AMS 595 Project 1: Monte Carlo Estimation

Jane Condon

September 11, 2025

1 Task 1: Using a For Loop to Estimate π

In this task, we will compute π with a fixed number of points using a for loop, and plot the computed value of π vs. its deviation from the true value of π as the number of points increases. We will also measure the execution time and plot the precision vs. the computational cost. For a cleaner solution, we will define a helper function 'estimate,' which will calculate the estimate of pi using the Monte Carlo simulation.

1.1 Constructing the 'Estimate' Function

The Monte Carlo simulation involves simulating random points inside a square and determining how many points lie within the unit circle. To elaborate, we start by generating random x and y values within the range [-1,1]x[-1,1] to obtain the random points. Once we have obtained our random points, we find the number of points that lie within the unit circle by calculating the distance of each respective point from the origin using the following formula:

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

We then add the number of points that fall within the unit circle, as well as the total number of points. To calculate the estimate of π, we can use the following formula:

$$\frac{N_{inside}}{N_{total}} \cdot 4 = \pi$$

```
% Initializing variables
ninside = 0; % Number of points inside the circle
piestimate = 0; % Estimate of pi

% Monte Carlo Simulation
for i = 1:n
    x = -1 + 2*rand(); % Generating x value
    y = -1 + 2*rand(); % Generating y value
    if (x^2 + y^2) <= 1 % Checking for points inside the circle
        ninside = ninside + 1; % Adding up points inside the circle
    end
end
piestimate = 4*(ninside./n); % Calculating estimate of pi

end
```

Figure 1: Code used to calculate the estimate of π.

1.2 Calling the 'Estimate' Function and Putting Everything Together

To complete the rest of the task, we must now call the 'estimate' helper function. Before calling the function, we can choose the range of exponents for 2^k points. Arbitrarily, we choose k to be 10:20

so that we can generate 1 million points. Next, we call the 'estimate' function inside our for loop to compute π within a fixed number of deviations. We use the 'tic' and 'toc' functions to measure the time elapsed. This will be important when we want to plot the precision vs. computational cost. Finally, we calculate the error (i.e., the deviation of the estimate of π from its true value) using this formula:

$$|\pi_{estimate}(idx) - \pi|$$

```
% Choosing range of exponents for 2^k points: 1 million points
k = 10:20;
N = 2.^k;
% Initializing variables
piestimate = zeros(size(N)); % Estimate of pi
elapsed_time = zeros(size(N)); % Amount of time elapsed in seconds
pierror = zeros(size(N)); % Error

% Constructing for loop to compute pi within fixed number of deviation
for idx = 1:length(k)
    n = N(idx); % Number of points
    tic; % Timing the simulation
    piestimate(idx) = estimate(n); % Calling helper function 'estimate
    elapsed_time(idx) = toc; % Measuring time elapsed
    pierror(idx) = abs(piestimate(idx) - pi); % Calculating precision
end
```

Figure 2: Code used to calculate the deviation of estimates of π from its true value.

1.3 Plotting the Estimate of π and Error vs. Number of Points

Next, we want to plot the computed value of π and its deviation from the true value of π as the number of points increases. For simplicity, we will plot both lines on the same plot.

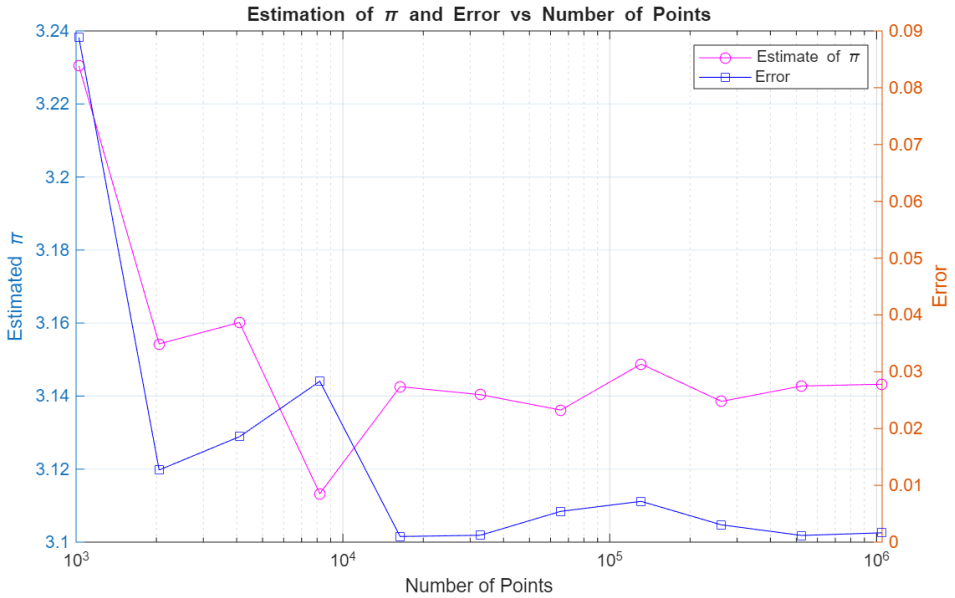

Figure 3: Plot of Estimate of π and Deviation From its True Value vs. Number of Points.

As illustrated by the plot above, it is clear that the estimate of π becomes more accurate as the number of points increases. Looking at the estimate of π line on the plot, we can see that it gets closer to the true value of π as the number of points increases. Similarly, the error (i.e., deviation from true value) shrinks toward zero as the number of points increases.

We will also look at a plot of precision vs. computational cost.

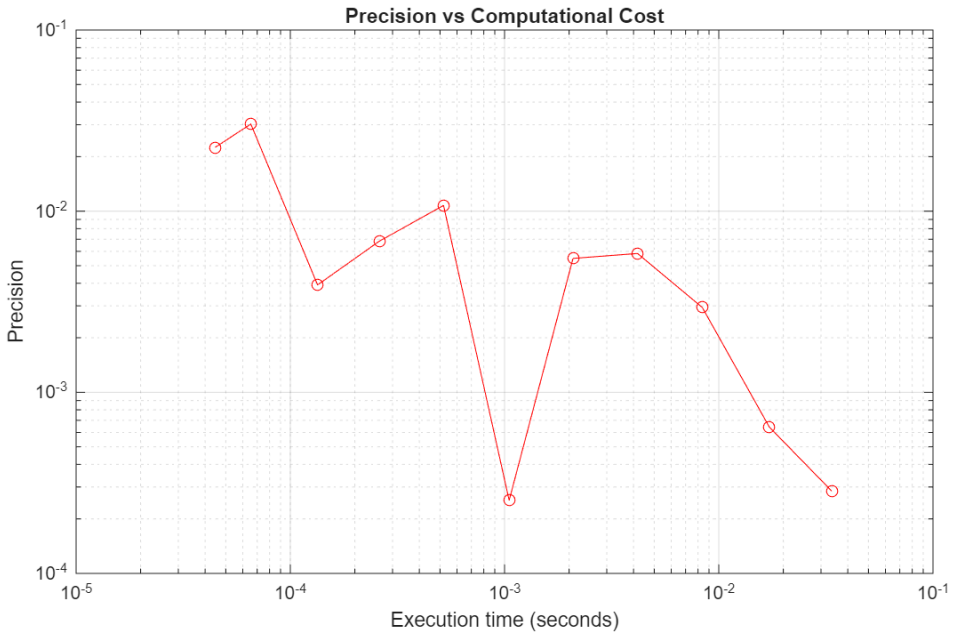

Figure 4: Plot of Precision vs. Computational Cost.

As shown in the plot above, there is a trade-off between precision and computational cost. For a higher level of precision, it can be costly in terms of execution time. That is, as the execution time increases, the precision also increases (the deviation from the true value decreases).

2 Task 2: Using a While Loop to Estimate π to a Specified Level of Precision

In this task, we will construct a while loop to compute π to a specified level of precision (measured in the number of significant figures) and record the number of iterations. required to achieve each level of precision. It is important to mention that we will **NOT** rely on the true value of π to determine precision. Similar to the previous task, we will use a helper function for a cleaner solution.

2.1 Constructing the 'Estimate2' Function

This function will estimate pi to a specified level of precision (number of significant figures) without using the true value of π using the Monte Carlo simulation. For stability and accuracy, we will use a 'buffer.' This buffer will store the last few estimates of π so that we are requiring multiple consecutive estimates to match the number of significant figures, rather than just a single estimate of π. We choose a buffer size of 10, meaning that we will store the last 10 estimates of π in this buffer.

```
% Creating buffer to store the last few estimates of pi so that
% we are requiring multiple consecutive estimates to match: makes
% the estimate more stable and accurate.

buffersize = 10; % Number of consecutive estimates to check
buffer = zeros(1, buffersize); % Storing last 10 estimates
```

Figure 5: Code used to create the buffer.

Inside the while loop, we calculate the estimate of π using the Monte Carlo simulation, as explained

in the previous section. We update the current estimate of π, as well as the current value of the buffer. Lastly, we check if the values in the buffer match the desired number of significant figures.

```
% Checking if values in buffer match the desired significant
% figures
    if n >= buffersize
    rounded = round(buffer, sig_figs, 'significant');
    if length(unique(rounded)) == 1
        break % Stop when desired precision is achieved
    end
end
```

Figure 6: Code used to check if the estimates match the desired number of significant figures.

After putting everything together, the while loop looks like this:

```
% Monte Carlo Simulation
while true
    n = n + 1; % Adding 1 to number of points
    x = -1 + 2*rand(); % Generating random x value
    y = -1 + 2*rand(); % Generating random y value
    if (x^2 + y^2) <= 1 % Checking if point lies inside unit circle
        ninside = ninside + 1;
    end

    piest = 4 * ninside / n; % Updating current estimate of pi
    buffer = [buffer(2:end), piest]; % Updating buffer

    % Checking if values in buffer match the desired significant
    % figures
        if n >= buffersize
        rounded = round(buffer, sig_figs, 'significant');
        if length(unique(rounded)) == 1
            break % Stop when desired precision is achieved
        end
    end
end
end
```

Figure 7: Full code used to compute π to a specified level of precision (measured in the number of significant figures) and record the number of iterations.

2.2 Calling the 'Estimate2' Function and Putting Everything Together

In the main code file, we loop through 2-4 significant figures. Inside the loop, we call the estimate2 function and use the 'tic' and 'toc' functions to measure time elapsed. If more precision is needed, we can loop through more significant figures. For the sake of running time, we will stop at 4 significant figures.

```
for i = 1:3 % Looping over 2-4 significant figures
    sig = i + 1; % 2 - 4 significant figures
    tic; % Timing the simulation
    % Calling helper function 'estimate2' to estimate pi to
    % specificed precision
    [piest(sig), n(sig)] = estimate2(sig);
    time(sig) = toc; % Recording time elapsed
end
```

Figure 8: Calling the estimate2 function for different levels of precision.

We can create a table displaying the results:

Sig Figs	Pi Estimate	Iterations	Time
2	3.0538	93	0.0011517
3	3.1429	560	0.0020185
4	3.1024	3810	0.020585

Figure 9: Table displaying the estimate of π, number of iterations, and execution time for different levels of precision.

As shown in the table above, the number of iterations, as well as execution time, tend to increase as the level of precision increases. Although it is not clearly evident in this table, the estimate of π also tends to be more accurate at higher levels of precision.

3 Task 3: Modifying the Code in Task 2 to be a Function

Input: The function will take a user-specified level of precision as the input.

Features: The function will do all of the following:

- Plot the random points as they are generated, with points inside the circle plotted in a different color than those outside the circle.
- Display the final computed value of π, written out to the user-specified precision, in the command window and print it on the plot.
- Return the computed value of π.

To complete this task, we will define a function called 'plotestimates.' To ensure accuracy, we will set the minimum number of iterations to 50. To generate the points as they are being plotted, we first set up the plot:

```
% Setting up the plot
figure;
hold on;
axis equal;
xlim([-1 1]); % Holding x-values between -1 and 1
ylim([-1 1]); % Holding y-values between -1 and 1
xlabel('X'); % Labeling the x-axis as 'X'
ylabel('Y'); % Labeling the y-axis as 'Y'
title('Monte Carlo Estimation of \pi'); % Creating a title for the plot

% Plotting the unit circle
theta = linspace(0,2*pi,100);
plot(cos(theta), sin(theta), 'k-', 'LineWidth', 2);
```

Figure 10: Code used to set up the plot.

Next, we run the Monte Carlo simulation that we ran in task 2 with a few modifications.

```
% Monte Carlo Simulation
while true
    n = n + 1; % Adding 1 to number of points
    x = -1 + 2*rand(); % Generating random x value
    y = -1 + 2*rand(); % Generating random y value
    if x^2 + y^2 <= 1 % Checking if the point lies inside the unit circl
        ninside = ninside + 1;
        plot(x, y, 'b.'); % Inside the circle: plot a blue point
    else
        plot(x, y, 'r.'); % Outside the circle: plot a red point
    end

    piest = 4 * ninside / n; % Updating estimate

    % Updating buffer
    buffer = [buffer(2:end), piest];

    % Checking if the estimate has stabilized
    if n >= max(min_iter, buffer_size)
        rounded = round(buffer, sig_figs, 'significant');
        if length(unique(rounded)) == 1
            break
        end
    end
end
```

Figure 11: Code used to plot the points as they are being generated.

Finally, we display the final estimate and print the value on the plot:

```
% Displaying final estimate
fprintf('Estimated value of pi to %d significant figures: %.*f\n', ...
        sig_figs, sig_figs, piest);

% Printing value on the plot
text(0, -1.1, sprintf('\\pi ≈ %.*f', sig_figs, piest), ...
    'HorizontalAlignment', 'center', 'FontSize', 12, 'FontWeight', 'bold
```

Figure 12: Code used to display the final estimate and print the value on the plot.

See the figure below for an example using the plotestimate function for a user-specified input of 4 significant figures:

```
>> plot_estimates(4)
Estimated value of pi to 4 significant figures: 3.1545
```

Figure 13: Output given for a user-specified input of 4 significant figures.

See the plot below:

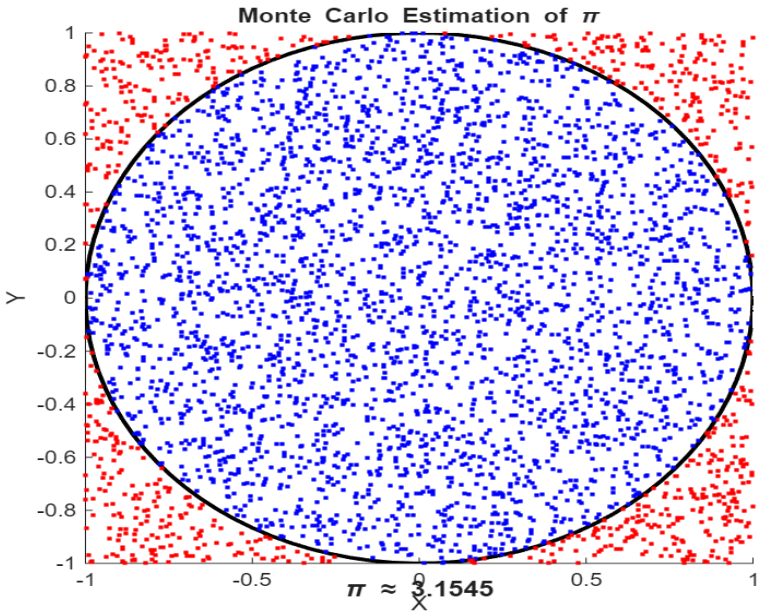

Figure 14: Plot displayed for a user-specified input of 4 significant figures.