LAB 5

1.Trapezoidal rule

Working Principle:
The Trapezoidal Rule is a numerical method for approximating the definite integral of a function. It works by dividing the area under the curve into trapezoidal segments rather than rectangles (as in the Riemann sum method). The area of each trapezoid is calculated and summed to provide an estimate for the total area under the curve.

Formula for Trapezoidal Rule:
Given a function f(x) the integral of f(x) over the interval [a,b] is approximated by:

$$I \approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right)$$

Where:

- $h = \frac{b-a}{n} \text{ is the width of each subinterval.}$
- $x_i = a + i \cdot h \; for \; i = 1, 2, \ldots, n \text{ are the intermediate points.}$
- f(a) and f(b) are the function values at the endpoints of the interval.

Steps for Trapezoidal Rule:
Divide the interval $[a, b]$ [a,b] into n n subintervals. Compute the function values at the endpoints and at intermediate points.
Calculate the area of each trapezoid formed between consecutive points.
Sum the areas of all trapezoids to obtain the approximate integral.

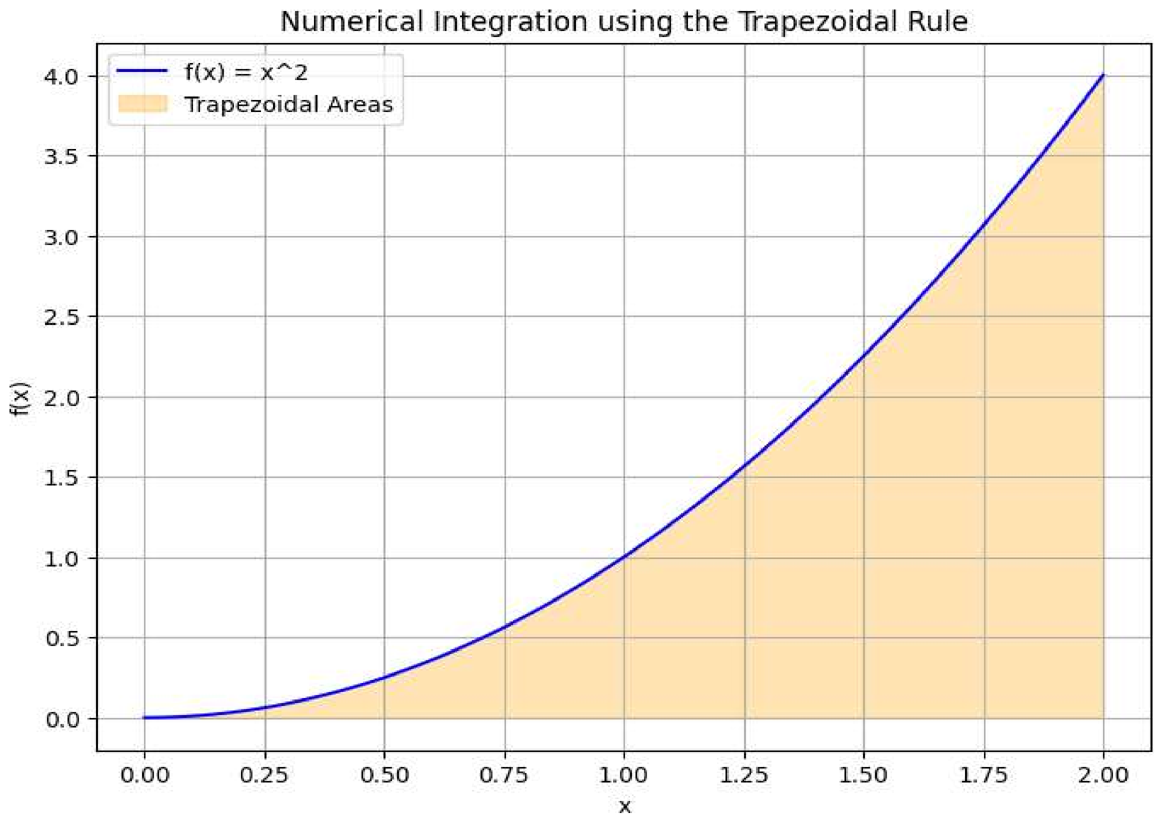
Pseudocode: Input: Function f(x), interval [a, b], number of subintervals n Output: Approximate integral I

1. Calculate the width of each subinterval: h = (b - a) / n
2. Initialize sum to 0: sum = 0
3. Loop through the intermediate points: for i = 1 to n-1: x = a + i * h sum = sum + f(x)
4. Add the function values at the endpoints to the sum: sum = sum + (f(a) + f(b)) / 2
5. Multiply the sum by the width of each subinterval: I = h * sum
6. Return the approximate integral I.

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

# Function to calculate the integral using the trapezoidal rule
def trapezoidal_rule(f, a, b, n):
    # Calculate the width of each subinterval
    h = (b - a) / n

    # Initialize sum
    sum = 0.5 * (f(a) + f(b))

    # Loop through the intermediate points
    for i in range(1, n):
        x = a + i * h
        sum += f(x)

    # Multiply the sum by the width of each subinterval
    integral = h * sum
    return integral

# Example function to integrate: f(x) = x^2
def example_function(x):
    return x**2

# Test the trapezoidal rule
a = 0  # Lower limit
b = 2  # Upper limit
n = 10  # Number of subintervals

# Calculate the approximate integral
approx_integral = trapezoidal_rule(example_function, a, b, n)
print(f"Approximate integral using Trapezoidal Rule:
{approx_integral}")

# Exact integral (for comparison)
```

```python
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization of the function and the trapezoidal rule
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

# Plot the function
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='f(x) = x^2', color='blue')

# Plot the trapezoids
x_trap = np.linspace(a, b, n+1)
y_trap = example_function(x_trap)
plt.fill_between(x_trap, 0, y_trap, color='orange', alpha=0.3,
label='Trapezoidal Areas')

# Labels and legend
plt.title('Numerical Integration using the Trapezoidal Rule')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Approximate integral using Trapezoidal Rule: 2.6800000000000006
Exact integral: 2.6666666666666665
```




```
import sympy as sp

def f(x):
    return 1 / (1 + x**2)

def func_input():
    function_str = input("Enter your function (use 'x' as the variable)
(Example: 1/x): ")
    x = sp.symbols('x')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify(x, sp_function, modules=['numpy'])
    return func, sp_function

def trapezoidal(func, x0, xn, n):
    # Calculate step size
    h = (xn - x0) / n

    # Compute the initial and final function values
    integration = func(x0) + func(xn)

    # Summing function values at internal points
    for i in range(1, n):
        k = x0 + i * h
        integration += 2 * func(k)
```

```python
    # Final integration value
    integration *= h / 2
    return integration

def main():
    print("Trapezoidal Rule for Numerical Integration")
    print()

    function_str = "1 / (1 + x**2)"
    func = f
    lower_limit = 0
    upper_limit = 1
    sub_interval = 6


    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        lower_limit = float(input("Enter lower limit of integration: "))
        upper_limit = float(input("Enter upper limit of integration: "))
        sub_interval = int(input("Enter number of subintervals: "))

    print(f"f(x) = {function_str}")
    print(f"Lower Limit: {lower_limit}")
    print(f"Upper Limit: {upper_limit}")
    print(f"Subintervals: {sub_interval}")
    print()

    result = trapezoidal(func, lower_limit, upper_limit, sub_interval)

    print("Integration Result:")
    print(f"Using the Trapezoidal Method, the approximate value is:
{result:.6f}")

if __name__ == "__main__":
    main()
```

Trapezoidal Rule for Numerical Integration

Use default limits? (y/n): y
f(x) = 1 / (1 + x**2)
Lower Limit: 0
Upper Limit: 1
Subintervals: 6

Integration Result:
    Using the Trapezoidal Method, the approximate value is: 0.784241

Test Case:

Test Case 1:

- Function: f(x)=x^2
- Interval: [0,2][0, 2]
- Number of subintervals: n=10
- Expected Output: The approximate integral using the trapezoidal rule should be close to the exact integral 8/3 = 2.666738.

Test Case 2:

- Function: f(x)=e^x
- Interval: [0,1]
- Number of subintervals: n=20
- Expected Output: The approximate integral should be close to the exact value of $e^x$from 0 to 1, which is approximately 1.71828.

2.Simpson's 1/3 rule or Simpson's 3/8 rule

Working Principle
Simpson's 1/3 Rule:
Simpson's 1/3 Rule is a method of numerical integration that approximates the integral of a function by dividing the area under the curve into a series of parabolic segments. It uses quadratic polynomials to estimate the area.

The formula for Simpson's 1/3 Rule is:

$$I \approx \frac{h}{3}\left(f(a) + 4 \sum_{\{i=1,3,5,\dots\}} f(x_i) + 2 \sum_{\{i=2,4,6,\dots\}} f(x_i) + f(b) \right)$$

Where:

- h=$\frac{b-a}{n}$ is the width of each subinterval.
- x_i are the intermediate points, with odd indices receiving a weight of 4 and even indices receiving a weight of 2.

Simpson's 3/8 Rule:

Simpson's 3/8 Rule is another method for approximating the definite integral of a function, and it is a bit more accurate than Simpson's 1/3 Rule for certain functions. It approximates the integral by fitting cubic polynomials to the data.

The formula for Simpson's 3/8 Rule is:

$$I \approx \frac{3h}{8}\left(f(a) + 3 \sum_{\{i=1,4,7,\dots\}} f(x_i) + 3 \sum_{\{i=2,5,8,\dots\}} f(x_i) + f(b) \right)$$

Where:

- h=$\frac{b-a}{n}$ is the width of each subinterval.

Pseudo code:simpson's1/3 rule:

Input: Function f(x), interval [a, b], number of subintervals n (n must be even) Output: Approximate integral I

1. Calculate the width of each subinterval: h = (b - a) / n
2. Initialize sum: sum = f(a) + f(b)
3. Loop through the odd indexed points and add weighted contributions: for i = 1, 3, 5, ..., n-1: sum += 4 * f(a + i * h)

4. Loop through the even indexed points and add weighted contributions: for i = 2, 4, 6, ..., n-2: sum += 2 * f(a + i * h)

5. Multiply the sum by h/3 to get the integral:

   I = (h / 3) * sum

6. Return the approximate integral I

Pseudocode:simpson's 3/8 rule

Input: Function f(x), interval [a, b], number of subintervals n (n must be a multiple of 3) Output: Approximate integral I

1. Calculate the width of each subinterval: h = (b - a) / n

2. Initialize sum: sum = f(a) + f(b)

3. Loop through the points and add weighted contributions: for i = 1, 4, 7, ..., n-2: sum += 3 * f(a + i * h)

4. Loop through the points and add weighted contributions: for i = 2, 5, 8, ..., n-1: sum += 3 * f(a + i * h)

5. Multiply the sum by 3h/8 to get the integral:

   I = (3 * h / 8) * sum

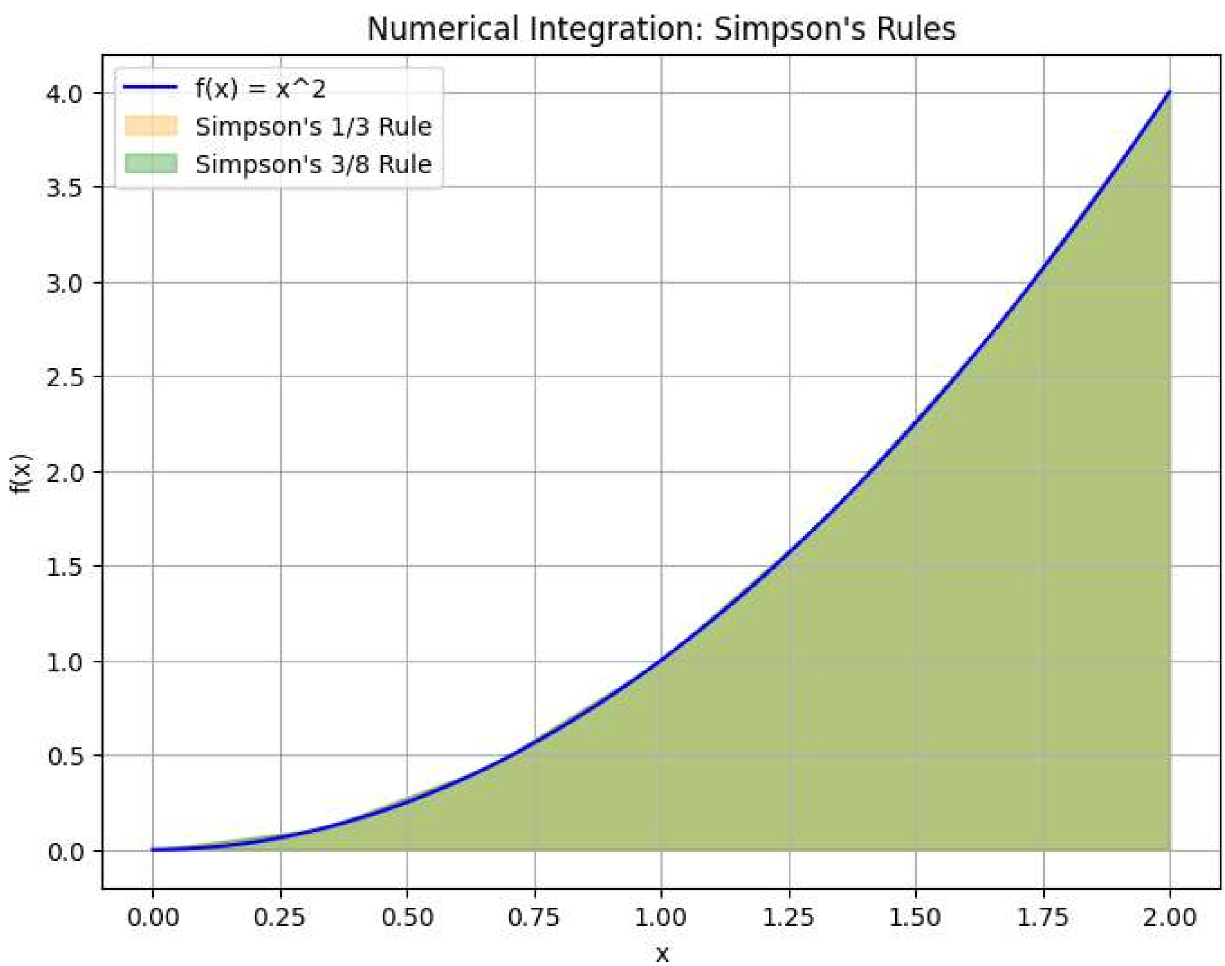
6. Return the approximate integral I

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

# Function for Simpson's 1/3 Rule
def simpsons_1_3_rule(f, a, b, n):
    if n % 2 == 1:  # n must be even
        n += 1
    h = (b - a) / n
    sum = f(a) + f(b)
    for i in range(1, n, 2):
        sum += 4 * f(a + i * h)
    for i in range(2, n-1, 2):
        sum += 2 * f(a + i * h)
    return (h / 3) * sum

# Function for Simpson's 3/8 Rule
def simpsons_3_8_rule(f, a, b, n):
    if n % 3 != 0:  # n must be a multiple of 3
        n += 3 - (n % 3)
    h = (b - a) / n
    sum = f(a) + f(b)
    for i in range(1, n, 3):
        sum += 3 * f(a + i * h)
```

```
    for i in range(2, n-1, 3):
        sum += 3 * f(a + i * h)
    return (3 * h / 8) * sum

# Example function to integrate
def example_function(x):
    return x**2

# Test the methods
a = 0  # Lower limit
b = 2  # Upper limit
n = 6  # Number of subintervals for Simpson's 1/3 Rule (even)

# Calculate using Simpson's 1/3 Rule
approx_integral_1_3 = simpsons_1_3_rule(example_function, a, b, n)
print(f"Approximate integral using Simpson's 1/3 Rule:
{approx_integral_1_3}")

# Test Simpson's 3/8 Rule
n_38 = 6  # Number of subintervals for Simpson's 3/8 Rule (multiple of
3)
approx_integral_3_8 = simpsons_3_8_rule(example_function, a, b, n_38)
print(f"Approximate integral using Simpson's 3/8 Rule:
{approx_integral_3_8}")

# Exact integral for comparison
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='f(x) = x^2', color='blue')

# Plot the areas for Simpson's 1/3 Rule
x_simpson_1_3 = np.linspace(a, b, n+1)
y_simpson_1_3 = example_function(x_simpson_1_3)
plt.fill_between(x_simpson_1_3, 0, y_simpson_1_3, color='orange',
alpha=0.3, label="Simpson's 1/3 Rule")

# Plot the areas for Simpson's 3/8 Rule
x_simpson_3_8 = np.linspace(a, b, n_38+1)
y_simpson_3_8 = example_function(x_simpson_3_8)
plt.fill_between(x_simpson_3_8, 0, y_simpson_3_8, color='green',
alpha=0.3, label="Simpson's 3/8 Rule")

# Labels and legend
plt.title("Numerical Integration: Simpson's Rules")
```

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

```
Approximate integral using Simpson's 1/3 Rule: 2.6666666666666665
Approximate integral using Simpson's 3/8 Rule: 1.375
Exact integral: 2.6666666666666665
```




```
import sympy as sp

def f(x):
    return 1 / (1 + x**2)

def func_input():
    function_str = input("Enter your function (use 'x' as the variable)
(Example: 1/x): ")
    x = sp.symbols('x')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify(x, sp_function, modules=['numpy'])
    return func, sp_function
```

```python
def simpson13(func, x0, xn, n):
    if n % 2 != 0:
        raise ValueError("Number of subintervals (n) must be even.")

    # Calculate step size
    h = (xn - x0) / n

    # Compute the initial and final function values
    integration = func(x0) + func(xn)

    # Summing function values at internal points
    for i in range(1, n):
        k = x0 + i * h
        if i % 2 == 0:
            integration += 2 * func(k)
        else:
            integration += 4 * func(k)

    # Final integration value
    integration *= h / 3
    return integration

def main():
    print("Simpson's 1/3 Rule for Numerical Integration")
    print()

    function_str = "1 / (1 + x**2)"
    func = f
    lower_limit = 0
    upper_limit = 1
    sub_interval = 6

    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        lower_limit = float(input("Enter lower limit of integration: "))
        upper_limit = float(input("Enter upper limit of integration: "))
        sub_interval = int(input("Enter number of subintervals (must be even):
"))

    print(f"f(x) = {function_str}")
    print(f"Lower Limit: {lower_limit}")
    print(f"Upper Limit: {upper_limit}")
    print(f"Subintervals: {sub_interval}")
    print()

    result = simpson13(func, lower_limit, upper_limit, sub_interval)
    print("Integration Result:")
```

```
    print(f"Using Simpson's 1/3 method, the approximate value is:
{result:.6f}")

if __name__ == "__main__":
    main()
```

Simpson's 1/3 Rule for Numerical Integration

Use default limits? (y/n): y
f(x) = 1 / (1 + x**2)
Lower Limit: 0
Upper Limit: 1
Subintervals: 6

Integration Result:
Using Simpson's 1/3 method, the approximate value is: 0.785398

3.Boole’s Rule or Weddle’s Rule

Working Principle

Boole’s Rule (Weddle’s Rule) is a higher-order numerical integration method that approximates the integral of a function using a polynomial of degree four (quartic polynomial). It is a specific case of a more general family of Newton-Cotes formulas.

- Formula: The formula for Boole's Rule (also known as Weddle's Rule) for approximating the integral of a function f(x)f(x)f(x) over the interval [a,b] is given by:
- $\int_a^b f(x)\ dx \approx \frac{2(b-a)}{45}\left[7f(a) + 32f\left(\frac{a+b}{2}\right) + 12f\left(\frac{a+3b}{4}\right) + 32f\left(\frac{3a+b}{4}\right) + 7f(b)\right]$
- Description:
    - Step 1: Divide the integral into subintervals.
    - Step 2: Use weighted averages of function values at specific points in the interval to approximate the area under the curve.
    - Step 3: The result gives a good approximation with high accuracy for polynomial functions and smooth curves.

Pseudocode: Algorithm: Boole's Rule Integration

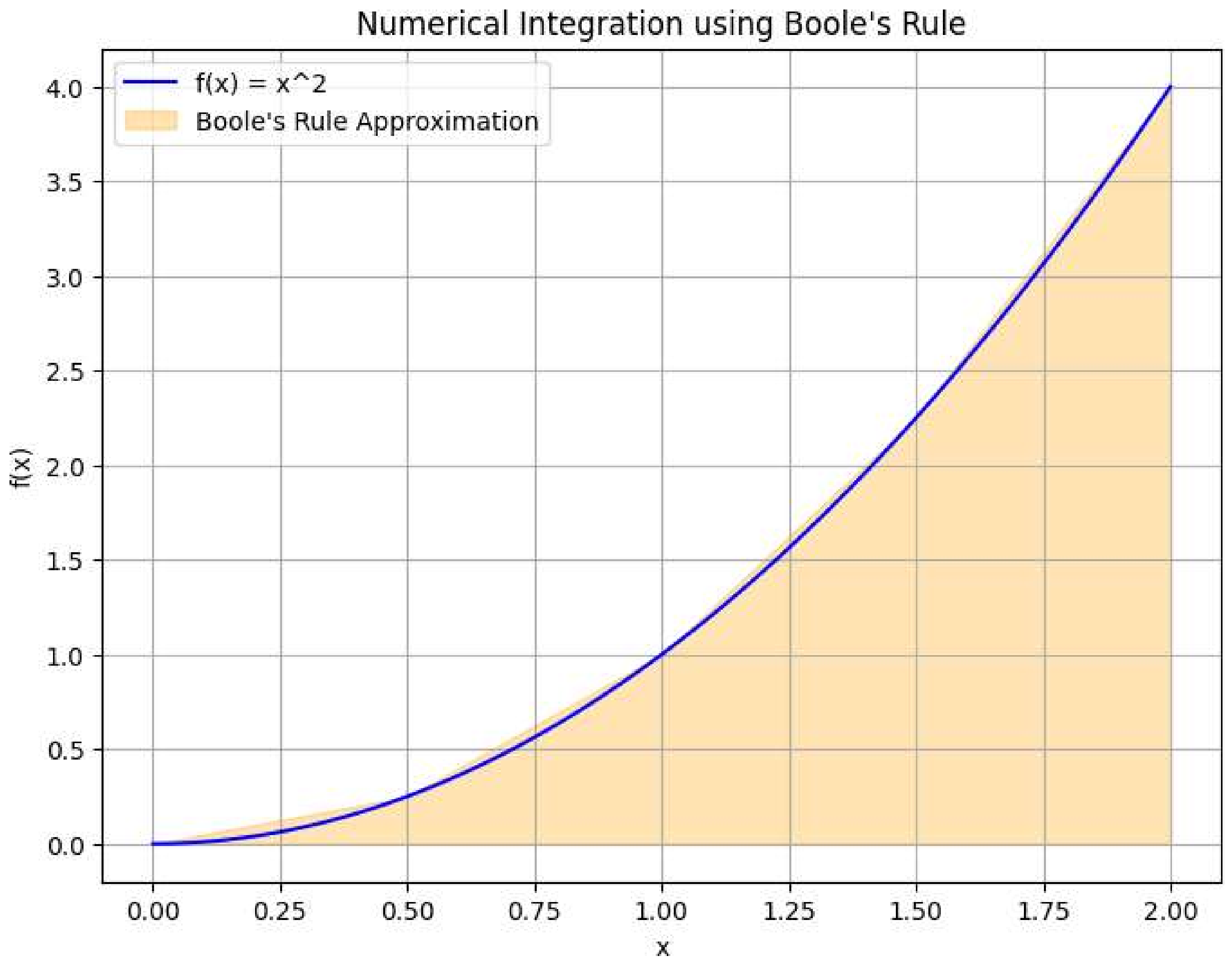
Input: Function f(x), Lower limit a, Upper limit b Output: Approximate integral value

1. Define the function f(x) to be integrated
2. Set the limits of integration: a (lower bound), b (upper bound)
3. Compute the midpoints:
    - c = (a + b) / 2 (Midpoint)
    - d = (3a + b) / 4 (Another intermediate point)
    - e = (a + 3b) / 4 (Another intermediate point)
4. Evaluate the function at the points a, b, c, d, e
5. Apply Boole’s Rule formula: result = (b - a) / 45 * [7 * f(a) + 32 * f(c) + 12 * f(d) + 32 * f(e) + 7 * f(b)]
6. Return the result

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

# Define the function to integrate (for example, f(x) = x^2)
def example_function(x):
    return x**2

# Boole's Rule (Weddle's Rule) for numerical integration
def booles_rule_fixed(f, a, b):
    # Divide the interval into four subintervals
    h = (b - a) / 4  # Step size
```

```
    x0 = a
    x1 = a + h
    x2 = a + 2 * h
    x3 = a + 3 * h
    x4 = b

    # Apply Boole's Rule formula
    result = (2 * h / 45) * (7 * f(x0) + 32 * f(x1) + 12 * f(x2) + 32
* f(x3) + 7 * f(x4))
    return result

# Test the function with specific bounds
a = 0  # Lower bound
b = 2  # Upper bound

# Calculate the approximate integral using the corrected Boole's Rule
approx_integral = booles_rule_fixed(example_function, a, b)
print(f"Approximate integral using Boole's Rule: {approx_integral}")

# Exact integral for comparison (for f(x) = x^2, exact integral is
(b^3 - a^3)/3)
exact_integral = (b**3 - a**3) / 3
print(f"Exact integral: {exact_integral}")

# Visualization of the function and the integration area
x_values = np.linspace(a, b, 100)
y_values = example_function(x_values)

# Plot the function
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label="f(x) = x^2", color='blue')

# Shading the area under the curve for integration using Boole's Rule
x_boole = np.array([a, a + (b - a) / 4, a + 2 * (b - a) / 4, a + 3 *
(b - a) / 4, b])
y_boole = example_function(x_boole)
plt.fill_between(x_boole, 0, y_boole, color='orange', alpha=0.3,
label="Boole's Rule Approximation")

# Labels and legend
plt.title("Numerical Integration using Boole's Rule")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Approximate integral using Boole's Rule: 2.666666666666667
Exact integral: 2.6666666666666665
```



Test case:

Test Case 1:

- Function: f(x)=x^2
- Interval: [0, 2]
- Exact Integral: $\frac{2^3-0^3}{3}$≈2.6667
- Expected Output: Approximate integral ≈ 2.6667

Test Case 2:

- Function: f(x)= e^x
- Interval: [0, 1]
- Exact Integral: e^1-e^0=e−1≈1.7183
- Expected Output: Approximate integral ≈ 1.7183

Gauss-Legendre integration

Working Principle

Gauss-Legendre Integration is a numerical method for approximating definite integrals using orthogonal polynomials. It is based on the concept that the integral:

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

can be approximated as:

$$\int_a^b f(x)\ dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i)$$

Here:

- x_i : Roots (nodes) of the Legendre polynomial $P_{n(x))}$.
- w_i: Weights determined for each root x_i.
- n: Degree of the polynomial or the number of nodes used.

The method transforms the interval [a,b] to [−1,1]for computation and uses tabulated weights and nodes.

Pseudocode

1. Input:
   - Function f(x) lower limit a, upper limit b, and number of nodes n.
2. Steps:
   1. Retrieve the roots (x_i) and weights (w_i) for the Legendre polynomial of degree nnn.
   2. Map the roots from the interval [−1,1to [a,b]:

      $$x_m = \frac{b-a}{2} \cdot x_i + \frac{b+a}{2}$$

   3. Scale the weights:

      $$w_m = \frac{b-a}{2} \cdot w_m$$

   4. Compute the weighted sum of the function evaluations:

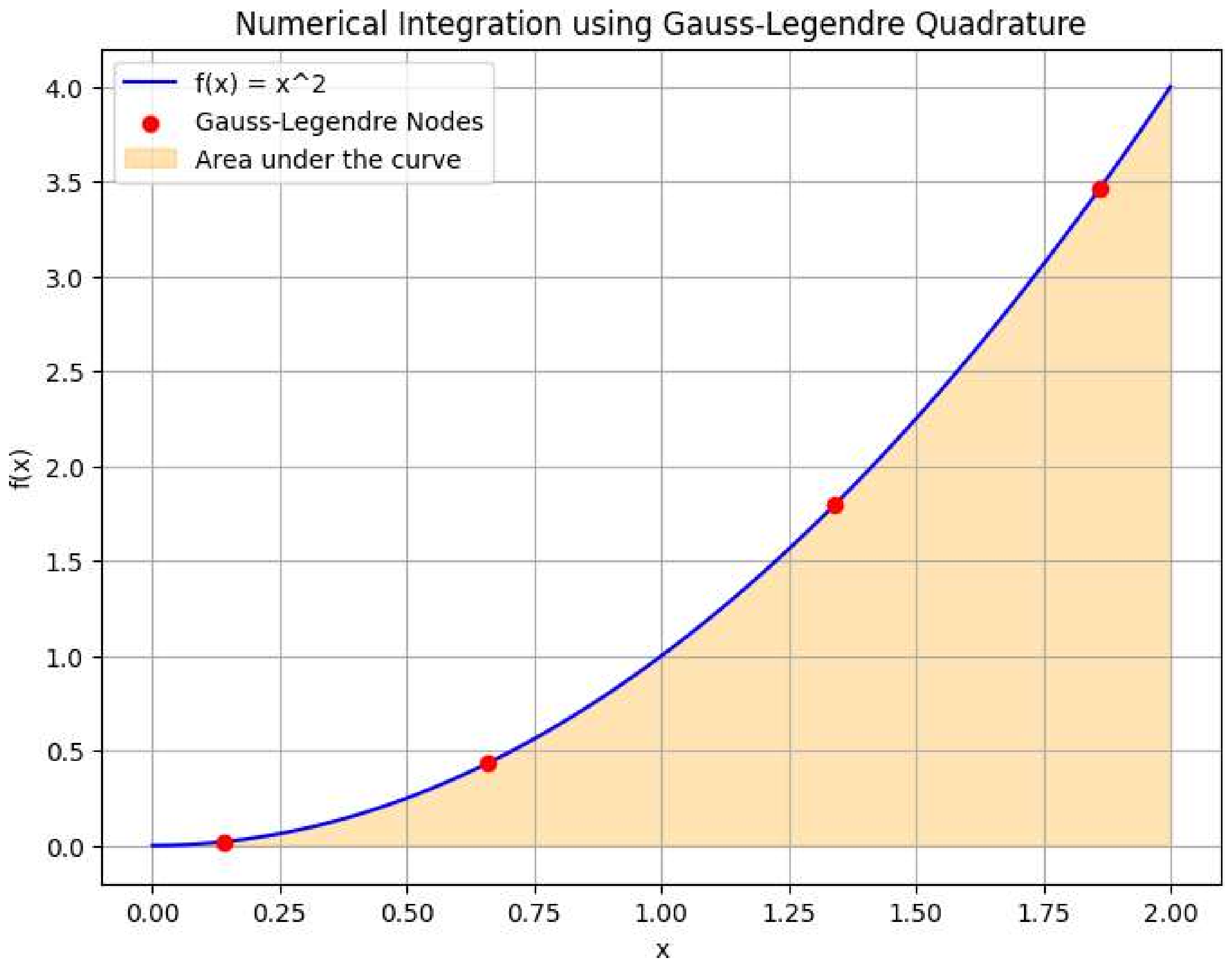
      Integral=$\sum_{\{i=1\}}^{n} w_m \cdot f(x_m)$
3. Output:
   - Approximate integral value.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import roots_legendre
```

```python
# Function to integrate
def function_to_integrate(x):
    return x**2  # Example: f(x) = x^2

# Gauss-Legendre Quadrature implementation
def gauss_legendre_integration(f, a, b, n):
    # Get the roots and weights for Legendre polynomial of degree n
    roots, weights = roots_legendre(n)

    # Map roots and weights to the interval [a, b]
    mapped_roots = 0.5 * (b - a) * roots + 0.5 * (b + a)
    mapped_weights = 0.5 * (b - a) * weights

    # Compute the integral
    integral = np.sum(mapped_weights * f(mapped_roots))
    return integral

# Define the limits and number of nodes
a = 0  # Lower limit
b = 2  # Upper limitn = 4  # Number of nodes (degree of Legendre
polynomial)

# Compute the approximate integral
approx_integral = gauss_legendre_integration(function_to_integrate, a,
b, n)
print(f"Approximate integral using Gauss-Legendre Quadrature:
{approx_integral}")

# Compute the exact integral for comparison
exact_integral = (b**3 - a**3) / 3  # Integral of x^2 is x^3 / 3
print(f"Exact integral: {exact_integral}")

# Visualization
x = np.linspace(a, b, 100)
y = function_to_integrate(x)

# Plot the function and nodes
plt.figure(figsize=(8, 6))
plt.plot(x, y, label="f(x) = x^2", color='blue')

# Plot the Gauss-Legendre nodes
roots, _ = roots_legendre(n)
mapped_roots = 0.5 * (b - a) * roots + 0.5 * (b + a)
plt.scatter(mapped_roots, function_to_integrate(mapped_roots),
color='red', label="Gauss-Legendre Nodes", zorder=5)

# Fill area under the curve
plt.fill_between(x, 0, y, color='orange', alpha=0.3, label="Area under
the curve")

# Labels and legend
plt.title("Numerical Integration using Gauss-Legendre Quadrature")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
```

```
plt.show()
```

```
Approximate integral using Gauss-Legendre Quadrature:
2.6666666666666665
Exact integral: 2.6666666666666665
```



Numerical Integration using Gauss-Legendre Quadrature
f(x) = x^2
Gauss-Legendre Nodes
Area under the curve
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
f(x)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x