**Lecture 3:**

**Example1:** A root of lies in the interval (0, 1). Use rootsearch to compute this root with four-digit accuracy.

from numpy import sign

def rootsearch(f,a,b,dx):

x1 = a; f1 = f(a)

x2 = a + dx; f2 = f(x2)

while sign(f1) == sign(f2):

if x1 >= b: return None,None

x1 = x2; f1 = f2

x2 = x1 + dx; f2 = f(x2)

print('\nx1 is:', np.round(x1,3), ' f(x1) = ', np.round(f1,3), ' x2 is: ', np.round(x2,3),' f(x2) = ', np.round(f2,3) )

else:

return x1,x2

print(x1,x2)

import numpy as np

def f(x): return x\*\*3 - 10.0\*x\*\*2 + 5.0

x1 = 0.0; x2 = 1.0

for i in range(4):

print('\n\nstep:',i,':')

dx = (x2 - x1)/10.0

x1,x2 = rootsearch(f,x1,x2,dx)

print('\ndx=',dx)

x = (x1 + x2)/2.0

print('x =', np.round(x,4))

**Example 2:** Use bisection to find the root of that lies in the interval (0,1) to four-digit accuracy.

import numpy as np

import math

# import error

from numpy import sign

def bisection(f,x1,x2,switch=1,tol=1.0e-9):

f1 = f(x1)

if f1 == 0.0: return x1

f2 = f(x2)

if f2 == 0.0: return x2

if sign(f1) == sign(f2):

error.err('Root is not bracketed')

n = int(math.ceil(math.log(abs(x2 - x1)/tol)/math.log(2.0)))

for i in range(n):

x3 = 0.5\*(x1 + x2); f3 = f(x3)

if (switch == 1) and (abs(f3) > abs(f1)) \

and (abs(f3) > abs(f2)):

return None

if f3 == 0.0: return x3

if sign(f2)!= sign(f3): x1 = x3; f1 = f3

else: x2 = x3; f2 = f3

return (x1 + x2)/2.0

# from bisection import \*

def f(x): return x\*\*3 - 10.0\*x\*\*2 + 5.0

x = bisection(f, 0.0, 1.0, tol = 1.0e-4)

print('x =', np.round(x,4))



import math  
  
# Define the function  
def f(x):  
 return math.sqrt(x) - math.cos(x)  
  
  
p3 = bisection(f, 0, 1)  
print("\nFinal approximation p3 =", round(p3, 6))



## module ridder

# import error

import math

from numpy import sign

def ridder(f,a,b,tol=1.0e-9):

fa = f(a)

if fa == 0.0: return a

fb = f(b)

if fb == 0.0: return b

if sign(fa) == sign(fb): error.err('Root is not bracketed')

for i in range(30):

# Compute the improved root x from Ridder's formula

c = 0.5\*(a + b); fc = f(c)

s = math.sqrt(fc\*\*2 - fa\*fb)

if s == 0.0: return None

dx = (c - a)\*fc/s

if (fa - fb) < 0.0: dx = -dx

x = c + dx; fx = f(x)

# Test for convergence

if i > 0:

if abs(x - xOld) < tol\*max(abs(x),1.0): return x

xOld = x

# Re-bracket the root as tightly as possible

if sign(fc) == sign(fx):

if sign(fa)!= sign(fx): b = x; fb = fx

else: a = x; fa = fx

else:

a = c; b = x; fa = fc; fb = fx

return None

print('Too many iterations')

import numpy as np

def f(x): return x\*\*3 - 10.0\*x\*\*2 + 5.0

x = ridder(f, 0.0, 1.0, tol = 1.0e-4)

print('x =', np.round(x,4))

**Lecture 4:**

**Example 1:** Use the Newton-Raphson method to obtain successive approximations of as the ratio of two integers.

def f(x): return x\*\*2-2

def df(x): return 2\*x

def newtonRaphson(x, tol=1.0e-7):

for i in range(30):

dx=-f(x)/df(x)

x = x+dx

if abs(dx) < tol: return x,i

print('Too many iterations\n')

root,numIter = newtonRaphson(2.0)

print('Root=',root)

print('Number of iterations=',numIter)

**Example 2:** Find the smallest positive zero of

def f(x): return x\*\*4-6.4\*x\*\*3+6.45\*x\*\*2+20.538\*x-31.752

def df(x): return 4\*x\*\*3-19.2\*x\*\*2+12.9\*x+20.538

def newtonRaphson(x, tol=1.0e-9):

for i in range(30):

dx=-f(x)/df(x)

x = x+dx

if abs(dx) < tol: return x,i

print('Too many iterations\n')

root,numIter = newtonRaphson(2.0)

print('Root=',root)

print('Number of iterations=',numIter)

**Example 3:** Use Newton’s method to find solutions accurate to within for the following problems.

, [1,4]

def f(x):

return x\*\*3 - 2\*x\*\*2 - 5

def df(x):

return 3\*x\*\*2 - 4\*x

def newtonRaphson(f, df, a, b, tol=1.0e-9):

from numpy import sign

fa = f(a)

fb = f(b)

if abs(fa) < tol:

return a, 0

if abs(fb) < tol:

return b, 0

if sign(fa) == sign(fb):

raise ValueError("Root is not bracketed")

x = 0.5 \* (a + b)

for i in range(1, 31):

fx = f(x)

dfx = df(x)

if abs(fx) < tol:

return x, i

# Bracket update

if sign(fa) != sign(fx):

b = x

fb = fx

else:

a = x

fa = fx

try:

dx = -fx / dfx

except ZeroDivisionError:

dx = b - a # fallback to bisection

x\_new = x + dx

# Keep x within [a, b]

if (x\_new - a) \* (x\_new - b) > 0:

dx = 0.5 \* (b - a)

x\_new = a + dx

x = x\_new

if abs(dx) < tol \* max(abs(x), 1.0):

return x, i

raise RuntimeError("Too many iterations in Newton-Raphson")

# Call the function

root, iterations = newtonRaphson(f, df, a=1.0, b=4.0)

print("Root =", root)

print("Iterations =", iterations)

**Systems of Equations:**

**Example 4: Find a solution of**

**Using newtonRaphson2. Start with the point (1,1,1).**

import numpy as np

import math

def newtonRaphson2(f, x, tol=1.0e-9):

def jacobian(f, x):

h = 1.0e-4

n = len(x)

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

f0 = f(x)

for i in range(n):

x1 = x.copy()

x1[i] += h

f1 = f(x1)

jac[:, i] = (f1 - f0) / h

return jac, f0

for i in range(30):

jac, f0 = jacobian(f, x)

if math.sqrt(np.dot(f0, f0) / len(x)) < tol:

return x, i

dx = np.linalg.solve(jac, -f0)

x = x + dx

if math.sqrt(np.dot(dx, dx)) < tol \* max(np.max(np.abs(x)), 1.0):

return x, i

raise RuntimeError("Too many iterations")

# Define the system of equations

def f(x):

f = np.zeros(len(x))

f[0] = math.sin(x[0]) + x[1]\*\*2 + math.log(x[2]) - 7.0

f[1] = 3.0\*x[0] + 2.0\*\*x[1] - x[2]\*\*3 + 1.0

f[2] = x[0] + x[1] + x[2] - 5.0

return f

x0 = np.array([1.0, 1.0, 1.0])

root, iterations = newtonRaphson2(f, x0)

print("Root:", root)

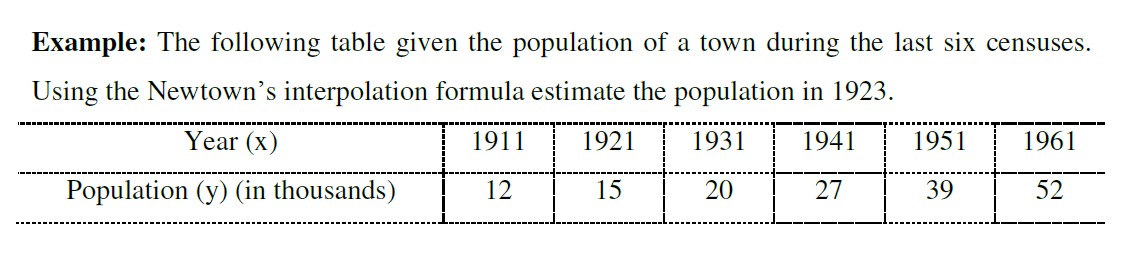
print("Iterations:", iterations)

print("f(root):", f(root))

input("\nPress return to exit")

**Lecture 5:**

**Newton-Gregory Forward interpolation formula**



import math

years = [1911, 1921, 1931, 1941, 1951, 1961]

populations = [12, 15, 20, 27, 39, 52]

h = years[1] - years[0]

x = 1923

x0 = years[0]

t = (x - x0) / h

n = len(populations)

diff\_table = [populations[:]]

for i in range(1, n):

column = []

for j in range(n - i):

delta = diff\_table[i-1][j+1] - diff\_table[i-1][j]

column.append(delta)

diff\_table.append(column)

def newtons\_forward(t, diff\_table):

result = diff\_table[0][0]

u\_term = 1

for i in range(1, len(diff\_table)):

u\_term \*= (t - i + 1)

term = (u\_term \* diff\_table[i][0]) / math.factorial(i)

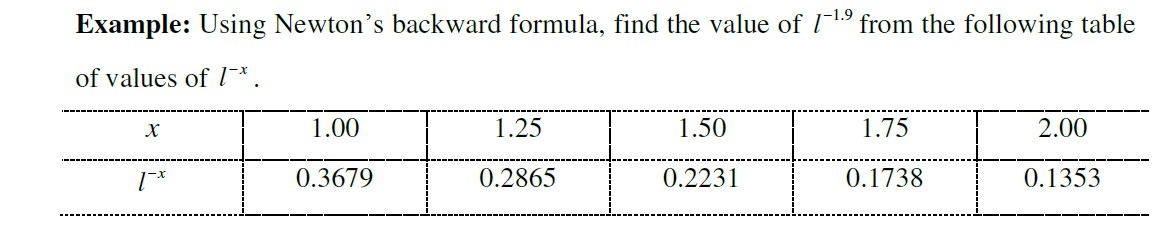
result += term

return result

estimated\_population = newtons\_forward(t, diff\_table)

print(f"Estimated population in {x} is approximately {estimated\_population:.2f} thousand")

**Newton-Gregory Backward interpolation formula:**



import math

x\_values = [1.00, 1.25, 1.50, 1.75, 2.00]

y\_values = [0.3679, 0.2865, 0.2231, 0.1738, 0.1353]

h = x\_values[1] - x\_values[0]

x = 1.9

n = len(x\_values)

u = (x - x\_values[-1])/h

diff\_table = [y\_values[:]]

for i in range(1, n):

column = []

for j in range(n - i):

delta = diff\_table[i-1][j+1] - diff\_table[i-1][j]

column.append(delta)

diff\_table.append(column)

def newtons\_backward(u, diff\_table):

result = diff\_table[0][-1]

u\_term = 1

for i in range(1, len(diff\_table)):

u\_term \*= (u + i - 1)

term = (u\_term \* diff\_table[i][-1]) / math.factorial(i)

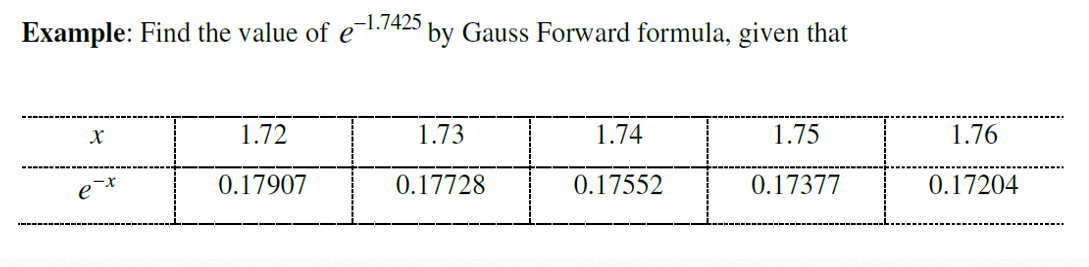
result += term

return result

estimated\_value = newtons\_backward(u, diff\_table)

print(f"Estimated value of x ln(x) at x = {x} is approximately {estimated\_value:.4f}")

**Lecture 6:**



import numpy as np

x\_values = [1.72, 1.73, 1.74, 1.75, 1.76]

y\_values = [0.17907, 0.17728, 0.17552, 0.17377, 0.17204]

n = len(x\_values)

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i + 1][j - 1] - diff\_table[i][j - 1]

x = 1.7425

x0 = x\_values[2]

h = x\_values[1] - x\_values[0]

p = (x - x0) / h

result = diff\_table[2][0]

p\_term = 1

fact = 1

for k in range(1, 4):

if k == 1:

p\_term \*= p

elif k == 2:

p\_term \*= (p - 1)

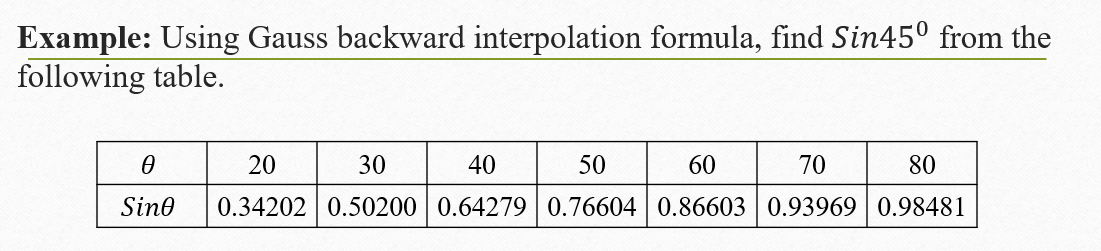
elif k == 3:

p\_term \*= (p + 1)

fact \*= k

result += (p\_term / fact) \* diff\_table[2 - (k // 2)][k]

print(f"Estimated value of e^(-1.7425) using Gauss Forward Interpolation: {result:.6f}")



import numpy as np

x\_values = [20, 30, 40, 50, 60, 70, 80]

y\_values = [0.3420, 0.5000, 0.6428, 0.7660, 0.8660, 0.9397, 0.9848]

n = len(x\_values)

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n-j):

diff\_table[i][j] = diff\_table[i][j - 1] - diff\_table[i - 1][j - 1]

x = 45

x\_n\_index = x\_values.index(50)

x\_n = x\_values[x\_n\_index]

h = 10

p = (x - x\_n) / h

result = diff\_table[x\_n\_index][0]

p\_term = 1

fact = 1

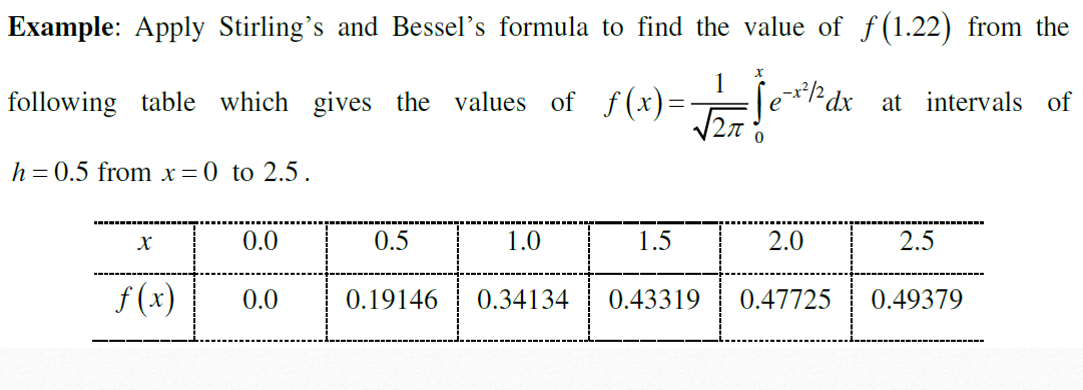
for k in range(1, 5): # up to 4th order

p\_term \*= (p + (k - 1))

fact \*= k

result += (p\_term / fact) \* diff\_table[x\_n\_index][k]

print(f"Estimated sin(45°) using Gauss Backward Interpolation: {result:.6f}")



**#Stirling's Interpolation:**

import numpy as np

x\_values = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5]

y\_values = [0.0, 0.19146, 0.34134, 0.43319, 0.47725, 0.49379]

n = len(x\_values)

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i + 1][j - 1] - diff\_table[i][j - 1]

x = 1.22

h = 0.5

p = (x - x\_values[2]) / h

result = diff\_table[2][0]

factorial = 1

p\_term = 1

sign = 1

for k in range(1, 5): # up to 4th order

factorial \*= k

if k % 2 == 1:

term = (diff\_table[2 - k//2][k] + diff\_table[2 - k//2 + 1][k]) / 2

p\_term \*= (p \*\* k)

else:

term = diff\_table[2 - k//2][k]

p\_term \*= (p \*\* k)

result += p\_term \* term / factorial

print(f"Estimated f(1.22) using Stirling's Interpolation: {result:.6f}")

**#Bessel’s Interpolation**

import numpy as np

x\_values = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5]

y\_values = [0.0, 0.19146, 0.34134, 0.43319, 0.47725, 0.49379]

n = len(x\_values)

h = 0.5

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i + 1][j - 1] - diff\_table[i][j - 1]

x\_interp = 1.22

p = (x\_interp - (x\_values[2] + x\_values[3]) / 2) / h

y0 = diff\_table[2][0]

y1 = diff\_table[3][0]

delta\_y0 = diff\_table[2][1]

delta\_y1 = diff\_table[3][1]

delta2\_y0 = diff\_table[2][2]

delta3\_y0 = diff\_table[2][3]

delta4\_y0 = diff\_table[2][4]

f\_x = (y0 + y1)/2 \

+ p \* (delta\_y1 - delta\_y0)/2 \

+ (p\*\*2) \* delta2\_y0 / 2 \

+ (p\*(p\*\*2 - 1)) \* (delta3\_y0)/6 \

+ ((p\*\*2) \* (p\*\*2 - 1)) \* delta4\_y0 / 24

print(f"Estimated f(1.22) using Bessel's Interpolation: {f\_x:.6f}")

**Lecture 7:**

**Example1:** Using Lagrange’s interpolation formula find from the following data.

x=[0,1,2,15]

y=[2,3,12,3587]

xv=4

def lagrange\_interpolation(x,y,xv):

n=len(x)

result=0.0

for i in range(n):

term=y[i]

for j in range(n):

if i!=j:

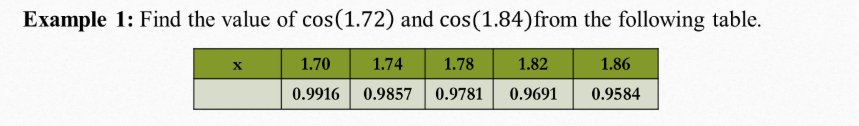
term \*= (xv-x[j])/(x[i]-x[j])

result += term

return result

estimated\_value=lagrange\_interpolation(x,y,xv)

print(f"Estimated value of f(4): {estimated\_value:.2f}")



#part 1

import numpy as np

import math

x\_values = [1.70, 1.74, 1.78, 1.82, 1.86]

y\_values = [0.9916, 0.9857, 0.9781, 0.9691, 0.9584]

h = x\_values[1] - x\_values[0]

diff = [

[y\_values[0],

y\_values[1] - y\_values[0],

y\_values[2] - 2 \* y\_values[1] + y\_values[0],

y\_values[3] - 3 \* y\_values[2] + 3 \* y\_values[1] - y\_values[0],

y\_values[4] - 4 \* y\_values[3] + 6 \* y\_values[2] - 4 \* y\_values[1] + y\_values[0]]

]

t = (1.72 - 1.70) / h

def derivative\_newton\_forward(t, h, diff):

result = diff[0][1]

if len(diff[0]) >= 3:

result += ((2 \* t - 1) / 2) \* diff[0][2]

if len(diff[0]) >= 4:

result += ((3 \* t\*\*2 - 6 \* t + 2) / 6) \* diff[0][3]

if len(diff[0]) >= 5:

result += ((4 \* t\*\*3 - 18 \* t\*\*2 + 22 \* t - 6) / 24) \* diff[0][4]

return result / h

approx\_cos\_1\_72 = derivative\_newton\_forward(t, h, diff)

print(f"Approximate cos(1.72): {approx\_cos\_1\_72}")

#part 2

import numpy as np

x\_values = [1.70, 1.74, 1.78, 1.82, 1.86]

y\_values = [0.9916, 0.9857, 0.9781, 0.9691, 0.9584]

h = x\_values[1] - x\_values[0]

diff = [[

y\_values[-1],

y\_values[-1] - y\_values[-2],

y\_values[-1] - 2\*y\_values[-2] + y\_values[-3],

y\_values[-1] - 3\*y\_values[-2] + 3\*y\_values[-3] - y\_values[-4],

y\_values[-1] - 4\*y\_values[-2] + 6\*y\_values[-3] - 4\*y\_values[-4] + y\_values[-5]

]]

t = (1.84 - x\_values[-1]) / h

def derivative\_newton\_backward(t, h, diff):

result = diff[0][1]

if len(diff[0]) >= 3:

result += ((2 \* t + 1) / 2) \* diff[0][2]

if len(diff[0]) >= 4:

result += ((3 \* t\*\*2 + 6 \* t + 2) / 6) \* diff[0][3]

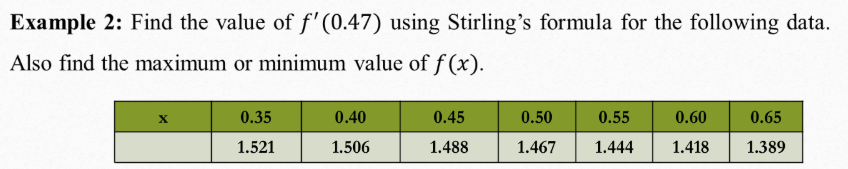
if len(diff[0]) >= 5:

result += ((4 \* t\*\*3 + 18 \* t\*\*2 + 22 \* t + 6) / 24) \* diff[0][4]

return result / h

approx\_cos\_1\_84 = derivative\_newton\_backward(t, h, diff)

print(f"Approximate cos(1.84): {approx\_cos\_1\_84:.6f}")



import numpy as np

x\_values = [0.35, 0.40, 0.45, 0.50, 0.55, 0.60, 0.65]

y\_values = [1.521, 1.506, 1.488, 1.467, 1.444, 1.418, 1.389]

h = x\_values[1] - x\_values[0] # uniform interval = 0.05

n = len(x\_values)

# Build the difference table

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i+1][j-1] - diff\_table[i][j-1]

# Find central value index near 0.47

mid\_index = 2 # x = 0.45 (3rd value) is closest to 0.47

x0 = x\_values[mid\_index]

p = (0.47 - x0) / h

# Apply Stirling’s formula for the first derivative:

dy\_dx = (1 / h) \* (

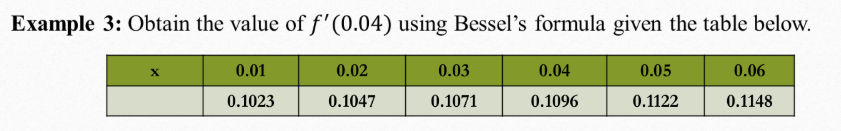
(diff\_table[mid\_index][1] + diff\_table[mid\_index - 1][1]) / 2

+ p \* (diff\_table[mid\_index - 1][2] / 2 + diff\_table[mid\_index - 1][2] / 2) # approximate central second difference

+ (p\*\*2 - 1) \* (diff\_table[mid\_index - 1][3] + diff\_table[mid\_index - 2][3]) / 4

)

print(f"Approximate derivative at x = 0.47 using Stirling's formula: {dy\_dx:.6f}")



import numpy as np

x\_values = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06]

y\_values = [0.1023, 0.1047, 0.1071, 0.1096, 0.1122, 0.1148]

h = x\_values[1] - x\_values[0] # Assuming equal spacing

n = len(x\_values)

# Build forward difference table

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i + 1][j - 1] - diff\_table[i][j-1]

mid\_index = 2 # y at x=0.03

p = (0.04 - x\_values[mid\_index]) / h

# Bessel’s first derivative formula components:

dy\_dx = (1 / h) \* (

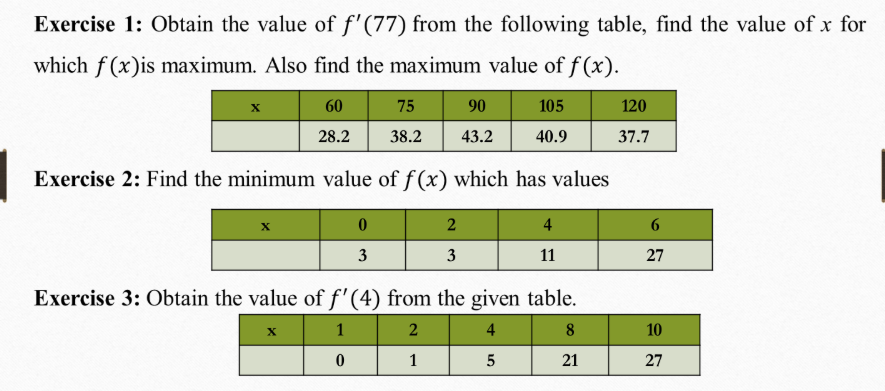
diff\_table[mid\_index][1] + diff\_table[mid\_index + 1][1]

) / 2

dy\_dx += (p - 0.5) \* diff\_table[mid\_index][2]

dy\_dx += ((3 \* p\*\*2 - 1) / 12) \* (diff\_table[mid\_index][3] + diff\_table[mid\_index + 1][3]) / 2

print(f"Approximate derivative at x = 0.04 using Bessel's formula: {dy\_dx:.6f}")



# Exercise 1:

import numpy as np

x\_values = [60, 75, 90, 105, 120]

y\_values = [28.2, 38.2, 43.2, 40.9, 37.7]

h = x\_values[1] - x\_values[0] # step size

n = len(x\_values)

# Build forward difference table

diff\_table = np.zeros((n, n))

diff\_table[:, 0] = y\_values

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i+1][j-1] - diff\_table[i][j-1]

# Stirling’s formula around 75

mid\_index = 1 # x = 75

p = (77 - x\_values[mid\_index]) / h

fprime\_77 = (1/h) \* (

(diff\_table[mid\_index][1] + diff\_table[mid\_index-1][1]) / 2

+ (p/2) \* diff\_table[mid\_index-1][2]

+ ((2\*p\*\*2 - 1)/6) \* ((diff\_table[mid\_index][3] + diff\_table[mid\_index-1][3]) / 2)

)

print(f"Approximate f'(77) = {fprime\_77:.6f}")

print(f"Maximum value of f(x) = {max(y\_values)} at x = {x\_values[y\_values.index(max(y\_values))]}")

#Exercise 2:

x\_values = [0, 2, 4, 6]

y\_values = [3, 3, 11, 27]

min\_val = min(y\_values)

x\_min = x\_values[y\_values.index(min\_val)]

print(f"Minimum value of f(x) = {min\_val} at x = {x\_min}")

#Exercise 3:

import numpy as np

x\_values = [1, 2, 4, 8, 10]

y\_values = [0, 1, 5, 21, 27]

# Lagrange interpolation derivative

def lagrange\_derivative(x\_vals, y\_vals, x0):

n = len(x\_vals)

result = 0

for i in range(n):

Li\_prime = 0

for j in range(n):

if j != i:

prod = 1

for k in range(n):

if k != i and k != j:

prod \*= (x0 - x\_vals[k])

denom = np.prod([x\_vals[i] - x\_vals[m] for m in range(n) if m != i])

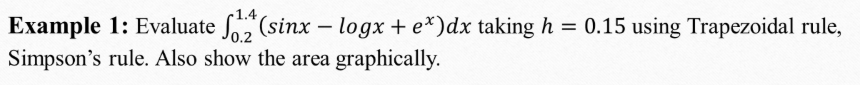
Li\_prime += prod / denom

result += y\_vals[i] \* Li\_prime

return result

fprime\_4 = lagrange\_derivative(x\_values, y\_values, 4)

print(f"Approximate f'(4) = {fprime\_4:.6f}")



#part 1

import numpy as np

def f(x):

return np.sin(x) - np.log(x) + np.exp(x)

a = 0.2

b = 1.4

h = 0.15

x = np.arange(a, b + h, h)

y = f(x)

print("x values:", x)

print("f(x) values:", y)

t=(h/2)\*(y[0]+2\*(y[1]+y[2]+y[3]+y[4]+y[5]+y[6]+y[7])+y[8])

s1=(h/3)\*(y[0]+4\*(y[1]+y[3]+y[5]+y[7])+2\*(y[2]+y[4]+y[6])+y[8])

s2=(3\*h/8)\*(y[0]+3\*(y[1]+y[2]+y[4]+y[5]+y[7])+2\*(y[3]+y[6])+y[8])

print("Trapezoidal value:", t)

print("Simpson's one-third value:", s1)

print("Simpson's three-eight value:", s2)

#part 2

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

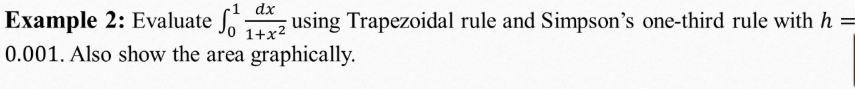
plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()



#part 1

import numpy as np

def f(x):

return 1/(1+(x\*\*2))

a = 0

b = 1

h = 0.001

x = np.arange(a, b + h, h)

y = f(x)

n=len(x)

print("length:", n)

print("x values:", x)

print("f(x) values:", y)

t1=0

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

t1 += y[i]

trap=(h/2)\*(y[0]+2\*t1+y[n-1])

print("Trapezoidal value:", trap)

#part 2

s1=0

s2=0

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

s1 += y[i]

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

s2 += y[i]

sim1=(h/3)\*(y[0]+4\*s1+2\*s2+y[n-1])

print("Simpson's one-third value:", sim1)

#part 3

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3 \* h / 8) \* (y[0] + 3 \* s3 + 2 \* s4 + y[n - 1])

print("Simpson's three-eighth value:", sim3)

#part 4

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()



#part 1

import numpy as np

def f(x):

return x\*np.exp(x)

a = 0

b = 1

h = 0.001

x = np.arange(a, b + h, h)

y = f(x)

n=len(x)

print("length:", n)

print("x values:", x)

print("f(x) values:", y)

t1=0

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

t1 += y[i]

trap=(h/2)\*(y[0]+2\*t1+y[n-1])

s1=0

s2=0

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

s1 += y[i]

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

s2 += y[i]

sim1=(h/3)\*(y[0]+4\*s1+2\*s2+y[n-1])

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3 \* h / 8) \* (y[0] + 3 \* s3 + 2 \* s4 + y[n - 1])

print("Trapezoidal value:", trap)

print("Simpson's one-third value:", sim1)

print("Simpson's three-eighth value:", sim3)

#part 2

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

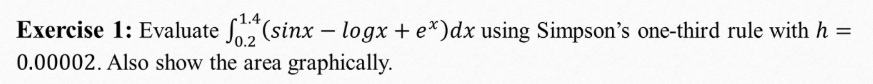
plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()



#part 1

import numpy as np

def f(x):

return np.sin(x) - np.log(x) + np.exp(x)

a = 0.2

b = 1.4

h = 0.00002

x = np.arange(a, b + h, h)

y = f(x)

n=len(x)

print("length:", n)

print("x values:", x)

print("f(x) values:", y)

t1=0

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

t1 += y[i]

trap=(h/2)\*(y[0]+2\*t1+y[n-1])

s1=0

s2=0

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

s1 += y[i]

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

s2 += y[i]

sim1=(h/3)\*(y[0]+4\*s1+2\*s2+y[n-1])

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3 \* h / 8) \* (y[0] + 3 \* s3 + 2 \* s4 + y[n - 1])

print("Trapezoidal value:", trap)

print("Simpson's one-third value:", sim1)

print("Simpson's three-eighth value:", sim3)

#part 2

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

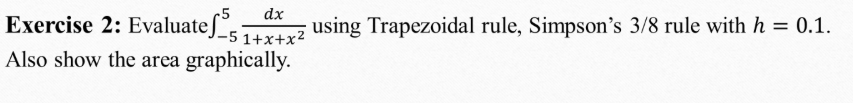
plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()



#part 1

import numpy as np

def f(x):

return 1/(1+x+x\*\*2)

a = -5.0

b = 5.0

h = 0.1

x = np.arange(a, b + h, h)

y = f(x)

n=len(x)

print("length:", n)

print("x values:", x)

print("f(x) values:", y)

t1=0

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

t1 += y[i]

trap=(h/2)\*(y[0]+2\*t1+y[n-1])

s1=0

s2=0

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

s1 += y[i]

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

s2 += y[i]

sim1=(h/3)\*(y[0]+4\*s1+2\*s2+y[n-1])

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3 \* h / 8) \* (y[0] + 3 \* s3 + 2 \* s4 + y[n - 1])

print("Trapezoidal value:", trap)

print("Simpson's one-third value:", sim1)

print("Simpson's three-eighth value:", sim3)

#part 2

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

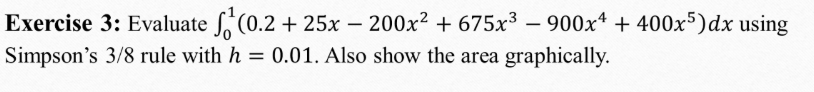
plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()



import numpy as np

import matplotlib.pyplot as plt

# Define the function

def f(x):

return 0.2 + 25\*x - 200\*x\*\*2 + 675\*x\*\*3 - 900\*x\*\*4 + 400\*x\*\*5

# Limits and step size

a = 0.0

b = 1.0

h = 0.01

# Generate x and y values

x = np.arange(a, b + h, h)

y = f(x)

n = len(x)

# Trapezoidal Rule

t1 = 0

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

t1 += y[i]

trap = (h/2) \* (y[0] + 2\*t1 + y[n-1])

# Simpson's 1/3 Rule

s1 = 0

s2 = 0

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

s1 += y[i]

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

s2 += y[i]

sim1 = (h/3) \* (y[0] + 4\*s1 + 2\*s2 + y[n-1])

# Simpson's 3/8 Rule

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3\*h/8) \* (y[0] + 3\*s3 + 2\*s4 + y[n-1])

print("Trapezoidal value:", trap)

print("Simpson's one-third value:", sim1)

print("Simpson's three-eighth value:", sim3)

# Plot the function and shaded area

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

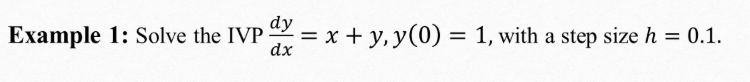
plt.legend()

plt.show()

print("length:", n)

print("x values:", x)

print("f(x) values:", y)



# Given initial values

x = 0

y = 1

h = 0.1

# First derivative

def dy\_dx(x, y):

return x + y

# Second derivative

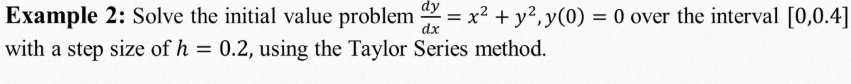
def d2y\_dx2(x, y):

return x + y + 1

# Apply Taylor Series

y\_next = y + h \* dy\_dx(x, y) + (h\*\*2 / 2) \* d2y\_dx2(x, y)

print(f"Approximate value of y(0.1) using Taylor Series: {y\_next:.3f}")



# Given initial values

x = 0

y = 0

h = 0.4

# First derivative

def dy\_dx(x, y):

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

# Second derivative

def d2y\_dx2(x, y):

return 2\*x + (2\*y\*(x\*\*2 + y\*\*2))

# Third derivative

def d3y\_dx3(x, y):

return 2 + 4\*x\*y + (6\*y\*(x\*\*2 + y\*\*2))

# Apply Taylor Series

y\_next = y + h \* dy\_dx(x, y) + (h\*\*2 / 2) \* d2y\_dx2(x, y) +(h\*\*3/6) \* d3y\_dx3(x, y)

print(f"Approximate value of y(0.4) using Taylor Series: {y\_next:.3f}")

# Euler's Method

# Given initial values

x = 0

y = 1

h = 0.1

# First derivative

def dy\_dx(x, y):

return x + y

x1=0.1

y1 = y + h \* dy\_dx(x, y)

def dy1\_dx(x1, y1):

return x1 + y1

y2 = y1 + h \* dy1\_dx(x1, y1)

print(f"Approximate value of y(2) using Euler's Method: {y1:.3f}")

#Alternative for n

# Initial values

x = 0

y = 1

h = 0.1

n\_steps = 2

# Store results

x\_vals = [x]

y\_vals = [y]

# Define first derivative

def dy\_dx(x, y):

return x + y

# Loop to calculate y values

for i in range(n\_steps):

y\_new = y + h \* dy\_dx(x, y)

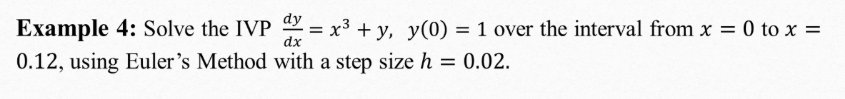
x += h

y = y\_new

x\_vals.append(x)

y\_vals.append(y)

print(f"Approximate value of y(2) using Taylor Series: {y\_vals[-1]:.3f}")



import numpy as np

a=0

b=0.12

x = np.arange(a, b+h, h)

y = 1

h = 0.02

x

n=len(x)

n

x = 0

y = 1

h = 0.02

n\_steps = 7 # To reach x = 0.12

# Store results

x\_vals = [x]

y\_vals = [y]

# Define first derivative

def dy\_dx(x, y):

return x\*\*3 + y

# Loop to calculate y values

for i in range(n\_steps):

y\_new = y + h \* dy\_dx(x, y)

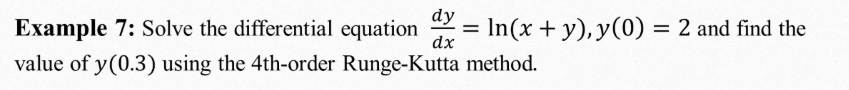
x += h

y = y\_new

x\_vals.append(x)

y\_vals.append(y)

print(f"Approximate value of y(2) using Taylor Series: {y\_vals[-1]:.3f}")



import math as m

# Function definition

def f(x, y):

return m.log(x + y)

# Initial conditions

x0 = 0

y0 = 2

h = 0.15

# Step 1: from x0 to x1

k1 = h \* f(x0, y0)

k2 = h \* f(x0 + h/2, y0 + k1/2)

k3 = h \* f(x0 + h/2, y0 + k2/2)

k4 = h \* f(x0 + h, y0 + k3)

y1 = y0 + (1/6)\*(k1 + 2\*k2 + 2\*k3 + k4)

# Step 2: from x1 to x2

x1 = x0 + h

x2 = x1 + h

k1 = h \* f(x1, y1)

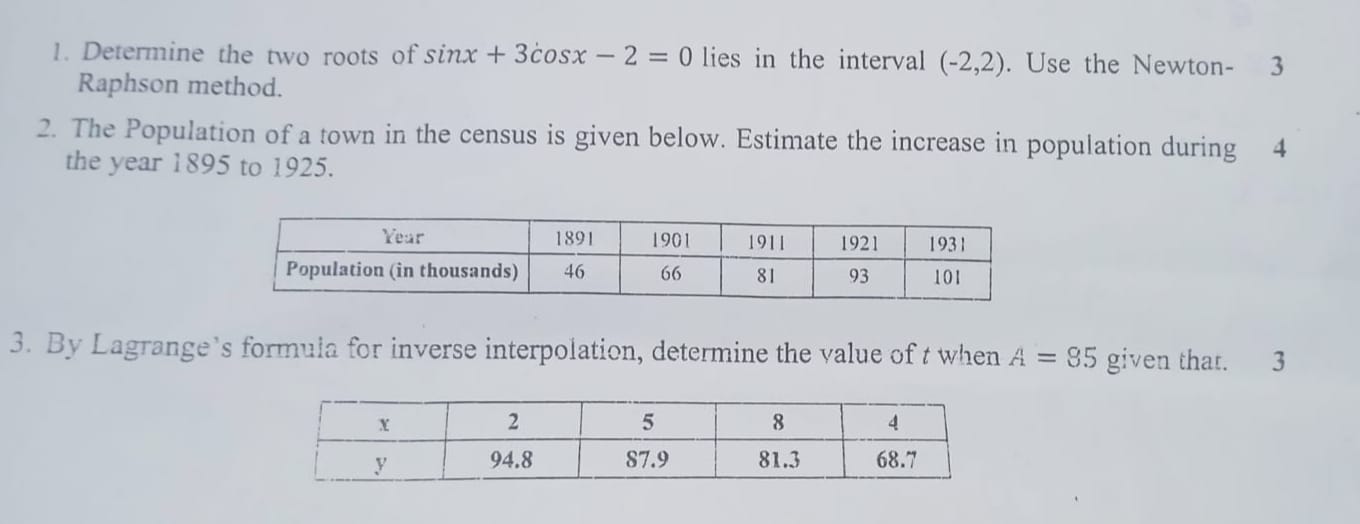
k2 = h \* f(x1 + h/2, y1 + k1/2)

k3 = h \* f(x1 + h/2, y1 + k2/2)

k4 = h \* f(x2, y1 + k3)

y2 = y1 + (1/6)\*(k1 + 2\*k2 + 2\*k3 + k4)

print(f"Approximate value of y(0.3) using RK4: {y2:.6f}")



**#CT 1**

**#Question 1**

import numpy as np

import math

def newtonRaphson(f, df, x0, tol=1.0e-9, max\_iter=100):

for i in range(max\_iter):

fx = f(x0)

dfx = df(x0)

if abs(dfx) < 1e-12: # Avoid division by zero

raise ZeroDivisionError("Derivative too small.")

x1 = x0 - fx / dfx

if abs(x1 - x0) < tol:

return x1, i + 1

x0 = x1

raise RuntimeError("Too many iterations")

# Define the function and its derivative

def f(x):

return math.sin(x) + 3 \* math.cos(x) - 2

def df(x):

return math.cos(x) - 3 \* math.sin(x)

# Initial guesses (chosen to find two roots in (-2, 2))

x0\_list = [-1.0, 1.0]

roots = []

for x0 in x0\_list:

root, iterations = newtonRaphson(f, df, x0)

roots.append(root)

print(f"Initial guess: {x0}")

print(f"Root found: {root:.6f}")

print(f"Iterations: {iterations}\n")

**#question 2**

# Newton-Gregory Forward Interpolation Formula

import math

# Given data

years = [1891, 1901, 1911, 1921, 1931]

populations = [46, 66, 81, 93, 101]

# Step size (h)

h = years[1] - years[0]

# Function to build forward difference table

def forward\_diff\_table(y\_values):

n = len(y\_values)

diff\_table = [y\_values[:]]

for i in range(1, n):

column = []

for j in range(n - i):

delta = diff\_table[i - 1][j + 1] - diff\_table[i - 1][j]

column.append(delta)

diff\_table.append(column)

return diff\_table

# Newton’s Forward Interpolation Function

def newtons\_forward(x, x0, h, diff\_table):

t = (x - x0) / h

result = diff\_table[0][0]

u\_term = 1

for i in range(1, len(diff\_table)):

u\_term \*= (t - i + 1)

term = (u\_term \* diff\_table[i][0]) / math.factorial(i)

result += term

return result

# Build difference table

diff\_table = forward\_diff\_table(populations)

# Estimate population for 1895 and 1925

x1 = 1895

x2 = 1925

pop\_1895 = newtons\_forward(x1, years[0], h, diff\_table)

pop\_1925 = newtons\_forward(x2, years[0], h, diff\_table)

# Population increase

increase = pop\_1925 - pop\_1895

# Results

print(f"Estimated population in {x1} = {pop\_1895:.2f} thousand")

print(f"Estimated population in {x2} = {pop\_1925:.2f} thousand")

print(f"Estimated increase in population (1895–1925) = {increase:.2f} thousand")

**#Question 3**

# Lagrange’s Inverse Interpolation Formula

x = [94.8, 87.9, 81.3, 68.7] # A values

y = [2, 5, 8, 4] # Corresponding t values

xv = 85 # Given A = 85, find t

def lagrange\_interpolation(x, y, xv):

n = len(x)

result = 0.0

for i in range(n):

term = y[i]

for j in range(n):

if i != j:

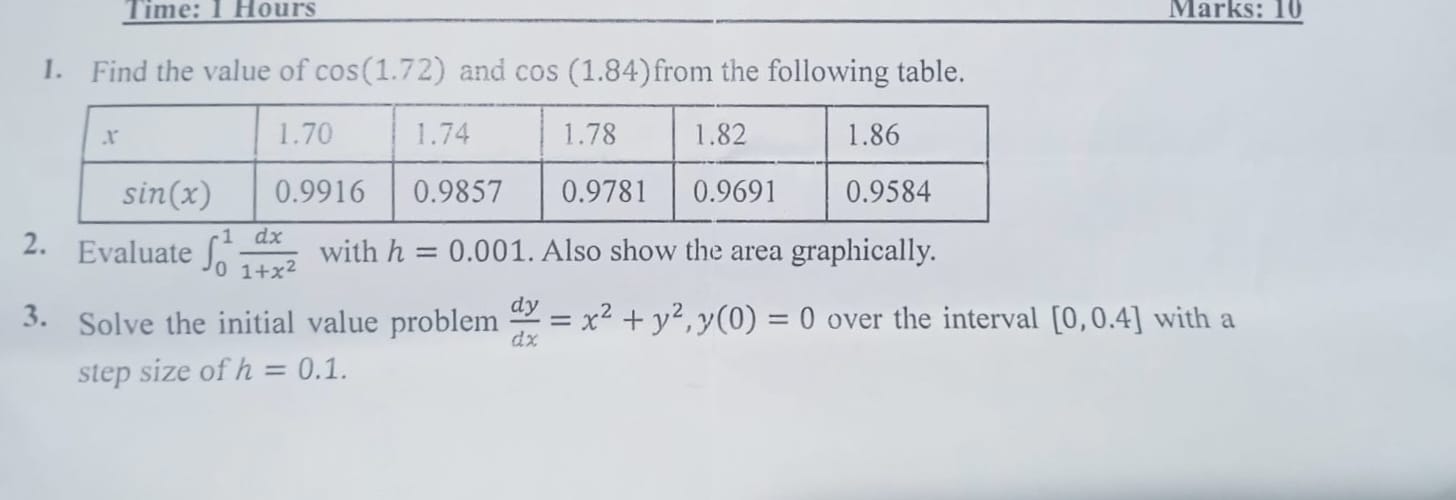
term \*= (xv - x[j]) / (x[i] - x[j])

result += term

return result

estimated\_t = lagrange\_interpolation(x, y, xv)

print(f"Estimated value of t when A = {xv} is approximately {estimated\_t:.4f}")



**#CT 2**

**#Question 1**

import numpy as np

import math

x\_values = [1.70, 1.74, 1.78, 1.82, 1.86]

y\_values = [0.9916, 0.9857, 0.9781, 0.9691, 0.9584]

h = x\_values[1] - x\_values[0]

diff = [

[y\_values[0],

y\_values[1] - y\_values[0],

y\_values[2] - 2 \* y\_values[1] + y\_values[0],

y\_values[3] - 3 \* y\_values[2] + 3 \* y\_values[1] - y\_values[0],

y\_values[4] - 4 \* y\_values[3] + 6 \* y\_values[2] - 4 \* y\_values[1] + y\_values[0]]

]

t = (1.72 - 1.70) / h

def derivative\_newton\_forward(t, h, diff):

result = diff[0][1]

if len(diff[0]) >= 3:

result += ((2 \* t - 1) / 2) \* diff[0][2]

if len(diff[0]) >= 4:

result += ((3 \* t\*\*2 - 6 \* t + 2) / 6) \* diff[0][3]

if len(diff[0]) >= 5:

result += ((4 \* t\*\*3 - 18 \* t\*\*2 + 22 \* t - 6) / 24) \* diff[0][4]

return result / h

approx\_cos\_1\_72 = derivative\_newton\_forward(t, h, diff)

print(f"Approximate cos(1.72): {approx\_cos\_1\_72}")

import numpy as np

x\_values = [1.70, 1.74, 1.78, 1.82, 1.86]

y\_values = [0.9916, 0.9857, 0.9781, 0.9691, 0.9584]

h = x\_values[1] - x\_values[0]

diff = [[

y\_values[-1],

y\_values[-1] - y\_values[-2],

y\_values[-1] - 2\*y\_values[-2] + y\_values[-3],

y\_values[-1] - 3\*y\_values[-2] + 3\*y\_values[-3] - y\_values[-4],

y\_values[-1] - 4\*y\_values[-2] + 6\*y\_values[-3] - 4\*y\_values[-4] + y\_values[-5]

]]

t = (1.84 - x\_values[-1]) / h

def derivative\_newton\_backward(t, h, diff):

result = diff[0][1]

if len(diff[0]) >= 3:

result += ((2 \* t + 1) / 2) \* diff[0][2]

if len(diff[0]) >= 4:

result += ((3 \* t\*\*2 + 6 \* t + 2) / 6) \* diff[0][3]

if len(diff[0]) >= 5:

result += ((4 \* t\*\*3 + 18 \* t\*\*2 + 22 \* t + 6) / 24) \* diff[0][4]

return result / h

approx\_cos\_1\_84 = derivative\_newton\_backward(t, h, diff)

print(f"Approximate cos(1.84): {approx\_cos\_1\_84:.6f}")

**#Question 2**

import numpy as np

def f(x):

return 1/(1+(x\*\*2))

a = 0

b = 1

h = 0.001

x = np.arange(a, b + h, h)

y = f(x)

n=len(x)

print("length:", n)

print("x values:", x)

print("f(x) values:", y)

t1=0

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

t1 += y[i]

trap=(h/2)\*(y[0]+2\*t1+y[n-1])

print("Trapezoidal value:", trap)

s1=0

s2=0

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

s1 += y[i]

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

s2 += y[i]

sim1=(h/3)\*(y[0]+4\*s1+2\*s2+y[n-1])

print("Simpson's one-third value:", sim1)

s3 = 0

s4 = 0

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

if i % 3 == 0:

s4 += y[i]

else:

s3 += y[i]

sim3 = (3 \* h / 8) \* (y[0] + 3 \* s3 + 2 \* s4 + y[n - 1])

print("Simpson's three-eighth value:", sim3)

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))

plt.plot(x, y, 'bo-', label='f(x)')

plt.fill\_between(x, y, alpha=0.3, color='skyblue')

plt.title('Area of the function')

plt.xlabel('Value of x')

plt.ylabel('Value of f(x)')

plt.legend()

plt.show()

**#Question 3**

import math as m

# Define the differential equation dy/dx = x^2 + y^2

def f(x, y):

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

# Initial conditions

x0 = 0

y0 = 0

h = 0.1

# RK4 method

def rk4\_step(x, y, h):

k1 = h \* f(x, y)

k2 = h \* f(x + h/2, y + k1/2)

k3 = h \* f(x + h/2, y + k2/2)

k4 = h \* f(x + h, y + k3)

y\_next = y + (1/6) \* (k1 + 2\*k2 + 2\*k3 + k4)

return y\_next

# Solve over [0, 0.4]

print("x\t\ty")

print(f"{x0:.1f}\t{y0:.6f}")

for i in range(4): # 4 steps from 0 to 0.4

y0 = rk4\_step(x0, y0, h)

x0 += h

print(f"{x0:.1f}\t{y0:.6f}")