Numerical Analysis Project | MAD3703

By: Tyler Boshaw

Section I: Statement of Problem

A comparative analysis of Monte Carlo Approximation and the Trapezoidal rule of Integration for estimating π with minimal error. The focus is to develop algorithms that reach a tradeoff between low error and minimized computation time, while evaluating their strengths and weaknesses as numerical analysis techniques.

Section II: Description of the Mathematics

For this project, I am utilizing and comparing the results of two methods, those being Monte Carlo Approximation and the Trapezoidal rule of Integration. Both methods have slightly different setups for their mathematics, thus I will break them up and review them individually. Let's get started with the Monte Carlo method.

Monte Carlo Approximation

For the Monte Carlo method, we will be utilizing a circular region within the bounds of a square that will host a random distribution of points. For points being placed within the bounded region of the circle, calculated by (1.1) below, they will be tallied into the 'inside' classification.

(1.1)

$$x^2 + y^2 \leq 1$$

Once all the points are distributed and all values inside the circular region are classified, an approximation will be calculated. This approximation is calculated by the following formula, (1.2), which compares the ratio of the area of the circular region over the area of the square region to the ratio of the number of inside points over the total number of distributed points.

(1.2)

$$\frac{Area_{circle}}{Area_{square}} = \frac{\pi}{4} \cong \frac{r}{n}$$

$$r = number\ of\ points\ inside\ circular\ region, n = number\ of\ total\ points\ distributed$$

The mathematics can then be simplified to showcase the approximation of π is equivalent to four times the ratio of inside points to the total number of points, as seen by formula (1.3) below.

(1.3)

$$\pi \cong 4 * \frac{r}{n}$$

As we will see within the results sections, we can expect the approximation to become more accurate as the total number of points that are distributed increases.

Trapezoidal rule of Integration

For the Trapezoidal rule, we will be utilizing a half circle with bounds [0, π] to measure the approximation of π, integrating over the bound with the Trapezoidal rule. Formula (1.4) below is this representation.

(1.4)

$$function = \sqrt{1 - x^2}$$

$bounds\ from\ [0, \pi]$

Then, we applied the Trapezoidal rule to integrate the function over the bound. The approximation is determined by the number of points over the interval and the step size of the intervals, ultimately then being calculated by formula (1.5) below.

(1.5)

$$approx = \int_a^b f(x)dx \cong \frac{h}{2}(y_0 + 2\sum_{i=1}^{n-1} y_i + y_n)$$

$where$

$$h = \frac{b-a}{n-1}$$

With the proper formulation to approximate π with the Trapezoidal rule of Integration, we can now build the algorithm to program it and run test cases.

Section III: Description of the Algorithm

Let's take a closer look at the implementation of the mathematics by showcasing the algorithms used for this project.

For the two methods, we begin by applying algorithms like the ones below, written in a pseudo code format.

Monte Carlo:

- *Set the total number of points being distributed*
- *Create a loop to categorize random points*
- *Within the loop, recognize those points inside the circular bound*
- *Measure the total of points within the circular region and apply approximation calculation with total number of points placed*
- *Compare to π to find error*

Snapshot of Monte Carlo Algorithm:

```python
# define the number of points for the test
num_points = 100000

# create some data to track trade-offs later
point_range = []
error_range = []
time_range = []
approx_range = []

for i in range(0, num_points, 1):
    # define x any y, creating a circular coordinate system for random points
    # x and y are give "num_points" random values, and these points are then
    # distributed within a square region with x and y coordinates within [-1,1]
    x = np.random.uniform(-1, 1, num_points)
    y = np.random.uniform(-1, 1, num_points)

    # distinguish the points inside the circular region
    # take sum of the inside points for approximation
    bound = ((x**2 + y**2) <= 1)
    inside = np.sum(bound)

    # approximate π for the cirle
    approx = 4 * (inside / num_points)

    # calculate error to math.pi
    error = abs(math.pi - approx)

    # calculate the time taken for approximation
    end_time = time.time()
    duration = end_time - start_time

    # append info for trade-offs (similar to .pushback in C++)
    point_range.append(i)
    error_range.append(error)
    time_range.append(duration)
    approx_range.append(approx)
```

Trapezoidal Rule:

- *Define the function to integrate upon*
- *Define the number of intervals (essentially number of points) to integrate with*
- *Create a loop to evaluate each interval as we integrate*
- *Define bounds of the integral*
- *Calculate the step size (h), and complete the calculation for the approximation*
- *Compare to π to find error*

Snapshot of Trapezoidal rule Algorithm:

```
# define number of intervals (should ultimately determine accuracy)
num_points = 1000

for i in range(2, num_points, 1):

    # define bounds for integration
    b = math.pi
    a = 0

    ## trapezoidal rule algorithm
    # interval generation
    x = np.linspace(a, b, num_points)
    # evaluating the function at each subinterval point
    y = function(x)
    # step size calculation
    h = (b - a) / (i - 1)
    # approximate the function using the trapezoidal rule
    trap = (h / 2) * (y[0] + 2 * np.sum(y[1:-1] + y[-1]))

    # approximate PI
    approx = 4 * trap

    # calculate error to math.pi
    error = abs(math.pi - approx)

    # calculate the time taken for approximation
    end_time = time.time()
    duration = end_time - start_time

    # append info for trade-offs (similar to .pushback in C++)
    point_range.append(i)
    error_range.append(error)
    time_range.append(duration)
    approx_range.append(approx)
```

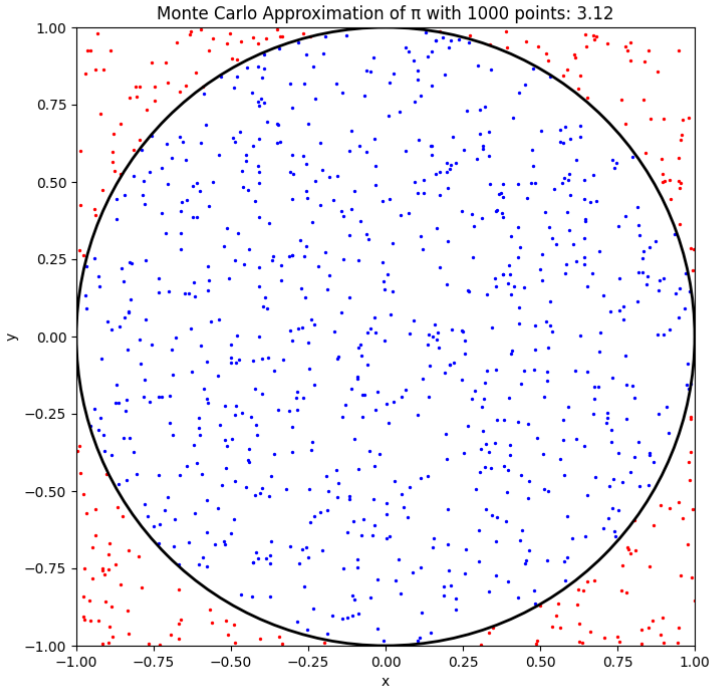
Section IV: Results

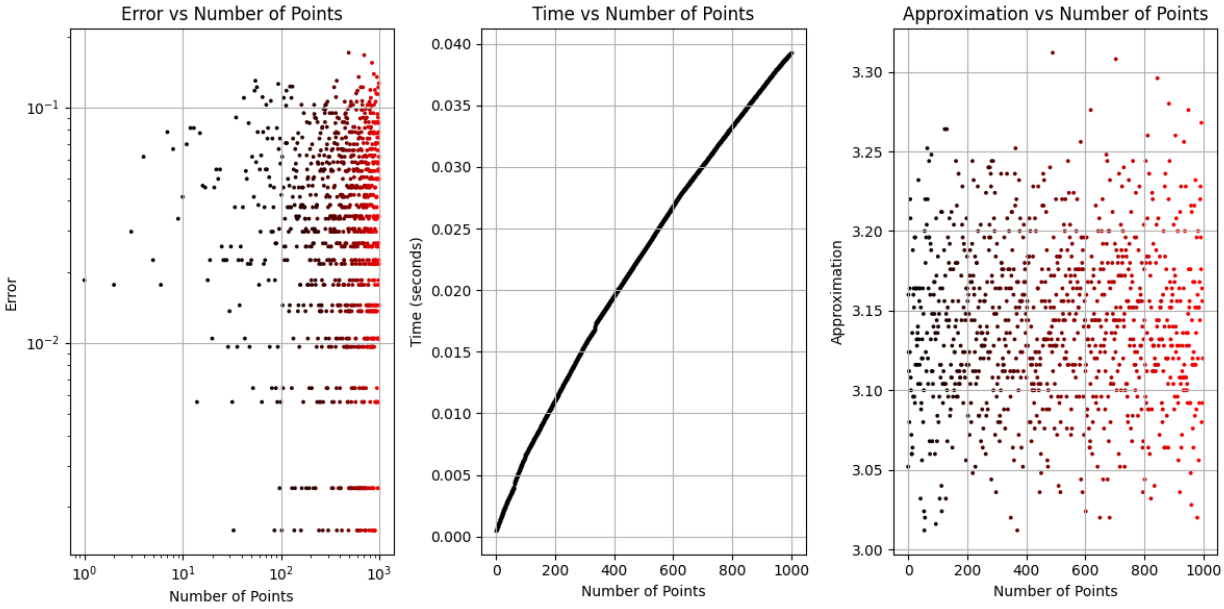
For the results, we will run a few tests for each, comparing them, signifying their strengths and weaknesses. I will include individual run results and charts/graphs for visual purposes.

Monte Carlo Test Runs:

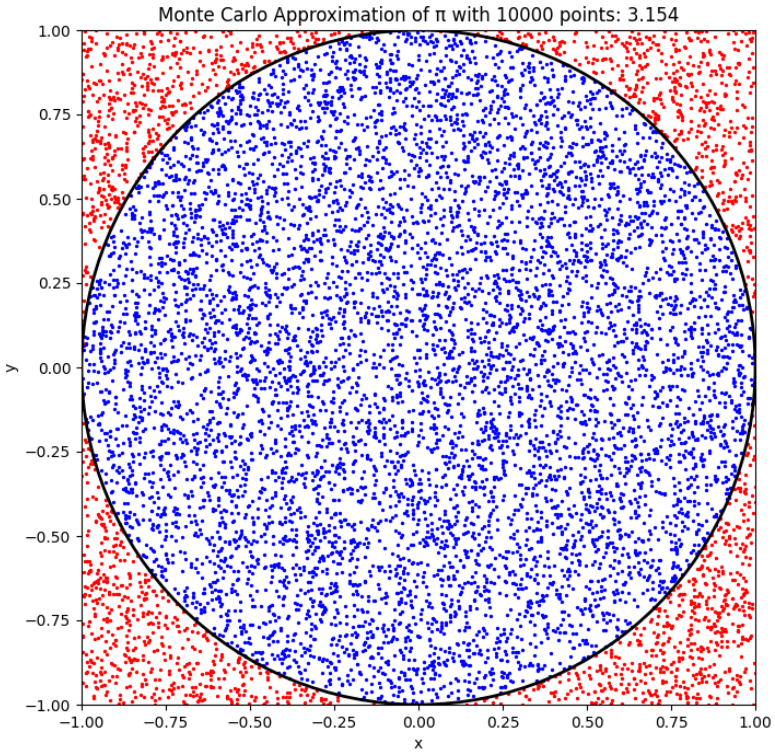
(I have them on their own pages below for clarity purposes.)

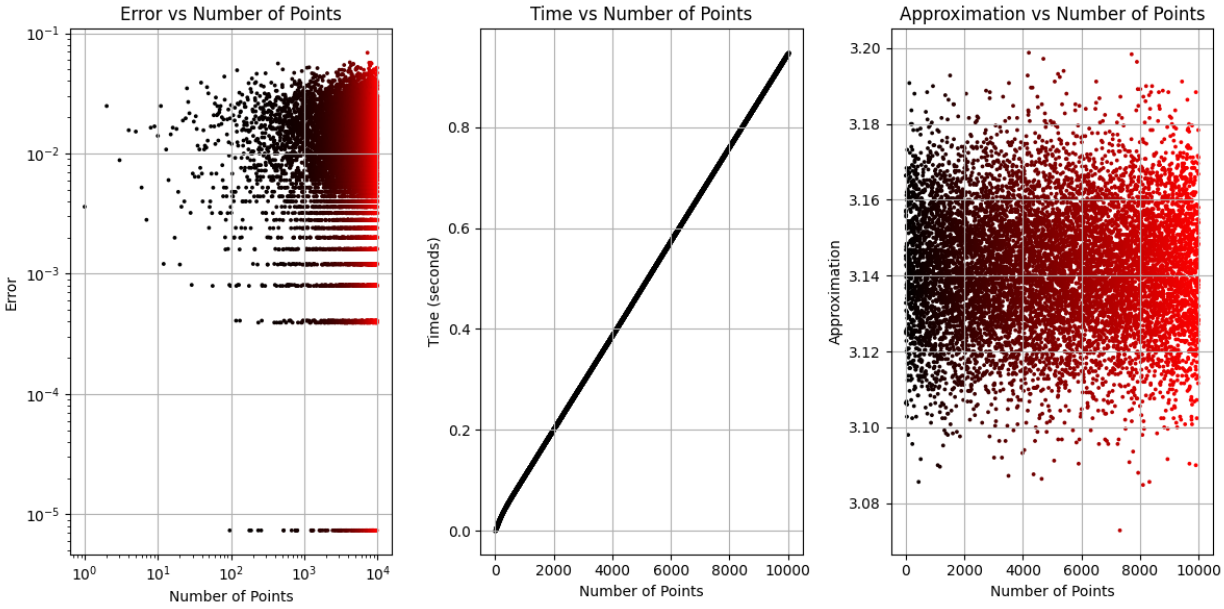
- Number of Points Distributed = 1000
- Approximation = 3.12
- Error = 0.02159265358979301
- Duration = 0.03927302360534668 seconds



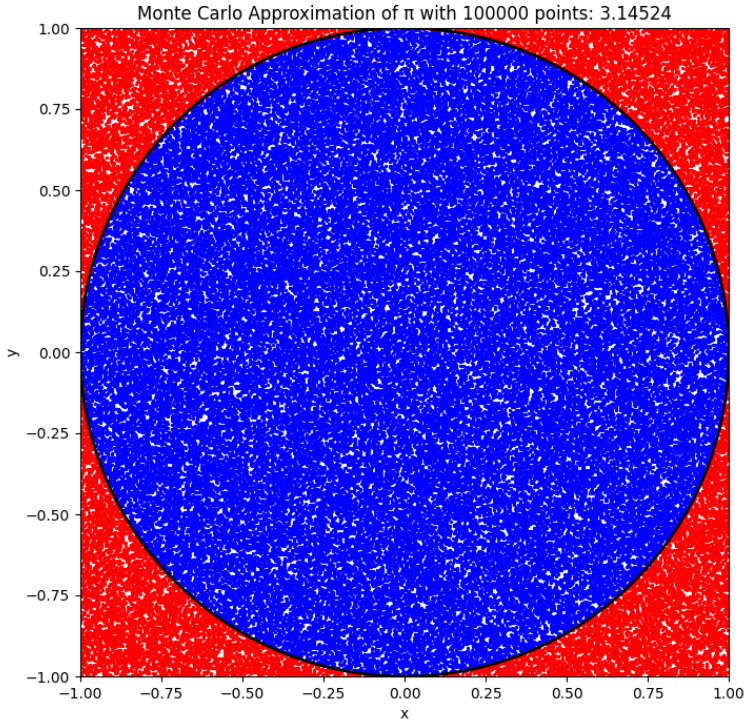


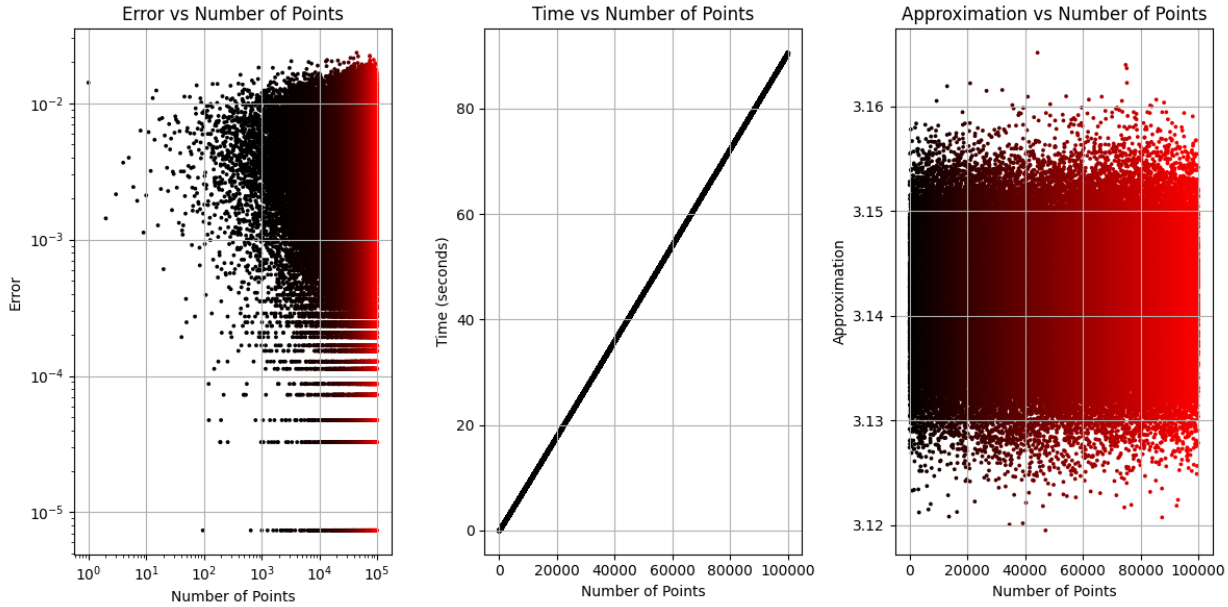
- Number of Points Distributed = 10000
- Approximation = 3.154
- Error = 0.012407346410206799
- Duration = 0.9477250576019287 seconds





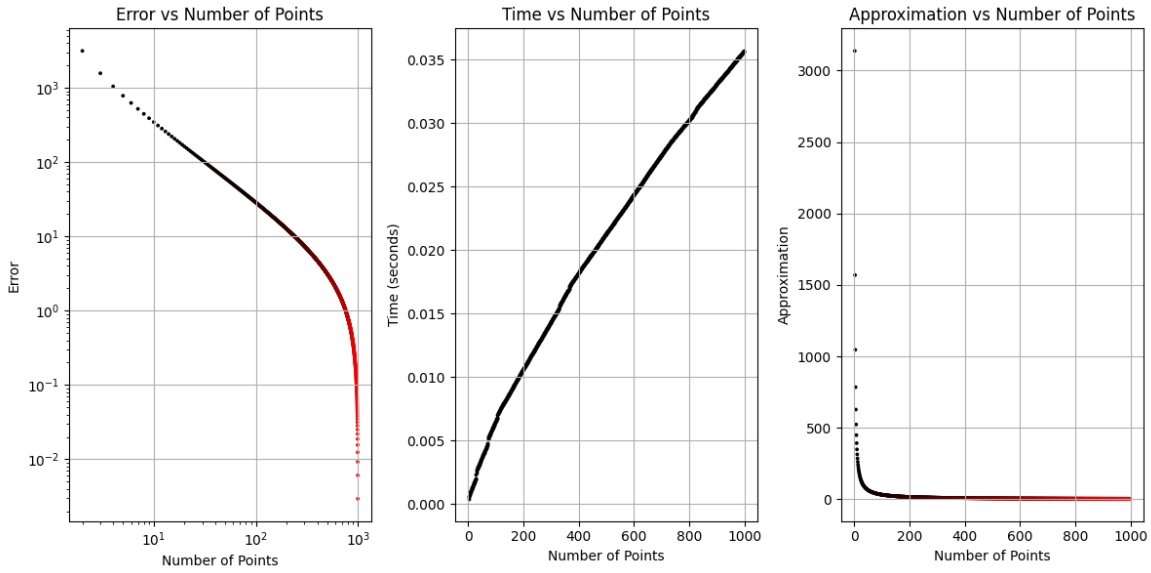
- Number of Points Distributed = 100000
- Approximation = 3.14524
- Error = 0.0036473464102066977
- Duration = 90.57041096687317 seconds



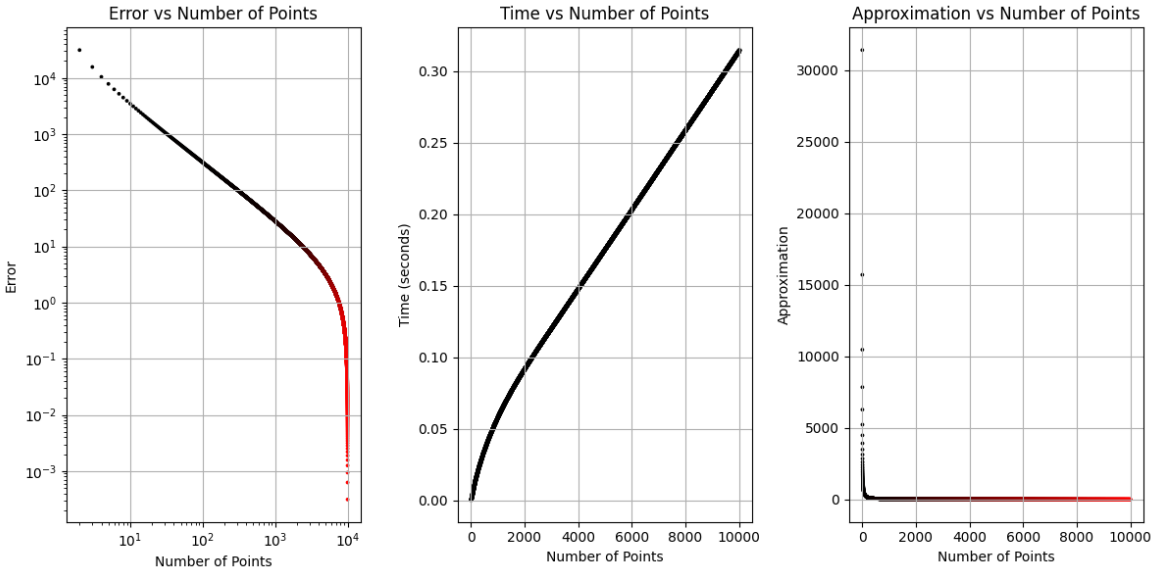


Trapezoidal rule Test Runs:

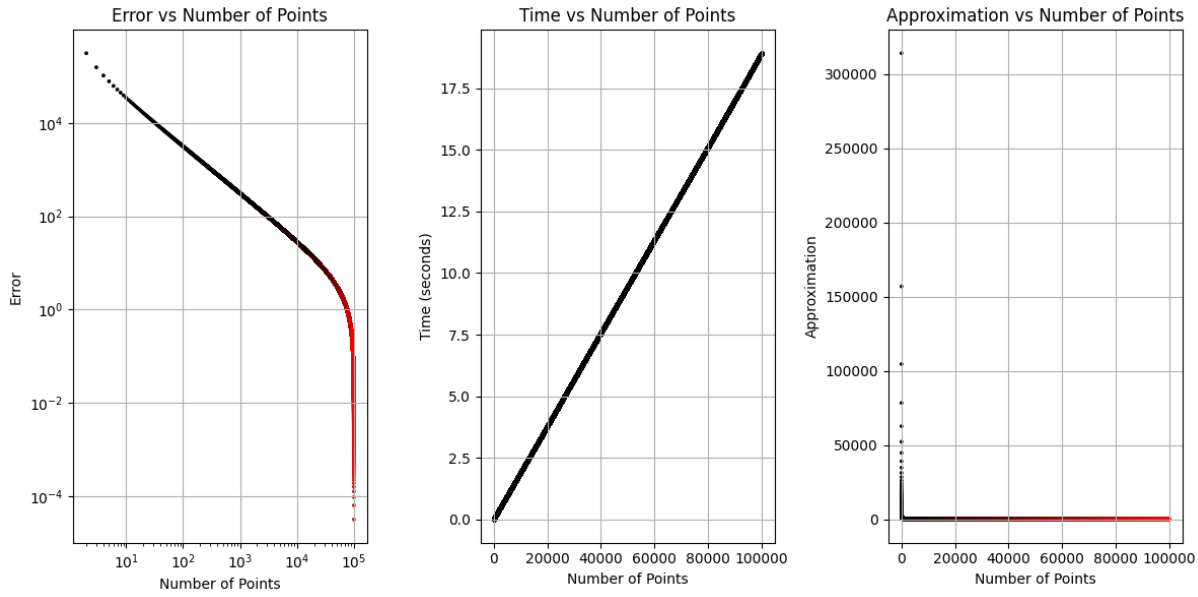
- Number of Intervals = 1000
- Approximation = 3.144539086147047
- Error = 0.0029464325572536865
- Duration = 0.03570199012756348 seconds



- Number of Intervals = 10000
- Approximation = 3.141904847178118
- Error = 0.000312193588325016
- Duration = 0.31470727920532227 seconds



- Number of Intervals = 100000
- Approximation = 3.141624063848107
- Error = 3.141025831387978e-05
- Duration = 18.916288137435913 seconds



Section V: Conclusions

From our results, we can clearly see that the Trapezoidal rule performs exceptionally better, offering better precision with a faster calculation. An important note to point out is that the Monte Carlo method is based on a random distribution, which returns distributions of the approximation and error, as seen in the Monte Carlo figures. This is what causes the "clump" of points in the error plots.

The Monte Carlo approximation may be worse for simple problems such as the one shown here, however, it is very flexible to irregular boundaries, and even offers the ability of working in multi-dimensional spaces. This means, we could essentially transform the circular region used into a 3-dimensional region, allowing us to work with a sphere. The Monte Carlo method is limited though, as the random distribution and its slow convergence rate tends to make it not as efficient as other methods. In this simpler problem, this tends to be the driving factor between the two methods.

The Trapezoidal rule on the other hand offers a quick convergence rate, making it optimal for smooth functions and simple problems. That can easily be seen by the quick and accurate results generated in the test runs. Unlike the Monte Carlo method though, it does not work very well with high-dimensional problems and tends to also fail when the functions are not smooth, likely where there may be discontinuities or sharp variations.

Overall, it was interesting to see the two methods in action, building their algorithms, and showcasing their strengths and weaknesses. This project has allowed me to truly understand the

importance of numerical analysis techniques, offering me insight on how to better use it in my daily life, whether that be for work or school.

I did want to include some references for these two methods, which I used to help me build my program and better understand the methods. I will link them down below.

References:
https://en.wikipedia.org/wiki/Trapezoidal_rule
https://en.wikipedia.org/wiki/Monte_Carlo_integration

Section VI: Program Listing

```
# Course: Numerical Analysis / MAD3703
# Project: Comparing Monte Carlo Approximation and Trapezoidal Rule
Approximation of π.
# Name: Tyler Boshaw
# Draft Due Date: Nov 22, 2024
# Final Due Date: Dec 6, 2024

# ------------------------------------------------------------------------
--- #
# Imports
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import math
import time

# ------------------------------------------------------------------------
--- #
## Technique - 1 / Monte Carlo Approximation
def monte_carlo():
    # start time (for measuring speed of approximation)
    start_time = time.time()

    # define the number of points for the test
    num_points = 100000

    # create some data to track trade-offs later
    point_range = []
    error_range = []
```

```
    time_range = []
    approx_range = []

    for i in range(0, num_points, 1):
        # define x any y, creating a circular coordinate system for random
points
        # x and y are give "num_points" random values, and these points
are then
        # distributed within a square region with x and y coordinates
within [-1,1]
        x = np.random.uniform(-1, 1, num_points)
        y = np.random.uniform(-1, 1, num_points)

        # distinguish the points inside the circular region
        # take sum of the inside points for approximation
        bound = ((x**2 + y**2) <= 1)
        inside = np.sum(bound)

        # approximate π for the cirle
        approx = 4 * (inside / num_points)

        # calculate error to math.pi
        error = abs(math.pi - approx)

        # calculate the time taken for approximation
        end_time = time.time()
        duration = end_time - start_time

        # append info for trade-offs (similar to .pushback in C++)
        point_range.append(i)
        error_range.append(error)
        time_range.append(duration)
        approx_range.append(approx)

    # results with speed
    print(f"Monte Carlo Approximation of π with {num_points} points:
{approx}")
    print(f"Speed of Approximation: {duration} seconds")
    print(f"Error of Approximation: {error}")
```

```
    # === === === === === === === === === === === === === === === === ===
#

    ## Visualization of Approximation
    # create a plot for the data points to visual the approximation
    plt.figure(figsize=(8,8))
    plt.scatter(x[bound], y[bound], color="blue", s=2)
    plt.scatter(x[~bound], y[~bound], color="red", s=2)

    # plot the circular boundary
    circle = plt.Circle((0, 0), 1, color="black", fill=False, linewidth=2)
    plt.gca().add_artist(circle)

    # set plot limits and labels
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(f"Monte Carlo Approximation of π with {num_points} points:
{approx}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()

    # === === === === === === === === === === === === === === === === ===
#

    ## Visualization of Error and Speed Trade-off
    # I made my own color heatmap, so yeah this is what this is below
    btr = LinearSegmentedColormap.from_list("BlackRed", ["black", "red"])
    colorway = btr(np.linspace(0, 1, len(point_range)))
    plt.figure(figsize=(12, 6))


    # plotting the error vs number of points
    plt.subplot(1, 3, 1)
    #plt.plot(point_range, error_range, marker='', color='blue',
linestyle='-', label='Error')
    plt.scatter(point_range, error_range, color=colorway,s=3)
    plt.title("Error vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Error")
```

```
    plt.yscale('log')
    plt.xscale('log')
    plt.grid(True)
    #plt.legend()

    # plotting the time vs number of points
    plt.subplot(1, 3, 2)
    #plt.plot(point_range, time_range, marker='', color='blue',
linestyle='-', label='Time')
    plt.scatter(point_range, time_range, color='black',s=3)
    plt.title("Time vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Time (seconds)")
    plt.grid(True)
    #plt.legend()

    # plotting the approximation vs number of points
    plt.subplot(1, 3, 3)
    #plt.plot(point_range, approx_range, marker='', color='blue',
linestyle='-', label='Approximation')
    plt.scatter(point_range, approx_range, color=colorway,s=3)
    plt.title("Approximation vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Approximation")
    plt.grid(True)
    #plt.legend()

    plt.tight_layout()
    plt.show()

    return

# ------------------------------------------------------------------------
--- #
## Technique - 2 / Trapezoid Rule Approximation
def function(x):
    # define a function to integrate
    func = np.sqrt(np.clip(1 - x**2, 0, None))

    return func
```

```
def trap_rule():
    # start time (for measuring speed of approximation)
    start_time = time.time()

    # create some data to track trade-offs later
    point_range = []
    error_range = []
    time_range = []
    approx_range = []

    # define number of intervals (should ultimately determine accuracy)
    num_points = 1000

    for i in range(2, num_points, 1):

        # define bounds for integration
        b = math.pi
        a = 0

        ## trapezoidal rule algorithm
        # interval generation
        x = np.linspace(a, b, num_points)
        # evaluating the function at each subinterval point
        y = function(x)
        # step size calculation
        h = (b - a) / (i - 1)
        # approximate the function using the trapezoidal rule
        trap = (h / 2) * (y[0] + 2 * np.sum(y[1:-1] + y[-1]))

        # approximate PI
        approx = 4 * trap

        # calculate error to math.pi
        error = abs(math.pi - approx)

        # calculate the time taken for approximation
        end_time = time.time()
        duration = end_time - start_time

        # append info for trade-offs (similar to .pushback in C++)
        point_range.append(i)
```

```
        error_range.append(error)
        time_range.append(duration)
        approx_range.append(approx)

    # results with speed
    print(f"Trapezoidal Rule Approximation of π with {num_points} points:
{approx}")
    print(f"Speed of Approximation: {duration} seconds")
    print(f"Error of Approximation: {error}")

    # === === === === === === === === === === === === === === === === ===
#

    ## Visualization of Error and Speed Trade-off
    # same personal color heatmap I made earlier
    btr = LinearSegmentedColormap.from_list("BlackRed", ["black", "red"])
    colorway = btr(np.linspace(0, 1, len(point_range)))
    plt.figure(figsize=(12, 6))

    # plotting the error vs number of points
    plt.subplot(1, 3, 1)
    #plt.plot(point_range, error_range, marker='', color='blue',
linestyle='-', label='Error')
    plt.scatter(point_range, error_range, color=colorway,s=3)
    plt.title("Error vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Error")
    plt.yscale('log')
    plt.xscale('log')
    plt.grid(True)
    #plt.legend()

    # plotting the time vs number of points
    plt.subplot(1, 3, 2)
    #plt.plot(point_range, time_range, marker='', color='blue',
linestyle='-', label='Time')
    plt.scatter(point_range, time_range, color='black',s=3)
    plt.title("Time vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Time (seconds)")
    plt.grid(True)
```

```
    #plt.legend()

    # plotting the approximation vs number of points
    plt.subplot(1, 3, 3)
    #plt.plot(point_range, approx_range, marker='', color='blue',
linestyle='-', label='Approximation')
    plt.scatter(point_range, approx_range, color=colorway,s=3)
    plt.title("Approximation vs Number of Points")
    plt.xlabel("Number of Points")
    plt.ylabel("Approximation")
    plt.grid(True)
    #plt.legend()

    plt.tight_layout()
    plt.show()

    return

# ------------------------------------------------------------------------
--- #
## Main Menu
def main():
    # menu for choosing method to run
    print(" -- Please choose an algorithm to approximate PI -- ")
    print(" >> a) Monte Carlo Method")
    print(" >> b) Trapezoidal Rule Integration")
    choice = input(">> ")

    # choice selection for menu
    if (choice == "a" or choice == "A"):
        print("// Processing //")
        monte_carlo()
    elif (choice == "b" or choice == "B"):
        print("// Processing //")
        trap_rule()
    else:
        print("###/ ERROR: Please select a valid option /###")
        main()

main()
```

```
# ------------------------------------------------------------------------
--- #
```