Numerical Integration & Differentiation Lab

E/20/168 - Jayasinghe BVRR

15 June 2024

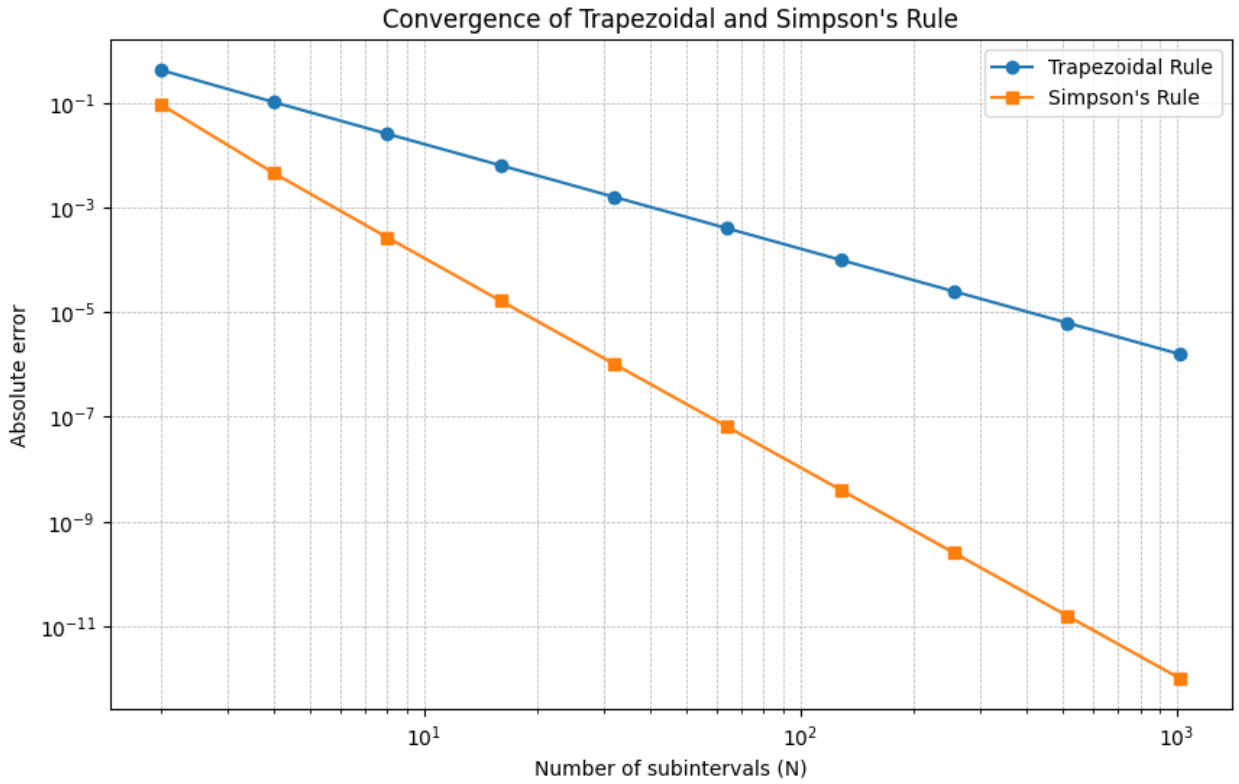
Numerical Integration

❄ Q1

Study the given codes and explain that the absolute error for the composite trapezoidal rule decays at the rate of 1/n^2, and the absolute error for the composite Simpson's rule decays at the rate of 1/n^4, where n is the number of subintervals.

```
# import libs
import matplotlib.pyplot as plt
from scipy import integrate
from numpy import array, linspace, size, log, polyfit
import numpy as np

# Composite Simpson's Rule
def simpson(func, a, b, N):
    if N % 2:
        raise ValueError("N must be even for Simpson's rule.")
    x = np.linspace(a, b, N + 1)
    h = (b - a) / N
    I = (func(x[0]) + func(x[-1]) +
         4 * np.sum(func(x[1:-1:2])) +
         2 * np.sum(func(x[2:-2:2]))) * h / 3.0
    return I

# Composite Trapezoidal Rule
def trapezoidal(func, a, b, N):
    x = np.linspace(a, b, N + 1)
    h = x[1] - x[0]
    I = np.sum(func(x[1:-1])) + 0.5 * (func(x[0]) + func(x[-1]))
    return I * h


import numpy as np
import matplotlib.pyplot as plt

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

# Exact integral of sin(x) from 0 to pi
```

```
exact_integral = 2.0

# Integration bounds
a, b = 0, np.pi

# Range of N values
N_values = np.array([2**k for k in range(1, 11)])  # N = 2, 4, 8, ..., 1024

# Compute errors
trapezoidal_errors = []
simpson_errors = []

for N in N_values:
    I_trapezoidal = trapezoidal(func, a, b, N)
    I_simpson = simpson(func, a, b, N)

    trapezoidal_errors.append(np.abs(I_trapezoidal - exact_integral))
    simpson_errors.append(np.abs(I_simpson - exact_integral))

# Plotting the errors
plt.figure(figsize=(10, 6))
plt.loglog(N_values, trapezoidal_errors, 'o-', label='Trapezoidal Rule')
plt.loglog(N_values, simpson_errors, 's-', label='Simpson\'s Rule')

plt.xlabel('Number of subintervals (N)')
plt.ylabel('Absolute error')
plt.title('Convergence of Trapezoidal and Simpson\'s Rule')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```



❄Q2

Determine n that ensures the composite Simpson's rule approximates $\int x log x \, dx 21$ with an absolute error of at most 10-6. Numerically verify if the value for n is reasonable.

```
import numpy as np

# Define the function to integrate
def func(x):
    return x * np.log(x)

# Integration bounds
a, b = 1, 2

# Exact integral of x * log(x) from 1 to 2
exact_integral = 2 * np.log(2) - 0.75

# Number of subintervals
N = 12  # Start with the minimum even number above our calculated
minimum

# Compute numerical integral using Simpson's rule
numerical_integral = simpson(func, a, b, N)

# Compute absolute error
error = np.abs(numerical_integral - exact_integral)
```

```
print(f"Numerical integral: {numerical_integral}")
print(f"Exact integral: {exact_integral}")
print(f"Absolute error: {error}")
```

```
Numerical integral: 0.6362945608313058
Exact integral: 0.6362943611198906
Absolute error: 1.9971141518304592e-07
```

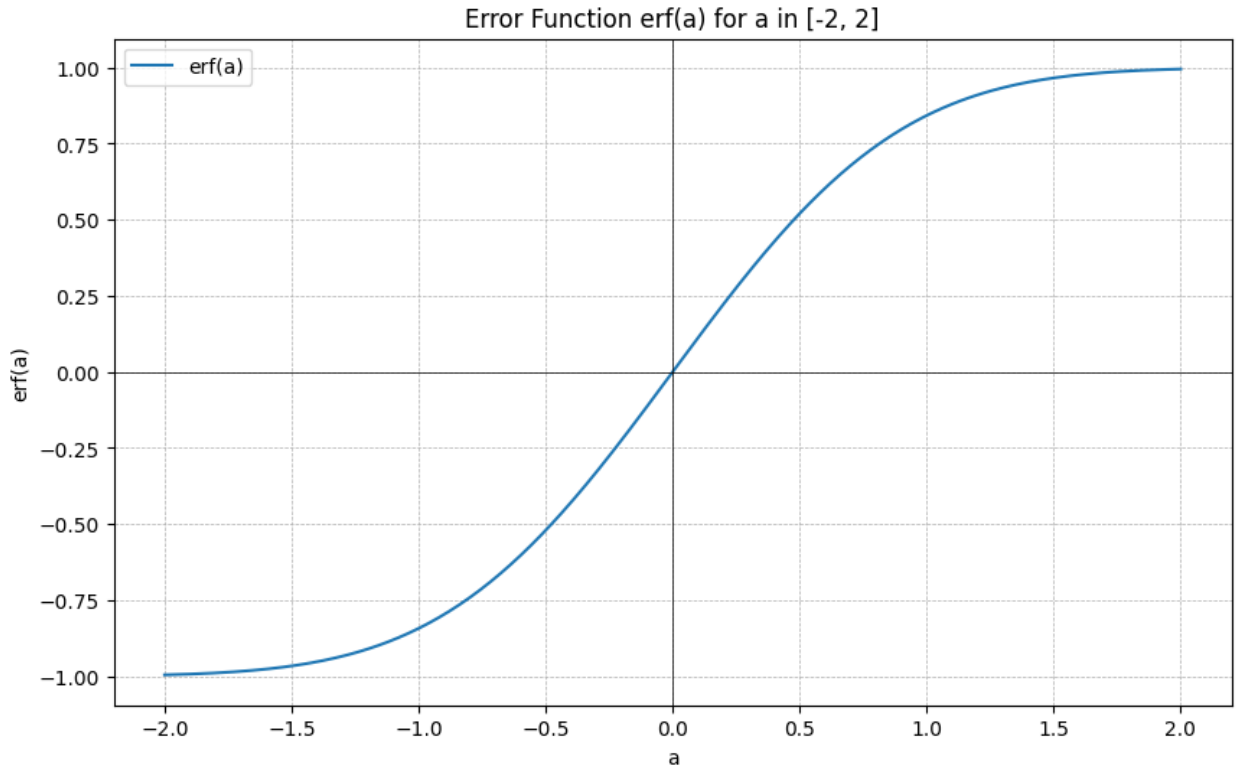
❄Q3

```
# Define the integrand function
def integrand(x):
    return np.exp(-x**2)

# Define the upper limit of the integral
a = 1.5

# Compute the integral using scipy.integrate.quad
result, error = quad(integrand, 0, a)

# Compute the error function value
erf_value = (2 / np.sqrt(np.pi)) * result

print(f"The value of the integral from 0 to {a} is approximately:
{result}")
print(f"The value of the error function erf({a}) is approximately:
{erf_value}")
```

```
The value of the integral from 0 to 1.5 is approximately:
0.8561883936249011
The value of the error function erf(1.5) is approximately:
0.9661051464753108
```

```
from scipy.special import erf

# Define the range of a
a_values = np.linspace(-2, 2, 500)

# Compute the error function values for the range of a
erf_values = erf(a_values)

# Plotting the error function
plt.figure(figsize=(10, 6))
plt.plot(a_values, erf_values, label='erf(a)')
plt.xlabel('a')
plt.ylabel('erf(a)')
plt.title('Error Function erf(a) for a in [-2, 2]')
plt.axhline(0, color='black',linewidth=0.5)
```

```
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```




```
from scipy.special import roots_legendre
from scipy.integrate import quad

# Define the integrand function
def integrand(x):
    return np.exp(-x**2)

# Define the upper limit of the integral
a = 1.5

# Compute the exact integral using scipy.integrate.quad
exact_integral, _ = quad(integrand, 0, a)

# Function to perform Gauss-Legendre quadrature
def gauss_legendre_quadrature(func, a, b, n):
    nodes, weights = roots_legendre(n)
    # Change of variables to fit the interval [a, b]
    t = 0.5 * (nodes + 1) * (b - a) + a
    return 0.5 * (b - a) * np.sum(weights * func(t))
```

```python
# Number of points for Gauss-Legendre quadrature
points = [2, 3, 4, 5]

# Compute integrals and relative errors
for n in points:
    gauss_integral = gauss_legendre_quadrature(integrand, 0, a, n)
    relative_error = np.abs(gauss_integral - exact_integral) /
exact_integral
    print(f"Number of points: {n}")
    print(f"Gauss-Legendre Integral: {gauss_integral}")
    print(f"Relative Error: {relative_error}\n")
```

```
Number of points: 2
Gauss-Legendre Integral: 0.8633384521923847
Relative Error: 0.008351034212472744

Number of points: 3
Gauss-Legendre Integral: 0.8556538986506874
Relative Error: 0.0006242726229337497

Number of points: 4
Gauss-Legendre Integral: 0.8562100942814074
Relative Error: 2.534565601202751e-05

Number of points: 5
Gauss-Legendre Integral: 0.8561877895596879
Relative Error: 7.055283833338084e-07
```

```python
# Define the integrand function
def integrand(x):
    return np.exp(-x**2)

# Define the upper limit of the integral
a = 1.5

# Compute the exact integral using scipy.integrate.quad
exact_integral, _ = quad(integrand, 0, a)

# Gauss-Legendre quadrature for 3 points
def gauss_legendre_quadrature(func, a, b, n):
    nodes, weights = roots_legendre(n)
    # Change of variables to fit the interval [a, b]
    t = 0.5 * (nodes + 1) * (b - a) + a
    return 0.5 * (b - a) * np.sum(weights * func(t))

# Number of points for Gauss-Legendre quadrature
n = 3
gauss_integral_3 = gauss_legendre_quadrature(integrand, 0, a, n)
```

```
error_3_point = np.abs(gauss_integral_3 - exact_integral)

print(f"3-point Gauss-Legendre Integral: {gauss_integral_3}")
print(f"Error: {error_3_point}")
```

```
3-point Gauss-Legendre Integral: 0.8556538986506874
Error: 0.0005344949742136507
```

```
from scipy.optimize import minimize_scalar

# Define the second derivative of the integrand
def second_derivative(x):
    return (4 * x**2 - 2) * np.exp(-x**2)

# Find the maximum absolute value of the second derivative on the
interval [0, 1.5]
result = minimize_scalar(lambda x: -np.abs(second_derivative(x)),
bounds=(0, 1.5), method='bounded')
max_f2 = -result.fun

print(f"Maximum |f''(x)| on [0, 1.5]: {max_f2}")
```

```
Maximum |f''(x)| on [0, 1.5]: 0.8925206405935732
```

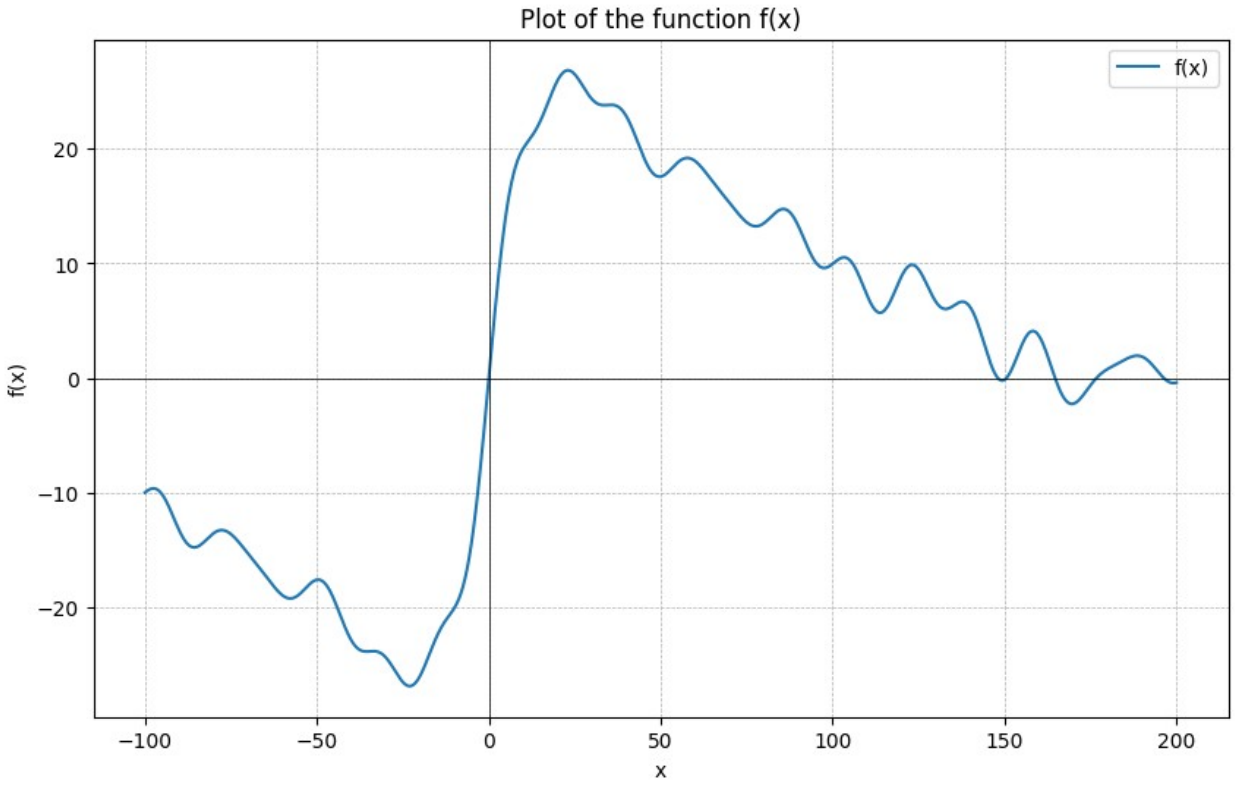
```
# Calculate the number of trapezoids required
b = 1.5
required_error = error_3_point
N = int(np.ceil(np.sqrt(((b - 0)**3 * max_f2) / (12 *
required_error))))

print(f"Number of trapezoids required: {N}")

# Trapezoidal rule implementation
def trapezoidal(func, a, b, N):
    x = np.linspace(a, b, N + 1)
    h = (b - a) / N
    integral = 0.5 * (func(x[0]) + func(x[-1])) + np.sum(func(x[1:-
1]))
    return integral * h

# Evaluate the integral using the trapezoidal rule with the obtained
number of trapezoids
trapezoidal_integral = trapezoidal(integrand, 0, b, N)
relative_error_trapezoidal = np.abs(trapezoidal_integral -
exact_integral) / exact_integral

print(f"Trapezoidal Integral: {trapezoidal_integral}")
print(f"Relative Error: {relative_error_trapezoidal}")
```

```
Number of trapezoids required: 22
Trapezoidal Integral: 0.8560659282046451
Relative Error: 0.000143035599603806
```

Numerical Differentiation

❄Q1

```python
import numpy as np
from math import sin
# Define the function
def f(x):
    n = 27
    t = 0
    for i in range(50):
        t += sin(10 * x / n)
        n = (n // 2) if n % 2 == 0 else 3 * n + 1
    return t

# Generate x values from -100 to 200
x_values = np.linspace(-100, 200, 1000)

# Compute the corresponding y values
y_values = [f(x) for x in x_values]

# Plotting the function
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of the function f(x)')
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```

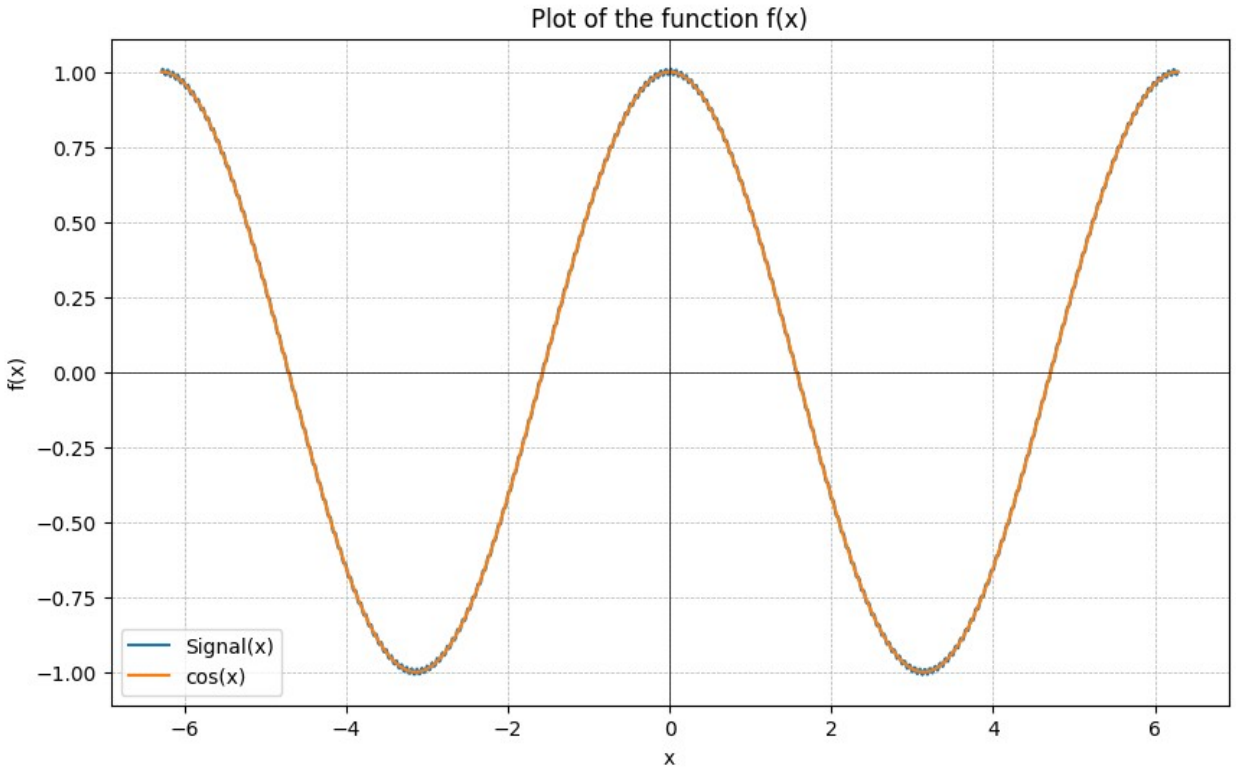



```
def forward_diff(f,x,h):
    return (f(x+h)-f(x))/h

x = 2
h = 1e-8
print(f"Estimated deri. f`(2): {forward_diff(f,x,h)}")

Estimated deri. f`(2): 3.079641430758784
```

❄Q2

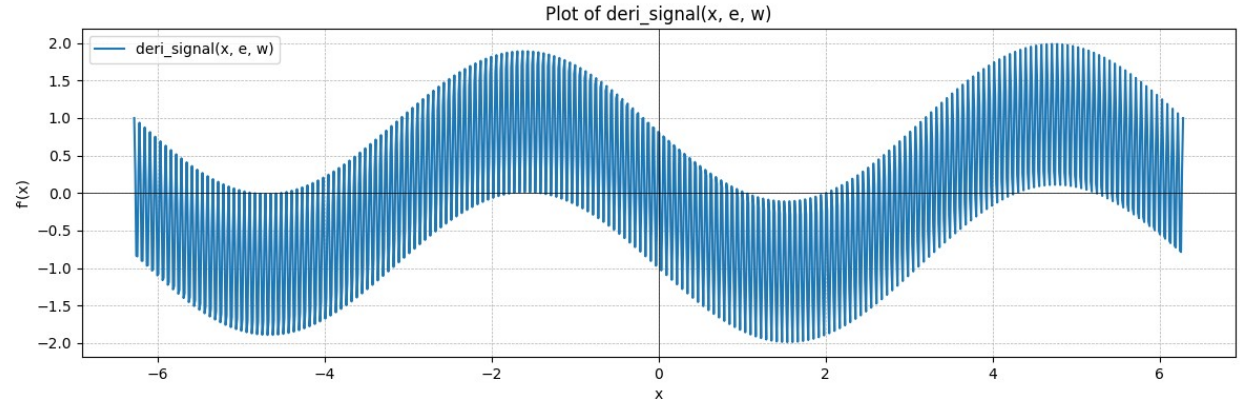
```
def signal(x,e,w):
    return np.cos(x)+e*np.sin(w*x)

def cos(x):
    return np.cos(x)

e = 0.01
w = 100

# Generate x values from -100 to 200
x_values = np.linspace(-2*np.pi, 2*np.pi, 1000)

# Compute the corresponding y values
y_values = [signal(x,e,w) for x in x_values]
y_cos_values = [cos(x) for x in x_values]
```

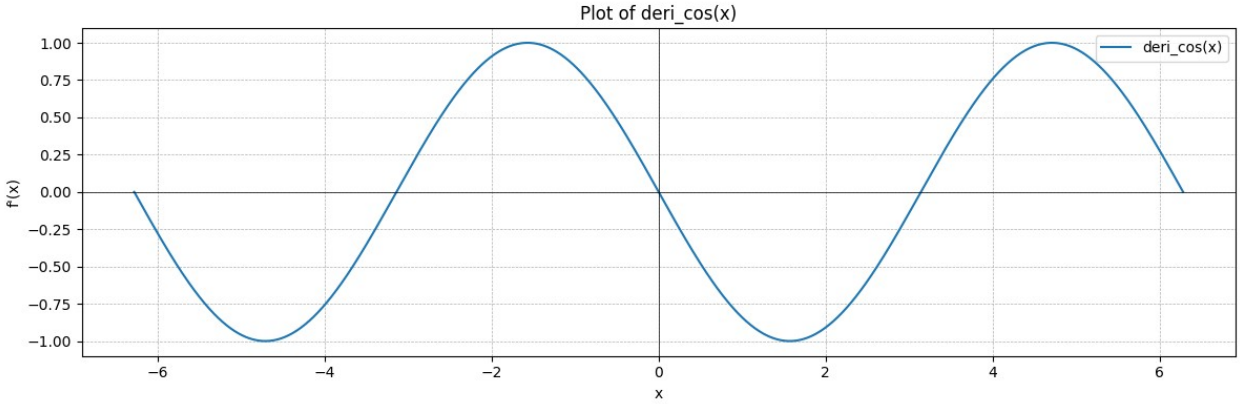
```
# Plotting the function
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, label='Signal(x)')
plt.plot(x_values,y_cos_values, label = 'cos(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of the function f(x)')
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```




```
def deri_signal(x,e,w):
    return (-1)*np.sin(x) + e*w* np.cos(w*x)
def deri_cos(x):
    return (-1)*np.sin(x)

# Parameters
e = 0.01
w = 100

# Generate x values from -2*pi to 2*pi
x_values = np.linspace(-2*np.pi, 2*np.pi, 1000)
```

```
# Compute the corresponding y values
y_deri_signal_values = [deri_signal(x, e, w) for x in x_values]
y_deri_cos_values = [deri_cos(x) for x in x_values]

# Create subplots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

# Plot deri_signal(x, e, w)
ax1.plot(x_values, y_deri_signal_values, label="deri_signal(x, e, w)")
ax1.set_xlabel('x')
ax1.set_ylabel("f'(x)")
ax1.set_title("Plot of deri_signal(x, e, w)")
ax1.axhline(0, color='black', linewidth=0.5)
ax1.axvline(0, color='black', linewidth=0.5)
ax1.grid(True, which='both', linestyle='--', linewidth=0.5)
ax1.legend()

# Plot deri_cos(x)
ax2.plot(x_values, y_deri_cos_values, label="deri_cos(x)")
ax2.set_xlabel('x')
ax2.set_ylabel("f'(x)")
ax2.set_title("Plot of deri_cos(x)")
ax2.axhline(0, color='black', linewidth=0.5)
ax2.axvline(0, color='black', linewidth=0.5)
ax2.grid(True, which='both', linestyle='--', linewidth=0.5)
ax2.legend()

# Adjust layout and show plot
plt.tight_layout()
plt.show()
```





❄Q4

```
# Define the values of f(x) given in the table
x_values = [0.1, 0.2, 0.3, 0.4]
f_values = [0.09983, 0.19867, 0.29552, 0.38942]

# Step size h
h = 0.1


# Forward difference formula for f'(0.1)
def forward_difference(f_values, h,i):
    f_prime = (-f_values[i+3] + 9*f_values[i+1] - 8*f_values[i]) / (6
* h)
    return f_prime

# Backward difference formula for f'(0.3)
def backward_difference( f_values, h,i):
    f_prime = (3*f_values[i] - 4*f_values[i-1] + 3*f_values[i-2]) / (2
* h)
    return f_prime

# Calculate derivatives
f_prime_0_1 = forward_difference(f_values, h,0)
f_prime_0_3 = backward_difference(f_values, h,2)
```

```python
# Output the results
print(f"f'(0.1) ≈ {f_prime_0_1:.5f}")
print(f"f'(0.3) ≈ {f_prime_0_3:.5f}")
```

```
f'(0.1) ≈ 0.99995
f'(0.3) ≈ 1.95685
```