VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sam Manu Jacob (1BM23CS291)

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 **Sam Manu Jacob (1BM23CS291),** 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.

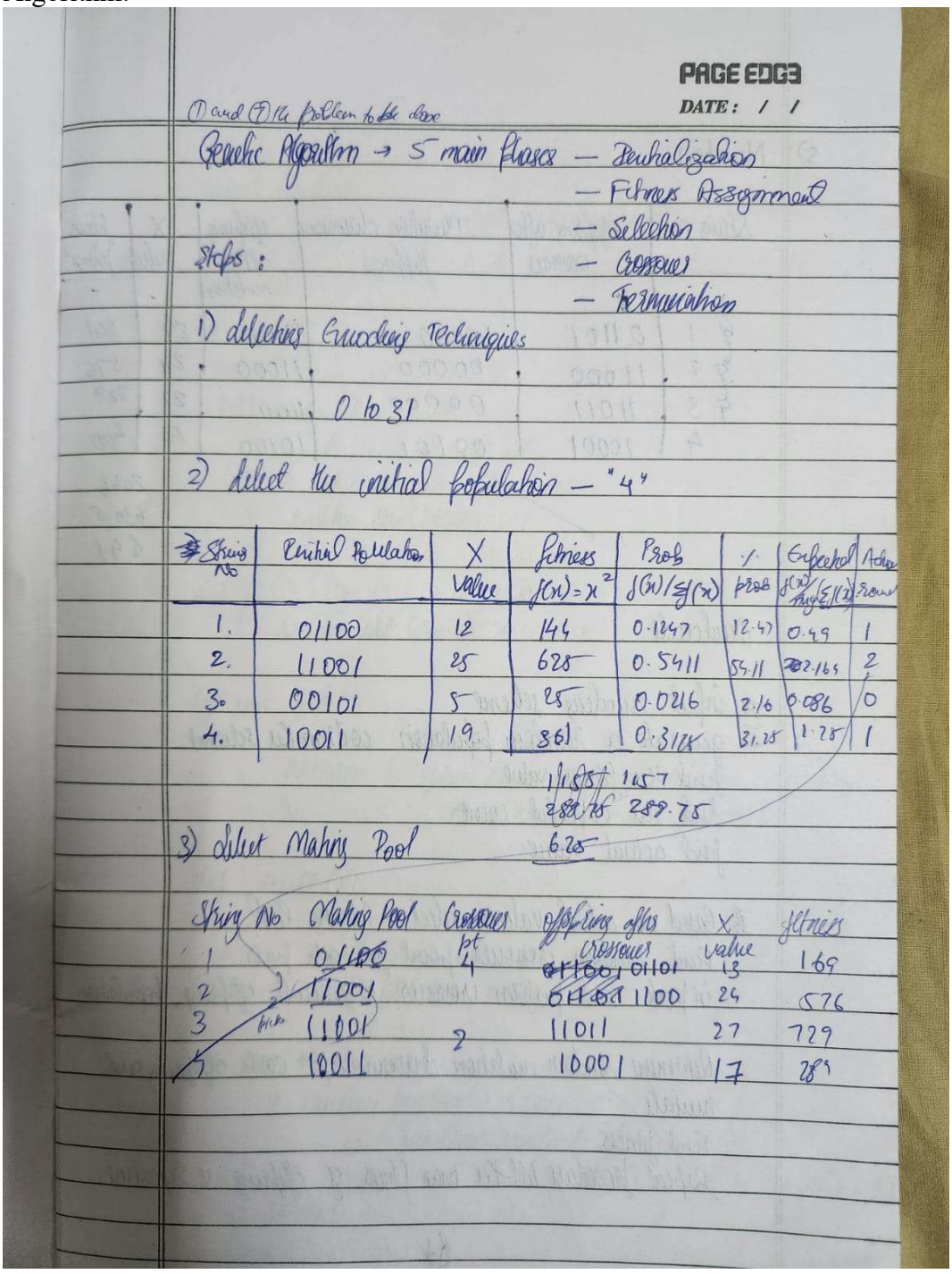
Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/SamManuJacob/BIS_lab_1BM23CS291

Program 1
Genetic Algorithm

Algorithm:



PAGE EDGE
DATE : / /

① and ② the problem to be done

Genetic Algorithm → 5 main phases — Initialization
— Fitness Assignment
— Selection
— Crossover
— Termination

Steps :

1) Selecting Encoding Techniques

0 to 31

2) Select the initial population — "4"

String No	Initial Population	X value	Fitness $f(x)=x^2$	Prob $f(x)/\sum f(x)$	% Prob	Expected $f(x)/\text{Avg}(\sum f(x))$	Actual Count
1.	01100	12	144	0.1247	12.47	0.49	1
2.	11001	25	625	0.5411	54.11	2.164	2
3.	00101	5	25	0.0216	2.16	0.086	0
4.	10011	19	361	0.3125	31.25	1.25	1
			1155	1157			
			288.75	289.75			

3) Select Mating Pool 625

String No	Mating Pool	Crossover pt	Offspring after crossover	X value	Fitness
1	01100	4	01101	13	169
2	11001		11100	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

5) Mutation

String no	offspring after crossover	Mutation chromosome / Mutation	offspring after mutation	X value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

2546
630.5
841

Pseudocode

- input encoding scheme
- generate a random population within the scheme
  find the fitness value
  find the expected count
  find actual value
- Based on actual values select a Mating Pool
  Find random crossover point for each pair in pool and perform crossover to generate offspring population
  Determine random mutation chromosome for each offspring and mutate
  Find fitness
  Repeat procedure till the max fitness of offspring is reaching.

PAGE EDGE
DATE: / /

```
        nxt.extend([mutate(c1), mutate(c2)])

    pop = nxt[:POP]

print("\nStopped after " + gen + " generations")
print("\nFinal best: " + best + " (x = " + int(best,2) + "),
      Fit = " + best_fit)

run()
```

Output:

Gen 1: Best = 01011 (x=11), Fit = 121, Avg Fit = 34.33
Gen 2: Best = 01110 (x=14), Fit = 196, Avg Fit = 66.50
Gen 3: Best = 01111 (x=15), Fit = 225, Avg Fit = 131.33
Gen 4: Best = 01111 (x=15), Fit = 225, Avg Fit = 176.83
Gen 5: Best = 01111 (x=15), Fit = 225, Avg Fit = 175.50
Gen 6: Best = 11111 (x=31) Fit = 961, Avg Fit = 310.67
Gen 7: Best = 11111 (x=31) Fit = 961 Avg Fit = 524.17
Gen 8: Best = 11111 (x=31) Fit = 961 Avg Fit = 553.50
Gen 9: Best = 11111 (x=31) Fit = 961 Avg Fit = 596.00
Gen 10: Best = 11111 (x=31), Fit = 961 Avg Fit = 558.50
Gen 11: Best = 11111 (x=31), Fit = 961 Avg Fit = 816.67

Stopped at 11 generation, Reaching best fit value of 961 for 5 consecutive generations

Final Best: 11111 (x = 31) Fit = 961

Code:

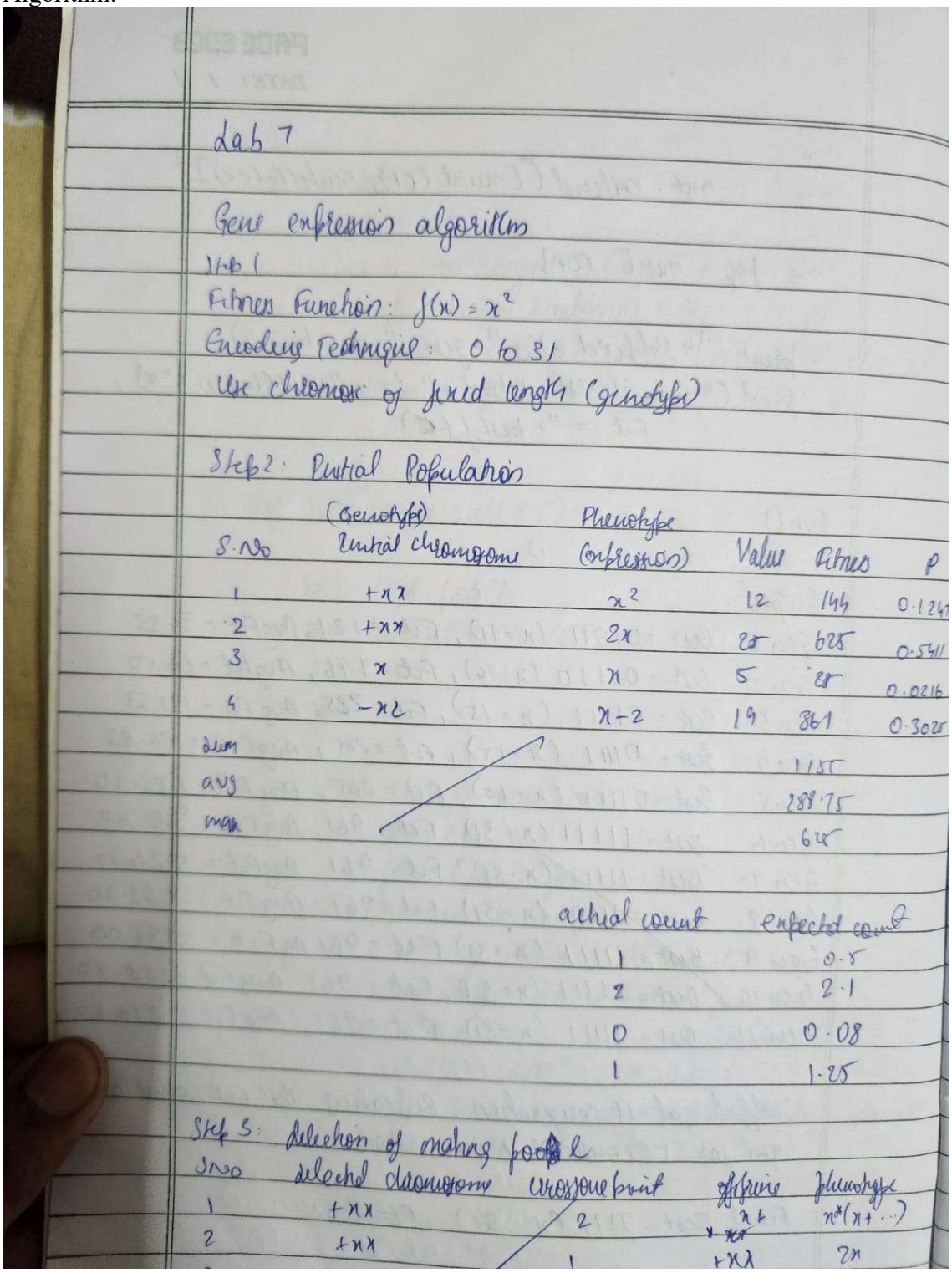
```
import random

# Parameters
POP = 6
BITS = 5
CROSS = 0.7
MUT = 0.1
SEED = 42   # set None for random runs

if SEED is not None:
    random.seed(SEED)

def fit(s):
    """Fitness = square of decoded integer."""
    return int(s, 2) ** 2

def init_pop():
    return [''.join(random.choice('01') for _ in range(BITS)) for _ in range(POP)]

def select(p):
    """Tournament selection of 2, return best."""
    a, b = random.sample(p, 2)
    return a if fit(a) > fit(b) else b

def cross(a, b):
    """Single-point crossover."""
    if random.random() < CROSS:
        pt = random.randint(1, BITS - 1)
        return a[:pt] + b[pt:], b[:pt] + a[pt:]
    return a, b

def mutate(s):
    """Bit-flip mutation."""
    bits = list(s)
    for i in range(len(bits)):
        if random.random() < MUT:
            bits[i] = '1' if bits[i] == '0' else '0'
    return ''.join(bits)

def run(max_stall=5):
    """
    Run until best solution repeats (stagnation).
    max_stall = number of generations with no improvement before stopping.
    """
    pop = init_pop()
    best, best_fit = None, -1
    stall = 0
    gen = 0
```

```
    while stall < max_stall:
        gen += 1
        fits = [fit(x) for x in pop]
        gen_best = max(fits)
        gen_avg = sum(fits) / len(fits)
        best_idx = fits.index(gen_best)

        if gen_best > best_fit:
            best, best_fit = pop[best_idx], gen_best
            stall = 0  # reset if improvement
        else:
            stall += 1

        print(f"Gen {gen}: Best={best} (x={int(best,2)}), Fit={best_fit}, AvgFit={gen_avg:.2f}")

        nxt = []
        while len(nxt) < POP:
            p1, p2 = select(pop), select(pop)
            c1, c2 = cross(p1, p2)
            nxt.extend([mutate(c1), mutate(c2)])
        pop = nxt[:POP]

    print(f"\nStopped after {gen} generations (no improvement for {max_stall}).")
    print(f"Final Best: {best} (x={int(best,2)}), Fit={best_fit}")

if __name__ == "__main__":
    run()
```

Program 2

Gene Expression

Algorithm:



Lab 7

Gene expression algorithm

Step 1

Fitness Function: $f(x) = x^2$

Encoding Technique: 0 to 31

Use chromosome of fixed length (genotype)

Step 2: Initial Population

S.No	(Genotype) Initial chromosome	Phenotype (expression)	Value	Fitness	P
1	+xx	x^2	12	144	0.1247
2	+xx	2x	25	625	0.5411
3	x	x	5	25	0.0216
4	-x2	x-2	19	361	0.3025
sum				1155	
avg				288.75	
max				625	

	actual count	expected count
1		0.5
2		2.1
0		0.08
1		1.25

Step 3: Selection of mating pool

S.No	selected chromosome	crossover point	offspring	phenotype
1	+xx	2	x+	$x*(x+\cdots)$
2	+xx	1	+xx	2x

Step 4: Crossover: Perform crossover randomly chosen gene position (not raw bits)

max Fitness after crossover = 729

Step 5: Mutation

S.No	Offspring before mutation	mutation applied	offspring after mutation	phenotype
1	*x+	+ → −	*x−	x*(x−−)
2	+xx	None	xxx	2x
3	+x−	− → +	−x+	x+x*x
4	+x2	None	+x2	x+2

x value	Fitness
29	841
24	576
27	729
20	400

Step 6: Gene expression and evaluation

decode each genotype → phenotype
calculate fitness

$\Sigma f(x) = 841 + 576 + 729 + 400 = 2546$

avg = 636.5

max = 841

Step 7: Iterate until convergence

Repeat Step 3 to 6 until fitness improvement is negligible or generation limit has reached.

Pseudocode

Define Fitness
Define Parameters
Generate Population
Select Mating Pool
Mutate
Gene expression and evaluation
Iterate till ~~repeating~~ values ~~is~~ occurs
Output Best

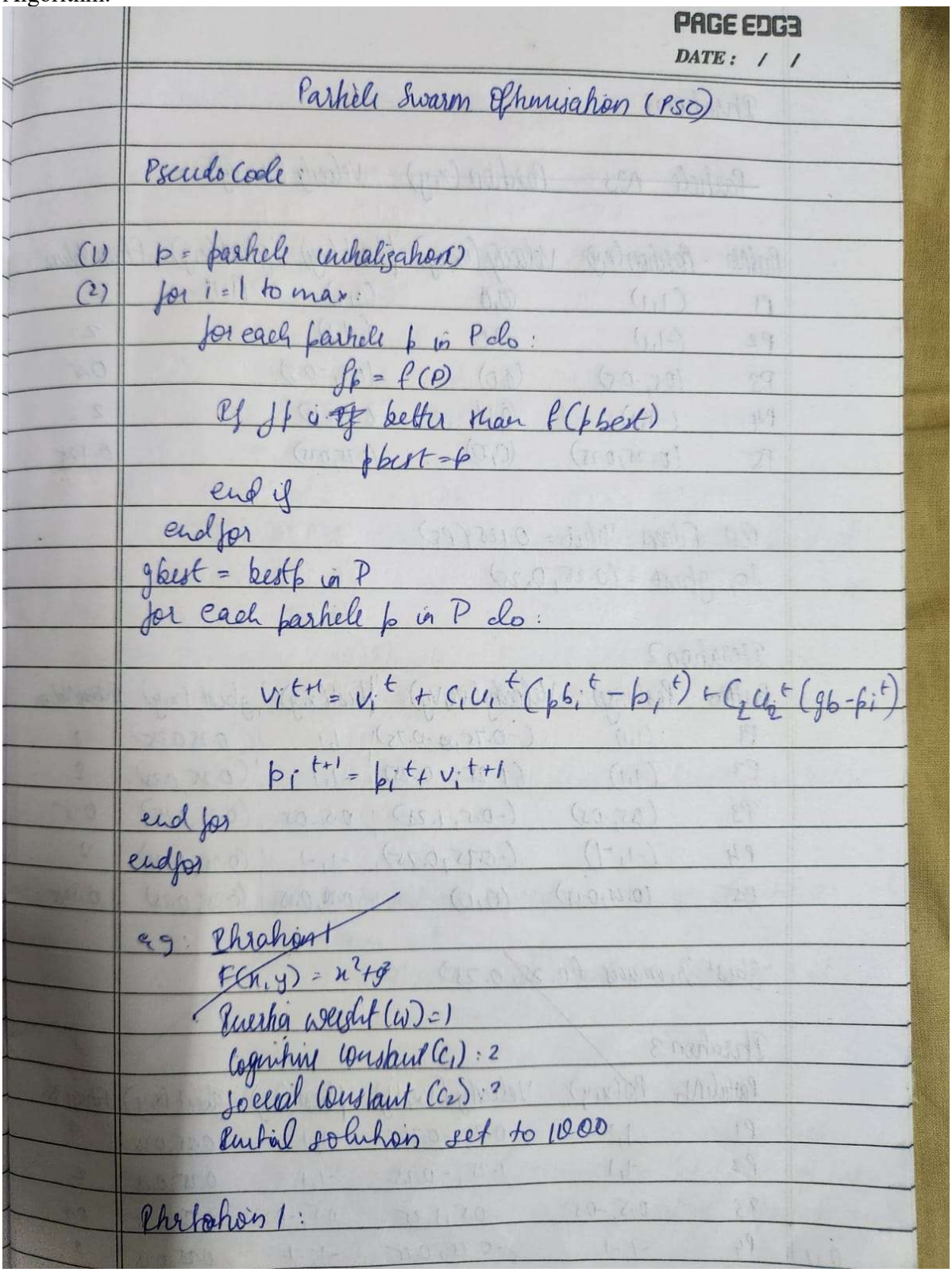
Code:

```
import random, math

def fitness(x):
    return x * math.sin(10 * math.pi * x) + 2

def init_pop(size):
    return [random.uniform(-1, 2) for _ in range(size)]

def select(pop, fits):
    # shift fitnesses if any are negative (roulette requires non-negative)
    min_fit = min(fits)
    if min_fit < 0:
        fits = [f - min_fit + 1e-6 for f in fits]

    total = sum(fits)
    r = random.uniform(0, total)
    cum = 0
    for p, f in zip(pop, fits):
        cum += f
        if cum >= r:
            return p
    return random.choice(pop)

def crossover(a, b, rate):
    if random.random() < rate:
        alpha = random.random()
        return alpha * a + (1 - alpha) * b, alpha * b + (1 - alpha) * a
    return a, b

def mutate(x, rate, sigma=0.1):
    if random.random() < rate:
        return min(2, max(-1, x + random.gauss(0, sigma)))
    return x

def run():
    POP = 6
    CROSS, MUT = 0.8, 0.05
    pop = init_pop(POP)
    best, best_fit = None, float("-inf")
    stall, max_stall = 0, 5
    gen = 0

    while stall < max_stall:  # stop when no improvement
        gen += 1
        fits = [fitness(x) for x in pop]
        gen_best_fit = max(fits)
        gen_best = pop[fits.index(gen_best_fit)]
```

```
        if gen_best_fit > best_fit:
            best, best_fit = gen_best, gen_best_fit
            stall = 0
        else:
            stall += 1

        avg_fit = sum(fits) / len(fits)
        print(f"Gen {gen}: Best Fitness = {best_fit:.4f}, x = {best:.4f}, AvgFit = {avg_fit:.4f}")

        new_pop = []
        while len(new_pop) < POP:
            p1, p2 = select(pop, fits), select(pop, fits)
            c1, c2 = crossover(p1, p2, CROSS)
            new_pop += [mutate(c1, MUT), mutate(c2, MUT)]
        pop = new_pop[:POP]

    print(f"\nStopped after {gen} generations (no improvement for {max_stall}).")
    print(f"Best solution found: x = {best:.4f}, f(x) = {best_fit:.4f}")

if __name__ == "__main__":
    run()
```

Program 3
Particle Swarm Optimization

Algorithm:



PAGE EDGE
DATE : / /

Particle Swarm Optimization (PSO)

Pseudo Code :

(1) P = particle initialization()
(2) for i = 1 to max :
    for each particle p in P do :
        fp = f(p)
        If fp is better than f(pbest)
            pbest = p
    end if
    end for
    gbest = best p in P
    for each particle p in P do :

$$v_i^{t+1} = v_i^t + c_1 u_1^t (pb_i^t - p_i^t) + c_2 u_2^t (gb - p_i^t)$$

$$p_i^{t+1} = p_i^t + v_i^{t+1}$$

    end for
end for

eg: Rhrahon 1
F(x, y) = $x^2 + y^2$
Inertia weight (w) = 1
Cognitive constant (c_1) : 2
Social constant (c_2) : 2
Initial solution set to 1000

Rhrahon 1 :

Iteration 1

~~Particle No Position (x,y) Velocity (vx,vy)~~

Part No	Position (x,y)	Velocity (vx,vy)	pbest (xy)	gbest (x,y)	Fitness Value
P1	(1,1)	(0,0)	(1,1)	(1,1)	2
P2	(-1,1)	(0,0)	(-1,1)		2
P3	(0.5,-0.5)	(0,0)	(0.5,-0.5)		0.5
P4	(-1,1)	(0,0)	(-1,-1)		2
P5	(0.25,0.25)	(0,0)	(0.25,0.25)		0.125

Best Fitness Value = 0.125 (P5)
So, gbest = (0.25, 0.25)

Iteration 2

Part No	Pos (x,y)	Velocity (vx,vy)	pbest (x,y)	gbest (x,y)	Fitness Value
P1	(1,1)	(-0.75,-0.75)	1,1	(0.25,0.25)	2
P2	(-1,1)	(1.25,-0.25)	-1,1	(0.25,0.25)	2
P3	(0.5,-0.5)	(-0.5,1.25)	0.5,-0.5	(0.25,0.25)	0.5
P4	(-1,-1)	(-0.75,0.75)	-1,-1	(0.25,0.25)	2
P5	(0.25,0.25)	(0,0)	0.25,0.25	(0.25,0.25)	0.125

gbest remains (0.25, 0.25)

Iteration 3

Particle No	Pos (x,y)	Velocity (vx,vy)	pbest (x,y)	gbest (x,y)	Fitness Value
P1	1,1	-0.75,-0.75	1,1	0.25,0.25	2
P2	-1,1	1.25,-0.25	-1,1	0.25,0.25	2
P3	0.5,-0.5	-0.5,1.25	0.5,-0.5	0.25,0.25	0.5
P4	-1,-1	-0.75,0.75	-1,-1	0.25,0.25	2
P5	0.25,0.25	0,0	0.25,0.25	0.25,0.25	0.125

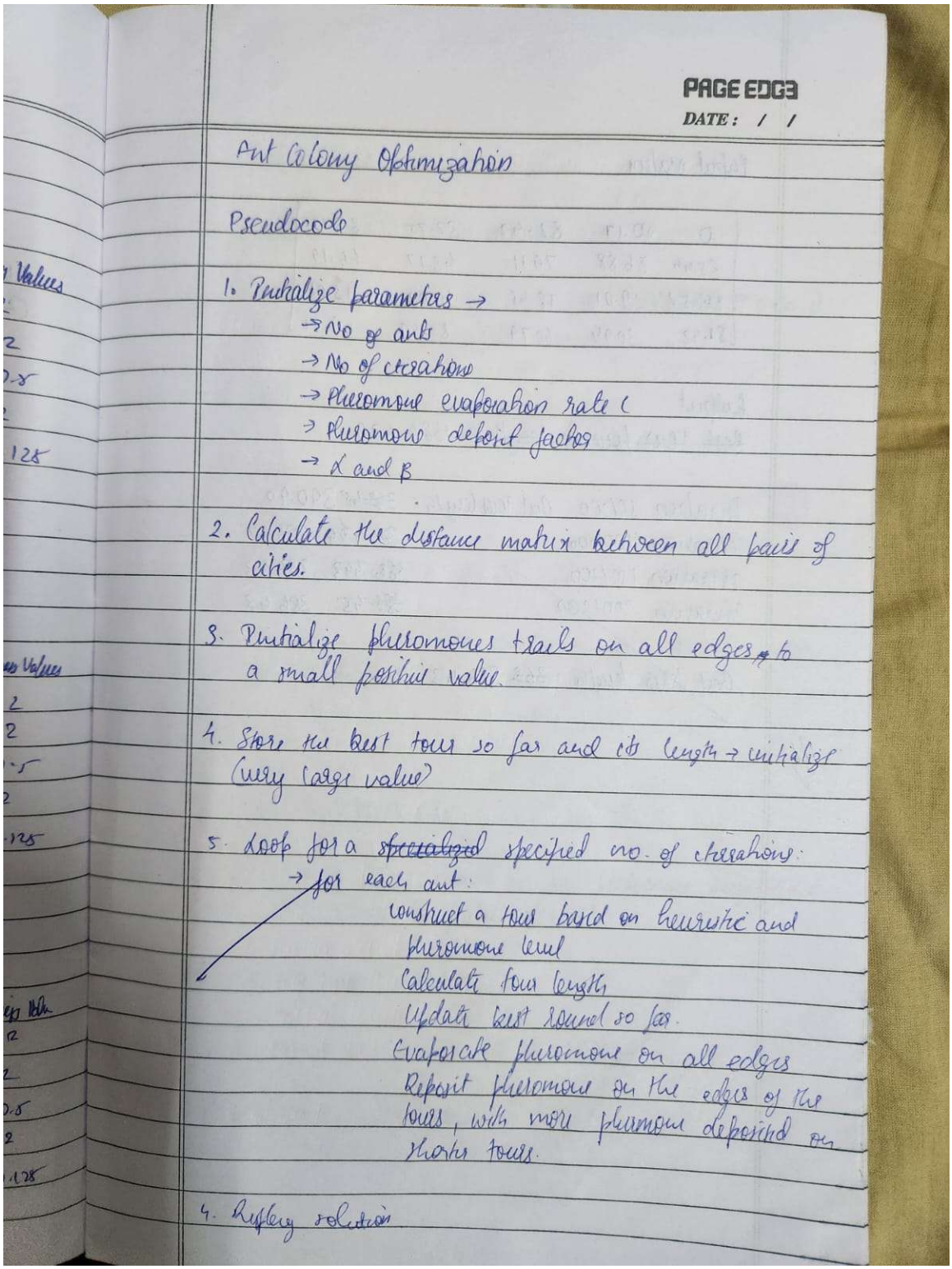
Code:

```
import random

# --- Objective Function (De Jong Sphere Function) ---
def evaluate(position):
    x, y = position
    return x**2 + y**2   # We aim to minimize this

# --- Hyperparameters ---
NUM_PARTICLES = 10
ITERATIONS = 50
INERTIA = 0.3       # Weight for previous velocity
COG_COEFF = 2.0     # Personal learning rate
SOC_COEFF = 2.0     # Social learning rate

# --- Particle Initialization ---
positions = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in range(NUM_PARTICLES)]
vels = [[0.0, 0.0] for _ in range(NUM_PARTICLES)]

# --- Personal and Global Bests ---
personal_best_positions = [pos[:] for pos in positions]
personal_best_scores = [evaluate(pos) for pos in positions]

best_particle_idx = personal_best_scores.index(min(personal_best_scores))
global_best_position = personal_best_positions[best_particle_idx][:]
global_best_score = personal_best_scores[best_particle_idx]

# --- PSO Iterations ---
for step in range(ITERATIONS):
    for i in range(NUM_PARTICLES):
        r1, r2 = random.random(), random.random()

        # --- Velocity Update ---
        for d in range(2):  # For each dimension
            inertia_term = INERTIA * vels[i][d]
            cognitive_term = COG_COEFF * r1 * (personal_best_positions[i][d] - positions[i][d])
            social_term = SOC_COEFF * r2 * (global_best_position[d] - positions[i][d])
            vels[i][d] = inertia_term + cognitive_term + social_term

        # --- Position Update ---
        positions[i][0] += vels[i][0]
        positions[i][1] += vels[i][1]

        # --- Evaluate New Fitness ---
        score = evaluate(positions[i])

        # --- Update Personal & Global Bests ---
        if score < personal_best_scores[i]:
            personal_best_scores[i] = score
            personal_best_positions[i] = positions[i][:]
```

```
            if score < global_best_score:
                global_best_score = score
                global_best_position = positions[i][:]

    print(f"Iteration {step + 1:02d}/{ITERATIONS} → Best Score: {global_best_score:.6f} at
{global_best_position}")

# --- Results ---
print("\n✅ Optimization Complete:")
print(f"→ Best Position Found: {global_best_position}")
print(f"→ Minimum Value: {global_best_score:.6f}")
```

Program 4
Ant Colony Optimization

Algorithm:



PAGE EDGE
DATE : / /

Ant Colony Optimization

Pseudocode

1. Initialize parameters →
   → No of ants
   → No of iterations
   → Pheromone evaporation rate (
   → Pheromone deposit factor
   → α and β

2. Calculate the distance matrix between all pairs of cities.

3. Initialize pheromones trails on all edges to a small positive value.

4. Store the best tours so far and its length → initialize (very large value)

5. Loop for a ~~specialized~~ specified no. of iterations:
   → for each ant:
   construct a tour based on heuristic and pheromone level
   Calculate tour length
   Update best round so far.
   Evaporate pheromone on all edges
   Deposit pheromone on the edges of the tours, with more pheromone deposited on shorter tours.

4. Return solution

Input Matrix

0	50.17	82.42	32.75	83.19
35.44	86.88	79.11	43.17	66.19
84.52	59.01	18.45	47.02	93.01
81.42	30.94	60.79	85.59	59.71

Output

~~Best Tour found: Length → 386.429~~

Iteration 50/200: Best Tour length: ~~368.48~~ 390.90
Iteration 100/200 ~~386.44~~ 386.47
Iteration 150/200 ~~386.443~~ 386.43
Iteration 200/200 ~~386.45~~ 386.43

Best Tour length: ~~368~~ 386.43

Code:

```
import numpy as np

num_cities = 20

np.random.seed(42) # for reproducibility
city_coordinates = np.random.rand(num_cities, 2) * 100 # Coordinates between 0 and 100

# 1. Define ACO parameters
num_ants = 50
num_iterations = 200
pheromone_evaporation_rate = 0.5
pheromone_deposit_factor = 1.0

# 2. Calculate the distance matrix
distance_matrix = np.linalg.norm(city_coordinates[:, np.newaxis, :] - city_coordinates[np.newaxis, :,
:], axis=2)

# Initialize pheromone trails
pheromone_trails = np.ones((num_cities, num_cities)) * 0.1

# Store the best tour found
best_tour = None
best_tour_length = float('inf')

for iteration in range(num_iterations):
    all_tours = []
    all_tour_lengths = []

    for ant in range(num_ants):
        # 3. Implement ant movement and 4. Tour construction
        current_city = np.random.randint(num_cities)
        tour = [current_city]
        visited_cities = {current_city}

        while len(tour) < num_cities:
            possible_next_cities = np.array([city for city in range(num_cities) if city not in visited_cities])
            if len(possible_next_cities) == 0:
                break

            # Calculate probabilities
            pheromone_values = pheromone_trails[current_city, possible_next_cities]
            heuristic_values = 1.0 / (distance_matrix[current_city, possible_next_cities] + 1e-9) # Add a
small constant to avoid division by zero
            probabilities = (pheromone_values**1.0) * (heuristic_values**5.0) # Alpha and Beta
parameters (typically between 1 and 5)
            probabilities /= probabilities.sum()
```

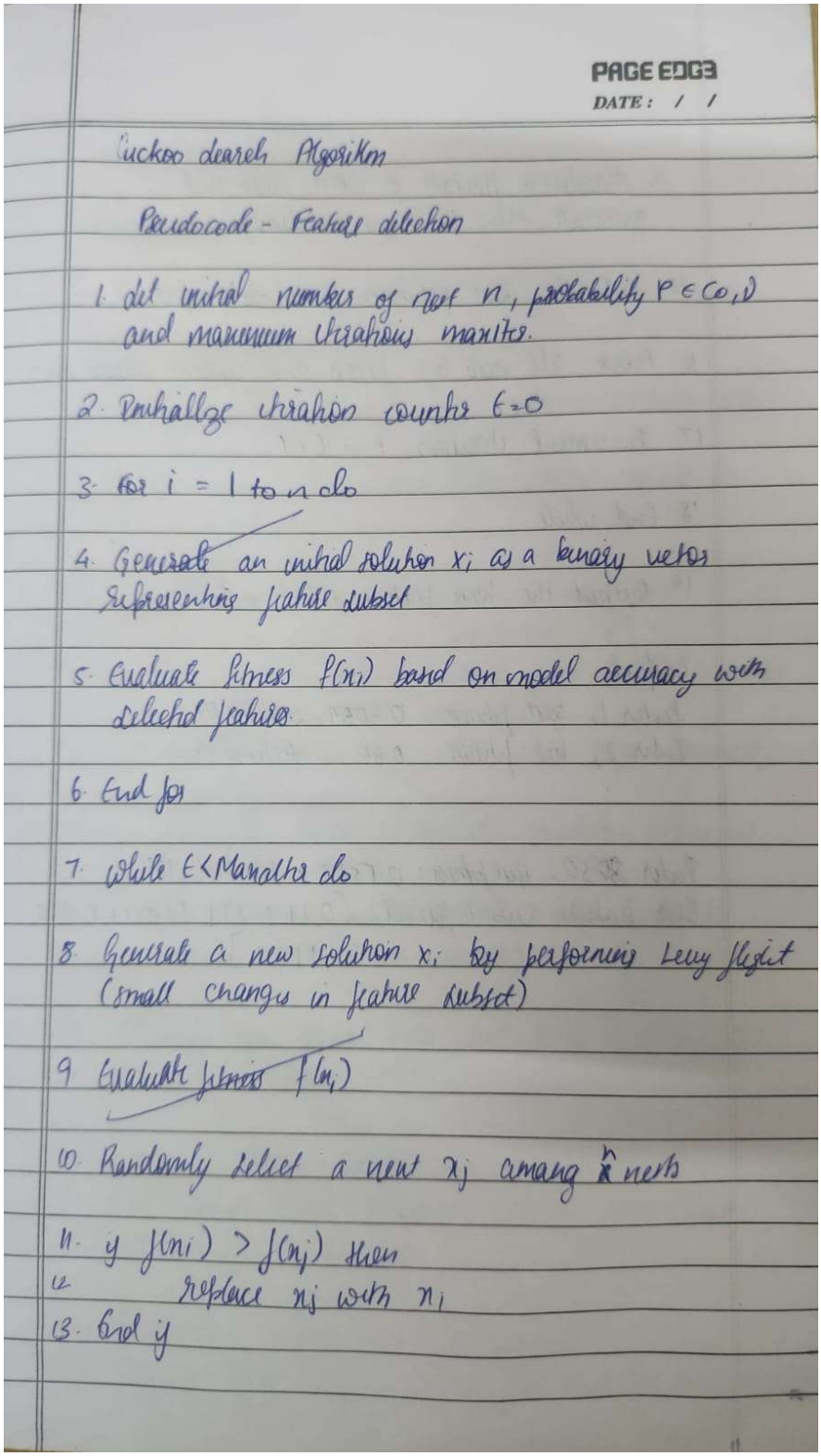
```
            # Select the next city
            next_city = np.random.choice(possible_next_cities, p=probabilities)
            tour.append(next_city)
            visited_cities.add(next_city)
            current_city = next_city

        # Complete the tour by returning to the starting city
        if len(tour) == num_cities:
            tour.append(tour[0])
            tour_length = sum(distance_matrix[tour[i], tour[i+1]] for i in range(num_cities))
            all_tours.append(tour)
            all_tour_lengths.append(tour_length)

            # Update the best tour found so far
            if tour_length < best_tour_length:
                best_tour_length = tour_length
                best_tour = tour

    # 5. Implement pheromone update rule
    pheromone_trails *= (1 - pheromone_evaporation_rate)

    # Deposit pheromone
    for tour, tour_length in zip(all_tours, all_tour_lengths):
        if tour_length > 0: # Avoid division by zero
            pheromone_deposit = pheromone_deposit_factor / tour_length
            for i in range(num_cities):
                pheromone_trails[tour[i], tour[i+1]] += pheromone_deposit
            pheromone_trails[tour[num_cities], tour[0]] += pheromone_deposit # Deposit on the edge
returning to the start

    if (iteration + 1) % 50 == 0:
        print(f"Iteration {iteration + 1}/{num_iterations}, Best tour length: {best_tour_length:.2f}")

# 7. Keep track of the best tour found so far (already done within the loop)

print("\nACO algorithm completed.")
print("Best tour found:", best_tour)
print("Best tour length:", best_tour_length)
```

```
Iteration 50/200, Best tour length: 386.43
Iteration 100/200, Best tour length: 386.43
Iteration 150/200, Best tour length: 386.43
Iteration 200/200, Best tour length: 386.43

ACO algorithm completed.
```

Best tour found: [4, np.int64(12), np.int64(0), np.int64(16), np.int64(5), np.int64(3), np.int64(13), np.int64(8), np.int64(11), np.int64(7), np.int64(2), np.int64(18), np.int64(9), np.int64(15), np.int64(10), np.int64(14), np.int64(6), np.int64(19), np.int64(1), np.int64(17), 4]
Best tour length: 386.4296894765134

Program 5
Cuckoo Search

Algorithm:



PAGE EDGE
DATE : / /

Cuckoo search Algorithm

Pseudocode - Feature selection

1. Set initial number of nest n, probability $P \in (0,1)$ and maximum iteration maxIter.
2. Initialize iteration counter $t = 0$
3. for i = 1 to n do
4. Generate an initial solution x_i as a binary vector representing feature subset
5. Evaluate fitness $f(x_i)$ based on model accuracy with selected features
6. End for
7. while t < MaxIter do
8. Generate a new solution x_i by performing Levy flight (small changes in feature subset)
9. Evaluate fitness $f(x_i)$
10. Randomly select a nest x_j among n nests
11. if $f(x_i) > f(x_j)$ then
12. replace x_j with x_i
13. End if

14 Abandon a fraction of worst nests and generate new solutions randomly

15 Keep the best solution found ~~so~~ so far.

16 Rank all nests by fitness and update current best

17 Increment iteration $t := t+1$

18 End while

19 Output the best feature subset & best

Output:

Iter 1, best fitness : 0.4039, selected 14

Iter 2, best fitness 0.44 selected 14

⋮

Iter ~~30~~ 50, Best fitness : 0.5588, selected : 18

Best feature subset found : [0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1]

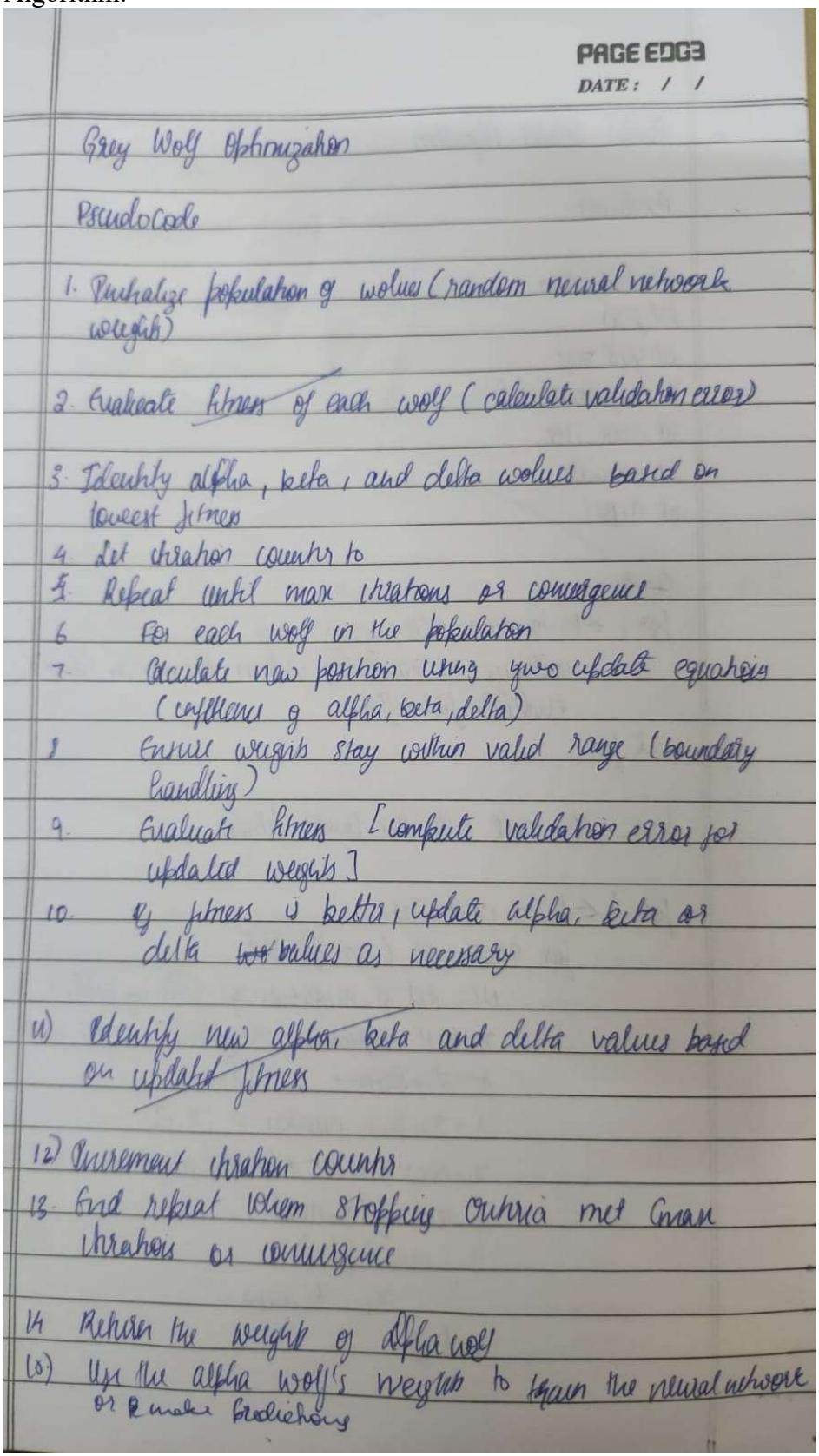
Code:

```
import numpy as np

# Objective function
def objective(x):
    return np.sum(x**2)  # Example (Sphere); replace with your function

# Levy flight
def levy_flight(Lambda):
    u = np.random.randn() * 0.01
    v = np.random.randn()
    step = u / (abs(v)**(1/Lambda))
    return step

def cuckoo_search(n=20, dim=2, lb=-10, ub=10, pa=0.25, max_iter=200):
    nests = np.random.uniform(lb, ub, (n, dim))
    fitness = np.array([objective(x) for x in nests])

    for _ in range(max_iter):
        # Generate new cuckoo by levy flights
        cuckoo = nests[np.random.randint(n)] + levy_flight(1.5) * np.random.randn(dim)
        cuckoo = np.clip(cuckoo, lb, ub)
        f_cuckoo = objective(cuckoo)

        # Random nest to compare
        j = np.random.randint(n)
        if f_cuckoo < fitness[j]:
            nests[j] = cuckoo
            fitness[j] = f_cuckoo

        # Abandon worst nests
        abandon = np.random.rand(n, dim) < pa
        steps = np.random.randn(n, dim) * (nests[np.random.permutation(n)] - nests)
        new_nests = nests + abandon * steps
        new_nests = np.clip(new_nests, lb, ub)

        new_fitness = np.array([objective(x) for x in new_nests])

        # Replace improved ones
        for i in range(n):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

    best_idx = np.argmin(fitness)
    return nests[best_idx], fitness[best_idx]


best_cs, fit_cs = cuckoo_search()
```

```
print("Cuckoo Search Best Solution:", best_cs)
print("Fitness:", fit_cs)
```

Program 6
Grey Wolf

Algorithm:



PAGE EDGE
DATE : / /

Grey Wolf Optimization

Pseudocode

1. Initialize population of wolves (random neural network weights)
2. Evaluate fitness of each wolf (calculate validation error)
3. Identify alpha, beta, and delta wolves based on lowest fitness
4. Set iteration counter to
5. Repeat until max iterations or convergence
6. For each wolf in the population
7. Calculate new position using two update equations (influence of alpha, beta, delta)
8. Ensure weights stay within valid range (boundary handling)
9. Evaluate fitness [compute validation error for updated weights]
10. If fitness is better, update alpha, beta or delta values as necessary
11) Identify new alpha, beta and delta values based on updated fitness
12) Increment iteration counter
13. End repeat when stopping criteria met (max iterations or convergence
14 Return the weights of alpha wolf
15) Use the alpha wolf's weights to train the neural network or make predictions

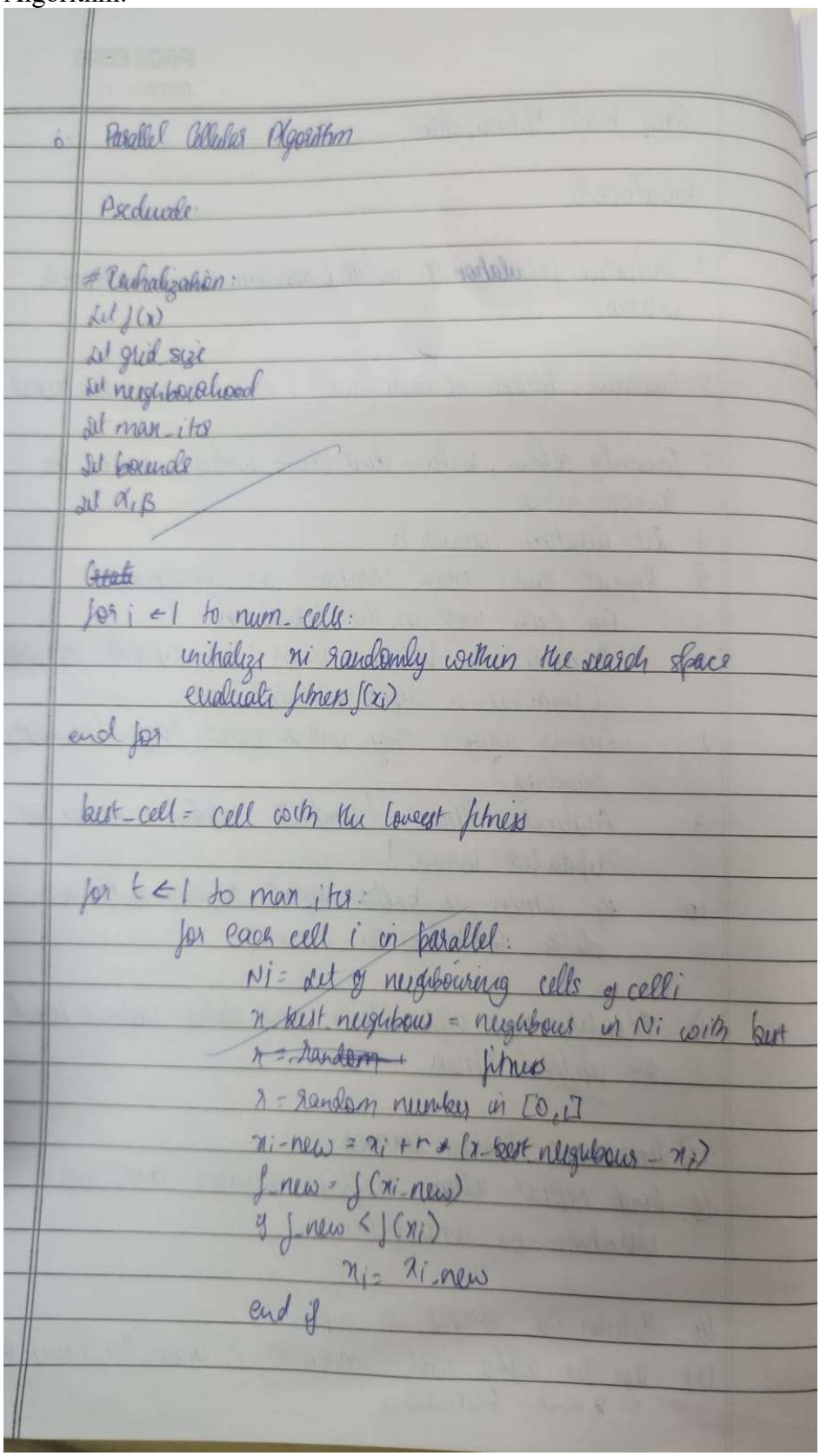
Code:

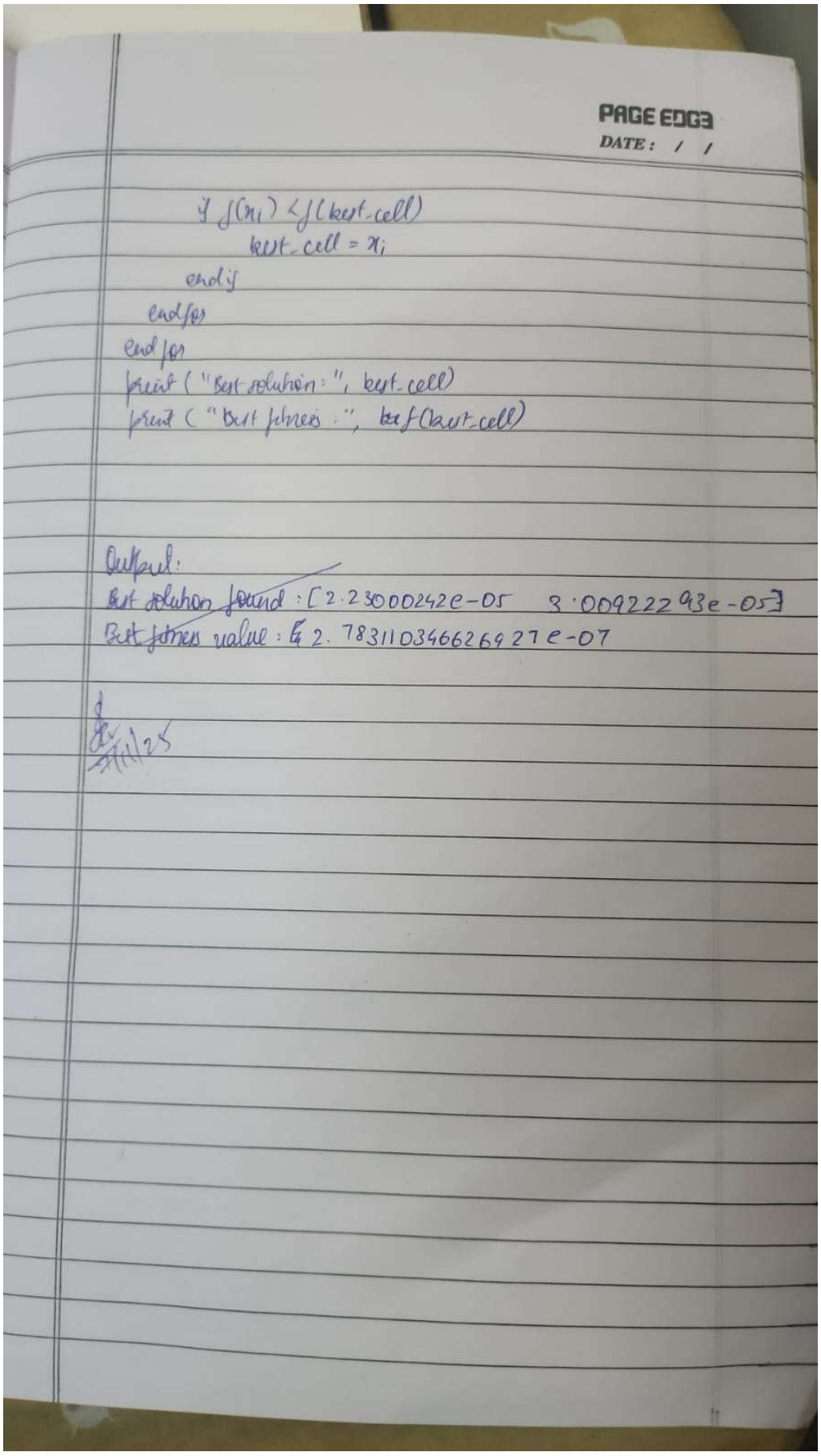
```
import numpy as np

def objective(x):
    return np.sum(x**2)  # Example (Sphere); replace with your function

def gwo(n=20, dim=2, lb=-10, ub=10, max_iter=200):
    wolves = np.random.uniform(lb, ub, (n, dim))
    fitness = np.array([objective(w) for w in wolves])

    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = np.inf, np.inf, np.inf

    for _ in range(max_iter):
        for i, wolf in enumerate(wolves):
            score = fitness[i]

            if score < alpha_score:
                alpha_score, alpha = score, wolf.copy()
            elif score < beta_score:
                beta_score, beta = score, wolf.copy()
            elif score < delta_score:
                delta_score, delta = score, wolf.copy()

        a = 2 - _ * (2/max_iter)

        for i in range(n):
            for leader, leader_pos in zip([alpha, beta, delta], [alpha, beta, delta]):
                r1, r2 = np.random.rand(), np.random.rand()
                A = 2*a*r1 - a
                C = 2*r2
                D = abs(C*leader_pos - wolves[i])
                X = leader_pos - A*D

                if leader is alpha:
                    X1 = X
                elif leader is beta:
                    X2 = X
                else:
                    X3 = X

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

        wolves = np.clip(wolves, lb, ub)
        fitness = np.array([objective(w) for w in wolves])

    return alpha, alpha_score
```

```
best_gwo, fit_gwo = gwo()
print("Grey Wolf Best Solution:", best_gwo)
print("Fitness:", fit_gwo)
```

Program 7
Parallel Cellular

Algorithm:





PAGE EDGE
DATE : / /

```
        if f(xi) < f(best_cell)
            best_cell = xi
        endif
    endfor
endfor
print("Best solution:", best_cell)
print("Best fitness:", f(best_cell))
```

Output:

~~Best solution found~~ : [2.23000242e-05 3.00922293e-05]

~~Best fitness value~~ : 2.7831103466269271e-07

Code:

```
import numpy as np

# Objective function (Rastrigin)
def f(x):
    return 10 * len(x) + np.sum(x**2 - 10 * np.cos(2 * np.pi * x))


# Moore neighborhood with input 3/5/7/9...
def get_neighbors_indices(i, j, num_rows, num_cols, neigh_size=3):
    """
    neigh_size must be odd: 3 → 3x3, 5 → 5x5, 7 → 7x7 Moore neighborhood.
    """
    if neigh_size % 2 == 0 or neigh_size < 3:
        raise ValueError("neigh_size must be an odd integer >= 3")

    radius = (neigh_size - 1) // 2  # 3→1, 5→2, 7→3

    neighbors = []
    for dx in range(-radius, radius + 1):
        for dy in range(-radius, radius + 1):

            if dx == 0 and dy == 0:
                continue  # skip center

            ni = (i + dx) % num_rows
            nj = (j + dy) % num_cols
            neighbors.append((ni, nj))

    return neighbors


# ------------------------
# PARAMETERS
# ------------------------
num_rows = 10
num_cols = 10
dim = 2
max_iter = 100
bounds = [-5.12, 5.12]

neigh_size = 5   # can be 3, 5, 7, 9...


# ------------------------
# Step 1: Initialize population
# ------------------------
grid = np.random.uniform(bounds[0], bounds[1], size=(num_rows, num_cols, dim))
fitness = np.zeros((num_rows, num_cols))
```

```
for i in range(num_rows):
    for j in range(num_cols):
        fitness[i, j] = f(grid[i, j])

# Best solution so far
best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
best_cell = grid[best_pos]
best_fitness = fitness[best_pos]


# ------------------------
# Step 2: Parallel Cellular Evolution
# ------------------------
for t in range(max_iter):

    new_grid = grid.copy()
    new_fitness = fitness.copy()

    for i in range(num_rows):
        for j in range(num_cols):

            # Neighborhood (custom size)
            neighbors_idx = get_neighbors_indices(
                i, j, num_rows, num_cols, neigh_size=neigh_size
            )

            neighbors = [grid[ni, nj] for ni, nj in neighbors_idx]
            neighbor_fitness = [fitness[ni, nj] for ni, nj in neighbors_idx]

            # Best neighbor
            best_neighbor = neighbors[np.argmin(neighbor_fitness)]

            # Movement toward best neighbor
            r = np.random.rand()
            xi_new = grid[i, j] + r * (best_neighbor - grid[i, j])

            # Boundary enforcement
            xi_new = np.clip(xi_new, bounds[0], bounds[1])

            # Evaluate
            f_new = f(xi_new)

            # Synchronous selection (update goes to new_grid only)
            if f_new < fitness[i, j]:
                new_grid[i, j] = xi_new
                new_fitness[i, j] = f_new

                # Global best
```

```
            if f_new < best_fitness:
                best_fitness = f_new
                best_cell = xi_new

    # Synchronous update
    grid = new_grid
    fitness = new_fitness


# ------------------------
# OUTPUT
# ------------------------
print("Best solution found:", best_cell)
print("Best fitness value:", best_fitness)
```