**Unit - 1**

**Rayleigh power method**

def rayleigh\_power\_method(A, num\_iterations):

n = len(A)

x = [1] \* n

for \_ in range(num\_iterations):

y = [0] \* n

for i in range(n):

for j in range(n):

y[i] += A[i][j] \* x[j]

norm = max(y)

x = [val / norm for val in y]

return norm, x

# Example usage

A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

num\_iterations = 10

eigenvalue, eigenvector = rayleigh\_power\_method(A, num\_iterations)

print("Eigenvalue:", eigenvalue)

print("Eigenvector:", eigenvector)

**Jacobi Method**

def jacobi\_method(A, b, x0, epsilon, max\_iterations):

n = len(A)

x = x0.copy()

for \_ in range(max\_iterations):

x\_new = x.copy()

for i in range(n):

sigma = sum(A[i][j] \* x[j] for j in range(n) if j != i)

x\_new[i] = (b[i] - sigma) / A[i][i]

if max(abs(x\_new[i] - x[i]) for i in range(n)) < epsilon:

return x\_new

x = x\_new

return x

# Example usage

A = [[4, -1, 1],

[4, -8, 1],

[-2, 1, 5]]

b = [7, -21, 15]

x0 = [0, 0, 0]

epsilon = 0.001

max\_iterations = 100

solution = jacobi\_method(A, b, x0, epsilon, max\_iterations)

print("Solution:", solution)

**Given Method**

import numpy as np

def givens\_rotation(A, b):

m, n = A.shape

Q = np.eye(m)

R = np.copy(A)

for j in range(n):

for i in range(m-1, j, -1):

if R[i, j] != 0:

r = np.sqrt(R[i-1, j]\*\*2 + R[i, j]\*\*2)

c = R[i-1, j] / r

s = -R[i, j] / r

G = np.eye(m)

G[[i-1, i], [i-1, i]] = c

G[i-1, i] = s

G[i, i-1] = -s

R = np.dot(G, R)

Q = np.dot(Q, G.T)

b = np.dot(G, b)

return Q, R, b

# Example usage

A = np.array([[4, 3], [3, 2]])

b = np.array([1, 1])

Q, R, b = givens\_rotation(A, b)

print("Q:")

print(Q)

print("R:")

print(R)

print("b:")

print(b)

**Fixed point iteration method**

import numpy as np

def fixed\_point\_iteration(g, x0, tol=1e-6, max\_iter=100):

x = x0

for i in range(max\_iter):

x\_new = g(x)

if np.abs(x\_new - x) < tol:

return x\_new, i+1

x = x\_new

return x, max\_iter

# Example usage

def g(x):

return np.exp(-x)

x0 = 0 # Initial guess

root, num\_iter = fixed\_point\_iteration(g, x0)

print("Approximate root:", root)

print("Number of iterations:", num\_iter)

**Thomas Algorithm for Tridiagonal systems**

import numpy as np

def thomas\_algorithm(a, b, c, d):

n = len(d)

c\_temp = np.copy(c)

d\_temp = np.copy(d)

# Forward elimination

for i in range(1, n):

m = a[i] / c\_temp[i-1]

c\_temp[i] = c\_temp[i] - m \* b[i-1]

d\_temp[i] = d\_temp[i] - m \* d\_temp[i-1]

# Backward substitution

x = np.zeros\_like(d)

x[-1] = d\_temp[-1] / c\_temp[-1]

for i in range(n-2, -1, -1):

x[i] = (d\_temp[i] - b[i] \* x[i+1]) / c\_temp[i]

return x

# Example usage

a = np.array([1, 2, 3]) # Lower diagonal

b = np.array([4, 5, 6]) # Main diagonal

c = np.array([7, 8, 9]) # Upper diagonal

d = np.array([10, 11, 12]) # Right-hand side

x = thomas\_algorithm(a, b, c, d)

print("Solution:")

print(x)

**Newton Method for solving nonlinear systems**

def newton\_method(f, df, x0, tol=1e-6, max\_iter=100):

x = x0

for i in range(max\_iter):

fx = f(x)

if abs(fx) < tol:

return x

dfx = df(x)

if dfx == 0:

break

x = x - fx / dfx

return None

# Example usage:

def f(x):

return x\*\*2 - 2

def df(x):

return 2 \* x

root = newton\_method(f, df, 1.5)

if root is not None:

print("Root:", root)

else:

print("No root found within the specified tolerance.")

**Unit - 2**

**Linear interpolation**

def linear\_interpolation(x, x\_values, y\_values):

n = len(x\_values)

if x <= x\_values[0]:

return y\_values[0]

elif x >= x\_values[n-1]:

return y\_values[n-1]

for i in range(1, n):

if x < x\_values[i]:

slope = (y\_values[i] - y\_values[i-1]) / (x\_values[i] - x\_values[i-1])

y = y\_values[i-1] + slope \* (x - x\_values[i-1])

return y

# Example usage

x\_values = [1, 2, 3, 4, 5] # Known x values

y\_values = [10, 15, 7, 9, 12] # Known y values

x = 2.5 # Value to interpolate

y = linear\_interpolation(x, x\_values, y\_values)

print("Interpolated value at x =", x, ":", y)

**Piecewise polynomial interpolation : Cubic spline interpolation**

from scipy.interpolate import CubicSpline

def piecewise\_polynomial\_interpolation(x, x\_values, y\_values):

cs = CubicSpline(x\_values, y\_values)

return cs(x)

# Example usage

x\_values = [1, 2, 3, 4, 5] # Known x values

y\_values = [10, 15, 7, 9, 12] # Known y values

x = 2.5 # Value to interpolate

y = piecewise\_polynomial\_interpolation(x, x\_values, y\_values)

print("Interpolated value at x =", x, ":", y)

**Sterling’s formula**

def sterling\_interpolation(x, x\_values, y\_values):

n = len(x\_values)

if n != len(y\_values):

raise ValueError("x\_values and y\_values must have the same length")

result = 0.0

for i in range(n):

term = y\_values[i]

for j in range(n):

if i != j:

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

result += term

return result

# Example usage:

x\_values = [1, 2, 4, 5]

y\_values = [3, 5, 6, 8]

x = 3.5

interpolated\_value = sterling\_interpolation(x, x\_values, y\_values)

print("Interpolated value at x =", x, ":", interpolated\_value)

**Bessel’s formula**

import numpy as np

def bessel\_interpolation(x, y, x\_interp):

n = len(x)

m = len(x\_interp)

y\_interp = np.zeros(m)

for k in range(m):

for i in range(n):

numerator = np.math.factorial(n - 1 + i) \* np.math.factorial(n - 1 - i)

denominator = (np.math.factorial(2 \* n - 2) \* (x\_interp[k] - x[i])\*\*(n - 1))

y\_interp[k] += y[i] \* numerator / denominator

return y\_interp

# Example usage:

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

y = np.array([0, 1, 0, -1])

x\_interp = np.linspace(1, 4, 100)

y\_interp = bessel\_interpolation(x, y, x\_interp)

print("Interpolated value at x =", x, ":", y\_interp )

**Richardson Extrapolation**

def richardson\_extrapolation(f, x, h, num\_terms):

R = np.zeros((num\_terms, num\_terms))

for i in range(num\_terms):

R[i, 0] = (f(x + h) - f(x - h)) / (2 \* h)

for j in range(1, i+1):

R[i, j] = R[i, j-1] + (R[i, j-1] - R[i-1, j-1]) / ((4 \*\* j) - 1)

h /= 2

return R[num\_terms-1, num\_terms-1]

# Example usage

import numpy as np

def f(x):

return np.sin(x)

x = 1.0 # Value at which to evaluate the derivative

h = 0.1 # Initial step size

num\_terms = 5 # Number of Richardson extrapolation terms

derivative = richardson\_extrapolation(f, x, h, num\_terms)

print("Approximate derivative:", derivative)

**Boole and Romberg Intergrations**

import numpy as np

def boole\_integration(f, a, b, n):

h = (b - a) / n

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

y = f(x)

integral = 0.0

for i in range(0, n//4):

integral += (7\*y[4\*i] + 32\*y[4\*i+1] + 12\*y[4\*i+2] + 32\*y[4\*i+3] + 7\*y[4\*i+4]) \* h / 90

return integral

def romberg\_integration(f, a, b, n):

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

h = b - a

R[0, 0] = 0.5 \* h \* (f(a) + f(b))

for i in range(1, n):

h /= 2

sum\_term = 0.0

for k in range(1, 2\*\*(i-1) + 1):

sum\_term += f(a + (2\*k - 1) \* h)

R[i, 0] = 0.5 \* R[i-1, 0] + h \* sum\_term

for j in range(1, i+1):

R[i, j] = R[i, j-1] + (R[i, j-1] - R[i-1, j-1]) / (4\*\*j - 1)

return R[n-1, n-1]

# Example usage:

def f(x):

return np.sin(x)

a = 0

b = np.pi

n = 10

boole\_integral = boole\_integration(f, a, b, n)

romberg\_integral = romberg\_integration(f, a, b, n)

print("Boole's Integral:", boole\_integral)

print("Romberg's Integral:", romberg\_integral)

**Evaluation of Double Integrals using Numerical Methods:**

1. **Trapezoidal Rule**

def f(x, y):

"""The integrand function"""

return x\*\*2 + y\*\*2

def trapezoidal\_double\_integral(f, a, b, c, d, n, m):

h\_x = (b - a) / n # width of each interval in x-direction

h\_y = (d - c) / m # width of each interval in y-direction

integral\_sum = 0.0

for i in range(n):

x\_i = a + i \* h\_x # x-coordinate of the left endpoint of the interval

x\_ip1 = x\_i + h\_x # x-coordinate of the right endpoint of the interval

for j in range(m):

y\_j = c + j \* h\_y # y-coordinate of the lower endpoint of the interval

y\_jp1 = y\_j + h\_y # y-coordinate of the upper endpoint of the interval

# Calculate the average of the function values at the four corners of the interval

average = (f(x\_i, y\_j) + f(x\_i, y\_jp1) + f(x\_ip1, y\_j) + f(x\_ip1, y\_jp1)) / 4

integral\_sum += average \* h\_x \* h\_y # add the average multiplied by the area of the interval

return integral\_sum

# Example usage

result = trapezoidal\_double\_integral(f, 0, 1, 0, 1, 100, 100)

print("Approximate value of the double integral:", result)

1. **Simpson Rule**

import numpy as np

def double\_integral\_simpson13(f, a, b, c, d, m, n):

hx = (b - a) / m

hy = (d - c) / n

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

y = np.linspace(c, d, n+1)

integral = 0.0

for i in range(m):

for j in range(n):

if (i+j) % 2 == 0:

coefficient = 1

else:

coefficient = 4

integral += coefficient \* f(x[i], y[j])

integral \*= (hx \* hy) / 9

return integral

# Example usage:

def f(x, y):

return np.sin(x) + np.cos(y)

a = 0

b = np.pi

c = 0

d = np.pi

m = 10

n = 10

integral = double\_integral\_simpson13(f, a, b, c, d, m, n)

print("Double Integral:", integral)