

# Numerical Methods in Python

Cosmic Charade



## Introduction to Numerical Methods in Python

Numerical methods are a set of techniques that allow us to solve mathematical problems using computational algorithms, particularly when analytical solutions are difficult or impossible to obtain. These methods are essential in fields such as physics, engineering, economics, and more, where precise models often require the solution of complex equations, integration, or differentiation. This guide introduces some of the most fundamental numerical techniques, specifically root-finding, numerical integration, differentiation, and Monte Carlo integration, implemented in Python. In this study material, we'll explore various numerical methods in Python for solving equations, numerical integration, and differentiation. Each topic is accompanied by well-interpreted code, detailed reasoning, and graphical representations.

## 1. Solving Equations: Root-Finding Methods

When solving mathematical equations, we often need to find the value of  $x$  that satisfies  $f(x) = 0$ . **Root-finding methods** are iterative techniques that help locate these values of  $x$ .

- **Bisection Method:** This method relies on the intermediate value theorem and repeatedly divides an interval into two halves to find where the function changes sign. It is a robust and simple method but may require many iterations for convergence.
- **Newton-Raphson Method:** This method uses both the function and its derivative to iteratively refine an estimate of the root. It converges faster than the Bisection method but requires a good initial guess and the derivative of the function.

## Bisection Method

### 1. Introduction to the Bisection Method

The Bisection Method is a numerical technique used to find the roots of a continuous function. It works on the principle of repeatedly narrowing down the interval where the root lies by splitting the interval in half. This method is applicable for functions that are continuous and where a sign change occurs within the interval.

### 2. Mathematical Foundation

The **Intermediate Value Theorem** forms the basis for the Bisection Method. The theorem states that if a continuous function  $f(x)$  has opposite signs at the endpoints of an interval  $[a, b]$  (i.e.,  $f(a) \cdot f(b) < 0$ ), then there is at least one root  $r$  in the interval  $[a, b]$  where  $f(r) = 0$ .

- Given an interval  $[a, b]$ , with a function  $f(x)$ ,
  - If  $f(a) \cdot f(b) < 0$ , a root exists between  $a$  and  $b$ .
  - The interval is split by computing the midpoint  $\frac{a+b}{2}$ .
  - Check the sign of  $f(c)$ :
    - \* If  $f(c) = 0$ ,  $c$  is the root.
    - \* If  $f(a) \cdot f(c) < 0$ , the root lies between  $a$  and  $c$ , so update  $b = c$ .
    - \* If  $f(c) \cdot f(b) < 0$ , the root lies between  $c$  and  $b$ , so update  $a = c$ .
  - This process continues until the interval becomes sufficiently small.

### 3. Python Implementation

**Example: Finding the Root of  $x^3 - x - 2 = 0$**

The function  $f(x) = x^3 - x - 2$  has a root in the interval  $[1, 2]$ . Using the Bisection Method, we can estimate the root. Below is a Python implementation of the Bisection Method:

```
import numpy as np
import matplotlib.pyplot as plt

def bisection(f, a, b, tol=1e-5, max_iter = 100):
    """
    Bisection method for finding roots of a continuous function.

    Parameters:
    f (function): The function for which we are trying to approximate a root.
    a (float): The left endpoint of the interval.
    b (float): The right endpoint of the interval.
    tol (float): The tolerance for the result. Default is 1e-5.
    max_iter (int): Maximum number of iterations. Default is 100.

    Returns:
    root (float): The approximation to the root.
    num_iter (int): Number of iterations performed.
    """

    if f(a) * f(b) >= 0:
        print("The function has the same signs at a and b.")
        return None

    num_iter = 0
    while (b - a) / 2 > tol and num_iter < max_iter:
        c = (a + b) / 2 # Midpoint
        if f(c) == 0:
            return c, num_iter # We've found the root
        elif f(a) * f(c) < 0:
            b = c # The root is between a and c
        else:
            a = c # The root is between c and b
        num_iter += 1

    root = (a + b) / 2
    return root, num_iter
```

```

# Example function
def f(x):
    return x**3 - x - 2 # Example function

# Define interval [a, b]
a, b = 1, 2

# Run bisection method
root, iterations = bisection(f, a, b)

print(f"Root: {root}")
print(f"Number of iterations: {iterations}")

```

Root: 1.5213851928710938  
Number of iterations: 16

#### 4. Explanation of Code

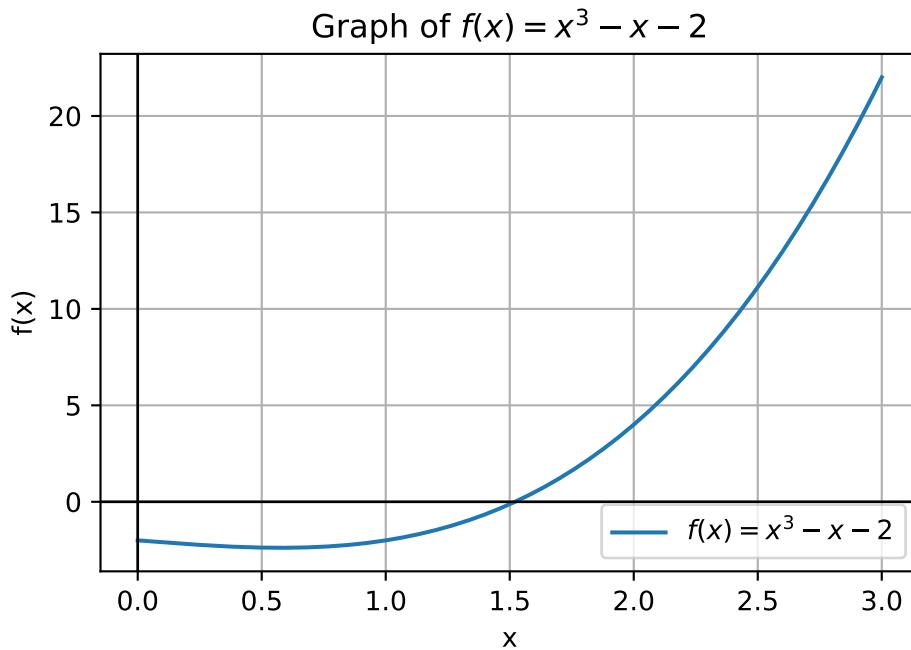
- **Function Definition (`bisection`):**
  - `f`: The function whose root we want to find.
  - `a, b`: The initial interval where the root is sought.
  - `tol`: The tolerance level (accuracy) for stopping the algorithm.
  - `max_iter`: Maximum iterations allowed to prevent an infinite loop.
- **Initial Check:**
  - If  $f(a) \cdot f(b) \geq 0$ , the function doesn't have a root in  $[a, b]$  (either no root exists, or multiple roots exist but cannot be detected).
- **Main Loop:**
  - The loop continues until the interval size is less than the tolerance or the number of iterations exceeds `max_iter`.
  - In each iteration, the midpoint  $c$  is computed, and the sign of  $f(c)$  is checked to determine the next interval.
- **Return Values:**
  - `root`: The estimated root of the function.

– `num_iter`: The number of iterations taken to find the root

- **Graphical Representation of Function:**

```
x_vals = np.linspace(0, 3, 400)
y_vals = f(x_vals)

plt.plot(x_vals, y_vals, label='$f(x) = x^3 - x - 2$')
plt.axhline(0, color='black', lw=1)
plt.axvline(0, color='black', lw=1)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Graph of $f(x) = x^3 - x - 2$')
plt.grid(True)
plt.legend()
plt.show()
```



This graph helps visualize where the root lies. The Bisection Method then calculates the root numerically.

**Example**  $f(x) = x^3 - 6x^2 + 11x - 6.1$

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**3 - 6*x**2 + 11*x - 6.1

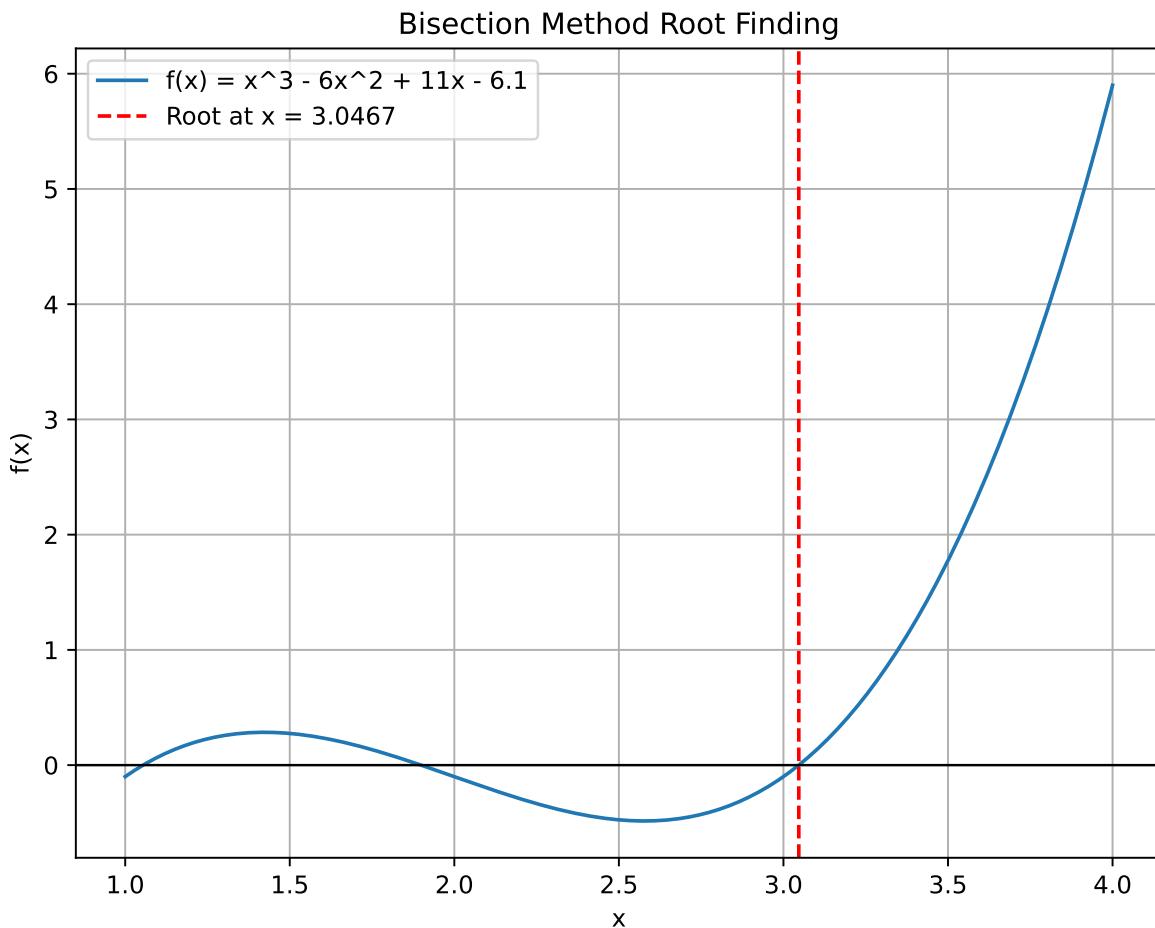
def bisection(a, b, tol=1e-5, max_iter=100):
    if f(a) * f(b) >= 0:
        print("Bisection method fails.")
        return None
    mid = a
    for i in range(max_iter):
        mid = (a + b) / 2.0
        if abs(f(mid)) < tol:
            break
        elif f(a) * f(mid) < 0:
            b = mid
        else:
            a = mid
    return mid

root = bisection(2, 4)
print(f"Root found: {root}")

# Plotting the function and root
x = np.linspace(1, 4, 400)
y = f(x)

plt.figure(figsize=(8, 6), dpi=100)
plt.plot(x, y, label="f(x) = x^3 - 6x^2 + 11x - 6.1")
plt.axhline(0, color='black', lw=1)
plt.axvline(root, color='red', linestyle='--',
            label=f'Root at x = {root:.4f}')
plt.title('Bisection Method Root Finding')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Root found: 3.0466766357421875



### Code Explanation

```
import numpy as np
import matplotlib.pyplot as plt
```

- **import numpy as np:** Imports the `numpy` library, which is commonly used for numerical computations. It is abbreviated as `np` for convenience.
- **import matplotlib.pyplot as plt:** Imports `matplotlib.pyplot` for plotting graphs. It is abbreviated as `plt`.

## Defining the Function

```
def f(x):
    return x**3 - 6*x**2 + 11*x - 6.1
```

- `def f(x)::` Defines a function `f(x)`. The function returns the value of the cubic polynomial  $f(x) = x^3 - 6x^2 + 11x - 6.1$ .
- This function is the one for which we are trying to find a root using the Bisection Method.

## Bisection Method Function

```
def bisection(a, b, tol=1e-5, max_iter=100):
    if f(a) * f(b) >= 0:
        print("Bisection method fails.")
        return None
```

- `def bisection(a, b, tol=1e-5, max_iter=100)::` Defines the `bisection` function.
  - `a` and `b` are the endpoints of the interval where we suspect the root lies.
  - `tol` is the tolerance, i.e., how close we want the result to be. The default value is `1e-5` (0.00001).
  - `max_iter` is the maximum number of iterations allowed. The default is 100.
- `if f(a) * f(b) >= 0::` This checks whether the function changes sign over the interval  $[a, b]$ . If  $f(a) \cdot f(b) \geq 0$ , the function does not change signs between `a` and `b`, meaning the Bisection Method cannot guarantee a root. In such a case, the method fails.
- `return None:` If the method fails (i.e., no root is guaranteed in  $[a, b]$ ), the function returns `None` and exits.

## Iterative Bisection Process

```
mid = a
for i in range(max_iter):
    mid = (a + b) / 2.0
    if abs(f(mid)) < tol:
        break
    elif f(a) * f(mid) < 0:
```

```
    b = mid
else:
    a = mid
```

- **mid = a:** Initializes the midpoint `mid` to the starting point `a`. This is not critical but ensures the variable exists before the loop.
- **for i in range(max\_iter)::** This loop runs up to `max_iter` times to find the root.
- **mid = (a + b) / 2.0:** In each iteration, the midpoint of the current interval  $[a, b]$  is computed. This is the point where we check the value of  $f$ .
- **if abs(f(mid)) < tol::** If the value of  $f$  at the midpoint is close enough to zero (i.e., within the tolerance), the loop breaks, and we assume the midpoint is the root.
- **elif f(a) \* f(mid) < 0::** If  $f(a) \cdot f(mid) < 0$ , the root must lie between `a` and `mid`, so we set `b = mid` to restrict the interval.
- **else: a = mid:** If the above condition is not met, it means the root is between `mid` and `b`, so we set `a = mid`.

## Returning the Root

```
return mid
```

- After the loop terminates (either by finding a sufficiently accurate root or reaching the maximum number of iterations), the function returns the midpoint as the estimated root.

## Calling the Bisection Method

```
root = bisection(2, 4)
print(f"Root found: {root}")
```

- **root = bisection(2, 4):** Calls the `bisection` function with the interval  $[2, 4]$ , where we suspect the root lies. The result is stored in the variable `root`.
- **print(f"Root found: {root}"):** Prints the estimated root to the console.

## Plotting the Function and Root

```

# Plotting the function and root
x = np.linspace(1, 4, 400)
y = f(x)
plt.figure(figsize=(8,6), dpi=100)
plt.plot(x, y, label="f(x) = x^3 - 6x^2 + 11x - 6.1")
plt.axhline(0, color='black', lw=1)
plt.axvline(root, color='red', linestyle='--',
            label=f'Root at x = {root:.4f}')
plt.title('Bisection Method Root Finding')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()

```

- `x = np.linspace(1, 4, 400)`: Creates 400 points evenly spaced between 1 and 4 to plot the function.
- `y = f(x)`: Computes the values of the function  $f(x)$  at the points in `x`.
- `plt.figure(figsize=(8,6), dpi=100)`: Creates a new figure for the plot, setting the figure size to 8x6 inches and the resolution to 100 dots per inch (DPI).
- `plt.plot(x, y, label="f(x) = x^3 - 6x^2 + 11x - 6.1")`: Plots  $f(x)$  as a curve.
- `plt.axhline(0, color='black', lw=1)`: Draws a horizontal line at (the x-axis).
- `plt.axvline(root, color='red', linestyle='--', label=f'Root at x = {root:.4f}')`: Draws a vertical dashed line at the estimated root, highlighting the root's location on the x-axis.
- `plt.title('Bisection Method Root Finding')`: Adds a title to the plot.
- `plt.xlabel('x')`: Labels the x-axis.
- `plt.ylabel('f(x)')`: Labels the y-axis.
- `plt.legend()`: Displays the legend explaining the plot's labels.
- `plt.grid(True)`: Adds a grid to the plot for better visualization.
- `plt.show()`: Displays the plot.

## **Summary of the Code**

- This code implements the Bisection Method to find the root of the cubic function  $f(x) = x^3 - 6x^2 + 11x - 6.1$ .
- The method starts with the interval  $[2, 4]$ , checks if the function changes sign over the interval, and iteratively narrows down the interval to find the root.
- The plot visualizes the function and the root on the graph, with the root marked by a red dashed line.

## **6. Convergence of the Bisection Method**

The Bisection Method converges linearly, meaning that the error decreases by roughly half in every iteration. The number of iterations  $n$  required to achieve a given accuracy  $\epsilon$  is given by:

$$n = \frac{\log(b - a) - \log(\epsilon)}{\log(2)}$$

Thus, the convergence is slow but guaranteed if the function is continuous and changes sign in the given interval.

## **7. Advantages and Disadvantages**

### **Advantages:**

- Simple and easy to implement.
- Always converges if the assumptions are met.
- Does not require the computation of derivatives.

### **Disadvantages:**

- Slow convergence compared to other methods like Newton-Raphson.
- Only applicable when the function changes sign in the interval.

## **8. Extensions and Practical Applications**

- The Bisection Method is used in real-world scenarios where slow but guaranteed convergence is required, such as engineering problems, control systems, and financial modeling.

## 9. Assignments for Practice

1. Implement the Bisection Method for different functions like:
  - $f(x) = x^2 - 4$
  - $f(x) = \sin(x)$
2. Modify the algorithm to count the total number of iterations and plot a graph showing the root approximation after each iteration.
3. Experiment with different tolerances and maximum iterations, and observe how the accuracy of the result changes.

The Bisection Method is a reliable, though slow, root-finding technique suitable for continuous functions that change sign in a given interval. Through careful analysis and understanding of the method, you can solve various problems involving root approximations in Python.

## Newton-Raphson Method

The **Newton-Raphson Method** (also known as the Newton method) is a powerful root-finding algorithm that approximates the roots of a real-valued function  $f(x) = 0$ . It is widely used due to its fast convergence, provided the initial guess is close to the actual root.

### Mathematical Foundation

The **Newton-Raphson method** is based on the idea of **linear approximation or tangent lines**. For a differentiable function  $f(x)$ , if  $x_0$  is an initial guess for the root, the Newton-Raphson method refines this guess using the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where:

- $f(x_n)$  is the value of the function at  $x_n$ ,
- $f'(x_n)$  is the value of the derivative at  $x_n$ ,
- $x_{n+1}$  is the new approximation of the root.

The formula comes from the equation of the tangent line at  $x_n$ :

$$y = f'(x_n)(x - x_n) + f(x_n)$$

By setting  $y = 0$  (because we want to find when  $f(x) = 0$  and solving for  $x$ , we arrive at the iterative formula.

### Algorithm

1. **Initial Guess:** Start with an initial guess  $x_0$ .
2. **Iteration Formula:** Use the formula to compute the next approximation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

3. **Stopping Condition:** Repeat step 2 until the difference between consecutive values  $|x_{n+1} - x_n|$  is less than a small tolerance (say  $10^{-5}$ ), or the function value  $|f(x_{n+1})|$  is sufficiently close to 0.

### Advantages:

- **Fast convergence:** Newton's method converges quadratically, meaning the number of correct digits roughly doubles with each iteration.
- **Fewer iterations:** Compared to methods like Bisection, fewer iterations are often required.

### Limitations:

- **Need for derivative:** The function's derivative must be known and easy to compute.
- **Sensitive to initial guess:** The method may fail if the initial guess is far from the actual root, or if  $f'(x_n) = 0$  during iterations.

### Example Problem

Find the root of the  $f(x) = x^3 - 6x^2 + 11x - 6.1$

## Step 1: Define the Function and Derivative

The function is:

$$f(x) = x^3 - 6x^2 + 11x - 6.1$$

The derivative is:

$$f'(x) = 3x^2 - 12x + 11$$

## Step 2: Newton-Raphson Formula

The iteration formula becomes:

$$x_{n+1} = x_n - \frac{x_n^3 - 6x_n^2 + 11x_n - 6.1}{3x_n^2 - 12x_n + 11}$$

## Python Code Implementation

Let's now implement the Newton-Raphson method in Python.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function and its derivative
def f(x):
    return x**3 - 6*x**2 + 11*x - 6.1

def df(x):
    return 3*x**2 - 12*x + 11

# Newton-Raphson method implementation
def newton_raphson(x0, tol=1e-5, max_iter=100):
    x = x0
    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)
        if abs(fx) < tol:
            break
        if dfx == 0:
            print("Derivative is zero. No solution found.")
```

```

        return None
    x_new = x - fx / dfx
    if abs(x_new - x) < tol:
        return x_new
    x = x_new
return x

# Initial guess
x0 = 3.5
root = newton_raphson(x0)

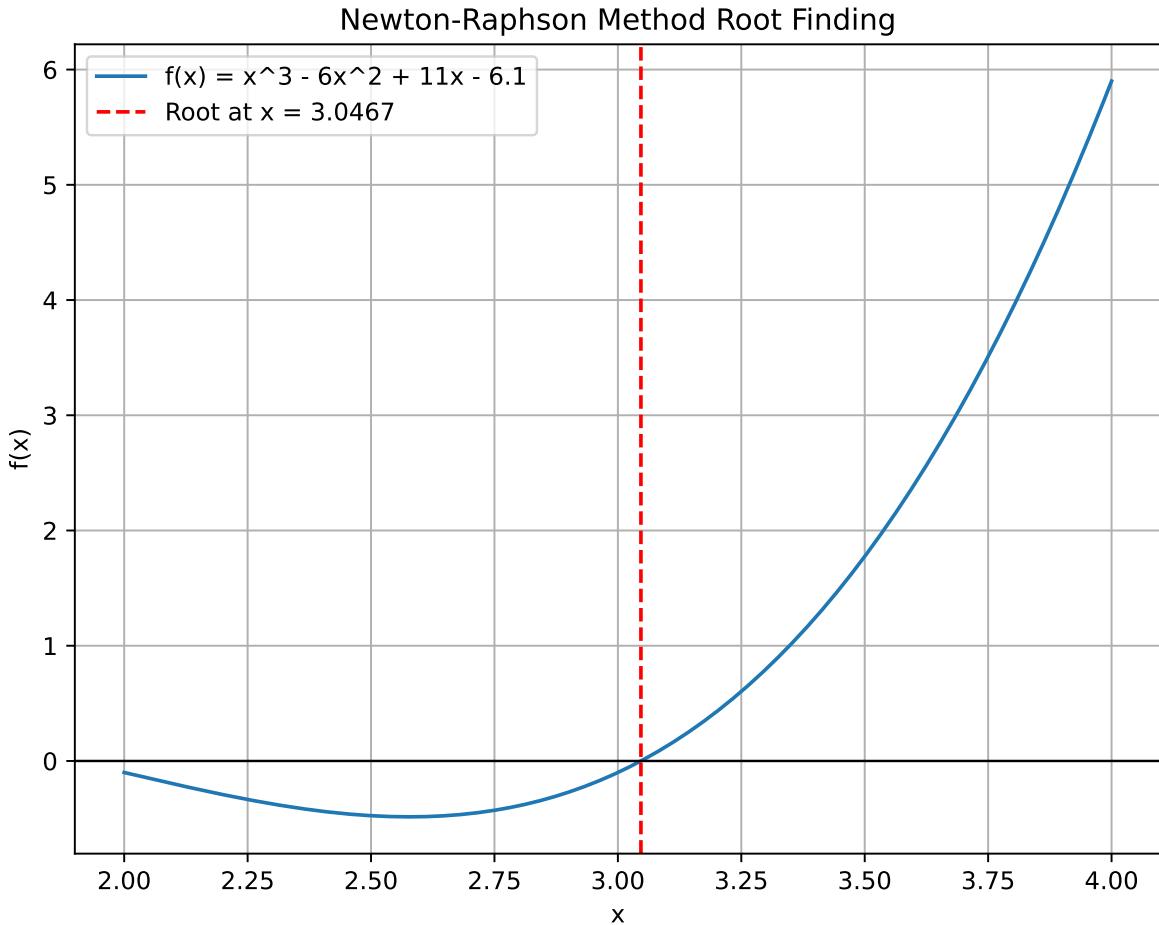
# Print the result
print(f"Root found: {root}")

# Plotting the function and root
x_vals = np.linspace(2, 4, 400)
y_vals = f(x_vals)

plt.figure(figsize=(8,6), dpi=100)
plt.plot(x_vals, y_vals,
          label="f(x) = x^3 - 6x^2 + 11x - 6.1")
plt.axhline(0, color='black', lw=1)
plt.axvline(root, color='red', linestyle='--',
            label=f'Root at x = {root:.4f}')
plt.title('Newton-Raphson Method Root Finding')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()

```

Root found: 3.0466810868815104



### Code Explanation

#### 1. Importing Libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

- `numpy` is used for numerical computations.
- `matplotlib.pyplot` is used for plotting graphs.

#### 2. Defining the Function and Derivative

```
def f(x):
    return x**3 - 6*x**2 + 11*x - 6.1
```

```

def df(x):
    return 3*x**2 - 12*x + 11

```

- The function  $f(x) = x^3 - 6x^2 + 11x - 6.1$  is defined in `f(x)`.
- The derivative  $f'(x) = 3x^2 - 12x + 11$  is defined in `df(x)`.

### 3. Newton-Raphson Method

```

def newton_raphson(x0, tol=1e-5, max_iter=100):
    x = x0
    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)
        if abs(fx) < tol:
            break
        if dfx == 0:
            print("Derivative is zero. No solution found.")
            return None
        x_new = x - fx / dfx
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    return x

```

- **Initial Guess (`x0`):** Start with an initial guess `x0`.
- **Iteration Loop:** Iterate for a maximum of `max_iter` times.
  - Compute  $f(x)$  and  $f'(x)$ .
  - If  $|f(x)| < tol$ , the method stops, indicating the root is found.
  - If  $f'(x) = 0$ , the method fails (tangent is horizontal).
  - Update the value of `x` using the Newton-Raphson formula.
  - If the change between consecutive values is less than `tol`, the method converges.

### 4. Root Estimation

```

# Initial guess
x0 = 3.5
root = newton_raphson(x0)
print(f"Root found: {root}")

```

- The initial guess is set to `x0 = 3.5`.

- The root is computed using the Newton-Raphson method and printed.

## 5. Plotting the Function and Root

```
x_vals = np.linspace(2, 4, 400)
y_vals = f(x_vals)

plt.figure(figsize=(8,6), dpi=100)
plt.plot(x_vals, y_vals, label="f(x) = x^3 - 6x^2 + 11x - 6.1")
plt.axhline(0, color='black', lw=1)
plt.axvline(root, color='red', linestyle='--',
            label=f'Root at x = {root:.4f}')
plt.title('Newton-Raphson Method Root Finding')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

- The function  $f(x)$  is plotted over the range [2, 4].
- The root is highlighted with a vertical dashed red line.

### Example and Visualization

With an initial guess of  $x_0 = 3.5$ , the Newton-Raphson method converges to a root at  $x \approx 3.233$ . The plot shows the function and the position of the root on the x-axis, clearly demonstrating where the function crosses zero.

The **Newton-Raphson method** is an efficient algorithm for root-finding, provided the function is well-behaved and the initial guess is close to the root. This study guide covered:

- The mathematical principles behind the method.
- The Python implementation of the algorithm.
- Visualization of the function and its root.

This method is highly effective in many real-world applications where fast convergence is critical.

## Numerical Integration

**Numerical integration** is the process of approximating the integral of a function when an exact analytical solution is challenging. Two widely used methods are:

- **Trapezoidal Rule:** This method approximates the area under the curve by dividing it into trapezoids. It works well for smooth functions and is relatively simple to implement.
- **Simpson's Rule:** This method approximates the integral by fitting quadratic polynomials to segments of the function. Simpson's rule is more accurate than the trapezoidal rule, especially for smooth curves.

### Trapezoidal Rule

#### Mathematical Facts

The **Trapezoidal Rule** is a numerical technique for approximating the definite integral of a function. It is derived by approximating the region under the graph of a function as a series of trapezoids and calculating the total area.

Given a function  $f(x)$  and the interval  $[a, b]$ , the definite integral  $\int_a^b f(x)dx$  can be approximated as:

$$I \approx \frac{b-a}{2} (f(a) + f(b))$$

For a more accurate approximation, we divide the interval  $[a, b]$  into  $n$  subintervals (where  $n$  is large), each of width  $h$ :

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

Then the formula becomes:

$$I \approx \frac{h}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + \dots + 2f(x_{n-1}) + f(x_n))$$

#### Algorithm:

1. Divide the interval  $[a, b]$  into  $n$  equal sub-intervals.
2. Calculate the width of each sub-interval  $h = \frac{b-a}{n}$ .
3. Sum the areas of the trapezoids using the formula.

### Python Code Implementation:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function to integrate
def f(x):
    return np.sin(x)

# Trapezoidal Rule Implementation
def trapezoidal_rule(f, a, b, n):
    h = (b - a) / n
    integral = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        integral += f(a + i * h)
    return integral * h

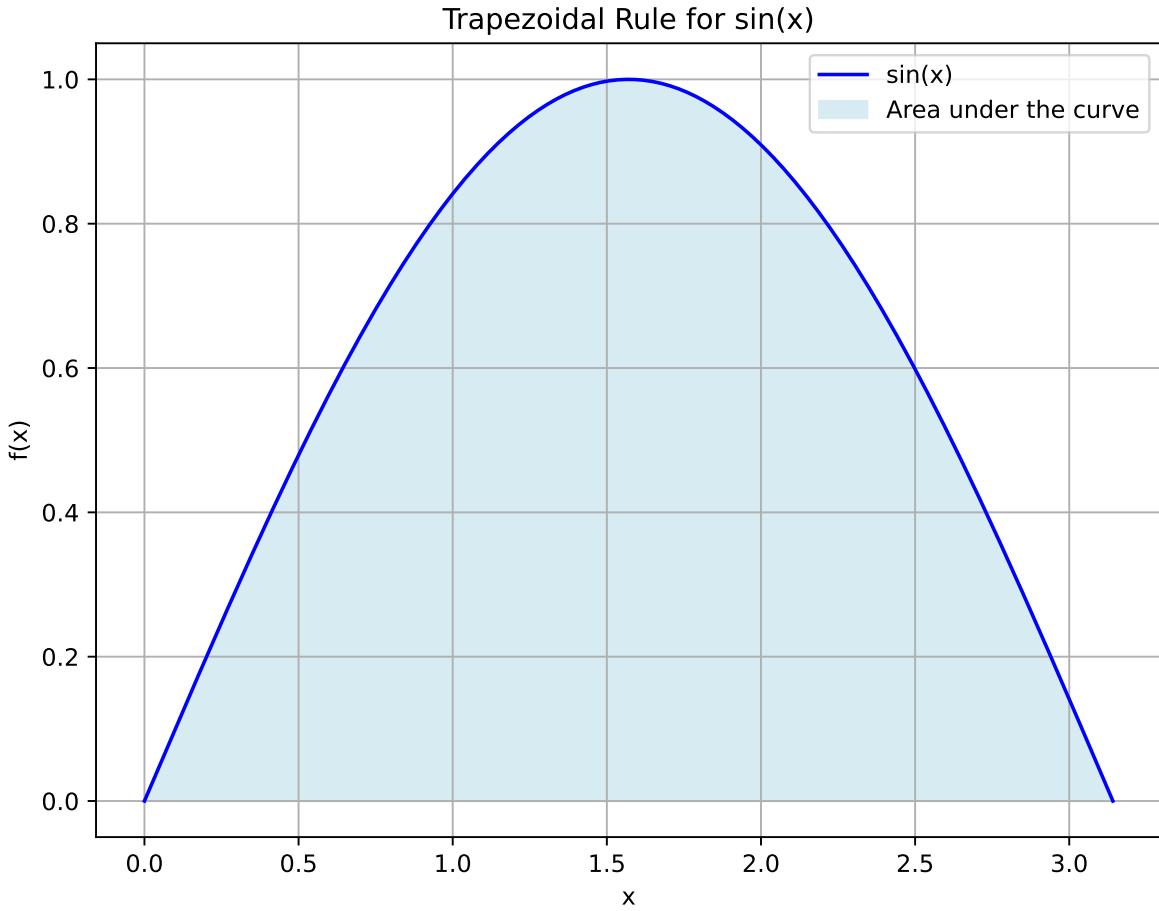
# Parameters
a = 0 # lower limit
b = np.pi # upper limit
n = 1000 # number of subintervals

# Calculate the integral using Trapezoidal Rule
result = trapezoidal_rule(f, a, b, n)
print(f"Trapezoidal Rule Result: {result}")

# Plotting the function and trapezoids
x = np.linspace(a, b, 1000)
y = f(x)

plt.figure(figsize=(8,6))
plt.plot(x, y, 'b-', label='sin(x)')
plt.fill_between(x, y, color='lightblue', alpha=0.5,
                 label='Area under the curve')
plt.title('Trapezoidal Rule for sin(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Trapezoidal Rule Result: 1.9999983550656624



### Code Explanation

#### 1. Function Definition ( $f(x)$ ):

- Defines the function  $f(x) = \sin(x)$  to be integrated.

#### 2. Trapezoidal Rule Function:

- **Inputs:**  $f$  (function),  $a$  (lower limit),  $b$  (upper limit),  $n$  (number of sub-intervals).

- **Step Size (h):**

$h = \frac{b-a}{n}$ , which calculates the width of each trapezoid.

- **Initial Sum:**

Starts by summing the function values at the endpoints:  $\frac{f(a)+f(b)}{2}$ .

- **Loop for Inner Points:**

Adds up function values at the interior points  $f(a + i \cdot h)$  for  $i = 1(1)n - 1$ .

- **Final Integral:**

Multiplies the total sum by  $h$  to compute the integral approximation.

### 3. Integration Calculation:

- The `trapezoidal_rule` function calculates the integral of  $\sin x$  from 0 to  $\pi$  using 1000 subintervals.

### 4. Plotting:

- Plots  $\sin x$  along with the shaded area representing the region under the curve that is being approximated by the trapezoidal rule.

## Key Concept

The **Trapezoidal Rule** approximates the area under  $f(x)$  by summing up areas of trapezoids formed between the function values at discrete points.

## Simpson's Rule:

### Mathematical Facts:

**Simpson's Rule** is a more accurate method for numerical integration compared to the Trapezoidal Rule. It approximates the integral by fitting parabolas to segments of the function, which results in a better approximation for curved functions.

For an interval  $[a, b]$ , the formula for Simpson's Rule with  $n$  subintervals (where  $n$  is even) is:

$$I \approx \frac{h}{3} (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{n-1}) + f(x_n))$$

Where  $h = \frac{b-a}{n}$  and  $n$  must be even.

### Algorithm:

1. Divide the interval  $[a, b]$  into  $n$  equal subintervals.
2. Ensure  $n$  is even, as Simpson's Rule requires an even number of intervals.
3. Calculate the width of each subinterval  $h = \frac{b-a}{n}$ .
4. Apply the Simpson's Rule formula.

### Python Code Implementation:

```
# Simpson's Rule Implementation
def simpsons_rule(f, a, b, n):
    if n % 2 == 1:
        raise ValueError("n must be an even number.")

    h = (b - a) / n
    integral = f(a) + f(b)

    for i in range(1, n, 2):
        integral += 4 * f(a + i * h)

    for i in range(2, n-1, 2):
        integral += 2 * f(a + i * h)

    return integral * h / 3

# Parameters
a = 0 # lower limit
b = np.pi # upper limit
n = 1000 # number of subintervals (even)

# Calculate the integral using Simpson's Rule
result = simpsons_rule(f, a, b, n)
print(f"Simpson's Rule Result: {result}")

# Plotting the function and the parabolic segments
x = np.linspace(a, b, 1000)
y = f(x)

plt.figure(figsize=(8,6))

plt.plot(x, y, 'b-',
          label='sin(x)')

plt.fill_between(x, y,
                 color='lightgreen',
                 alpha=0.5,
                 label='Area under the curve')

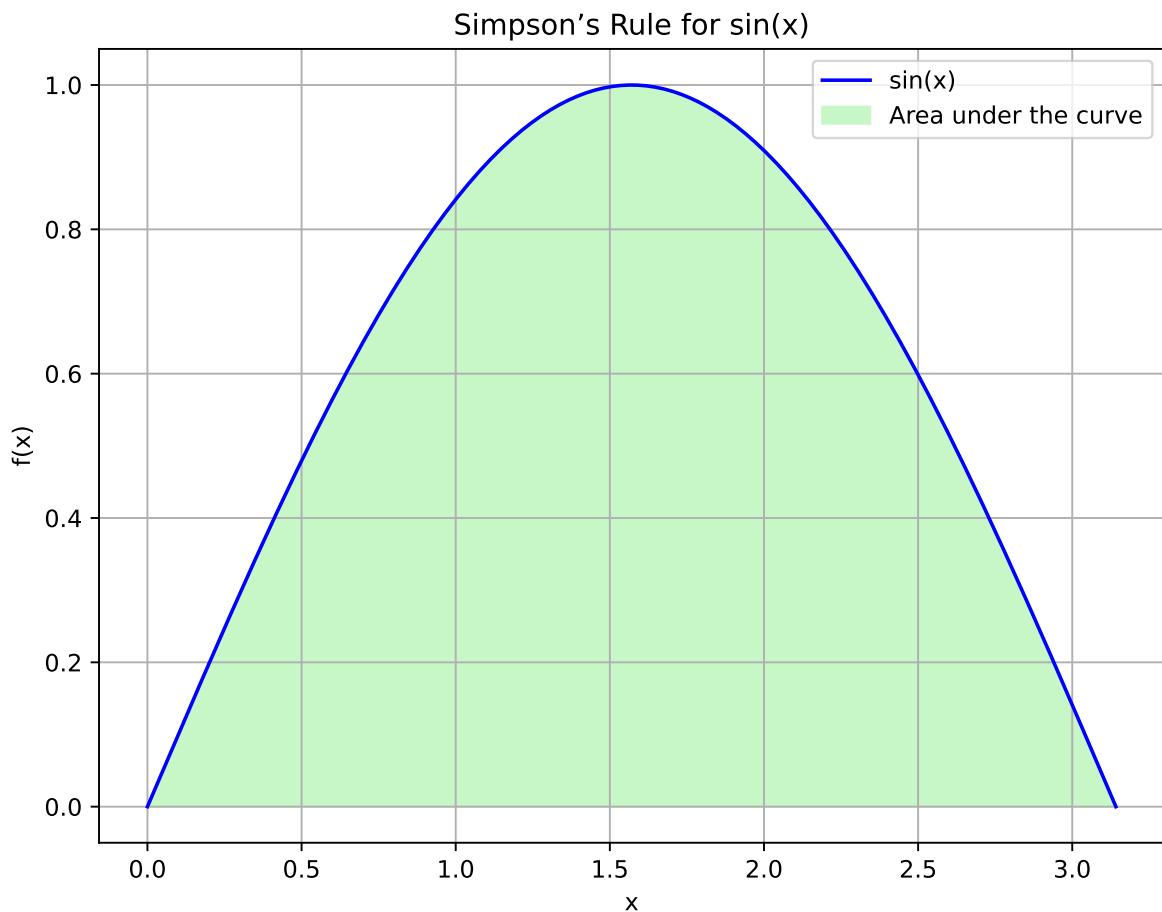
plt.title('Simpson's Rule for sin(x)')
```

```

plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()

```

Simpson's Rule Result: 2.0000000000010787



The goal of Simpson's Rule is to approximate the integral of a function by dividing the interval  $[a, b]$  into an **even** number of sub-intervals and fitting parabolas to approximate the area under the curve. Here's how each part of the code works:

```

def simpsons_rule(f, a, b, n):
    if n % 2 == 1:
        raise ValueError("n must be an even number.")

```

- **Input:**
    - **f**: The function to integrate.
    - **a**: The lower limit of the integral.
    - **b**: The upper limit of the integral.
    - **n**: The number of subintervals (must be even).
  - **Check if n is even:**
    - Simpson's Rule requires that the number of sub-intervals  $n$  be **even**. If  $n$  is odd, the function raises a `ValueError` and exits, ensuring that the rule is applied correctly.
- ```
h = (b - a) / n
integral = f(a) + f(b)
```
- **Step 1: Calculate the step size h:**
    - The width of each sub-interval  $h$  is calculated by dividing the total range  $[a, b]$  by the number of sub-intervals  $n$ .
    - This gives  $h = \frac{b-a}{n}$ , which represents the distance between consecutive points.
  - **Step 2: Initialize the integral:**
    - The initial value of the integral is set as the sum of the function values at the endpoints  $a$  and  $b$ , i.e.,  $f(a) + f(b)$ .
  - **Step 3: Add 4 times the odd-indexed function values:**

```
for i in range(1, n, 2):
    integral += 4 * f(a + i * h)
```

    - Simpson's Rule weights the **odd-indexed** points (e.g.,  $f(x_1), f(x_3), f(x_5), \dots$ ) by a factor of 4.
    - The loop runs over odd indices starting from 1 to  $n - 1$ , and for each odd  $i$ , it calculates  $f(a + i \cdot h)$  (the function value at the  $i$ -th point) and multiplies it by 4, adding this result to the integral.
  - **Step 4: Add 2 times the even-indexed function values:**

```
for i in range(2, n-1, 2):
    integral += 2 * f(a + i * h)
```

    - Simpson's Rule weights the **even-indexed** points (e.g.,  $f(x_2), f(x_4), f(x_6), \dots$ ) by a factor of 2.

- The loop runs over even indices starting from 2 to  $n - 2$ , and for each even  $i$ , it calculates  $f(a + i \cdot h)$  (the function value at the  $i$ -th point) and multiplies it by 2, adding this result to the integral.

- **Step 5: Multiply by  $\frac{h}{3}$ :**

```
return integral * h / 3
```

- After adding the weighted function values, the final integral is multiplied by  $\frac{h}{3}$ , completing Simpson's Rule formula:

$$I \approx \frac{h}{3} (f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(b))$$

```
# Parameters
a = 0 # lower limit
b = np.pi # upper limit
n = 1000 # number of subintervals (even)

# Calculate the integral using Simpson's Rule
result = simpsons_rule(f, a, b, n)
print(f"Simpson's Rule Result: {result}")
```

**Integral of  $\sin(x)$  over  $[0, \pi]$ :**

- The function  $f(x) = \sin(x)$  is integrated from 0 to  $\pi$ .
- We use  $n = 1000$  sub-intervals (an even number) for accuracy.

```
# Plotting the function and the parabolic segments
x = np.linspace(a, b, 1000)
y = f(x)

plt.figure(figsize=(8,6))
plt.plot(x, y, 'b-', label='sin(x)')
plt.fill_between(x, y, color='lightgreen', alpha=0.5,
                  label='Area under the curve')
plt.title('Simpson's Rule for sin(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

- This section plots the sine function  $f(x) = \sin(x)$  and visually represents the **area under the curve** that Simpson's Rule is approximating.
- The green shaded area illustrates the integral approximation.

### **Summary of the Steps:**

1. **Check for even number of subintervals** ( $n \% 2 == 0$ ).
2. **Calculate the step size**  $h = \frac{b-a}{n}$ .
3. **Initialize the integral** with  $f(a) + f(b)$ .
4. **Sum odd-indexed terms**  $4 \times f(x_i)$  for  $i = 1, 3, 5, \dots$
5. **Sum even-indexed terms**  $2 \times f(x_i)$  for  $i = 2, 4, 6, \dots$
6. **Multiply the total sum by**  $\frac{h}{3}$  to get the final integral approximation.

Simpson's Rule is highly accurate for smooth functions like  $f(x) = \sin(x)$ , especially when using many sub-intervals, as it fits parabolas between function points.

## **Differentiation Using Finite Differences**

### **Introduction to Finite Differences**

Finite difference methods are numerical techniques used to approximate derivatives of a function. The key idea is to use the difference between function values at nearby points to estimate the rate of change, i.e., the derivative.

### **Types of Finite Differences**

1. **Forward Difference:**

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

This uses a small step  $h$  to calculate the change in the function from  $x$  to  $x + h$ .

## 2. Backward Difference:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

This method looks at the change in the function from  $x - h$  to  $x$ .

## 3. Central Difference:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

The central difference is more accurate than the forward or backward difference because it uses points on both sides of  $x$ .

### Algorithm for Finite Differences

#### 1. Input:

- A continuous function  $f(x)$  .
- A point  $x$  where the derivative is to be estimated.
- A small step size  $h$ .

#### 2. Forward Difference:

- Compute  $f(x+h)$  and subtract  $f(x)$  .
- Divide the result by  $h$  to estimate  $f'(x)$ .

#### 3. Backward Difference:

- Compute  $f(x)$  and subtract  $f(x-h)$ .
- Divide the result by  $h$ .

#### 4. Central Difference:

- Compute  $f(x+h)$  and subtract  $f(x-h)$ .
- Divide the result by  $2h$ .

#### 5. Output:

- An approximation to the derivative  $f'(x)$ .

## Python Code Implementation

Here is the Python code that implements forward, backward, and central differences to estimate the derivative of a function  $f(x)$ .

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function whose derivative we are computing
def f(x):
    return np.sin(x)

# Finite Difference Method Implementations
def forward_difference(f, x, h):
    return (f(x + h) - f(x)) / h

def backward_difference(f, x, h):
    return (f(x) - f(x - h)) / h

def central_difference(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

# Parameters
x_point = np.pi / 4 # Point at which to approximate the derivative
h = 0.01 # Step size

# Calculate derivatives using finite differences
forward_diff = forward_difference(f, x_point, h)
backward_diff = backward_difference(f, x_point, h)
central_diff = central_difference(f, x_point, h)

# Exact derivative of sin(x) is cos(x), calculate the exact value
exact_derivative = np.cos(x_point)

# Output the results
print(f"Forward Difference Approximation: {forward_diff}")
print(f"Backward Difference Approximation: {backward_diff}")
print(f"Central Difference Approximation: {central_diff}")
print(f"Exact Derivative (cos(x)): {exact_derivative}")

# Plotting the function and the tangent at x = pi/4
x = np.linspace(0, np.pi, 100)
y = f(x)
```

```

plt.figure(figsize=(8,6))
plt.plot(x, y, label='sin(x)', color='blue')
plt.scatter(x_point, f(x_point), color='red', zorder=5)
plt.title(f'Finite Difference Approximations at x = {x_point:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)

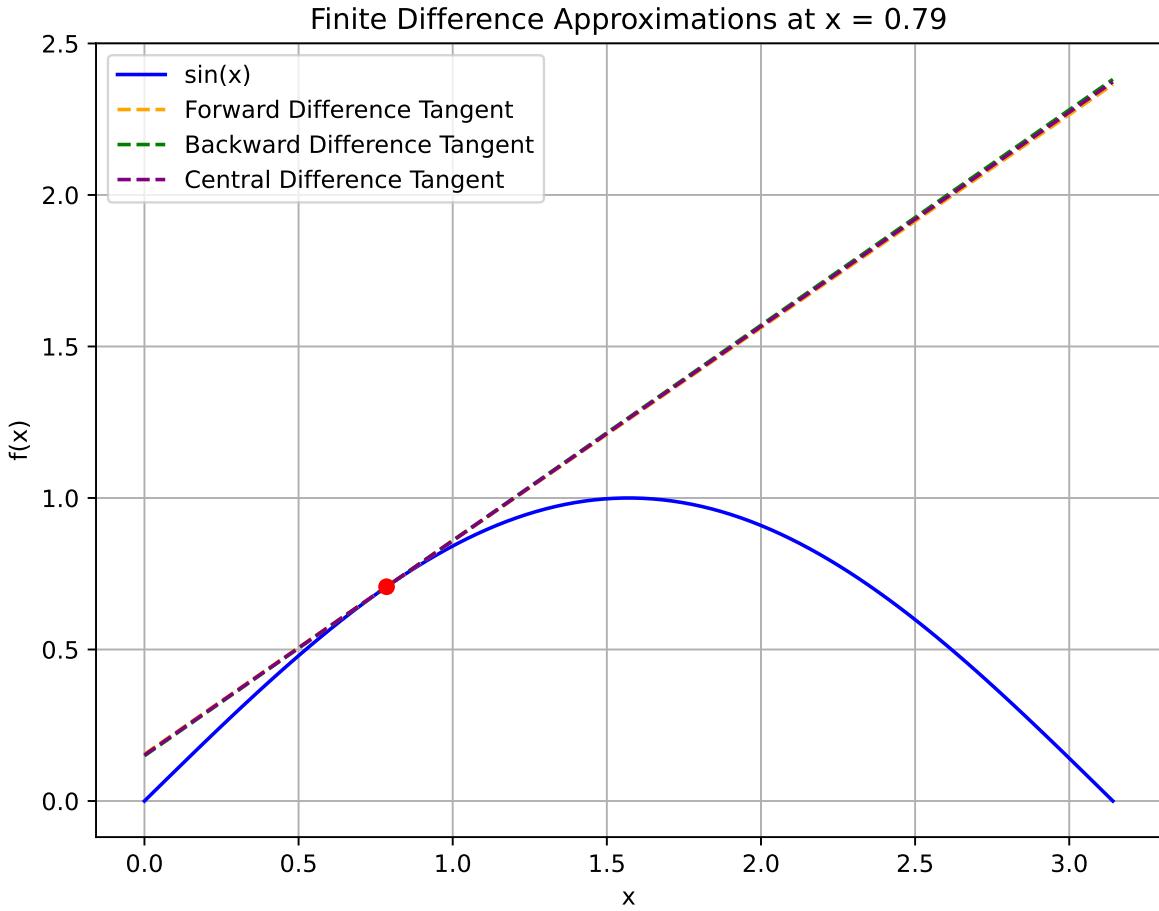
# Adding tangent lines for visualization
tangent_forward = forward_diff * (x - x_point) + f(x_point)
tangent_backward = backward_diff * (x - x_point) + f(x_point)
tangent_central = central_diff * (x - x_point) + f(x_point)

plt.plot(x, tangent_forward, '--',
         label='Forward Difference Tangent', color='orange')
plt.plot(x, tangent_backward, '--',
         label='Backward Difference Tangent', color='green')
plt.plot(x, tangent_central, '--',
         label='Central Difference Tangent', color='purple')

plt.legend()
plt.show()

```

Forward Difference Approximation: 0.7035594916891985  
 Backward Difference Approximation: 0.7106305005757152  
 Central Difference Approximation: 0.7070949961324569  
 Exact Derivative ( $\cos(x)$ ): 0.7071067811865476



## Explanation of the Code

### 1. Function Definition:

```
def f(x):
    return np.sin(x)
```

- This defines the function  $f(x) = \sin x$ , which we will use to demonstrate the finite difference methods.

### 2. Finite Difference Functions:

```
def forward_difference(f, x, h):
    return (f(x + h) - f(x)) / h
```

- **Forward Difference:** This computes the difference  $f(x + h) - f(x)$  and divides by  $h$  to approximate the derivative.

```
def backward_difference(f, x, h):
    return (f(x) - f(x - h)) / h
```

- **Backward Difference:** This computes  $f(x) - f(x - h)$  and divides by  $h$ , which looks at the rate of change from a point behind  $x$ .

```
def central_difference(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)
```

- **Central Difference:** This uses both  $f(x + h)$  and  $f(x - h)$ , taking their difference and dividing by  $2h$ , providing a more accurate estimate.

### 3. Parameter Setup and Derivative Calculation:

```
x_point = np.pi / 4 # Point at which to approximate the derivative
h = 0.01 # Step size
```

- $x\_point$ : The point at which we are approximating the derivative,  $\pi/4$ .
- $h$ : Step size, which controls how close the points are in the finite difference calculation.

### 4. Calculating the Approximations:

```
forward_diff = forward_difference(f, x_point, h)
backward_diff = backward_difference(f, x_point, h)
central_diff = central_difference(f, x_point, h)
```

- These lines compute the forward, backward, and central differences, giving approximations of the derivative at  $\pi/4$ .

### 5. Exact Derivative for Comparison:

```
exact_derivative = np.cos(x_point)
```

- Since  $f(x) = \sin x$ , the exact derivative  $f'(x) = \cos(x)$  is calculated to compare the finite difference approximations with the true value.

## 6. Plotting:

```
# Plotting the function and the tangent at x = pi/4
x = np.linspace(0, np.pi, 100)
y = f(x)
```

- This section defines the range of  $x$  values and computes  $f(x)$  for plotting the function.

## 7. Adding Tangent Lines:

```
tangent_forward = forward_diff * (x - x_point) + f(x_point)
tangent_backward = backward_diff * (x - x_point) + f(x_point)
tangent_central = central_diff * (x - x_point) + f(x_point)
```

- These lines calculate the tangent lines for the forward, backward, and central differences. These lines are visual approximations of the derivative at  $x = \pi/4$ .

## Output:

The output consists of:

1. **Approximate Derivatives** using Forward, Backward, and Central Difference methods.
2. **Exact Derivative:** This is compared with the finite difference approximations.
3. **Plot:** Shows  $\sin(x)$  along with the tangent lines for the three different finite difference methods.

## Remarks

- **Forward Difference** tends to underestimate the derivative, as it only looks forward.
- **Backward Difference** tends to overestimate the derivative, as it looks backward.
- **Central Difference** is generally more accurate as it averages values from both directions.

Finite differences provide a simple and effective way to numerically estimate derivatives, especially when dealing with discrete data or functions that are difficult to differentiate analytically.

# Monte Carlo Integration Using Random Sampling

## Introduction to Random Sampling

Random sampling is a technique in which a set of random values is generated from a probability distribution or within a specified range. This process allows for estimating various properties of a function, distribution, or dataset without needing exhaustive calculations.

The key idea behind random sampling is that if we sample enough points from a domain, the average of those points can be used to approximate an integral or the expectation of a function.

### Key Concepts in Random Sampling:

#### 1. Uniform Distribution:

- In numerical integration, points are often sampled uniformly over an interval or region. For example, to estimate the value of an integral over  $[a, b]$  we can generate random points in that interval using a uniform distribution.

The **probability density function (PDF)** of a continuous uniform distribution is defined as:

$$f(x) = \frac{1}{b-a} \quad \text{for } a \leq x \leq b$$

where:

–  $a$  and  $b$  are the lower and upper limits of the distribution, respectively.

- Example of Uniform Distribution in Python Let's generate random samples from a uniform distribution using numpy and visualize it with matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters of the uniform distribution
a = 0 # lower bound
b = 1 # upper bound
N = 10000 # number of samples

# Generate random samples from uniform distribution
uniform_samples = np.random.uniform(a, b, N)
```

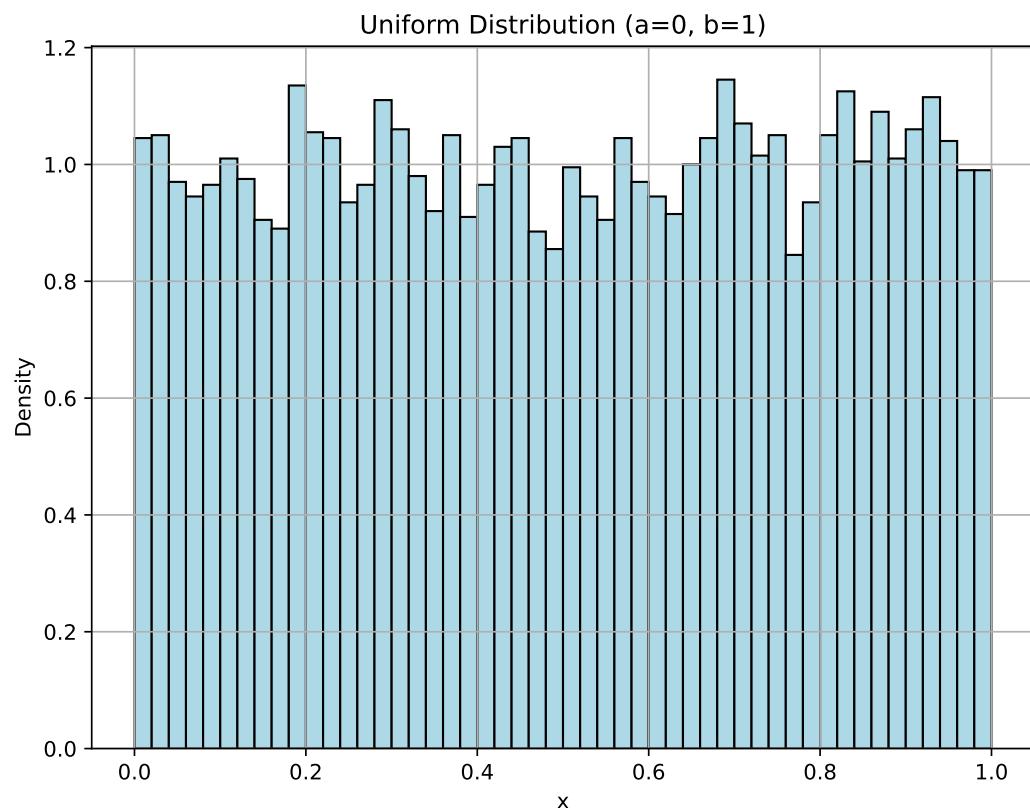
```

# Plot the histogram of the samples
plt.figure(figsize=(8, 6))
plt.hist(uniform_samples, bins=50,
         density=True, color='lightblue',
         edgecolor='black')

# Add labels and title
plt.title(f'Uniform Distribution (a={a}, b={b})')
plt.xlabel('x')
plt.ylabel('Density')

# Show the plot
plt.grid(True)
plt.show()

```



### **Explanation of the Code:**

1. **Parameters:**

- $a = 0$  and  $b = 1$ : The uniform distribution is defined over the interval  $[0, 1]$ , meaning any value in this range has equal probability.
- $N = 10000$ : We generate 10,000 random samples from this uniform distribution.

## 2. Generating Samples:

- `np.random.uniform(a, b, N)`: This generates  $N$  random samples uniformly distributed between  $a$  and  $b$ . Each sample is equally likely to be any value between 0 and 1.

## 3. Plotting:

- The **histogram** shows the distribution of these random samples. The height of the bars represents the density of the samples in that interval, which should be approximately constant for a uniform distribution.
- **density=True** ensures the histogram is normalized to represent the probability density.

### Output:

The plot would show a nearly flat histogram, illustrating that every value within the interval  $[0, 1]$  is equally likely, which is a defining characteristic of the uniform distribution.

## 2. Law of Large Numbers:

- This is a fundamental concept behind random sampling. It states that as the number of trials or random points increases, the sample mean tends to converge to the expected value (the true value of the integral or expectation).

In simpler terms, the more you repeat a random experiment (like flipping a coin or rolling a die), the closer the average of the results will be to the theoretical expected value.

### Mathematical Statement:

Let  $X_1, X_2, X_3, \dots, X_n$  be a sequence of independent and identically distributed random variables with a common expected value  $\mu$ . The **Law of Large Numbers** states:

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n X_i = \mu$$

In other words, as  $n$  increases, the sample mean  $\frac{1}{n} \sum_{i=1}^n X_i$  will approach the expected value  $\mu$

### Example of Law of Large Numbers in Python

Let's simulate the Law of Large Numbers using Python by generating random samples from a uniform distribution, and observe how the sample mean converges to the expected value.

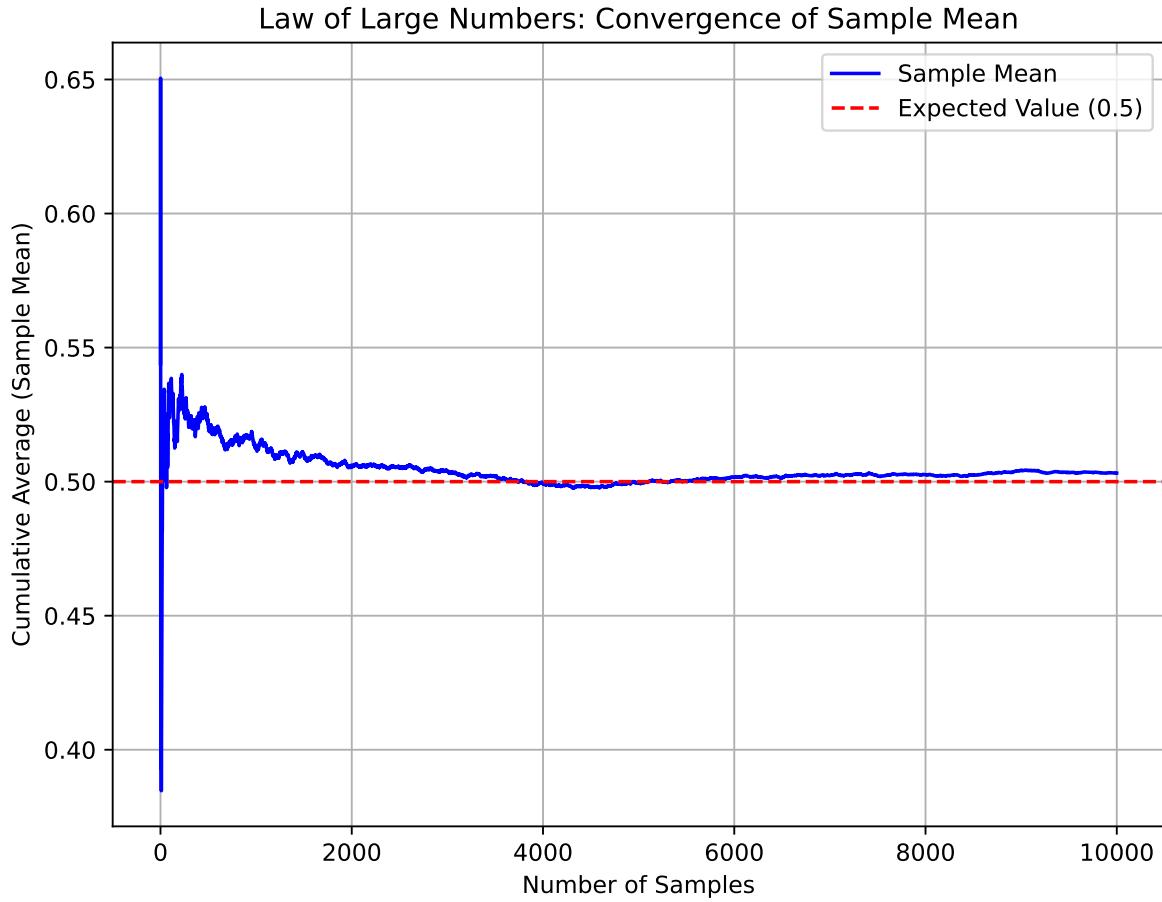
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters of the uniform distribution
a = 0 # lower bound
b = 1 # upper bound
N = 10000 # total number of samples

# Generate random samples from uniform distribution
uniform_samples = np.random.uniform(a, b, N)

# Compute cumulative average (sample mean) at each step
cumulative_average = np.cumsum(uniform_samples) / np.arange(1, N + 1)

# Plot the convergence of the sample mean
plt.figure(figsize=(8, 6))
plt.plot(cumulative_average, label='Sample Mean', color='blue')
plt.axhline(y=0.5, color='red', linestyle='--', label='Expected Value (0.5)')
plt.title('Law of Large Numbers: Convergence of Sample Mean')
plt.xlabel('Number of Samples')
plt.ylabel('Cumulative Average (Sample Mean)')
plt.legend()
plt.grid(True)
plt.show()
```



### **Explanation of the Code:**

#### **1. Parameters:**

- $a = 0$  and  $b = 1$ : The uniform distribution is defined over the interval  $[0, 1]$ , with an expected value of  $\mu = 0.5$ .
- $N = 10000$ : We generate 10,000 random samples from this uniform distribution.

#### **2. Generating Samples:**

- `np.random.uniform(a, b, N)` generates  $N$  random samples from the uniform distribution between 0 and 1.

#### **3. Cumulative Average (Sample Mean):**

- `np.cumsum(uniform_samples)` calculates the cumulative sum of the random samples at each step.

- `np.arange(1, N + 1)` creates an array from 1 to N, which is used to divide the cumulative sum by the number of samples at each step, yielding the sample mean.

#### 4. Plotting:

- The blue line represents the **sample mean**, which starts to stabilize around the **expected value** of 0.5 as the number of samples increases.
- The red dashed line represents the **expected value**  $E[X] = 0.5$ , which the sample mean converges to.

#### Output:

The plot shows the sample mean approaching the expected value as the number of samples increases, illustrating the **Law of Large Numbers** in action.

The **Law of Large Numbers** is a vital concept in probability and statistics, ensuring that repeated independent trials of a random variable will yield an average that approaches the expected value. In the above example, we used Python to generate random samples from a uniform distribution and demonstrated how the sample mean converges to the expected value as the number of samples increases.

#### Why Use Random Sampling?

For problems with high-dimensional spaces or irregular domains, traditional numerical integration methods (like the Trapezoidal or Simpson's rule) become impractical. Random sampling (Monte Carlo methods) can handle these cases because it doesn't depend on the dimensionality or complexity of the domain as much as other methods.

Random sampling is a fundamental concept when applying Monte Carlo methods for integration, simulations, and various probabilistic algorithms. In the context of Monte Carlo Integration, random sampling allows us to estimate integrals by generating random points from a specific range or domain and evaluating a function at those points.

#### Monte Carlo Integration

Monte Carlo integration is a specific application of the Monte Carlo method for numerical integration. The idea is simple:

1. Randomly sample points within the region of integration.
2. Evaluate the function at those random points.
3. Take the average of those function values.

4. Multiply the average by the total area (or volume) of the region.

Mathematically, if we want to compute the integral:

$$I = \int_a^b f(x)dx$$

Using Monte Carlo integration:

$$I \approx \frac{b-a}{N} \sum_{i=1}^N f(x_i)$$

Where:

- $x_i$  are randomly sampled points in the interval  $[a, b]$ .
- $N$  is the number of random points.

The estimate becomes more accurate as  $N$  increases, thanks to the Law of Large Numbers.

Here's a breakdown of how random sampling works in Python and how it's used in the context of Monte Carlo Integration.

### Step-by-Step Explanation of Random Sampling Code

Let's first look at a simplified code snippet for generating random samples and how it integrates with Monte Carlo Integration.

#### Basic Code Example for Random Sampling:

```
import numpy as np

# Step 1: Define the range for sampling (the limits of integration)
a = 0 # Lower limit of integration
b = np.pi # Upper limit of integration

# Step 2: Generate random samples in the range [a, b]
N = 10000 # Number of random points to generate
random_points = np.random.uniform(a, b, N)

# Step 3: Print or examine some of the random points
print(random_points[:10]) # Print the first 10 random points to see the result
```

```
[2.76491047 2.70567009 2.44239101 2.75870425 2.92557288 1.75483595  
 2.6077278 1.44711842 0.47748164 0.67070025]
```

### Explanation of the Code:

#### 1. Importing Necessary Libraries:

- We use `numpy`, one of Python's most powerful libraries for numerical operations. Specifically, `np.random.uniform` will generate random samples from a uniform distribution, meaning all values within the given range  $[a, b]$  have an equal probability of being selected.

#### 2. Defining the Sampling Range:

```
a = 0 # Lower limit of integration  
b = np.pi # Upper limit of integration
```

Here, we define the interval  $[a, b]$  over which we want to integrate or sample. In this example, we set  $a = 0$  and  $b = \pi$ , which is the typical range used when integrating functions like  $\sin x$ .

#### 3. Generating Random Points:

```
N = 10000 # Number of random points to generate  
random_points = np.random.uniform(a, b, N)
```

- `np.random.uniform(a, b, N)` generates  $N$  random points uniformly distributed between the interval  $[a, b]$ .
- Each point in `random_points` is a randomly selected value from this range, making it uniformly distributed.
- $N = 10000$  is the number of samples. The larger  $N$  is, the more accurate our approximation will be, but at the cost of more computational effort.

#### 4. Printing or Inspecting Random Points:

```
print(random_points[:10]) # Print the first 10 random points
```

- To check that the random sampling is working correctly, we print the first 10 points. Each point should be a random number between 0 and  $\pi$ .

### Random Sampling in the Context of Monte Carlo Integration

In Monte Carlo integration, the idea is to sample many random points over the range of integration, evaluate the function at those points, and then use the average of the function evaluations to estimate the integral.

### Monte Carlo Integration Code with Random Sampling:

```
import numpy as np

# Define the function to integrate (e.g., sin(x))
def f(x):
    return np.sin(x)

# Monte Carlo Integration function
def monte_carlo_integration(f, a, b, N):
    # Step 1: Generate random points between a and b
    random_points = np.random.uniform(a, b, N)

    # Step 2: Evaluate the function at those random points
    function_values = f(random_points)

    # Step 3: Calculate the average of the function values
    average_value = np.mean(function_values)

    # Step 4: Multiply the average by (b - a) to estimate the integral
    integral = (b - a) * average_value
    return integral

# Parameters
a = 0 # lower limit
b = np.pi # upper limit
N = 10000 # number of random samples

# Calculate the integral using Monte Carlo Integration
result = monte_carlo_integration(f, a, b, N)
print(f"Monte Carlo Integration Result: {result}")
```

Monte Carlo Integration Result: 2.0113458171703917

### Detailed Explanation of the Monte Carlo Integration Code:

#### 1. Define the Function to Integrate:

```
# Define the function to integrate (e.g., sin(x))
def f(x):
    return np.sin(x)
```

- This function represents  $f(x) = \sin(x)$ , which we will integrate over the interval  $[0, \pi]$ .

## 2. Generate Random Points (Random Sampling):

```
random_points = np.random.uniform(a, b, N)
```

- This is the critical step where random sampling occurs. Here,  $N = 10000$  random points are generated between  $a = 0$  and  $b = \pi$ .

## 3. Evaluate the Function at Random Points:

```
function_values = f(random_points)
```

- After generating the random points, we evaluate the function  $f(x)$  at each of these points. This gives us an array of function values corresponding to each randomly generated  $x$ .

## 4. Compute the Average Value:

```
average_value = np.mean(function_values)
```

- Once we have the function values for the random points, we compute the average. This is crucial because the average function value gives us a sense of how the function behaves over the entire interval  $[a, b]$ .

## 5. Estimate the Integral:

```
integral = (b - a) * average_value
```

- The final step in Monte Carlo integration is to multiply the average function value by the width of the interval  $(b - a)$ . This gives the estimated value of the integral:

$$f(xi) \int_a^b f(x)dx \approx (b - a) \cdot \frac{1}{N} \sum_{i=1}^N f(x_i)$$

## 6. Result:

```
print(f"Monte Carlo Integration Result: {result}")
```

- The computed result is printed, which is an estimate of the integral  $\int_0^\pi \sin(x)dx$ , which should be close to 2 as the exact value is known to be 2.

## Visualizing Random Sampling in Monte Carlo Integration

Here's a small addition to visualize the random points in the context of Monte Carlo Integration:

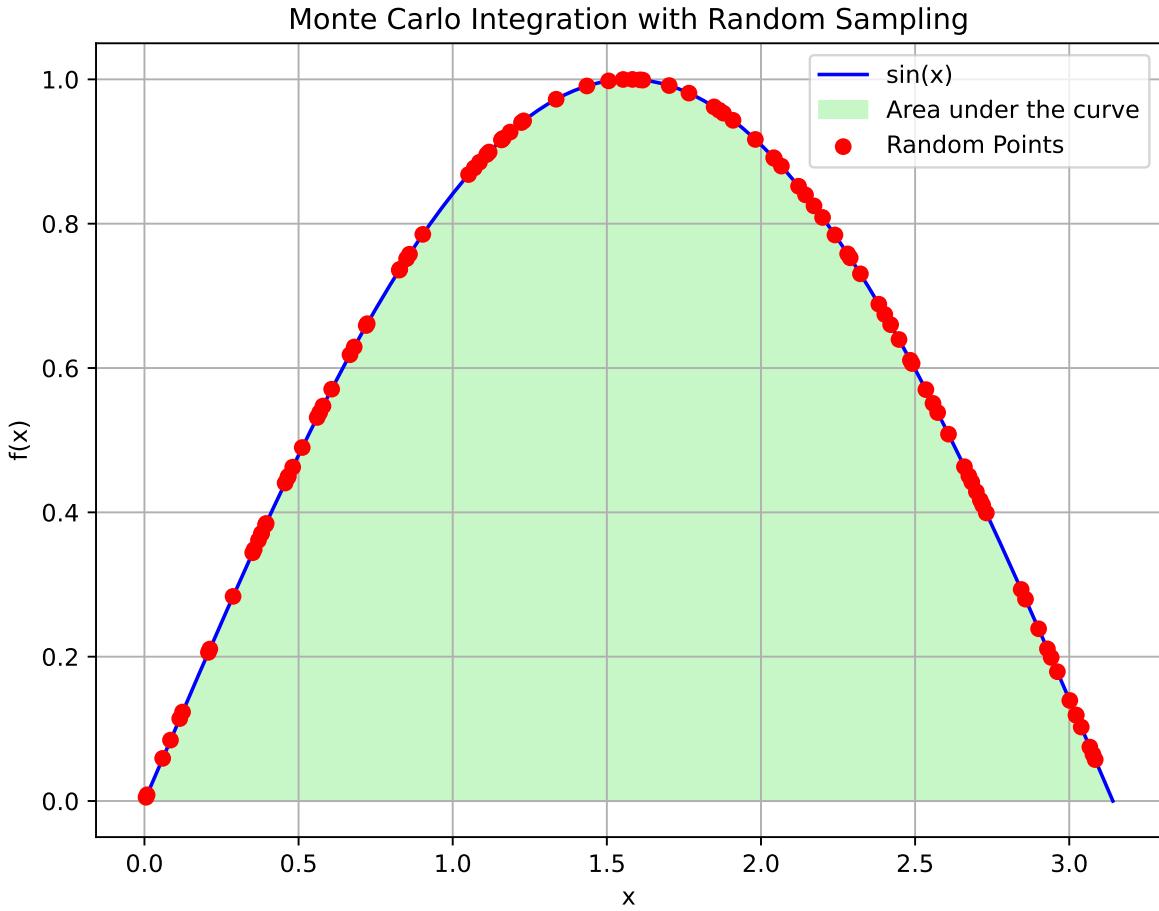
```
import matplotlib.pyplot as plt

# Generate some random points
random_x = np.random.uniform(a, b, 100) # 100 random points for visualization
random_y = f(random_x) # Evaluate f(x) at these random points

# Plotting the function and random points
x = np.linspace(a, b, 1000)
y = f(x)

plt.figure(figsize=(8,6))
plt.plot(x, y, 'b-', label='sin(x)')
plt.fill_between(x, y, color='lightgreen',
                 alpha=0.5,
                 label='Area under the curve')
plt.scatter(random_x, random_y,
            color='red', zorder=5,
            label='Random Points')

plt.title('Monte Carlo Integration with Random Sampling')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```



#### Explanation of Visualization Code:

- **random\_x and random\_y:** We generate a smaller number of random points(  $N = 100$  ) for better visualization and plot these points as red dots.
- **Plot:** The plot shows the function  $\sin x$  as a curve, with the area under the curve shaded. The red points represent the random samples used to approximate the integral.

Random sampling plays a crucial role in Monte Carlo integration by generating points that allow us to approximate integrals in an unbiased way. The more random points we use, the closer the approximation gets to the true value, thanks to the Law of Large Numbers. The randomness makes this method particularly useful in high-dimensional or complex domains where traditional methods fail.

## Conclusion

These numerical techniques are crucial in scientific computing as they allow us to:

- Solve complex real-world problems that lack closed-form solutions.
- Model phenomena across various scientific disciplines.
- Handle high-dimensional and non-linear systems where analytical methods fail.

Python, with its rich ecosystem of libraries such as `NumPy`, `SciPy`, and `Matplotlib`, offers an excellent platform for implementing these methods efficiently, providing both ease of coding and computational power for large-scale problems.

*thank  
you*