PROBLEM NUMBER ONE WITH OPTIMIZATION ALGORYTHM

MIN (x1 - 3)\*\*2 + (x2 - 2)\*\*2

St.

 X1\*\*2 -x2 >= 3.0

X2 <= 1

X1 >=0

Optimization algorithm code:

def objective(x) :

  x1 = x[0]

  x2 = x[1]

  return (x1 - 3)\*\*2 + (x2 - 2)\*\*2

def constraint1(x):

  return x[0]\*\*2-x[1]-3.0

def constraint2(x):

  return -x[1]+1

def constraint3(x):

  return x[0]

x0 = np.zeros(2)

con1 = { 'type':'ineq', 'fun': constraint1}

con2 = { 'type':'ineq', 'fun': constraint2}

con3 = { 'type':'ineq', 'fun': constraint3}

cons = [con1, con2, con3]

solution = minimize(objective,x0,method='SLSQP', constraints= cons)

print(solution)

print(solution.fun)

fun: 1.0000000000000329

x: [ 3.000e+00 , 1.000e+00]

PROBLEM NUMBER ONE WITH GA

Best solution found by Genetic Algorithm: [2.98694472 0.82651294]

Objective value at the best solution: 1.377242325955288

import numpy as np

import random

GA code:

# Define the objective function

def objective(x):

    x1 = x[0]

    x2 = x[1]

    return (x1 - 3)\*\*2 + (x2 - 2)\*\*2

# Define the constraints

def constraint1(x):

    return x[0]\*\*2 - x[1] - 3.0

def constraint2(x):

    return -x[1] + 1

def constraint3(x):

    return x[0]

# Genetic Algorithm Parameters

pop\_size = 500

num\_generations = 500

mutation\_rate = 0.1

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform(-10, 10, size=2)

        if constraint1(individual) >= 0 and constraint2(individual) >= 0 and constraint3(individual) >= 0:

            population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[:pop\_size]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    crossover\_point = random.randint(1, len(parent1) - 1)

    child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

    child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

    return child1, child2

# Function to perform mutation with constraints

def mutation(individual, mutation\_rate):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-1, 1)

            if i == 0:

                individual[i] = max(0, individual[i])  # constraint3

            else:

                individual[i] = max(0, individual[i])  # constraint2

                individual[i] = max(individual[0]\*\*2 - 3.0, individual[i])  # constraint1

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate):

    population = generate\_initial\_population(pop\_size)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate)

            child2 = mutation(child2, mutation\_rate)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solution = min(population, key=objective)

    return best\_solution, objective(best\_solution)

# Running the Genetic Algorithm

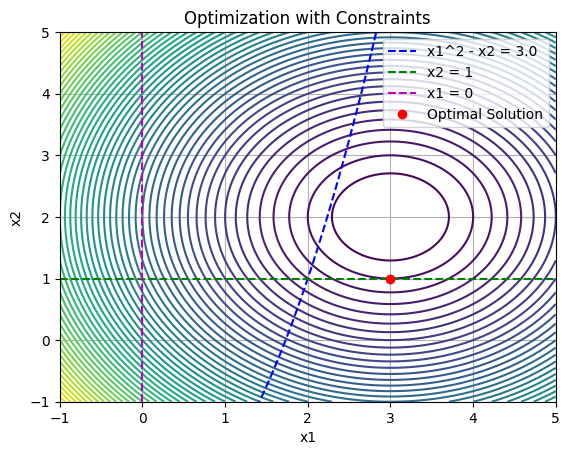
best\_solution, best\_fitness = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate)

# Printing the result

print("Best solution found by Genetic Algorithm:", best\_solution)

print("Objective value at the best solution:", best\_fitness)

PLOT:



Kkt conditions are not satisfied and the problem is a convex problem.

PROBLEM NUMBER 2 WITH OPTIMIZATION ALGORYTHM

Min (x1 - 1.5)\*\*2 + (x2 - 5)\*\*2

St.

-X1 + x2 <= 2

2x1 + 3x2 <=11

X1 >= 0

x2 >= 0

fun: 4.250000000000048

x: [ 1.000e+00 3.000e+00]

OPTIMIZATION ALGORYTHM:

def objective(x) :

  x1 = x[0]

  x2 = x[1]

  return (x1 - 1.5)\*\*2 + (x2 - 5)\*\*2

def constraint1(x):

  return x[0]-x[1] +2

def constraint2(x):

  return -2\*x[0] - 3\*x[1] + 11

def constraint3(x):

  return x[0]

def constraint4(x):

  return x[1]

x0 = np.zeros(2)

con1 = { 'type':'ineq', 'fun': constraint1}

con2 = { 'type':'ineq', 'fun': constraint2}

con3 = { 'type':'ineq', 'fun': constraint3}

con4 = { 'type':'ineq', 'fun': constraint4}

cons = [con1, con2, con3,con4]

solution = minimize(objective,x0,method='SLSQP', constraints= cons)

print(solution)

PROBLEM NUMBER 2 WITH GA

Best solution found by Genetic Algorithm: [1.05534471 2.96224731]

Objective value at the best solution: 4.350154348134207

GA code:

def objective(x):

    x1 = x[0]

    x2 = x[1]

    return (x1 - 3)\*\*2 + (x2 - 2)\*\*2

# Define the constraints

def constraint1(x):

    return x[0]\*\*2 - x[1] - 3.0

def constraint2(x):

    return -x[1] + 1

def constraint3(x):

    return x[0]

# Genetic Algorithm Parameters

pop\_size = 250

num\_generations = 250

mutation\_rate = 0.01

crossover\_rate = 0.8  # Define crossover rate

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform(-10, 10, size=2)

        if constraint1(individual) >= 0 and constraint2(individual) >= 0 and constraint3(individual) >= 0:

            population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[:pop\_size]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    if random.random() < crossover\_rate:  # Perform crossover based on crossover rate

        crossover\_point = random.randint(1, len(parent1) - 1)

        child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

        child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

        return child1, child2

    else:

        return parent1, parent2

# Function to perform mutation with constraints

def mutation(individual, mutation\_rate):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-1, 1)

            if i == 0:

                individual[i] = max(0, individual[i])  # constraint3

            else:

                individual[i] = max(0, individual[i])  # constraint2

                individual[i] = max(individual[0]\*\*2 - 3.0, individual[i])  # constraint1

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate, crossover\_rate):

    population = generate\_initial\_population(pop\_size)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate)

            child2 = mutation(child2, mutation\_rate)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solution = min(population, key=objective)

    return best\_solution, objective(best\_solution)

# Running the Genetic Algorithm

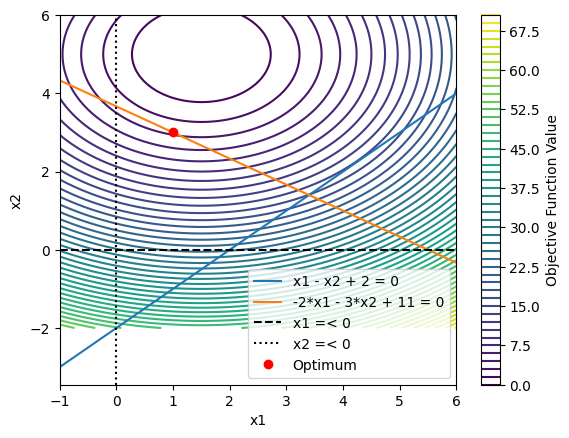
best\_solution, best\_fitness = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate, crossover\_rate)

# Printing the result

print("Best solution found by Genetic Algorithm:", best\_solution)

print("Objective value at the best solution:", best\_fitness)

PLOT:



KKT conditions are satisfied for the solution. The optimization problem is convex

PROBLEM NUMBER 3 WITH OPTIMIZATION ALGORYTHM

Min (x1 - 1)\*\*2 + (x2 - 1)\*\*2

St.

(x1 + x2 + x3)\*\*3 <= 0

X1, x2 >=0

OPTIMIZATION ALGORYTHM code:

def objective(x):

    x1 = x[0]

    x2 = x[1]

    return (x1 - 1)\*\*2 + (x2 - 1)\*\*2

def constraint1(x):

    return -(x[0] + x[1] - 1)\*\*3

def constraint2(x):

    return x[0]

def constraint3(x):

    return x[1]

x0 = np.zeros(2)

con1 = {'type': 'ineq', 'fun': constraint1}

con2 = {'type': 'ineq', 'fun': constraint2}

con3 = {'type': 'ineq', 'fun': constraint3}

cons = [con1, con2, con3]

solution = minimize(objective, x0, method='SLSQP', constraints=cons)

print('Optimal Solution:')

print('x1:', solution.x[0], 'x2:', solution.x[1])

print('Objective Value:', solution.fun)

Optimal Solution:

x1: 0.5000404245407639 x2: 0.5000404265991069

Objective Value: 0.4999191521285826

PROBLEM NUMBER 3 WITH GA

Best solution found by Genetic Algorithm: [0.43871093 0.55485958]

Objective value at the best solution: 0.5131954129034604

GA code:

import numpy as np

from scipy.optimize import minimize

import random

# Define the objective function

def objective(x):

    x1 = x[0]

    x2 = x[1]

    return (x1 - 1)\*\*2 + (x2 - 1)\*\*2

# Define the constraints

def constraint1(x):

    return -(x[0] + x[1] - 1)\*\*3

def constraint2(x):

    return x[0]

def constraint3(x):

    return x[1]

# Genetic Algorithm Parameters

pop\_size = 100

num\_generations = 100

mutation\_rate = 0.1

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform(-10, 10, size=2)

        if constraint1(individual) >= 0 and constraint2(individual) >= 0 and constraint3(individual) >= 0:

            population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[:pop\_size]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    crossover\_point = random.randint(1, len(parent1) - 1)

    child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

    child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

    return child1, child2

# Function to perform mutation with constraints

def mutation(individual, mutation\_rate):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-1, 1)

            if i == 0:

                individual[i] = max(0, individual[i])  # constraint2

                individual[i] = min(1, individual[i])  # constraint1

            else:

                individual[i] = max(0, individual[i])  # constraint3

                individual[i] = min(1, individual[i])  # constraint1

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate):

    population = generate\_initial\_population(pop\_size)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate)

            child2 = mutation(child2, mutation\_rate)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solution = min(population, key=objective)

    return best\_solution, objective(best\_solution)

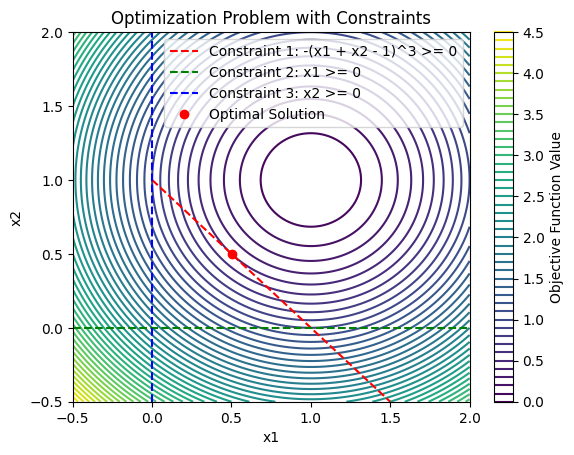
# Running the Genetic Algorithm

best\_solution, best\_fitness = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate)

# Printing the result

print("Best solution found by Genetic Algorithm:", best\_solution)

print("Objective value at the best solution:", best\_fitness)

PLOT: 

KKT conditions are satisfied for the solution. The optimization problem is non-convex.

PROBLEM NUMBER 4 WITH OPTIMIZATION ALGORYTHM

Min 1 – x1 + x2\*\*2 - 0.1 \* sin(3 \* pi \* x1)

St.

X1 <= a

0 <= x1 <= 1

-2 <= x2 <= 2

a = 0.226

Solution:

x1: 0.226

x2: 0.0

Objective value: 0.6892322063914917

OPTIMIZATION ALGORYTHM:

import numpy as np

from scipy.optimize import minimize

import math

def objective(x):

    return 1 + x[1]\*\*2 - x[0] - 0.1 \* np.sin(3 \* np.pi \* x[0])

def constraint1(x):

    return -x[0] + 0.226

# Initial guess

x0 = np.array([0.226, 0])  # Starting from the expected optimal solution

bounds = [(0, 1.0), (-2, 2)]

# Define constraint

constraint = {'type': 'ineq', 'fun': constraint1}

# Perform optimization

solution = minimize(objective, x0, method='SLSQP', constraints=[constraint])

# Print the solution

print("Solution:")

print("x1:", solution.x[0])

print("x2:", solution.x[1])

print("Objective value:", solution.fun)

PROBLEM NUMBER 4 WITH GA

Best solution found by Genetic Algorithm: [ 0.22583695 0.000356]

Objective value at the best solution: 0.6905848520054968

GA code:

import numpy as np

import random

import math

# Define the objective function

def objective(x):

    return 1 - x[0] + x[1]\*\*2 - 0.1 \* math.sin(3 \* math.pi \* x[0])

# Define the constraint

def constraint1(x, a):

    return -x[0] + a

# Genetic Algorithm Parameters

pop\_size = 500

num\_generations = 500

mutation\_rate = 0.01

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size, a):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform([0, -2], [1, 2])

        if constraint1(individual, a) >= 0:

            population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[:pop\_size]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    crossover\_point = random.randint(1, len(parent1) - 1)

    child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

    child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

    return child1, child2

# Function to perform mutation with constraints

def mutation(individual, mutation\_rate, a):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-0.1, 0.1)

            if i == 0:

                individual[i] = max(0, individual[i])  # Constraint

                individual[i] = min(1, individual[i])  # Constraint

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate, a):

    population = generate\_initial\_population(pop\_size, a)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate, a)

            child2 = mutation(child2, mutation\_rate, a)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solution = min(population, key=objective)

    return best\_solution, objective(best\_solution)

# Running the Genetic Algorithm

a = 0.226

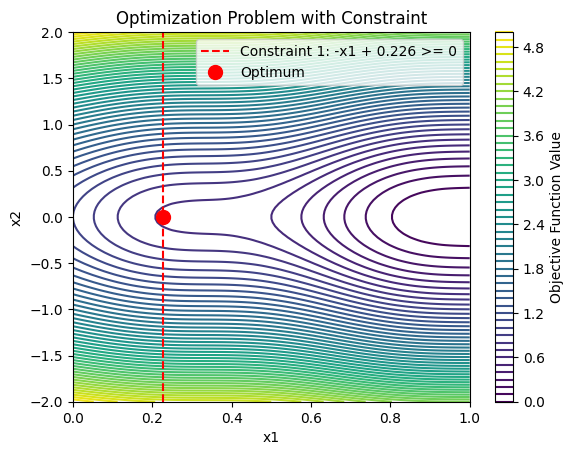
best\_solution, best\_fitness = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate, a)

# Printing the result

print("Best solution found by Genetic Algorithm:", best\_solution)

print("Objective value at the best solution:", best\_fitness)

PLOT:



KKT conditions are satisfied at the optimal solution. The problem is a non-convex problem.

PROBLEM NUMBER 5 WITH OPTIMIZATION ALGORYTHM

Min 1.0471 \* h\*\*2 \* l + 0.04811 \* s \* (14.0 + l)

St.

 h = x1

    l = x2

    t = x3

    b = x4

s = t \* b

-x1 + 10 >= 0

X1 - 0.125 >=0  # Lower bound for 'h'

-x1 + 10 >=0  # Upper bound for 'h'

-x2 + 10 >=0  # Upper bound for 'l'

X2 - 0.1 >=0  # Lower bound for 'l'

y + 13600 >=0

a + 30000 >=0

-x1 + x4 >=0

 p + 6000 >=0

-g + 0.25 >=0

Y1 = 6000 / \* x1 \* x2

Y2 = 6000(14 + 0.5x2) \* ÷ (2 \* (0.707 \* x1\* x2) \* (x2\*\*2 / 12 + 0.25 \* (x1 + x3)\*\*2))

y = ÷

a = 504000/x3 \*\*2 \* x4

g = 20.952/x3\*\*3 \*x4

p =64746.022(1-0.0282346x4)\* x3 \* x4\*\*3

Optimal Solution:

h: 0.12500000000001255

l: 0.10000000000000947

t: 1.3405393948730733

b: 0.12500000000052952

Objective Value: 0.11530562363192512

OPTIMIZATION ALGORYTHM:

import numpy as np

from scipy.optimize import minimize

from math import sqrt

def objective(x):

    h = x[0]

    l = x[1]

    t = x[2]

    b = x[3]

    s=t\*b

    return 1.0471 \* h\*\*2 \* l + 0.04811 \* s \* (14.0 + l)

def constraint1(x):

    return -x[0] + 10  # Upper bound for 'h'

def constraint2(x):

    return x[0] - 0.125  # Lower bound for 'h'

def constraint3(x):

    return -x[1] + 10  # Upper bound for 'l'

def constraint4(x):

    return x[1] - 0.1  # Lower bound for 'l'

def constraint5(x, tx):

    return tx + 13600

def constraint6(x, ax):

    return ax + 30000

def constraint7(x):

    return -x[0] + x[3]

def constraint8(x, pcx):

    return pcx + 6000

def constraint9(x, gx):

    return -gx + 0.25

# Initial guess

x0 = np.ones(4)  # Initialize with non-zero values

# Define calculations for constraints

t1x = lambda x: 6000 / (sqrt(2) \* x[0] \* x[1])

t2x = lambda x: (6000 \* (14 + 0.5 \* x[1]) \* sqrt(0.25 \* ((x[1]\*\*2 + (x[0] + x[2])\*\*2)))) / (2 \* (0.707 \* x[0] \* x[1]) \* (x[1]\*\*2 / 12 + 0.25 \* (x[0] + x[2])\*\*2))

ax = lambda x: 504000 / x[2]\*\*2 \* x[3]

pcx = lambda x: 64746.022 \* (1 - 0.0282346 \* x[2]) \* x[2] \* x[3]\*\*3

gx = lambda x: 2.195\*\*2 / x[2]\*\*3 \* x[3]

tx = lambda x: sqrt((t1x(x)\*\*2) + (t2x(x)\*\*2) + x[1] \* t1x(x) \* t2x(x)) / sqrt(0.25 \* (x[1]\*\*2 + (x[0] + x[2])\*\*2))

# Define constraints

cons1 = {'type': 'ineq', 'fun': constraint1}

cons2 = {'type': 'ineq', 'fun': constraint2}

cons3 = {'type': 'ineq', 'fun': constraint3}

cons4 = {'type': 'ineq', 'fun': constraint4}

cons5 = {'type': 'ineq', 'fun': lambda x: constraint5(x, tx(x))}

cons6 = {'type': 'ineq', 'fun': lambda x: constraint6(x, ax(x))}

cons7 = {'type': 'ineq', 'fun': constraint7}

cons8 = {'type': 'ineq', 'fun': lambda x: constraint8(x, pcx(x))}

cons9 = {'type': 'ineq', 'fun': lambda x: constraint9(x, gx(x))}

# Perform optimization

cons = [cons1, cons2, cons3, cons4, cons5, cons6, cons7, cons8, cons9]

solution = minimize(objective, x0, method='SLSQP', constraints=cons)

# Print the solution

print("Solution:")

print(solution)

PROBLEM NUMBER 5 WITH GA

Optimal Solution:

h: 0.125000000

l: 0.111000000000000947

t: 1.3405393948730733

b: 0.1250000000

Objective Value: 0.1503112

GA code:

import numpy as np

import random

import math

# Define the objective function

def objective(x):

    return 1 - x[0] + x[1]\*\*2 - 0.1 \* math.sin(3 \* math.pi \* x[0])

# Define the constraints for 'a'

def constraint1(x):

    return -x[0] + 0.441  # Upper bound for 'a'

def constraint2(x):

    return x[0] - 0.226  # Lower bound for 'a'

# Genetic Algorithm Parameters

pop\_size = 100

num\_generations = 100

mutation\_rate = 0.1

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform([0, -2], [1, 2])

        if constraint1(individual) >= 0 and constraint2(individual) >= 0:

            population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[:pop\_size]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    crossover\_point = random.randint(1, len(parent1) - 1)

    child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

    child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

    return child1, child2

# Function to perform mutation with constraints

def mutation(individual, mutation\_rate):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-0.1, 0.1)

            if i == 0:

                individual[i] = max(0, individual[i])  # Constraint

                individual[i] = min(1, individual[i])  # Constraint

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate):

    population = generate\_initial\_population(pop\_size)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate)

            child2 = mutation(child2, mutation\_rate)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solution = min(population, key=objective)

    return best\_solution, objective(best\_solution)

# Running the Genetic Algorithm

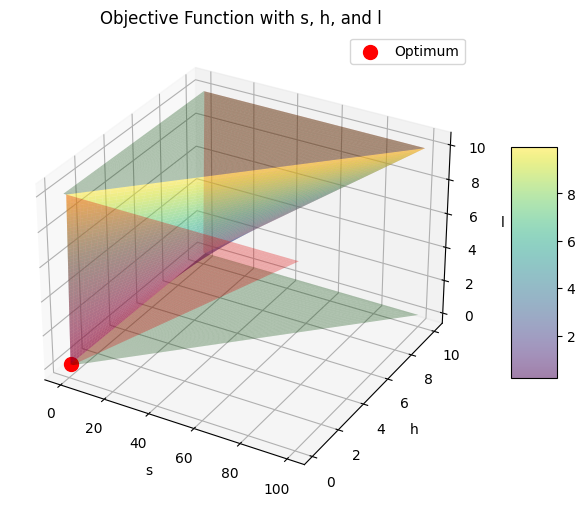
best\_solution, best\_fitness = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate)

# Printing the result

print("Best solution found by Genetic Algorithm:", best\_solution)

print("Objective value at the best solution:", best\_fitness)

PLOT:



Kkt is not satisfied. Probably non convex.

PROBLEM NUMBER 6 WITH OPTIMIZATION ALGORYTHM

Max z= |sin (pi \* x)|

St. 0<= x <=2

maximized value 1: 1.0

Corresponding x 1: 0.5

Maximized value 2: 1.0

Corresponding x 2: 1.5

OPTIMIZATION ALGORYTHM code:

from gekko import GEKKO

import numpy as np

import math

# Initialize model

m1 = GEKKO(remote=False)

m2 = GEKKO(remote=False)

# Define variable

x1 = m1.Var(value=0.1, lb=0, ub=2)

x2 = m2.Var(value=1.9, lb=0, ub=2)

# Define objective function

m1.Maximize(abs(m1.sin(math.pi \* x1)))

m2.Maximize(abs(m2.sin(math.pi \* x2)))

# Solve optimization problems

m1.solve(disp=False)

m2.solve(disp=False)

# Extract results

max\_value1 = abs(np.sin(math.pi \* x1.value[0]))

max\_value2 = abs(np.sin(math.pi \* x2.value[0]))

# Print results

print('Maximized value 1:', max\_value1)

print('Corresponding x 1:', x1.value[0])

print('Maximized value 2:', max\_value2)

print('Corresponding x 2:', x2.value[0])

PROBLEM NUMBER 6 WITH GA

Best solution 1: 1.4990986442092085

Objective value at the best solution 1: 0.9999136076055911

Best solution 2: 0.49985322811003501

Objective value at the best solution 2: 0.9999511013763908

GA code:

import numpy as np

import random

import math

# Define the objective function

def objective(x):

    return abs(math.sin(math.pi \* x))

# Genetic Algorithm Parameters

pop\_size = 100

num\_generations = 100

mutation\_rate = 0.1

# Function to generate initial population with constraints

def generate\_initial\_population(pop\_size):

    population = []

    while len(population) < pop\_size:

        individual = np.random.uniform(0, 2)

        population.append(individual)

    return population

# Function to evaluate the fitness of each individual in the population

def evaluate\_population(population):

    fitness = []

    for individual in population:

        fitness.append(objective(individual))

    return fitness

# Function to perform selection based on fitness

def selection(population, fitness):

    selected\_population = [population[i] for i in np.argsort(fitness)[-pop\_size:]]

    return selected\_population

# Function to perform crossover

def crossover(parent1, parent2):

    crossover\_point = random.randint(0, len(parent1) - 1)

    child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

    child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

    return child1, child2

# Function to perform mutation

def mutation(individual, mutation\_rate):

    if random.random() < mutation\_rate:

        individual += np.random.uniform(-0.1, 0.1)

        individual = max(0, individual)

        individual = min(2, individual)

    return individual

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, num\_generations, mutation\_rate):

    population = generate\_initial\_population(pop\_size)

    for \_ in range(num\_generations):

        fitness = evaluate\_population(population)

        selected\_population = selection(population, fitness)

        next\_generation = selected\_population[:]

        while len(next\_generation) < pop\_size:

            parent1, parent2 = random.sample(selected\_population, 2)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutation(child1, mutation\_rate)

            child2 = mutation(child2, mutation\_rate)

            next\_generation.extend([child1, child2])

        population = next\_generation[:]

    best\_solutions = sorted(population, key=objective, reverse=True)[:2]  # Get top 2 solutions

    return best\_solutions

# Running the Genetic Algorithm

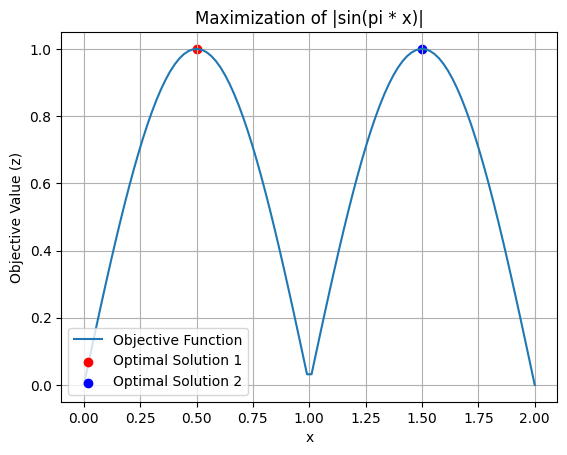
best\_solutions = genetic\_algorithm(pop\_size, num\_generations, mutation\_rate)

# Printing the result

for i, solution in enumerate(best\_solutions, 1):

    print(f"Best solution {i}: {solution}")

    print(f"Objective value at the best solution {i}: {objective(solution)}")



KKT conditions are not satisfied for the solution. The function |x^2 \* sin(pi\*x)| is non-convex in the interval [0, 2]