Using PYTHON show how the following is achieved(PRACTICAL) i. Differentiation

```
# Differentiation code

import numpy as np

# Define a function
def f(x):
  return x**2 + 2*x - 1

# Calculate the derivative using finite differences
h = 1e-5  # Step size
x = 2  # Point at which to calculate the derivative
derivative = (f(x + h) - f(x)) / h

print("Approximate derivative at x = 2:", derivative)

# For higher-order derivatives or more complex functions, you might consider using the `scipy.misc.derivative` function.
```

```
Approximate derivative at x = 2: 6.000009999951316
```

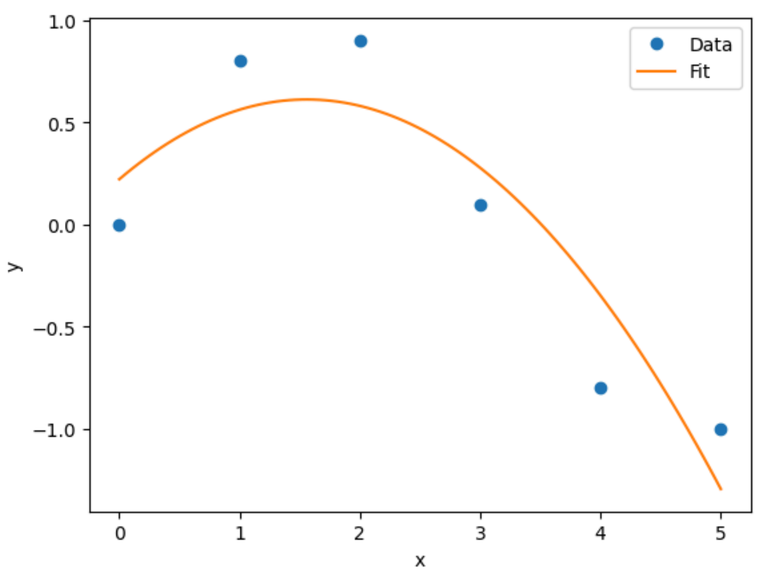
ii. Numerical integration

```
# python Numerical integration

import numpy as np
from scipy.integrate import quad

# Define the function to integrate
def f(x):
  return x**2 + 2*x - 1

# Integrate the function from 0 to 2
result, error = quad(f, 0, 2)

print("Result of integration:", result)
print("Estimated error:", error)
```

```
Result of integration: 4.666666666666666
Estimated error: 5.666271351443603e-14
```

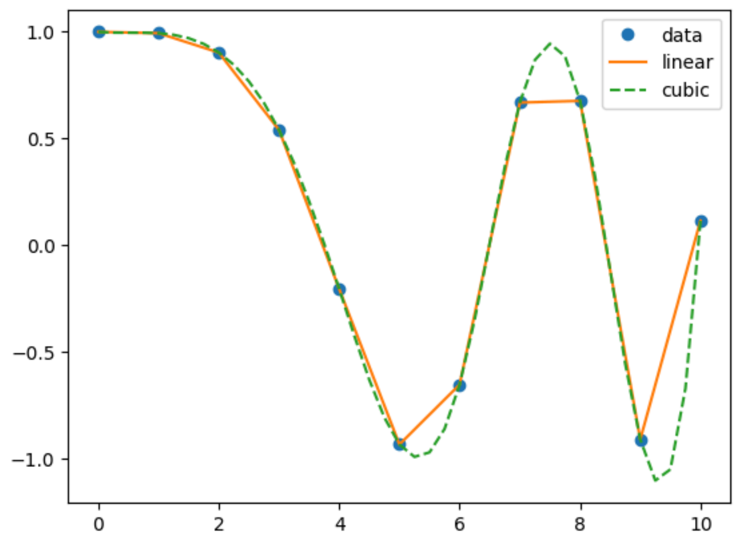
iii. Curve fitting

```
# Curve Fitting

import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Sample data
x_data = np.array([0, 1, 2, 3, 4, 5])
y_data = np.array([0, 0.8, 0.9, 0.1, -0.8, -1])

# Define the function to fit (example: a quadratic function)
def func(x, a, b, c):
  return a * x**2 + b * x + c

# Perform curve fitting
popt, pcov = curve_fit(func, x_data, y_data)

# Extract the fitted parameters
a_fit, b_fit, c_fit = popt

# Generate points for the fitted curve
x_fit = np.linspace(x_data.min(), x_data.max(), 100)
y_fit = func(x_fit, a_fit, b_fit, c_fit)

# Plot the data and the fitted curve
plt.plot(x_data, y_data, 'o', label='Data')
plt.plot(x_fit, y_fit, '-', label='Fit')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



iv. Spline interpolation

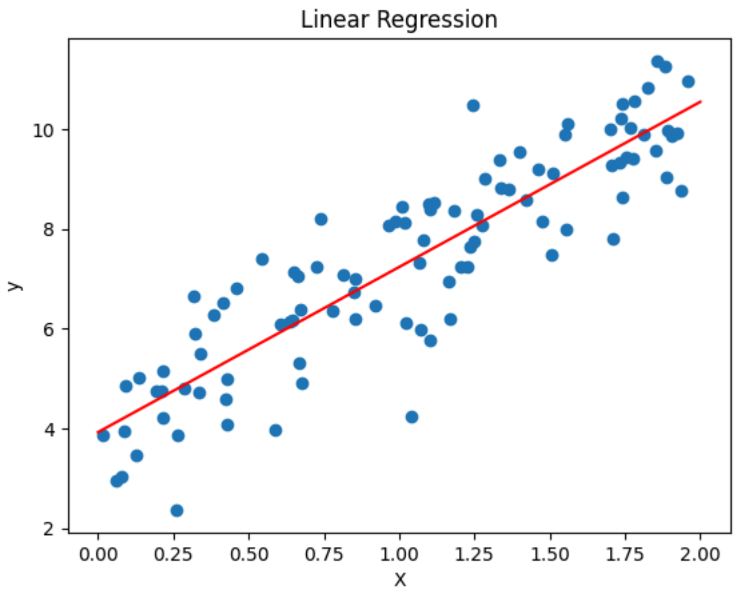
```
# Spline Interpolation

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d

# Sample data
x = np.linspace(0, 10, num=11, endpoint=True)
y = np.cos(-x**2/9.0)

# Create a linear interpolation function
f = interp1d(x, y)
f2 = interp1d(x, y, kind='cubic')

# Generate new x values for interpolation
xnew = np.linspace(0, 10, num=41, endpoint=True)

# Plot the original data and the interpolated curves
plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
plt.legend(['data', 'linear', 'cubic'], loc='best')
plt.show()
```



v. linear regression

```
# prompt: Linear Regression

import matplotlib.pyplot as plt
import numpy as np
# Generate some random data
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add a bias term to X
X_b = np.c_[np.ones((100, 1)), X]

# Calculate the optimal weights using the normal equation
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# Make predictions
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)

# Print the results
print("Optimal weights:", theta_best)
print("Predictions for X_new:", y_predict)

# Plot the data and the linear regression line
plt.scatter(X, y)
plt.plot(X_new, y_predict, "r-")
plt.xlabel("X")
plt.ylabel("y")
plt.title("Linear Regression")
plt.show()
```

```
Optimal weights: [[3.92538965]
 [3.30986098]]
Predictions for X_new: [[ 3.92538965]
 [10.54511161]]
```



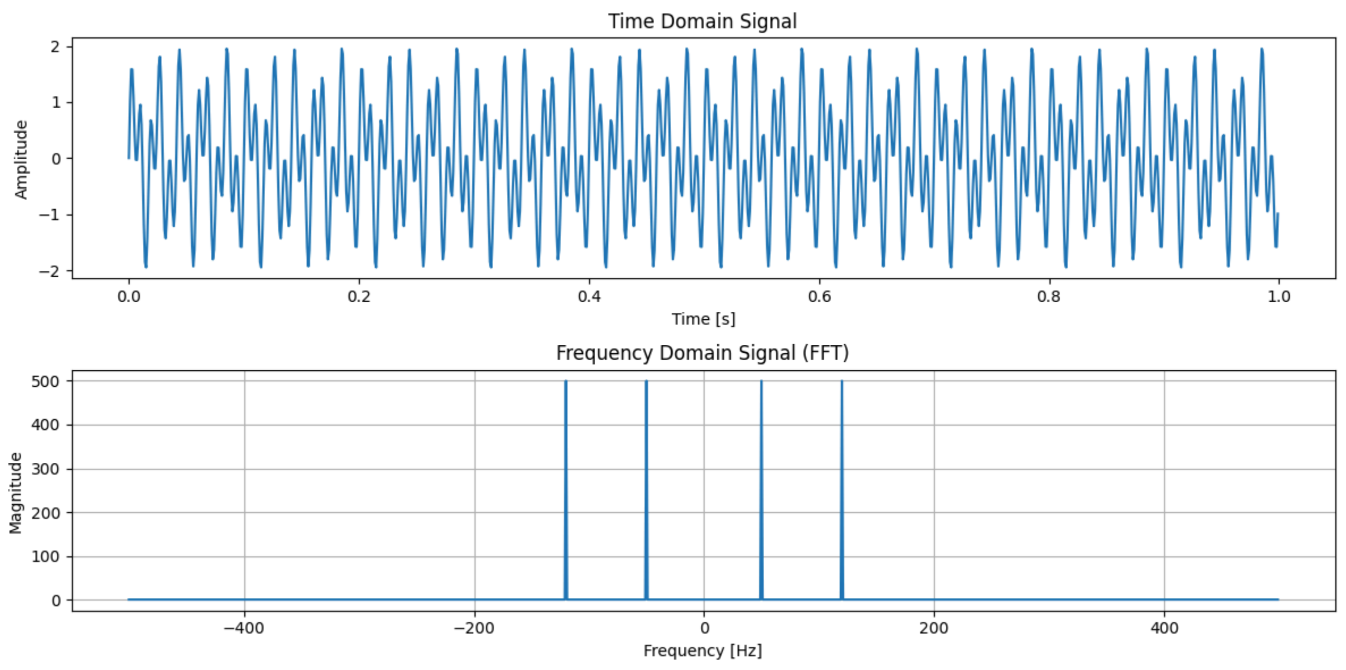
) A smart robotic agent with a laser Scanner is doing a quick quality check on holes drilled in a rectangular plate. The centers of the hole in the plate describes the path the arm needs to take, and the hole centers are located on a Cartesian coordinate system as shown X (in) 2.00 4.25 5.25 7.81 9.20 10.60 Y(in) 7.2 7.1 6.0 5.0 3.5 5.0 If the laser scanner is traversing from x=2.00 to x =4.25 in a linear path, what is the value of y at x= 4.0 using the linear spline formula , show how this problem can be solved using PYTHON (PRACTICAL)

```
# Given points question c
x1, y1 = 2.00, 7.2
x2, y2 = 4.25, 7.1
x = 4.0

# Linear interpolation formula
y = y1 + (y2 - y1) / (x2 - x1) * (x - x1)

print(f"The value of y at x = {x} is {y}")
```

```
The value of y at x = 4.0 is 7.111111111111111
```

Fast fourier Transform (FFT)

```
#Fast Fourier Transform (FFT) question E
import numpy as np
import matplotlib.pyplot as plt

def compute_fft():
    # Parameters
    f1 = 50    # Frequency of the first sine wave
    f2 = 120   # Frequency of the second sine wave
    fs = 1000  # Sampling frequency
    t = np.linspace(0, 1, fs, endpoint=False)  # Time vector for 1 second

    # Signal
    s = np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)

    # Compute FFT
    fft_s = np.fft.fft(s)
    fft_s = np.fft.fftshift(fft_s)  # Shift zero frequency component to the center
    freq = np.fft.fftfreq(len(s), d=1/fs)
    freq = np.fft.fftshift(freq)  # Shift zero frequency component to the center

    # Plot the signal
    plt.figure(figsize=(12, 6))

    plt.subplot(2, 1, 1)
    plt.plot(t, s)
    plt.title('Time Domain Signal')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')

    # Plot the FFT
    plt.subplot(2, 1, 2)
    plt.plot(freq, np.abs(fft_s))
    plt.title('Frequency Domain Signal (FFT)')
    plt.xlabel('Frequency [Hz]')
    plt.ylabel('Magnitude')
    plt.grid()

    plt.tight_layout()
    plt.show()

# Run the function
compute_fft()
```



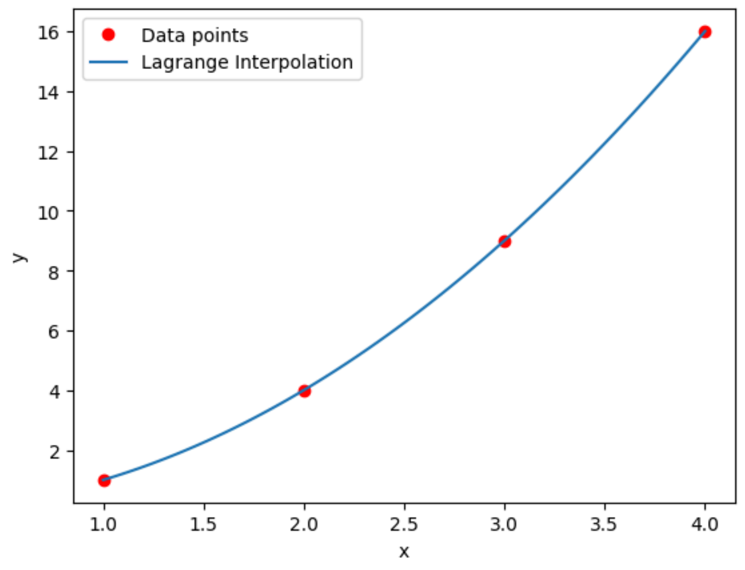
Write a program to show how the trapezoidal rule of integration works in PYTHON (PRACTICAL

```
# Trapezoidal Method

# Define function to integrate
def f(x):
    return 1/(1 + x**2)

# Implementing trapezoidal method
def trapezoidal(x0,xn,n):
    # calculating step size
    h = (xn - x0) / n

    # Finding sum
    integration = f(x0) + f(xn)

    for i in range(1,n):
        k = x0 + i*h
        integration = integration + 2 * f(k)

    # Finding final integration value
    integration = integration * h/2

    return integration

# Input section
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 sub intervals: "))

# Call trapezoidal() method and get result
result = trapezoidal(lower_limit, upper_limit, sub_interval)
print("Integration result by Trapezoidal method is: %0.6f" % (result) )
```

```
Enter lower limit of integration: 0
Enter upper limit of integration: 1
Enter number of sub intervals: 6
Integration result by Trapezoidal method is: 0.784241
```

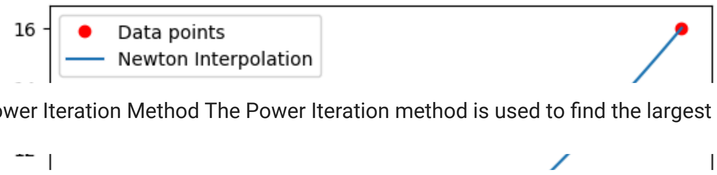
question i a) Lagrange Polynomial Interpolation

```
#large range question I
import numpy as np

def lagrange_interpolation(x_points, y_points, x):
    n = len(x_points)
    P_x = 0
    for i in range(n):
        L_i = 1
        for j in range(n):
            if j != i:
                L_i *= (x - x_points[j]) / (x_points[i] - x_points[j])
        P_x += y_points[i] * L_i
    return P_x

x_points = [1, 2, 3, 4]
y_points = [1, 4, 9, 16]

x_values = np.linspace(1, 4, 100)
lagrange_values = [lagrange_interpolation(x_points, y_points, x) for x in x_values]

import matplotlib.pyplot as plt

plt.plot(x_points, y_points, 'ro', label='Data points')
plt.plot(x_values, lagrange_values, label='Lagrange Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



B. Newton's Divided Difference Method

```
 #Newton's Divided Difference Method
 def divided_differences(x_points, y_points):
    n = len(y_points)
    coef = np.zeros([n, n])
    coef[:,0] = y_points

    for j in range(1, n):
        for i in range(n - j):
            coef[i][j] = (coef[i + 1][j - 1] - coef[i][j - 1]) / (x_points[i + j] - x_points[i])

    return coef[0, :]  # Return the first row (coefficients)

def newton_interpolation(coef, x_points, x):
    n = len(coef)
    P_x = coef[0]
    for i in range(1, n):
        term = coef[i]
        for j in range(i):
            term *= (x - x_points[j])
        P_x += term
    return P_x

# Example usage:
coef = divided_differences(x_points, y_points)

newton_values = [newton_interpolation(coef, x_points, x) for x in x_values]

plt.plot(x_points, y_points, 'ro', label='Data points')
plt.plot(x_values, newton_values, label='Newton Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



1. Power Iteration Method The Power Iteration method is used to find the largest eigenvalue and its corresponding eigenvector of a matrix.

```
#Eigenvalues and Eigenvector  question J
#power iteration
import numpy as np

def power_iteration(A, num_simulations):
    n, d = A.shape
    b_k = np.random.rand(n)
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.