import numpy as np

# Define the optimization problem (e.g., f(x, y) = x^2 + y^2)

def optimization\_function(solution):

return solution[0]\*2 + solution[1]\*2

# Initialize parameters

def initialize\_parameters():

population\_size = 50

num\_genes = 2 # Dimensionality of the solution

mutation\_rate = 0.1

crossover\_rate = 0.8

num\_generations = 100

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

# Initialize population

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

return np.random.uniform(-10, 10, (population\_size, num\_genes))

# Evaluate fitness

def evaluate\_fitness(population):

return np.array([optimization\_function(ind) for ind in population])

# Selection

def select\_parents(population, fitness):

probabilities = 1 / (fitness + 1e-6) # Avoid division by zero

probabilities /= probabilities.sum()

indices = np.random.choice(len(population), size=len(population), p=probabilities)

return population[indices]

# Crossover

def crossover(parents, crossover\_rate):

offspring = []

for i in range(0, len(parents), 2):

if i + 1 < len(parents) and np.random.rand() < crossover\_rate:

point = np.random.randint(1, parents.shape[1])

offspring1 = np.concatenate((parents[i, :point], parents[i + 1, point:]))

offspring2 = np.concatenate((parents[i + 1, :point], parents[i, point:]))

offspring.extend([offspring1, offspring2])

else:

offspring.extend([parents[i], parents[i + 1] if i + 1 < len(parents) else parents[i]])

return np.array(offspring)

# Mutation

def mutate(offspring, mutation\_rate):

for individual in offspring:

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

gene = np.random.randint(individual.size)

individual[gene] += np.random.normal(0, 1) # Gaussian mutation

return offspring

# Gene expression (identity mapping in this example)

def gene\_expression(population):

return population

# Gene Expression Algorithm

def gene\_expression\_algorithm():

# Step 1: Initialize parameters

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

# Step 2: Initialize population

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

best\_solution = None

best\_fitness = float('inf')

for generation in range(num\_generations):

# Step 3: Evaluate fitness

fitness = evaluate\_fitness(population)

# 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]

# Step 4: Selection

parents = select\_parents(population, fitness)

# Step 5: Crossover

offspring = crossover(parents, crossover\_rate)

# Step 6: Mutation

population = mutate(offspring, mutation\_rate)

# Step 7: Gene Expression

population = gene\_expression(population)

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

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

return best\_solution, best\_fitness

# Run the algorithm

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

gene\_expression\_algorithm()