**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)

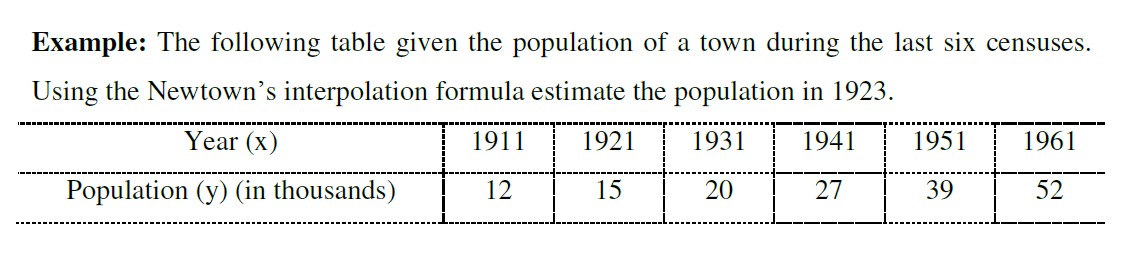
print("Iterations:", iterations)

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)

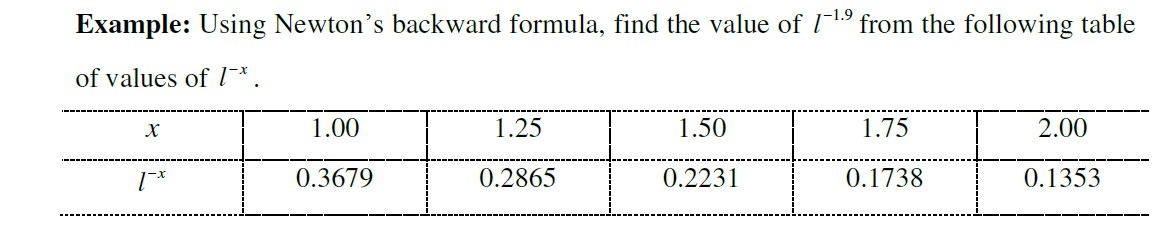
result += term

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)

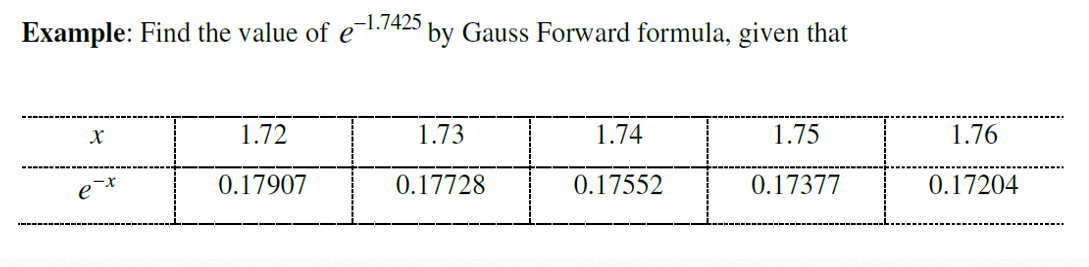
result += term

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)

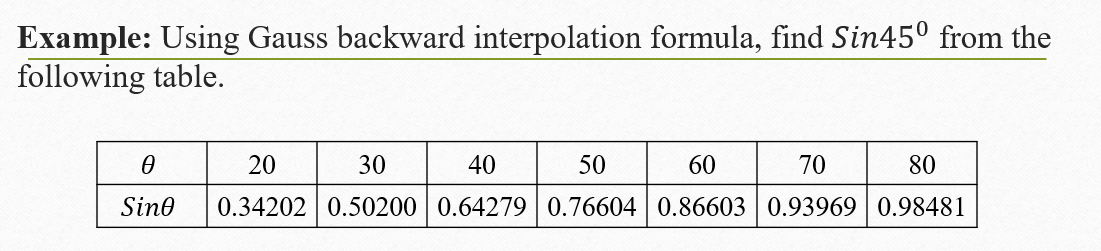
elif k == 3:

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

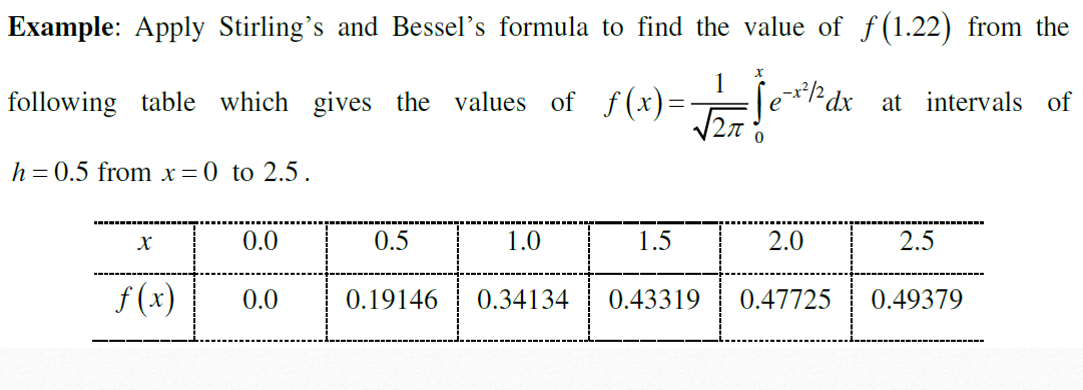
for k in range(1, 5): # up to 4th order

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}")