**Lab-1**-Genetic Algorithm for Optimization Problems

**Code:**

#1

import random

def fitness\_function(x):

return x\*\*2

population\_size = 500

mutation\_rate = 0.01

crossover\_rate = 0.7

generations = 5

lower\_bound = -10

upper\_bound = 10

def create\_population():

population = [random.uniform(lower\_bound, upper\_bound) for \_ in range(population\_size)]

return population

def evaluate\_fitness(population):

fitness\_scores = [fitness\_function(individual) for individual in population]

return fitness\_scores

def select(population, fitness\_scores):

total\_fitness = sum(fitness\_scores)

selection\_probs = [score / total\_fitness for score in fitness\_scores]

selected = random.choices(population, weights=selection\_probs, k=2)

return selected

def crossover(parent1, parent2):

if random.random() < crossover\_rate:

# Crossover at a single point (since we're dealing with floats, we'll use averaging)

child1 = (parent1 + parent2) / 2

child2 = (parent1 + parent2) / 2

else:

child1, child2 = parent1, parent2

return child1, child2

def mutate(individual):

if random.random() < mutation\_rate:

mutation\_value = random.uniform(-1, 1)

individual += mutation\_value

return max(min(individual, upper\_bound), lower\_bound)

def genetic\_algorithm():

population = create\_population()

for generation in range(generations):

fitness\_scores = evaluate\_fitness(population)

best\_fitness = max(fitness\_scores)

best\_individual = population[fitness\_scores.index(best\_fitness)]

new\_population = []

while len(new\_population) < population\_size:

parent1, parent2 = select(population, fitness\_scores)

child1, child2 = crossover(parent1, parent2)

child1 = mutate(child1)

child2 = mutate(child2)

new\_population.append(child1)

if len(new\_population) < population\_size:

new\_population.append(child2)

population = new\_population

print(f"Generation {generation + 1}: Best Fitness = {best\_fitness}, Best Individual = {best\_individual}")

return best\_individual, best\_fitness

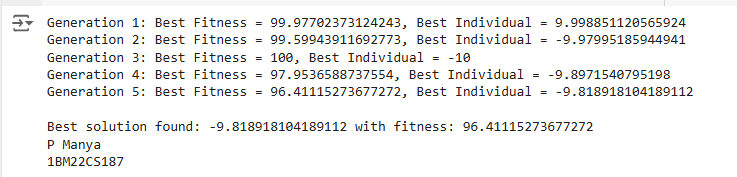
best\_solution, best\_fitness = genetic\_algorithm()

print(f"\nBest solution found: {best\_solution} with fitness: {best\_fitness}")

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-2**-Particle Swarm Optimization for Function Optimization

**Code:**

#2

import numpy as np

# Define the Rastrigin function

def rastrigin(x):

A = 10

return A \* len(x) + sum([xi\*\*2 - A \* np.cos(2 \* np.pi \* xi) for xi in x])

class Particle:

def \_\_init\_\_(self, dim):

self.position = np.random.uniform(-5.12, 5.12, dim)

self.velocity = np.random.uniform(-1, 1, dim)

self.best\_position = np.copy(self.position)

self.best\_value = rastrigin(self.position)

def update\_velocity(self, global\_best\_position, inertia\_weight, cognitive\_coef, social\_coef):

r1, r2 = np.random.rand(2)

cognitive\_velocity = cognitive\_coef \* r1 \* (self.best\_position - self.position)

social\_velocity = social\_coef \* r2 \* (global\_best\_position - self.position)

self.velocity = inertia\_weight \* self.velocity + cognitive\_velocity + social\_velocity

def update\_position(self):

self.position += self.velocity

# Keep the particle within the bounds

self.position = np.clip(self.position, -5.12, 5.12)

# Update the best position if necessary

current\_value = rastrigin(self.position)

if current\_value < self.best\_value:

self.best\_value = current\_value

self.best\_position = np.copy(self.position)

def pso(num\_particles, dim, num\_iterations):

inertia\_weight = 0.7

cognitive\_coef = 1.5

social\_coef = 1.5

# Initialize particles

particles = [Particle(dim) for \_ in range(num\_particles)]

global\_best\_position = particles[0].best\_position

global\_best\_value = particles[0].best\_value

# Main PSO loop

for \_ in range(num\_iterations):

for particle in particles:

particle.update\_velocity(global\_best\_position, inertia\_weight, cognitive\_coef, social\_coef)

particle.update\_position()

# Update global best

if particle.best\_value < global\_best\_value:

global\_best\_value = particle.best\_value

global\_best\_position = particle.best\_position

return global\_best\_position, global\_best\_value

# User input for parameters

num\_particles = int(input("Enter the number of particles: "))

dim = int(input("Enter the number of dimensions: "))

num\_iterations = int(input("Enter the number of iterations: "))

# Run PSO

best\_position, best\_value = pso(num\_particles, dim, num\_iterations)

# Output the best position and value

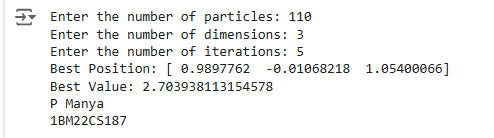
print(f"Best Position: {best\_position}")

print(f"Best Value: {best\_value}")

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-3**-Ant Colony Optimization for the Traveling Salesman Problem

**Code:**

#3

import numpy as np

import random

class AntColony:

def \_\_init\_\_(self, cities, num\_ants, alpha, beta, rho, q0, iterations):

self.cities = cities

self.num\_ants = num\_ants

self.alpha = alpha

self.beta = beta

self.rho = rho # pheromone evaporation rate

self.q0 = q0 # exploration vs. exploitation parameter

self.iterations = iterations

self.distance\_matrix = self.calculate\_distance\_matrix()

self.pheromone = np.ones(self.distance\_matrix.shape) / len(cities)

def calculate\_distance\_matrix(self):

num\_cities = len(self.cities)

distance\_matrix = np.zeros((num\_cities, num\_cities))

for i in range(num\_cities):

for j in range(num\_cities):

distance\_matrix[i][j] = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))

return distance\_matrix

def select\_next\_city(self, current\_city, visited):

probabilities = []

for next\_city in range(len(self.cities)):

if next\_city not in visited:

pheromone = self.pheromone[current\_city][next\_city] \*\* self.alpha

heuristic = (1 / self.distance\_matrix[current\_city][next\_city]) \*\* self.beta

probabilities.append(pheromone \* heuristic)

else:

probabilities.append(0)

probabilities = np.array(probabilities)

probabilities /= probabilities.sum() # Normalize

return np.random.choice(range(len(self.cities)), p=probabilities)

def construct\_solution(self):

for \_ in range(self.num\_ants):

visited = [0]

current\_city = 0

for \_ in range(len(self.cities) - 1):

current\_city = self.select\_next\_city(current\_city, visited)

visited.append(current\_city)

visited.append(0) # Return to starting city

yield visited

def update\_pheromones(self, solutions):

self.pheromone \*= (1 - self.rho) # Evaporation

for solution in solutions:

distance = self.calculate\_tour\_length(solution)

for i in range(len(solution) - 1):

self.pheromone[solution[i]][solution[i + 1]] += 1 / distance

def calculate\_tour\_length(self, tour):

return sum(self.distance\_matrix[tour[i]][tour[i + 1]] for i in range(len(tour) - 1))

def run(self):

best\_solution = None

best\_length = float('inf')

for \_ in range(self.iterations):

solutions = list(self.construct\_solution())

self.update\_pheromones(solutions)

for solution in solutions:

length = self.calculate\_tour\_length(solution)

if length < best\_length:

best\_length = length

best\_solution = solution

return best\_solution, best\_length

def main():

# User input for cities

num\_cities = int(input("Enter the number of cities: "))

cities = []

for i in range(num\_cities):

x, y = map(float, input(f"Enter coordinates for city {i + 1} (x y): ").split())

cities.append((x, y))

# Parameters for ACO

num\_ants = int(input("Enter the number of ants: "))

alpha = float(input("Enter the importance of pheromone (alpha): ")) # Importance of pheromone

beta = float(input("Enter the importance of heuristic (beta): "))

rho = float(input("Enter the pheromone evaporation rate (rho): "))

q0 = float(input("Enter the exploration parameter (q0): "))

iterations = int(input("Enter the number of iterations: "))

# Run the ACO algorithm

aco = AntColony(cities, num\_ants, alpha, beta, rho, q0, iterations)

best\_solution, best\_length = aco.run()

print("Best solution:", best\_solution)

print("Best tour length:", best\_length)

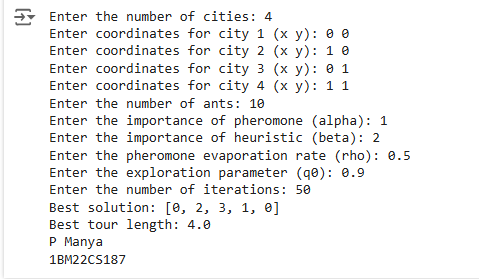
if \_\_name\_\_ == "\_\_main\_\_":

main()

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-4**-Cuckoo Search (CS)

**Code:**

#4

import numpy as np

import random

# Step 1: Define the Problem (Optimization Function)

def objective\_function(x):

# Example: Sphere function (minimization problem)

return sum(x\*\*2)

# Step 2: Initialize Parameters

def cuckoo\_search(num\_nests, max\_iter, pa, dim, lower\_bound, upper\_bound):

# Initialize nests randomly within the bounds

nests = np.random.uniform(low=lower\_bound, high=upper\_bound, size=(num\_nests, dim))

# Evaluate fitness of each nest

fitness = np.apply\_along\_axis(objective\_function, 1, nests)

# Track the best solution found

best\_fitness = np.min(fitness)

best\_nest = nests[np.argmin(fitness)]

# Step 3: Iterate the process

for iter in range(max\_iter):

# Generate new solutions using Lévy flights

new\_nests = np.copy(nests)

for i in range(num\_nests):

# Lévy flight (random walk)

step\_size = np.random.normal(0, 1, dim) \* (np.abs(np.random.normal(0, 1, dim)) \*\* (1/2))

new\_nests[i] = nests[i] + step\_size

# Boundary check (Keep the nest within bounds)

new\_nests[i] = np.clip(new\_nests[i], lower\_bound, upper\_bound)

# Evaluate fitness of new nests

new\_fitness = np.apply\_along\_axis(objective\_function, 1, new\_nests)

# Step 4: Abandon Worst Nests and Replace

for i in range(num\_nests):

if new\_fitness[i] < fitness[i]: # If new nest is better

nests[i] = new\_nests[i]

fitness[i] = new\_fitness[i]

# Discovering worst nests and abandon them with probability pa

if random.random() < pa:

abandon\_indices = np.argsort(fitness)[:int(num\_nests \* 0.25)] # abandon worst 25%

nests[abandon\_indices] = np.random.uniform(low=lower\_bound, high=upper\_bound,

size=(len(abandon\_indices), dim))

fitness[abandon\_indices] = np.apply\_along\_axis(objective\_function, 1,

nests[abandon\_indices])

# Track best solution so far

current\_best\_fitness = np.min(fitness)

current\_best\_nest = nests[np.argmin(fitness)]

if current\_best\_fitness < best\_fitness:

best\_fitness = current\_best\_fitness

best\_nest = current\_best\_nest

# Output the current best solution (optional)

print(f"Iteration {iter + 1}/{max\_iter}, Best Fitness: {best\_fitness}")

return best\_nest, best\_fitness

# Step 5: User Input for Parameters

if \_\_name\_\_ == "\_\_main\_\_":

# User input for parameters

num\_nests = int(input("Enter number of nests: "))

max\_iter = int(input("Enter number of iterations: "))

pa = float(input("Enter probability of discovery (pa) [0, 1]: "))

dim = int(input("Enter the number of dimensions: "))

lower\_bound = float(input("Enter lower bound for search space: "))

upper\_bound = float(input("Enter upper bound for search space: "))

# Run the Cuckoo Search Algorithm

best\_solution, best\_fitness = cuckoo\_search(num\_nests, max\_iter, pa, dim, lower\_bound, upper\_bound)

# Output the best solution found

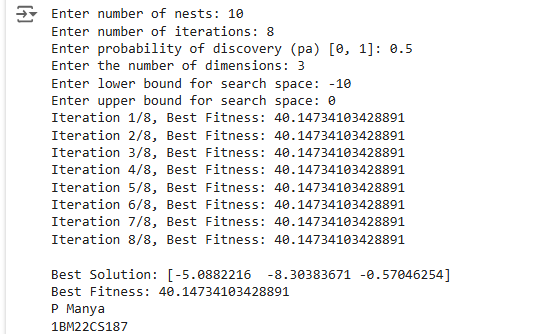
print(f"\nBest Solution: {best\_solution}")

print(f"Best Fitness: {best\_fitness}")

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-5**-Grey Wolf Optimizer (GWO

**Code:**

#5

import numpy as np

# Define the Grey Wolf Optimizer (GWO) algorithm

class GreyWolfOptimizer:

def \_\_init\_\_(self, func, dim, num\_wolves, max\_iter, lb, ub):

"""

:param func: The objective function to optimize (should take a numpy array as input)

:param dim: The dimension of the search space

:param num\_wolves: The number of wolves (population size)

:param max\_iter: The maximum number of iterations

:param lb: Lower bound of the search space

:param ub: Upper bound of the search space

"""

self.func = func

self.dim = dim

self.num\_wolves = num\_wolves

self.max\_iter = max\_iter

self.lb = lb

self.ub = ub

self.alpha\_pos = np.zeros(dim)

self.beta\_pos = np.zeros(dim)

self.delta\_pos = np.zeros(dim)

self.alpha\_score = float("inf")

self.beta\_score = float("inf")

self.delta\_score = float("inf")

# Initialize the positions of the wolves randomly

self.positions = np.random.rand(self.num\_wolves, self.dim) \* (self.ub - self.lb) + self.lb

def fitness(self, position):

"""Calculate the fitness value of a wolf"""

return self.func(position)

def update\_position(self, wolf, alpha\_pos, beta\_pos, delta\_pos, a, A, C):

"""Update the position of a wolf based on the positions of alpha, beta, and delta wolves"""

# Update for alpha, beta, and delta wolves, using the respective A and C values

D\_alpha = np.abs(C[0] \* alpha\_pos - wolf)

D\_beta = np.abs(C[1] \* beta\_pos - wolf)

D\_delta = np.abs(C[2] \* delta\_pos - wolf)

# Updated positions

X1 = alpha\_pos - A[0] \* D\_alpha

X2 = beta\_pos - A[1] \* D\_beta

X3 = delta\_pos - A[2] \* D\_delta

# New position is the average of the three updated positions

new\_position = (X1 + X2 + X3) / 3

return new\_position

def optimize(self):

"""Run the optimization process"""

for t in range(self.max\_iter):

a = 2 - t \* (2 / self.max\_iter) # Declining a over iterations

A = 2 \* a \* np.random.rand(self.num\_wolves, self.dim) - a # Random vector for wolves' A values

C = 2 \* np.random.rand(self.num\_wolves, self.dim) # Random vector for wolves' C values

# Loop over each wolf

for i in range(self.num\_wolves):

# Evaluate fitness of each wolf

fitness\_value = self.fitness(self.positions[i])

# Update alpha, beta, and delta wolves

if fitness\_value < self.alpha\_score:

self.alpha\_score = fitness\_value

self.alpha\_pos = self.positions[i]

elif fitness\_value < self.beta\_score:

self.beta\_score = fitness\_value

self.beta\_pos = self.positions[i]

elif fitness\_value < self.delta\_score:

self.delta\_score = fitness\_value

self.delta\_pos = self.positions[i]

# Update the positions of all wolves

for i in range(self.num\_wolves):

self.positions[i] = self.update\_position(self.positions[i], self.alpha\_pos, self.beta\_pos,

self.delta\_pos, a, A[i], C[i])

# Optionally, print progress

if t % 1 == 0: # Adjust the interval for printing

print(f"Iteration {t}/{self.max\_iter}, Best fitness: {self.alpha\_score}")

return self.alpha\_pos, self.alpha\_score

# Example usage:

def objective\_function(x):

"""A simple objective function: Sphere function"""

return np.sum(x \*\* 2)

# User input for parameters

dim = int(input("Enter the number of dimensions: "))

num\_wolves = int(input("Enter the number of wolves: "))

max\_iter = int(input("Enter the maximum number of iterations: "))

lb = float(input("Enter the lower bound of the search space: "))

ub = float(input("Enter the upper bound of the search space: "))

# Initialize and run the GWO algorithm

optimizer = GreyWolfOptimizer(func=objective\_function, dim=dim, num\_wolves=num\_wolves,

max\_iter=max\_iter, lb=lb, ub=ub)

best\_position, best\_score = optimizer.optimize()

# Output the best solution found

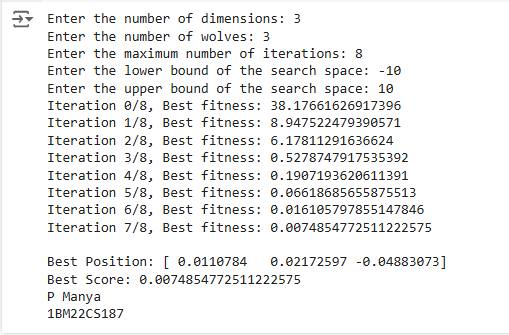
print(f"\nBest Position: {best\_position}")

print(f"Best Score: {best\_score}")

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-6**-Parallel Cellular Algorithms and Programs

**Code:**

#6

import numpy as np

import random

import math

import concurrent.futures

# Define the Rastrigin function (objective function to optimize)

def rastrigin\_function(x):

A = 10

return A \* len(x) + sum([(xi \*\* 2 - A \* math.cos(2 \* math.pi \* xi)) for xi in x])

# Initialize the population of cells (solutions) randomly

def initialize\_population(num\_cells, grid\_size):

population = []

for \_ in range(num\_cells):

cell = np.random.uniform(-5.12, 5.12, grid\_size) # Rastrigin function bounds [-5.12, 5.12]

population.append(cell)

return population

# Evaluate the fitness of a cell (solution)

def evaluate\_fitness(cell):

return rastrigin\_function(cell)

# Function to update the state of each cell based on its neighbors

def update\_cell\_state(cell, neighbors, grid\_size):

# For simplicity, we'll use an average of the neighbors' states (here it's just a random update)

best\_neighbor = min(neighbors, key=lambda n: evaluate\_fitness(n))

# Move towards the best neighbor in the fitness landscape

new\_cell = cell + 0.1 \* (best\_neighbor - cell)

return np.clip(new\_cell, -5.12, 5.12) # Clamping to the function's bounds

# Parallelized function to perform the main update step for each cell

def update\_population(population, grid\_size):

updated\_population = []

# Parallelize the update step for all cells in the population

with concurrent.futures.ThreadPoolExecutor() as executor:

future\_to\_cell = {}

for i in range(len(population)):

# Get the neighbors (simple example: using adjacent cells)

neighbors = population[max(i - 1, 0):min(i + 2, len(population))]

future\_to\_cell[executor.submit(update\_cell\_state, population[i], neighbors, grid\_size)] = population[i]

# Gather the results

for future in concurrent.futures.as\_completed(future\_to\_cell):

updated\_population.append(future.result())

return updated\_population

# Main function to perform the optimization

def parallel\_cellular\_algorithm(num\_cells, grid\_size, num\_iterations):

# Step 1: Initialize Population

population = initialize\_population(num\_cells, grid\_size)

best\_solution = None

best\_fitness = float('inf')

# Step 2: Main Loop (Iterate for a fixed number of iterations)

for iteration in range(num\_iterations):

# Step 3: Evaluate fitness of all cells

fitness\_scores = [evaluate\_fitness(cell) for cell in population]

# Track the best solution

min\_fitness = min(fitness\_scores)

if min\_fitness < best\_fitness:

best\_fitness = min\_fitness

best\_solution = population[fitness\_scores.index(min\_fitness)]

print(f"Iteration {iteration + 1}/{num\_iterations}, Best Fitness: {best\_fitness}")

# Step 4: Update population based on neighbors

population = update\_population(population, grid\_size)

# Output the best solution found

return best\_solution, best\_fitness

# Main function to handle user input and execution

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input for parameters

num\_cells = int(input("Enter the number of cells: "))

grid\_size = int(input("Enter the grid size (dimension of each solution): "))

num\_iterations = int(input("Enter the number of iterations: "))

# Run the Parallel Cellular Algorithm

best\_solution, best\_fitness = parallel\_cellular\_algorithm(num\_cells, grid\_size, num\_iterations)

# Output the best solution found

print("\nOptimization complete!")

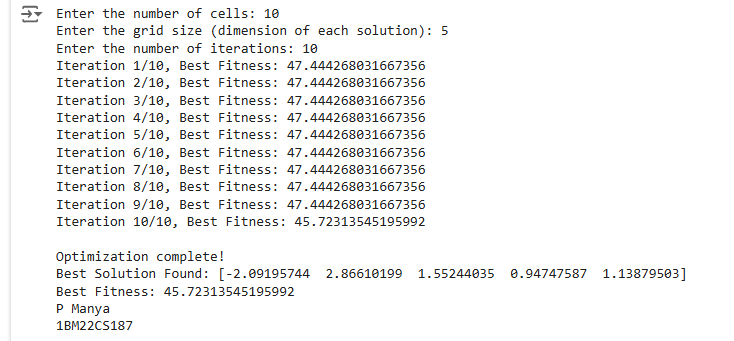
print("Best Solution Found:", best\_solution)

print("Best Fitness:", best\_fitness)

print("P Manya")

print("1BM22CS187")

**Output:**

****

**Lab-7**-Optimization via Gene Expression Algorithms

**Code:**

#7

import numpy as np

# Step 1: Define the Problem

# Define your mathematical optimization function here

# Example: Minimize the function f(x) = x^2 - 4x + 4 (a simple quadratic function)

def objective\_function(x):

return x\*\*2 - 4\*x + 4

# Step 2: Initialize Parameters

def initialize\_parameters():

population\_size = int(input("Enter population size: ")) # Number of individuals in the population

num\_genes = int(input("Enter number of genes: ")) # Number of genes in each individual

mutation\_rate = float(input("Enter mutation rate (0 to 1): ")) # Probability of mutation

crossover\_rate = float(input("Enter crossover rate (0 to 1): ")) # Probability of crossover

num\_generations = int(input("Enter number of generations: ")) # Number of generations

return population\_size, num\_genes, mutation\_rate, crossover\_rate, num\_generations

# Step 3: Initialize Population

def initialize\_population(population\_size, num\_genes):

population = np.random.uniform(low=-5, high=5, size=(population\_size, num\_genes))

return population

# Step 4: Evaluate Fitness

def evaluate\_fitness(population, objective\_function):

fitness = np.apply\_along\_axis(objective\_function, 1, population) # Apply the objective function to each individual

return fitness

# Step 5: Selection (Tournament Selection)

def selection(population, fitness, num\_parents):

parents = []

for \_ in range(num\_parents):

tournament = np.random.choice(population.shape[0], size=3, replace=False) # Select 3 individuals for tournament

tournament\_fitness = fitness[tournament]

winner = tournament[np.argmin(tournament\_fitness)] # Select the individual with the best fitness

parents.append(population[winner])

return np.array(parents)

# Step 6: Crossover (Single-point crossover)

def crossover(parents, crossover\_rate):

num\_parents = parents.shape[0]

offspring = []

for i in range(0, num\_parents, 2):

if np.random.rand() < crossover\_rate:

# Ensure that we have more than 1 gene to perform crossover

if parents.shape[1] > 1:

crossover\_point = np.random.randint(1, parents.shape[1])

offspring1 = np.concatenate([parents[i, :crossover\_point], parents[i+1, crossover\_point:]])

offspring2 = np.concatenate([parents[i+1, :crossover\_point], parents[i, crossover\_point:]])

offspring.append(offspring1)

offspring.append(offspring2)

else:

# No crossover if there's only 1 gene

offspring.append(parents[i])

if i + 1 < num\_parents:

offspring.append(parents[i + 1])

else:

offspring.append(parents[i])

if i + 1 < num\_parents:

offspring.append(parents[i + 1])

return np.array(offspring)

# Step 7: Mutation

def mutation(offspring, mutation\_rate):

for i in range(offspring.shape[0]):

for j in range(offspring.shape[1]):

if np.random.rand() < mutation\_rate:

offspring[i, j] += np.random.normal(0, 0.1) # Apply Gaussian mutation

return offspring

# Step 8: Gene Expression (Translate Genes into Solutions)

def gene\_expression(offspring):

# In this case, the genes are directly the solutions

return offspring

# Step 9: Iterate (Repeat the selection, crossover, mutation, and gene expression processes)

def run\_ge\_algorithm(objective\_function):

population\_size, num\_genes, mutation\_rate, crossover\_rate, num\_generations = initialize\_parameters()

# Initialize population

population = initialize\_population(population\_size, num\_genes)

# Start optimization process

best\_solution = None

best\_fitness = float('inf')

for generation in range(num\_generations):

# Evaluate fitness

fitness = evaluate\_fitness(population, objective\_function)

# Track the best solution

min\_fitness\_idx = np.argmin(fitness)

if fitness[min\_fitness\_idx] < best\_fitness:

best\_fitness = fitness[min\_fitness\_idx]

best\_solution = population[min\_fitness\_idx]

# Selection

parents = selection(population, fitness, population\_size // 2)

# Crossover

offspring = crossover(parents, crossover\_rate)

# Mutation

offspring = mutation(offspring, mutation\_rate)

# Gene Expression (Directly apply the offspring as solutions)

population = gene\_expression(offspring)

# Print the progress

print(f"Generation {generation + 1}: Best Fitness = {best\_fitness}")

return best\_solution, best\_fitness

# Step 10: Output the Best Solution

best\_solution, best\_fitness = run\_ge\_algorithm(objective\_function)

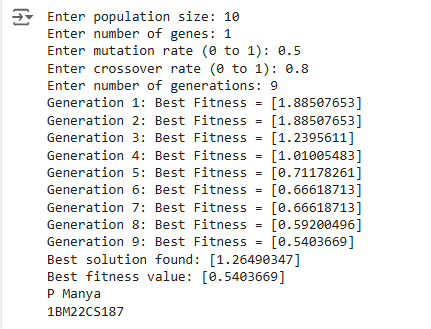
print(f"Best solution found: {best\_solution}")

print(f"Best fitness value: {best\_fitness}")

print("P Manya")

print("1BM22CS187")

**Output:**

****