VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Raghav Kaushal(1BM23CS257)

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 **Raghav Kaushal (1BM23CS257),** 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.

Prof. Swathi S. Assistant Professor Department of CSE, BMSCE https://github.com/ChiraiyaSethiya/BIS_LAB	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

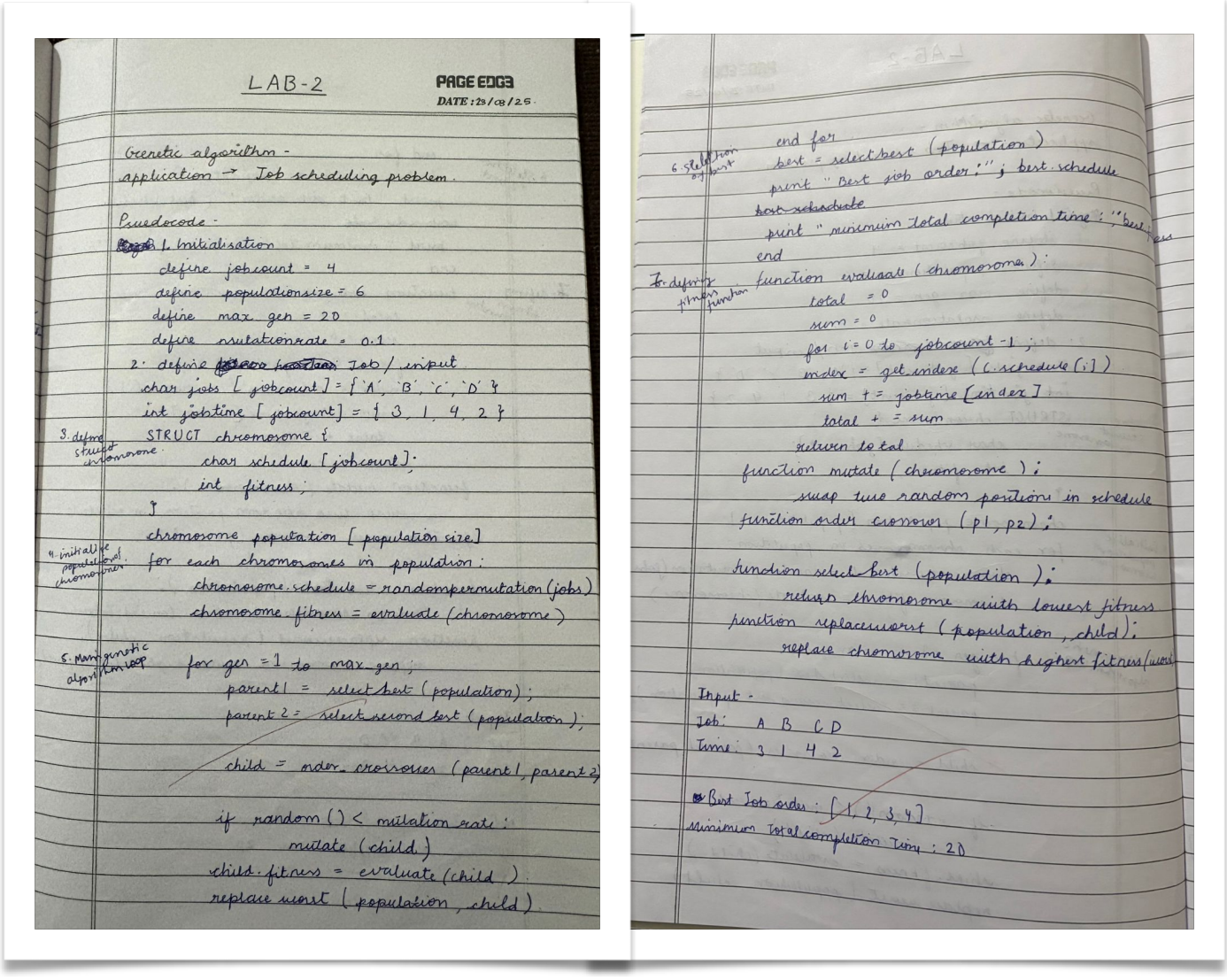
Github Link:
https://github.com/raghavkaushal123/BIS

Program 1

Genetic Algorithm for Optimization Problems:

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

Algorithm:



LAB-2

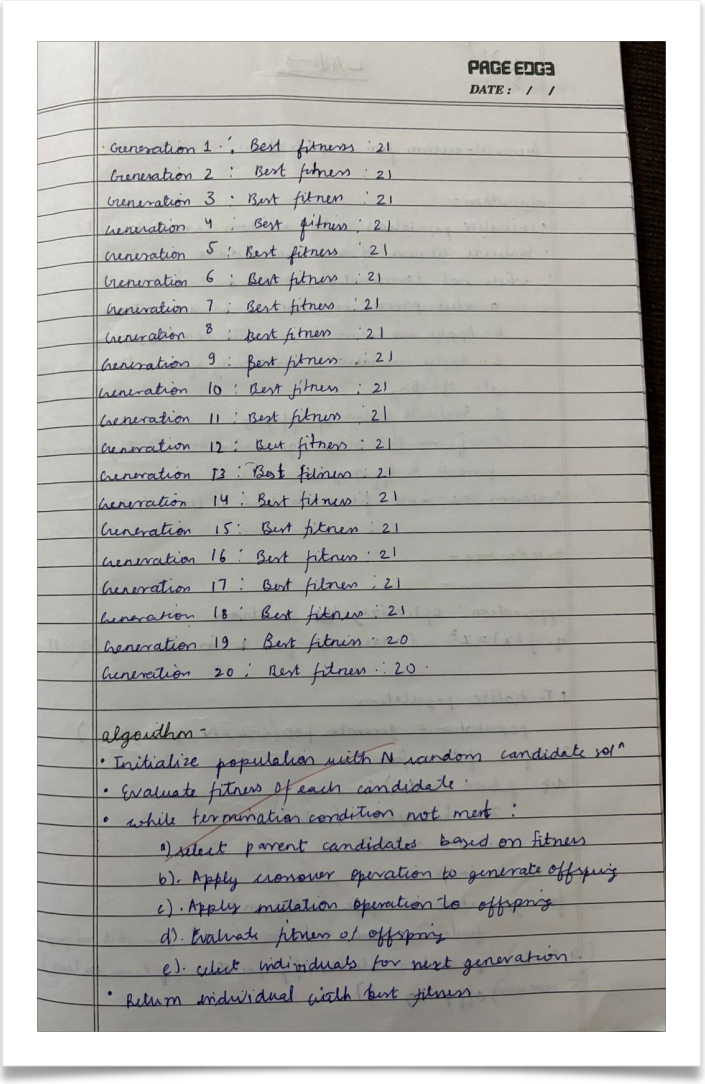
PAGE EDGE
DATE : 25/08/25

Genetic algorithm -
application → Job scheduling problem.

Pseudocode -
1. Initialisation
define jobcount = 4
define populationsize = 6
define max_gen = 20
define mutationrate = 0.1
2. define Job / input
char jobs [jobcount] = {'A', 'B', 'C', 'D'};
int jobtime [jobcount] = {3, 1, 4, 2};
3. define structure chromosome
STRUCT chromosome {
char schedule [jobcount];
int fitness;
}
chromosome population [population size]
4. initialize population of chromosomes
for each chromosomes in population:
chromosome.schedule = randompermutation(jobs)
chromosome fitness = evaluate (chromosome)
5. Main genetic algorithm loop
for gen = 1 to max_gen;
parent1 = select best (population);
parent2 = select second best (population);
child = order_crossover (parent1, parent2)
if random() < mutation rate:
mutate (child)
child.fitness = evaluate (child)
replace worst (population, child)
6. Selection of best
end for
best = select best (population)
print "Best job order:"; best.schedule
best schedule
print "minimum total completion time:"; best.fitness
end
7. define fitness function
function evaluate (chromosome):
total = 0
sum = 0
for i = 0 to jobcount - 1;
index = get index (c.schedule[i])
sum += jobtime [index]
total += sum
return total
function mutate (chromosome);
swap two random positions in schedule
function order crossover (p1, p2);
function select best (population);
return chromosome with lowest fitness
function replaceworst (population, child);
replace chromosome with highest fitness(worst)

Input -
Job: A B C D
Time: 3 1 4 2

Best Job order: [1, 2, 3, 4]
minimum total completion Time: 20



PAGE EDGE
DATE : / /

Generation 1 : Best fitness : 21
Generation 2 : Best fitness : 21
Generation 3 : Best fitness : 21
Generation 4 : Best fitness : 21
Generation 5 : Best fitness : 21
Generation 6 : Best fitness : 21
Generation 7 : Best fitness : 21
Generation 8 : Best fitness : 21
Generation 9 : Best fitness : 21
Generation 10 : Best fitness : 21
Generation 11 : Best fitness : 21
Generation 12 : Best fitness : 21
Generation 13 : Best fitness : 21
Generation 14 : Best fitness : 21
Generation 15 : Best fitness : 21
Generation 16 : Best fitness : 21
Generation 17 : Best fitness : 21
Generation 18 : Best fitness : 21
Generation 19 : Best fitness : 20
Generation 20 : Best fitness : 20

algorithm -
- Initialize population with N random candidate sol^n
- Evaluate fitness of each candidate.
- while termination condition not met :
  a) select parent candidates based on fitness
  b). Apply crossover operation to generate offspring
  c). Apply mutation operation to offspring
  d). Evaluate fitness of offspring
  e). select individuals for next generation.
- Return individual with best fitness

Code:

```
import random

job_times = [3, 1, 4, 2,]
POP_SIZE = 6
GENS = 20
MUTATION_RATE = 0.1
TOURNAMENT_SIZE = 5

def create_population(jobs, size):
    return [random.sample(jobs, len(jobs)) for _ in range(size)]
```

```
def fitness(job_order):
    time = 0
    total = 0
    for job in job_order:
        time += job
        total += time
    return total

def select(population):
    tournament = random.sample(population, TOURNAMENT_SIZE)
    tournament.sort(key=fitness)
    return tournament[0]

def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[start:end] = parent1[start:end]
    fill_values = [item for item in parent2 if item not in child]
    idx = 0
    for i in range(size):
        if child[i] is None:
            child[i] = fill_values[idx]
            idx += 1
    return child

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

def genetic_algorithm(job_times):
    population = create_population(job_times, POP_SIZE)
    best = min(population, key=fitness)
    for gen in range(GENS):
        new_population = []
        for _ in range(POP_SIZE):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)
        population = new_population
        current_best = min(population, key=fitness)
        if fitness(current_best) < fitness(best):
            best = current_best
```

```
        print(f"Generation {gen+1}: Best Fitness = {fitness(best)}")
    return best
best_order = genetic_algorithm(job_times)
print("\nBest Job Order:", best_order)
print("Minimum Total Completion Time:", fitness(best_order))
```

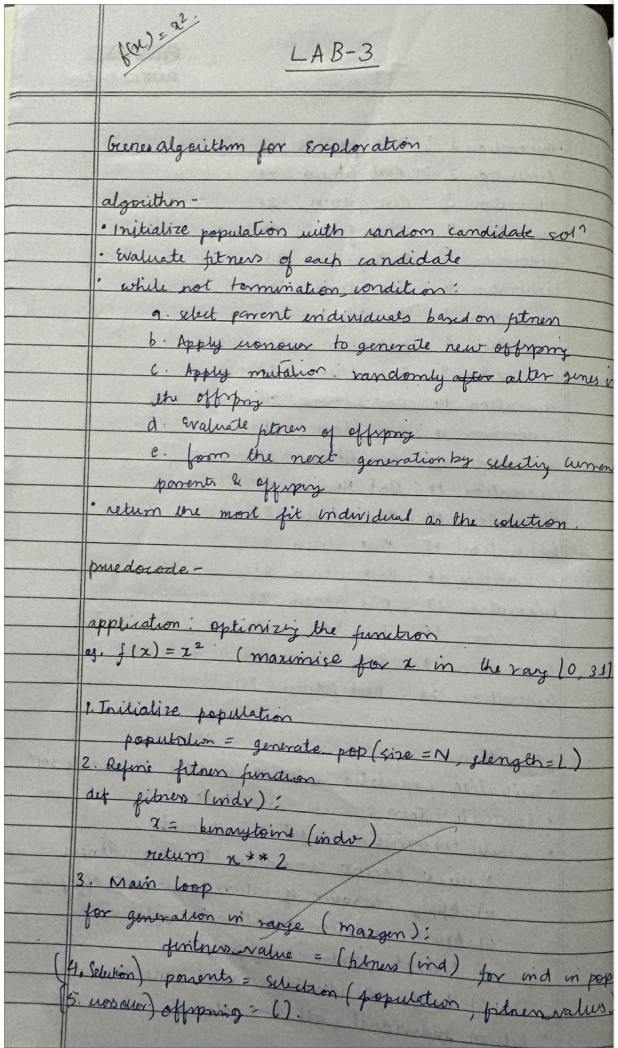
```
Generation 1: Best Fitness = 21
Generation 2: Best Fitness = 21
Generation 3: Best Fitness = 21
Generation 4: Best Fitness = 21
Generation 5: Best Fitness = 21
Generation 6: Best Fitness = 21
Generation 7: Best Fitness = 21
Generation 8: Best Fitness = 21
Generation 9: Best Fitness = 21
Generation 10: Best Fitness = 21
Generation 11: Best Fitness = 21
Generation 12: Best Fitness = 21
Generation 13: Best Fitness = 21
Generation 14: Best Fitness = 21
Generation 15: Best Fitness = 21
Generation 16: Best Fitness = 21
Generation 17: Best Fitness = 21
Generation 18: Best Fitness = 21
Generation 19: Best Fitness = 20
Generation 20: Best Fitness = 20

Best Job Order: [1, 2, 3, 4]
Minimum Total Completion Time: 20
```

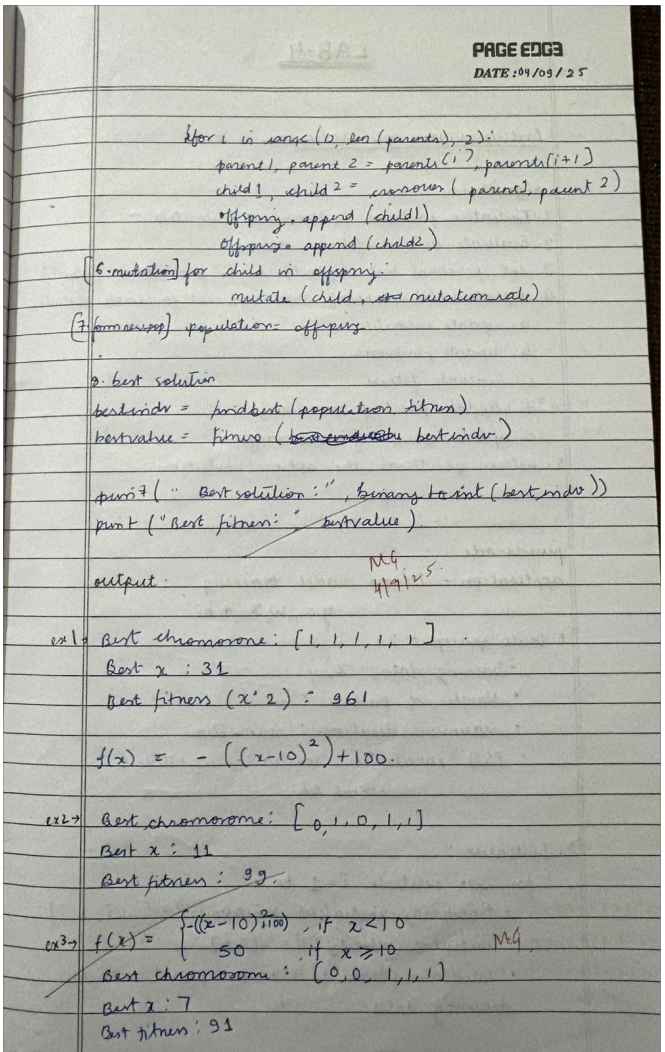
Program 2

Optimization via Gene Expression algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:





Code:

```
import random
CHROMOSOME_LENGTH = 5
POPULATION_SIZE = 6
MAX_GEN = 20
MUTATION_RATE = 0.1
def decode(chromosome):
    return int("".join(str(bit) for bit in chromosome), 2)
def fitness(chromosome):
    x = decode(chromosome)
    return x ** 2
def random_chromosome():
    return [random.randint(0, 1) for _ in range(CHROMOSOME_LENGTH)]
def selection(population):
    fitness_values = [fitness(ind) for ind in population]
    total_fit = sum(fitness_values)
    pick = random.uniform(0, total_fit)
    current = 0
    for ind, fit in zip(population, fitness_values):
        current += fit
        if current > pick:
            return ind
    return population[-1]  # This line should be unreachable if the logic is correct
def crossover(p1, p2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = p1[:point] + p2[point:]
    child2 = p2[:point] + p1[point:]
    return child1, child2

def mutate(chromosome):
    for i in range(CHROMOSOME_LENGTH):
```

```
        if random.random() < MUTATION_RATE:
            chromosome[i] = 1 - chromosome[i]
    return chromosome
def gene_algorithm():
    population = [random_chromosome() for _ in range(POPULATION_SIZE)]
    for gen in range(MAX_GEN):
        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            child1, child2 = crossover(parent1, parent2)
            new_population.append(mutate(child1))
            new_population.append(mutate(child2))
        population = new_population
    best = max(population, key=fitness)
    best_x = decode(best)
    print("Best Chromosome:", best)
    print("Best x:", best_x)
    print("Best Fitness (x^2):", fitness(best))
gene_algorithm()
```

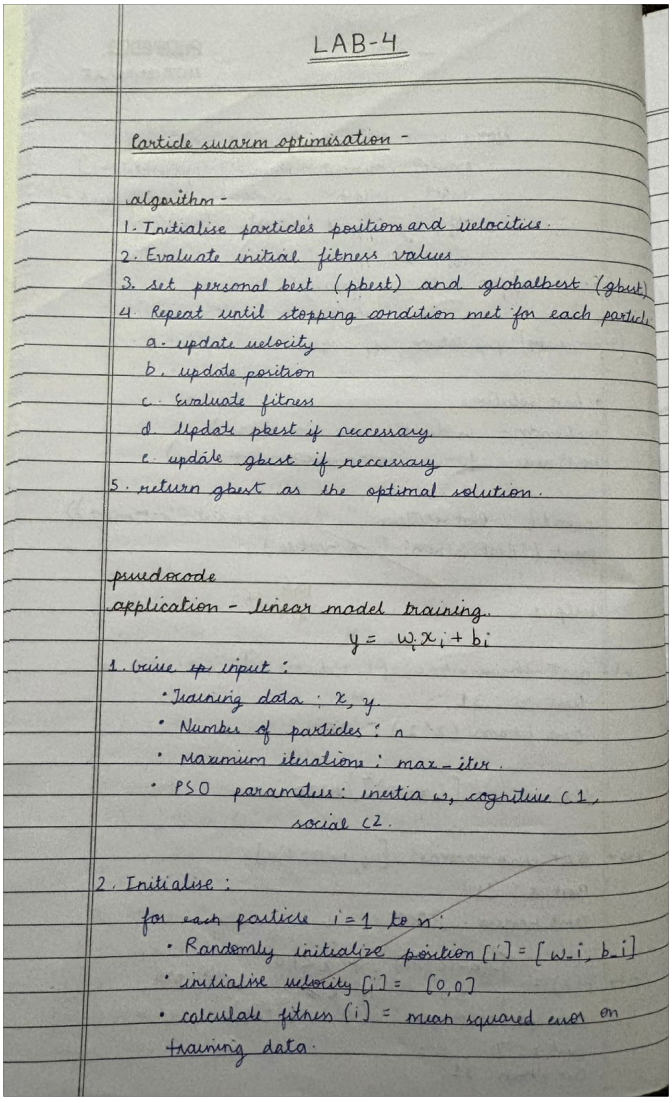
```
Best Chromosome: [1, 1, 1, 1, 1]
Best x: 31
Best Fitness (x^2): 961
```

Program 3

Particle swarm optimisation

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

Algorithm:



LAB-4

Particle swarm optimisation -

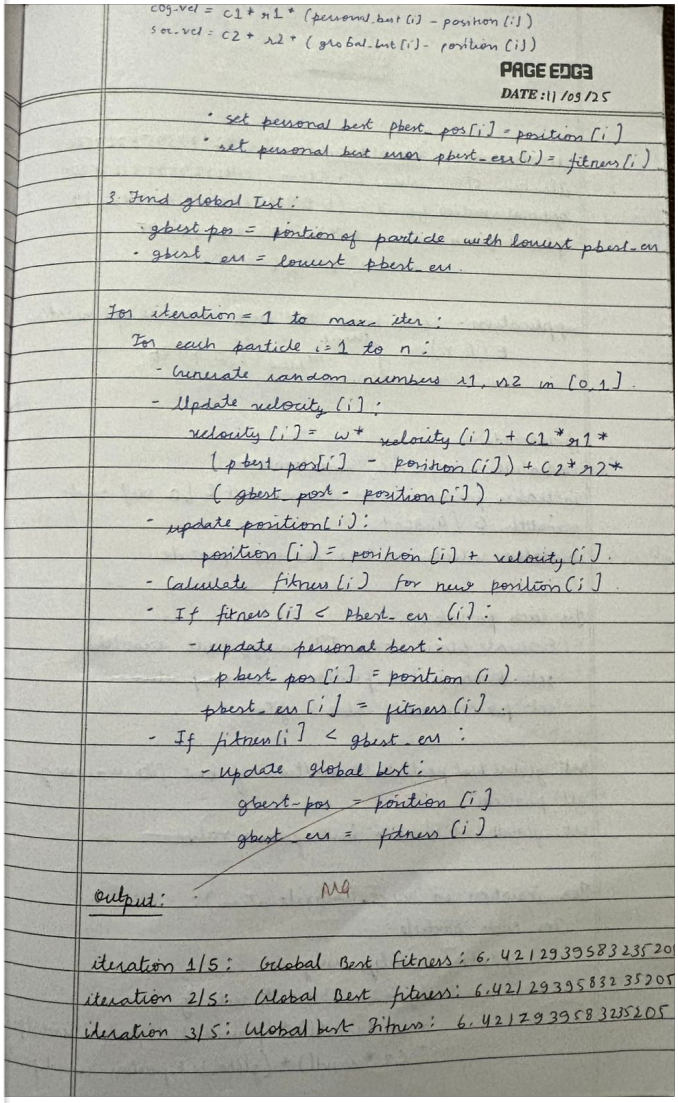
algorithm -
1. Initialise particles positions and velocities.
2. Evaluate initial fitness values
3. set personal best (pbest) and globalbest (gbest)
4. Repeat until stopping condition met for each particle
   a. update velocity
   b. update position
   c. evaluate fitness
   d. update pbest if necessary.
   e. update gbest if neccessary
5. return gbest as the optimal solution.

pseudocode
application - linear model training.

$$y = w_i x_i + b_i$$

1. Initialise input :
   - Training data : x, y.
   - Number of particles : n
   - Maximum iterations : max_iter.
   - PSO parameters : inertia w, cognitive c1, social c2.

2. Initialise :
   for each particle i = 1 to n :
   - Randomly initialize position [i] = [w_i, b_i]
   - initialise velocity [i] = [0,0]
   - calculate fitness (i) = mean squared error on training data.



cog_vel = c1 * r1 * (personal_best [i] - position [i])
soc_vel = c2 * r2 * (global_best [i] - position (i))

PAGE EDGE
DATE : 11/09/25

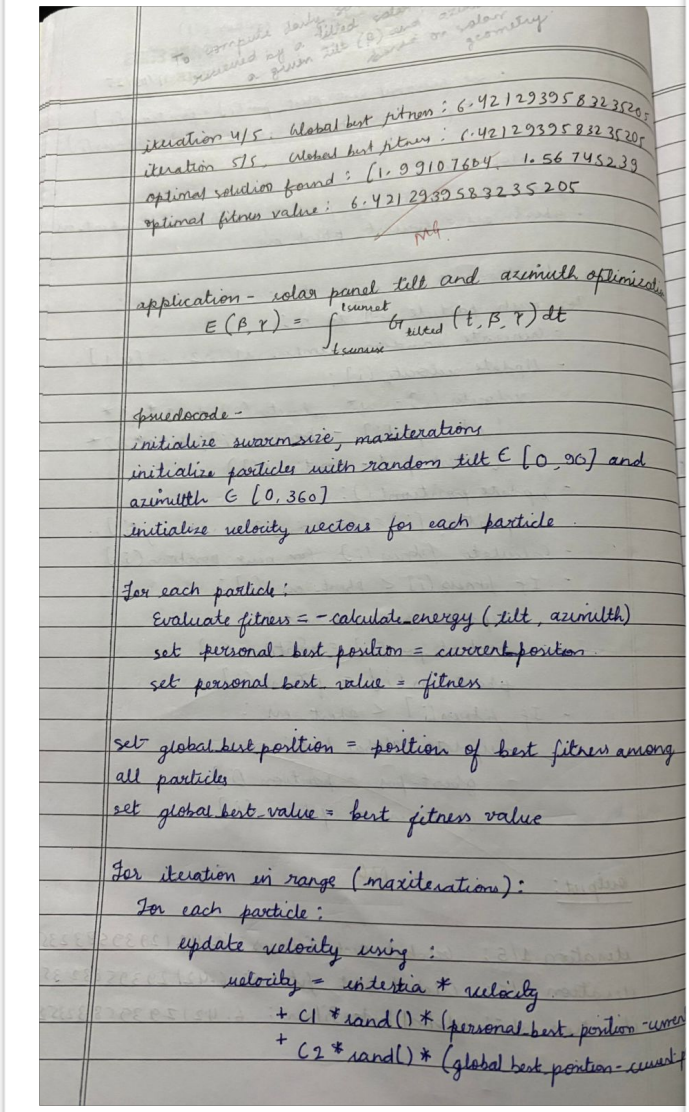
   - set personal best pbest_pos[i] = position [i]
   - set personal best error pbest_err [i] = fitness [i]

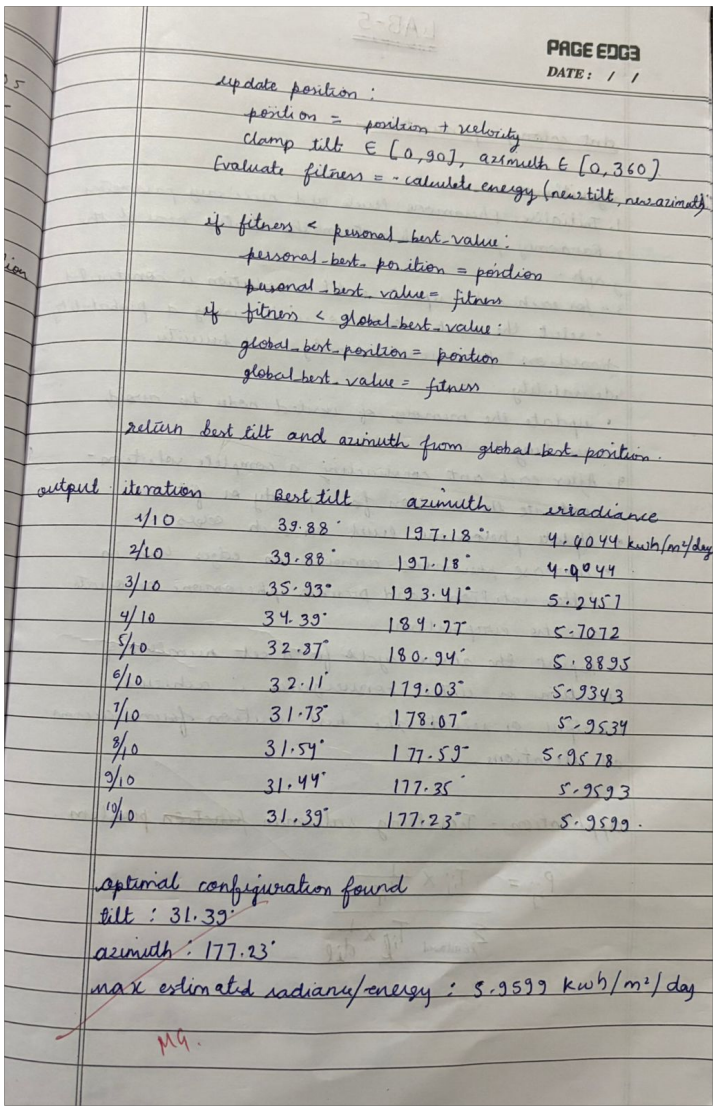
3. Find global best :
   - gbest_pos = position of particle with lowest pbest_err
   - gbest_err = lowest pbest_err.

For iteration = 1 to max_iter :
   For each particle i = 1 to n :
   - Generate random numbers r1, r2 in [0,1].
   - Update velocity (i) :
     velocity [i] = w * velocity (i) + c1 * r1 * (pbest_pos[i] - position (i)) + c2 * r2 * (gbest_pos - position [i])
   - update position (i) :
     position [i] = position [i] + velocity [i].
   - Calculate fitness (i) for new position (i)
   - If fitness (i) < Pbest_err (i) :
     - update personal best :
       pbest_pos [i] = position (i)
       pbest_err [i] = fitness (i)
   - If fitness [i] < gbest_err :
     - Update global best :
       gbest_pos = position [i]
       gbest_err = fitness (i)

output :

iteration 1/5 : Global Best fitness : 6.421293958323520
iteration 2/5 : Global Best fitness : 6.421293958323520
iteration 3/5 : Global best fitness : 6.421293958323520





Code:

```
import numpy as np

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

num_particles = 10
num_dimensions = 2
num_iterations = 5
w = 5
c1 = 20
```

```
c2 = 25

positions = np.random.uniform(1, 5, (num_particles, num_dimensions))
velocities = np.zeros((num_particles, num_dimensions))

personal_best_positions = positions.copy()
personal_best_fitness = np.array([fitness_function(p) for p in positions])

global_best_index = np.argmin(personal_best_fitness)
global_best_position = personal_best_positions[global_best_index]
global_best_fitness = personal_best_fitness[global_best_index]

for iteration in range(num_iterations):
    for i in range(num_particles):
        r1 = np.random.rand(num_dimensions)
        r2 = np.random.rand(num_dimensions)

        cognitive_velocity = c1 * r1 * (personal_best_positions[i] -
positions[i])
        social_velocity = c2 * r2 * (global_best_position - positions[i])
        velocities[i] = w * velocities[i] + cognitive_velocity +
social_velocity

        positions[i] += velocities[i]
        fitness = fitness_function(positions[i])

        if fitness < personal_best_fitness[i]:
            personal_best_fitness[i] = fitness
            personal_best_positions[i] = positions[i].copy()

        if fitness < global_best_fitness:
            global_best_fitness = fitness
            global_best_position = positions[i].copy()

    print(f"Iteration {iteration+1}/{num_iterations}, Global Best Fitness:
{global_best_fitness}")

print("Optimal solution found:", global_best_position)
print("Optimal fitness value:", global_best_fitness)
```

```
Iteration 1/5, Global Best Fitness: 6.4212939583235205
Iteration 2/5, Global Best Fitness: 6.4212939583235205
Iteration 3/5, Global Best Fitness: 6.4212939583235205
Iteration 4/5, Global Best Fitness: 6.4212939583235205
Iteration 5/5, Global Best Fitness: 6.4212939583235205
Optimal solution found: [1.99107684 1.56745239]
Optimal fitness value: 6.4212939583235205
```

Program 4

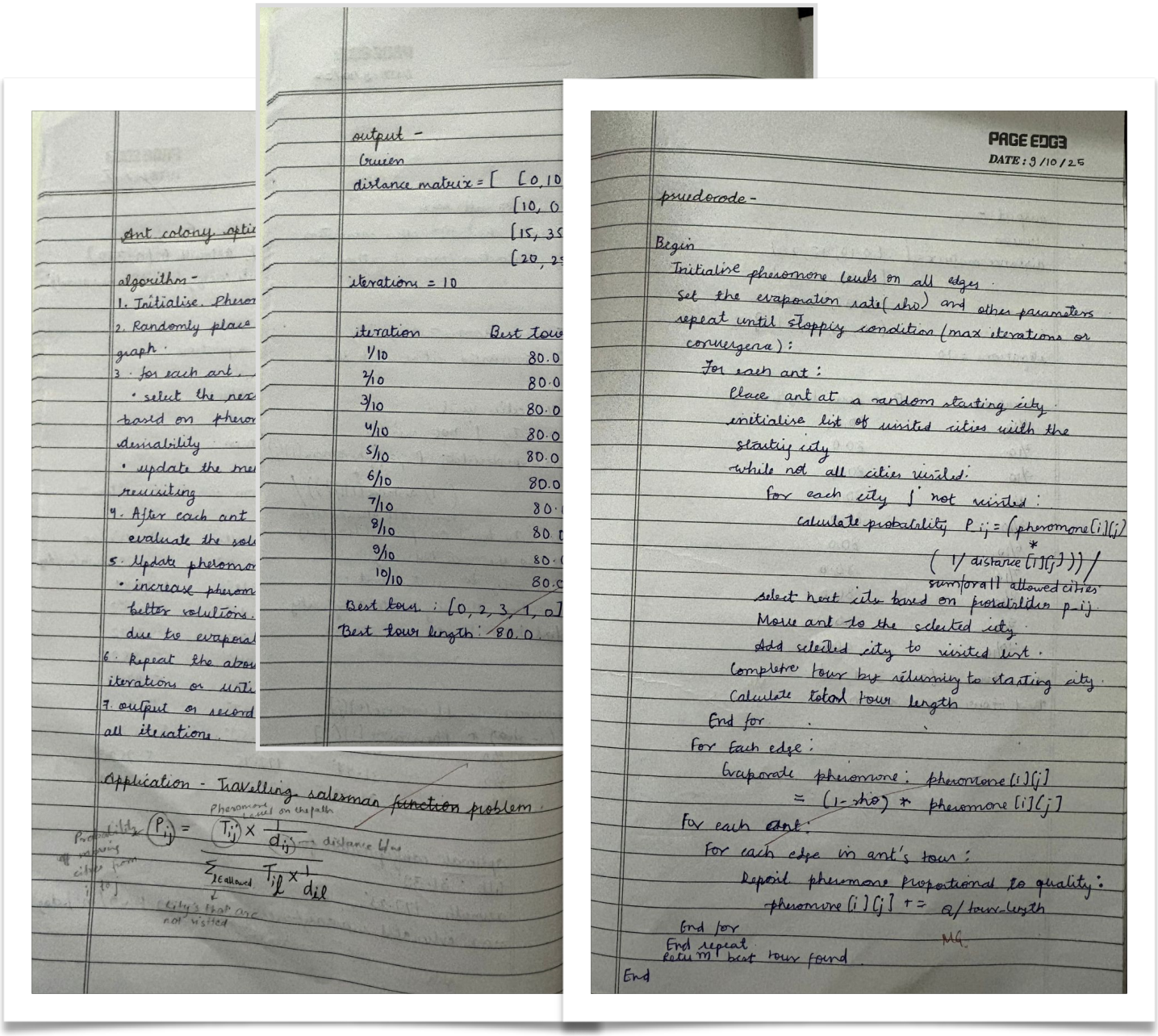
Ant Colony optimisation

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

Algorithm:

Code:

```
import random

def initialize_pheromone_matrix(num_cities, initial_pheromone):
    return [[initial_pheromone for _ in range(num_cities)] for _ in
range(num_cities)]

def calculate_probability(current_city, unvisited_cities, pheromone,
distance):
    pheromone_list = []
    heuristic_list = []
    for city in unvisited_cities:
        pheromone_list.append(pheromone[current_city][city])
        heuristic_list.append(1.0 / distance[current_city][city])
    total = sum([p * h for p, h in zip(pheromone_list, heuristic_list)])
```




```
    probabilities = [(pheromone[current_city][city] * (1.0 /
distance[current_city][city])) / total for city in unvisited_cities]
    return probabilities

def select_next_city(unvisited_cities, probabilities):
    return random.choices(unvisited_cities, weights=probabilities)[0]

def update_pheromone(pheromone, ants_tours, ants_lengths, evaporation_rate,
Q):
    num_cities = len(pheromone)
    for i in range(num_cities):
        for j in range(num_cities):
            pheromone[i][j] *= (1 - evaporation_rate)
            if pheromone[i][j] < 1e-10:
```

```
                pheromone[i][j] = 1e-10

    for k in range(len(ants_tours)):
        tour = ants_tours[k]
        length = ants_lengths[k]
        deposit = Q / length
        for i in range(len(tour)-1):
            a = tour[i]
            b = tour[i+1]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit

def calculate_tour_length(tour, distance):
    length = 0.0
    for i in range(len(tour)-1):
        length += distance[tour[i]][tour[i+1]]
    return length

def ant_colony_optimization(distance, num_ants=10, num_iterations=10,
evaporation_rate=0.5, Q=100, initial_pheromone=1.0):
    num_cities = len(distance)
    pheromone = initialize_pheromone_matrix(num_cities, initial_pheromone)
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        ants_tours = []
        ants_lengths = []

        for _ in range(num_ants):
            start_city = random.randint(0, num_cities - 1)
            tour = [start_city]
            unvisited_cities = set(range(num_cities))
            unvisited_cities.remove(start_city)
            current_city = start_city

            while unvisited_cities:
                probabilities = calculate_probability(current_city,
list(unvisited_cities), pheromone, distance)
                next_city = select_next_city(list(unvisited_cities),
probabilities)
                tour.append(next_city)
                unvisited_cities.remove(next_city)
                current_city = next_city

            tour.append(start_city)
            tour_length = calculate_tour_length(tour, distance)
```

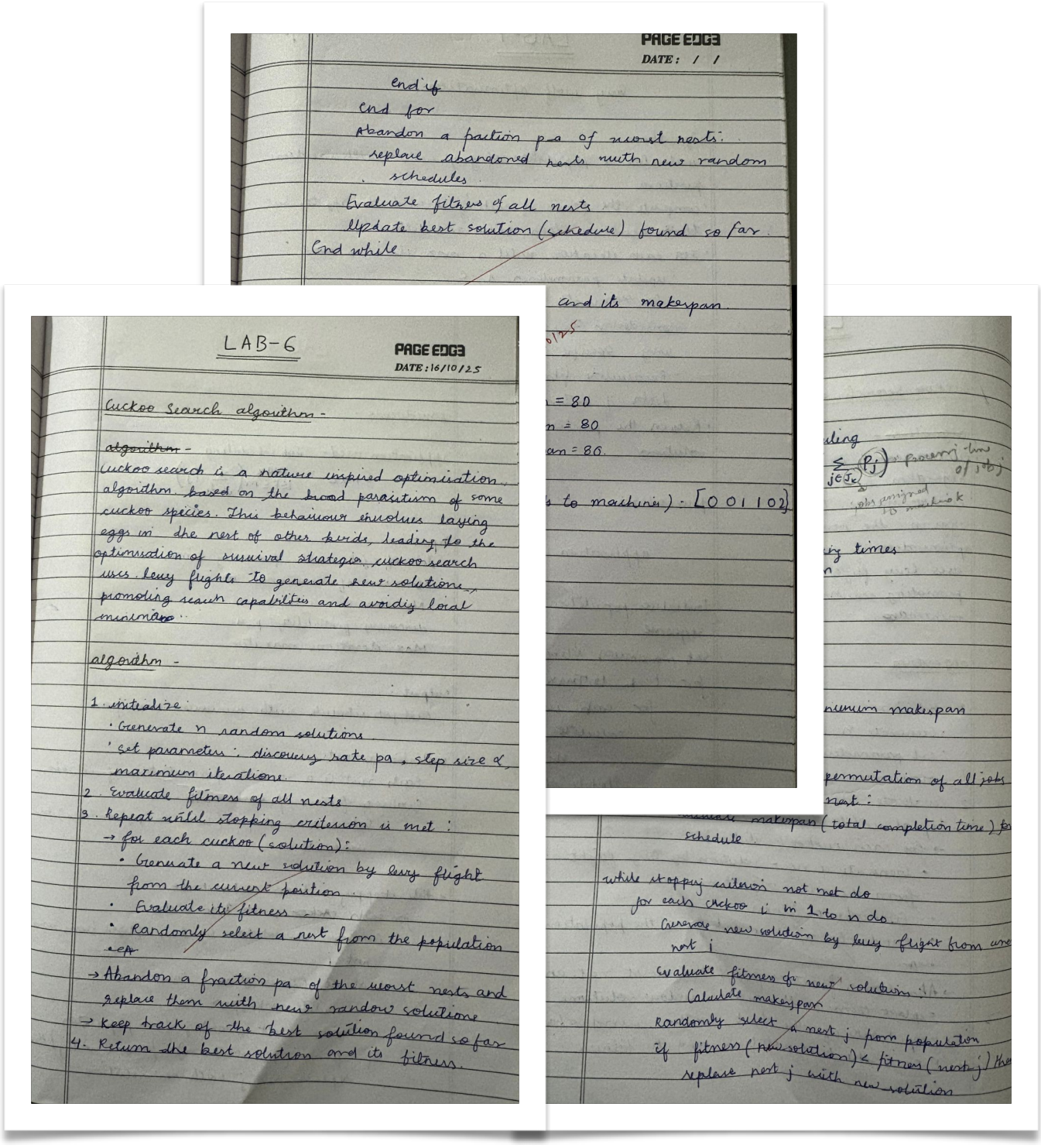
```
            ants_tours.append(tour)
            ants_lengths.append(tour_length)

            if tour_length < best_length:
                best_length = tour_length
                best_tour = tour

        update_pheromone(pheromone, ants_tours, ants_lengths,
evaporation_rate, Q)
        print(f"Iteration {iteration+1}/{num_iterations}, Best tour length:
{best_length}")

    return best_tour, best_length

distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

best_tour, best_length = ant_colony_optimization(distance_matrix)
print("Best tour:", best_tour)
print("Best tour length:", best_length)
```

```
Iteration 1/10, Best tour length: 80.0
Iteration 2/10, Best tour length: 80.0
Iteration 3/10, Best tour length: 80.0
Iteration 4/10, Best tour length: 80.0
Iteration 5/10, Best tour length: 80.0
Iteration 6/10, Best tour length: 80.0
Iteration 7/10, Best tour length: 80.0
Iteration 8/10, Best tour length: 80.0
Iteration 9/10, Best tour length: 80.0
Iteration 10/10, Best tour length: 80.0
Best tour: [0, 2, 3, 1, 0]
Best tour length: 80.0
```

Program 5

Cuckoo Search

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

class CuckooSearch:
    def __init__(self, jobs, machines, max_iter=3, population_size=30,
pa=0.25, alpha=0.01, beta=1.5):
        self.jobs = jobs
        self.machines = machines
        self.max_iter = max_iter
        self.population_size = population_size
        self.pa = pa
        self.alpha = alpha
        self.beta = beta
        self.num_jobs = len(jobs)

        self.population = np.random.randint(0, machines, (population_size,
self.num_jobs))
        self.fitness = np.array([self.calculate_makespan(ind) for ind in
self.population])

        self.best_solution = self.population[np.argmin(self.fitness)]
        self.best_fitness = np.min(self.fitness)

    def calculate_makespan(self, solution):
        machine_times = [0] * self.machines
        for job, machine in enumerate(solution):
            machine_times[machine] += self.jobs[job]
        makespan = max(machine_times)
        return makespan

    def levy_flight(self):
        return np.random.normal(0, 1, self.num_jobs)

    def generate_new_solution(self, solution):
        new_solution = solution + self.alpha * self.levy_flight()
        new_solution = np.clip(new_solution, 0, self.machines - 1)
        new_solution = np.round(new_solution).astype(int)
        return new_solution

    def run(self):
        for iter_num in range(self.max_iter):
            for i in range(self.population_size):
                new_solution = self.generate_new_solution(self.population[i])
                new_fitness = self.calculate_makespan(new_solution)

                if new_fitness < self.fitness[i]:
```

```
                    self.population[i] = new_solution
                    self.fitness[i] = new_fitness

            best_solution_index = np.argmin(self.fitness)
            best_solution = self.population[best_solution_index]
            best_fitness = self.fitness[best_solution_index]

            if best_fitness < self.best_fitness:
                self.best_solution = best_solution
                self.best_fitness = best_fitness

            for i in range(self.population_size):
                if np.random.rand() < self.pa:
                    self.population[i] = self.best_solution
                    self.fitness[i] = self.best_fitness

            print(f"Iteration {iter_num + 1}/{self.max_iter}: Best Makespan =
{self.best_fitness}")

        return self.best_solution, self.best_fitness


jobs = [10, 20, 30, 40, 50, 60]
machines = 3

cuckoo = CuckooSearch(jobs, machines, max_iter=3)
best_solution, best_makespan = cuckoo.run()

print("\nBest Solution (Job assignments to machines):", best_solution)
print("Best Makespan:", best_makespan)
```

```
Iteration 1/3: Best Makespan = 80
Iteration 2/3: Best Makespan = 80
Iteration 3/3: Best Makespan = 80

Best Solution (Job assignments to machines): [0 0 1 1 0 2]
Best Makespan: 80
```

Program 6

Grey Wolf Optimizer (GWO)

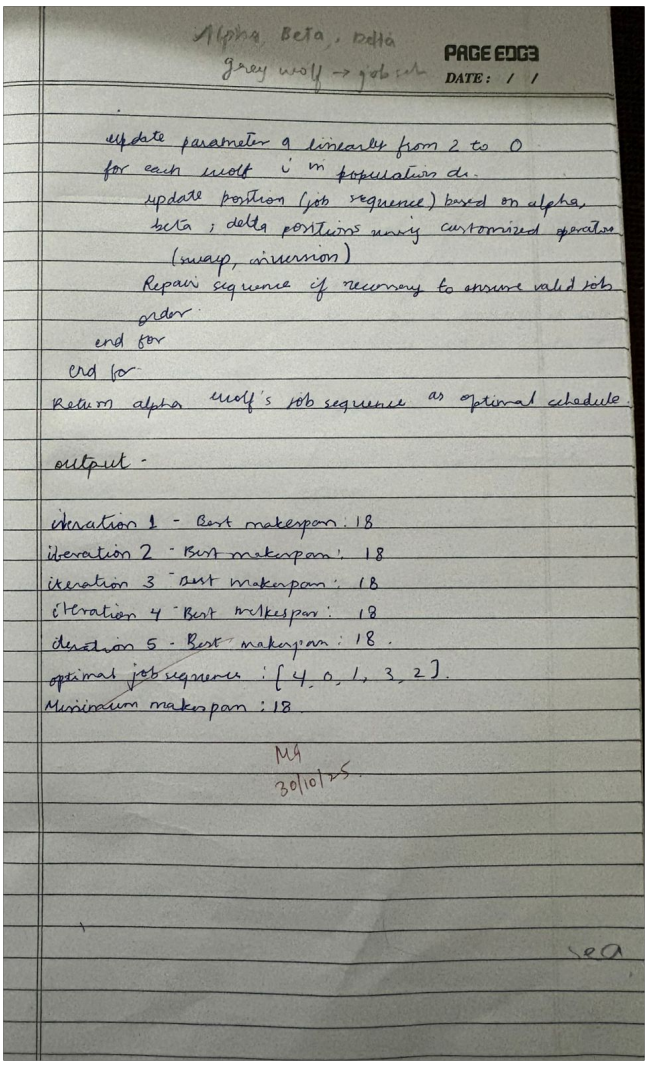
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies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

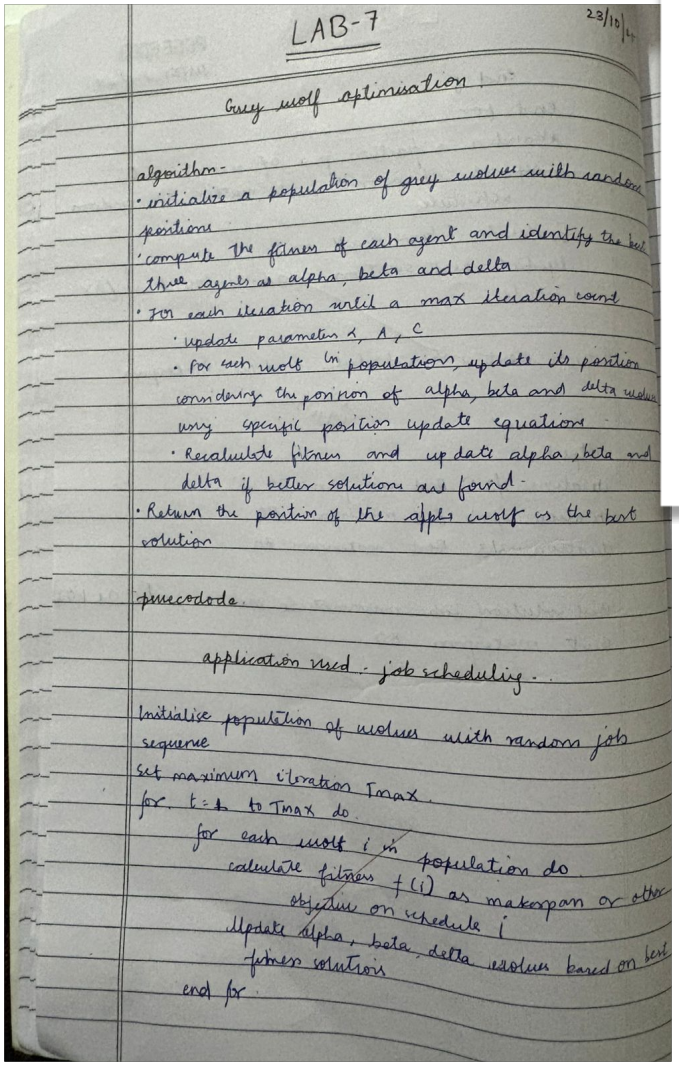
Algorithm:

Code:

```
Import bumpy as np
import random
def calculate_makespan(sequence, processing_times):
    time = 0
```

```
    for job in sequence:
        time +=
processing_times[job]
    return time

def swap_positions(sequence, idx1,
idx2):
    sequence = sequence.copy()
    sequence[idx1], sequence[idx2] =
sequence[idx2], sequence[idx1]
    return sequence

def update_position(current, alpha,
beta, delta, a):
```






```
    num_swaps = max(1, int(a *
len(current)))
    new_seq = current.copy()

    for _ in range(num_swaps):
        idx1 = random.randint(0,
len(current) - 1)
        idx2 =
alpha.index(new_seq[idx1])
        new_seq =
swap_positions(new_seq, idx1,
idx2)
```

```
    for _ in range(num_swaps):
        idx1 = random.randint(0, len(current) - 1)
        idx2 = beta.index(new_seq[idx1])
        new_seq = swap_positions(new_seq, idx1, idx2)

    for _ in range(num_swaps):
        idx1 = random.randint(0, len(current) - 1)
        idx2 = delta.index(new_seq[idx1])
        new_seq = swap_positions(new_seq, idx1, idx2)

    return new_seq

def gwo_job_scheduling(processing_times, population_size=10, max_iter=10):
    num_jobs = len(processing_times)

    population = [random.sample(range(num_jobs), num_jobs) for _ in
range(population_size)]

    alpha, beta, delta = None, None, None
    alpha_score, beta_score, delta_score = float('inf'), float('inf'),
float('inf')

    for iter in range(max_iter):
        fitness = []
        for wolf in population:
            makespan = calculate_makespan(wolf, processing_times)
            fitness.append(makespan)



            if makespan < alpha_score:
                alpha, alpha_score = wolf, makespan
            elif makespan < beta_score:
                beta, beta_score = wolf, makespan
```

```
            elif makespan < delta_score:
                delta, delta_score = wolf, makespan

        a = 2 - iter * (2 / max_iter) # Linearly decrease a from 2 to 0
        new_population = []
        for wolf in population:
            new_seq = update_position(wolf, alpha, beta, delta, a)
            new_population.append(new_seq)

        population = new_population

        print(f"Iteration {iter+1} - Best Makespan: {alpha_score}")

    return alpha, alpha_score
processing_times = [2, 3, 7, 5, 1] # Processing times of 5 jobs
best_sequence, best_makespan = gwo_job_scheduling(processing_times)
print("Optimal job sequence:", best_sequence)
print("Minimum makespan:", best_makespan)
```

```
Iteration 1 - Best Makespan: 18
Iteration 2 - Best Makespan: 18
Iteration 3 - Best Makespan: 18
Iteration 4 - Best Makespan: 18
Iteration 5 - Best Makespan: 18
Iteration 6 - Best Makespan: 18
Iteration 7 - Best Makespan: 18
Iteration 8 - Best Makespan: 18
Iteration 9 - Best Makespan: 18
Iteration 10 - Best Makespan: 18
Optimal job sequence: [2, 1, 0, 3, 4]
Minimum makespan: 18
```

Program 7

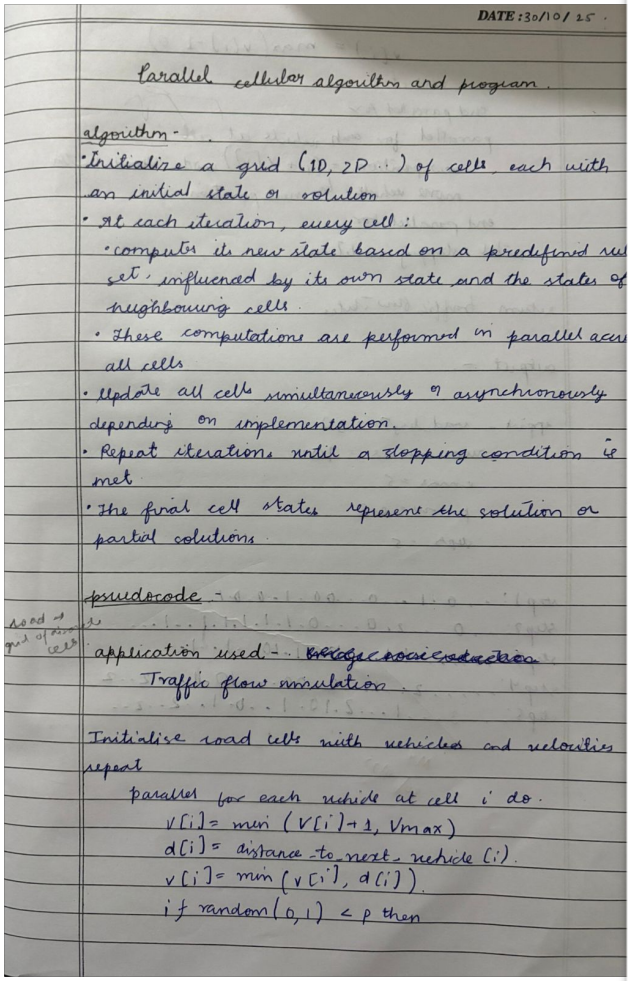
Parallel Cellular Algorithms and Programs

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

Algorithm:

Code:

```
import numpy as np


def traffic_flow_simulation(road_length, num_vehicles, V_max, p_slow, steps):
    road = -1 * np.ones(road_length, dtype=int)
    vehicle_positions = np.random.choice(range(road_length), num_vehicles,
replace=False)
    for pos in vehicle_positions:
```



DATE : 30/10/25

Parallel cellular algorithm and program.

algorithm -

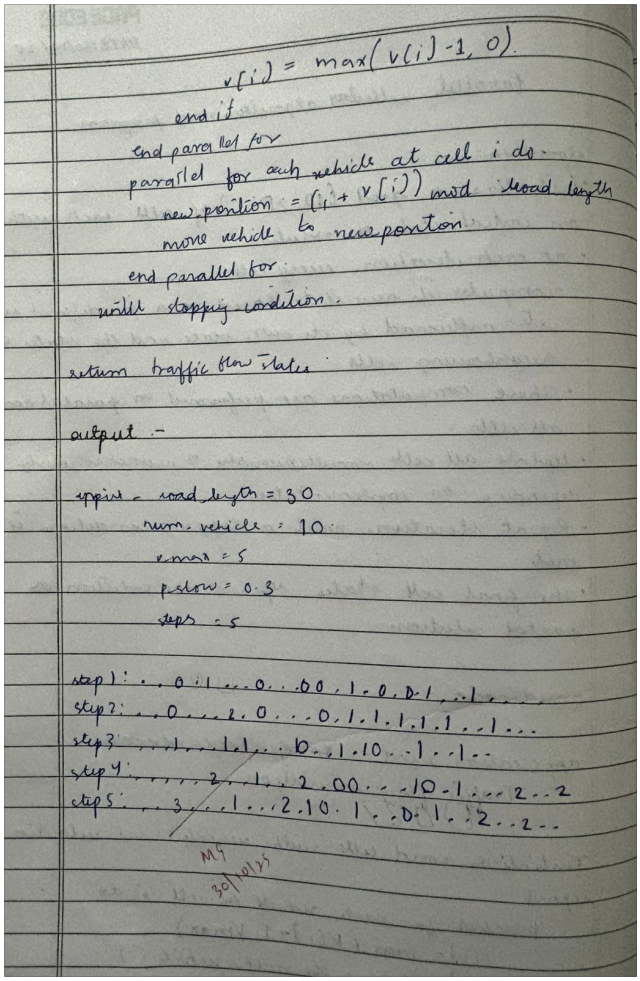
- Initialize a grid (1D, 2D ...) of cells each with an initial state or solution
- At each iteration, every cell :
  - computes its new state based on a predefined rule set, influenced by its own state and the states of neighbouring cells.
  - These computations are performed in parallel across all cells
  - Update all cells simultaneously or asynchronously depending on implementation.
  - Repeat iterations until a stopping condition is met
  - The final cell states represent the solution or partial solutions

pseudocode -

road = grid of cells

application used - ~~Bridge construction~~
Traffic flow simulation.

Initialise road cells with vehicles and velocities
repeat
parallel for each vehicle at cell i do.
v[i] = min (v[i]+1, Vmax)
d[i] = distance_to_next_vehicle(i).
v[i] = min (v[i], d[i])
if random(0,1) < p then



v[i] = max(v[i]-1, 0).
end if
end parallel for
parallel for each vehicle at cell i do.
new_position = (i + v[i]) mod road length
move vehicle to new_position.
end parallel for
until stopping condition.

return traffic flow states

output :-

input - road_length = 30
num. vehicle = 10
vmax = 5
pslow = 0.3
steps = 5

step 1: . . 0 . 1 . . . 0 . . 0 0 . 1 . 0 . 0 . 1 . . 1 . . .
step 2: . . 0 . . . 2 . 0 . . . 0 . 1 . 1 . 1 . 1 . 1 . . 1 . . .
step 3 . . . 1 . . . 1 . 1 . . . 0 . . 1 . 1 0 . . 1 . . 1 . .
step 4 . . . . . 2 . . 1 . . . 2 . 0 0 . . . 1 0 . 1 . . . 2 . . 2
step 5 . . . 3 . . . 1 . . . 2 . 1 0 . 1 . . 0 . 1 . . 2 . . 2 . .

M4
30/10/25

```
        road[pos] = 0


    def distance_to_next_vehicle(pos):
        distance = 1
        while road[(pos + distance) % road_length] == -1 and distance < road_length:
            distance += 1
        return distance


    for step in range(steps):
        new_road = -1 * np.ones(road_length, dtype=int)
        velocities = np.copy(road)
```

```
        for i in range(road_length):
            if road[i] != -1:
                v = road[i]
                v = min(v + 1, V_max)
                d = distance_to_next_vehicle(i) - 1
                v = min(v, d)
                if np.random.rand() < p_slow:
                    v = max(v - 1, 0)
                velocities[i] = v

        for i in range(road_length):
            if road[i] != -1:
                v = velocities[i]
                new_pos = (i + v) % road_length
                new_road[new_pos] = v

        road = new_road
        road_state = ''.join(['.' if x == -1 else str(x) for x in road])
        print(f"Step {step + 1}: {road_state}")

road_length = 30
num_vehicles = 10
V_max = 5
p_slow = 0.3
steps = 5

traffic_flow_simulation(road_length, num_vehicles, V_max, p_slow, steps)
```

```
Step 1: ..0.1...0...00.1.0.0.1...1....
Step 2: ..0...2.0...0.1.1.1.1.1...1...
Step 3: ...1...1.1...10..1.10..1...1..
Step 4: .....2..1..2.00:..10.1...2...2
Step 5: ..3...1...2.10.1..0.1..2...2..
```