CS 115 Final Assignment:
Monte Carlo vs. Cube Integration

Doan, Kenneth (19469742)

University of California, Irvine

Project Assumptions

This project was tested on a system with the following specs:

- Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz
- 8.00 GB RAM
- Windows 10 - 64-bit
- Python 3.7.2 (32bit)

The project is separated into multiple files with *monte.py* and *cubeintegration.py* containing the methods for integration of the n-dimensional sphere.

This project also have the following dependencies included in order to run:

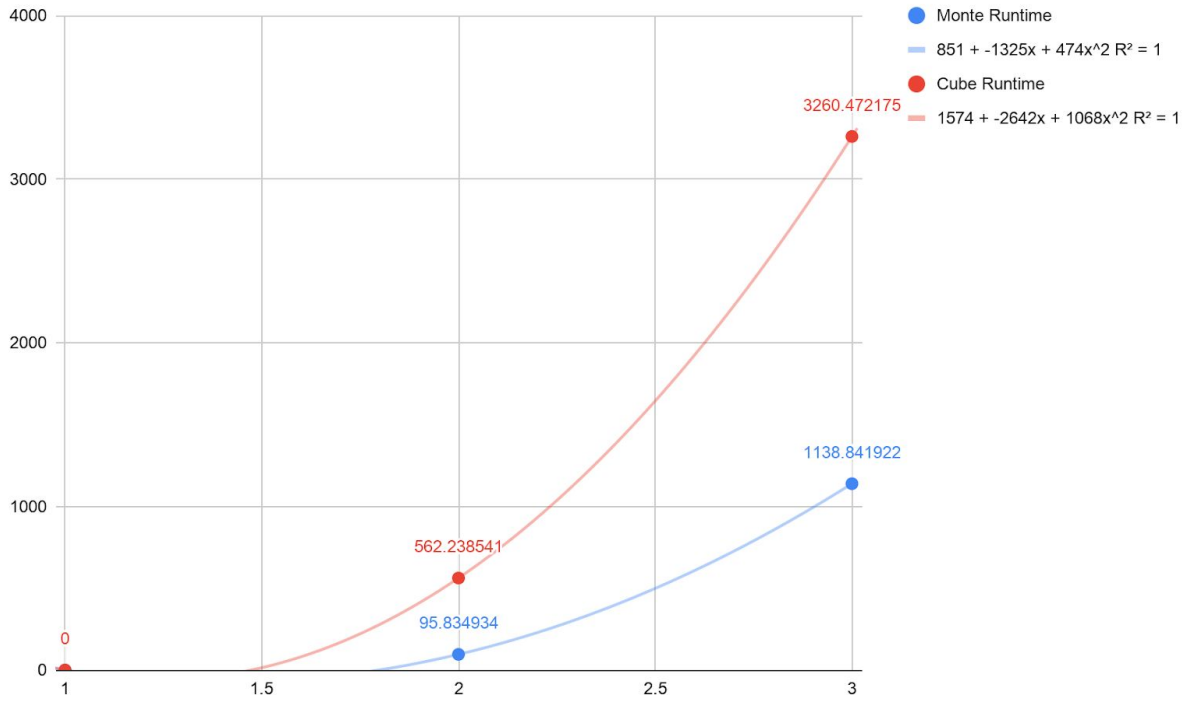
- numpy - 1.17.4
    - Package that allows for better calculations in array type points
    - *pip install numpy*
- scipy - 1.3.3
    - Package that contains some logic for statistical analysis
    - *pip install scipy*

main.py contains all the functions that a user would use, commented out as needed.
The default number of points generated in the Monte Carlo method is 250,000 points.

T**his project makes use of multiprocessing.** The Monte Carlo method makes use of multiprocessing with **20 runs per batch**. This greatly helps to reduce runtimes in decreasing the width of the confidence interval.

Part 1: Runtimes

From the initial testing of this program, it is very easy to see that the Monte Carlo method of estimating a volume (random points) is much more efficient at estimating than the cube based method. The following table shows runtimes for both methods.



The Cube Integration method will grow much faster than the Monte Carlo method based on the cubes generated. For this method, if we section one dimension into k cut points, the number of cubes we will have is $k^{dimensions}$. This is much more expensive than the $n * num\_samples$ for the Monte Carlo method. The Monte Carlo method will NOT experience an exponential growth.

If the program demands 8 digits of precision, then the Monte Carlo method will experience a $\sqrt{N}$ growth factor. To generate the interval for the Monte Carlo method, we must use $\sqrt{N}$ in $\bar{x} \pm z\frac{S}{\sqrt{N}}$. The width of the interval will decrease based on N, however, it is not a factor of N but $\sqrt{N}$.

The runtime when dimension = 1 is ALWAYS 2*RADIUS, therefore, the runtime is about 0 seconds.

This section can be tested using:

- *monte.absolute_monte(RADIUS, DIMENSION)*
- *cubeintegration.absolute_cube(RADIUS, DIMENSION, K_VALUE)*

Part 2: Method Differences

In this test, we will choose N = 1,000,000. We know that if we generate 1,000,000 samples of the Monte Carlo run, we have N samples (trivial). However, when we integrate with the cube method, we use $k^{dimension}$ samples. In order to choose the same CPU cost for both, we can use:

$$k^{dimension} = N$$

$$\sqrt[dimension]{k^{dimension}} = \sqrt[dimension]{N}$$

$$k = \sqrt[dimension]{N}$$

In the case of N = 1,000,000, we will use k = 1000. For a single run, our Monte Caro run returns 3.141476 while the Deterministic method returns 3.1413119999999997.

The total difference (absolute value) is 0.00016400000000027504. This is surprisingly accurate considering a single run of the deterministic method takes quite some time. The difference for this run however, is that we only use about k = 1000 which is far off from the ~17000 sample size needed for 4 digits of precision.

Comparing this to the actual value of pi, 3.1415, both methods return a precision of 4 digits (Great!). We have to be careful of taking this as the absolute answer however. Recall in part one, we had to generate multiple runs of the Monte Carlo method to ensure a 99% confidence interval whereas here, we only generate a single run.

Below are the results of runs of dimensions [1,5]. It seems as the size of dimensions grows, Cube Difference From Actual grows further away from the “actual” value.

Dimension	Samples	Monte Result	Cube Result	Difference	Actual	Monte Difference From Actual	Cube Difference From Actual
1	1000000	2	2	0	2	0	0
2	1000000	3.14147	3.14131	0.00016	3.14159	0.000116653589	0.000280653589

		6	2	4	2654	8	8
3	1000000	4.184192	4.188096	0.003904	4.188790205	0.004598204786	0.0006942047864
4	1000000	4.93736	5.012084961	0.07472496094	4.934802201	0.002557799455	0.07728276039
5	1000000	5.253984	6.026367188	0.7723831875	5.263789014	0.009805013914	0.7625781736
6	1000000	5.163648	8.479104	3.315456	5.16771278	0.00406478005	3.31139122

Despite the single run using dimension = 3, it seems that the Monte Carlo method will return a more accurate value.

This section can be tested using:

- *assignment_tools.fixed_costs(RADIUS, DIMENSION, samples)*

Part 3: Most Accurate Value

We realized that in part 2 that the Monte Carlo method of integration is more accurate in the long term. So we will use that method in this section. Using dimension = 2 (which should equate to π), we will generate the N samples.

We can use a modified version of our 4 digit precision method to compute the answer. Here, we generate all N samples instead of running until 4 digits of precision are found.

The greatest impact that we can make (in terms of variance reduction) will be how many points are generated for each run. The more points we generate, the "closer" we get to the actual value of π. It is the multiple runs, however, that account for "how sure we are" i.e. the confidence interval. We reduce the variance this way in the fact that the individual runs will all converge to the "actual" value, therefore reducing the distance between runs.

This section can be tested using:

- *assignment_tools.most_accurate_value(RADIUS, DIMENSION, samples)*