# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**“JnanaSangama”, Belgaum -590014, Karnataka.**

**LAB RECORD**

Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**Shreya Bharamanna Patil(1BM23CS420)**

***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 **Shreya Bharamanna Patil(1BM23CS420),** 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 mention subject and the work prescribed for the said degree.

| Saritha A N Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Soodha Professor & HOD Department of CSE, BMSCE |
| --- | --- |

# Index

| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| --- | --- | --- | --- |
| 1 | 24-10-24 | Genetic Algorithm for Optimization Problems | 1 |
| 2 | 7-11-24 | Particle Swarm Optimization for Function Optimization | 4 |
| 3 | 14-11-24 | Ant Colony Optimization for the Traveling Salesman Problem | 7 |
| 4 | 21-11-24 | Cuckoo Search (CS) | 12 |
| 5 | 28-11-24 | Grey Wolf Optimizer (GWO) | 14 |
| 6 | 5-12-24 | Parallel Cellular Algorithms and Programs | 17 |
| 7 | 12-12-24 | Optimization via Gene Expression Algorithms | 19 |

Github Link:

https://github.com/PatilShreya22/BIS

## Program 1

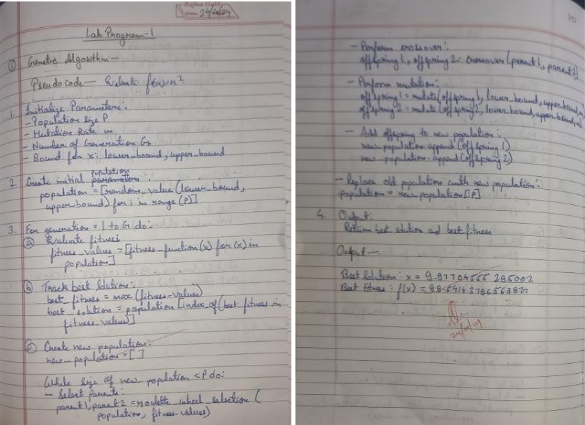
Genetic Algorithm for Optimization Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:



Code:

#GENETIC ALGORITHM

import numpy as np import random

# Define the fitness function def fitness\_function(x):

return x \*\* 2

# Initialize parameters population\_size = 100

mutation\_rate = 0.1

num\_generations = 50

bounds = (-10, 10)

# Step 1: Create initial population

def create\_initial\_population(size, bounds):

return [random.uniform(bounds[0], bounds[1]) for \_ in range(size)]

# Step 2: Evaluate fitness of the population def evaluate\_population(population):

return [fitness\_function(individual) for individual in population]

# Step 3: Selection using roulette-wheel selection def selection(population, fitness):

total\_fitness = sum(fitness)

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

return np.random.choice(population, size=2, p=selection\_probs)

# Step 4: Crossover operation def crossover(parent1, parent2):

alpha = random.uniform(0, 1)

offspring1 = alpha \* parent1 + (1 - alpha) \* parent2 offspring2 = alpha \* parent2 + (1 - alpha) \* parent1 return offspring1, offspring2

# Step 5: Mutation operation def mutate(individual, bounds):

if random.random() < mutation\_rate:

return random.uniform(bounds[0], bounds[1]) return individual

# Main Genetic Algorithm loop def genetic\_algorithm(bounds):

# Step 1: Create initial population

population = create\_initial\_population(population\_size, bounds)

best\_solution = None best\_fitness = float('-inf')

for generation in range(num\_generations): # Step 2: Evaluate fitness

fitness = evaluate\_population(population)

# Track the best solution current\_best\_fitness = max(fitness)

if current\_best\_fitness > best\_fitness: best\_fitness = current\_best\_fitness

best\_solution = population[fitness.index(current\_best\_fitness)]

# Step 3: Create new population new\_population = []

while len(new\_population) < population\_size: parent1, parent2 = selection(population, fitness) offspring1, offspring2 = crossover(parent1, parent2)

new\_population.append(mutate(offspring1, bounds)) new\_population.append(mutate(offspring2, bounds))

# Replace the old population with the new population population = new\_population[:population\_size]

return best\_solution, best\_fitness # Run the Genetic Algorithm

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

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

Output:



## Program 2

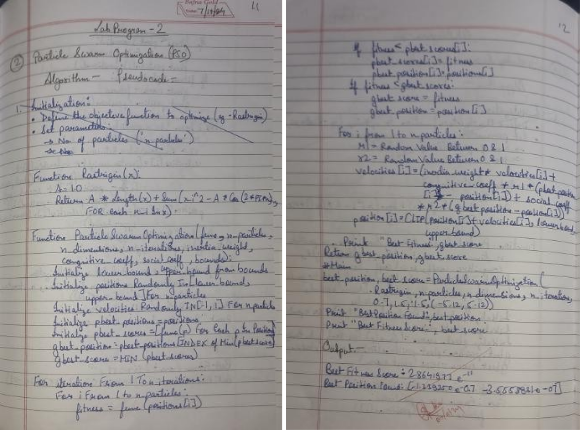
Particle Swarm Optimization for Function Optimization

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

#PARTICLE SWARM OPTIMIZATION

import numpy as np

# Rastrigin function: A benchmark function for optimization problems def rastrigin(x):

A = 10

# Calculate the Rastrigin function value based on the input vector x return A \* len(x) + sum(x\_i\*\*2 - A \* np.cos(2 \* np.pi \* x\_i) for x\_i in x)

# Particle Swarm Optimization class class ParticleSwarmOptimizer:

def init (self, func, n\_particles, n\_dimensions, n\_iterations, inertia\_weight=0.7, cognitive\_coeff=1.5, social\_coeff=1.5, bounds=(-5.12, 5.12)):

self.func = func # The function to optimize

self.n\_particles = n\_particles # Number of particles in the swarm self.n\_dimensions = n\_dimensions # Dimensions of the search space self.n\_iterations = n\_iterations # Number of iterations for the optimization self.lower\_bound, self.upper\_bound = bounds # Bounds for the search space

# Initialize particle positions randomly within the specified bounds

self.positions = np.random.uniform(self.lower\_bound, self.upper\_bound, (n\_particles, n\_dimensions))

# Initialize particle velocities randomly

self.velocities = np.random.uniform(-1, 1, (n\_particles, n\_dimensions)) # Personal best positions and scores for each particle self.pbest\_positions = np.copy(self.positions)

self.pbest\_scores = np.array([func(p) for p in self.positions]) # Evaluate initial fitness # Global best position and score among all particles

self.gbest\_position = self.pbest\_positions[np.argmin(self.pbest\_scores)] self.gbest\_score = np.min(self.pbest\_scores)

def optimize(self):

# Main loop for the optimization process for \_ in range(self.n\_iterations):

for i in range(self.n\_particles):

# Evaluate the fitness of the current position fitness = self.func(self.positions[i])

# Update personal best if the current fitness is better if fitness < self.pbest\_scores[i]:

self.pbest\_scores[i] = fitness self.pbest\_positions[i] = self.positions[i]

# Update global best if the current fitness is better if fitness < self.gbest\_score:

self.gbest\_score = fitness self.gbest\_position = self.positions[i]

# Generate random coefficients for cognitive and social components

r1, r2 = np.random.rand(self.n\_dimensions), np.random.rand(self.n\_dimensions) # Update velocities based on inertia, personal best, and global best self.velocities = (self.velocities \* 0.7 + # Inertia weight

1.5 \* r1 \* (self.pbest\_positions - self.positions) + # Cognitive component

1.5 \* r2 \* (self.gbest\_position - self.positions)) # Social component # Update positions based on new velocities and clip to stay within bounds

self.positions = np.clip(self.positions + self.velocities, self.lower\_bound, self.upper\_bound)

# Print the best fitness found so far in this iteration print(f"Best Fitness: {self.gbest\_score}")

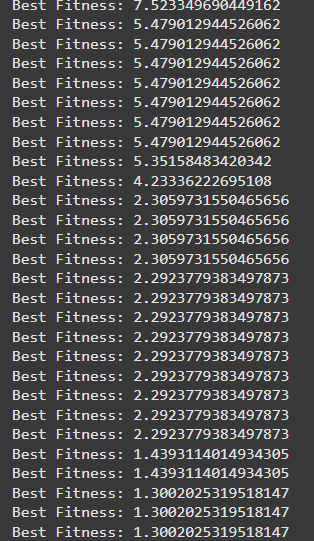
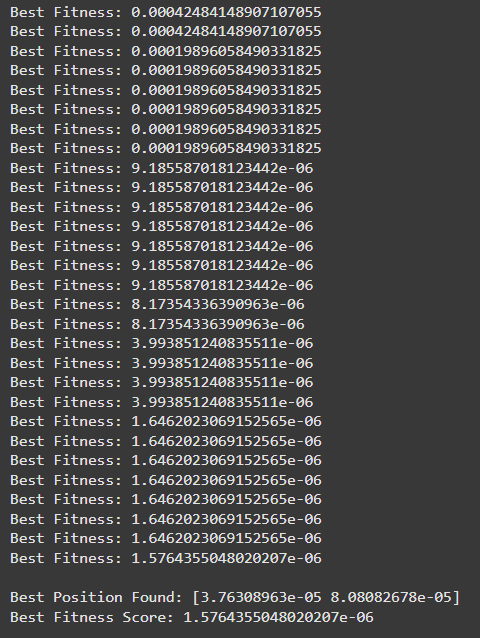
# Return the best position and score found after all iterations return self.gbest\_position, self.gbest\_score

# Create and run the optimizer

pso = ParticleSwarmOptimizer(func=rastrigin, n\_particles=30, n\_dimensions=2, n\_iterations=100) best\_position, best\_score = pso.optimize()

# Print the best position and corresponding fitness score found print("\nBest Position Found:", best\_position)

print("Best Fitness Score:", best\_score) Output:

## Program 3

Ant Colony Optimization for the Traveling Salesman Problem

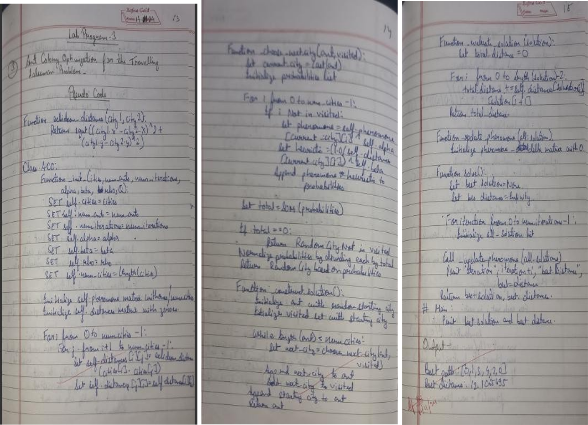
The foraging behaviour of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest.

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.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:



Code:

#ANT COLONY OPTIMIZATION

import random import numpy as np

# Distance calculation (Euclidean distance) def euclidean\_distance(city1, city2):

return np.sqrt((city1[0] - city2[0])\*\*2 + (city1[1] - city2[1])\*\*2)

# Ant Colony Optimization Algorithm class ACO:

def init (self, cities, num\_ants=10, num\_iterations=100, alpha=1.0, beta=2.0, rho=0.5, Q=100): self.cities = cities

self.num\_ants = num\_ants self.num\_iterations = num\_iterations

self.alpha = alpha # Importance of pheromone

self.beta = beta # Importance of heuristic information (distance) self.rho = rho # Pheromone evaporation rate

self.Q = Q # Total pheromone deposited per ant per tour self.num\_cities = len(cities)

# Initialize pheromone matrix (for each pair of cities)

self.pheromone = np.ones((self.num\_cities, self.num\_cities)) / self.num\_cities self.distances = np.zeros((self.num\_cities, self.num\_cities))

# Compute distance matrix

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

for j in range(i + 1, self.num\_cities):

self.distances[i][j] = self.distances[j][i] = euclidean\_distance(cities[i], cities[j])

def \_choose\_next\_city(self, ant, visited):

# Calculate the probability of moving to each city current\_city = ant[-1]

probabilities = []

for i in range(self.num\_cities): if i not in visited:

pheromone = self.pheromone[current\_city][i] \*\* self.alpha heuristic = (1.0 / self.distances[current\_city][i]) \*\* self.beta probabilities.append(pheromone \* heuristic)

else:

probabilities.append(0)

# Normalize probabilities total = sum(probabilities)

if total == 0: # In case there's no valid path (shouldn't happen with good settings)

return random.choice([i for i in range(self.num\_cities) if i not in visited]) probabilities = [prob / total for prob in probabilities]

# Choose next city based on probabilities

next\_city = random.choices(range(self.num\_cities), probabilities)[0] return next\_city

def \_construct\_solution(self):

# Each ant starts at a random city

ant = [random.randint(0, self.num\_cities - 1)] visited = set(ant)

while len(ant) < self.num\_cities:

next\_city = self.\_choose\_next\_city(ant, visited) ant.append(next\_city)

visited.add(next\_city)

# Return to the starting city ant.append(ant[0])

return ant

def \_evaluate\_solution(self, solution):

# Calculate the total distance of the tour total\_distance = 0

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

total\_distance += self.distances[solution[i]][solution[i + 1]] return total\_distance

def \_update\_pheromone(self, all\_solutions): # Initialize pheromone update matrix

pheromone\_delta = np.zeros((self.num\_cities, self.num\_cities))

# For each solution, deposit pheromone for solution in all\_solutions:

tour\_length = self.\_evaluate\_solution(solution) for i in range(len(solution) - 1):

pheromone\_delta[solution[i]][solution[i + 1]] += self.Q / tour\_length

# Evaporate pheromone

self.pheromone = (1 - self.rho) \* self.pheromone + pheromone\_delta

def solve(self): best\_solution = None best\_distance = float('inf')

for iteration in range(self.num\_iterations): all\_solutions = []

# Each ant constructs a solution for ant in range(self.num\_ants):

solution = self.\_construct\_solution() all\_solutions.append(solution)

tour\_length = self.\_evaluate\_solution(solution)

# Update best solution if necessary if tour\_length < best\_distance:

best\_solution = solution best\_distance = tour\_length

# Update pheromones based on solutions found self.\_update\_pheromone(all\_solutions)

print(f"Iteration {iteration + 1}, Best Distance: {best\_distance}") return best\_solution, best\_distance

# Function to take user input for cities def get\_user\_input():

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

print("Enter the coordinates of each city (x, y):") for i in range(num\_cities):

x, y = map(float, input(f"City {i+1}: ").split()) cities.append((x, y))

return cities

# Example usage:

if name == " main ": # Take user input for cities cities = get\_user\_input()

# Take user input for ACO parameters

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

alpha = float(input("Enter the value of alpha (pheromone importance): ")) beta = float(input("Enter the value of beta (distance importance): "))

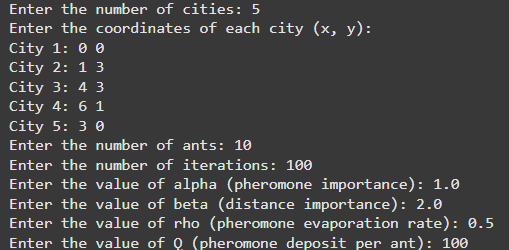
rho = float(input("Enter the value of rho (pheromone evaporation rate): ")) Q = float(input("Enter the value of Q (pheromone deposit per ant): "))

# Create an instance of ACO and solve the problem

aco = ACO(cities, num\_ants, num\_iterations, alpha, beta, rho, Q) best\_solution, best\_distance = aco.solve()

print(f"\nBest Solution (Tour): {best\_solution}") print(f"Best Distance: {best\_distance}")

Output:



## Program 4

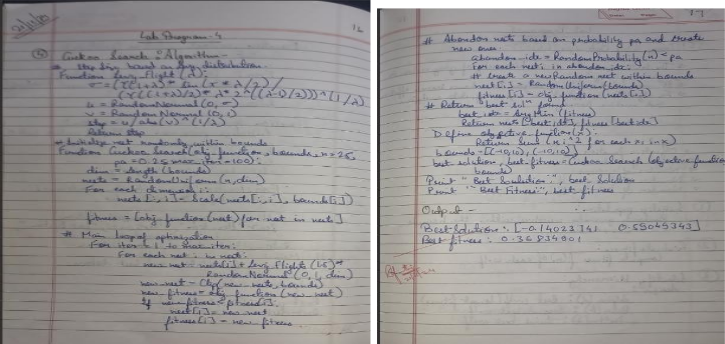
Cuckoo Search (CS)

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

#CUCKOO SEARCH

import numpy as np

import math # Import the standard math module def levy\_flight(Lambda):

sigma = (math.gamma(1 + Lambda) \* math.sin(math.pi \* Lambda / 2) /

(math.gamma((1 + Lambda) / 2) \* Lambda \* 2 \*\* ((Lambda - 1) / 2))) \*\* (1 / Lambda) u = np.random.normal(0, sigma, 1)

v = np.random.normal(0, 1, 1) step = u / abs(v) \*\* (1 / Lambda) return step

def cuckoo\_search(obj\_function, bounds, n=25, pa=0.25, max\_iter=100): # Initialize nests

dim = len(bounds)

nests = np.random.rand(n, dim) for i in range(dim):

nests[:, i] = nests[:, i] \* (bounds[i][1] - bounds[i][0]) + bounds[i][0] fitness = np.array([obj\_function(nest) for nest in nests])

# Start optimization

for \_ in range(max\_iter): for i in range(n):

# Generate a new solution via Levy flight

new\_nest = nests[i] + levy\_flight(1.5) \* np.random.randn(dim) # Apply bounds

new\_nest = np.clip(new\_nest, [b[0] for b in bounds], [b[1] for b in bounds]) new\_fitness = obj\_function(new\_nest)

# Update if new solution is better if new\_fitness < fitness[i]:

nests[i] = new\_nest fitness[i] = new\_fitness

# Abandon some nests and create new ones abandon\_idx = np.random.rand(n) < pa

for i in np.where(abandon\_idx)[0]:

nests[i] = np.random.rand(dim) \* (np.array([b[1] for b in bounds]) - np.array([b[0] for b in bounds])) + np.array([b[0] for b in bounds])

fitness[i] = obj\_function(nests[i])

# Return the best solution best\_idx = np.argmin(fitness)

return nests[best\_idx], fitness[best\_idx]

# Example usage: Minimize f(x) = x^2 def objective(x):

return sum(xi\*\*2 for xi in x)

bounds = [(-10, 10), (-10, 10)] # 2D problem

best\_solution, best\_fitness = cuckoo\_search(objective, bounds) print("Best Solution:", best\_solution)

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

## Program 5

Grey Wolf Optimizer (GWO)

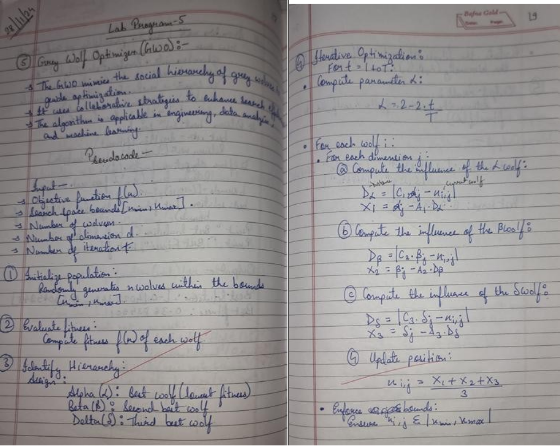
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses

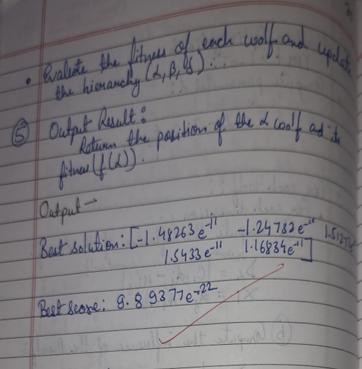
these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

#Grey Wolf Optimizer (GWO) import numpy as np

def objective\_function(x):

"""Example objective function: Sphere function.""" return sum(x\*\*2)

def initialize\_population(dim, n\_wolves, bounds):

"""Initialize the positions of the wolves randomly within the given bounds.""" return np.random.uniform(bounds[0], bounds[1], (n\_wolves, dim))

def gwo(objective\_function, bounds, dim, n\_wolves, n\_iterations): # Initialize population

wolves = initialize\_population(dim, n\_wolves, bounds) fitness = np.apply\_along\_axis(objective\_function, 1, wolves)

# Initialize alpha, beta, and delta

alpha, beta, delta = np.argsort(fitness)[:3]

alpha\_pos, alpha\_score = wolves[alpha], fitness[alpha] beta\_pos, beta\_score = wolves[beta], fitness[beta] delta\_pos, delta\_score = wolves[delta], fitness[delta]

# Main optimization loop

for iteration in range(n\_iterations):

a = 2 - 2 \* (iteration / n\_iterations) # Linearly decreasing a

for i in range(n\_wolves): for j in range(dim):

# Update each wolf's position

r1, r2 = np.random.rand(), np.random.rand() A1, C1 = 2 \* a \* r1 - a, 2 \* r2

D\_alpha = abs(C1 \* alpha\_pos[j] - wolves[i, j])

X1 = alpha\_pos[j] - A1 \* D\_alpha

r1, r2 = np.random.rand(), np.random.rand() A2, C2 = 2 \* a \* r1 - a, 2 \* r2

D\_beta = abs(C2 \* beta\_pos[j] - wolves[i, j])

X2 = beta\_pos[j] - A2 \* D\_beta

r1, r2 = np.random.rand(), np.random.rand() A3, C3 = 2 \* a \* r1 - a, 2 \* r2

D\_delta = abs(C3 \* delta\_pos[j] - wolves[i, j])

X3 = delta\_pos[j] - A3 \* D\_delta

# Average position update wolves[i, j] = (X1 + X2 + X3) / 3.0

# Enforce bounds

wolves[i, :] = np.clip(wolves[i, :], bounds[0], bounds[1])

# Evaluate fitness and update alpha, beta, delta

fitness = np.apply\_along\_axis(objective\_function, 1, wolves) sorted\_indices = np.argsort(fitness)

alpha, beta, delta = sorted\_indices[:3]

alpha\_pos, alpha\_score = wolves[alpha], fitness[alpha] beta\_pos, beta\_score = wolves[beta], fitness[beta] delta\_pos, delta\_score = wolves[delta], fitness[delta]

return alpha\_pos, alpha\_score # Example usage

dim = 5 # Number of dimensions

bounds = (-10, 10) # Search space bounds n\_wolves = 30 # Number of wolves n\_iterations = 100 # Number of iterations

best\_solution, best\_score = gwo(objective\_function, bounds, dim, n\_wolves, n\_iterations) print(f"Best solution: {best\_solution}")

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

## Program 6

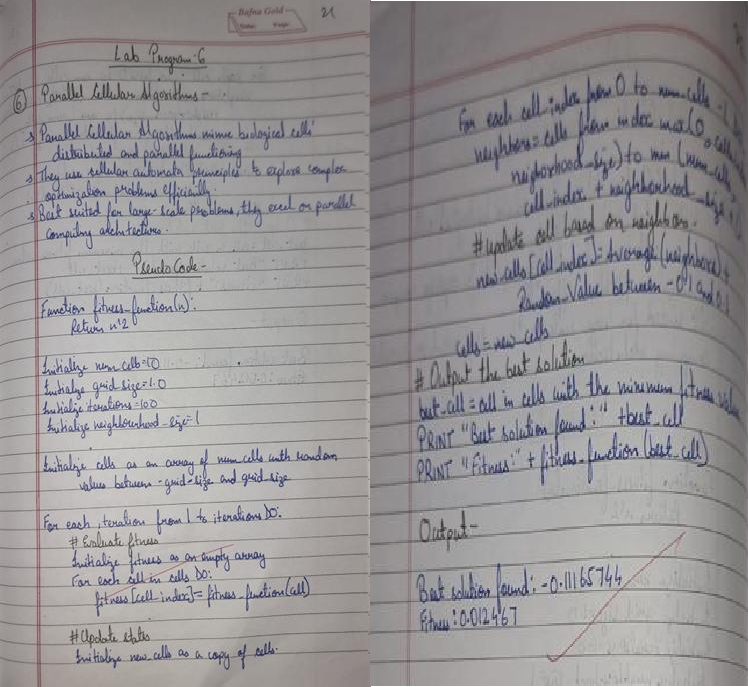
Parallel Cellular Algorithms and Programs

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

#Parallel Cellular Algorithms and Programs import numpy as np

# Define the optimization function def fitness\_function(x):

return x\*\*2

# Initialize parameters num\_cells = 10

grid\_size = 1.0

iterations = 100

neighborhood\_size = 1

# Initialize population

cells = np.random.uniform(-grid\_size, grid\_size, num\_cells)

# Main loop

for \_ in range(iterations): # Evaluate fitness

fitness = np.array([fitness\_function(cell) for cell in cells])

# Update states

new\_cells = np.copy(cells) for i in range(num\_cells):

# Get neighbors

neighbors = cells[max(0, i-neighborhood\_size):min(num\_cells, i+neighborhood\_size+1)] # Update cell based on neighbors

new\_cells[i] = np.mean(neighbors) + np.random.uniform(-0.1, 0.1) # Add some noise cells = new\_cells

# Output the best solution

best\_cell = cells[np.argmin(fitness)] print(f"Best solution found: {best\_cell}") print(f"Fitness: {fitness\_function(best\_cell)}")

Output:



## Program 7

Optimization via Gene Expression Algorithms

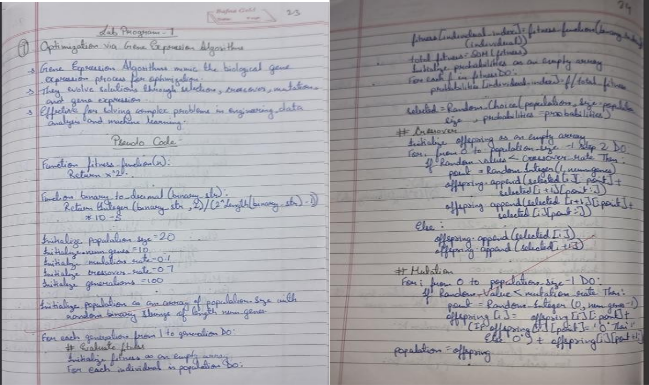
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar

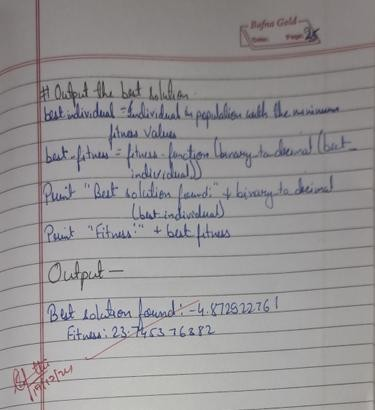
to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

#Optimization via Gene Expression Algorithms import numpy as np

# Define the optimization function def fitness\_function(x):

return x\*\*2

# Convert binary string to decimal def binary\_to\_decimal(binary\_str):

return int(binary\_str, 2) / (2\*\*len(binary\_str) - 1) \* 10 - 5 # Scale to [-5, 5]

# Initialize parameters population\_size = 20

num\_genes = 10

mutation\_rate = 0.1

crossover\_rate = 0.7

generations = 100

# Initialize population

population = [''.join(np.random.choice(['0', '1'], num\_genes)) for \_ in range(population\_size)]

# Main loop

for \_ in range(generations): # Evaluate fitness

fitness = [fitness\_function(binary\_to\_decimal(ind)) for ind in population]

# Selection (roulette wheel) total\_fitness = sum(fitness)

probabilities = [f / total\_fitness for f in fitness]

selected = np.random.choice(population, size=population\_size, p=probabilities)

# Crossover offspring = []

for i in range(0, population\_size, 2):

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

point = np.random.randint(1, num\_genes) offspring.append(selected[i][:point] + selected[i+1][point:]) offspring.append(selected[i+1][:point] + selected[i][point:])

else:

offspring.append(selected[i]) offspring.append(selected[i+1])

# Mutation

for i in range(population\_size):

if np.random.rand() < mutation\_rate: point = np.random.randint(num\_genes)

offspring[i] = offspring[i][:point] + ('1' if offspring[i][point] == '0' else '0') + offspring[i][point+1:]

population = offspring # Output the best solution

best\_individual = min(population, key=lambda ind: fitness\_function(binary\_to\_decimal(ind))) best\_fitness = fitness\_function(binary\_to\_decimal(best\_individual))

print(f"Best solution found: {binary\_to\_decimal(best\_individual)}") print(f" Fitness: {best\_fitness}")

Output:

