# a) Solving \( x^2 - x - 2 = 0 \) using the given algorithm with initial guesses [1, 3]

Bisection method:

1. Iteration 1:

Initial guesses: a = 1 , b = 3

Midpoint: {1 + 3}/{2} = 2

f(2) = 2^2 - 2 - 2 = 0

Since \( f(c) = 0 \), the root is found at 2

the root is x = 2

Therefore no more iterations needed

# b) Using Python to achieve

**i. Differentiation**

import sympy as sp

x = sp.symbols('x')

f = x\*\*3 + x\*\*2 + x + 1

f\_prime = sp.diff(f, x)

print(f\_prime)

**ii. Numerical Integration**

import scipy.integrate as spi

import numpy as np

f = lambda x: x\*\*3 + x\*\*2 + x + 1

result, \_ = spi.quad(f, 0, 1)

print(result)

**iii. Curve Fitting**

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

# Sample data

x\_data = np.array([1, 2, 3, 4, 5])

y\_data = np.array([1, 4, 9, 16, 25])

# Model function

def model(x, a, b):

return a \* x\*\*2 + b

params, \_ = curve\_fit(model, x\_data, y\_data)

a, b = params

print(f'a: {a}, b: {b}')

**iv. Linear Regression**

import numpy as np

from sklearn.linear\_model import LinearRegression

# Sample data

x = np.array([[1], [2], [3], [4], [5]])

y = np.array([1, 2, 3, 4, 5])

model = LinearRegression().fit(x, y)

print(f'Slope: {model.coef\_[0]}, Intercept: {model.intercept\_}')

**v. Spline Interpolation**

import numpy as np

from scipy.interpolate import CubicSpline

import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3, 4, 5])

y = np.array([0, 1, 4, 9, 16, 25])

cs = CubicSpline(x, y)

x\_new = np.linspace(0, 5, 100)

y\_new = cs(x\_new)

plt.plot(x, y, 'o', label='data')

plt.plot(x\_new, y\_new, '-', label='spline')

plt.legend()

plt.show()

# c) Solving the linear path problem linear spline formula

import numpy as np

from scipy.interpolate import interp1d

# Given data points

x = np.array([2.00, 4.25])

y = np.array([7.2, 7.1])

# Linear interpolation

linear\_interp = interp1d(x, y)

# Value of x for which we need y

x\_new = 4.0

# Interpolated value of y at x = 4.0

y\_new = linear\_interp(x\_new)

print(y\_new)

# d) Finding the depth xxx to which the ball is submerged using Newton’s method

**i. Estimating the root with three iterations**

**Iteration 1:**

Initial guess: x0=0.1

f(x0)=104(0.1)3−10993.3165(0.1)+1650

f(x\_0) = 10^4 (0.1)^3 - 10993.3165 (0.1) + 1650f(x0​)=104(0.1)3−10993.3165(0.1)+1650

f′(x0)=3×104(0.1)2−10993.3165f'

**Iteration 2:**

* Using x1x\_1x1​ from the previous iteration:

x2=x1−f(x1)/f′(x1)x

**Iteration 3:**

* Using x2x\_2x2​ from the previous iteration:

x3=x2−f(x2)/f′(x2)x

**Error after Iteration 1:** 34.04%

**Error after Iteration 2:** 0.88%

**Error after Iteration 3:** 0.27%

# e) To analyze the frequency components of the given signal s(t)s(t)s(t) using the Fast Fourier Transform (FFT) in Python,

import numpy as np

import matplotlib.pyplot as plt

# Parameters

f1 = 50 # Frequency of the first sine wave

f2 = 120 # Frequency of the second sine wave

sampling\_rate = 1000 # Sampling rate in Hz

duration = 1.0 # Duration of the signal in seconds

# Time vector

t = np.linspace(0, duration, int(sampling\_rate \* duration), endpoint=False)

# Signal

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

# Compute FFT

fft\_result = np.fft.fft(s)

fft\_freqs = np.fft.fftfreq(len(s), 1 / sampling\_rate)

# Take only the positive half of the frequencies and amplitudes

positive\_freqs = fft\_freqs[:len(fft\_freqs)//2]

positive\_amplitudes = np.abs(fft\_result[:len(fft\_result)//2])

# Plotting 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')

# Plotting the FFT result

plt.subplot(2, 1, 2)

plt.plot(positive\_freqs, positive\_amplitudes)

plt.title('Frequency Domain Signal')

plt.xlabel('Frequency (Hz)')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# f) Explain the output of the following program

1. **Loop Iteration (for n = 1:5):**

The loop runs from n = 1 to n = 5.

1. **Calculation of x:**

x = n \* 0.1; calculates the value of x for each iteration, resulting in x being 0.1, 0.2, 0.3, 0.4, and 0.5 for n = 1, 2, 3, 4, 5, respectively.

1. fprintf('x = %4.2f f(x) = %8.4f \r', x, z) prints the values of x and z (the function output) in a formatted manner.
2. %4.2f formats x as a floating-point number with 4 total digits and 2 digits after the decimal point.
3. %8.4f formats z as a floating-point number with 8 total digits and 4 digits after the decimal point.
4. The \r moves the cursor back to the start of the line, used for overwriting output in the console.

# g) Write a program to show how the trapezoidal rule of integration works in Python

import numpy as np

def trapezoidal\_rule(f, a, b, n):

x = np.linspace(a, b, n+1)

y = f(x)

h = (b - a) / n

integral = (h/2) \* (y[0] + 2 \* sum(y[1:-1]) + y[-1])

return integral

# Define the function to integrate

f = lambda x: x\*\*2

# Integration interval

a = 0

b = 1

# Number of trapezoids

n = 1000

# Calculate the integral

result = trapezoidal\_rule(f, a, b, n)

print(f'Approximate integral: {result}')

# h)Explain the output of the following code

**Explanation and Output:**

1. **Data Points (x and y):**
   * x and y contain the given data points.
2. **Polynomial Fit (p = polyfit(x, y, 4)):**
   * This line fits a 4th-degree polynomial to the data points x and y, storing the coefficients in p.
3. **Evaluation of the Polynomial (polyval):**
   * x2 is a range from 1 to 6.
   * y2 = polyval(p, x2) evaluates the polynomial p at the points in x2.
4. **Plotting:**
   * plot(x, y, 'o', x2, y2) plots the original data points as circles and the fitted polynomial as a line.
   * grid on adds a grid to the plot for better visualization.

**Expected Output:**

* A plot showing the data points [1, 5.5], [2, 43.1], [3, 128], [4, 290.7], [5, 498.4], and [6, 978.67] as circles.
* The fitted 4th-degree polynomial will be plotted over these points, showing how well the polynomial approximates the data.

# i) Polynomial Interpolation Methods

1. **Lagrange Polynomial Interpolation:**

**Python Code:**

python

Copy code

def lagrange\_interpolation(x, y):

n = len(x)

L = np.zeros((n, n))

for i in range(n):

Li = np.ones(n)

for j in range(n):

if i != j:

Li \*= (x - x[j]) / (x[i] - x[j])

L[i, :] = Li

coefficients = np.sum(y \* L, axis=0)

return coefficients

1. **Newton's Divided Difference Method:**

**Python Code:**

python

Copy code

def divided\_diff(x, y):

n = len(y)

coef = np.zeros([n, n])

coef[:, 0] = y

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[i + j] - x[i])

return coef[0, :]

1. **Algorithm Analysis:**
   * **Lagrange Method:** Easy to understand and implement but computationally expensive for large datasets.
   * **Newton Method:** More efficient for adding new data points; utilizes incremental building of the polynomial.

# j) Eigenvalues and Eigenvectors

1. **Power Iteration Method:**

**Python Code:**

python

Copy code

def power\_iteration(A, num\_simulations: int):

b\_k = np.random.rand(A.shape[1])

for \_ in range(num\_simulations):

b\_k1 = np.dot(A, b\_k)

b\_k1\_norm = np.linalg.norm(b\_k1)

b\_k = b\_k1 / b\_k1\_norm

eigenvalue = np.dot(b\_k.T, np.dot(A, b\_k))

eigenvector = b\_k

return eigenvalue, eigenvector

1. **QR Algorithm:**

**Python Code:**

python

Copy code

def qr\_algorithm(A, num\_iterations: int):

n = A.shape[0]

Q = np.eye(n)

R = A.copy()

for \_ in range(num\_iterations):

Q\_i, R\_i = np.linalg.qr(R)

Q = Q @ Q\_i

R = R\_i @ Q\_i

eigenvalues = np.diag(R)

return eigenvalues, Q

1. **Comparison:**
   * **Power Iteration:** Effective for the largest eigenvalue, but not suitable for finding all eigenvalues.
   * **QR Algorithm:** More comprehensive and robust, capable of finding all eigenvalues and eigenvectors but computationally intensive.

# K) Gradient Descent Method

import numpy as np

def f(x, y):

return x\*\*2 + y\*\*2 - x\*y + x - y + 1

def grad\_f(x, y):

df\_dx = 2\*x - y + 1

df\_dy = 2\*y - x - 1

return np.array([df\_dx, df\_dy])

def gradient\_descent(learning\_rate=0.1, num\_iterations=1000):

x, y = 0, 0 # Initial guess

for \_ in range(num\_iterations):

gradient = grad\_f(x, y)

x -= learning\_rate \* gradient[0]

y -= learning\_rate \* gradient[1]

return x, y

x\_min, y\_min = gradient\_descent()

print(f"Minimum at x = {x\_min}, y = {y\_min}")