Week 2: Monte Carlo in Geometry

Vectorization, Integration, and Convergence

Introduction

In Week 1, you learned Python basics and OOP. Now, we apply **NumPy** and **Matplotlib** to perform actual Monte Carlo simulations.

The core principle of Monte Carlo integration is simple: If you cannot calculate an area mathematically, enclose it in a box of known area, throw random points at it, and count how many land inside.

$$\text{Area}_{shape} \approx \text{Area}_{box} \times \frac{\text{Points}_{\text{inside}}}{\text{Total Points}}$$

Assignment 2.1 and 2.2 involve estimating transcendental numbers, assignment 2.3 takes a step deeper into calculus. **Assignment 2.4 is optional to attempt however it is highly recommended you just read through the problems and think about them.**

Assignment 2.1: Estimating Pi (π)

2.1.1 The Logic

We know that the area of a circle is πr^2. Consider a circle of radius $r = 1$ centered at $(0, 0)$ inscribed inside a square with side length 2 (from $x = -1$ to 1, $y = -1$ to 1).

- Area of Square $= (2r)^2 = 4$.
- Area of Circle $= \pi(1)^2 = \pi$.
- Ratio $\frac{\text{Circle}}{\text{Square}} = \frac{\pi}{4}$.

Task:

1. Generate N random (x, y) coordinates uniformly distributed between -1 and 1.
2. Calculate the distance of each point from the origin: $d = \sqrt{x^2 + y^2}$.
3. Count how many points satisfy the condition $d \leq 1$.
4. Use the ratio formula to estimate π.

2.1.2 Vectorization (The "No For-Loop" Rule)

In scientific computing with Python, **Loops are slow**. NumPy allows us to perform operations on entire arrays at once (Vectorization).

Requirement

You must implement the calculation in **Assignment 2.1.3** without using a single `for` loop for the point generation or distance checking. This should be easy given assignment 1.1.

Hint:

```
# Bad (Slow)
count = 0
for i in range(N):
    if x[i]**2 + y[i]**2 <= 1:
        count += 1

# Good (Vectorized)
distances = x**2 + y**2  # Computes array of all distances instantly
inside_mask = distances <= 1 # Creates boolean array
count = np.sum(inside_mask)
```

2.1.3 Convergence Analysis

The accuracy of Monte Carlo increases with the number of samples N. **Task:**

1. Create a generic function (or class method) that takes N as input and returns the estimated π.
2. Run this function for various values of N ranging from 10^1 to 10^7(preferably use exponential scaling, and scale output by a constant factor say double for each test)[1].
3. **Plot 1:** x-axis = N (Log Scale), y-axis = Estimated π. Draw a horizontal line at the true value of π.
4. **Plot 2:** x-axis = N (Log Scale), y-axis = Percent Error ($\frac{|\pi_{est} - \pi_{true}|}{\pi_{true}} \times 100$).

[1]You can use a `for` loop here

Assignment 2.2: Estimating Euler's Number (e)

Now that you have a framework for π, let's estimate e. We know that:

$$\int_1^t \frac{1}{x}\,dx = [\ln(x)]_1^t = \ln(t) - \ln(1) = \ln(t)$$

Procedure:

- Define a bounding box. x goes from 1 to 2. Since $1 \geq \frac{1}{x} \geq 0$, let y go from 0 to 1.
- Generate random points in this rectangle $[1, 2] \times [0, 1]$.
- A point is "inside" the integral if $y_{point} < \frac{1}{x_{point}}$.
- Calculate the Area under the curve.
- Now Area $= \ln(2) = \frac{1}{\log_2 e}$, hence $e = 2^{\frac{1}{\text{Area}}}$
- Like in previous step we used value of t as 2, try using other values.
- Report the error relative to `np.e`.

It can be argued that if we do not e exponentiation cannot be well defined, so a more generic and kind of "magical" method is given below[2],

Suppose for the moment that $0 \leq u_1 = G \leq 1$. Generate samples $u_2, u_3, \cdots$ from the uniform distribution so long as the numbers are decreasing, and then stop. In other words, find $n \geq 1$ such that

$$G = u_1 > u_2 > u_3 > \cdots > u_n \leq u_{n+1} \tag{1}$$

The probability that

$$G > u_2 > u_3 > \cdots > u_n > u_{n+1}\text{is}$$

$$\frac{G^n}{n!} = \frac{\text{Prob}(\max(u_2, \cdots, u_n + 1) < G)}{n!}$$

so the probability of 1 is

$$p_n = \frac{G^{n-1}}{(n-1)!} - \frac{G^n}{n!}$$

Here $p_1 + p_2 + \cdots = 1$ by telescoping series, so the algorithm terminates with probability 1. The magic is expected value of n is $\exp(G)$, so substituting G as 1 and applying Monte Carlo, we get expected value of n as e.[3]

As a part of the assignment you must write a python function that takes in number of tests to generate, generates that many test cases, finds n in each and takes average to find an estimate of e.

[2]George Forsythe's last paper, Presented at the "Stanford 50" meeting, Stanford University, 29 March 2007

[3]Formal proof might be added in week 3 assignment

A small bonus here is +5 chocolate points if you implement this part using only numpy and no `for` loops

Assignment 2.3: The "Modular" Shape Estimator

The code for 2.1 and 2.2 probably looks very similar. In software engineering, we avoid repeating code (DRY Principle).

Task: Refactor your code to be highly modular. Create a class or a master function that accepts a **Function/Predicate** as an argument.

Conceptual Design:

```python
def run_simulation(predicate_func, bounds, num_samples):
    # 1. Generate points within 'bounds'
    # 2. logical_mask = predicate_func(x, y)
    # 3. Calculate Area
    return area

# Example usage for Pi
def is_in_circle(x, y):
    return (x**2 + y**2) <= 1

pi_area = run_simulation(is_in_circle, bounds=[-1,1,-1,1], N=10000)
```

Sub-tasks: Using this modular code, estimate and plot convergence errors for:

1. **Circle:** (Standard π estimation).
2. **Parabola:** Area under $y = x^2$ for $x \in [0, 1]$. (True Area = 1/3).
3. **Gaussian**[4]**:** Area under $y = e^{-x^2}$ for $x \in [0, 2]$. (Compare against `scipy.special.erf` or numerical integration).

Assignment 2.4 (Optionals)

Following parts are optional given the lack of time, however if you want a much deeper understanding of the topic do try these out. Just reading through and thinking about the problem will help you appreciate Monte Carlo simulations.

[4]area under $y = e^{-x^2}$ also known as error function has no closed form and is written in form of integral. Hence this example shows how Monte Carlo can estimate values which are hard to calculate by general methods. For this specific case Monte Carlo is not the best option, look at assignment 2.4.2 for better picture.

2.4.1 Some more involved geometory

If you finish early:

- **Hypersphere:** Estimate the volume of a 4-dimensional or 10-dimensional unit sphere. Note how the volume *decreases* as dimensions increase.
- **Intersection*:** Find the area of intersection* between two overlapping circles with different centers.

2.4.2 Limitations: The Hammer and the Scalpel

You might have noticed in Assignment 2.3 that to get a precise value for the area of a circle (e.g., 5 decimal places), you needed millions of points.

Theory: The error in Monte Carlo integration decreases proportional to $1/\sqrt{N}$. To reduce the error by a factor of 10, you need 100 times more points. For 1D or 2D integrals of smooth functions (like x^2 or $\sin(x)$), standard deterministic numerical methods (like the Trapezoidal Rule or Simpson's Rule) are vastly superior. They converge much faster (often $1/N^2$ or $1/N^4$).

However, these deterministic grid-based methods suffer from the **Curse of Dimensionality**. If you need 10 grid points per axis to get a good result:

- 1 Dimension: 10 points.
- 2 Dimensions: $10^2 = 100$ points.
- 10 Dimensions: $10^{10} = 10$ Billion points (Infeasible).

Monte Carlo does not rely on a grid. It relies on random sampling, making it one of the few viable methods for high-dimensional integration.

In this section, we will rigorously test the limits of Monte Carlo integration against other mathematical techniques. We will look at three scenarios: a simple 1D problem, a high-dimensional geometric problem, and a complex function integration.

The Three Competitors

For each task, you will attempt to calculate the result using three approaches:

1. **Analytical (Closed Form):** Using calculus to find the exact mathematical answer.
2. **Deterministic (Riemann Sum):** Using a grid of evenly spaced points (e.g., `np.linspace`) to approximate the area.
3. **Stochastic (Monte Carlo):** Using random sampling.

Task A: The 1D Sprint (Low Dimension)

Problem: Calculate $\int_0^\pi \sin(x)\,dx$.

1. **Analytical:** Solve the indefinite integral on paper. Calculate the exact value (Answer: 2.0).
2. **Riemann Sum:** Create a grid of $N = 100,000$ points. Calculate the sum of rectangles.
3. **Monte Carlo:** Generate $N = 100,000$ random points in the bounding box $[0, \pi] \times [0, 1]$.

Benchmark: Calculate the **Absolute Error** ($|\text{Estimated} - \text{True}|$) for both computational methods. *Hypothesis: You should find that for 1D, the Riemann Sum is significantly more accurate than Monte Carlo.*

Task B: The Hypersphere (High Dimension)

Problem: Calculate the volume of a 10-Dimensional Unit Hypersphere (Radius $R = 1$).

1. **Analytical:** The volume of an n-ball is given by:

$$V_n = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)} R^n$$

   For $n = 10$, this equals $\frac{\pi^5}{120} \approx 2.550$. Use this as your ground truth.
2. **Riemann Sum (The Curse):** Attempt to design a grid solution. If you use just 4 points per dimension, you need $4^{10} \approx 1,000,000$ points.[5] *Task: Implement a grid with only 4 points per axis and measure the error.*
3. **Monte Carlo:** Generate $1,000,000$ random points in a 10-D hypercube (Volume 2^{10}). Check how many satisfy $\sum x_i^2 \leq 1$.

Benchmark: Compare the error of the 4-point-per-axis Grid vs the Monte Carlo simulation.

[5] Trying more points per axis can potentially consume your entire RAM, Do not try more points as it may crash your system

Task C: The "Intractable" Gaussian

Problem: Integrate the Gaussian function $f(\mathbf{x}) = e^{-\sum x_i^2}$ over the 6-Dimensional hypercube defined by $-1 \leq x_i \leq 1$.

In real-world physics, we often encounter integrals that have no "Elementary Closed Form" (i.e., you cannot write the anti-derivative using basic algebra).

1. **Approach 1: Taylor Series Approximation (Calculus method):** Since we cannot integrate e^{-x^2} easily, we can approximate the function using a Taylor expansion around 0:
$$e^{-x^2} \approx 1 - x^2 + \frac{x^4}{2}$$
Integrate this polynomial from -1 to 1 for one dimension, then raise the result to the power of 6 (since the dimensions are independent). *Calculate this value manually or via code.*[6]

2. **Approach 2: Riemann Sum:** Try to run a grid simulation. Note that for 6 Dimensions, to get reasonable accuracy (say 10 points per axis), you need 10^6 evaluations. Run this.

3. **Approach 3: Monte Carlo:** Run a simulation with 10^6 random samples.

4. **The "True" Value:** For error checking, use Python's scientific library to get the precise value:

```
from scipy.special import erf
import numpy as np
true_val = (np.sqrt(np.pi) * erf(1)) ** 6
```

Analysis Questions:

- How close was the Taylor Series approximation? Would it get worse if we integrated from -5 to 5 instead of -1 to 1?
- Which method (Riemann vs MC) provided a better error-to-runtime ratio in 6 dimensions?

Deliverables

Submission will be through a google form. You can `zip` or `tar` your files. You can generate a tar file from terminal using the following command:

```
    tar -cvzf <your_name>.tar.gz <folder_name>
```

[6]Maybe try expanding to more terms

1. Python files (e.g., `mc_geometry.py`) containing your modular `MonteCarlo` class/-functions.
2. A Jupyter Notebook or script that imports your module and generates the requested plots (Log-Log convergence plots).
3. A short text reflection(as a comment in main.py or as markdown in jupyter note-book) on why vectorization made the code faster.