#### Name – Nikhil Kumar Sahu

#### Roll No- 16137

#### Sub- Numerical optimization(Practical)

#### #### 1. WAP for finding optimal solution using Line Search method.

def obj\_fun(x):

return x\*\*2 + 2\*x + 1

#derivative of obj fun

def derv(x):

return 2\*x + 2

# Define the Line Search method

def line\_search(learning\_rate,x0, num\_iterations):

x\_current = x0

iteration = 0

while iteration < num\_iterations:

gradient = derv(x\_current)

x\_current -= learning\_rate \* gradient

iteration += 1

return x\_current

# parameter passing

x0 =0.0

learning\_rate = 0.1

num\_iterations = 100

# perform line search optimz

optimal\_solution = line\_search(learning\_rate,x0, num\_iterations)

print("Optimal sol: ",optimal\_solution)

print("Obj value at Optima: ",obj\_fun(optimal\_solution))



#### 2. WAP to solve a LPP graphically.

!pip install pulp

import pulp

import matplotlib.pyplot as plt

"""This line creates a Linear Programming problem (LP) instance named lp\_problem with the objective of maximizing (LpMaximize)."""

lp\_problem = pulp.LpProblem("LPP", pulp.LpMaximize)

"""These lines define two decision variables, x and y, with lower bounds set to 0. These variables represent the values you want to find in the optimization."""

x = pulp.LpVariable("x", lowBound=0)

y = pulp.LpVariable("y", lowBound=0)

"""This line sets the objective function to be maximized. In this case, it's 3x + 2y."""

lp\_problem += 3 \* x + 2 \* y

"""These lines define the constraints of the LP. The first constraint, x <= 4, restricts x to be less than or equal to 4. The second constraint, y <= 6, restricts y to be less than or equal to 6. The third constraint, 2x + y <= 12, restricts the linear combination of x and y to be less than or equal to 12."""

lp\_problem += x <= 4

lp\_problem += y <= 6

lp\_problem += 2 \* x + y <= 12

"""This line solves the LP using the PuLP solver. It finds the optimal values for x and y that maximize the objective function within the given constraints."""

lp\_problem.solve()

"""This line prints the status of the LP solver, indicating whether it's solved successfully, infeasible, or unbounded."""

print("Status:", pulp.LpStatus[lp\_problem.status])

print("x =", x.varValue)

print("y =", y.varValue)

print("Optimal Value =", pulp.value(lp\_problem.objective))

x\_values = [x.varValue for x in [x, y]]

y\_values = [y.varValue for y in [x, y]]

print(x\_values, y\_values)

"""This line plots a red dot ('ro') at the coordinates determined by the optimal values of x and y, representing the optimal solution."""

plt.plot(x.varValue, y.varValue, 'ro', label="Optimal Value")

plt.fill([0, 4, 4, 3, 0], [0, 0, 4, 6, 6], 'b', alpha=0.2)

plt.xlabel("x")

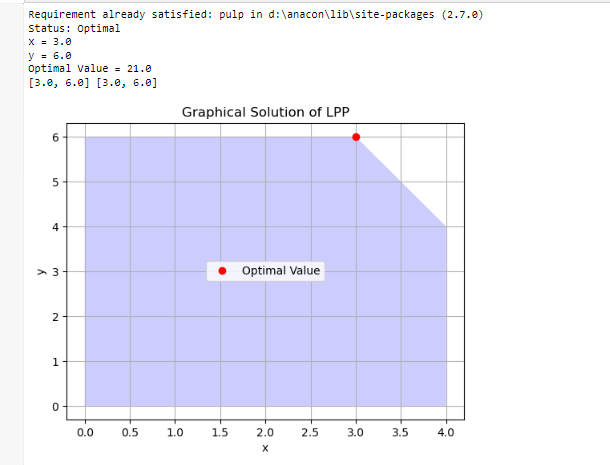
plt.ylabel("y")

plt.title("Graphical Solution of LPP")

plt.legend()

plt.grid(True)

plt.show()



#### 3. WAP to compute the gradient and Hessian of the function

𝑓(𝑥) = 100(𝑥2 − 𝑥1\*\*2)\*\*2 + (1 − 𝑥1)\*\*2

import math

import sympy

from sympy import symbols

print(math.sqrt(9))

sympy.sqrt(3)

sympy.sqrt(8)

x, y = symbols('x y')

expr = x + 2\*y

expr + 1

expr - x

x\*expr

from sympy import expand, factor

expanded\_expr = expand(x\*expr)

factor(expanded\_expr)

from sympy import \*

x, t, z, nu = symbols('x t z nu')

init\_printing(use\_unicode=True)

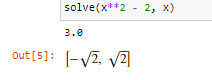
d\_ = diff(sin(x)\*exp(x), x)

d\_.subs(x, 10)

integrate(exp(x)\*sin(x) + exp(x)\*cos(x), x)

integrate(sin(x\*\*2), (x, -oo, oo))

solve(x\*\*2 - 2, x)



from sympy import symbols

# Define the variables

x1, x2 = symbols('x1 x2')

# Define the function f(x)

f = 100 \* (x2 - x1\*\*2)\*\*2 + (1 - x1)\*\*2

# Caln gradient

import sympy as sp

grad\_f = [sp.diff(f, x1), sp.diff(f, x2)]

# Cal Hessian matrix

hessian\_f = sp.hessian(f, (x1, x2))

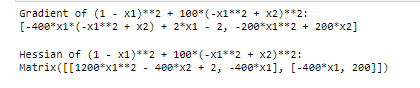
# Print the gradient and Hessian

print(f"Gradient of {f}:")

print(grad\_f)

print(f"\nHessian of {f}:")

print(hessian\_f)



#### 4. WAP to find Global Optimal Solution of a function

𝑓(𝑥) = −10𝐶𝑜𝑠(𝜋𝑥 − 2.2) + (𝑥 + 1.5)𝑥 algebraically

import numpy as np

import matplotlib.pyplot as plt

# Define the function

def f(x):

return -10 \* np.cos(np.pi \* x - 2.2) + (x + 1.5) \* x

# Generate x values

x = np.linspace(-10, 10, 1000)

# Calculate corresponding y values

y = f(x)

np.min(y)

# Find the index of the minimum value

min\_index = np.argmin(y)

# Find the global minimum

global\_min\_x = x[min\_index]

global\_min\_y = y[min\_index]

# Plot the function

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

plt.scatter(global\_min\_x, global\_min\_y, color='red', marker='o', label='Global Minimum')

plt.xlabel('x')

plt.ylabel('f(x)')

plt.legend()

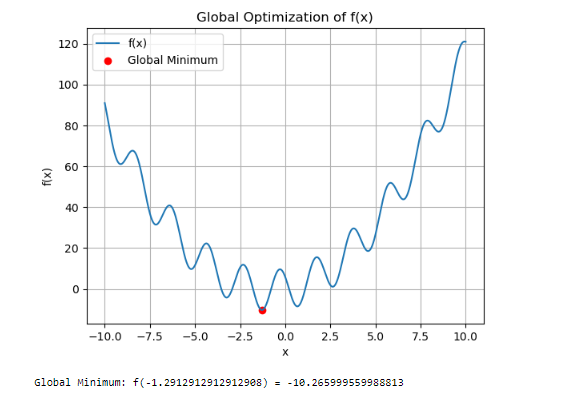
plt.title('Global Optimization of f(x)')

plt.grid(True)

plt.show()

# Display the global minimum

print(f"Global Minimum: f({global\_min\_x}) = {global\_min\_y}")



#### 5. WAP to find Global Optimal Solution of a function

𝑓(𝑥) = −10𝐶𝑜𝑠(𝜋𝑥 − 2.2) + (𝑥 + 1.5)𝑥 graphically

import numpy as np

from scipy.optimize import differential\_evolution

# Define the function to minimize

def objective\_function(x):

return -10 \* np.cos(np.pi \* x - 2.2) + (x + 1.5) \* x

# Define bounds for the optimization

bounds = [(-10, 10)] # Adjust bounds as needed

# Use differential evolution to find the global minimum

result = differential\_evolution(objective\_function, bounds)

# Extract the optimal solution

global\_min\_x = result.x

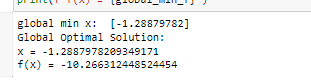
global\_min\_f = result.fun

print("global min x: ",global\_min\_x)

print("Global Optimal Solution:")

print(f"x = {global\_min\_x[0]}")

print(f"f(x) = {global\_min\_f}")



#### 6. WAP to solve constraint optimization problem.

from sympy import symbols, diff, solve, Matrix

# Define the variables

x, y, l = symbols('x y lambda')

# Define the objective function and constraint

f = x\*\*2 + y\*\*2

g = x + y - 1

# Define the Lagrangian

L = f - l \* g

# Compute partial derivatives

partials = [diff(L, var) for var in (x, y, l)]

# Solve the system of equations

solution = solve(partials, (x, y, l), dict=True)[0]

# Extract the optimal values

optimal\_x = solution[x]

optimal\_y = solution[y]

# Compute the Hessian matrix

# hessian\_matrix = Matrix([[diff(L.diff(var1), var2) for var1 in (x, y, l)] for var2 in (x, y, l)])

# Compute the Hessian matrix using a list of lists

hessian\_list = []

# Iterate over var2

for var2 in (x, y, l):

# Initialize a row for var2

row = []

# Iterate over var1

for var1 in (x, y, l):

# Calculate the second-order partial derivative and append to the row

row.append(diff(L.diff(var1), var2))

# Append the row to the Hessian list

hessian\_list.append(row)

# Create an instance of the Matrix class from the list of lists

hessian\_matrix = Matrix(hessian\_list)

# Display the Hessian matrix

print(hessian\_matrix)

# Check the definiteness of the Hessian at the stationary point

hessian\_determinant = hessian\_matrix.det()

if hessian\_determinant > 0:

print("Stationary point is a local minimum.")

elif hessian\_determinant < 0:

print("Stationary point is a local maximum.")

else:

print("Second-order test inconclusive (saddle point or test fails).")

# Display the result

print("Optimal solution:")

print(f"x: {optimal\_x}")

print(f"y: {optimal\_y}")

