VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Biswajeet (1BM23CS069)

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 Jan-2026

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 Biswajeet Behera **(1BM23CS069),** 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 sir Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

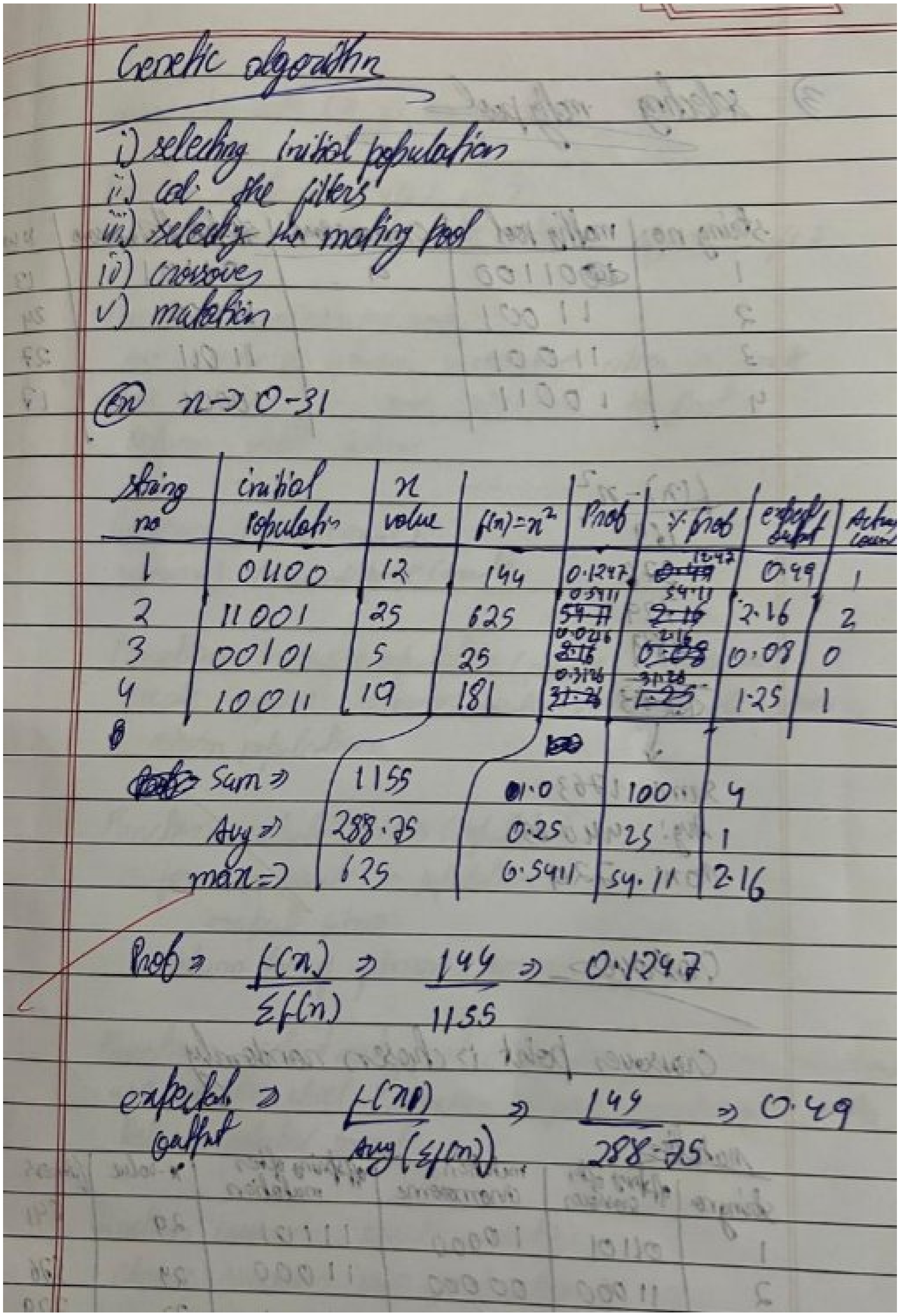
Index

Github Link:

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



Genetic algorithm

i) selecting initial population

ii) cal. the fitness

iii) selecting the mating pool

iv) crossover

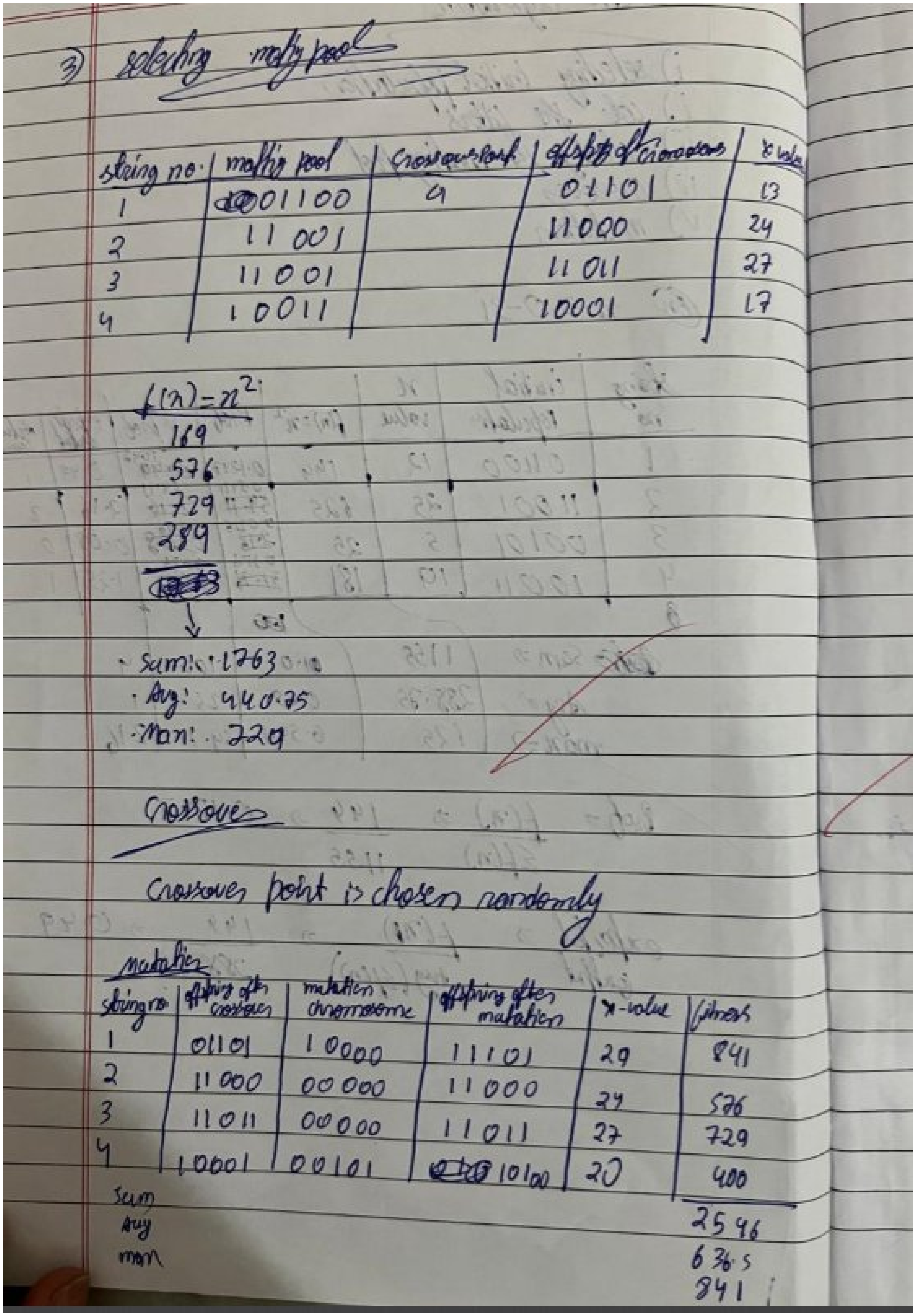
v) mutation

(Ex) $x \to 0-31$

string no	initial population	x value	$f(x)=x^2$	Prob	% Prob	expected output	Actual count
1	01100	12	144	0.1247	~~0.49~~ 12.47	0.49	1
2	11001	25	625	~~54.11~~ 0.5411	~~2.16~~ 54.11	2.16	2
3	00101	5	25	~~2.16~~ 0.0216	~~0.08~~ 2.16	0.08	0
4	10011	19	181	~~31.26~~ 0.3126	~~1.25~~ 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	

$$\text{Prob} \Rightarrow \frac{f(x)}{\Sigma f(x)} \Rightarrow \frac{144}{1155} \Rightarrow 0.1247$$

$$\text{expected output} \Rightarrow \frac{f(x_i)}{\text{Avg}(\Sigma f(x))} \Rightarrow \frac{144}{288.75} \Rightarrow 0.49$$



3) selecting mating pool

string no.	mating pool	crossover point	offspring after crossover	x value
1	01100	4	01101	13
2	11001		11000	24
3	11001		11011	27
4	10011		10001	17

$f(x) = x^2$:

169
576
729
289

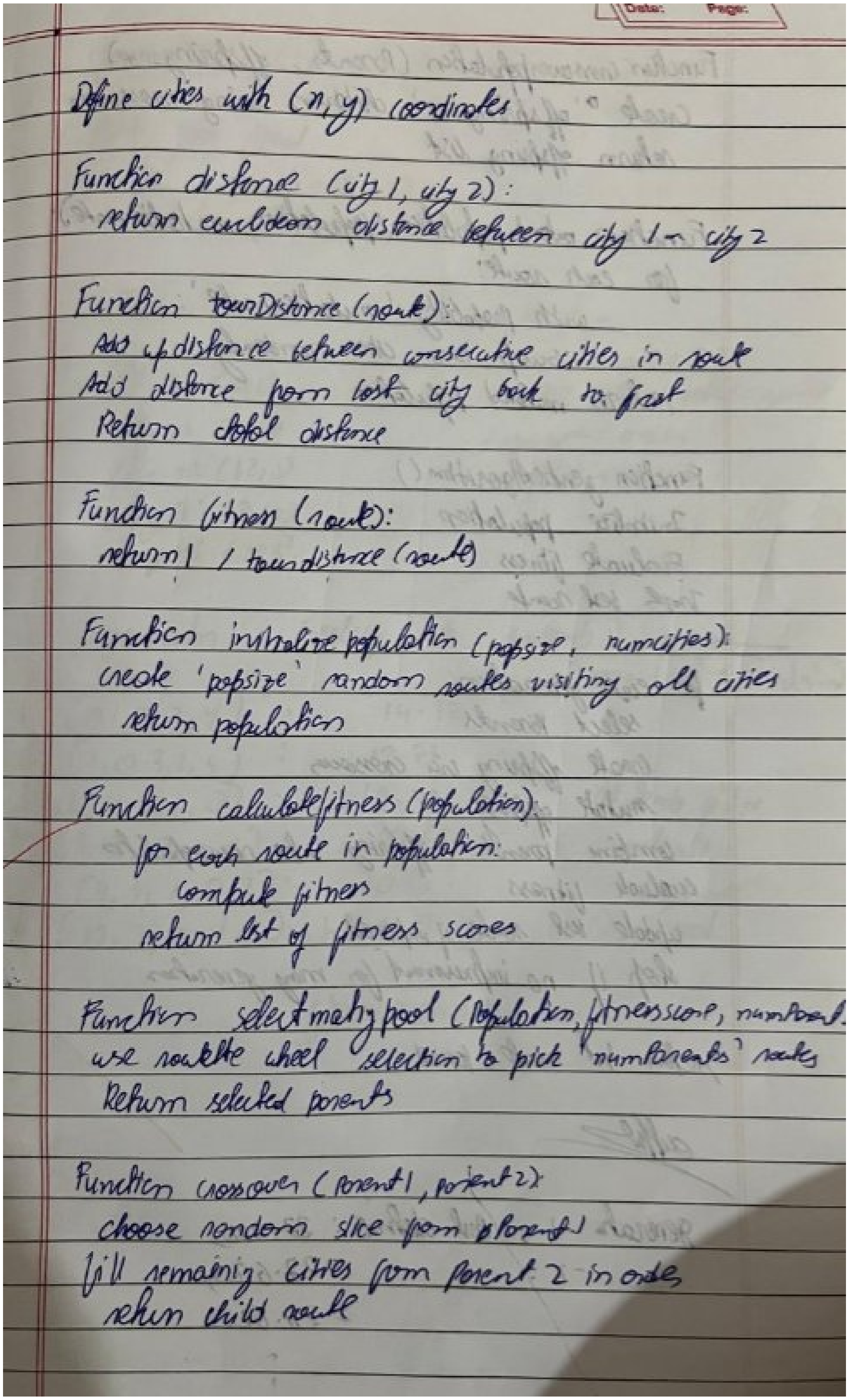
- Sum: 1763
- Avg: 440.75
- Max: 229

Crossover

Crossover point is chosen randomly

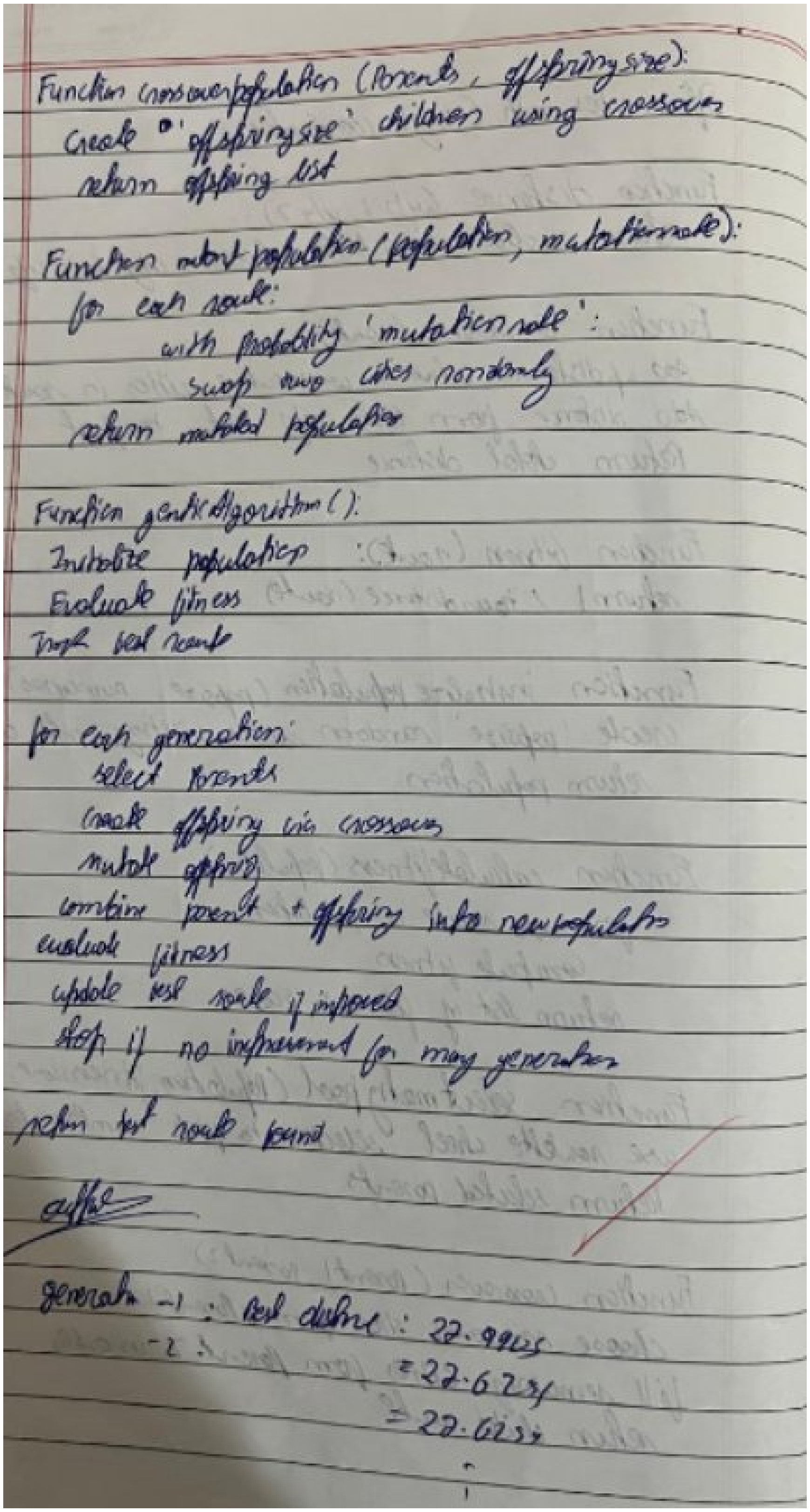
mutation

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	11011	00000	11011	27	729
4	10001	00101	10100	20	400
Sum					2546
Avg					636.5
max					841



Date: Page:

```
Define cities with (x, y) coordinates

Function distance (city1, city2):
  return euclidean distance between city1 & city2

Function tourDistance (route):
  Add up distance between consecutive cities in route
  Add distance from last city back to first
  Return total distance

Function fitness (route):
  return 1 / tourdistance (route)

Function initialize population (popsize, numcities):
  create 'popsize' random routes visiting all cities
  return population

Function calculatefitness (population):
  for each route in population:
    compute fitness
  return list of fitness scores

Function selectmatingpool (population, fitnessscore, numParent
  use roulette wheel selection to pick 'numParents' routes
  Return selected parents

Function crossover (parent1, parent2):
  choose random slice from parent1
  fill remaining cities from parent 2 in order
  return child route
```

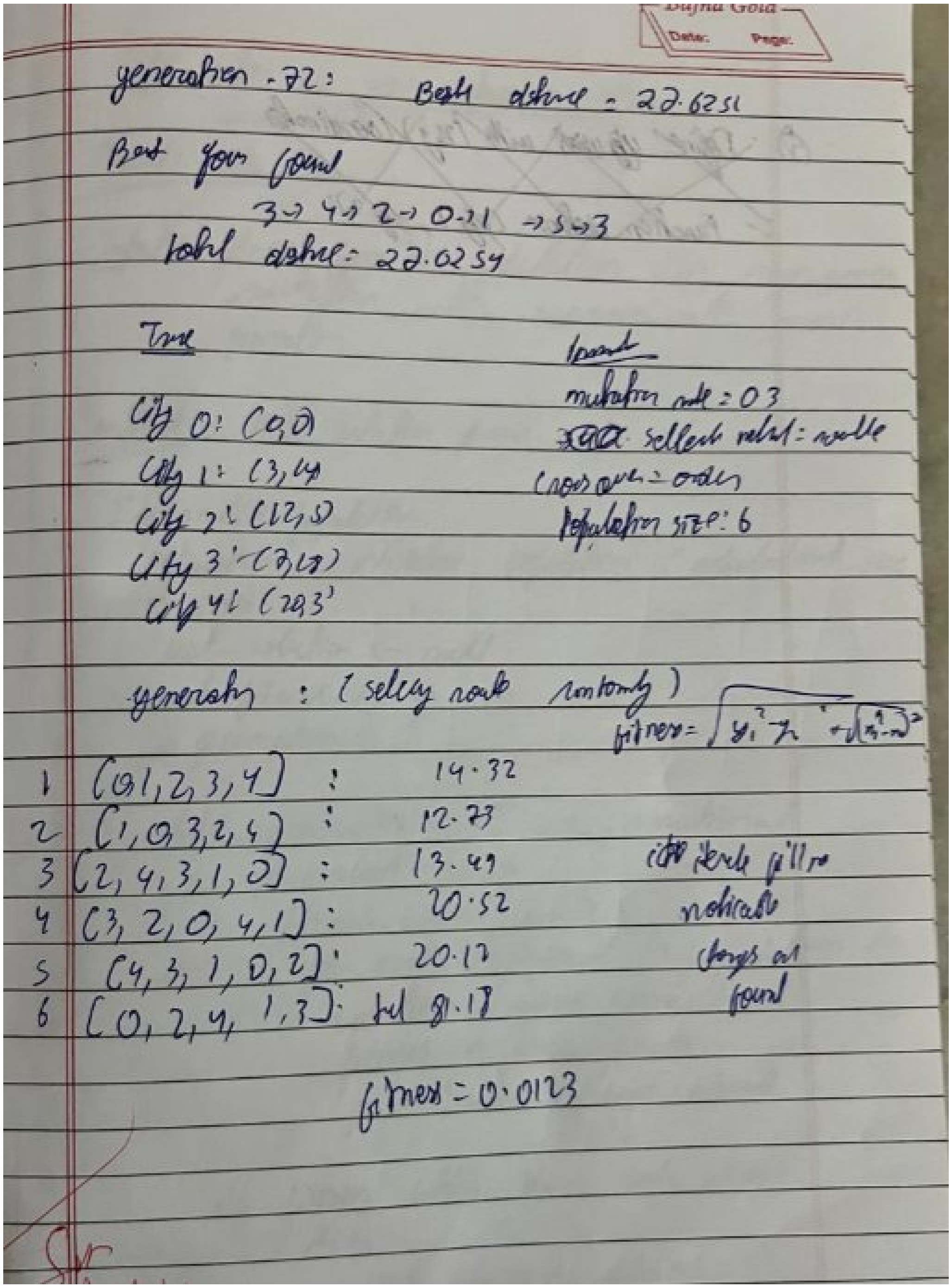


```
Function crossoverPopulation (Parents, offspringSize):
  Create 'offspringSize' children using crossover
   return offspring list

Function mutatePopulation (population, mutationRate):
  for each route:
        with probability 'mutationRate':
           swap two cities randomly
  return mutated population

Function geneticAlgorithm():
 Initialize population
 Evaluate fitness
Track best route

for each generation:
     select parents
    create offspring via crossover
    mutate offspring
   combine parents + offspring into new population
  evaluate fitness
  update best route if improved
   stop if no improvement for many generations

return best route found
```

generation -1 : best distance : 22.9925

= 22.6234

= 22.6234



Bajna Gold

Date: Page:

generation -72: Best distance = 22.6251

Best tour found

3→4→2→0→1 →3→3

total distance: 22.0254

True

City 0: (0,0)
City 1: (3,4)
City 2: (12,5)
City 3: (7,18)
City 4: (20,3)

Input

mutation rate: 0.3
~~[illegible]~~ selection method: roulette
crossover = order
Population size: 6

generation : (selecting route randomly)

fitness = $\sqrt{(y_i - x_i)^2 + (y_{i-1} - x_{i-1})^2}$

1	[0,1,2,3,4] :	14.32
2	[1,0,3,2,4] :	12.73
3	[2,4,3,1,0] :	13.49
4	[3,2,0,4,1] :	20.52
5	[4,3,1,0,2] :	20.17
6	[0,2,4,1,3] :	tel 8.18

city tends fill no noticable changes at round

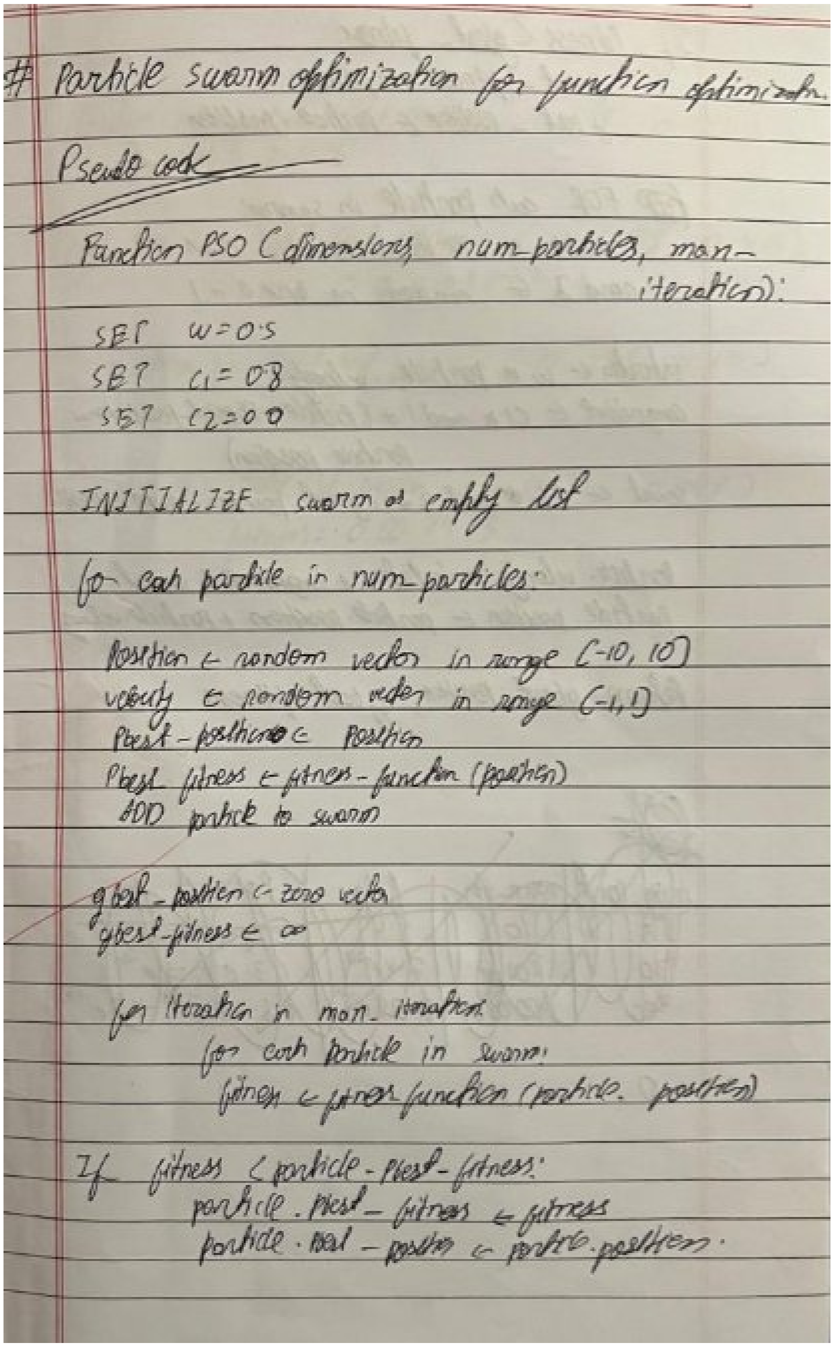
fitness = 0.0123

Code:

```
import random

def fitness_function(individual, w=5): x = individual["strength"]

    return -(x**2 - w*x + 3)

POP_SIZE = 20
N_GEN = 10
MUTATION_RATE = 0.1

TRAITS = ["strength"] def random_individual():
    return {trait: random.uniform(0, 10) for trait in TRAITS}

def crossover(parent1, parent2): child = {}
    for trait in TRAITS:
        child[trait] = random.choice([parent1[trait], parent2[trait]])
    return child

def mutate(individual): for trait in TRAITS:
        if random.random() < MUTATION_RATE: individual[trait] =
            random.uniform(0, 10)
    return individual

def evaluate_population(pop):
    return [(fitness_function(ind), ind) for ind in pop]
population = [random_individual() for _ in range(POP_SIZE)] for gen in

range(N_GEN):

    evaluated = evaluate_population(population)
    evaluated.sort(reverse=True, key=lambda x: x[0])
    parents = [ind for _, ind in evaluated[:10]] offspring = []

    for i in range(len(parents)//2):
        p1, p2 = parents[2*i], parents[2*i+1]
        child1 = mutate(crossover(p1, p2))
        child2 = mutate(crossover(p2, p1)) offspring.extend([child1, child2])

    population = offspring
    while len(population) < POP_SIZE: population.append(random_individual())

    print(f"\n=== Generation {gen+1} Offspring Population ===") for ind in
    population:
        fit = fitness_function(ind) print(f"Fitness={fit:.2f} | {ind}")

Output:
```

```
Generation 0: Best Solution = -9.967365011554792, Fitness = 99.34836527356666
Generation 1: Best Solution = -9.169251894044368, Fitness = 84.07518029643623
Generation 49: Best Solution = 9.123059138454053, Fitness = 83.23020804373002
Best Solution Found: x = 9.05670095588789, f(x) = 82.02383220438064
```

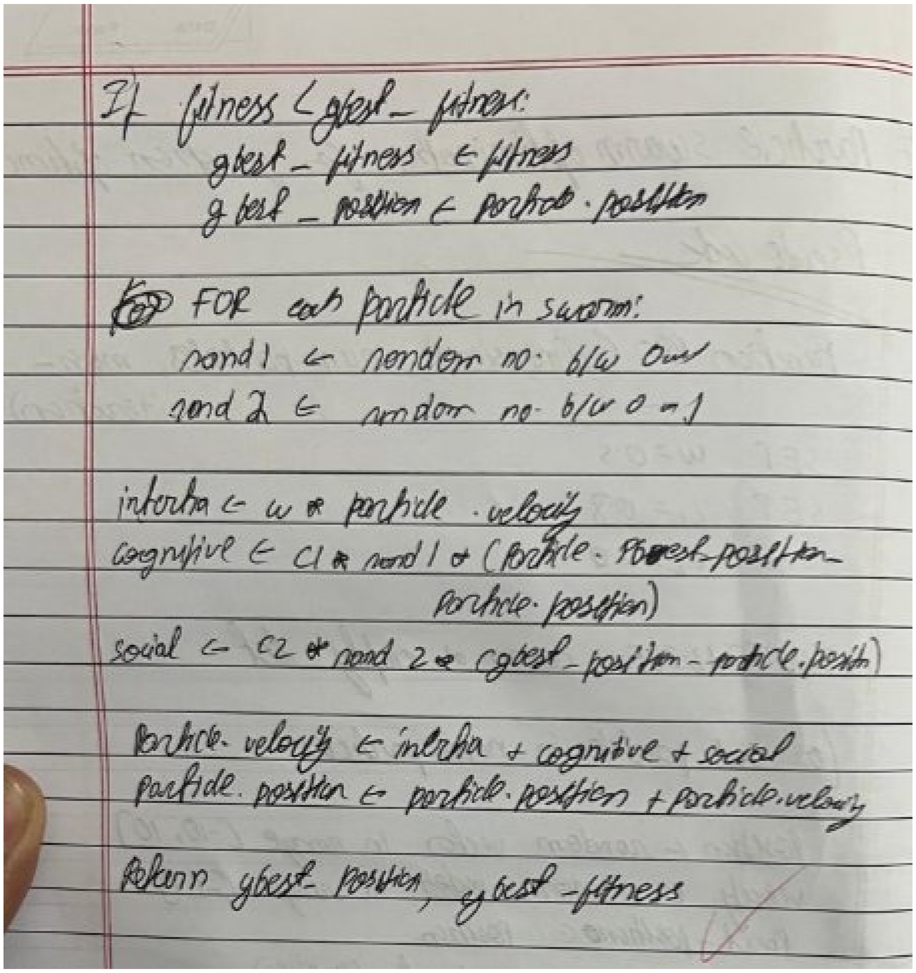
Program 2
Particle Swarm Optimization for Function Optimization
Algorithm:



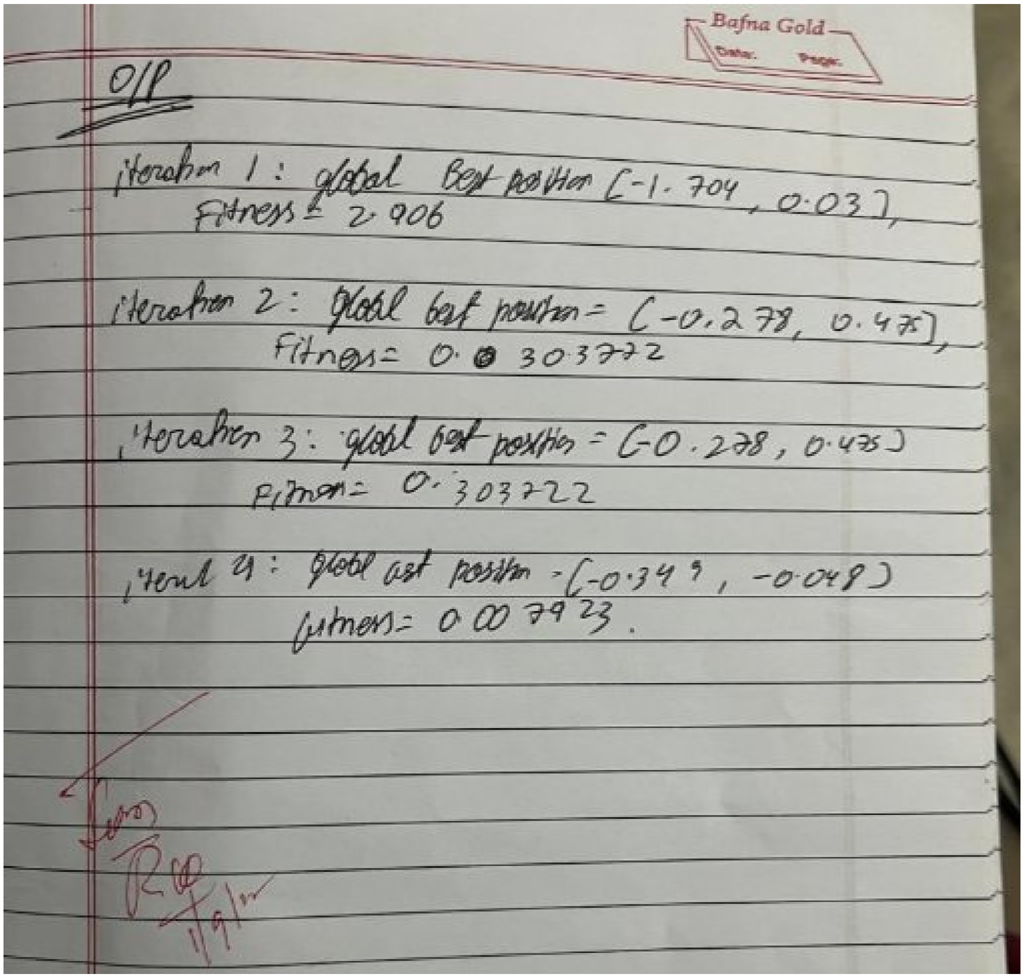
Particle swarm optimization for function optimizatn.

Pseudo code

```
Function PSO (dimensions, num-particles, max-
                                        iteration):
    SET  w=0.5
    SET  c1=0.8
    SET  c2=0.9

    INITIALIZE  swarm as empty list

    for each particle in num-particles:

        Position ← random vector in range [-10, 10]
        velocity ← random vector in range [-1, 1]
        Pbest-position ← position
        Pbest fitness ← fitness-function (position)
        ADD particle to swarm

    gbest-position ← zero vector
    gbest-fitness ← ∞

    for iteration in max-iteration
        for each particle in swarm:
            fitness ← fitness function (particle. position)

    If  fitness < particle-pbest-fitness:
        particle.pbest_fitness ← fitness
        particle.best - position ← particle.position.
```



```
If fitness < gbest_fitness:
    gbest_fitness ← fitness
    gbest_position ← particle.position

FOR each particle in swarm:
    rand1 ← random no. b/w 0-1
    rand2 ← random no. b/w 0-1

inertia ← w * particle.velocity
cognitive ← c1 * rand1 * (particle.pbest_position -
                particle.position)
social ← c2 * rand2 * (gbest_position - particle.position)

particle.velocity ← inertia + cognitive + social
particle.position ← particle.position + particle.velocity

Return gbest_position, gbest_fitness
```



Bafna Gold
Date: Page:

O/P

```
iteration 1: global Best position [-1.704, 0.03],
    Fitness = 2.906

iteration 2: global best position = [-0.278, 0.475],
    Fitness = 0.303772

iteration 3: global best position = [-0.278, 0.475]
    Fitness = 0.303722

iteration 4: global best position = [-0.349, -0.048]
    fitness = 0.007923.
```

Code:

```
import numpy as np import random

# Objective function
def objective_function(x):
    return -x**2 + 5*x + 20   # maximize this

class PSO:
    def __init__(self, n_particles=30, n_iterations=50, bounds=(-10, 10),
                 w=0.7, c1=1.5, c2=1.5):
        self.n_particles = n_particles
        self.n_iterations = n_iterations self.bounds = bounds


        self.w = w         # inertia
        self.c1 = c1       # personal influence
        self.c2 = c2       # social influence

        # Initialize particles
        self.positions = np.random.uniform(bounds[0], bounds[1], n_particles)
        self.velocities = np.zeros(n_particles)

        # Personal bests
        self.pbest_positions = np.copy(self.positions)
        self.pbest_values = np.array([objective_function(x) for x in self.positions])

        # Global best
        best_idx = np.argmax(self.pbest_values) self.gbest_position =
        self.pbest_positions[best_idx] self.gbest_value = self.pbest_values[best_idx]

    def optimize(self):
        for t in range(self.n_iterations): for i in range(self.n_particles):
                r1, r2 = random.random(), random.random()

                # Update velocity
                inertia = self.w * self.velocities[i]
                cognitive = self.c1 * r1 * (self.pbest_positions[i] - self.positions[i]) social
                = self.c2 * r2 * (self.gbest_position - self.positions[i]) self.velocities[i] =
                inertia + cognitive + social

                # Update position
                self.positions[i] += self.velocities[i]

                # Clamp position inside bounds
                self.positions[i] = np.clip(self.positions[i], self.bounds[0], self.bounds[1])

                # Evaluate
                value = objective_function(self.positions[i])

                # Update personal best
```

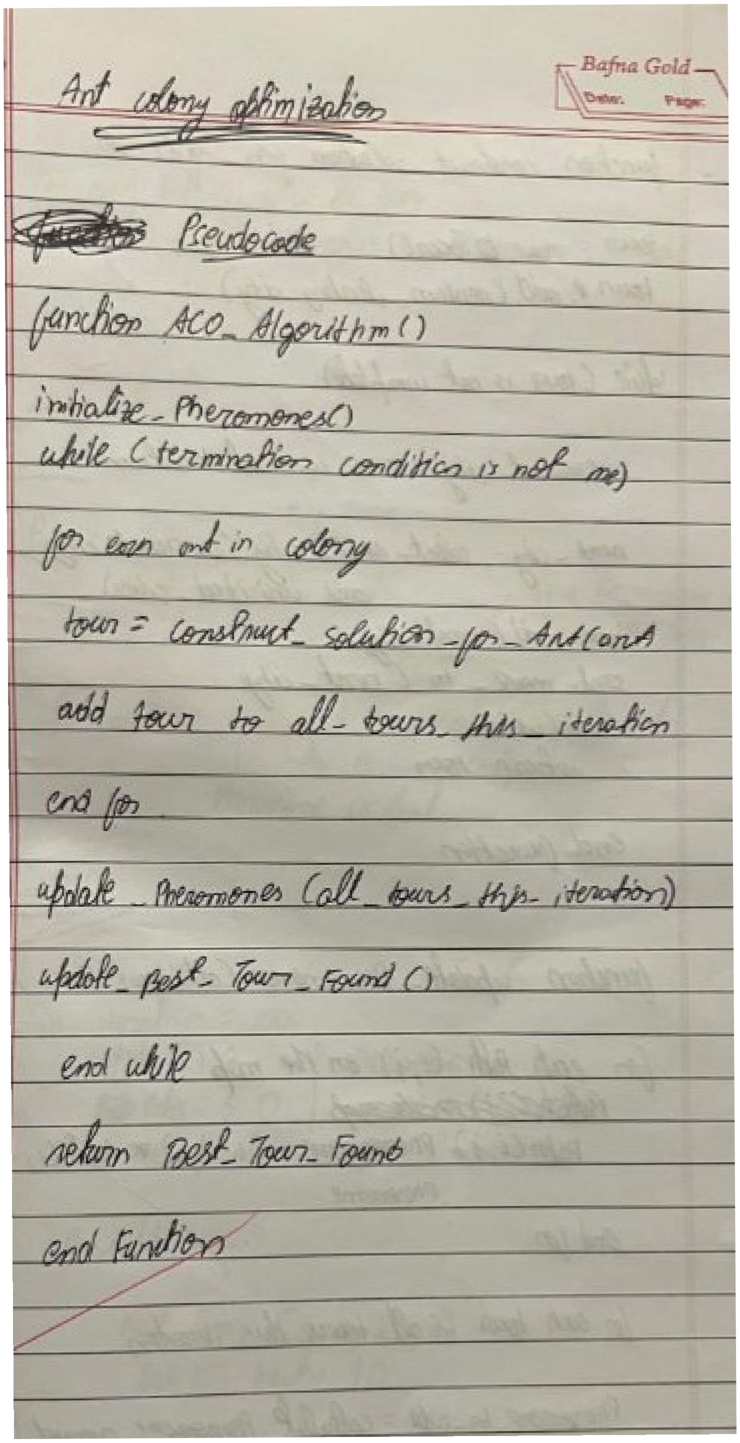
```
            if value > self.pbest_values[i]: self.pbest_positions[i] = self.positions[i]
                self.pbest_values[i] = value

                # Update global best
                if value > self.gbest_value: self.gbest_position = self.positions[i]
                    self.gbest_value = value

        print(f"Iteration {t+1}/{self.n_iterations}, Best = {self.gbest_value:.4f} at x =
{self.gbest_position:.4f}")

    return self.gbest_position, self.gbest_value


#
# Example Usage #
if __name__ == "__main__":
    pso = PSO(n_particles=20, n_iterations=50, bounds=(-10, 10), w=0.7, c1=1.5, c2=1.5)
    best_x, best_val = pso.optimize()
    print("\nBest Solution Found:")
    print(f"x = {best_x:.4f}, f(x) = {best_val:.4f}")
```

```
Output:

Best Position: [-9.19971249e-25  1.71937901e-24]
Best Fitness: 3.802611270068504e-48
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

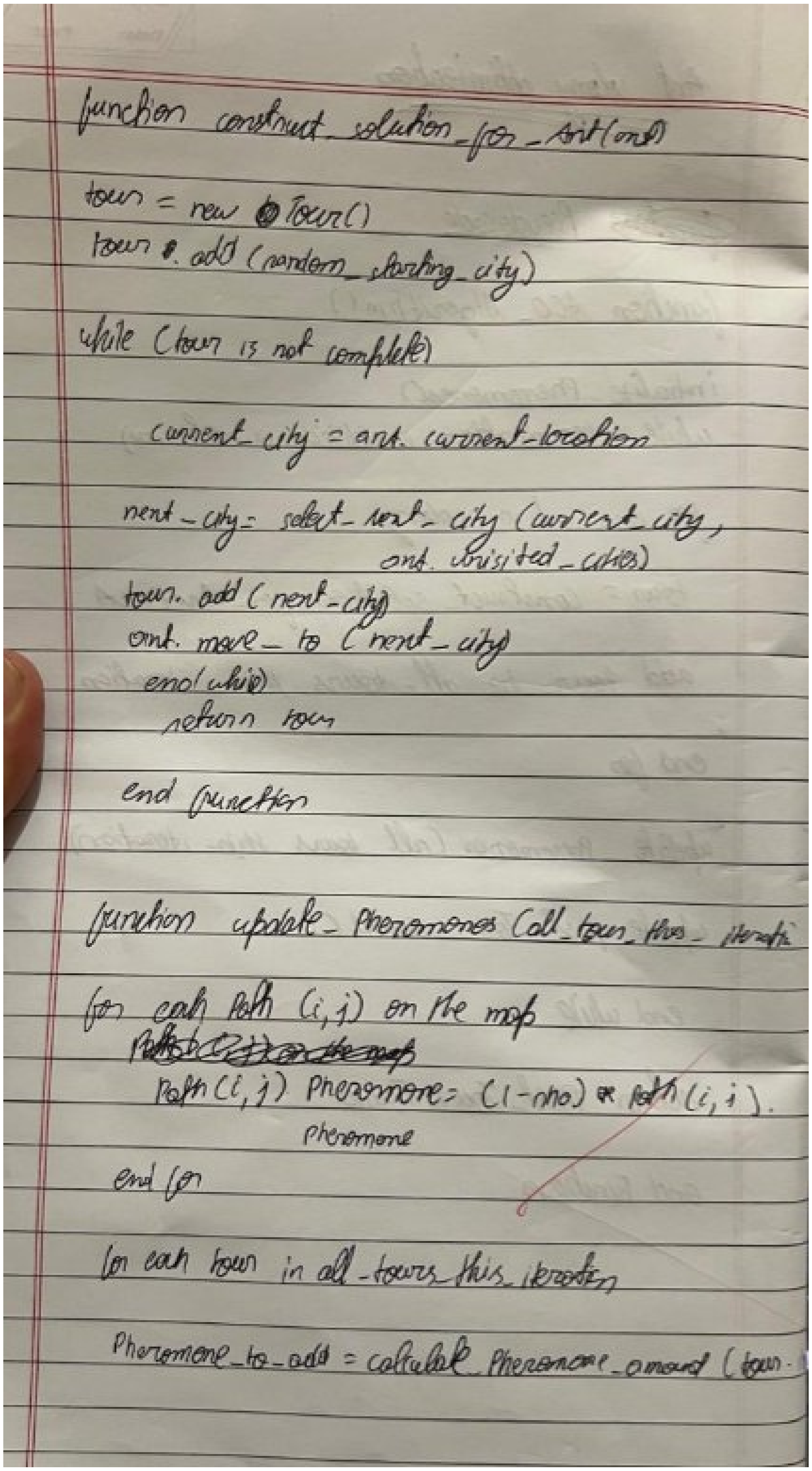
Algorithm:



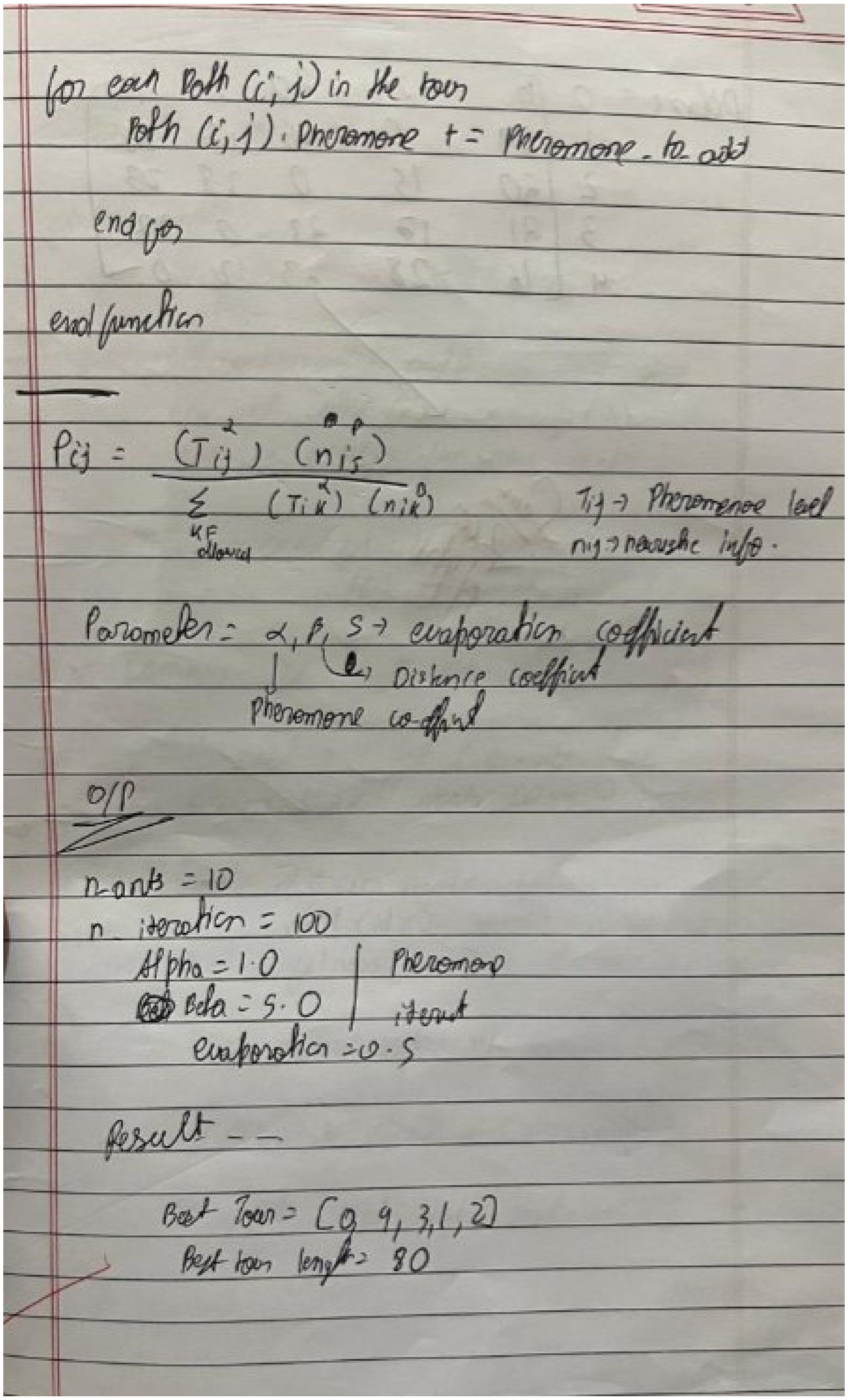
Ant colony optimization

Pseudocode

```
function ACO_Algorithm()
initialize_Pheromones()
while (termination condition is not me)
for each ant in colony
  tour = construct_solution_for_Ant(ant
  add tour to all_tours_this_iteration
end for
update_Pheromones (all_tours_this_iteration)
update_Best_Tour_Found ()
 end while
return Best_Tour_Found
end Function
```



```
function construct_solution_for_ant(ant)

tour = new Tour()
tour. add (random_starting_city)

while (tour is not complete)

    current_city = ant. current-location

    next_city = select_next_city (current_city,
                      ant. visited_cities)
    tour. add (next_city)
    ant. move_to (next_city)
  end(while)
   return tour

end function

function update_pheromones (all_tours_this_iterati

for each path (i,j) on the map
    ~~path(i,j) on the map~~
    path(i,j) pheromone = (1-rho) * path (i,j).
                 pheromone
end for

for each tour in all_tours_this_iteration

Pheromone_to_add = calculate_pheromone_amount (tour.
```



```
for each path (i, j) in the tour
    path (i, j). pheromone += pheromone_to_add

  end for

end function
```

$$P_{ij} = \frac{(\tau_{ij}^{\alpha})\,(\eta_{ij}^{\beta})}{\sum_{k \in \text{allowed}} (\tau_{ik}^{\alpha})\,(\eta_{ik}^{\beta})}$$

τ_{ij} → Pheromone level
η_{ij} → heuristic info.

Parameters = α, β, ρ → evaporation coefficient
β → Distance coefficient
α → pheromone co-efficient

O/P

n-ants = 10
n_iteration = 100
Alpha = 1.0 | Pheromone
Beta = 5.0 | iterat
evaporation = 0.5

Result --

Best Tour = [0, 4, 3, 1, 2]
Best tour length = 80

Code:

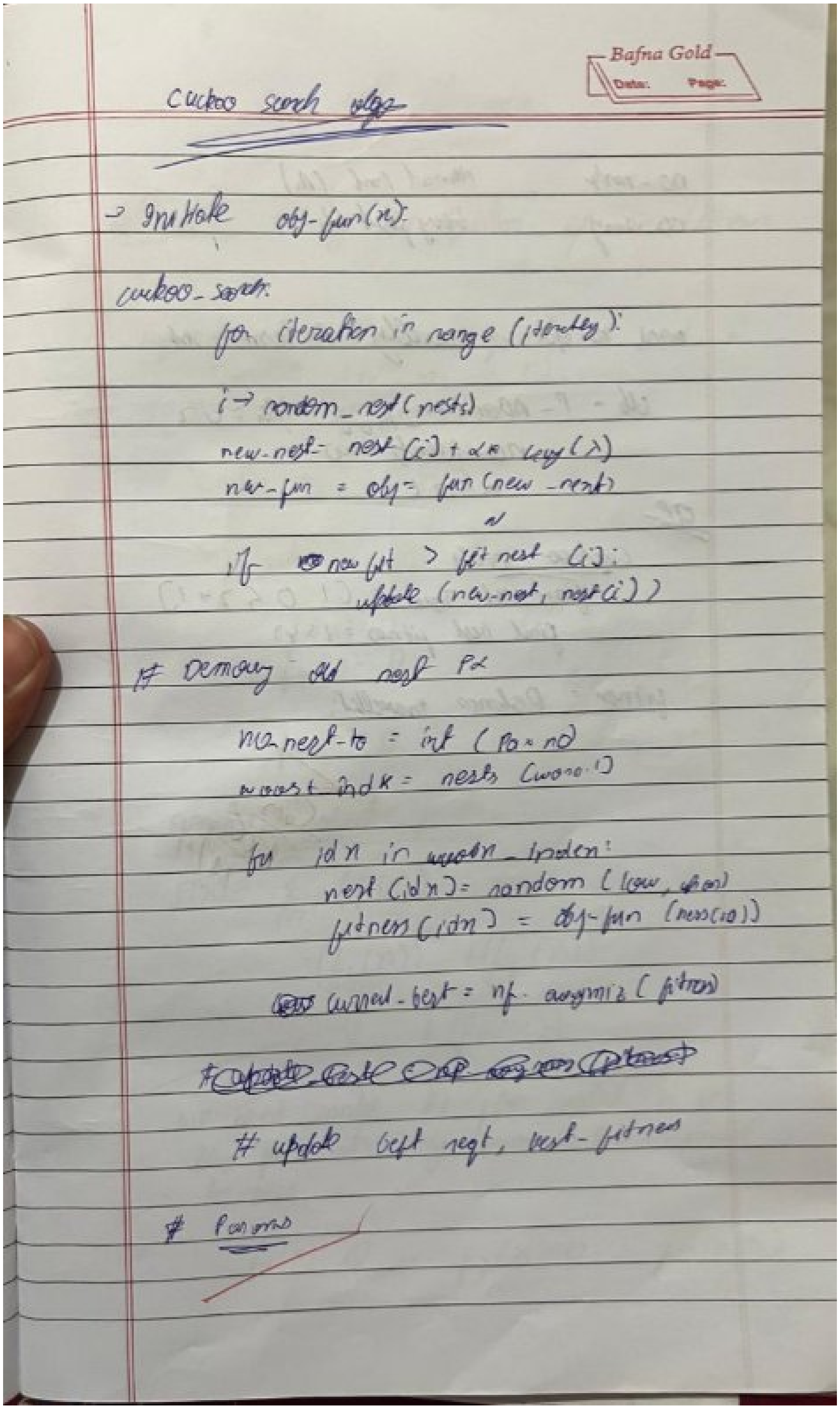
```
import numpy as np import random

class ACO_TSP:
    def __init__(self, dist_matrix, n_ants=10, n_iterations=100, alpha=1,
beta=2, rho=0.5, Q=100):
        """
        dist_matrix: distance matrix (2D numpy array) n_ants: number of ants
        n_iterations: number of iterations alpha: influence of pheromone
        beta: influence of heuristic (1/distance) rho: pheromone evaporation
        rate
        Q: pheromone deposit factor
        """
        self.dist_matrix = dist_matrix self.n_cities = dist_matrix.shape[0]
        self.n_ants = n_ants self.n_iterations = n_iterations self.alpha =
        alpha
        self.beta = beta self.rho = rho
        self.Q = Q

        # Initialize pheromone matrix
        self.pheromone = np.ones((self.n_cities, self.n_cities))

        # Best solution self.best_length = float("inf") self.best_path =
        None


    def run(self):
        for iteration in range(self.n_iterations):
            all_paths = self.construct_solutions()
            self.update_pheromones(all_paths)
            print(f"Iteration {iteration+1}/{self.n_iterations}, Best length
so far: {self.best_length}")
        return self.best_path, self.best_length

    def construct_solutions(self): all_paths = []
        for ant in range(self.n_ants):  path = self.build_path()  length =
            self.path_length(path)
            all_paths.append((path, length))
            # Update global best
            if length < self.best_length: self.best_length = length
                self.best_path = path
        return all_paths

    def build_path(self): path = []  visited = set()
        start = random.randint(0, self.n_cities - 1)
        path.append(start)
        visited.add(start)

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

        return path




    def choose_next_city(self, current_city, visited): pheromone
        = np.copy(self.pheromone[current_city])
        heuristic = 1 / (self.dist_matrix[current_city] + 1e-10)  # avoid div
by 0

        # Zero out visited cities for city in visited:
            pheromone[city] = 0
            heuristic[city] = 0

        probabilities = (pheromone ** self.alpha) * (heuristic ** self.beta)
        probabilities = probabilities / probabilities.sum()
        return np.random.choice(range(self.n_cities), p=probabilities) def

    path_length(self, path):

        length = 0
        for i in range(len(path)):
            length += self.dist_matrix[path[i]][path[(i+1) % self.n_cities]]
        return length

    def update_pheromones(self, all_paths): # Evaporation
        self.pheromone *= (1 - self.rho)
        # Deposit new pheromone
        for path, length in all_paths: deposit_amount = self.Q / length for i
            in range(len(path)):
                a, b = path[i], path[(i+1) % self.n_cities]
                self.pheromone[a][b] += deposit_amount
                self.pheromone[b][a] += deposit_amount



#
# ~~Example Usage #~~
if __name___== "__main__":
    ~~# Distance matrix (symm~~etric for TSP) dist_matrix = np.array([
        [0, 2, 9, 10],
        [1, 0, 6, 4],
        [15, 7, 0, 8],
        [6, 3, 12, 0]
    ])

    aco = ACO_TSP(dist_matrix, n_ants=5, n_iterations=50, alpha=1, beta=2,
rho=0.5, Q=100)
    best_path, best_length = aco.run() print("\nBest Path:", best_path)
    print("Best Path Length:", best_length)

Output:
```

```
Best Route: [0, 1, 4, 3, 2, 0]
Best Distance: 12.313755207963359
```

Program 4

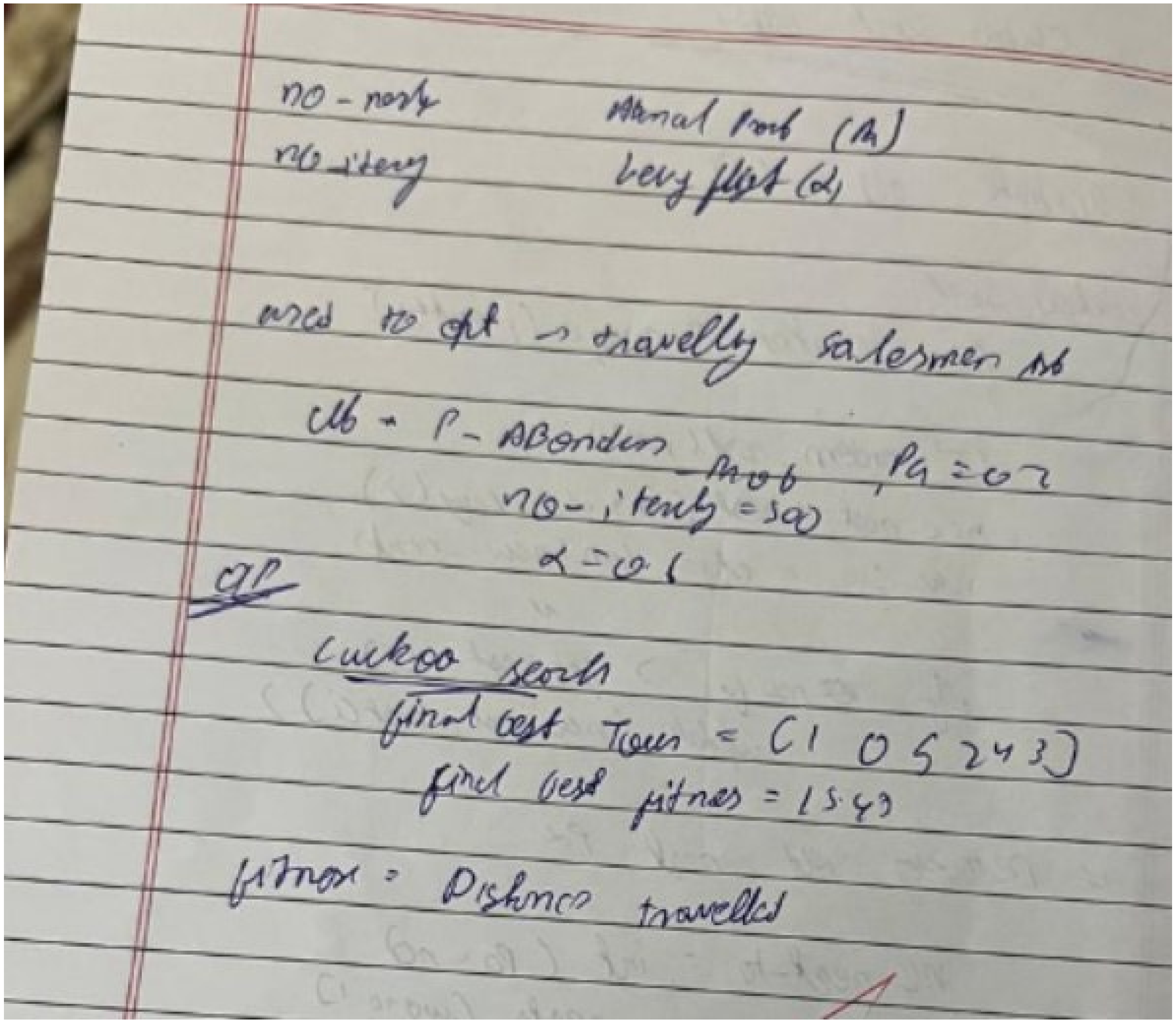
Cuckoo Search (CS)

Algorithm:



Cuckoo search algo

```
→ Initiate obj-fun(x):

cuckoo-search:
    for iteration in range (iterations):

        i → random_nest (nests)
        new-nest = nest (i) + α × levy (λ)
        new-fun = obj-fun (new-nest)

        if new fit > fit nest (i):
            update (new-nest, nest (i))

# Removing old nest Pα

        no.nest-to = int (Pα × n)
        worst index = nests (worst)

        for idx in worst_index:
            nest (idx) = random (low, upper)
            fitness (idx) = obj-fun (nests (idx))

        current-best = np. argmin (fitness)

    # [illegible]

        # update best nest, best-fitness

# Params
```



no - nests   Abanal Prob (Pa)

no - itery   Levy flight (α)

used to opt → travelling salesmen pb

cb → P - Abandon prob, $P_a = 0.2$

no - iterty = 500

$\alpha = 0.1$

<u>o/p</u>

<u>Cuckoo</u> search

final best Tour = [1 0 5 2 4 3]

final best fitness = 15.43

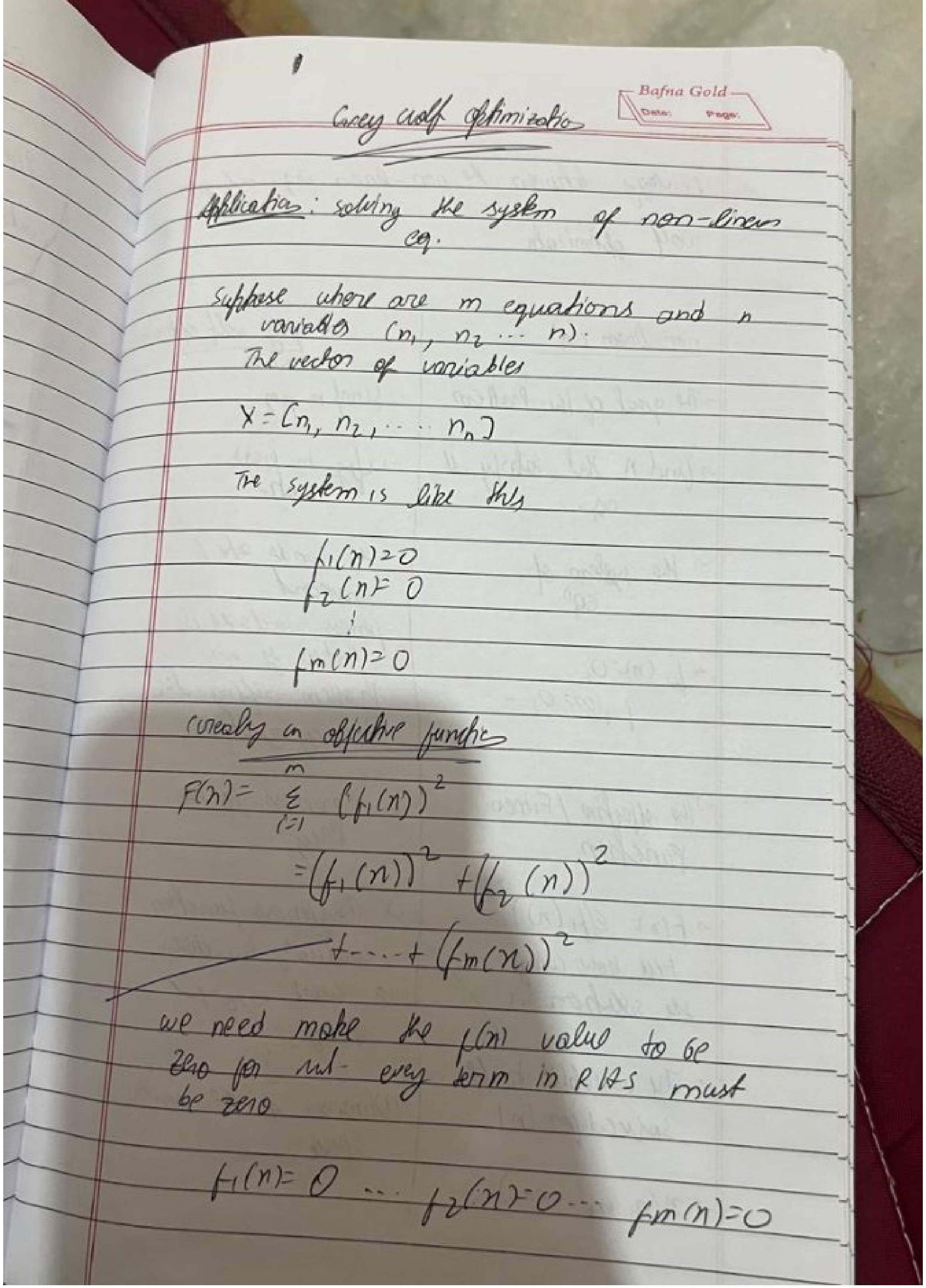
fitness = Distance travelled

Code:

```
import numpy as np import random

class CuckooSearch:
    def __init__(self, objective_func, bounds, n_nests=25, pa=0.25, max_iter=1000):
        """
        Cuckoo Search Algorithm

        Parameters:
        - objective_func: Function to minimize
        - bounds: List of tuples [(min, max)] for each dimension
        - n_nests: Number of host nests (population size)
        - pa: Discovery probability (probability of abandoning worst nests)
        - max_iter: Maximum number of iterations """
        self.objective_func = objective_func self.bounds = bounds
        self.n_nests = n_nests self.pa = pa
        self.max_iter = max_iter self.dim = len(bounds)

        # Initialize nests
        self.nests = np.zeros((n_nests, self.dim)) self.fitness = np.zeros(n_nests)

        # Best solution tracking self.best_nest = None self.best_fitness = float('inf')

    def initialize_nests(self):
        """Initialize nests with random positions within bounds""" for i in
        range(self.n_nests):
            for j in range(self.dim):
                lower, upper = self.bounds[j]
                self.nests[i, j] = lower + (upper - lower) * random.random() self.fitness[i]
            = self.objective_func(self.nests[i])

        # Find initial best
        best_idx = np.argmin(self.fitness) self.best_nest = self.nests[best_idx].copy()
        self.best_fitness = self.fitness[best_idx]

    def levy_flight(self, beta=1.5): """
        Generate step size using Levy flight
        """
        # Generate random direction
        u = np.random.normal(0, 1, self.dim) v = np.random.normal(0, 1, self.dim)

        # Calculate step size using Levy distribution
        sigma = (np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) / (np.math.gamma((1
                + beta) / 2) * beta * (2 ** ((beta - 1) / 2)))) ** (1 / beta)


        step = 0.01 * (u / (np.abs(v) ** (1 / beta))) * sigma return step

    def generate_new_solution(self, nest_idx): """Generate new solution using Levy
        flight""" step = self.levy_flight()
        new_nest = self.nests[nest_idx].copy()

        for j in range(self.dim):
            lower, upper = self.bounds[j] new_nest[j] += step[j]
            # Boundary check
            new_nest[j] = np.clip(new_nest[j], lower, upper)

        return new_nest

    def abandon_worst_nests(self):
        """Abandon worst nests and build new ones""" # Sort nests by fitness
        sorted_indices = np.argsort(self.fitness)
        num_abandon = int(self.n_nests * self.pa)

        # Abandon worst nests
        for i in range(num_abandon): idx = sorted_indices[-(i + 1)] for j in
            range(self.dim):
                lower, upper = self.bounds[j]
                self.nests[idx, j] = lower + (upper - lower) * random.random()
            self.fitness[idx] = self.objective_func(self.nests[idx])

    def optimize(self):
        """Main optimization loop""" self.initialize_nests()

        print(f"Initial best fitness: {self.best_fitness}") for iteration in

        range(self.max_iter):
            # Generate new solutions using Levy flight for i in
            range(self.n_nests):
                # Get a cuckoo randomly by Levy flight new_solution =
                self.generate_new_solution(i)
                new_fitness = self.objective_func(new_solution)

                # Choose a random nest
                j = random.randint(0, self.n_nests - 1)

                # If new solution is better, replace it if new_fitness <
                self.fitness[j]:
                    self.nests[j] = new_solution
                    self.fitness[j] = new_fitness

                    # Update global best
                    if new_fitness < self.best_fitness: self.best_nest =
                        new_solution.copy() self.best_fitness = new_fitness

            # Abandon worst nests self.abandon_worst_nests()

            # Keep the best solution
            current_best_idx = np.argmin(self.fitness)
            if self.fitness[current_best_idx] < self.best_fitness: self.best_nest
                = self.nests[current_best_idx].copy() self.best_fitness =
                self.fitness[current_best_idx]

            if iteration % 100 == 0:
                print(f"Iteration {iteration}, Best fitness: {self.best_fitness}")

        print(f"Final best fitness: {self.best_fitness}") print(f"Best solution:
        {self.best_nest}")

        return self.best_nest, self.best_fitness

# Example usage: Minimize f(x) = x^2 (as shown in the PDF) def
sphere_function(x):
    """Example objective function: f(x) = x^2""" return np.sum(x**2)

# Test with the example from the PDF (1D problem) if __name
___________== "__main__":
    # Define bounds for 1D problem
    bounds = [(-10, 10)]

    # Create Cuckoo Search instance
    cs = CuckooSearch(sphere_function, bounds, n_nests=15, pa=0.25, max_iter=500)

    # Run optimization
    best_solution, best_fitness = cs.optimize()

    print("\n" + "="*50)
                                print("CUCKOO SEARCH ALGORITHM RESULTS")
    print("="*50)
    print(f"Global minimum found at: x = {best_solution[0]:.6f}") print(f"Function value: f(x) =
    {best_fitness:.6f}") print(f"Expected: x = 0.0, f(x) = 0.0")

# Additional example: 2D Rosenbrock function def rosenbrock_function(x):
    """Rosenbrock function - common test function for optimization""" return sum(100.0 * (x[1:]
    - x[:-1]**2)**2 + (1 - x[:-1])**2)

# Test with 2D problem print("\n" + "="*50)
                            print("TESTING WITH ROSENBROCK FUNCTION (2D)")
print("="*50)

bounds_2d = [(-2, 2), (-2, 2)]
cs_2d = CuckooSearch(rosenbrock_function, bounds_2d, n_nests=20, pa=0.25, max_iter=1000)
best_solution_2d, best_fitness_2d = cs_2d.optimize()

print(f"Best solution: {best_solution_2d}") print(f"Best fitness: {best_fitness_2d:.6f}")
```

```
Output :

Best Solution: [0.64982748 0.55961241 2.01501756 0.93987275 0.31984962]
Best Fitness: 5.78140211553397
```

Program 5
Grey Wolf Optimizer (GWO)
Algorithm:



Bafna Gold
Date: Page:

Grey wolf optimization

Application: solving the system of non-linear eq.

Suppose where are m equations and n variables $(n_1, n_2 \cdots n)$.
The vector of variables

$X = [n_1, n_2, \cdots n_n]$

The system is like this

$f_1(n) = 0$
$f_2(n) = 0$
$\vdots$
$f_m(n) = 0$

creating an objective function

$F(n) = \sum_{i=1}^{m} (f_i(n))^2$

$= (f_1(n))^2 + (f_2(n))^2$

$+ \cdots + (f_m(n))^2$

we need make the $f(n)$ value to be zero for not every term in RHS must be zero

$f_1(n) = 0 \cdots f_2(n) = 0 \cdots f_m(n) = 0$

Code:

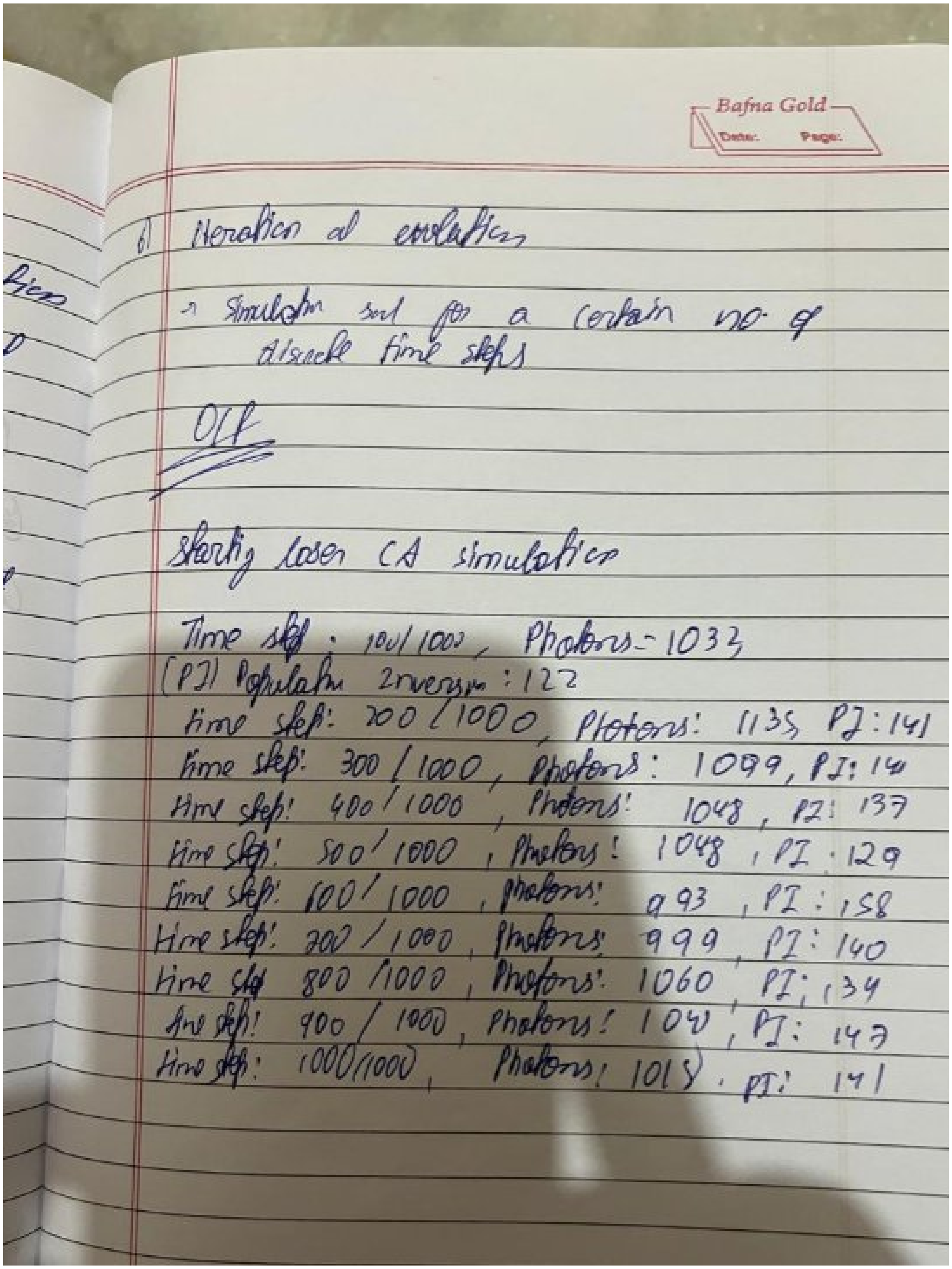
```
import numpy as np

# Objective function (example: Sphere function) def objective_function(x):
    return np.sum(x**2)
N, dim, T = 30, 10, 100  # Number of wolves, dimensions, iterations
lower_bound, upper_bound = -10, 10

wolves = np.random.uniform(lower_bound, upper_bound, (N, dim))

alpha_pos, beta_pos, delta_pos = np.zeros(dim), np.zeros(dim), np.zeros(dim)
alpha_score, beta_score, delta_score = float('inf'), float('inf'),
float('inf')
for t in range(T): for i in range(N):
        fitness = objective_function(wolves[i])  # Evaluate fitness if fitness <
        alpha_score:
            delta_score, delta_pos = beta_score, beta_pos.copy() beta_score,
            beta_pos = alpha_score, alpha_pos.copy()
            alpha_score, alpha_pos = fitness, wolves[i].copy() elif fitness <
        beta_score:
            delta_score, delta_pos = beta_score, beta_pos.copy() beta_score,
            beta_pos = fitness, wolves[i].copy()
        elif fitness < delta_score:
            delta_score, delta_pos = fitness, wolves[i].copy() a = 2 - t * (2 / T)
    for i in range(N):
        r1, r2 = np.random.rand(dim), np.random.rand(dim)
        A, C = 2 * a * r1 - a, 2 * r2
        wolves[i] += A * (abs(C * alpha_pos - wolves[i]) + abs(C * beta_pos -
                          wolves[i]) +
                          abs(C * delta_pos - wolves[i]))

        wolves[i] = np.clip(wolves[i], lower_bound, upper_bound) print("Best
Solution:", alpha_pos)
print("Best Score:", alpha_score)
```

```
Output:

Best Solution: [-1.28434275  1.94786008  0.82301541 -1.85113457 -2.08806377
3.74582237
  0.84065243  0.8938704  -1.22271966 -0.29007149]
Best Score: 31.023829961456407
```

Program 6

Parallel Cellular Algorithms and Programs

Algorithm:


Bafna Gold
Date:
Page:

6) Iteration of evolution

→ Simulation run for a certain no. of discrete time steps

O/P

```
Starting laser CA simulation

Time step: 100/1000, Photons=1033,
(PI) Population Inversion: 122
time step: 200/1000, Photons: 1135, PI: 141
time step: 300/1000, Photons: 1099, PI: 140
time step: 400/1000, Photons: 1048, PI: 137
time step: 500/1000, Photons: 1048, PI: 129
time step: 600/1000, photons: 993, PI: 158
time step: 700/1000, photons: 999, PI: 140
time step 800/1000, Photons: 1060, PI: 134
time step: 900/1000, Photons: 1040, PI: 147
time step: 1000/1000, Photons: 1015, PI: 141
```

Code:

```
import numpy as np import random
import concurrent.futures

def rastrigin(x):
   A = 10
   return A * len(x) + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in
x])

GRID_SIZE = (10, 10)
DIM = 2
RADIUS = 1
ITER = 100
BEST = None

def init_grid(size, dim):
   return [[np.random.uniform(-5.12, 5.12, size=(dim,)) for _ in
range(size[1])] for _ in range(size[0])]

def fitness(cell):   return rastrigin(cell)

def update_state(grid, i, j, radius): curr = grid[i][j]
   fitness_curr = fitness(curr)
   neighbors = [grid[ni][nj] for dx in range(-radius, radius+1) for dy in
range(-radius, radius+1)
            if 0 <= (ni := i+dx) < len(grid) and 0 <= (nj := j+dy) <
len(grid[0]) and (dx or dy)]
   if neighbors:
      best_neigh = min(neighbors, key=fitness) return curr + 0.1 *
      (best_neigh - curr)
   return curr

def run_iteration(grid, radius):
   new_grid = [[None for _ in range(len(grid[0]))] for _ in
range(len(grid))]
   with concurrent.futures.ThreadPoolExecutor() as ex:
      futures = [ex.submit(update_state, grid, i, j, radius) for i in
range(len(grid)) for j in range(len(grid[0]))]
      for idx, future in enumerate(futures):
         i, j = divmod(idx, len(grid[0])) new_grid[i][j] = future.result()
   return new_grid
```

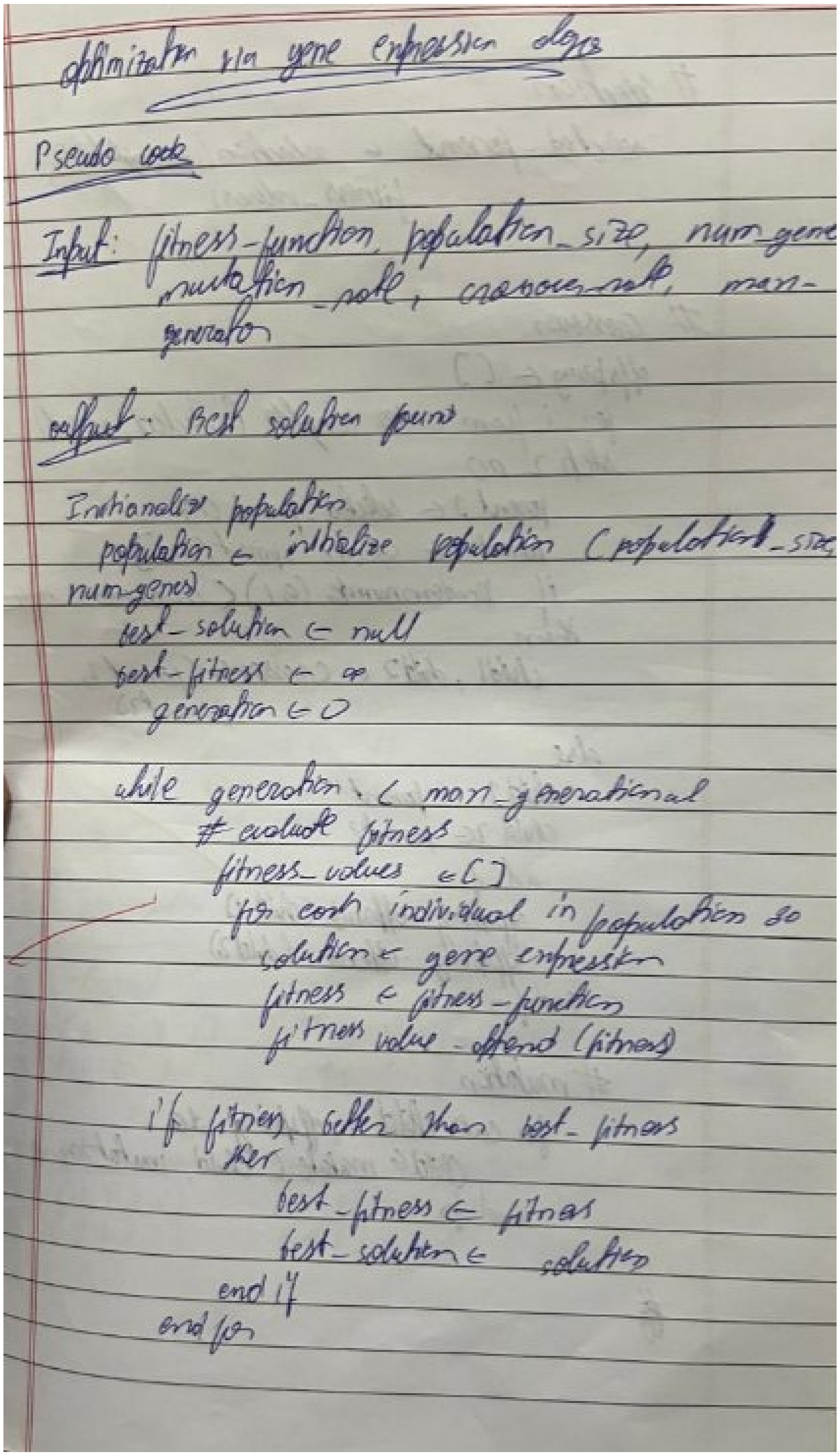
```
def track_best(grid): global BEST
    best_cell, best_fitness = None, float('inf')
    for row in grid: for cell in row:
            f = fitness(cell)
            if f < best_fitness: best_fitness = f best_cell =
                cell
    if BEST is None or best_fitness < fitness(BEST):
        BEST = best_cell

def parallel_cellular_algorithm(): global BEST
    grid = init_grid(GRID_SIZE, DIM)
    for _ in range(ITER):
        grid = run_iteration(grid, RADIUS)
        track_best(grid)
        print(f"Best Fitness: {fitness(BEST)}")
    print("Best Solution:", BEST) print("Best
    Fitness:", fitness(BEST))

parallel_cellular_algorithm()
```

```
Output:

Best Fitness: 2.4309484366586602
Best Fitness: 2.4309484366586602
Best Fitness: 0.0007801439196555293
Best Fitness: 0.0007801439196555293
Best Fitness: 0.0007801439196555293
Best Solution: [ 0.00129305 -0.00150346]
Best Fitness: 0.0007801439196555293
```

Program 7

Optimization via Gene Expression Algorithms

Algorithm:



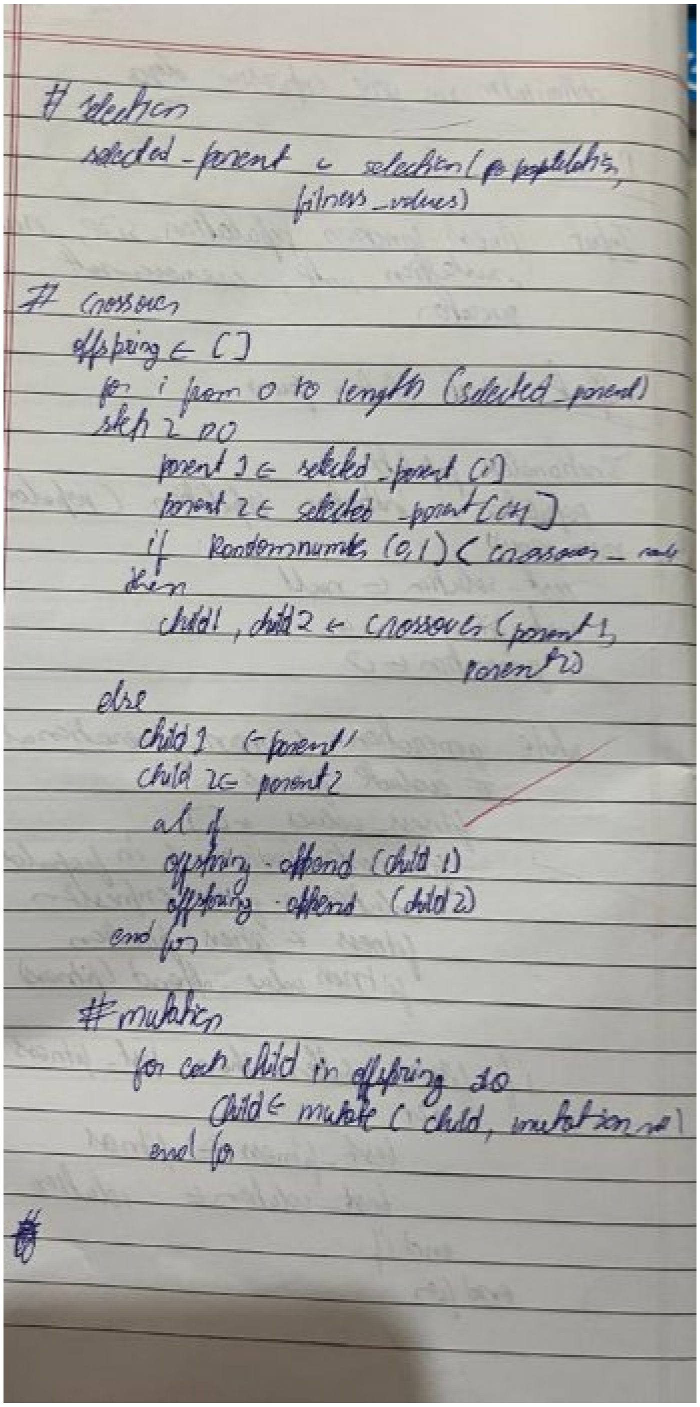
optimization via gene expression algos

Pseudo code

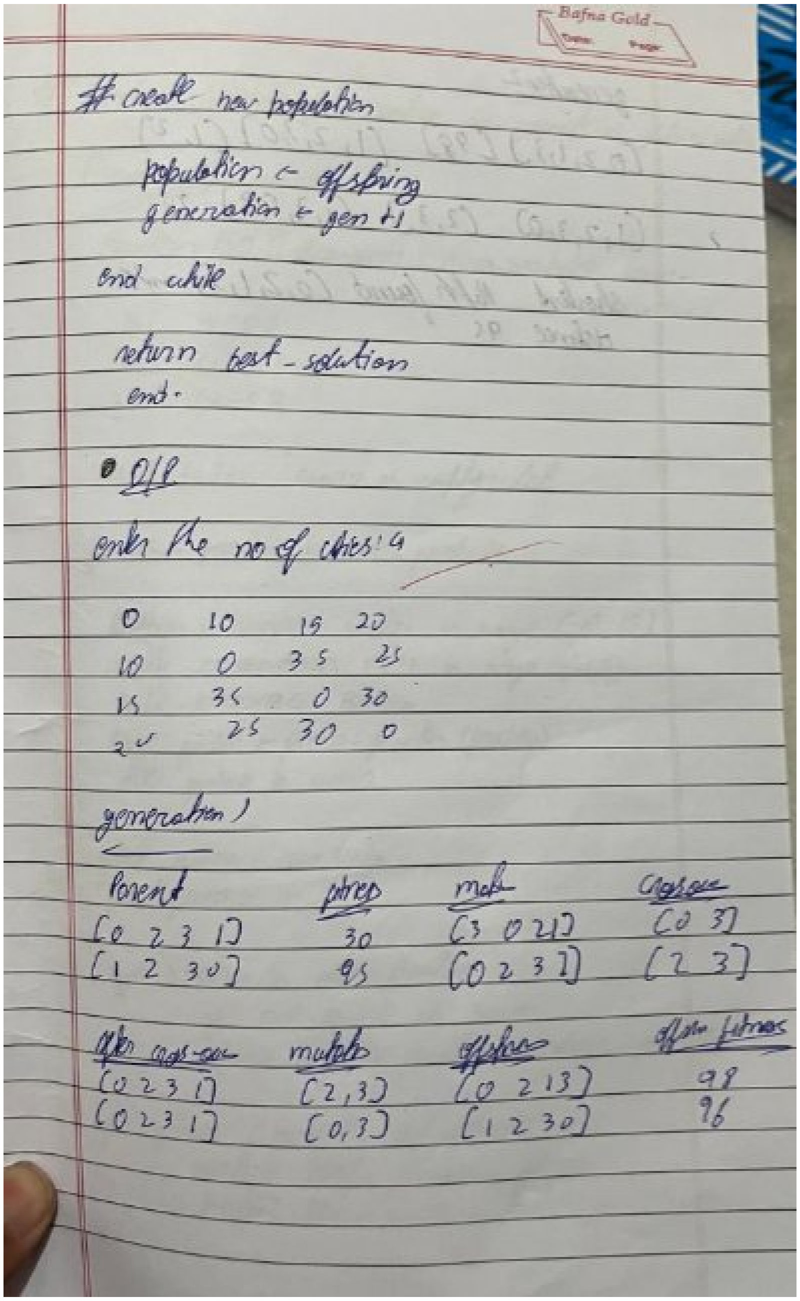
Input: fitness-function, population_size, num_genes, mutation_rate, crossover_rate, max-generation

output: Best solution found

```
Initionalize population
  population ← initialize population (population_size,
num_genes)
    best_solution ← null
    best_fitness ← ∞
      generation ← 0

    while generation < max_generational
          # evaluate fitness
          fitness_values ← []
           for each individual in population do
             solution ← gene expression
             fitness ← fitness_function
             fitness value.append (fitness)

      if fitness better than best_fitness
           then
             best_fitness ← fitness
             best_solution ← solution
          end if
      end for
```



```
# selection
    selected_parent ← selection(~~po~~ population,
                       fitness_values)

# crossover
    offspring ← []
      for i from 0 to length (selected_parent)
      step 2 DO
          parent 1 ← selected_parent (i)
          parent 2 ← selected_parent [i+1]
          if  Randomnumber (0,1) < crossover_rate
       then
          child1 , child2 ← crossover (parent1,
                                 parent2)
      else
          child1 ← parent1
          child 2 ← parent2
           end if
           offspring.append (child 1)
           offspring.append (child 2)
        end for
    # mutation
        for each child in offspring do
              child ← mutate ( child, mutation_rate)
          end for

#
```



```
# create new population
    population ← offspring
    generation ← gen +1
end while

return best_solution
end.
```

O/P

enter the no of cities: 4

```
0     10    15   20
10    0     35   25
15    35    0    30
20    25    30   0
```

generation 1

Parent	fitness	mate	crossover
[0 2 3 1]	30	[3 0 2 1]	[0 3]
[1 2 3 0]	95	[0 2 3 1]	[2 3]

after cross-over	mutation	offspring	offspring fitness
[0 2 3 1]	[2,3]	[0 2 1 3]	98
[0 2 3 1]	[0,3]	[1 2 3 0]	96

Code:

```
import numpy as np import random

# Objective Function: f(x) = 2x - sin(x) def objective_function(x):
    return 2 * x - np.sin(x)

# Parameters
population_size = 10  # Population size (updated to 10) mutation_rate = 0.15  # Mutation rate
(updated to 0.15) crossover_rate = 0.15  # Crossover rate (updated to 0.15) num_generations
= 5  # Number of generations (updated to 5) lower_bound = -5  # Lower bound of the search
space upper_bound = 5   # Upper bound of the search space

# Initialize Population (Random sequences for x in the given range) def
initialize_population(pop_size, lower_bound, upper_bound):
    population = []
    for _ in range(pop_size):
        individual = random.uniform(lower_bound, upper_bound)  # Random float in the range
        population.append(individual)
    return population

# Evaluate Fitness
def evaluate_fitness(population): fitness_values = []
    for individual in population:
        fitness = objective_function(individual) fitness_values.append(fitness)
    return fitness_values

# Selection (Roulette Wheel Selection)
def select_parents(population, fitness_values): total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]

    selected_parents = np.random.choice(population, size=2, p=probabilities) return
    selected_parents

# Crossover (Single-point crossover) def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        # For one gene, just take the average as the crossover child1 = (parent1 + parent2) / 2
        child2 = (parent1 + parent2) / 2
        return child1, child2
    return parent1, parent2

# Mutation (Uniform mutation)
def mutate(individual, mutation_rate, lower_bound, upper_bound): if
    random.random() < mutation_rate:
        mutation_value = random.uniform(lower_bound, upper_bound)
        individual = mutation_value return individual

# Main Genetic Algorithm def genetic_algorithm():
    population = initialize_population(population_size, lower_bound, upper_bound)
    best_solution = None best_fitness = -float('inf')

    for generation in range(num_generations): fitness_values =
        evaluate_fitness(population)

        # Track the best solution
        max_fitness_idx = np.argmax(fitness_values)
        if fitness_values[max_fitness_idx] > best_fitness: best_fitness =
            fitness_values[max_fitness_idx] best_solution =
            population[max_fitness_idx]

        # Selection
        selected_parents = select_parents(population, fitness_values)

        # Crossover and Mutation new_population = []
        for i in range(0, population_size, 2):
            parent1, parent2 = selected_parents
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate, lower_bound, upper_bound) child2 =
            mutate(child2, mutation_rate, lower_bound, upper_bound)
            new_population.extend([child1, child2])

        # Update population population = new_population

    return best_solution, best_fitness # Run the algorithm
best_solution, best_fitness = genetic_algorithm()

# Output the best solution found
print("Best solution (x value):", best_solution) print("Maximized fitness (f(x)
value):", best_fitness)
```

```
Output:


Generation 1: Best Fitness = 16.545885126119284
 Generation 2: Best Fitness =
 11.641082640808637
Generation 99: Best Fitness = 0.02233046748484963
...
Generation 100: Best Fitness = 0.02233046748484963

Optimal Solution Found:
Best Solution: [ 0.07226226 -0.11854791  0.03245473 -0.01236219  0.04299877]
Best Fitness: 0.02233046748484963
```