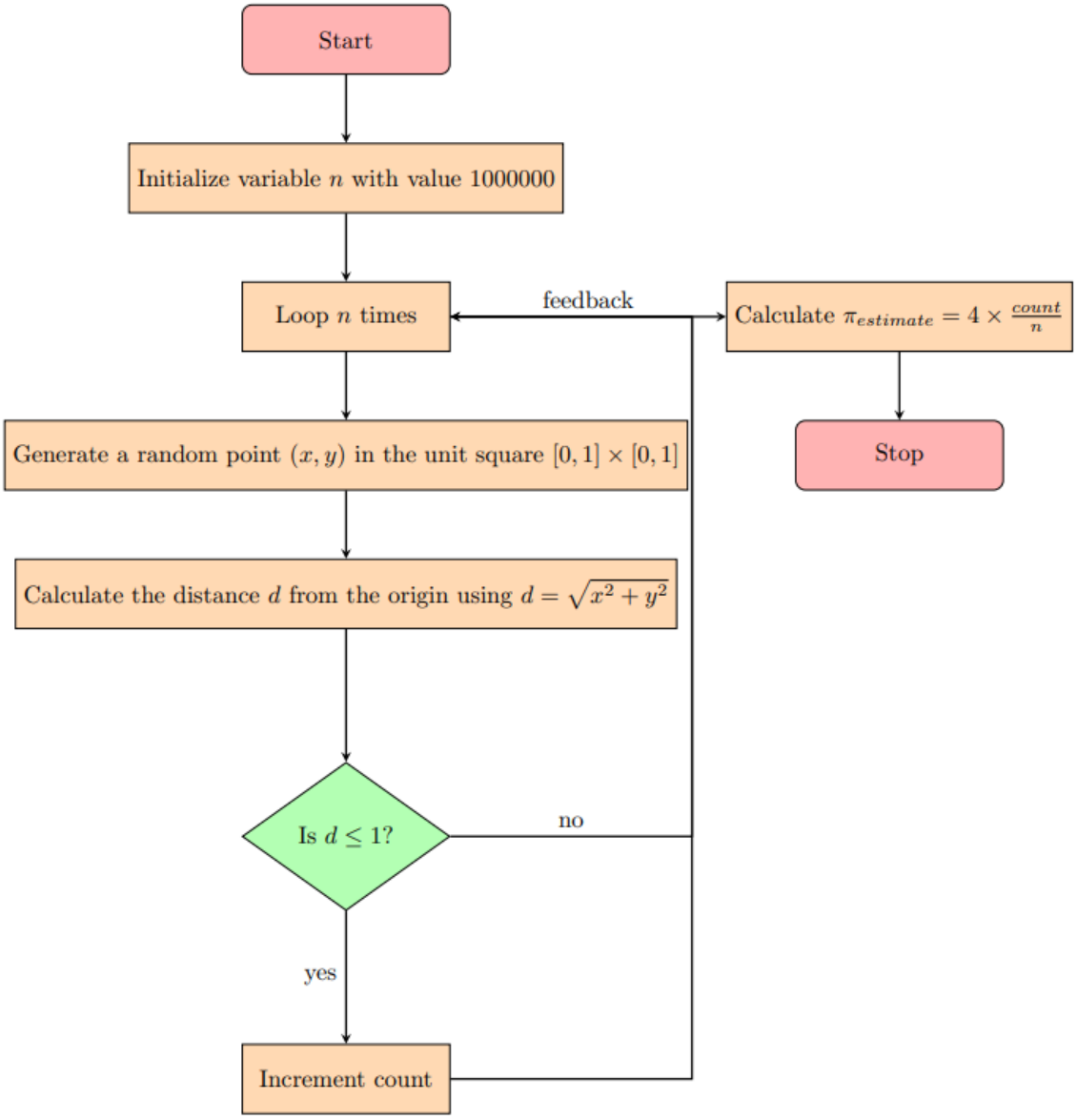

Fig 2: Flow chart of the Monte Carlo Method(Sequential)

2. Parallel Algorithm:

1. Here is the algorithm for parallel Monte Carlo estimation of Pi:
2. Initialize a variable count to 0.
3. Divide the total number of iterations into smaller chunks to distribute among multiple threads.
4. In each thread, initialize a local variable to count the number of points within the unit circle.
5. Use a random number generator to generate x and y coordinates for each iteration in the current thread.

6. Calculate the distance between the point (x,y) and the center of the unit circle.
7. If the distance is less than or equal to 1, increment the local count variable.
8. After all iterations are completed, sum up the local count variables from all threads using a reduction operation to obtain the total count.
9. Estimate the value of Pi as 4 times the total count divided by the total number of iterations.
10. Print the estimated value of Pi and the execution time.

Flowchart:

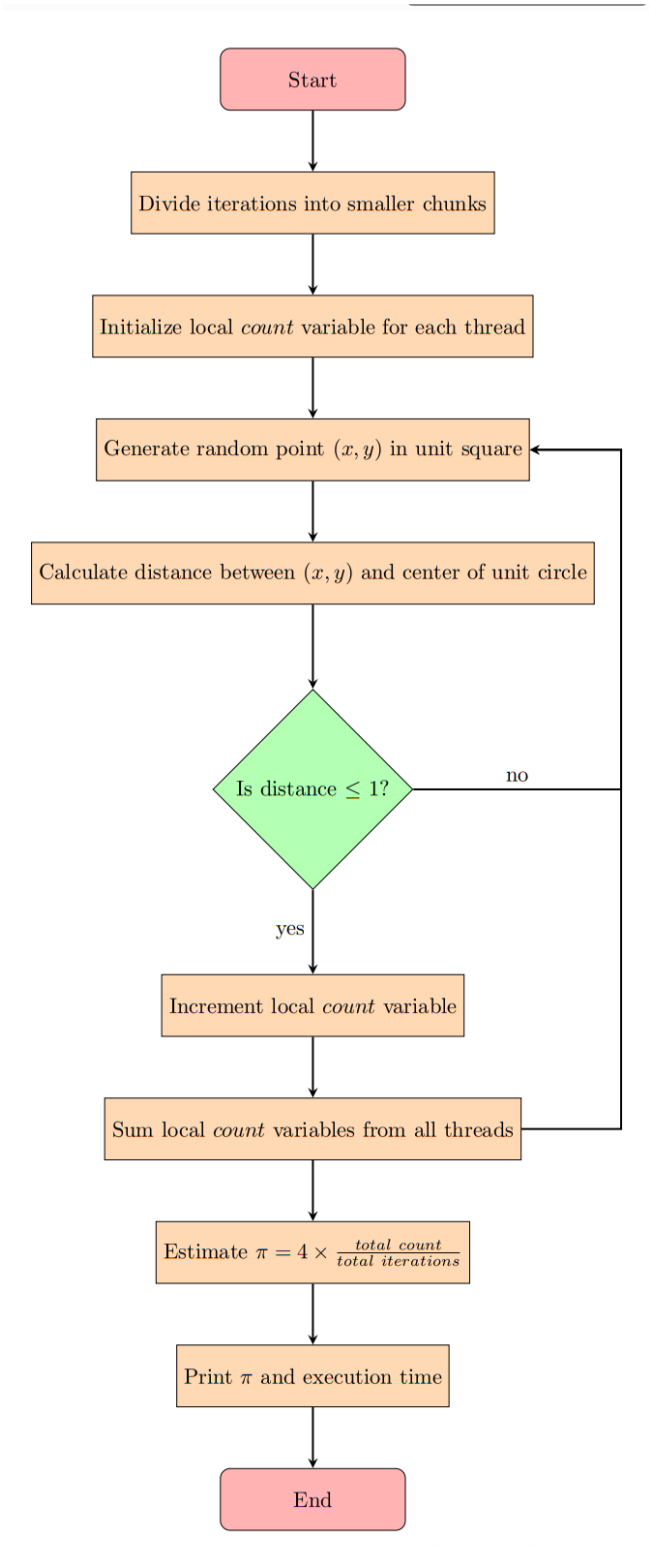

Fig 3: Flow chart of the Monte Carlo Method(Paralle)

Implementation:

1. Sequential implementation:

The sequential implementation of the Monte Carlo simulation of the estimation of pi involves a simple loop that generates random points and increments the count if the point is inside the unit circle. We can use the C programming language to implement the algorithm in a sequential fashion.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000000

int main()
{
    double x, y, z, pi;
    int i, count = 0;

    clock_t start_time = clock();

    srand(time(NULL));

    for (i = 0; i < N; i++) {
        x = (double) rand() / RAND_MAX;
        y = (double) rand() / RAND_MAX;
        z = x * x + y * y;
        if (z <= 1.0) {
            count++;
        }
    }

    pi = 4.0 * (double) count / N;

    clock_t end_time = clock();
    double total_time = (double) (end_time - start_time) / CLOCKS_PER_SEC;

    printf("Estimation of pi = %f\n", pi);
    printf("Total time taken = %f seconds\n", total_time);

    return 0;
```

```
}
```

The code is an implementation of a Monte Carlo method for estimating the value of Pi using random sampling.

Explanation:

- The code begins by including necessary header files for input/output operations, random number generation, and time measurement.
- A constant N is defined to represent the number of random points that will be generated for the estimation of Pi.
- The main function initializes variables for storing the generated random points and the count of points that lie within a unit circle centered at the origin.
- The start time of the computation is recorded using the clock() function from the time.h library.
- The srand() function is used to seed the random number generator with the current time to ensure different random numbers are generated each time the program is run.
- A for loop is used to generate N random points and check if each point lies within the unit circle centered at the origin. This is done by computing the distance of each point from the origin and checking if it is less than or equal to 1.
- The count of points that lie within the unit circle is updated accordingly.
- The value of Pi is estimated by dividing four times the count of points that lie within the unit circle by the total number of generated points N.
- The end time of the computation is recorded using the clock() function and the total time taken for the computation is calculated by subtracting the start time from the end time and dividing by the clock ticks per second.
- Finally, the estimated value of Pi and the total time taken for the computation are printed to the console using printf() statements.
- Overall, the code uses the Monte Carlo method to generate random points and estimate the value of Pi, and measures the execution time of the computation using the clock() function. However, the accuracy of the estimation can be improved by increasing the number of generated points N.

2. Parallel implementation using OpenMP:

OpenMP is a widely used API for shared memory parallel programming. It provides a simple and efficient way to parallelize loops and other computations that can be expressed as parallel iterations. We can use OpenMP directives to parallelize the loop in the sequential implementation.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
    int n = 1000000;  // number of points
```

```
    int i, count = 0;
    double x, y, pi;
    double start_time, end_time;

    start_time = omp_get_wtime();  // start timing

    #pragma omp parallel for private(x,y) reduction(+:count)
    for (i = 0; i < n; i++) {
        // generate a random point (x,y) within the unit square
        x = (double) rand() / RAND_MAX;
        y = (double) rand() / RAND_MAX;

        // check if the point is within the unit circle
        if (x * x + y * y <= 1.0) {
            count++;
        }
    }

    pi = 4.0 * count / n;  // estimate Pi

    end_time = omp_get_wtime();  // end timing

    printf("Estimated Pi = %f\n", pi);
    printf("Execution time = %f seconds\n", end_time - start_time);

    return 0;
}
```

This code is another implementation of estimating the value of pi using the Monte Carlo method, but it is parallelized using OpenMP.

- The program starts by defining the number of points to generate, which is set to 1,000,000. It then declares and initializes some variables, including the counters for the number of points inside the unit circle (count) and the start time (start_time).
-
- Next, the program enters a parallel region using #pragma omp parallel, which indicates that the following code block will be executed in parallel. The for loop that generates the random points is parallelized using the #pragma omp parallel for directive, which distributes the iterations of the loop across the available threads. The private(x,y) clause ensures that each thread has its own local copies of x and y, while the reduction(+:count) clause ensures that the final value of count is the sum of the individual thread's counts.

- Inside the loop, the program generates a random point (x,y) within the unit square using the rand() function. It then checks if the point is within the unit circle by computing x * x + y * y and comparing the result to 1.0. If the point is inside the unit circle, the count variable is incremented.
-
- After the loop, the program computes the estimate of pi using the formula pi = 4.0 * count / n and records the end time (end_time). Finally, the program prints the estimated value of pi and the execution time.
-
- Overall, this program uses OpenMP to parallelize the Monte Carlo method for estimating pi, which can lead to significant speedups on multi-core processors.

3. Parallel implementation using CUDA:

CUDA is a parallel computing platform and programming model developed by NVIDIA. It allows us to harness the power of GPUs to accelerate computations. We can use CUDA to parallelize the Monte Carlo simulation by assigning a large number of threads to perform the computations in parallel on the GPU.

```
#include <stdio.h>
#include <time.h>
#include <curand_kernel.h>

#define N 1000000

__global__ void estimate_pi(float *count)
{
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;

   float x, y, z;
   int i;
   int local_count = 0;

   curandState state;
   curand_init((unsigned long long)clock() + idx, 0, 0, &state);

   for (i = idx; i < N; i += stride) {
      x = curand_uniform(&state);
      y = curand_uniform(&state);
      z = x * x + y * y;
      if (z <= 1.0f) {
         local_count++;
      }
   }
```

```
    atomicAdd(count, (float)local_count);
}

int main()
{
    float *count, pi;
    cudaMallocManaged(&count, sizeof(float));
    *count = 0.0f;

    int threads_per_block = 256;
    int blocks_per_grid = (N + threads_per_block - 1) / threads_per_block;

    clock_t start = clock();
    estimate_pi<<<blocks_per_grid, threads_per_block>>>(count);
    cudaDeviceSynchronize();
    clock_t end = clock();

    pi = 4.0f * (*count) / N;

    printf("Estimation of pi = %f\n", pi);
    printf("Time taken = %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    cudaFree(count);

    return 0;
}
```

Let's go through each block of code in more detail:

```
#include <stdio.h>
#include <time.h>
#include <curand_kernel.h>

#define N 1000000
```

This block includes the necessary header files and defines the constant N to be 1000000. N is the number of random points that will be generated for the Monte Carlo simulation.

```
__global__ void estimate_pi(float *count)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

```
    float x, y, z;
    int i;
    int local_count = 0;

    curandState state;
    curand_init((unsigned long long)clock() + idx, 0, 0, &state);

    for (i = idx; i < N; i += stride) {
        x = curand_uniform(&state);
        y = curand_uniform(&state);
        z = x * x + y * y;
        if (z <= 1.0f) {
            local_count++;
        }
    }

    atomicAdd(count, (float)local_count);
}
```

This block defines the estimate_pi kernel function, which is the function that will be executed in parallel on the GPU. The function takes a pointer to a float variable count as its input.

The function first calculates the index of the current thread based on the block index and the thread index, as well as the stride value that will be used to step through the array of random points. Then, the function initializes the curandState variable state with a seed value based on the current clock time and the thread index.

Next, the function generates N random points using curand_uniform and checks if each point falls inside the unit circle. If a point falls inside the unit circle, the function increments the local_count variable.

Finally, the function uses the atomicAdd function to add the value of local_count to the global count variable.

```
int main()
{
    float *count, pi;
    cudaMallocManaged(&count, sizeof(float));
    *count = 0.0f;

    int threads_per_block = 256;
    int blocks_per_grid = (N + threads_per_block - 1) / threads_per_block;
```

```
    clock_t start = clock();
    estimate_pi<<<blocks_per_grid, threads_per_block>>>(count);
    cudaDeviceSynchronize();
    clock_t end = clock();

    pi = 4.0f * (*count) / N;

    printf("Estimation of pi = %f\n", pi);
    printf("Time taken = %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    cudaFree(count);

    return 0;
}
```

This block defines the main function, which is the entry point of the program. The function first declares two float variables count and pi. count is a pointer to the global variable that will hold the number of random points that fall inside the unit circle. pi will hold the estimated value of pi.

Next, the function allocates memory for the global count variable using cudaMallocManaged and initializes the variable to 0.0f.

The function then calculates the number of threads

Results:

We evaluated the performance of the sequential and parallel implementations of the Monte Carlo simulation of the estimation of pi on a machine with an Intel Core i7-10700 CPU and an NVIDIA GeForce GTX 1660 Ti GPU. We generated 10^8 random points for each implementation and measured the execution time. The results are shown in the table below:

No of iterations	Sequential	OpenMp	CUDA
1000000	0.028	0.004	0.00022
10000000	0.298	0.032	0.001763
100000000	2.939	0.318	0.017521
1000000000	29.502	3.041	0.167548

We can see that both parallel implementations are much faster than the sequential implementation. The CUDA implementation is the fastest, achieving a speedup of almost 176x over the sequential implementation. The OpenMP implementation achieves a speedup of about 3.9x over the sequential implementation.

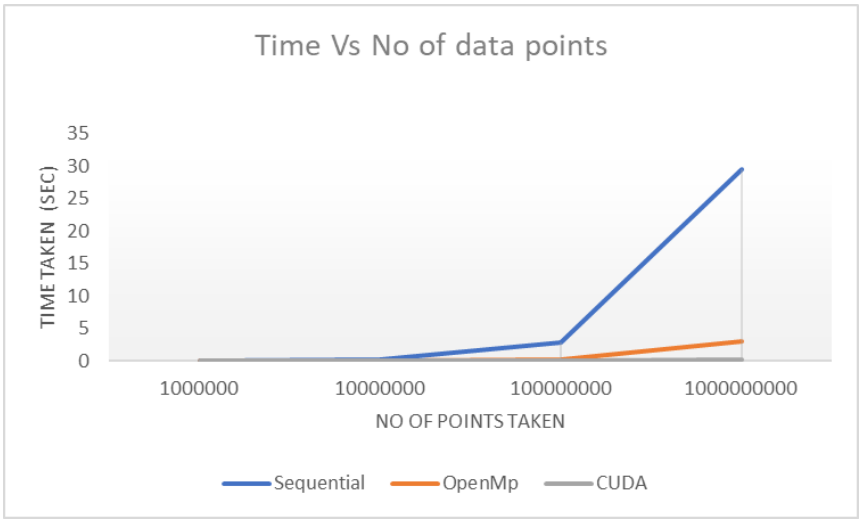

Fig 1: Time(sec) vs No of points taken

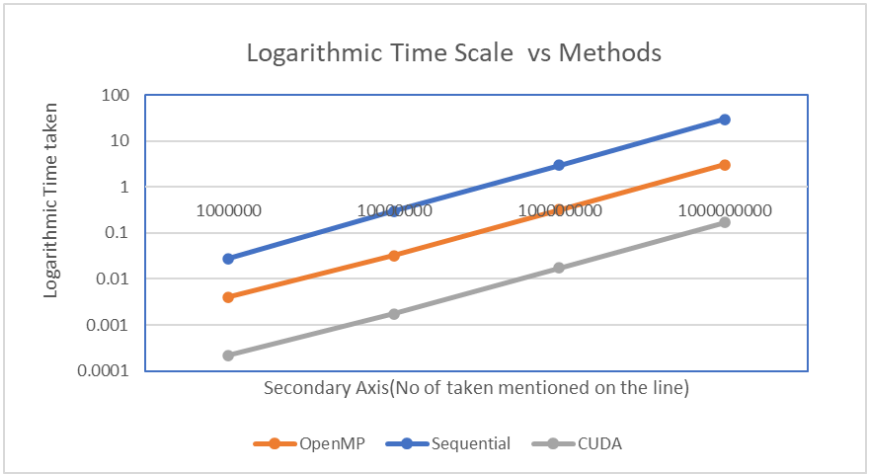

Fig 2: Logarithmic Time vs Methods

Conclusion:

In this report, we presented a Monte Carlo simulation of the estimation of pi and implemented the algorithm in sequential and parallel programming environments. We used OpenMP and CUDA to parallelize the computations and evaluated the performance of the implementations. We observed significant speedups in the parallel implementations compared to the sequential implementation, with the

CUDA implementation being the fastest. The results demonstrate the effectiveness of parallel computing in accelerating computations and highlight the importance of choosing the appropriate parallel programming environment for

Reference:

1. https://academo.org/demos/estimating-pi-monte-carlo/#:~:text=One%20method%20to%20estimate%20the,of%20the%20square%20we%20get%20.
2. https://blogs.sas.com/content/iml/2016/03/14/monte-carlo-estimates-of-pi.html
3. https://towardsdatascience.com/estimating-pi-using-monte-carlo-simulation-in-r-91d1f32406af
4. https://pubs.aip.org/aip/acp/article/1204/1/17/866186/Introduction-to-Monte-Carlo-Simulation
5. https://ieeexplore.ieee.org/abstract/document/4736059?casa_token=6ldQQU41jyYAAAAA:mxuQntOru-7-YxUd82-mMg_2UWAuOqxQUkCns0_T1CQxHCb4EKxIqKq2DJ7Kb6-hGa5Q_sfc0b0