VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sanjana Suresh(1BM22CS239)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Sanjana Suresh(1BM22CS239),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sonika Sharam D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

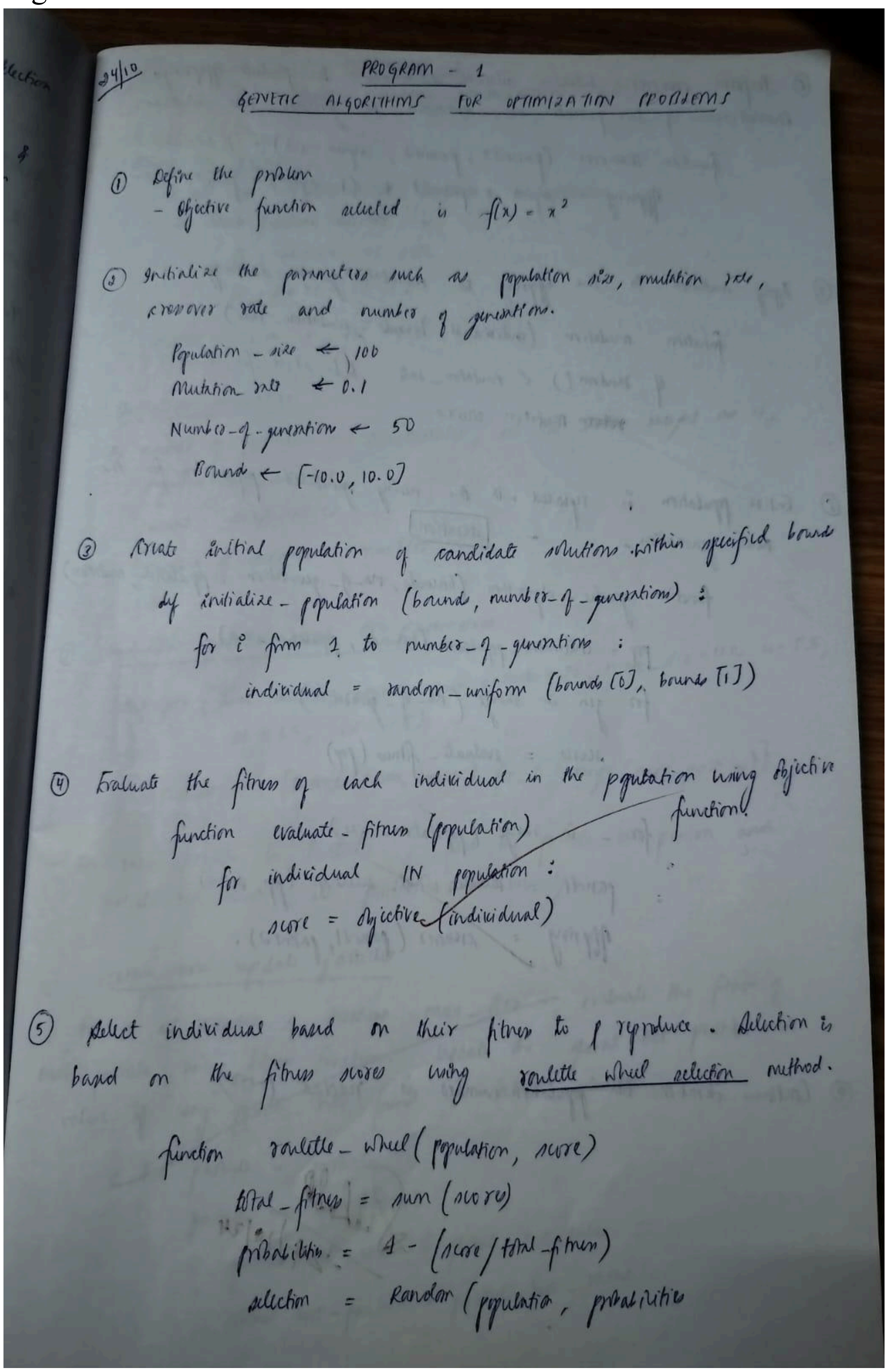
Github Link:

https://github.com/SanjanaSuresh30/BIS_LAB_1BM22CS239

Program 1
Genetic Algorithm for Optimization Problems.

Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:



24/10

PROGRAM - 1

GENETIC ALGORITHMS FOR OPTIMIZATION PROBLEMS

① Define the problem
- Objective function selected is $f(x) = x^2$

② Initialize the parameters such as population size, mutation rate, crossover rate and number of generations.

Population_size ← 100
Mutation rate ← 0.1
Number_of_generation ← 50
Bound ← [-10.0, 10.0]

③ Create initial population of candidate solutions within specified bounds

def initialize_population (bounds, number_of_generations) :
for i from 1 to number_of_generations :
individual = random_uniform (bounds [0], bounds [1])

④ Evaluate the fitness of each individual in the population using objective function.

function evaluate_fitness (population)
for individual IN population :
score = objective (individual)

⑤ Select individuals based on their fitness to reproduce. Selection is based on the fitness scores using roulette wheel selection method.

function roulette_wheel (population, score)
total_fitness = sum (scores)
probabilities = 1 - (score / total_fitness)
selection = Random (population, probabilities

⑥ Perform crossover between selected individuals to produce offspring. Characteristics of two parents are combined to generate new solutions.

function crossover (parent1, parent2, alpha = 0.5) : // contribution

offspring = alpha * parent1 + (1 - alpha) * parent2.

⑦ Apply mutation to the offspring to maintain genetic diversity

function mutation (individual, bounds, mutation_rate)

if random() < mutation_rate // to check for mutation

~~return~~ mutation occurs.

⑧ Entire population is replaced with the newly generated population for the next generation. - $\boxed{\text{iteration}}$

function genetic_algorithm (bounds, no_of_generation, population, mutation)

pop = initialize_population (bounds, population_size)

for gen in range (no_of_generation)

scores = evaluate_fitness (pop)

for _ in range (population_size)

parent1 = roulette_wheel_selection (pop, scores).

offspring = crossover (parent1, parent2).

⑨ Continue iterations till population number is reached (~50)

24/10/24

07/11/24

① Define the pr

- create o
- The fitnes

f

f

objective fu

② Initialize p

The number o

cognitive

social

number

③ Initialize

def pro

pa

- For each

particle

④ Iterate

For a

each particle

value if

Code:
```
# genetic algorithm search for continuous function optimization
from numpy.random import randint
from numpy.random import rand

# objective function
def objective(x):
  return x[0]**2.0 + x[1]**2.0

# decode bitstring to numbers
def decode(bounds, n_bits, bitstring):
  decoded = list()
  largest = 2**n_bits
  for i in range(len(bounds)):
    # extract the substring
    start, end = i * n_bits, (i * n_bits)+n_bits
    substring = bitstring[start:end]
    # convert bitstring to a string of chars
    chars = ''.join([str(s) for s in substring])
    # convert string to integer
    integer = int(chars, 2)
    # scale integer to desired range
    value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])
    # store
    decoded.append(value)
  return decoded

# tournament selection
def selection(pop, scores, k=3):
  # first random selection
  selection_ix = randint(len(pop))
  for ix in randint(0, len(pop), k-1):
    # check if better (e.g. perform a tournament)
    if scores[ix] < scores[selection_ix]:
      selection_ix = ix
  return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
  # children are copies of parents by default
  c1, c2 = p1.copy(), p2.copy()
  # check for recombination
  if rand() < r_cross:
    # select crossover point that is not on the end of the string
    pt = randint(1, len(p1)-2)
    # perform crossover
    c1 = p1[:pt] + p2[pt:]
```

```
    c2 = p2[:pt] + p1[pt:]
  return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
  for i in range(len(bitstring)):
    # check for a mutation
    if rand() < r_mut:
      # flip the bit
      bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut):
  # initial population of random bitstring
  pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in range(n_pop)]
  # keep track of best solution
  best, best_eval = 0, objective(decode(bounds, n_bits, pop[0]))
  # enumerate generations
  for gen in range(n_iter):
    # decode population
    decoded = [decode(bounds, n_bits, p) for p in pop]
    # evaluate all candidates in the population
    scores = [objective(d) for d in decoded]
    # check for new best solution
    for i in range(n_pop):
      if scores[i] < best_eval:
        best, best_eval = pop[i], scores[i]
        print(">%d, new best f(%s) = %f" % (gen,  decoded[i], scores[i]))
    # select parents
    selected = [selection(pop, scores) for _ in range(n_pop)]
    # create the next generation
    children = list()
    for i in range(0, n_pop, 2):
      # get selected parents in pairs
      p1, p2 = selected[i], selected[i+1]
      # crossover and mutation
      for c in crossover(p1, p2, r_cross):
        # mutation
        mutation(c, r_mut)
        # store for next generation
        children.append(c)
    # replace population
    pop = children
  return [best, best_eval]

# define range for input
bounds = [[-5.0, 5.0], [-5.0, 5.0]]
```

```
# define the total iterations
n_iter = 100
# bits per variable
n_bits = 16
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / (float(n_bits) * len(bounds))
# perform the genetic algorithm search
best, score = genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
decoded = decode(bounds, n_bits, best)
print('f(%s) = %f' % (decoded, score))
```

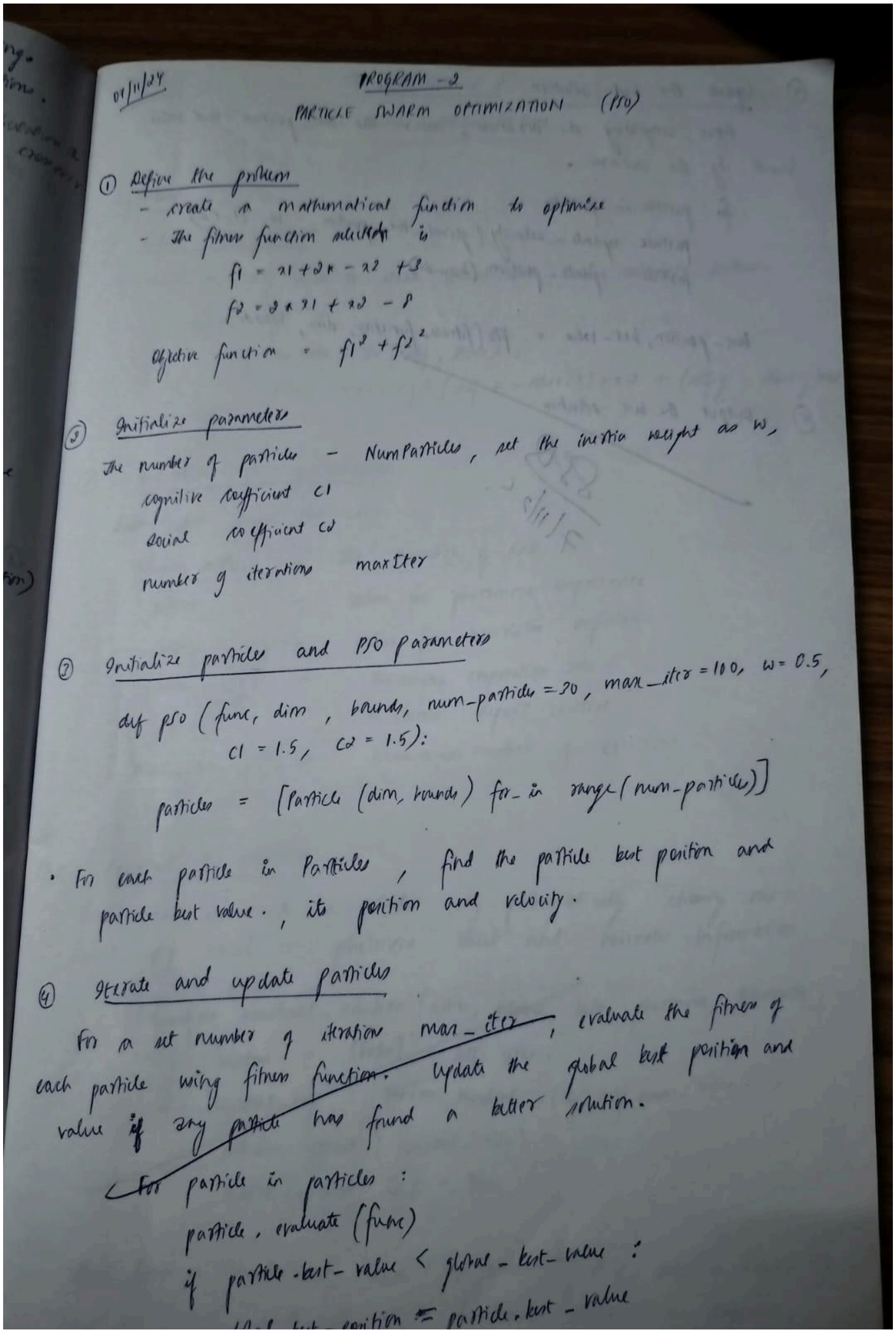
Output:

```
>0, new best f([-3.919219970703125, 0.202484130859375]) = 15.401285
>0, new best f([-1.178131103515625, 0.590362548828125]) = 1.736521
>0, new best f([-1.088714599609375, 0.171356201171875]) = 1.214662
>0, new best f([0.924224853515625, -0.46630859375]) = 1.071635
>0, new best f([-0.168609619140625, -0.45013427734375]) = 0.231050
>2, new best f([0.364532470703125, -0.026702880859375]) = 0.133597
>2, new best f([0.135498046875, -0.15380859375]) = 0.042017
>3, new best f([-0.020904541015625, -0.15380859375]) = 0.024094
>4, new best f([-0.0201416015625, -0.15380859375]) = 0.024063
>5, new best f([0.132293701171875, 0.019073486328125]) = 0.017865
>5, new best f([-0.019683837890625, -0.111236572265625]) = 0.012761
>6, new best f([-0.019683837890625, -0.0946044921875]) = 0.009337
>6, new best f([-0.0201416015625, -0.084075927734375]) = 0.007474
>6, new best f([-0.0555419921875, 0.066070556640625]) = 0.007450
>6, new best f([-0.025787353515625, 0.08056640625]) = 0.007156
>6, new best f([-0.002899169921875, -0.0836181640625]) = 0.007000
>7, new best f([-0.080108642578125, 0.019073486328125]) = 0.006781
>7, new best f([-0.019683837890625, -0.078582763671875]) = 0.006563
>8, new best f([-0.002899169921875, -0.019989013671875]) = 0.000408
>9, new best f([-0.002899169921875, -0.0164794921875]) = 0.000280
>10, new best f([-0.005645751953125, 0.0140380859375]) = 0.000229
>11, new best f([-0.003509521484375, -0.000457763671875]) = 0.000013
>12, new best f([-0.001983642578125, -0.000457763671875]) = 0.000004
>13, new best f([-0.000457763671875, -0.0006103515625]) = 0.000001
>16, new best f([-0.000152587890625, -0.0006103515625]) = 0.000000
>19, new best f([-0.000457763671875, -0.00030517578125]) = 0.000000
>19, new best f([-0.000152587890625, -0.00030517578125]) = 0.000000
>26, new best f([-0.000152587890625, -0.000152587890625]) = 0.000000
Done!
f([-0.000152587890625, -0.000152587890625]) = 0.000000
```

Program 2
Particle Swarm Optimization for Function Optimization.

Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:



01/11/24

PROGRAM - 2

PARTICLE SWARM OPTIMIZATION (PSO)

① Define the problem
- create a mathematical function to optimize
- The fitness function selected is

f1 = x1 + 2x - x2 + 3

f2 = 2 * x1 + x2 - 8

Objective function = $f1^2 + f2^2$

② Initialize parameters

The number of particles - NumParticles, set the inertia weight as w,
cognitive coefficient c1
social coefficient c2
number of iterations maxIter

③ Initialize particles and PSO parameters

def pso (func, dim, bounds, num_particles = 30, max_iter = 100, w = 0.5,
c1 = 1.5, c2 = 1.5):

particles = [Particle (dim, bounds) for _ in range(num_particles)]

- For each particle in Particles, find the particle best position and particle best value, its position and velocity.

④ Iterate and update particles

For a set number of iterations max_iter, evaluate the fitness of each particle using fitness function. Update the global best position and value if any particle has found a better solution.

~~for~~ particle in particles :
particle . evaluate (func)
if particle . best_ value < global _ best_ value :
global_best_position = particle . best _ value

⑤ Update the best solution

After completing the iterations, return the best position and value found by the swarm.

for particle in particles :
    particle . update - velocity (global - best - position, w, c1, c2)
    particle . update - position (bounds)

best - position, best - value = pso (fitness - function, dim, bounds)

⑤ Output the best solution

8 8 ✓
7/11/24.

Code:

```
import numpy as np

class Particle:
    def __init__(self, n_dimensions, minx, maxx):
        # Initialize particle's position and velocity
        self.position = np.random.uniform(minx, maxx, n_dimensions)
        self.velocity = np.random.uniform(-1, 1, n_dimensions)
        self.bestPos = np.copy(self.position)
        self.bestFitness = float('inf')  # Initialize to a large value
        self.fitness = float('inf')  # Fitness value will be updated later

    def evaluate(self, fitness_func):
        # Evaluate fitness of the current position
        self.fitness = fitness_func(self.position)

        # If current fitness is better than the best, update the best position
        if self.fitness < self.bestFitness:
            self.bestFitness = self.fitness
            self.bestPos = np.copy(self.position)

def pso(fitness_func, n_dimensions, N, max_iter, minx, maxx, w=0.5, c1=1.5, c2=1.5):
    # Initialize swarm (N particles)
    swarm = [Particle(n_dimensions, minx, maxx) for _ in range(N)]

    # Initialize the global best position and fitness
    best_fitness_swarm = float('inf')
    best_pos_swarm = np.zeros(n_dimensions)

    # Main PSO loop
    for gen in range(max_iter):
        avg_particle_best_fitness = 0  # Track the average best fitness of all particles

        for i in range(N):
            # Calculate new velocity
            r1, r2 = np.random.rand(2)
            swarm[i].velocity = (w * swarm[i].velocity +
                                 r1 * c1 * (swarm[i].bestPos - swarm[i].position) +
                                 r2 * c2 * (best_pos_swarm - swarm[i].position))

            # Update position
            swarm[i].position += swarm[i].velocity

            # Clip position to stay within bounds [minx, maxx]
            swarm[i].position = np.clip(swarm[i].position, minx, maxx)
```

```
            # Evaluate fitness and update personal best if necessary
            swarm[i].evaluate(fitness_func)

            # Update global best position if necessary
            if swarm[i].fitness < best_fitness_swarm:
                best_fitness_swarm = swarm[i].fitness
                best_pos_swarm = np.copy(swarm[i].position)

            # Accumulate the particle's best fitness for calculating average
            avg_particle_best_fitness += swarm[i].bestFitness

        # Calculate the average best fitness of all particles
        avg_particle_best_fitness /= N

        # Print progress (optional, can be commented out if not needed)
        # print(f"Generation {gen + 1}: Best Fitness = {best_fitness_swarm}, Avg Best Fitness =
{avg_particle_best_fitness}")

    # Return the best position found by the swarm and other metrics
    return best_pos_swarm, best_fitness_swarm, avg_particle_best_fitness, max_iter

# Example: Rastrigin function (a common benchmark in optimization)
def rastrigin_function(x):
    n = len(x)
    A = 10
    return A * n + np.sum(x**2 - A * np.cos(2 * np.pi * x))

# PSO parameters
n_dimensions = 2  # Example: 2D search space
N = 100           # Number of particles
max_iter = 100    # Number of iterations
minx = -100       # Minimum bound for position (for Rastrigin)
maxx = 100        # Maximum bound for position (for Rastrigin)

# Run the PSO algorithm with the Rastrigin function
best_position, best_fitness, avg_best_fitness, num_generations = pso(rastrigin_function,
n_dimensions, N, max_iter, minx, maxx)

# Output the final results
print(f"\nGlobal Best Position: {best_position}")
print(f"Global Best Fitness Value: {best_fitness}")
print(f"Average Particle Best Fitness Value: {avg_best_fitness}")
print(f"Number of Generations: {num_generations}")
```

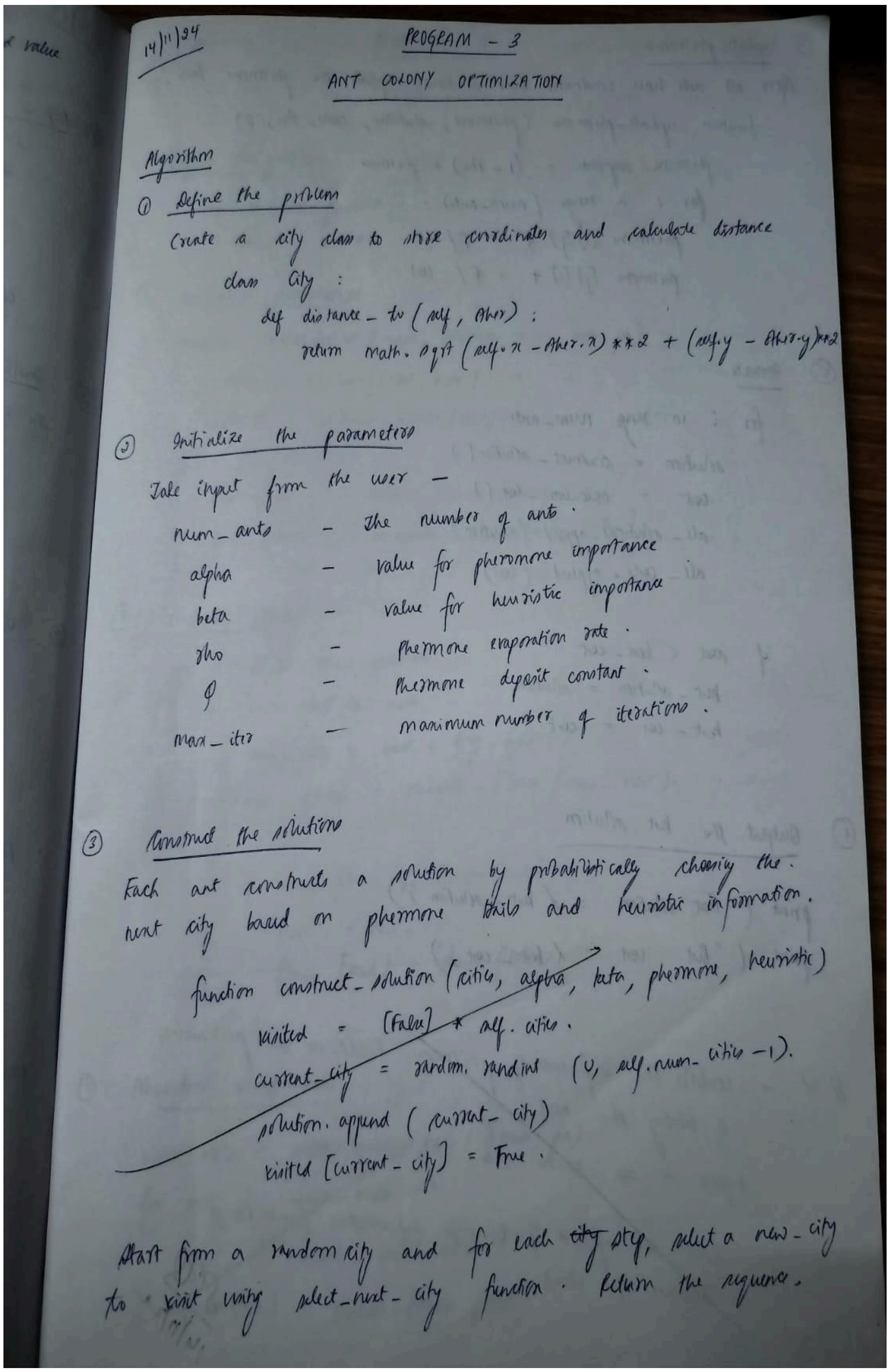
Output:

```
Global Best Position: [-2.13027709e-09 -1.66615351e-09]
Global Best Fitness Value: 0.0
Average Particle Best Fitness Value: 1.8100830097012023e-08
Number of Generations: 100
```

Program 3
Ant Colony Optimization for the Traveling Salesman Problem

Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:



14/11/24

PROGRAM - 3

ANT COLONY OPTIMIZATION

Algorithm

① Define the problem

Create a city class to store coordinates and calculate distance

```
class City :
    def distance_to (self, other) :
        return math.sqrt (self.x - other.x) ** 2 + (self.y - other.y) ** 2
```

② Initialize the parameters

Take input from the user -

num_ants	-	The number of ants.
alpha	-	value for pheromone importance
beta	-	value for heuristic importance
rho	-	Pheromone evaporation rate.
q	-	Pheromone deposit constant.
max_iter	-	maximum number of iterations.

③ Construct the solutions

Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.

```
function construct_solution (cities, alpha, beta, pheromone, heuristic)
    visited = [False] * self.cities.
    current_city = random.randint (0, self.num_cities - 1).
    solution.append ( current_city)
    visited [current_city] = True.
```

Start from a random city and for each ~~city~~ step, select a new city to visit using select_next_city function. Return the sequence.

④ Update phermone

After all ants have constructed their solutions, update the phermone trail.

```
function update_phermone (phermone, solutions, costs, rho, Q)
    phermone_evaporate = (1 - rho) * phermone.
    for i in range (num_ants) :
        phermone [i][j] + = Q / cost
        phermone [j][i] + = Q / cost.
```

⑤ Iterate.

```
for i in range num_ants.
    solution = construct_solution ()
    cost = calculate_cost ()
    all_solutions.append (solution)
    all_costs.append (cost)

    if cost < best_cost :
        best_solution = solution
        best_cost = cost.
```

⑥ Output the best solution

```
print ("Best solution : {best_solution}")
print ("Best cost ": {best_cost}")
```

// α - controls the influence of phermone trail on the probability of selecting the next city.

higher α ⇒ more ants likely to follow

β - controls the influence of heuristic information (inverse of distance b/w cities)

21/11/24

① Define the pr

def objec
return

⑤ Initialize

function

nests

fitness

po

best

best

⑦ Evaluate

for i =

for

⑨ Abandon

worst_

for n

Code:

```
import random as rn
import numpy as np
from numpy.random import choice as np_choice

class AntColony(object):

    def __init__(self, distances, n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        """
        Args:
            distances (2D numpy.array): Square matrix of distances. Diagonal is assumed to be np.inf.
            n_ants (int): Number of ants running per iteration
            n_best (int): Number of best ants who deposit pheromone
            n_iteration (int): Number of iterations
            decay (float): Rate it which pheromone decays. The pheromone value is multiplied by decay,
so 0.95 will lead to decay, 0.5 to much faster decay.
            alpha (int or float): exponent on pheromone, higher alpha gives pheromone more weight.
Default=1
            beta (int or float): exponent on distance, higher beta give distance more weight. Default=1

        Example:
            ant_colony = AntColony(german_distances, 100, 20, 2000, 0.95, alpha=1, beta=2)
        """
        self.distances  = distances
        self.pheromone = np.ones(self.distances.shape) / len(distances)
        self.all_inds = range(len(distances))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("placeholder", np.inf)
        for i in range(self.n_iterations):
            all_paths = self.gen_all_paths()
            self.spread_pheronome(all_paths, self.n_best, shortest_path=shortest_path)
            shortest_path = min(all_paths, key=lambda x: x[1])
            print (shortest_path)
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path
            self.pheromone = self.pheromone * self.decay
        return all_time_shortest_path
```

```
def spread_pheronome(self, all_paths, n_best, shortest_path):
    sorted_paths = sorted(all_paths, key=lambda x: x[1])
    for path, dist in sorted_paths[:n_best]:
        for move in path:
            self.pheromone[move] += 1.0 / self.distances[move]

def gen_path_dist(self, path):
    total_dist = 0
    for ele in path:
        total_dist += self.distances[ele]
    return total_dist

def gen_all_paths(self):
    all_paths = []
    for i in range(self.n_ants):
        path = self.gen_path(0)
        all_paths.append((path, self.gen_path_dist(path)))
    return all_paths

def gen_path(self, start):
    path = []
    visited = set()
    visited.add(start)
    prev = start
    for i in range(len(self.distances) - 1):
        move = self.pick_move(self.pheromone[prev], self.distances[prev], visited)
        path.append((prev, move))
        prev = move
        visited.add(move)
    path.append((prev, start)) # going back to where we started
    return path

def pick_move(self, pheromone, dist, visited):
    pheromone = np.copy(pheromone)
    pheromone[list(visited)] = 0

    row = pheromone ** self.alpha * (( 1.0 / dist) ** self.beta)

    norm_row = row / row.sum()
    move = np_choice(self.all_inds, 1, p=norm_row)[0]
    return move
distances = np.array([[np.inf, 2, 2, 5, 7],
                      [2, np.inf, 4, 8, 2],
                      [2, 4, np.inf, 1, 3],
                      [5, 8, 1, np.inf, 2],
                      [7, 2, 3, 2, np.inf]])
```

```
ant_colony = AntColony(distances, 1, 1, 100, 0.95, alpha=1, beta=1)
shortest_path = ant_colony.run()
print ("shorted_path: {}".format(shortest_path))
```

Output:

```
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 4), (4, 1), (1, 3), (3, 2), (2, 0)], 20.0)
([(0, 3), (3, 2), (2, 4), (4, 1), (1, 0)], 13.0)
([(0, 1), (1, 3), (3, 2), (2, 4), (4, 0)], 21.0)
([(0, 1), (1, 3), (3, 2), (2, 4), (4, 0)], 21.0)
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 2), (2, 4), (4, 3), (3, 0)], 16.0)
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 3), (3, 2), (2, 4), (4, 1), (1, 0)], 13.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 1), (1, 3), (3, 0)], 20.0)
([(0, 1), (1, 3), (3, 2), (2, 4), (4, 0)], 21.0)
([(0, 1), (1, 2), (2, 4), (4, 3), (3, 0)], 16.0)
([(0, 1), (1, 2), (2, 4), (4, 3), (3, 0)], 16.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 3), (3, 1), (1, 4), (4, 2), (2, 0)], 20.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 3), (3, 2), (2, 4), (4, 1), (1, 0)], 13.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
```

```
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 3), (3, 2), (2, 4), (4, 0)], 21.0)
([(0, 3), (3, 2), (2, 4), (4, 1), (1, 0)], 13.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 2), (2, 4), (4, 3), (3, 1), (1, 0)], 17.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
shorted_path: ([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
```

Program 4

Cuckoo Search (CS)

Algorithm:

21/11/24

PROGRAM - 4

CUCKOO SEARCH

① Define the problem

```
def objective_function(x)
    return x**2
```

② Initialize parameters

```
function cuckoosearch (num_iterations, num_nests, pa) :
    nests = random(num_nests)
    fitness = evaluate_fitness(nests)
    pa = 0.25
    best_nest = nests[fitness.indexof min()]
    best_fitness = _fitness[min_index]
```

③ Evaluate fitness

```
for i = 1 to max_iterations :
    for each nest in nests :
        new_nest = nest + levy_flight()
        new_fitness = evaluate_fitness(new_nest).

        if new_fitness < fitness[nest]
            nest = new_nest
            fitness[nest] = new_fitness.
```

④ Abandon worst nests

```
worst_nest = select_worst_nests(fitness, pa)
for nest in worst_nests :
    nest = random_initialization()
    fitness[nest] = evaluate_fitness(nest)
```

⑤ Update best solution

current_best_index = index_of_min (fitness)
current_best_fitness = fitness [current_best_index]
if current_best_fitness < best_fitness :
    best_fitness = current_best_fitness
    best_nest = nests [current_best_index].

⑥ Return the best solution

return best_nest , best_fitness.

OUTPUT

Best solution found : [0.00027413]
Best value : 7.5146110330 e-08.

APPLICATIONS

1. Feature selection in ML
2. Aerospace engineering - optimizing design of aircraft, wings
3. Training deep neural network
4. Drug design - optimize molecular structure

SSB
21/11/24

Code:

```
import numpy as np

# Objective function for 1D (x^2)
def objective_function_1d(x):
    return x[0]**2  # x is a 1D array, even though we just care about the first element

# Lévy Flight to generate new solutions
def levy_flight(num_dim, beta=1.5):
    sigma_u = (np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) /
            np.math.gamma((1 + beta) / 2) * beta * (2 ** ((beta - 1) / 2)))**(1 / beta)
    u = np.random.normal(0, sigma_u, num_dim)  # Lévy-distributed steps
    v = np.random.normal(0, 1, num_dim)
    return u / np.abs(v) ** (1 / beta)

# Cuckoo Search Algorithm for 1D
def cuckoo_search_1d(num_iterations, num_nests, pa=0.25):
    num_dim = 1  # 1D problem
    nests = np.random.rand(num_nests, num_dim) * 10 - 5  # Random initialization within [-5, 5]
    fitness = np.apply_along_axis(objective_function_1d, 1, nests)  # Evaluate initial fitness
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for _ in range(num_iterations):
        for i in range(num_nests):
            new_nest = nests[i] + levy_flight(num_dim)  # Generate new solution using Lévy flight
            new_fitness = objective_function_1d(new_nest)

            if new_fitness < fitness[i]:  # Replace if new solution is better
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon the worst nests
        worst_nests = np.argsort(fitness)[-int(pa * num_nests):]
        for j in worst_nests:
            nests[j] = np.random.rand(num_dim) * 10 - 5  # Randomly initialize new nests
            fitness[j] = objective_function_1d(nests[j])

        # Update best solution found so far
        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]
        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]

    return best_nest, best_fitness  # Return the best solution and its fitness
```

```
# Run the cuckoo search on the 1D problem
best_solution, best_fitness = cuckoo_search_1d(num_iterations=1000, num_nests=25)
print(f"Best solution found: {best_solution} with objective value: {best_fitness}")
```

Output:

```
    np.math.gamma((1 + beta) / 2) * beta * (2 ** ((beta - 1) / 2)))**(1 / beta)
  Best solution found: [-0.00014525] with objective value: 2.1097561074434205e-08
```

Program 5

Grey Wolf Optimizer (GWO):

Algorithm:

28/11/24

PROGRAM-5

GREY WOLF OPTIMIZER

① Define the problem

Create a mathematical function to optimize.

```
def fitness_function(x):
    return np.sum(x**2)
```

② Initialize parameters

Set the number of wolves and the number of iterations. and randomly generate the positions of the wolves within the defined search space.

```
def initialize(search_space, num_wolves, num_iterations):
    dimensions = len(search_space)
    search_space = np.array([[-5,5], [-5,5]])
    num_wolves = 10
    max_iterations = 100.
```

③ Initialize population

Generate an initial population of wolves with random position.

```
wolves = np.zeros((num_wolves, dimensions))
for i in range(num_wolves):
    wolves[i] = np.random.uniform(search_space[:,0],
                                  search_space[:,1]),

return wolves
```

5 ④ Evaluate fitness and update positions

```
def gwo_algorithm (search_space, num_wolves, max_iterations):
    alpha_wolf = np.zeros (dimensions)
    beta_wolf = np.zeros (dimensions)
    gamma_wolf = np.zeros (dimensions)

    for iteration in range (max_iteration) :
        a = 2 - (iteration / max_iterations) * 2

        for i in range (num_wolves) :
            fitness = fitness_function(wolves [i])
            if fitness < alpha_fitness :
                gamma_wolf = beta_wolf. copy ()
                gamma_fitness = beta_fitness
                beta_wolf = alpha_wolf. copy ()
                beta_fitness = alpha_fitness ()
                alpha_wolf = wolves [i]. copy ()
                alpha_fitness = fitness

            elif fitness < beta_fitness :
                gamma_wolf = beta_wolf. copy ()

            elif fitness < gamma_fitness :
                gamma_fitness = fitness.
```

⑤ Update positions of wolves

```
for i in range (num_wolves) :
    for j in range (dimensions) :
        r1 = np.random. random ()
        r2 = np.random. random ()

        A1 = 2 * a * r1 - a
```

⑥ Print b

OUTPUT

Opti

Opti

APPLICA

① Gr

② Me

③ Nu

D_alpha = np.abs (C1 * alpha-wolf [j] - wolves [i, j])
X1 = alpha-wolf [j] - A1 * D-alpha
X2 = beta - wolf [j] - A2 * D-beta
X3 = gamma - wolf [j] - A3 * D-gamma

wolves [i, j] = (X1 + X2 + X3) / 3

⑥ Print best solution

optimal_solution = gwo-algorithm (search-space, num-wolves, max)
print ("Optimal solution: ", optimal_solution)

OUTPUT

Optimal fitness : 2.581673 e -30
Optimal solution : [-1.089090e-15, -1.18122e-15]

APPLICATIONS

① Image processing and computer vision
② Medical Image Analysis
③ Network design and optimization

Code:

```
import numpy as np

def initialize_wolves(search_space, num_wolves):
    dimensions = len(search_space)
    wolves = np.zeros((num_wolves, dimensions))
    for i in range(num_wolves):
        wolves[i] = np.random.uniform(search_space[:, 0], search_space[:, 1])
    return wolves

def fitness_function(x):
    # Define your fitness function to evaluate the quality of a solution
    # Example: Sphere function (minimize the sum of squares)
    return np.sum(x**2)

def gwo_algorithm(search_space, num_wolves, max_iterations):
    dimensions = len(search_space)

    # Initialize wolves
    wolves = initialize_wolves(search_space, num_wolves)

    # Initialize alpha, beta, and gamma wolves
    alpha_wolf = np.zeros(dimensions)
    beta_wolf = np.zeros(dimensions)
    gamma_wolf = np.zeros(dimensions)

    # Initialize the fitness of alpha, beta, gamma wolves
    alpha_fitness = float('inf')
    beta_fitness = float('inf')
    gamma_fitness = float('inf')

    # Store the best fitness found
    best_fitness = float('inf')

    for iteration in range(max_iterations):
        a = 2 - (iteration / max_iterations) * 2  # Parameter a decreases linearly from 2 to 0

        print(f"Iteration {iteration + 1}/{max_iterations}")

        # Evaluate the fitness of all wolves
        for i in range(num_wolves):
            fitness = fitness_function(wolves[i])

            # Print the fitness of the current wolf
            print(f"Wolf {i+1} Fitness: {fitness}")
```

```
        # Update alpha, beta, gamma wolves based on fitness
        if fitness < alpha_fitness:
            gamma_wolf = beta_wolf.copy()
            gamma_fitness = beta_fitness
            beta_wolf = alpha_wolf.copy()
            beta_fitness = alpha_fitness
            alpha_wolf = wolves[i].copy()
            alpha_fitness = fitness
        elif fitness < beta_fitness:
            gamma_wolf = beta_wolf.copy()
            gamma_fitness = beta_fitness
            beta_wolf = wolves[i].copy()
            beta_fitness = fitness
        elif fitness < gamma_fitness:
            gamma_wolf = wolves[i].copy()
            gamma_fitness = fitness

    # Print the best fitness for this iteration
    print(f"Best Fitness in this Iteration: {alpha_fitness}")

    # Store the best overall fitness found so far
    if alpha_fitness < best_fitness:
        best_fitness = alpha_fitness

    # Update positions of wolves
    for i in range(num_wolves):
        for j in range(dimensions):
            r1 = np.random.random()
            r2 = np.random.random()

            A1 = 2 * a * r1 - a
            C1 = 2 * r2

            D_alpha = np.abs(C1 * alpha_wolf[j] - wolves[i, j])
            X1 = alpha_wolf[j] - A1 * D_alpha

            r1 = np.random.random()
            r2 = np.random.random()

            A2 = 2 * a * r1 - a
            C2 = 2 * r2

            D_beta = np.abs(C2 * beta_wolf[j] - wolves[i, j])
            X2 = beta_wolf[j] - A2 * D_beta

            r1 = np.random.random()
            r2 = np.random.random()
```

```
            A3 = 2 * a * r1 - a
            C3 = 2 * r2

            D_gamma = np.abs(C3 * gamma_wolf[j] - wolves[i, j])
            X3 = gamma_wolf[j] - A3 * D_gamma

            # Update the wolf's position
            wolves[i, j] = (X1 + X2 + X3) / 3

            # Ensure the new position is within the search space bounds
            wolves[i, j] = np.clip(wolves[i, j], search_space[j, 0], search_space[j, 1])

    print(f"Optimal Solution Found: {alpha_wolf}")
    print(f"Optimal Fitness: {best_fitness}")
    return alpha_wolf  # Return the best solution found

# Example usage
search_space = np.array([[-5, 5], [-5, 5]])  # Define the search space for the optimization problem
num_wolves = 10  # Number of wolves in the pack
max_iterations = 100  # Maximum number of iterations

# Run the GWO algorithm
optimal_solution = gwo_algorithm(search_space, num_wolves, max_iterations)

# Print the optimal solution
print("Optimal Solution:", optimal_solution)
```

Output:

```
Wolf 10 Fitness: 6.481498747178413e-28
Best Fitness in this Iteration: 6.427582965718472e-28
Iteration 100/100
Wolf 1 Fitness: 6.544397335300061e-28
Wolf 2 Fitness: 6.345120131054852e-28
Wolf 3 Fitness: 6.59801918052256e-28
Wolf 4 Fitness: 6.470220739353302e-28
Wolf 5 Fitness: 6.3692002008789615e-28
Wolf 6 Fitness: 6.491410895376178e-28
Wolf 7 Fitness: 6.402209940951431e-28
Wolf 8 Fitness: 6.588847523931703e-28
Wolf 9 Fitness: 6.4737400856791505e-28
Wolf 10 Fitness: 6.270157835039149e-28
Best Fitness in this Iteration: 6.270157835039149e-28
Optimal Solution Found: [1.74560701e-14 1.79527546e-14]
Optimal Fitness: 6.270157835039149e-28
Optimal Solution: [1.74560701e-14 1.79527546e-14]
```

Program 6

Parallel Cellular Algorithms and Programs:

Algorithm:

PROGRAM - 6

PARALLEL CELLULAR ALGORITHMS

Algorithm :

Algorithm Cellular Optimization

// Input : Grid size (G), Dimensionality (D), Search space bounds (S), Max iterations (N), fitness functions (F)

Output : Best position and its fitness value.

① Define the problem

```
def fitness_function (position):
    return sum (x ** 2)
```

② Initialize parameters

```
grid_size = (10, 10)
dim = 2
minx, maxx = -10.0, 10.0    // search space bounds
max_iteration = 50
```

③ Initialize population

```
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range (grid_size[0]):
        for j in range (grid_size[1]):
            fitness_grid [i, j] = fitness_function(population [i, j])
```

④ Update states

```
def update_cell (population, fitness_grid, i, j, minx, maxx):
    neighbours = get_neighbours (i, j)
    ~~best~~ new_position = population [best_neighbour [0], best_neighbour [1] + np.
    new_position = np.clip (new_positio
```

⑤ Iterate

population = in
for iteration
fitness_
for i in
for j in
new

⑥ Output t

best_position
best_fitnes
print ("
print ("

OUTPUT

Best positi
Best fitn

⑤ Iterate

```
population = initialize_population (grid_size, dim, minx, maxx)
for iteration in range (max_iterations) :
    fitness_grid = evaluate_fitness (population)

for i in range (grid_size [0]) :
    for j in range (grid_size [1]) :
        new_population [i, j] = update_cell (population, fitness_grid, i, j, minx, maxx)
```

⑥ Output the best solution

```
best_position = population[[best_index [0], best_index [1]]
best_fitness = np. min (fitness_grid)
print ("Best position found ", best_position)
print ("Best fitness found ", best_fitness)
```

OUTPUT

Best position found : [0.03123, 0.085816]
Best fitness found : [6.2945e-0.5]

Code:
```
import numpy as np
import random

# Step 1: Define the Problem (Optimization Function)
def fitness_function(position):
    """Example fitness function: Sphere function"""
    return sum(x**2 for x in position)

# Step 2: Initialize Parameters
grid_size = (10, 10)  # Grid size (10x10 cells)
dim = 2  # Dimensionality of each cell's position
minx, maxx = -10.0, 10.0  # Search space bounds
max_iterations = 50  # Number of iterations

# Step 3: Initialize Population (Random positions)
def initialize_population(grid_size, dim, minx, maxx):
    population = np.zeros((grid_size[0], grid_size[1], dim))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            population[i, j] = [random.uniform(minx, maxx) for _ in range(dim)]
    return population

# Step 4: Evaluate Fitness (Calculate fitness for each cell)
def evaluate_fitness(population):
    fitness_grid = np.zeros((grid_size[0], grid_size[1]))
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            fitness_grid[i, j] = fitness_function(population[i, j])
    return fitness_grid

# Step 5: Update States (Update each cell based on its neighbors)
def get_neighbors(i, j):
    """Returns the coordinates of neighboring cells."""
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if not (di == 0 and dj == 0):  # Exclude the cell itself
                ni, nj = (i + di) % grid_size[0], (j + dj) % grid_size[1]
                neighbors.append((ni, nj))
    return neighbors

def update_cell(population, fitness_grid, i, j, minx, maxx):
    """Update the state of a cell based on the average state of its neighbors."""
    neighbors = get_neighbors(i, j)
    best_neighbor = min(neighbors, key=lambda x: fitness_grid[x[0], x[1]])
```

```
    # Update cell position to move towards the best neighbor's position
    new_position = population[best_neighbor[0], best_neighbor[1]] + \
                   np.random.uniform(-0.1, 0.1, dim)  # Small random perturbation

    # Ensure the new position stays within bounds
    new_position = np.clip(new_position, minx, maxx)
    return new_position

# Step 6: Iterate (Repeat for a fixed number of iterations)
population = initialize_population(grid_size, dim, minx, maxx)
for iteration in range(max_iterations):
    fitness_grid = evaluate_fitness(population)

    # Update each cell in parallel (simultaneously)
    new_population = np.zeros_like(population)
    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            new_population[i, j] = update_cell(population, fitness_grid, i, j, minx, maxx)

    population = new_population

    # Print best fitness at each iteration
    best_fitness = np.min(fitness_grid)
    print(f"Iteration {iteration + 1}, Best Fitness: {best_fitness}")

# Step 7: Output the Best Solution
best_index = np.unravel_index(np.argmin(fitness_grid), fitness_grid.shape)
best_position = population[best_index[0], best_index[1]]
best_fitness = np.min(fitness_grid)
print("Best Position Found:", best_position)
print("Best Fitness Found:", best_fitness)
```

Output:

```
Iteration 1, Best Fitness: 0.10790716375710524
Iteration 2, Best Fitness: 0.05125805638738155
Iteration 3, Best Fitness: 0.0218929053373741
Iteration 4, Best Fitness: 0.01235767952739731
Iteration 5, Best Fitness: 0.00040347335375352995
Iteration 6, Best Fitness: 4.784045468576452e-05
Iteration 7, Best Fitness: 1.3808440429860477e-05
Iteration 8, Best Fitness: 5.4281094522260926e-05
Iteration 9, Best Fitness: 4.546725207857838e-05
Iteration 10, Best Fitness: 1.8955982987296623e-05
Iteration 11, Best Fitness: 0.00026101063881429817
Iteration 12, Best Fitness: 0.00016406794985828426
Iteration 13, Best Fitness: 0.00012569781575646З
Iteration 14, Best Fitness: 0.00010673203849З6587
Iteration 15, Best Fitness: 0.00016110358880238045
Iteration 16, Best Fitness: 0.00014646766126233117
Iteration 17, Best Fitness: 6.564956616113555e-06
Iteration 18, Best Fitness: 2.7219360396320874e-06
Iteration 19, Best Fitness: 3.0485909672568264e-05
Iteration 20, Best Fitness: 4.672570817565934e-05
Iteration 21, Best Fitness: 8.27468781151064e-05
Iteration 22, Best Fitness: 1.7773028005650622e-05
Iteration 23, Best Fitness: 0.00016541330204756794
Iteration 24, Best Fitness: 0.00019532486561694786
Iteration 25, Best Fitness: 6.565308646310312e-05
Iteration 26, Best Fitness: 0.00013093104531390992
Iteration 27, Best Fitness: 0.000274556470941331
Iteration 28, Best Fitness: 3.5147714361893384e-05
Iteration 29, Best Fitness: 5.8744735159171534e-05
Iteration 30, Best Fitness: 1.1537072703071819e-05
Iteration 31, Best Fitness: 0.00011259129611035464
Iteration 32, Best Fitness: 8.204048347105135e-05
Iteration 33, Best Fitness: 0.00020594083592498773
Iteration 34, Best Fitness: 0.00016843445682215467
Iteration 35, Best Fitness: 6.115630628534281e-05
Iteration 36, Best Fitness: 0.00013827314720531877
Iteration 37, Best Fitness: 0.00034035049575529826
Iteration 38, Best Fitness: 5.436954614480845e-05
Iteration 39, Best Fitness: 0.00016534943291831238
Iteration 40, Best Fitness: 4.380711441069458e-06
Iteration 41, Best Fitness: 0.00020957408918770374
Iteration 42, Best Fitness: 0.0006829011407880717
Iteration 43, Best Fitness: 0.0001689502798770622
Iteration 44, Best Fitness: 1.2244250202786437e-05
Iteration 45, Best Fitness: 4.835683058991596e-05
Iteration 46, Best Fitness: 1.4640390719511358e-05
Iteration 47, Best Fitness: 0.0005766573004094843
Iteration 48, Best Fitness: 0.00029028005771969964
Iteration 49, Best Fitness: 0.00019497022896451086
Iteration 50, Best Fitness: 0.00010544267384798719
Best Position Found: [-0.04658244 -0.08144629]
Best Fitness Found: 0.00010544267384798719
```

Program 7

Optimization via Gene Expression Algorithms:

Algorithm:

PROGRAM - 7

OPTIMIZATION VIA GENE EXPRESSION ALGORITHM

5

1. Algorithm

Algorithm GeneExpressionAlgorithm

Input : Population of size (P), Number of genes (G), Mutation rate (M), Crossover rate (C), Number of generation (N), Search space, Fitness function (F)

Output : Best solution and its fitness value

1. Define the problem

```
def fitness_function(x):
    return sum(x_i ** 2)
```

2. Initialize parameters

```
population_size = 50
num_genes = 5
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 100
search_space = (-10, 10)
```

3. Initialize population

```
def initialize_population(pop_size, num_gene, search_space):
    return np.random.uniform(search_space[0], search_space[1])
```

4. Evaluate fitness

```
def evaluate_fitness(population)
    return np.array([fitness_function(individual) for individual in population]).

def tournament_selection(population, fitness):
    for i in range(len(population)):
        selected.append(population[i] if fitness[i] > fitness[j] else population[j])
```

def crossover

def mu

5. Iterate

for genera

```
def crossover (parent1, parent2, rate) :
    if random.random() < rate :
        child1 = np.concatenate ((parent1[:point], parent2 [point :]))
        child2 = np. concatenate ((parent 2 [:point], parent 1[point :]))

def mutate (individual, rate, search-space) :
    for i in range (individual)) :
        if random. random () < rate :
            individual [i] + = np. random. normal (0, 1)
    return individual
```

⑤ Iterate

```
for generation in range (num- generation) :
    fitness = evaluate- fitness (population)
    selected- population = tournament - selection (population, fitness)
    ~~crossover~~ next_generation []= cross - over
    next- generation = [mutate ( ind, mutate - rate, search space)
```

⑥ Output.

```
print (f" Best solution found : {best solution}")
print (f" Best fitness : { best- fitness } ")
```

Code:

```
import numpy as np
import random

# Define any optimization function to minimize (can be changed as needed)
def custom_function(x):
    # Example function: x^2 to minimize
    return np.sum(x ** 2)  # Ensuring the function works for multidimensional inputs

# Initialize population of genetic sequences (each individual is a sequence of genes)
def initialize_population(population_size, num_genes, lower_bound, upper_bound):
    # Create a population of random genetic sequences
    population = np.random.uniform(lower_bound, upper_bound, (population_size, num_genes))
    return population

# Evaluate the fitness of each individual (genetic sequence) in the population
def evaluate_fitness(population, fitness_function):
    fitness = np.zeros(population.shape[0])
    for i in range(population.shape[0]):
        fitness[i] = fitness_function(population[i])  # Apply the fitness function to each individual
    return fitness

# Perform selection: Choose individuals based on their fitness (roulette wheel selection)
def selection(population, fitness, num_selected):
    # Select individuals based on their fitness (higher fitness, more likely to be selected)
    probabilities = fitness / fitness.sum()  # Normalize fitness to create selection probabilities
    selected_indices = np.random.choice(range(len(population)), size=num_selected, p=probabilities)
    selected_population = population[selected_indices]
    return selected_population

# Perform crossover: Combine pairs of individuals to create offspring
def crossover(selected_population, crossover_rate):
    new_population = []
    num_individuals = len(selected_population)
    for i in range(0, num_individuals - 1, 2):  # Iterate in steps of 2, skipping the last one if odd
        parent1, parent2 = selected_population[i], selected_population[i + 1]
        if len(parent1) > 1 and random.random() < crossover_rate:  # Only perform crossover if more
than 1 gene
            crossover_point = random.randint(1, len(parent1) - 1)  # Choose a random crossover point
            offspring1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            offspring2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            new_population.extend([offspring1, offspring2])  # Create two offspring
        else:
            new_population.extend([parent1, parent2])  # No crossover, retain the parents

    # If the number of individuals is odd, carry the last individual without crossover
    if num_individuals % 2 == 1:
```

```
        new_population.append(selected_population[-1])
    return np.array(new_population)

# Perform mutation: Introduce random changes in offspring
def mutation(population, mutation_rate, lower_bound, upper_bound):
    for i in range(population.shape[0]):
        if random.random() < mutation_rate:  # Apply mutation based on the rate
            gene_to_mutate = random.randint(0, population.shape[1] - 1)  # Choose a random gene to
mutate
            population[i, gene_to_mutate] = np.random.uniform(lower_bound, upper_bound)  # Mutate
the gene
    return population

# Gene expression: In this context, it is how we decode the genetic sequence into a solution
def gene_expression(individual, fitness_function):
    return fitness_function(individual)

# Main function to run the Gene Expression Algorithm
def gene_expression_algorithm(population_size, num_genes, lower_bound, upper_bound,
                              max_generations, mutation_rate, crossover_rate, fitness_function):
    # Step 2: Initialize the population of genetic sequences
    population = initialize_population(population_size, num_genes, lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    # Step 9: Iterate for the specified number of generations
    for generation in range(max_generations):
        # Step 4: Evaluate fitness of the current population
        fitness = evaluate_fitness(population, fitness_function)

        # Track the best solution found so far
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.argmin(fitness)]

        # Step 5: Perform selection (choose individuals based on fitness)
        selected_population = selection(population, fitness, population_size // 2)  # Select half of the
population

        # Step 6: Perform crossover to generate new individuals
        offspring_population = crossover(selected_population, crossover_rate)

        # Step 7: Perform mutation on the offspring population
        population = mutation(offspring_population, mutation_rate, lower_bound, upper_bound)

        # Print output every 10 generations
```

```
        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}/{max_generations}, Best Fitness: {best_fitness}")

    # Step 10: Output the best solution found
    return best_solution, best_fitness

# Parameters for the algorithm
population_size = 50  # Number of individuals in the population
num_genes = 1  # Number of genes (for a 1D problem, this is just 1, extendable for higher
dimensions)
lower_bound = -5  # Lower bound for the solution space
upper_bound = 5  # Upper bound for the solution space
max_generations = 100  # Number of generations to evolve the population
mutation_rate = 0.1  # Mutation rate (probability of mutation per gene)
crossover_rate = 0.7  # Crossover rate (probability of crossover between two parents)

# Run the Gene Expression Algorithm
best_solution, best_fitness = gene_expression_algorithm(
    population_size, num_genes, lower_bound, upper_bound,
    max_generations, mutation_rate, crossover_rate, custom_function)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_fitness)
```

Output:

```
Generation 10/100, Best Fitness: 0.0001382948285174357
Generation 20/100, Best Fitness: 0.0001382948285174357
Generation 30/100, Best Fitness: 0.0001382948285174357
Generation 40/100, Best Fitness: 0.0001382948285174357
Generation 50/100, Best Fitness: 0.0001382948285174357
Generation 60/100, Best Fitness: 0.0001382948285174357
Generation 70/100, Best Fitness: 0.0001382948285174357
Generation 80/100, Best Fitness: 0.0001382948285174357
Generation 90/100, Best Fitness: 0.00010209871847059898
Generation 100/100, Best Fitness: 0.00010209871847059898

Best Solution Found: [-0.01010439]
Best Fitness Value: 0.00010209871847059898
```