VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Chethan N(1BM23CS075)

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
Aug-2025 to Dec-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 **Chethan N (1BM23CS075),** 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.

Raghavendra C K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GitHub Link:
https://github.com/chethannhub/BIS_Lab

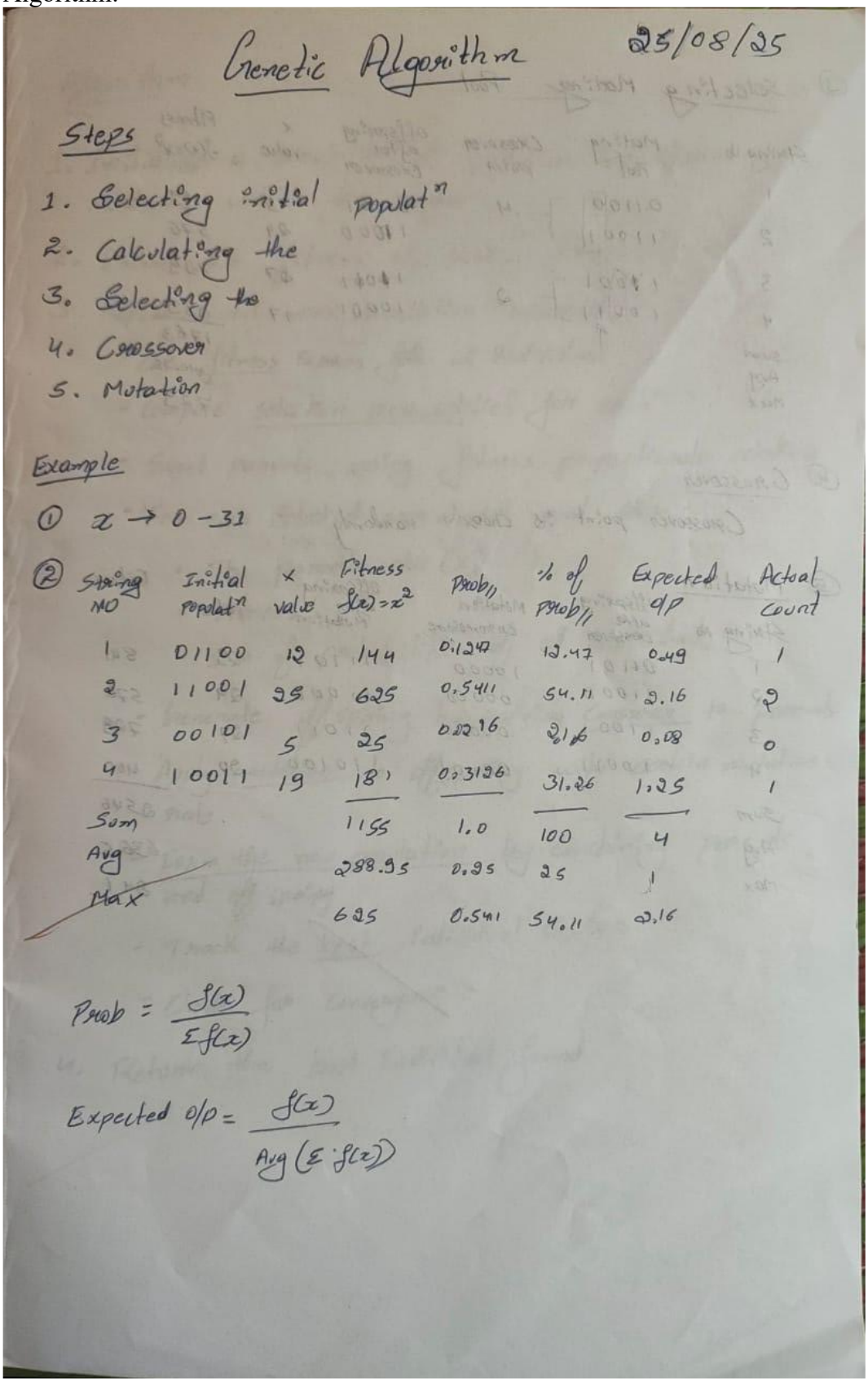
Program 1

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:



Genetic Algorithm 25/08/25

Steps

1. Selecting initial populat^{n}
2. Calculating the
3. Selecting the
4. Crossover
5. Mutation

Example

① $x \rightarrow 0 - 31$

②

String NO	Initial Populat^{n}	x value	Fitness $f(x) = x^2$	Prob$_{i}$	% of Prob$_{i}$	Expected o/p	Actual count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.16	2
3	00101	5	25	0.0216	2.16	0.08	0
4	10011	19	361	0.3126	31.26	1.25	1
Sum			1155	1.0	100	4	
Avg			288.75	0.25	25	1	
Max			625	0.5411	54.11	2.16	

$$Prob = \frac{f(x)}{\Sigma f(x)}$$

$$Expected\ o/p = \frac{f(x)}{Avg\ (\Sigma f(x))}$$

③ Selecting Mating Pool

String No	Mating Pool	Crossover point	Offspring after Crossover	x value	Fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
					1763
Sum					440.75
Avg					729
Max					

④ Crossover

Crossover point is chosen randomly

⑤ Mutation

String No	offspring after Crossover	Mutation Chromosome	Offspring after Mutation	x value	Fitness
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11001	00000	11011	27	729
4	10011	00101	10100	20	400
Sum					2546
Avg					636.5
Max					841

Algorithm

1. Initialize a population of pop-size individuals randomly within the range
2. Evaluate the fitness of each: $fitness(x) = x^2$
3. Repeat until max generation reached:
   - Calc fitness scores for all individual
   - Compute selection probabilities for each
   - Select parents using fitness proportionate selection
   - For each selected parent, calc and store:
     - The parent's value (x)
     - Fitness
     - Sum of probabilities of all individuals
   - Generate offspring by applying Crossover to parents
   - Apply mutation to offspring with certain mutation rate
   - Form the new population by combining parents and offspring
   - Track the best individual so far
   - Check for convergence
4. Return the best individual found

Output

Generation 1:

parent 1: x=31, Fitness= 961 Sele. prob. = 0.4433
parent 2: x=24 Fitness = 576 = 0.1328
parent 3: x=30 Fitness = 729 0.1681
parent 4: x=30 Fitness: 576 0.1328
Best individual so far: x = 31 Fitness 961

Generation 2:

Parent	X	Fitness	Sel. prob
1	31	961	0.3805
2	24	576	0.1521
3	27	729	0.3849
4	24	576	0.1521

Best individual so far X=31 : Fitness = 961

Generation 3:

Parent	X	Fitness	Sel. prob.
1	24	576	0.4809
2	31	961	0.4079
3	31	961	0.4079
4	24	576	0.4890

Best individual so far x=31, Fitness =961

Code:

```
import random
import math

# ------------------------------
# CONFIG
NUM_CITIES = 10
POP_SIZE = 100
GENERATIONS = 20
MUTATION_RATE = 0.02

# ------------------------------
# Generate random cities
cities = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(NUM_CITIES)]

# ------------------------------
# Distance between two cities
def distance(city1, city2):
    return math.hypot(city1[0] - city2[0], city1[1] - city2[1])

# Total path length (fitness is inverse)
def total_distance(tour):
    return sum(distance(cities[tour[i]], cities[tour[(i+1) % NUM_CITIES]]) for i in
range(NUM_CITIES))

# ------------------------------
# Create random individual (a tour)
def create_individual():
    tour = list(range(NUM_CITIES))
    random.shuffle(tour)
    return tour

# Crossover: Order Crossover (OX)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES

    # Copy slice from first parent
    child[start:end+1] = parent1[start:end+1]

    # Fill in the rest from second parent
    ptr = 0
    for city in parent2:
        if city not in child:
            while child[ptr] is not None:
                ptr += 1
            child[ptr] = city
```

```
    return child

# Mutation: Swap two cities
def mutate(tour):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
        tour[i], tour[j] = tour[j], tour[i]
    return tour

# ----------------------------
# Initial population
population = [create_individual() for _ in range(POP_SIZE)]

# ----------------------------
# Main GA loop
for gen in range(GENERATIONS):
    # Evaluate fitness
    scored = [(ind, total_distance(ind)) for ind in population]
    scored.sort(key=lambda x: x[1])  # lower distance is better
    best = scored[0]

    print(f"Generation {gen+1}: Best distance = {best[1]:.2f}")

    # Selection: keep top 50%
    selected = [ind for ind, _ in scored[:POP_SIZE // 2]]

    # Reproduce
    children = []
    while len(children) < POP_SIZE:
        p1, p2 = random.sample(selected, 2)
        child = crossover(p1, p2)
        child = mutate(child)
        children.append(child)

    population = children

# Final best route
best_tour = scored[0][0]
print("\nBest tour found:", best_tour)
```

Program 2

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:

25/8/25

Gene Expression

1. Initialise the parameters
   are population size, no. of generations, crossover rate, mutation rate, distance matrix
2. Generate initial population
   Create a random initial population of sol^n (tours)
   And Convert linear chromosomes into valid tours
3. Evaluate fitness
   For each individual in the population
   - Calc, the total distance of the tour (fitness)
   - Shorter distances = higher fitness
4. Repeat for max generations:
   - select individuals based on fitness
   - Create offspring by combining parents using methods like crossover
   - with a set probability, mutate offspring by swapping two cities
   - Compute fitness of the new offspring
   - Combine parents and offspring, then select the best individual for the next gen
5. Return the best tour found with the shortest distance

Output

Generation 1:

Tour: [1, 3, 2, 0] Dis = 80

Tour: [1, 2, 3, 0] Dis = 95

Tour: [1, 3, 2, 0] Dis = 80

Tour: [1, 0, 2, 3] Dis = 80

Best Min Dist = 80

Generation 2:

Tour: [2, 3, 0, 1] Dis = 95

Tour: [3, 1, 2, 0] Dis = 95

Tour: [1, 3, 2, 0] Dis = 80

Tour: [1, 2, 3, 0] Dis = 80

Min Dist = 80

Code:

```
import numpy as np
import random
import operator

# Generate noisy data from a nonlinear function
def target_function(x):
    return 2 * x**3 - x + 1

def generate_data(n_points=30):
    xs = np.linspace(-2, 2, n_points)
    ys = np.array([target_function(x) for x in xs]) + np.random.normal(0, 1, n_points)
    return xs, ys

X_data, y_data = generate_data()

# Parameters
POP_SIZE = 50
GENE_LENGTH = 20
NUM_GENERATIONS = 10
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7

# Function and terminal sets
FUNCTIONS = [('+', operator.add), ('-', operator.sub), ('*', operator.mul)]
TERMINALS = ['x', '1']
func_dict = {f[0]: f[1] for f in FUNCTIONS}

# Convert gene (list of symbols) into executable function
def express_gene(gene):
    stack = []
    for symbol in gene:
        if symbol in func_dict:
            try:
                b = stack.pop()
                a = stack.pop()
                func = func_dict[symbol]
                stack.append(lambda x, a=a, b=b, func=func: func(a(x), b(x)))
            except IndexError:
                stack.append(lambda x: 1)
        elif symbol == 'x':
            stack.append(lambda x: x)
        elif symbol == '1':
            stack.append(lambda x: 1)
        else:
            stack.append(lambda x: 1)
    return stack[-1] if stack else lambda x: 1
```

```
# Fitness: negative MSE
def fitness(gene):
    func = express_gene(gene)
    try:
        ys_pred = np.array([func(x) for x in X_data])
        mse = np.mean((ys_pred - y_data) ** 2)
        if np.isnan(mse) or np.isinf(mse):
            return -float('inf')
        return -mse
    except Exception:
        return -float('inf')

# Tournament selection
def selection(pop, k=3):
    selected = random.sample(pop, k)
    return max(selected, key=fitness)

# Crossover
def crossover(parent1, parent2):
    if len(parent1) != len(parent2):
        return parent1[:]
    point = random.randint(1, len(parent1) - 2)
    return parent1[:point] + parent2[point:]

# Mutation
def mutate(gene, mutation_rate=MUTATION_RATE):
    gene = gene[:]
    symbols = [f[0] for f in FUNCTIONS] + TERMINALS
    for i in range(len(gene)):
        if random.random() < mutation_rate:
            gene[i] = random.choice(symbols)
    return gene

# Initialize population
population = [[random.choice([f[0] for f in FUNCTIONS] + TERMINALS) for _ in
range(GENE_LENGTH)] for _ in range(POP_SIZE)]

best_gene = None
best_fit = -float('inf')

# Main loop
for gen in range(NUM_GENERATIONS):
    new_population = []
    for _ in range(POP_SIZE):
        parent1 = selection(population)
        parent2 = selection(population)
```

```
        child = crossover(parent1, parent2) if random.random() < CROSSOVER_RATE else parent1[:]
        child = mutate(child)
        new_population.append(child)

    population = new_population

    generation_best = max(population, key=fitness)
    generation_best_fit = fitness(generation_best)

    if generation_best_fit > best_fit:
        best_fit = generation_best_fit
        best_gene = generation_best

    if gen % 10 == 0 or gen == NUM_GENERATIONS - 1:
        print(f"Generation {gen}: Best fitness = {best_fit:.6f}")

# Output best gene and prediction results
print("Best gene (symbolic expression):", best_gene)
best_func = express_gene(best_gene)
print("Sample predictions vs actual values:")
for x, y_true in zip(X_data[:10], y_data[:10]):
    try:
        y_pred = best_func(x)
        print(f"x={x:.3f}, Predicted={y_pred:.3f}, Actual={y_true:.3f}")
    except Exception:
        print(f"x={x:.3f}, Predicted=Error, Actual={y_true:.3f}")
```

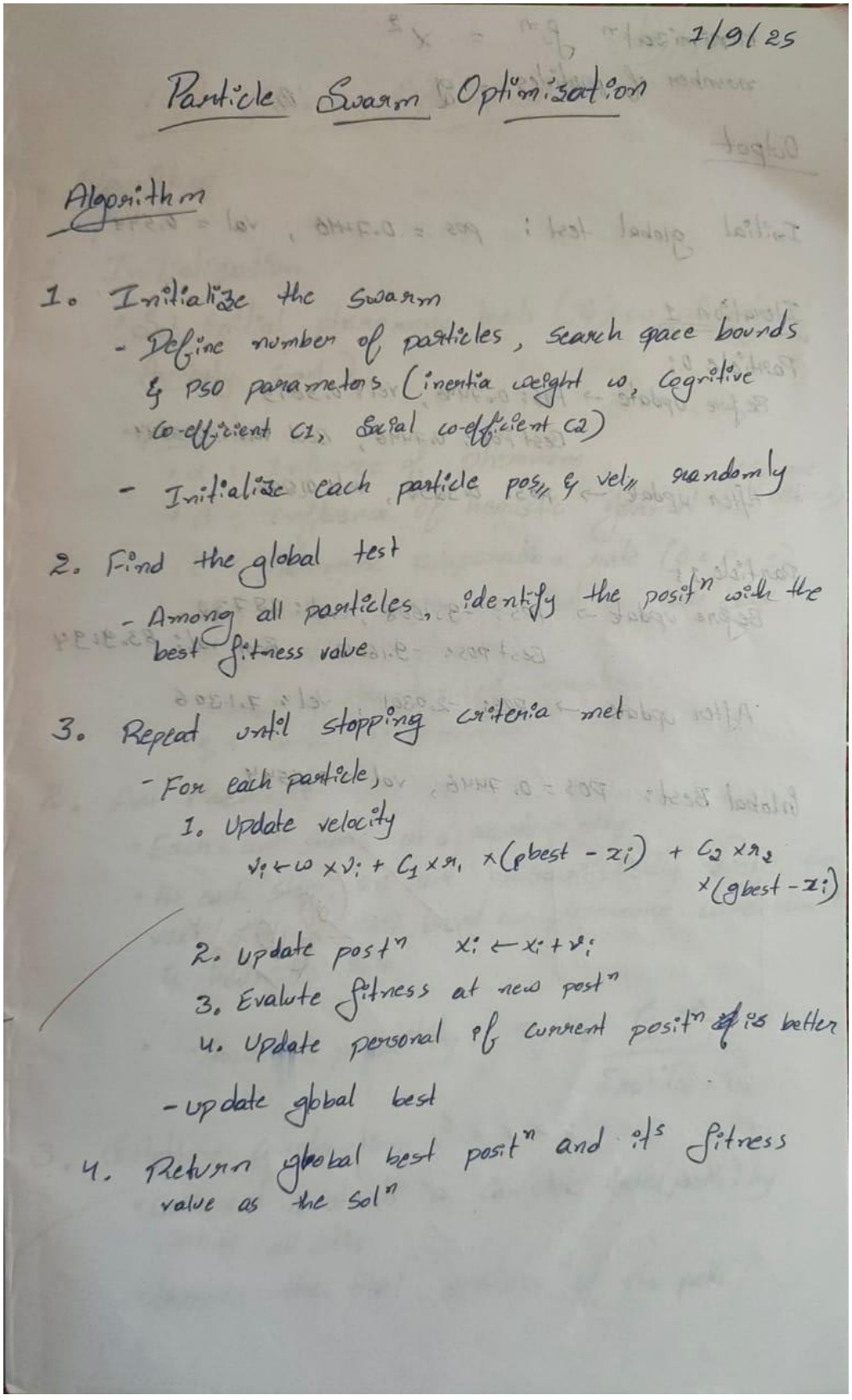
Program 3

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:



7/9/25

Particle Swarm Optimisation

Algorithm

1. Initialize the swarm
   - Define number of particles, search space bounds & PSO parameters (inertia weight w, cognitive co-efficient c_1, Social co-efficient c_2)
   - Initialize each particle pos^n & vel^n randomly
2. Find the global best
   - Among all particles, identify the posit^n with the best fitness value
3. Repeat until stopping criteria met
   - For each particle,
     1. Update velocity

$$v_i \leftarrow w \times v_i + c_1 \times r_1 \times (pbest - x_i) + c_2 \times r_2 \times (gbest - x_i)$$

     2. update post^n $x_i \leftarrow x_i + v_i$
     3. Evalute fitness at new post^n
     4. Update personal if current posit^n is better
   - update global best
4. Return global best posit^n and its fitness value as the sol^n

Minimization fun = x^2

number of particles = 2

Output

Initial global test : pos = 0.7446 , val = 0.544

Iteration 1

Particle 0:

Before update → Pos: 0.7446, vel: 0.3613

Best Pos: 0.7446, Best val: 0.5544

After update → pos: 0.9252, vel: 0.1806

Particle 1:

Before update → pos: -9.1608, vel: 9.8732

Best pos: -9.1608 Best val: 83.9194

After update → pos: -2.0301, vel: 7.1306

Global Best: pos = 0.7446, val = 0.5544

Code:

```
import numpy as np

def de_jong(position):
    x, y = position
    return x**2 + y**2

num_particles = 30
dimensions = 2
iterations = 10
w = 0.5
c1 = 1.5
c2 = 1.5
bounds = (-5.12, 5.12)

positions = np.random.uniform(bounds[0], bounds[1], (num_particles, dimensions))
velocities = np.random.uniform(-1, 1, (num_particles, dimensions))
pbest_positions = np.copy(positions)
pbest_scores = np.array([de_jong(p) for p in positions])

gbest_index = np.argmin(pbest_scores)
gbest_position = pbest_positions[gbest_index]
gbest_score = pbest_scores[gbest_index]

for t in range(iterations):
    for i in range(num_particles):
        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)

        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (pbest_positions[i] - positions[i]) +
                         c2 * r2 * (gbest_position - positions[i]))

        positions[i] += velocities[i]
        positions[i] = np.clip(positions[i], bounds[0], bounds[1])

        fitness = de_jong(positions[i])

        if fitness < pbest_scores[i]:
            pbest_positions[i] = positions[i]
            pbest_scores[i] = fitness

            if fitness < gbest_score:
                gbest_position = positions[i]
                gbest_score = fitness

    print(f"Iteration {t+1:3d} | Best Score: {gbest_score:.6f}")
```

```
print("\nBest Solution Found:")
print(f"Position: x = {gbest_position[0]:.6f}, y = {gbest_position[1]:.6f}")
print(f"Score: {gbest_score:.10f}")
```

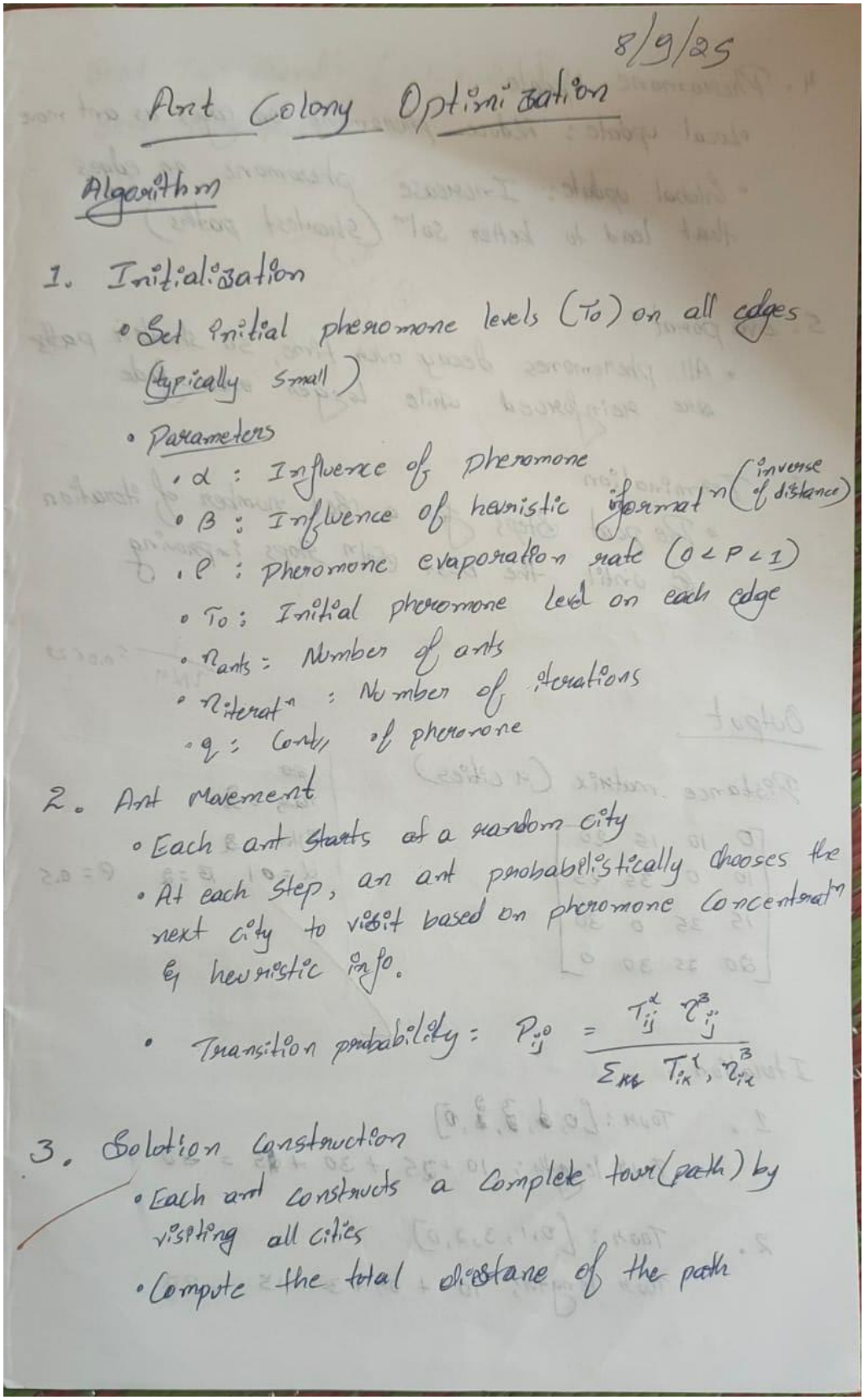
Program 4

Ant Colony Optimization for the Traveling Salesman Problem:

The foraging behavior 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:



8/9/25

Ant Colony Optimization

Algorithm

1. Initialization
   - Set initial pheromone levels (T_0) on all edges (typically small)
   - Parameters
     - α : Influence of pheromone
     - β : Influence of heuristic inform^n (inverse of distance)
     - ρ : Pheromone evaporation rate ($0 < \rho < 1$)
     - T_0 : Initial pheromone level on each edge
     - n_{ants} : Number of ants
     - n_{iterat^n} : Number of iterations
     - q : Const^n of pheromone
2. Ant Movement
   - Each ant starts at a random city
   - At each step, an ant probabelistically chooses the next city to visit based on pheromone concentrat^n & heuristic info.
   - Transition probability : $P_{ij} = \frac{T_{ij}^{\alpha} \eta_{ij}^{\beta}}{\sum_{k \in} T_{ik}^{\alpha} \eta_{ik}^{\beta}}$
3. Solution Construction
   - Each ant constructs a complete tour (path) by visiting all cities
   - Compute the total distance of the path

4. Pheromone Update
   - Local update: Reduce pheromone on edges as ant move
   - Global update: Increase pheromone on edges that lead to better soln (shortest paths)

5. Evaporatn
   - All pheromones decay over time, so shorter paths are reinforced while longer ones fade

6. Termination
   - Repeat steps for a set number of iteration
   - until the best soln stops improving

1/4×4 = 0.0625

Output

Distance matrix (4 cities)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

num = 4
n_iter = 2
α = 0.1 β = 2 ρ = 0.5

Iteration :

1. Tour : [0, 1, 3, 2, 0]
   Tour length : 10 + 25 + 30 + 15 = 80
2. Tour : [0, 1, 3, 2, 0]
   Tour length : 10 + 25 + 30 + 15 = 80

Best Tour Found : [0, 1, 3, 2, 0]
Best Tour length : 80

Code:

```
import numpy as np
import random

class ACO_TSP:
    def __init__(self, distances, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, Q=100):
        self.distances = distances
        self.n_cities = distances.shape[0]
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        self.pheromone = np.ones((self.n_cities, self.n_cities))

    def run(self):
        best_length = float("inf")
        best_path = None

        for iteration in range(self.n_iterations):
            paths = []
            lengths = []

            for ant in range(self.n_ants):
                path = self.construct_solution()
                length = self.calculate_length(path)
                paths.append(path)
                lengths.append(length)

                if length < best_length:
                    best_length = length
                    best_path = path

            self.update_pheromones(paths, lengths)
            print(f"Iteration {iteration+1}: Best Length = {best_length}")

        return best_path, best_length

    def construct_solution(self):
        start = random.randint(0, self.n_cities - 1)
        path = [start]
        visited = {start}

        for _ in range(self.n_cities - 1):
            current = path[-1]
            next_city = self.choose_next_city(current, visited)
```

```
            path.append(next_city)
            visited.add(next_city)

        return path

    def choose_next_city(self, current, visited):
        probabilities = []
        for city in range(self.n_cities):
            if city not in visited:
                tau = self.pheromone[current][city] ** self.alpha
                eta = (1 / self.distances[current][city]) ** self.beta
                probabilities.append((city, tau * eta))
            else:
                probabilities.append((city, 0))

        total = sum(prob for _, prob in probabilities)
        r = random.random() * total
        cumulative = 0
        for city, prob in probabilities:
            cumulative += prob
            if cumulative >= r:
                return city

    def calculate_length(self, path):
        length = 0
        for i in range(len(path) - 1):
            length += self.distances[path[i]][path[i+1]]
        length += self.distances[path[-1]][path[0]]
        return length

    def update_pheromones(self, paths, lengths):
        self.pheromone *= (1 - self.rho)   # evaporation
        for path, length in zip(paths, lengths):
            deposit = self.Q / length
            for i in range(len(path) - 1):
                self.pheromone[path[i]][path[i+1]] += deposit
                self.pheromone[path[i+1]][path[i]] += deposit


if __name__ == "__main__":
    distances = np.array([
        [0, 2, 9, 10, 1],
        [2, 0, 6, 4, 3],
        [9, 6, 0, 8, 5],
        [10, 4, 8, 0, 7],
        [1, 3, 5, 7, 0]
    ])
```

```
aco = ACO_TSP(distances, n_ants=10, n_iterations=10)
best_path, best_length = aco.run()
print("\nBest Path:", best_path)
print("Best Length:", best_length)
```

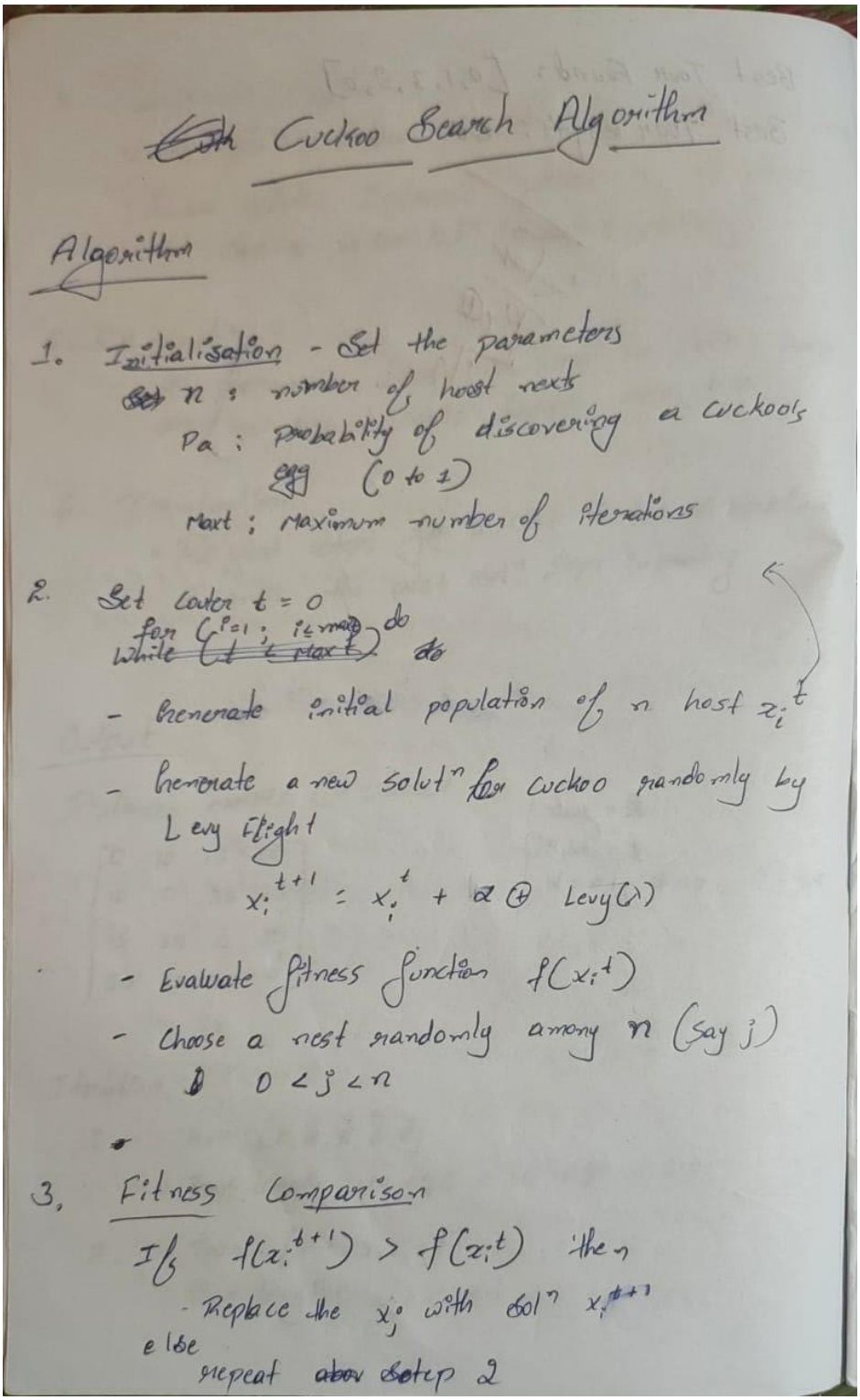
Program 5

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:



Cuckoo Search Algorithm

Algorithm

1. Initialisation - Set the parameters
   n : number of host nests
   Pa : probability of discovering a cuckoos egg (0 to 1)
   Maxt : Maximum number of iterations

2. Set counter t = 0
   for (i=1; i<max t) do
   - Generate initial population of n host x_i^t
   - Generate a new solut^n for cuckoo randomly by Levy flight

     $x_i^{t+1} = x_i^t + \alpha \oplus Levy(\lambda)$

   - Evaluate fitness function $f(x_i^t)$
   - Choose a nest randomly among n (say j)
     $0 < j < n$

3. Fitness Comparison
   If $f(x_i^{t+1}) > f(x_j^t)$ then
   - Replace the x_j with sol^n x_i^{t+1}
   else
   repeat step 2

→ ~~else if~~ If Host next found the cuckoo egg
- abandon a fraction Pa of worst nest
- Build new nest at new location using levy flight a fraction Pa of worst nest
- keep the best solution

4) ~~Keep~~
- Rank the solution and find current best solution
- ~~Fi~~

5) Update the counter $t = t+1$ until ($t \geq$ MaxT)

6) Output the best next x_i

~~Prob~~ Knapsack application

Input
N = ~~10~~ 5 (no. of items)
weights = [10, 20, 30, 40, 50]
Profits = [60, 100, 120, 200, 250]
Capacity = 110

Parameters
n = 10
P_a = 0.25
maxt = 10

Best Solution (items selected) : [0, 1, 0, 1, 1]
Total profit : 550
Total weight : 110

Sol Rec 11/9

Code:

```
import numpy as np
import random

values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
n_items = len(values)
n_nests = 15
Pa = 0.25
max_iter = 100

def fitness(x):
    total_weight = np.dot(x, weights)
    if total_weight > capacity:
        return 0
    return np.dot(x, values)

def repair(x):
    while np.dot(x, weights) > capacity:
        idx = random.choice([i for i in range(n_items) if x[i]])
        x[idx] = 0
    return x

def levy_flight(Lambda=1.5):
    return np.random.normal(0, Lambda, n_items)

nests = [np.random.randint(0, 2, n_items) for _ in range(n_nests)]
nest_scores = [fitness(repair(x.copy())) for x in nests]

for generation in range(max_iter):
    for i in range(n_nests):
        new_nest = nests[i].copy()
        step = levy_flight()
        new_nest = np.abs(new_nest + step) > 0.5
        new_nest = new_nest.astype(int)
        new_nest = repair(new_nest)
        f_new = fitness(new_nest)
        if f_new > nest_scores[i]:
            nests[i] = new_nest
            nest_scores[i] = f_new

    indices = np.argsort(nest_scores)[:int(Pa * n_nests)]
    for idx in indices:
        nests[idx] = np.random.randint(0, 2, n_items)
        nests[idx] = repair(nests[idx])
        nest_scores[idx] = fitness(nests[idx])
```

```
best_idx = np.argmax(nest_scores)
print("Best solution:", nests[best_idx], "Value:", nest_scores[best_idx])
```

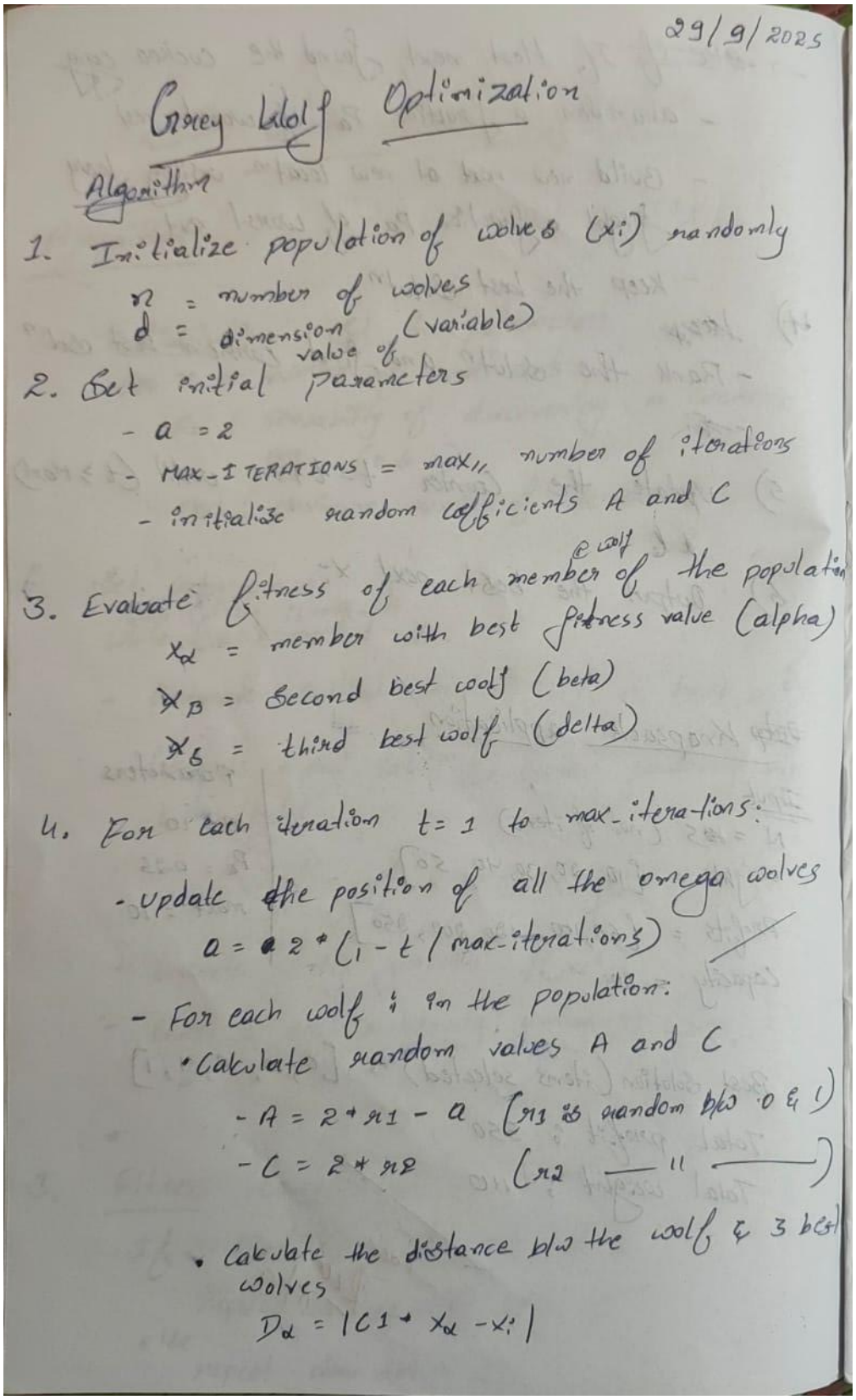
Program 6

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:



29/9/2025

Grey Wolf Optimization

Algorithm

1. Initialize population of wolves (x_i) randomly
   - n = number of wolves
   - d = dimension (variable)
2. Set initial parameters
   - $a = 2$
   - MAX_ITERATIONS = max no. number of iterations
   - initialize random coefficients A and C
3. Evaluate fitness of each member of the population
   - X_α = member with best fitness value (alpha)
   - X_β = second best wolf (beta)
   - X_δ = third best wolf (delta)
4. For each iteration $t = 1$ to max_iterations:
   - update the position of all the omega wolves
     $a = 2 * (1 - t / \text{max\_iterations})$
   - For each wolf i in the population:
     - Calculate random values A and C
       - $A = 2 * r_1 - a$ (r_1 is random b/w 0 & 1)
       - $C = 2 * r_2$ (r_2 — " —)
     - Calculate the distance b/w the wolf & 3 best wolves
       $D_\alpha = |C_1 * X_\alpha - X_i|$

$D_\beta = |C_2 * x_\beta - x_i|$

$D_\delta = |C_3 * x_\delta - x_i|$

- update the position of the wolfs
  - $x_1 = x_\alpha - A_1 * D_\alpha$

    $x_2 = x_\beta - A_2 * D_\beta$

    $x_3 = x_\delta - A_3 * D_\delta$

    $x_i = (x_1 + x_2 + x_3) / 3$

- update the fitness of all wolves after position change
- Reassign $x_\alpha, x_\beta, x_\alpha$ based on new fitness values

3. After all iterations, return x_α best solution found

~~Data~~ Input

Function : x^3

Problem dim = 1

Search bounds = (-2, 2)

iterations = 2

wolves = 4

Output

Iteration 1

wolf 1 : pos = [-1.407], Fitness = -2.909

wolf 2 : pos = [-0.3781] Fitness = -0.376

wolf 3 : pos = [-0.353] Fitness = -0.0441

wolf 4 : pos = [1.998] Fitness = 9.191

Code:

```
import numpy as np

class GreyWolfOptimizer:
    def __init__(self, func, lb, ub, dim, pop_size=30, max_iter=100):
        """
        :param func: Objective function to optimize
        :param lb: Lower bounds of the search space
        :param ub: Upper bounds of the search space
        :param dim: Number of dimensions (variables) in the search space
        :param pop_size: Number of wolves (population size)
        :param max_iter: Maximum number of iterations
        """
        self.func = func
        self.lb = lb
        self.ub = ub
        self.dim = dim
        self.pop_size = pop_size
        self.max_iter = max_iter

        self.position = np.random.uniform(low=self.lb, high=self.ub, size=(self.pop_size, self.dim))
        self.fitness = np.apply_along_axis(self.func, 1, self.position)

        self.alpha_pos = np.zeros(self.dim)
        self.alpha_score = float("inf")

        self.beta_pos = np.zeros(self.dim)
        self.beta_score = float("inf")

        self.delta_pos = np.zeros(self.dim)
        self.delta_score = float("inf")

    def optimize(self):
        for t in range(self.max_iter):
            for i in range(self.pop_size):

                fitness_val = self.func(self.position[i])
                if fitness_val < self.alpha_score:
                    self.alpha_score = fitness_val
                    self.alpha_pos = self.position[i]
                elif fitness_val < self.beta_score:
                    self.beta_score = fitness_val
                    self.beta_pos = self.position[i]
                elif fitness_val < self.delta_score:
                    self.delta_score = fitness_val
                    self.delta_pos = self.position[i]
```

```
            a = 2 - t * (2 / self.max_iter)
            for i in range(self.pop_size):
                A1 = 2 * a * np.random.random(self.dim) - a
                C1 = 2 * np.random.random(self.dim)
                D_alpha = np.abs(C1 * self.alpha_pos - self.position[i])
                X1 = self.alpha_pos - A1 * D_alpha

                A2 = 2 * a * np.random.random(self.dim) - a
                C2 = 2 * np.random.random(self.dim)
                D_beta = np.abs(C2 * self.beta_pos - self.position[i])
                X2 = self.beta_pos - A2 * D_beta

                A3 = 2 * a * np.random.random(self.dim) - a
                C3 = 2 * np.random.random(self.dim)
                D_delta = np.abs(C3 * self.delta_pos - self.position[i])
                X3 = self.delta_pos - A3 * D_delta

                self.position[i] = (X1 + X2 + X3) / 3

                self.position[i] = np.clip(self.position[i], self.lb, self.ub)

            print(f"Iteration {t + 1}/{self.max_iter}, Best Score: {self.alpha_score}")

        return self.alpha_pos, self.alpha_score

def sphere_function(x):
    return np.sum(x**2)


lower_bound = -5.0
upper_bound = 5.0
dim = 30
pop_size = 50
max_iter = 10


gwo = GreyWolfOptimizer(func=sphere_function, lb=lower_bound, ub=upper_bound, dim=dim,
pop_size=pop_size, max_iter=max_iter)


best_position, best_score = gwo.optimize()


print(f"Best Position: {best_position}")
print(f"Best Score: {best_score}")
```

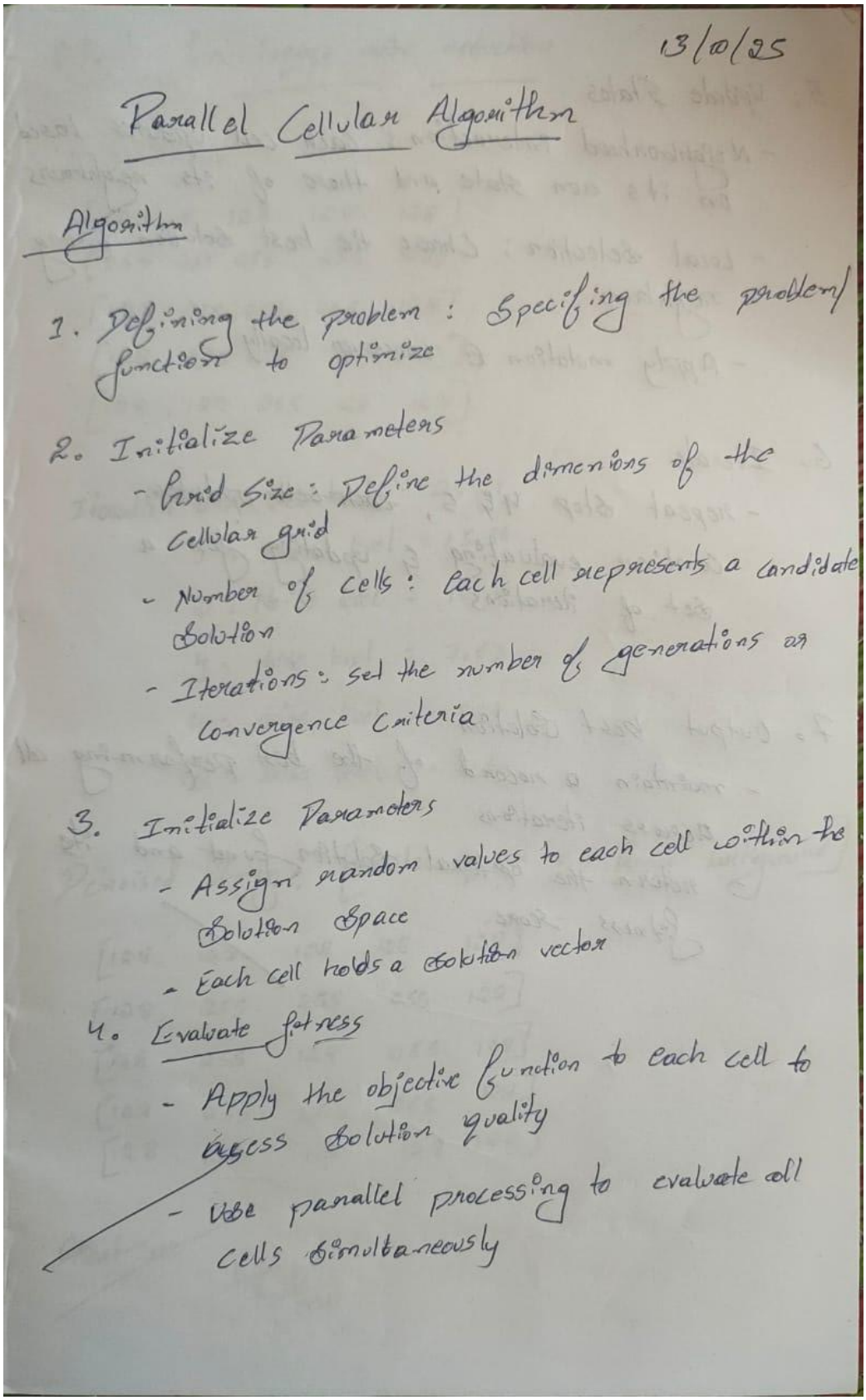
Program 7

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:



13/10/25

Parallel Cellular Algorithm

Algorithm

1. Defining the problem : Specifing the problem/ function to optimize

2. Initialize Parameters
   - Grid Size : Define the dimensions of the cellular grid
   - Number of cells : Each cell represents a candidate solution
   - Iterations : set the number of generations or convergence criteria

3. Initialize Parameters
   - Assign random values to each cell within the solution space
   - Each cell holds a solution vector

4. Evaluate fitness
   - Apply the objective function to each cell to assess solution quality
   - Use parallel processing to evaluate all cells simultaneously

5. Update States
   - Neighborhood interaction: Each cell updates based on its own state and those of its neighbours
   - Local Selection: Choose the best solution among neighbours
   - Apply mutation & crossover locally

6. Iterate
   - repeat step 4 & 5, ~~each cell updates~~ continue evaluating & updating for a set of iterations

7. Output best Solution
   - maintain a record of the best performing cell across iterations
   - return the optimal solution found and its fitness score.

Code:

```
import numpy as np
from multiprocessing import Pool

def edge_rule(subgrid):
    out = np.zeros_like(subgrid)
    rows, cols = subgrid.shape
    for i in range(1, rows-1):
        for j in range(1, cols-1):
            gx = (
                subgrid[i-1, j+1] + 2*subgrid[i, j+1] + subgrid[i+1, j+1] -
                subgrid[i-1, j-1] - 2*subgrid[i, j-1] - subgrid[i+1, j-1]
            )
            gy = (
                subgrid[i-1, j-1] + 2*subgrid[i-1, j] + subgrid[i-1, j+1] -
                subgrid[i+1, j-1] - 2*subgrid[i+1, j] - subgrid[i+1, j+1]
            )

            grad = np.sqrt(gx**2 + gy**2)
            out[i, j] = 255 if grad > 13 else 0

    return out

def get_chunks_with_overlap(img, num_workers):
    height = img.shape[0]
    chunk_size = height // num_workers
    chunks = []
    for i in range(num_workers):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i < num_workers - 1 else height

        if i > 0:
            start -= 1
        if i < num_workers - 1:
            end += 1

        chunks.append(img[start:end, :])
    return chunks

def parallel_edge_detection(img, num_workers=4):
    chunks = get_chunks_with_overlap(img, num_workers)
    with Pool(num_workers) as p:
        processed_chunks = p.map(edge_rule, chunks)


    results = []
    for i, chunk in enumerate(processed_chunks):
```

```
        if i > 0:
            chunk = chunk[1:]
        if i < num_workers - 1:
            chunk = chunk[:-1]
        results.append(chunk)

    return np.vstack(results)


if __name__ == "__main__":
    import imageio
    import matplotlib.pyplot as plt

    img = imageio.imread('path_to_image.jpg', mode='L')

    edges = parallel_edge_detection(img)

    plt.imshow(edges, cmap='gray')
    plt.show()
```

Output: