EM215 Numerical Methods
Numerical Integration and Differentiation
Lab1/Assignment 1

Malintha K.M.K. (E20243)

Numerical Quadrature

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 sub intervals.

Composite Trapezoidal Rule

The composite trapezoidal rule approximates the integral of a function $f(x)$ over an interval $[a, b]$ by dividing into the n number of intervals.

$$I_t = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] \text{ ; where } h = \frac{b-a}{n}$$

Error E_t of the composite rule is

$$E_t = -\frac{b-a}{12} \times h^2 \times f``(\xi) = -\frac{(b-a)^3}{12n^2} \times f``(\xi)$$

Therefore the absolute error of the composite trapezoidal rule decays at $\frac{1}{n^2}$ rate.

Composite Simpson's Rule

This uses quadratic polynomials to approximate the functions over the sub intervals.

$$I_s = \frac{h}{3}\left[f(a) + 4\sum_{i=1,odd}^{n-1} f(x_i) + 2\sum_{i=2,even}^{n-2} f(x_i) + b\right]$$

Error E_t of the composite Simpson's rule is

$$E_t = -\frac{b-a}{180} \times h^4 \times f^4(\xi) = -\frac{(b-a)^5}{180n^4} \times f^4(\xi)$$

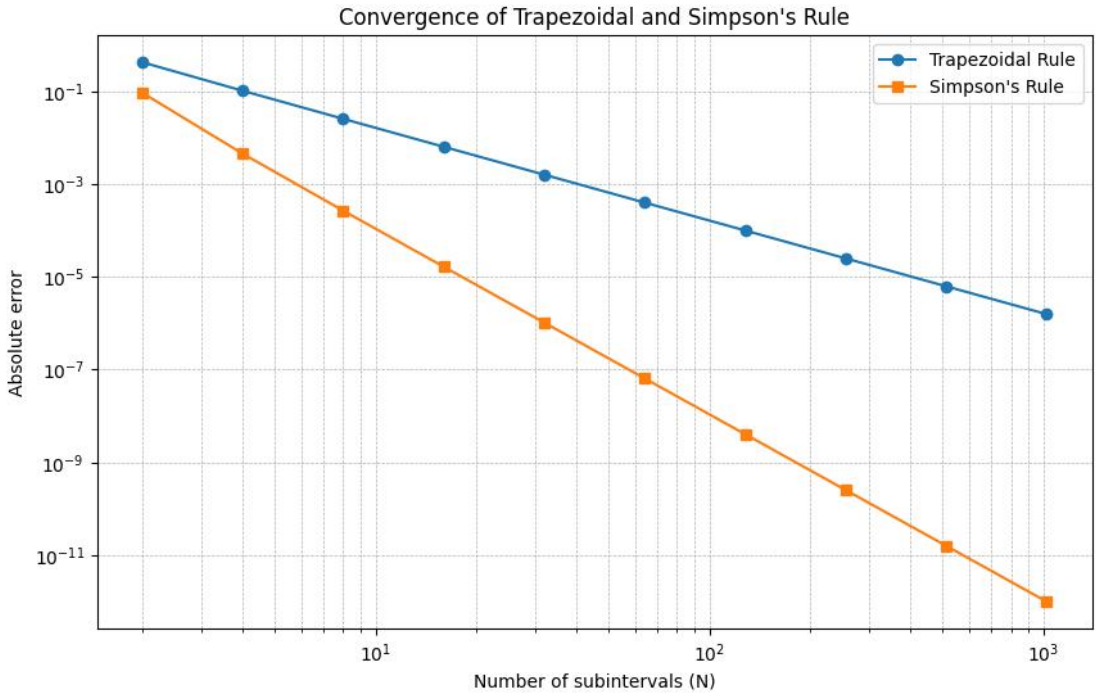
This shows that as the number of sub intervals n increases, the absolute error decreases proportionally to $\frac{1}{n^4}$

Following graph shows a comparison between the convergences of trapezoidal and Simpson's rules.

Code:

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

# 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

# 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

def calculate_errors(func, a, b, N_values, exact_integral):
    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))
    return trapezoidal_errors, simpson_errors

def plot_errors(N_values, trapezoidal_errors, simpson_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()

# Calculate errors
trapezoidal_errors, simpson_errors = calculate_errors(func, a, b, N_values, exact_integral)

# Plot errors
plot_errors(N_values, trapezoidal_errors, simpson_errors)
```



Q2. Determine n that ensures the composite Simpson's rule approximates

$$\int_1^2 x\log x\, dx$$

with an absolute error of at most 10^{-6}. Numerically verify if the value for n is reasonable.

Considering the error estimation of the composite Simpson's rule,

$E_s = -\frac{(b-a)}{180}h^4 f^4(\xi)$ and $|E_s| < 10^{-6}$

By differentiating $f(x)$, max $f^4(\xi) = \frac{2}{x^3} = \frac{2}{1} = 2$

Therefore, E_s max is, $|E_s| = \frac{h^4}{90}$

$$h < (9 \times 10^{-5})^{\frac{1}{4}}$$

$$n > \frac{1}{(9\times10^{-5})^{\frac{1}{4}}}$$

So, minimum n should be 12

Code:

```
import numpy as np

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

# 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

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

Output:

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

Q3. Consider the error function that occur in many areas of engineering and statistics:

$$\text{err f(a)} = \frac{2}{\sqrt{\pi}} \int_0^a e^{-x^2} dx$$

a) Find the value of the integral for a = 1.5 with scipy.integrate.quad and use this as the exact value of the integral.

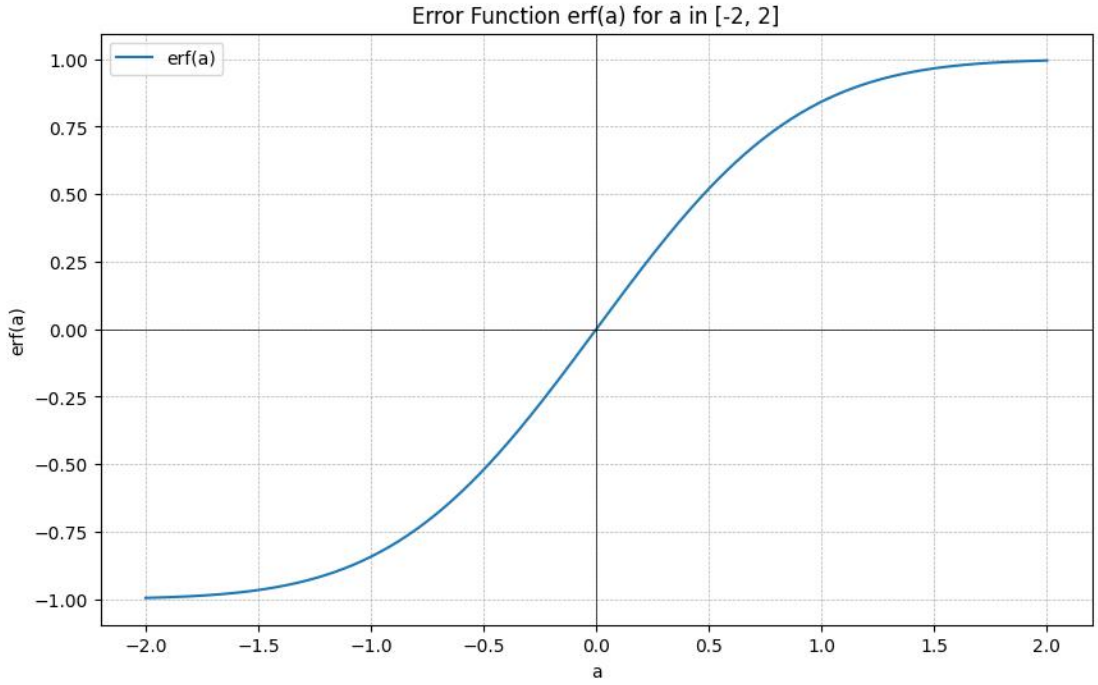
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
from scipy.special import erf

# 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}")

# 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()
```

Output:

```
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
```

b) Plot $errf(a)$ for $y \in [-2,2]$.



c) Evaluate the integral using the Gauss integration scheme with 2, 3, 4 and 5 integration points. Calculate the relative error in each case.

Code:

```
import numpy as np
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))

# 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")
```

Output:

```
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
```

d) Calculate the number of trapeziums required by the trapezoidal rule to produce an error similar to that produced by the 3-point Gauss integration scheme, then evaluate the integral using the trapezoidal rule with the obtained number of trapeziums.

Code:

```
import numpy as np
from scipy.special import roots_legendre
from scipy.integrate import quad
from scipy.optimize import minimize_scalar

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

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

# Compute the exact integral using scipy.integrate.quad
a = 1.5
exact_integral, _ = quad(integrand, 0, a)

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

# Gauss-Legendre quadrature for 3 points
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}")
```

```python
# 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}")

# Calculate the number of trapezoids required
required_error = error_3_point
b = 1.5
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}")
```

Output:

```
3-point Gauss-Legendre Integral: 0.8556538986506874
Error: 0.0005344949742136507
Maximum |f''(x)| on [0, 1.5]: 0.8925206405935732
Number of trapezoids required: 22
Trapezoidal Integral: 0.8560659282046451
Relative Error: 0.000143035599603806
```

Numerical Differentiation

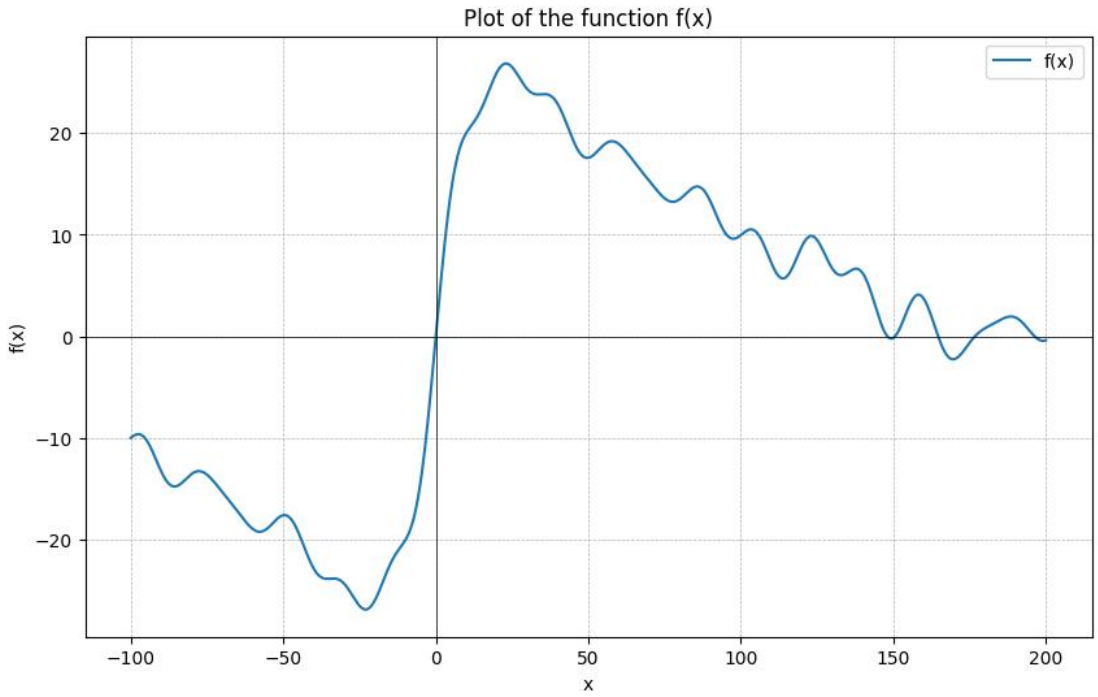
Q1. Use the forward difference formula with $h = 10^{-8}$ to estimate $f'(2)$

Code:

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

# Define the forward difference method
def forward_diff(f, x, h):
    return (f(x + h) - f(x)) / h

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

Output:

```
Estimated derivative f'(2): 3.079641430758784
```

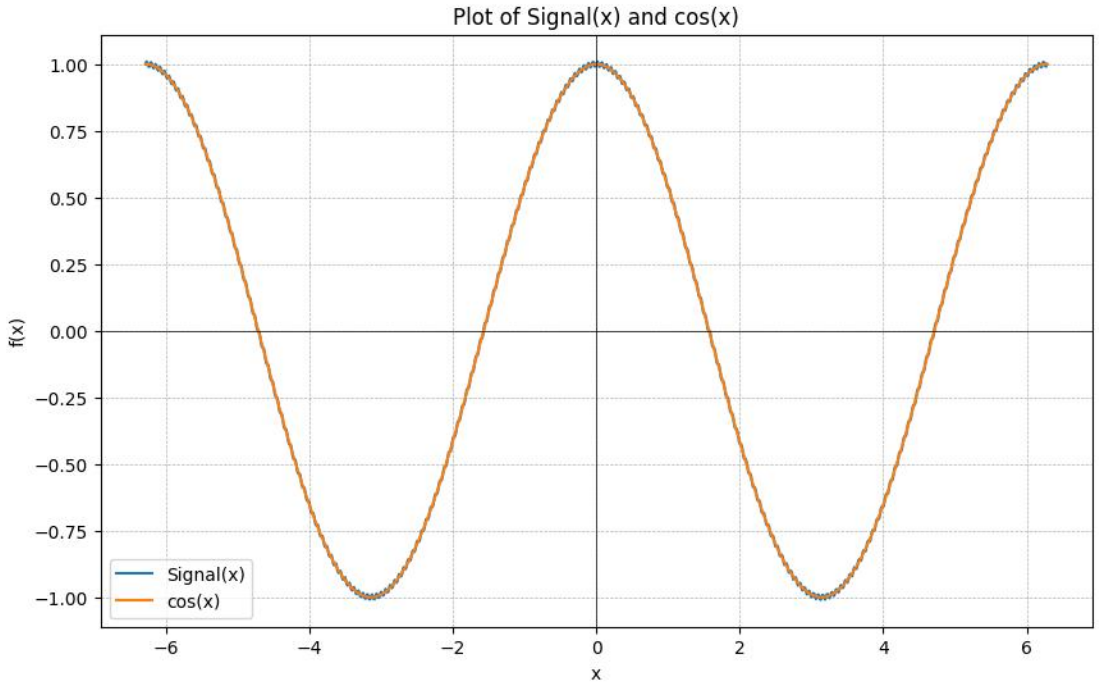


Q2. Suppose You have obtained a signal f=cos(x). But this signal is spoiled by some noise given by a small sine wave:
$f_{\epsilon,\omega}(x)=\cos(x)+\epsilon\sin(\omega x)$. where $0<\epsilon\ll 1$ is a very small number and ω is a large number.
For $\epsilon=0.01$ and $\omega=100$, first plot in the same graph the function f=cos(x) and the function with noise $f_{\epsilon,\omega}(x)=\cos(x)+\epsilon\sin(\omega x)$.

Code:

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

# Define the signal function
def signal(x, e, w):
    return np.cos(x) + e * np.sin(w * x)

# Define the cosine function
def cos(x):
    return np.cos(x)

# Parameters
e = 0.01
w = 100

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

# Compute the corresponding y values
y_values = signal(x_values, e, w)
y_cos_values = cos(x_values)

# Plotting the functions
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 Signal(x) and cos(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()
```

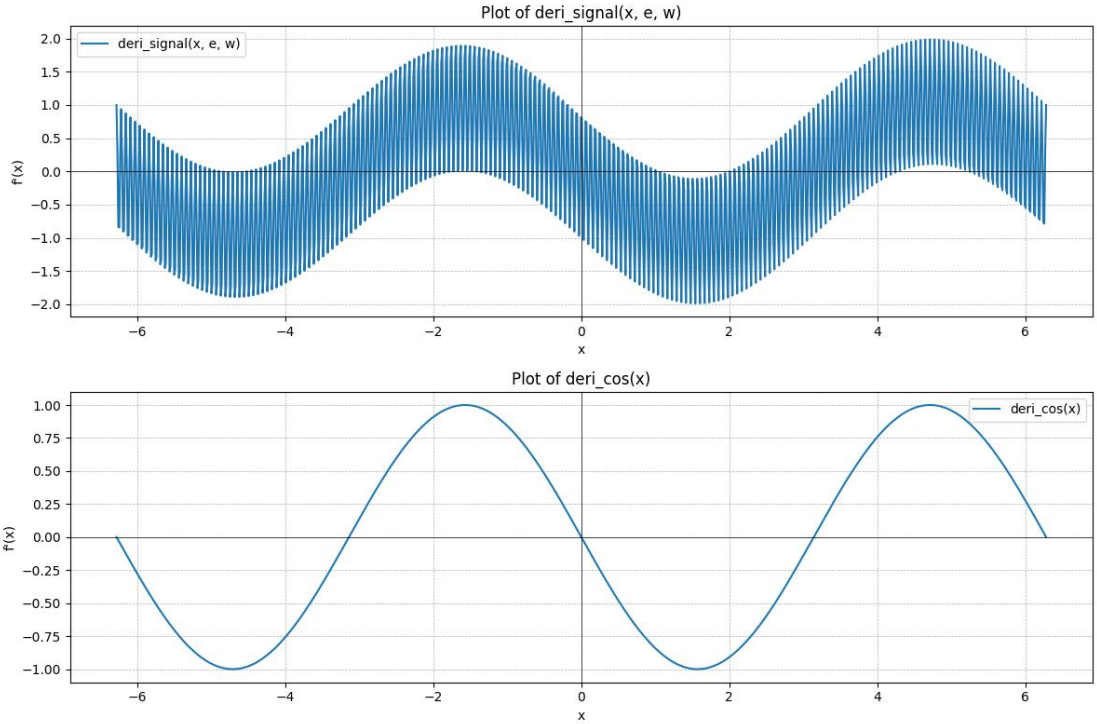


Then in a different graph plot the derivatives f' and f'ϵ,ω and comment on your observations.

Code:

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

# Define the derivative of signal function
def deri_signal(x, e, w):
    return (-1) * np.sin(x) + e * w * np.cos(w * x)

# Define the derivative of cosine function
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_values, e, w)
y_deri_cos_values = deri_cos(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()
```



The derivative of the signal function exhibits rapid oscillations, suggesting a highly variable rate of change that fluctuates significantly over short intervals. This variability indicates complex and unpredictable behavior, with frequent changes in slope direction. In contrast, the derivative of the cosine function shows smooth oscillations, reflecting the steady and predictable nature of its sine wave pattern. The smooth changes in slope highlight the regular and periodic behavior of the cosine function. Overall, these differences underscore the contrast between the variable and potentially complex nature of the signal function and the steady, periodic nature of the cosine function's derivative.

Q3. Suppose a formula for the first derivative using finite difference formula is given as:

$$f'(x_i) \approx \frac{-f(x_{i+3}) + 9f(x_{i+1}) - 8f(x_i)}{6h}$$

Where the points x_i, x_{i+1}, x_{i+2}, x_{i+3} are all equally spaced with step size h. Determine the order of the approximation.

$$f(x_i) = f_i$$
$$f(x_i + h) = f_i + \frac{hf_i}{1!} + \frac{h^2 f''_i}{2!} + \frac{h^3 f'''_i}{3!} + O(h^4)$$
$$f(x_i + 2h) = f_i + \frac{2hf_i}{1!} + \frac{4h^2 f''_i}{2!} + \frac{8h^3 f'''_i}{3!} + O(h^4)$$
$$f(x_i + 3h) = f_i + \frac{3hf_i}{1!} + \frac{9h^2 f''_i}{2!} + \frac{27h^3 f'''_i}{3!} + O(h^4)$$

Using above expansion

$$-f(x_{i+3}) + 9f(x_{i+1}) - 8f(x_i) = 6hf'_i - 3h^3 f'''_i + O(h^4)$$
$$\frac{-f(x_{i+3}) + 9f(x_{i+1}) - 8f(x_i)}{6h} = f'_i - \frac{h^2 f'''_i}{2} + O(h^3)$$

The major term of the truncation error is $\frac{h^2 f'''_i}{2}$. So this approximation is second order.

Q4. The following table gives the values of f(x) = sin x: Estimate f'(0.1); f'(0.3) using an appropriate three-point formula.

X	f(x)
0.1	0.09983
0.2	0.19867
0.3	0.29552
0.4	0.38942

Just to find the derivative at x = 0.1 , forward difference formula is used while at x = 0.3 backward difference formula is used.

Code:

```
# 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)

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

Output:

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