VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by:

Harbakhshish Singh Arora

USN: 1BM23CS104

in partial fulfilment 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 **Harbakhshish Singh Arora (1BM23CS104),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Sowmya T Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GitHub Link:
_https://github.com/harry13-bax/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.

Algorithm:

total distance. If best fitness in current generation is greater than global best then update global best.

(11) output : Print the best solution and the total distance.

Tracking

Enter the no. of Cities: 4

Enter the distance row by row:

Row 1 = 20. 16. 10. 32

Row 2 = 10 0 25 35

Row 3 = 16 0 35 20

Row 4 = 20 45 30 0

Enter population size = 4

Enter no. of generation = 3

Generation 1

Parent	fitness	Mate	Crossover
[3, 1, 2, 0]	117	(3, 0, 2, 1)	(0, 2)
[1, 3, 0, 2]	(illegible)	(illegible)	(0, 3)

After Crossover	Mutation	offspring	Fitness
(3, 0, 1, 2)	(2, 0)	(0, 3, 1, 2)	147
(1, 3, 0, 2)	(3, 1)	(1, 2, 0, 3)	117

Generation 2

Parent	fitness	Mate	Crossover
(0, 3, 1, 2)	117	(1, 2, 0, 3)	(0, 2)

After Crossover	Mutation	offspring	f
(0, 3, 1, 2)	(2, 0)	(3, 0, 1, 2)	6

(4) fill new population

- Add children until the population is restored (Pop-Size)

Step 3 - Replace the Population

- the new population becomes the current population

Step 4: Stopping Condition

- After Generations iterations, stop
- Report the best individual (highest fitness).

So it finally prints best fitness per generation.

(Ignore, decoded integer x, and fitness = x^2)

output:

Find Best Route:

Route: [1, 5, 0, 2, 8, 3, 7, 4, 6, 4]

Distance = 293.68

Gen 0 : Best Distance = 366.48

Gen 50 : Best Distance = 315.37

Gen 100 : " " = 284.76

Gen 200 : " " = 294.576

Gen 250 : " " = 284.76

Gen 300 : " " = 294.76

Gen 350 : " " = 294.76

Gen 400 : " " = 293.68

Code:

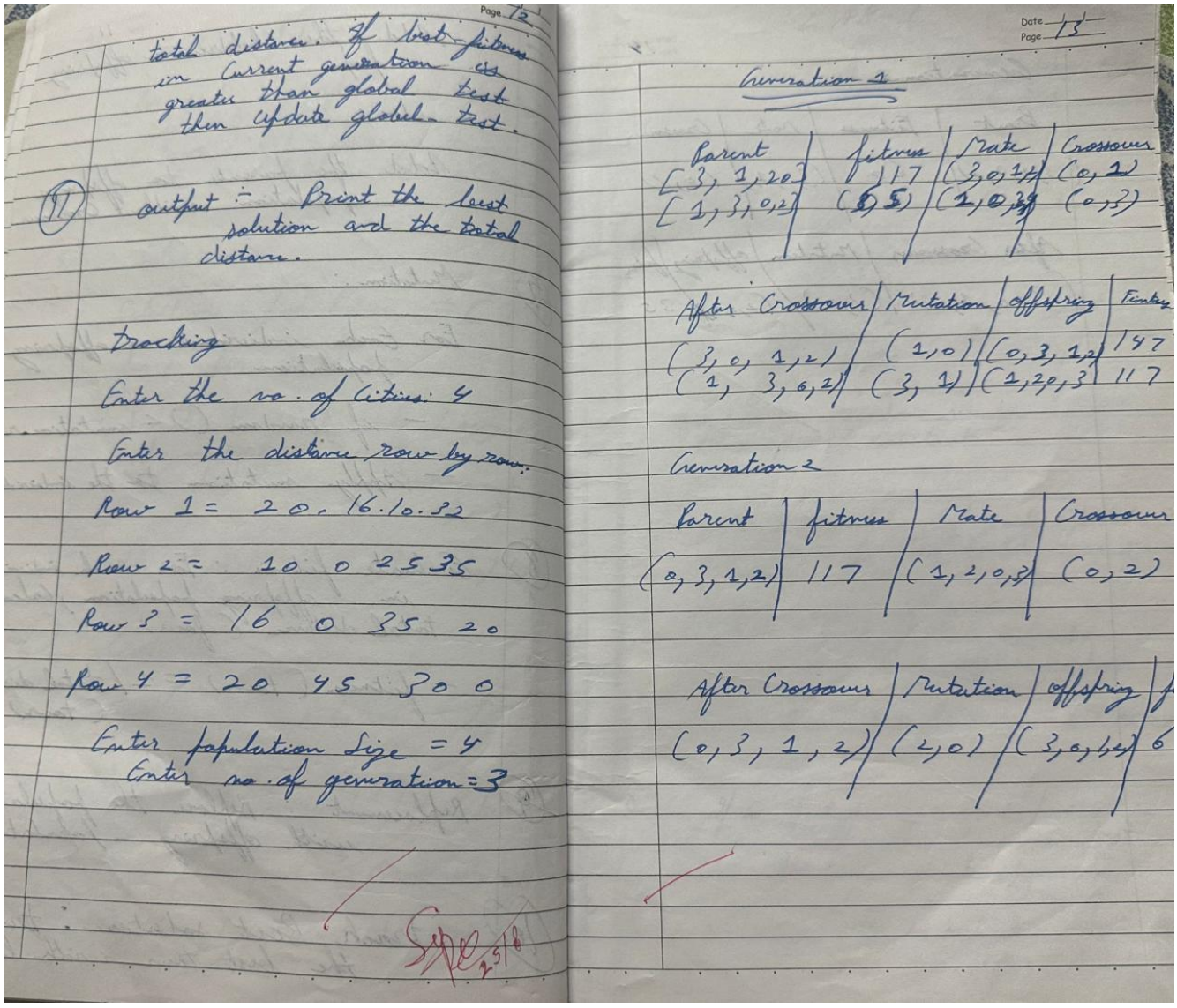
```
import random
import math

NUM_CITIES = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1

cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(NUM_CITIES)]

def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def total_distance(route):     dist = 0     for i in range(len(route)):         dist
+= distance(cities[route[i]], cities[route[(i + 1) % NUM_CITIES]])     return
dist

def fitness(route):     return 1 /
total_distance(route)

def generate_population():     return [random.sample(range(NUM_CITIES), NUM_CITIES) for _ in
range(POPULATION_SIZE)]

def selection(population, fitnesses):
    selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
return selected

def crossover(parent1, parent2):     start, end =
sorted(random.sample(range(NUM_CITIES), 2))     child =
[None] * NUM_CITIES     child[start:end] = parent1[start:end]
pointer = 0     for gene in parent2:         if gene not in child:
while child[pointer] is not None:
                pointer += 1
child[pointer] = gene     return
child

def mutate(route):     if random.random() <
MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
route[i], route[j] = route[j], route[i]
    return route
```

```
def genetic_algorithm():     population
= generate_population()
best_route = None
   best_distance = float('inf')

   for generation in range(GENERATIONS):
      fitnesses = [fitness(ind) for ind in population]
new_population = []

      for _ in range(POPULATION_SIZE):
         parent1, parent2 = selection(population, fitnesses)[:2]
child = crossover(parent1, parent2)            child =
mutate(child)
         new_population.append(child)

      population = new_population

      for route in population:
         dist = total_distance(route)
if dist < best_distance:
best_distance = dist
            best_route = route

      if generation % 50 == 0:            print(f"Generation {generation}: Best
Distance = {round(best_distance, 2)}")

   print("\n  Final Best Route:")     print("Route:",
best_route)
   print("Distance:", round(best_distance, 2))

genetic_algorithm()
```

<u>Program 2:</u> 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.

Algorithm:



Code:

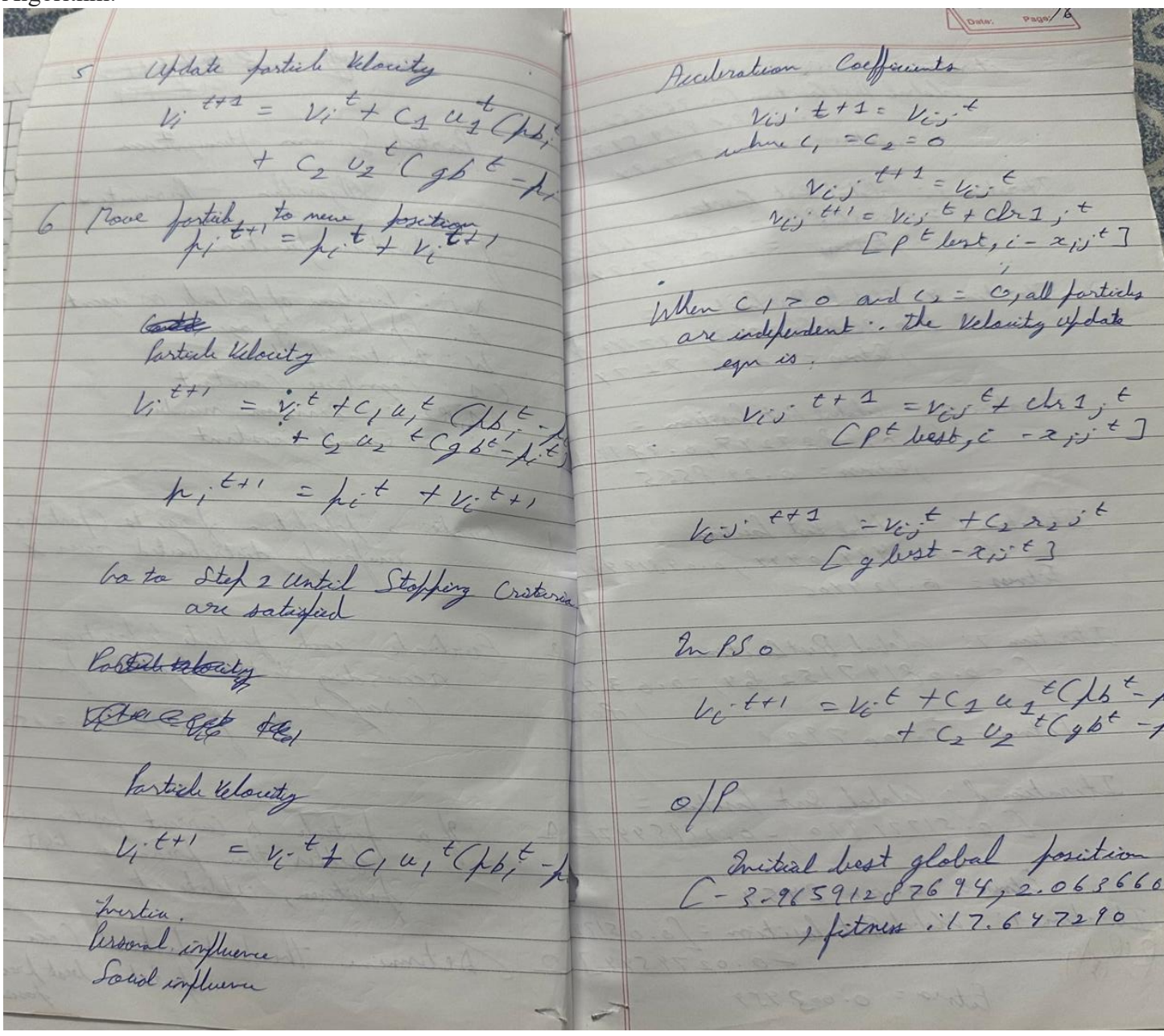
```
import random
import math

# Parameters
NUM_CITIES = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1

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

def distance(city1, city2):     return math.sqrt((city1[0] -
city2[0])**2 + (city1[1] - city2[1])**2)

def total_distance(route):     dist = 0     for i in range(len(route)):         dist
+= distance(cities[route[i]], cities[route[(i + 1) % NUM_CITIES]])     return
dist

def fitness(route):     return 1 /
total_distance(route)

def generate_population():     return [random.sample(range(NUM_CITIES), NUM_CITIES) for _ in
range(POPULATION_SIZE)]

def selection(population, fitnesses):
   selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
return selected

def crossover(parent1, parent2):     start, end =
sorted(random.sample(range(NUM_CITIES), 2))     child =
[None] * NUM_CITIES     child[start:end] = parent1[start:end]
pointer = 0     for gene in parent2:         if gene not in child:
while child[pointer] is not None:
            pointer += 1
child[pointer] = gene     return
child

def mutate(route):     if random.random() <
MUTATION_RATE:
```

```
        i, j = random.sample(range(NUM_CITIES), 2)
route[i], route[j] = route[j], route[i]
    return route

def genetic_algorithm():     population
= generate_population()
best_route = None
    best_distance = float('inf')

    for generation in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
new_population = []

        for _ in range(POPULATION_SIZE):
            parent1, parent2 = selection(population, fitnesses)[:2]
child = crossover(parent1, parent2)             child =
mutate(child)
            new_population.append(child)

        population = new_population

        # Track best         for
route in population:
            dist = total_distance(route)
if dist < best_distance:
best_distance = dist
                best_route = route

        if generation % 50 == 0:             print(f"Generation {generation}: Best
Distance = {round(best_distance, 2)}")

    print("\n  Final Best Route:")     print("Route:",
best_route)
    print("Distance:", round(best_distance, 2))

genetic_algorithm()
```

Program 3: Particle Swarm Optimization for Function Optimization:

Particle Swarm Optimization (PSO) is inspired by the social behaviour 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.

Algorithm:



iteration 2:
Global Best position [0.36866206
1.37889895977
, Fitness = 2.037274

Iteration 3: Global Best position
= [0.368662065 2, 1.37889
, Fitness = 2.037274

Iteration 4: Global Best position =
[0.36866206 3, 1.37889895
, Fitness = 2.037274

Iteration 5: Global Best position =
[-0.38272477, 0.3899 82
, Fitness = 0.298565

Iteration 6: Global Best position
[-0.3994 0944, 0.04819441]
Fitness = 0.124406

Iteration 7: Global Best position
[-0.08197152 804, -0.0 347
1 80d

Fitness = 0.007923

Iteration 8: Global Best position =
[0.51774930, -0.27954471

Fitness: 0.003459

Iteration 9: Global Best position = [0.05
-0.027954

Fitness = 0.003459

Code:

```
import random
import numpy as np

def fitness_function(position):
    x, y = position
    return -(x**2 + y**2 - 4*x - 6*y)

def particle_swarm_optimization(dimensions, num_particles, max_iterations, threshold):
    w = 0.5
c1 = 1.2
c2 = 1.4

    swarm = []     for _ in range(num_particles):         position =
np.random.uniform(-10, 10, size=dimensions)         velocity =
np.random.uniform(-1, 1, size=dimensions)
pbest_position = position.copy()         pbest_fitness =
fitness_function(position)
        swarm.append({'position': position, 'velocity': velocity,
                      'pbest_position': pbest_position, 'pbest_fitness': pbest_fitness})

    gbest_position = np.zeros(dimensions)
    gbest_fitness = -float('inf')

    for i in range(max_iterations):         for p in
swarm:             fitness =
fitness_function(p['position'])

            if fitness > p['pbest_fitness']:
p['pbest_fitness'] = fitness
                p['pbest_position'] = p['position'].copy()

            if fitness > gbest_fitness:
gbest_fitness = fitness
                gbest_position = p['position'].copy()
```

```
        if gbest_fitness >= threshold:
print(f"Early stopping at iteration {i}")
break

        for p in swarm:
            rand1 = random.random()
rand2 = random.random()

            inertia = w * p['velocity']             cognitive = c1 * rand1 *
(p['pbest_position'] - p['position'])             social = c2 * rand2 *
(gbest_position - p['position'])

            p['velocity'] = inertia + cognitive + social
            p['position'] = p['position'] + p['velocity']

    print("SOLUTION FOUND:")
    print(f"  Position: {gbest_position}")
print(f"  Fitness: {gbest_fitness}")
    return gbest_position, gbest_fitness

particle_swarm_optimization(dimensions=2, num_particles=20, max_iterations=5000, threshold=2)
```

Program4: Ant Colony Optimization for the Traveling Salesman Problem:

The foraging behaviour of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

Ant Colony optimization

Pseudocode

function ACO_Algorithm()

initialize_Pheromones()
while (termination Condition is not met)

for each ant in Colony

tour = Construct_Solution_for_AntCount

add tour to all_tours_this_iteration

end for

Update_Pheromones Call_tours_this_iteration)

Update_Best_Tour_Found()

end while

return Best_Tour_Found

end Function

function Construct_solution_for_AntCount

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, and visited_cities)

tour add (next_City)

ant move_to (next_City)

end (while)
return tour
end function

function Update_Pheromones Call_tour_this_iteration

for each path (i, j) on the map

Path (i, j) Pheromone = (1 - evaporation) Path (i, j)

end for

for each tour in all_tours_this_iteration

Pheromone_to_add = Calculate_Pheromone_amount (tour_length)

for each path (i, j) in the tour

Path (i, j) Pheromone += Pheromone_to_add

end for

end function

$$P_{ij} = \frac{(\tau_{ij}^{\alpha})(n_{ij}^{\beta} s^{\gamma})}{\sum_{k \in F} (\tau_{ij}^{\alpha})(n_{ij}^{\beta} s^{\gamma})}$$

τ_{ij} = Pheromone Level

n_{ij} = heuristic info

Parameter = α, β, s → evaporation Coefficient
→ Distance Coefficient
Pheromone Coefficient

O/P

n_ants = 10

n_iteration = 100

Alpha = 1.0 | Pheromone
Beta = 5.0 | intens
evaporation = 0.5

Result = -

Best tour = [0, 4, 3, 1, 2]

Best tour length = 80

Distance =

0	0	29	20	21	16
1	29	0	15	17	28
2	20	15	0	28	23
3	21	12	28	0	12
4	16	28	23	12	0

See Rec 11.9

Code:

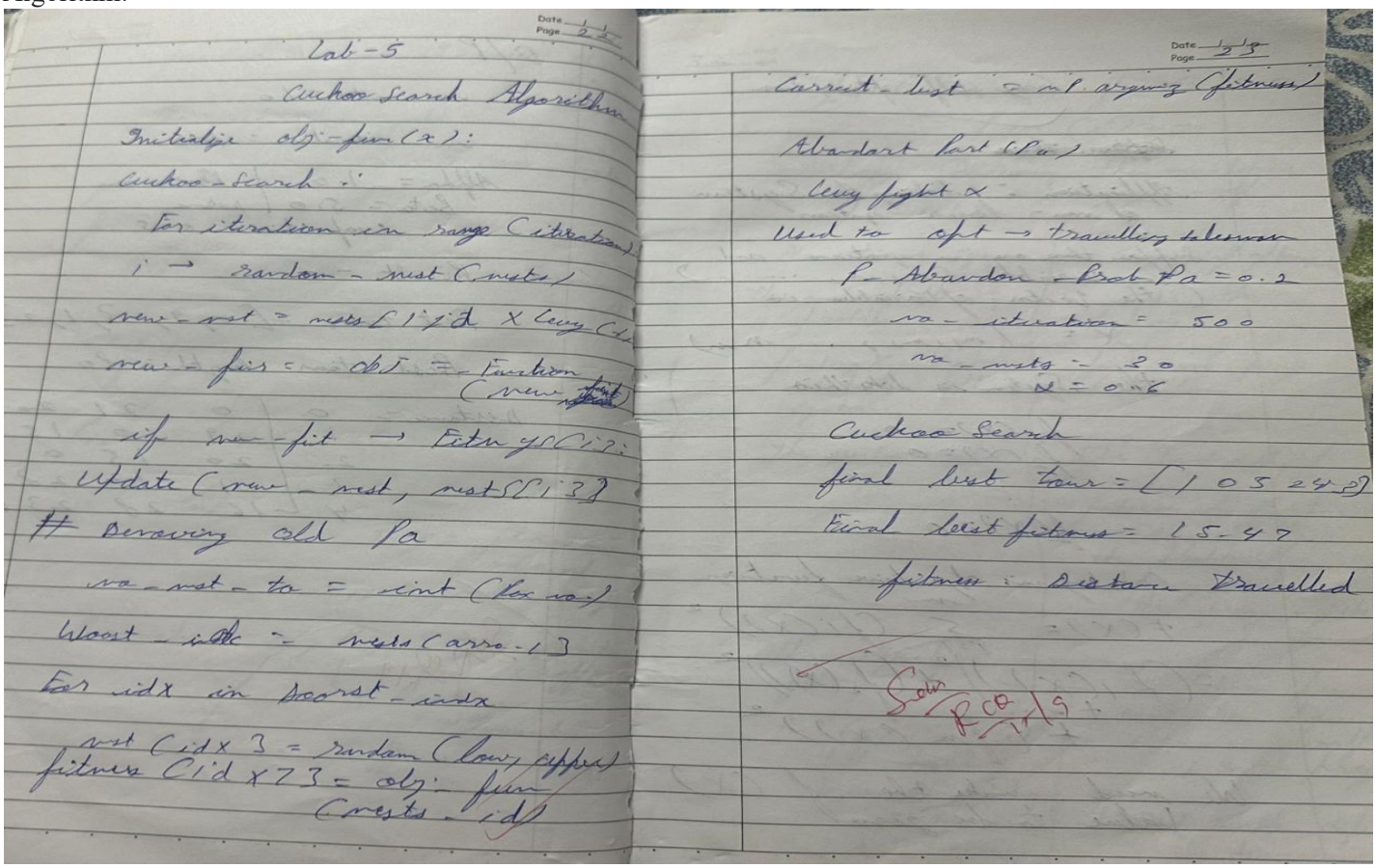
```
import numpy as np
import random


NUM_CITIES = 5
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
RHO = 0.5
Q = 100


cities = np.random.rand(NUM_CITIES, 2)
distance_matrix = np.linalg.norm(cities[:, None] - cities, axis=2) pheromone_matrix
= np.ones((NUM_CITIES, NUM_CITIES))


def calculate_probabilities(current_city, visited):
    probabilities = []        for next_city in
range(NUM_CITIES):            if next_city
in visited:
            probabilities.append(0)
else:
            pheromone = pheromone_matrix[current_city][next_city] ** ALPHA
heuristic = (1 / distance_matrix[current_city][next_city]) ** BETA
probabilities.append(pheromone * heuristic)      total = sum(probabilities)
return [p / total if total > 0 else 0 for p in probabilities]

def construct_tour():      start_city =
random.randint(0, NUM_CITIES - 1)      tour =
[start_city]      while len(tour) < NUM_CITIES:
        probs = calculate_probabilities(tour[-1], tour)          next_city =
np.random.choice(range(NUM_CITIES), p=probs)
tour.append(next_city)
    return tour
```

```
def compute_tour_length(tour):
    return sum(distance_matrix[tour[i]][tour[(i + 1) % NUM_CITIES]] for i in range(NUM_CITIES))


best_tour = None
best_length = float('inf')

for iteration in range(NUM_ITERATIONS):
    all_tours = []

    for _ in range(NUM_ANTS):
        tour = construct_tour()         length
= compute_tour_length(tour)
        all_tours.append((tour, length))

        if length < best_length:
best_tour = tour
            best_length = length


    pheromone_matrix *= (1 - RHO)


    for tour, length in all_tours:
for i in range(NUM_CITIES):
            a, b = tour[i], tour[(i + 1) % NUM_CITIES]
pheromone_matrix[a][b] += Q / length
            pheromone_matrix[b][a] += Q / length  # symmetric TSP


clean_tour = [int(city) for city in best_tour]
print("Best tour:", clean_tour)

print("Best length:", round(best_length, 4))
```

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.

Algorithm:



Code:

```
import numpy as np
import math

# --- Levy flight --- def levy_flight(Lambda, dim):     sigma =
(math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
          (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
u = np.random.normal(0, sigma, size=dim)     v = np.random.normal(0, 1, size=dim)
step = u / abs(v)**(1 / Lambda)
   return step

# --- Sigmoid for binary conversion --- def
sigmoid(x):
   return 1 / (1 + np.exp(-x))

# --- Fitness function for knapsack --- def
fitness_function(x_bin, weights, values, capacity):
total_weight = np.sum(x_bin * weights)     total_value
= np.sum(x_bin * values)     if total_weight > capacity:
return -1  # Penalize overweight solutions heavily
else:        return total_value

# --- Cuckoo Search for Binary Knapsack --- def cuckoo_search_knapsack(weights,
values, capacity, n=25, Pa=0.25, Maxt=500):
   dim = len(weights)
   # Initialize nests (continuous vectors)
   nests = np.random.uniform(low=-1, high=1, size=(n, dim))
   # Convert to binary solutions     nests_bin = np.array([sigmoid(nest) >
np.random.rand(dim) for nest in nests])     fitness = np.array([fitness_function(x,
weights, values, capacity) for x in nests_bin])

   best_idx = np.argmax(fitness)
best_nest = nests[best_idx].copy()
best_bin = nests_bin[best_idx].copy()
   best_fitness = fitness[best_idx]
```

```
    t = 0     while t <
Maxt:         for i in
range(n):
            # Generate new solution by Levy flight
            step = levy_flight(1.5, dim)
new_nest = nests[i] + 0.01 * step             #
Convert new_nest to binary
            new_bin = sigmoid(new_nest) > np.random.rand(dim)
            new_fitness = fitness_function(new_bin, weights, values, capacity)

            # If new solution is better, replace
            if new_fitness > fitness[i]:
nests[i] = new_nest
nests_bin[i] = new_bin
                fitness[i] = new_fitness

                if new_fitness > best_fitness:
best_fitness = new_fitness                    best_nest
= new_nest.copy()
                    best_bin = new_bin.copy()

        # Abandon fraction Pa of worst nests
num_abandon = int(Pa * n)         worst_indices =
np.argsort(fitness)[:num_abandon]         for idx in
worst_indices:
            nests[idx] = np.random.uniform(-1, 1, dim)
            nests_bin[idx] = sigmoid(nests[idx]) > np.random.rand(dim)
            fitness[idx] = fitness_function(nests_bin[idx], weights, values, capacity)

            if fitness[idx] > best_fitness:
best_fitness = fitness[idx]                best_nest
= nests[idx].copy()
                best_bin = nests_bin[idx].copy()

        t += 1

    return best_bin, best_fitness

if __name__ == "__main__":
    print("Enter the number of items:")
n_items = int(input())

    weights = []
    values = []

    print("Enter the weights of the items (space-separated):")
weights = np.array(list(map(float, input().split())))     if
len(weights) != n_items:
        raise ValueError("Number of weights does not match number of items.")
```

```
    print("Enter the values of the items (space-separated):")
values = np.array(list(map(float, input().split())))     if
len(values) != n_items:
        raise ValueError("Number of values does not match number of items.")

    print("Enter the knapsack capacity:")
    capacity = float(input())

    print("Enter population size (default 25):")
    n = input()
    n = int(n) if n.strip() else 25

    print("Enter abandonment probability Pa (default 0.25):")
    Pa = input()
    Pa = float(Pa) if Pa.strip() else 0.25

    print("Enter maximum iterations Maxt (default 500):")
    Maxt = input()
    Maxt = int(Maxt) if Maxt.strip() else 500

    best_solution, best_value = cuckoo_search_knapsack(weights, values, capacity, n=n, Pa=Pa,
Maxt=Maxt)

    print("\nBest solution (items selected):", best_solution.astype(int))
print("Total value:", best_value)
    print("Total weight:", np.sum(best_solution * weights))
```

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.

Algorithm:

Date 1/1/
Page 27

Pseudo Code

Grey - Wolf - optimizer ()

{

initialize the GWP

X_i → random initial guess

initialize a, A and C

Cal. the fitness of each search agent

X_α = the best search
X_β = the second best search
X_d = the third best search

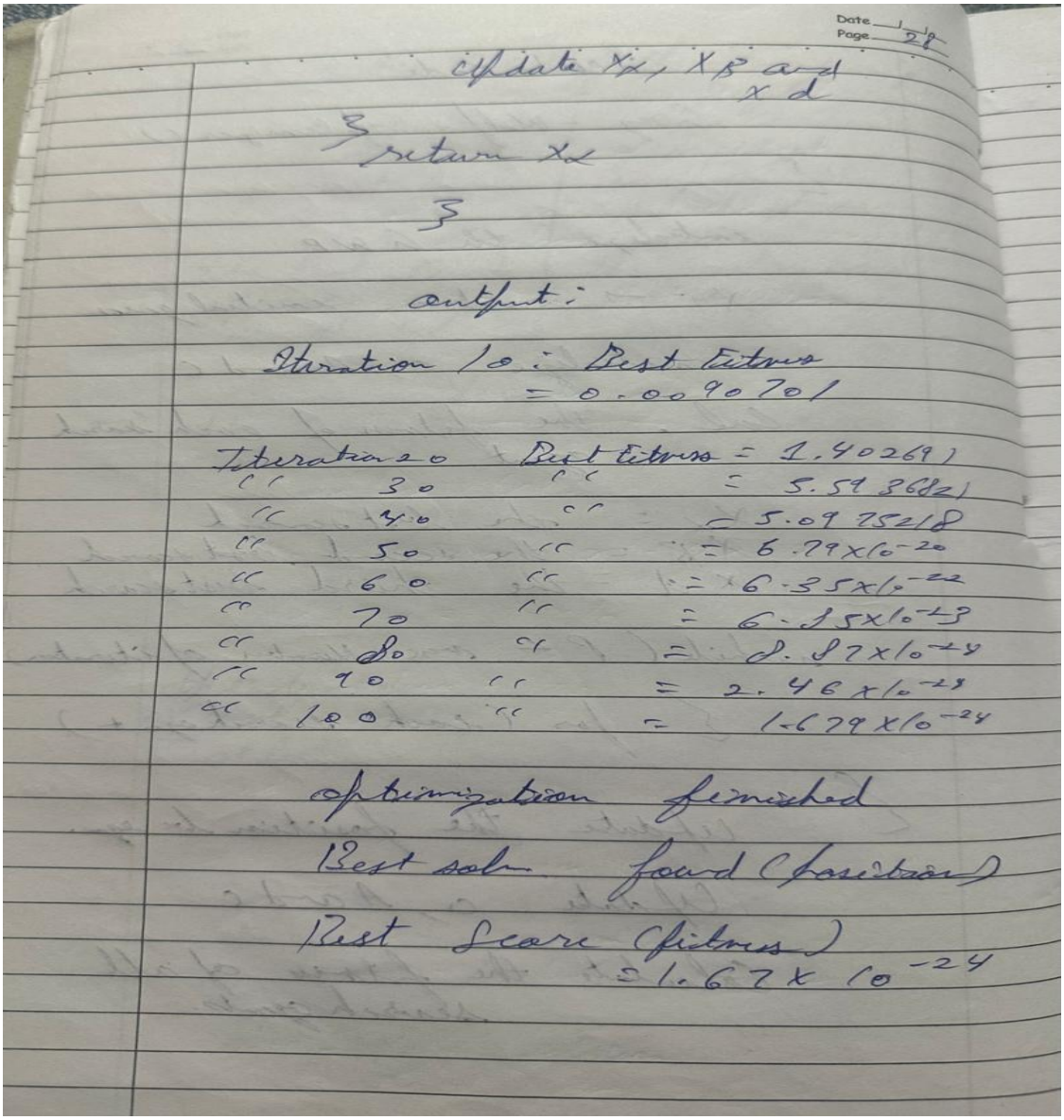
while (P < max - Number of iterations)

{ for (each search agent)

{ Update the position by eqn.

Update a, A and C

Calculate the fitness of all search agents.



Code:

```
import numpy as np
import random

def distance_matrix(cities):
n = len(cities)     dist =
np.zeros((n, n))     for i in
range(n):
        for j in range(n):             dist[i][j] =
np.linalg.norm(np.array(cities[i]) - np.array(cities[j]))     return dist
```

```
def tour_length(tour, dist):     return
sum(dist[tour[i]][tour[(i+1)%len(tour)]] for i in range(len(tour)))

def initialize_population(num_wolves, num_cities):     return
[random.sample(range(num_cities), num_cities) for _ in range(num_wolves)]

def gwo_tsp(cities, num_wolves=20, max_iter=100):
   dist = distance_matrix(cities)     population =
initialize_population(num_wolves, len(cities))     fitness =
[tour_length(tour, dist) for tour in population]

   alpha, beta, delta = sorted(zip(population, fitness), key=lambda x: x[1])[:3]

   for iter in range(max_iter):
a = 2 - iter * (2 / max_iter)
       new_population = []

       for wolf in population:
           new_tour = []            for
i in range(len(cities)):
               r1, r2 = random.random(), random.random()
               A1 = 2 * a * r1 - a
               C1 = 2 * r2
               D_alpha = abs(C1 * alpha[0][i] - wolf[i])
               X1 = alpha[0][i] - A1 * D_alpha

               # Repeat for beta and delta
               # Combine X1, X2, X3 and discretize
new_tour.append(int(X1) % len(cities))

           # Ensure it's a valid permutation            new_tour = list(dict.fromkeys(new_tour))
while len(new_tour) < len(cities):                new_tour.append(random.choice([i for i in
range(len(cities)) if i not in new_tour]))

           new_population.append(new_tour)

       population = new_population
       fitness = [tour_length(tour, dist) for tour in population]
       alpha, beta, delta = sorted(zip(population, fitness), key=lambda x: x[1])[:3]

   return alpha[0], alpha[1]

# Example usage
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)] best_tour,
best_distance = gwo_tsp(cities)
print("Best tour:", best_tour) print("Distance:",
best_distance)
```

Program 7: Parallel Cellular Algorithms and Programs

Parallel Cellular Algorithm (PCA) is inspired by the cooperative behavior and spatial organization observed in natural systems, such as the interactions within biological cells or individuals in a population. In PCA, candidate solutions (particles or individuals) are arranged in a structured grid where each one interacts only with its local neighbors, mimicking localized communication.
This neighborhood-based evolution promotes diversity and prevents premature convergence by maintaining multiple subpopulations that explore different regions of the search space.
PCA is often implemented in parallel to exploit computational efficiency, making it suitable for complex optimization problems where global search and local refinement must be balanced effectively.

Algorithm:

Date
Page 31

Pseudo Code

Algorithm: Laser - dynamics:

initialize grid where each cell
has electrons and photons

for each time - step:

for each cell in the grid

if (all two excited e^-)
and (its neighbours have photons)
Create a new photon in the cell

the electron returns to its ground state
and if
end for

for each cell in the grid:

if - date and remove - old - photons
if - date and decay - old - photons

if (cell has a ground - state e^-)
and random chance is met)

make the e^- excited
end if
end for

Add a few random photons to the
grid to simulate spontaneous emiss

Date 1/1
Page 32

record no. of photons for each step.

end loop;

output:

Starting laser CA simulation

time step: 100/1000, photons: 1032, Population Inversion = 141
" : 200/1000, " : 1135, " = 141
" : 300/1000, " : 1099, " = 141
" : 400/1000, " : 1048, " = 137
" : 500/1000, " : 993, " = 129
" : 600/1000, " : 999, " = 158
" : 700/1000, " : 1060, " = 140
" : 800/1000, " : 1041, " = 139
" : 900/1000, " : 1041, " = 147
" : 1000/1000, " : 1018, " = 141

Code:

```
import numpy as np
import matplotlib.pyplot as plt

class LaserCA:
    """
    A Cellular Automata model for simulating laser dynamics based on the paper
    "Application of Cellular Automata Algorithms to the Parallel Simulation of Laser Dynamics".
    """
    def __init__(self, L, tau_c, tau_a, lambda_p, M=10, stim_threshold=1, noise_photons=1):
        """
        Initializes the Laser CA simulation.

        Args:
            L (int): The side length of the square cellular lattice.
            tau_c (int): The lifetime of photons in time steps. [cite: 421]
            tau_a (int): The lifetime of excited electrons in time steps. [cite: 425]
            lambda_p (float): The pumping probability (0 to 1).
            M (int, optional): Maximum number of photons per cell. Defaults to 10.
            stim_threshold (int, optional): Threshold for stimulated emission. Defaults to 1.
            noise_photons (int, optional): Number of noise photons to add each step.
        """
        # --- System Parameters ---
        self.L = L
        self.tau_c = tau_c
        self.tau_a = tau_a
        self.lambda_p = lambda_p
        self.M = M
        self.stim_threshold = stim_threshold
        self.noise_photons = noise_photons

        # --- CA State Variables ---
        # a_ij(t): State of the electron (0=ground, 1=excited)
        self.electron_states = np.zeros((L, L), dtype=np.int8)

        # tla_ij: Remaining lifetime of an excited electron [cite: 426]
        self.electron_lifetimes = np.zeros((L, L), dtype=np.int32)

        # c_ij(t): Number of photons in the cell
        self.photon_counts = np.zeros((L, L), dtype=np.int32)

        # tlc_ijk: Remaining lifetime of each photon 'k' in cell {i,j} [cite: 422]
        # A 3D array: (L, L, M) where M is the max number of photons per cell
        self.photon_lifetimes = np.zeros((L, L, M), dtype=np.int32)

    def _get_moore_neighbors_sum(self, i, j):
```

```
        """
        Calculates the sum of photons in the Moore neighborhood of cell (i, j)
        using periodic boundary conditions. [cite: 394, 479]
        """
        total = 0
        # Loop over the 3x3 neighborhood
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                # Skip the center cell itself
                if di == 0 and dj == 0:
                    continue

                # Apply periodic boundary conditions using modulo operator
                ni, nj = (i + di) % self.L, (j + dj) % self.L
                total += self.photon_counts[ni, nj]
        return total

    def _apply_stimulated_emission(self):
        """
        Applies the stimulated emission rule. This must be done in a way that
        simulates a parallel update for all cells.
        """
        # A temporary array to store changes, as per the paper [cite: 419]
        new_photons_this_step = np.zeros((self.L, self.L), dtype=np.int32)

        for i in range(self.L):
            for j in range(self.L):
                # Condition 1: Electron is in the excited state
                if self.electron_states[i, j] == 1:
                    # Condition 2: Photon count in neighborhood is above threshold
                    neighbor_photons = self._get_moore_neighbors_sum(i, j)
                    if neighbor_photons >= self.stim_threshold:
                        # Find the first available photon "slot" in this cell
                        k = np.where(self.photon_lifetimes[i, j, :] == 0)[0]
                        if len(k) > 0:
                            slot = k[0]
                            # A new photon is emitted
                            self.photon_lifetimes[i, j, slot] = self.tau_c + 1 # +1 to account for decay in the
same step
                            new_photons_this_step[i, j] += 1
                            # The electron decays to the ground level
                            self.electron_states[i, j] = 0
                            self.electron_lifetimes[i, j] = 0

        # Update the main photon counts array
        self.photon_counts += new_photons_this_step

    def _apply_decays_and_pumping(self):
        """
```

```
        Applies the photon decay, electron decay, and pumping rules. [cite: 424]
        """
        # --- Photon Decay Rule --- [cite: 421]
        decaying_photons_mask = (self.photon_lifetimes > 0)
        self.photon_lifetimes[decaying_photons_mask] -= 1
        # Count how many photons just decayed in each cell
        newly_decayed_photons = np.sum((self.photon_lifetimes == 0) & decaying_photons_mask,
axis=2)
        self.photon_counts -= newly_decayed_photons

        # --- Electron Decay Rule --- [cite: 425]
        decaying_electrons_mask = (self.electron_lifetimes > 0)
        self.electron_lifetimes[decaying_electrons_mask] -= 1
        # If lifetime reaches zero, electron decays
        self.electron_states[(self.electron_lifetimes == 0) & decaying_electrons_mask] = 0

        # --- Pumping Rule ---
        ground_state_mask = (self.electron_states == 0)
        # Generate random numbers only for electrons in the ground state
        rand_values = np.random.rand(self.L, self.L)
        # Find which of those electrons get pumped
        pumped_mask = (rand_values < self.lambda_p) & ground_state_mask

        self.electron_states[pumped_mask] = 1
        self.electron_lifetimes[pumped_mask] = self.tau_a

    def _apply_noise(self):
        """
        Adds a small number of noise photons to random cells.
        """
        for _ in range(self.noise_photons):
            i, j = np.random.randint(0, self.L, size=2)
            # Find an empty photon slot
            k = np.where(self.photon_lifetimes[i, j, :] == 0)[0]
            if len(k) > 0 and self.photon_counts[i, j] < self.M:
                slot = k[0]
                self.photon_lifetimes[i, j, slot] = self.tau_c
                self.photon_counts[i, j] += 1

    def run_simulation(self, max_time_steps):
        """
        Runs the main simulation loop.

        Args:
            max_time_steps (int): The number of time steps to simulate.

        Returns:
            tuple: A tuple containing lists of total photons and population inversion over time.
        """
```

```
        total_photons_history = []
        population_inversion_history = []

        for t in range(max_time_steps):
            # Apply rules in the order specified by Algorithm 19.1 [cite: 398]
            self._apply_stimulated_emission()
            self._apply_decays_and_pumping()
            self._apply_noise()

            # Calculate macroscopic magnitudes [cite: 482, 483, 484]
            n_t = np.sum(self.photon_counts)
            N_t = np.sum(self.electron_states)

            total_photons_history.append(n_t)
            population_inversion_history.append(N_t)

            if (t + 1) % 100 == 0:
                print(f"Time Step: {t+1}/{max_time_steps}, Photons: {n_t}, Population Inversion: {N_t}")

        return total_photons_history, population_inversion_history

if __name__ == '__main__':
    # --- Simulation Parameters ---
    # These parameters are chosen to demonstrate oscillatory behavior, similar to Fig 19.4 (right)
    LATTICE_SIZE = 50       # L x L grid
    SIMULATION_STEPS = 1000
    TAU_C = 8               # Photon lifetime
    TAU_A = 100             # Electron lifetime
    LAMBDA_P = 0.05         # Pumping probability
    MAX_PHOTONS_PER_CELL = 20
    NOISE_PHOTONS = 5

    # --- Initialize and Run Simulation ---
    laser_sim = LaserCA(
        L=LATTICE_SIZE,
        tau_c=TAU_C,
        tau_a=TAU_A,
        lambda_p=LAMBDA_P,
        M=MAX_PHOTONS_PER_CELL,
        noise_photons=NOISE_PHOTONS
    )

    print("Starting Laser CA Simulation...")
    photons, inversion = laser_sim.run_simulation(SIMULATION_STEPS)
    print("Simulation finished.")

    # --- Plotting Results ---
    time_steps = np.arange(SIMULATION_STEPS)
```

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10), sharex=True)

# Plot 1: Time evolution of photons and population inversion
ax1.plot(time_steps, inversion, label='Population Inversion ($N(t)$)', color='gray')
ax1.plot(time_steps, photons, label='Laser Photons ($n(t)$)', color='black', linewidth=2)
ax1.set_ylabel('Population')
ax1.set_title('Laser Dynamics Simulation via Cellular Automata')
ax1.legend()
ax1.grid(True, linestyle='--', alpha=0.6)

# Plot 2: Phase space plot (Population Inversion vs. Laser Photons)
ax2.plot(inversion, photons, color='black', marker='.', linestyle='', markersize=2)
ax2.set_xlabel('Population Inversion ($N(t)$)')
ax2.set_ylabel('Laser Photons ($n(t)$)')
ax2.set_title('Phase Space Plot')
ax2.grid(True, linestyle='--', alpha=0.6)

plt.xlabel('Time Steps')
plt.tight_layout()
plt.show()
```