VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Rudraksh Singh (1BM23CS276)

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-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 **Rudraksh Singh (1BM23CS276),** who is a 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.

Swathi Sridharan Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

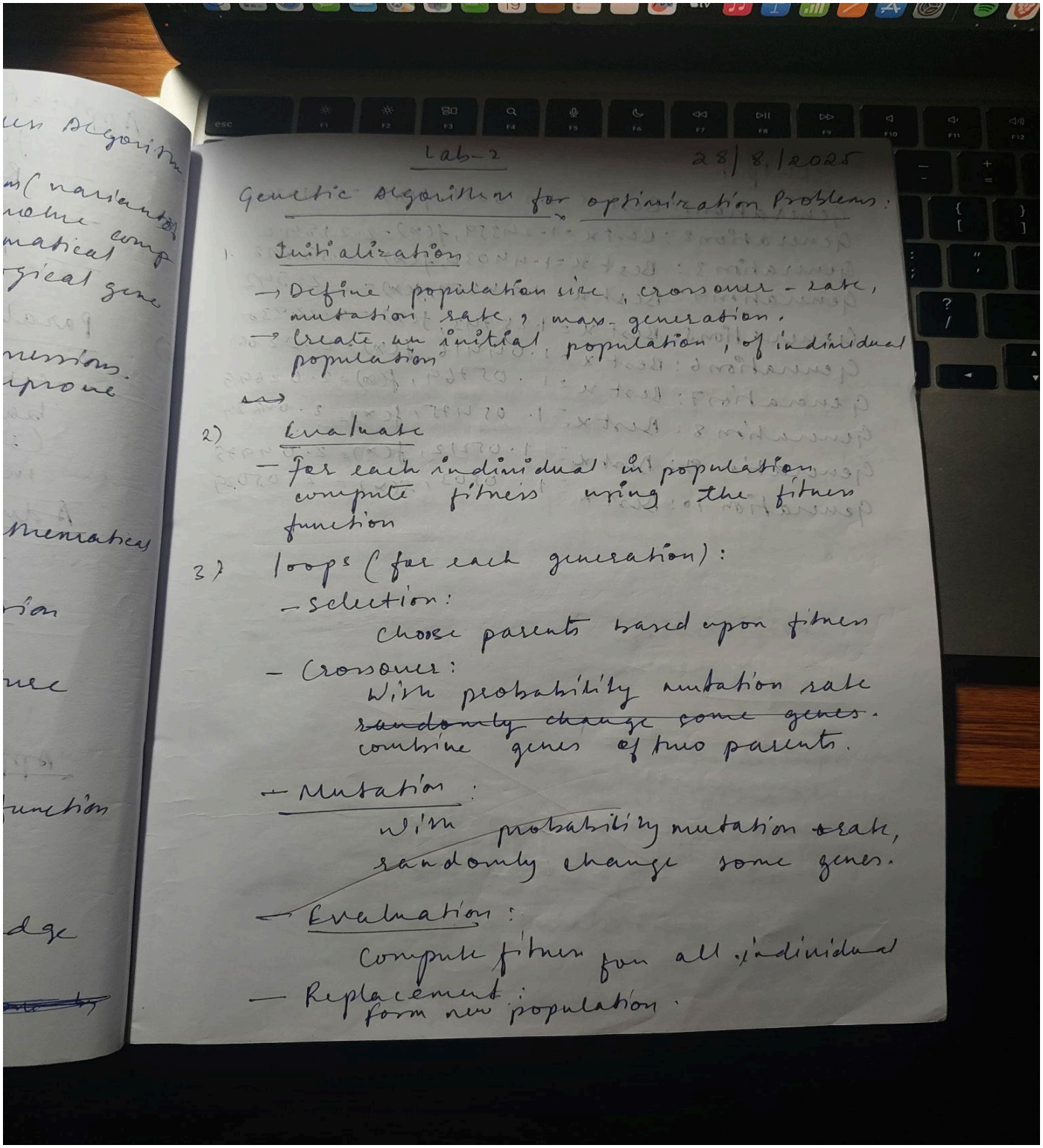
Github Link:

https://github.com/RudrakshSingh11/BIS-Lab.git

Program 1

Genetic Algorithm for Optimization Problems

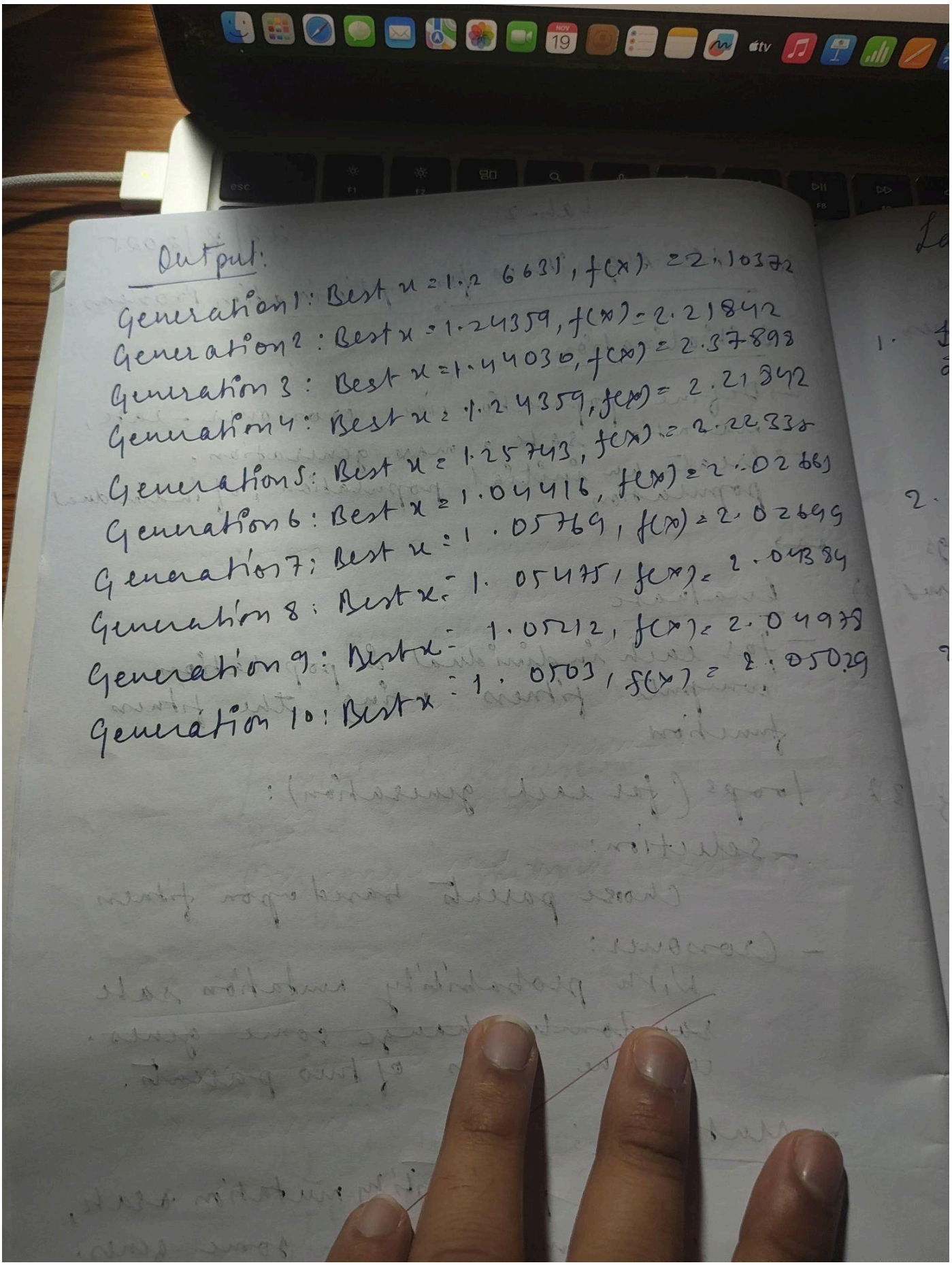
Algorithm:



Lab-2 28/8/2025

Genetic Algorithm for optimization Problems:

1. Initialization
   - → Define population size, crossover-rate, mutation rate, max-generation.
   - → Create an initial population, of individual population
2. Evaluate
   - For each individual in population compute fitness using the fitness function
3. loops (for each generation):
   - selection:
     choose parents based upon fitness
   - Crossover:
     With probability mutation rate ~~randomly change some genes.~~
     combine genes of two parents.
   - Mutation:
     with probability mutation rate, randomly change some genes.
   - Evaluation:
     Compute fitness for all individual
   - Replacement:
     form new population.



Output:

Generation 1: Best x = 1.26631, f(x) = 2.10372
Generation 2: Best x = 1.24359, f(x) = 2.21842
Generation 3: Best x = 1.44030, f(x) = 2.37893
Generation 4: Best x = 1.24359, f(x) = 2.21842
Generation 5: Best x = 1.25743, f(x) = 2.22338
Generation 6: Best x = 1.04416, f(x) = 2.02661
Generation 7: Best x = 1.05769, f(x) = 2.02699
Generation 8: Best x = 1.05475, f(x) = 2.04384
Generation 9: Best x = 1.05212, f(x) = 2.04933
Generation 10: Best x = 1.0503, f(x) = 2.05029

Code:

```
import numpy as np

# Objective function to maximize
def fitness_function(x):
    return x**2

# Initialize parameters
population_size = 50
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 50
lower_bound = -10
upper_bound = 10

# Create initial population
def initialize_population(size, lower, upper):
    return np.random.uniform(lower, upper, size)

# Evaluate fitness for the population
def evaluate_fitness(population):
    return np.array([fitness_function(x) for x in population])

# Selection using roulette wheel selection
def select_parents(population, fitness):
    total_fitness = np.sum(fitness)
    selection_probs = fitness / total_fitness
    parents_indices = np.random.choice(len(population), size=2,
p=selection_probs)
    return population[parents_indices]

# Crossover to create offspring
def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        return (parent1 + parent2) / 2 # Linear crossover
    return parent1

# Mutation to introduce diversity
def mutate(offspring):
    if np.random.rand() < mutation_rate:
        return np.random.uniform(lower_bound, upper_bound)
    return offspring

# Genetic Algorithm main function
def genetic_algorithm():
    # Initialize population
```

```
    population = initialize_population(population_size, lower_bound,
upper_bound)

    for generation in range(num_generations):
        # Evaluate fitness of the population
        fitness =
        evaluate_fitness(population)

        # Track the best solution
        best_fitness_idx = np.argmax(fitness)
        best_solution = population[best_fitness_idx]
        best_fitness_value = fitness[best_fitness_idx]

        print(f"Generation {generation}: Best Solution = {best_solution},
Fitness = {best_fitness_value}")

        # Create the next generation
        new_population = []
        for _ in range(population_size):
            parent1, parent2 = select_parents(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring)
            new_population.append(offspring)

        population = np.array(new_population)

    # Final evaluation
    final_fitness = evaluate_fitness(population)
    best_fitness_idx = np.argmax(final_fitness)
    best_solution = population[best_fitness_idx]
    best_fitness_value = final_fitness[best_fitness_idx]

    return best_solution, best_fitness_value

# Run the genetic algorithm
best_solution, best_fitness_value = genetic_algorithm()
print(f"Best Solution Found: x = {best_solution}, f(x) =
{best_fitness_value}")
```

Output :

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

Program 2

Particle Swarm Optimization for Function Optimization

Algorithm:

Lab 4: Particle Swarm Optimization (PSO algorithm).

1. Define the problem
   - Select the mathematical function (fcn) to optimize (minimize/maximize)
   - Define the search space (range of possible values for each variable)
2. Initialize Parameters
   - Number of particles in the swarm.
   - Number of dimensions of problem statement.
   - Number of ~~variables~~ iterations
   - Inertia Weight (w) — controls exploration vs exploitation.
   - Cognitive coefficient (c1) — tendency to follow personal best
   - Social Coefficient (c2) — tendency to follow global best.
3. Initialize Particle
   - Randomly assign each particle a position within search space.
   - Randomly assign an initial velocity to each particle.

- Set each particles' personal best position = current position.

4. Evaluate fitness
   - Compute the fitness of each particle using objective function.
   - Updating the personal best fitness if the current fitness is better.
   - Identify the global best position (best among particles).

5. Update Velocity

   Move each particle to its new position.

6. Update position. move particle to new position.
   $x(i)(t+1) = x(i)(t) + v(i)(t+1)$

7. Iterate:

   Repeat 4-6 steps until maximum

2. Output Best soln

   Return the global best position and its corresponding soln.

MG

Code:

```
mport numpy as np

# Step 1: Define the Problem def
fitness_function(position):
# Example: Minimize the Sphere function return
    np.sum(position**2)

# Step 2: Initialize Parameters def
initialize_parameters():
params = {
'N': 50,# Number of particles
'dim': 2,   # Dimensionality of the problem 'max_iter': 200,
        # Maximum number of iterations 'minx': -10,      #
        Minimum bound for position 'maxx': 10,           #
        Maximum bound for position 'w': 0.5,       # Inertia
        weight
'c1': 1.5,  # Cognitive coefficient 'c2': 1.5 #
        Social coefficient
    }
return params

# Step 3: Initialize Particles class
Particle:
    def _init_(self, position, velocity):
        self.position = position self.velocity
        = velocity self.bestPos =
        position.copy() self.bestFitness =
        float('inf')

def initialize_swarm(N, dim, minx, maxx): swarm =
    []
for _ in range(N):
position = np.random.uniform(minx, maxx, dim) velocity =
        np.random.uniform(-1, 1, dim)
        swarm.append(Particle(position, velocity))
return swarm

   # Step 4: Evaluate Fitness
   def evaluate_fitness(swarm):
       for particle in swarm:
           particle.fitness = fitness_function(particle.position)

   # Step 5: Update Velocities and Positions
   def update_particles(swarm, best_pos_swarm, w, c1, c2, minx, maxx):
       for particle in swarm:
           r1, r2 = np.random.rand(), np.random.rand()
```

```
            particle.velocity = (w * particle.velocity +
                                 r1 * c1 * (particle.bestPos - particle.position)
                                 r2 * c2 * (best_pos_swarm - particle.position))
            particle.position += particle.velocity
            # Clip position to be within bounds
            particle.position = np.clip(particle.position, minx, maxx)

    # Step 6: Iterate
    def pso():
        params = initialize_parameters()
        swarm = initialize_swarm(params['N'], params['dim'], params['minx'],
    params['maxx'])
        best_pos_swarm = swarm[0].position.copy()
        best_fitness_swarm = float('inf')

        for _ in range(params['max_iter']):
            evaluate_fitness(swarm)

            for particle in swarm:
                if particle.fitness < particle.bestFitness:
                    particle.bestFitness = particle.fitness
                    particle.bestPos = particle.position.copy()
                if particle.fitness < best_fitness_swarm:
                    best_fitness_swarm = particle.fitness
                    best_pos_swarm = particle.position.copy()

            update_particles(swarm, best_pos_swarm, params['w'], params['c1'],
    params['c2'], params['minx'], params['maxx'])

        # Step 7: Output the Best Solution
        return best_pos_swarm, best_fitness_swarm

    best_position, best_fitness = pso()

    print("Best Position:", best_position)
    print("Best Fitness:", best_fitness)
```

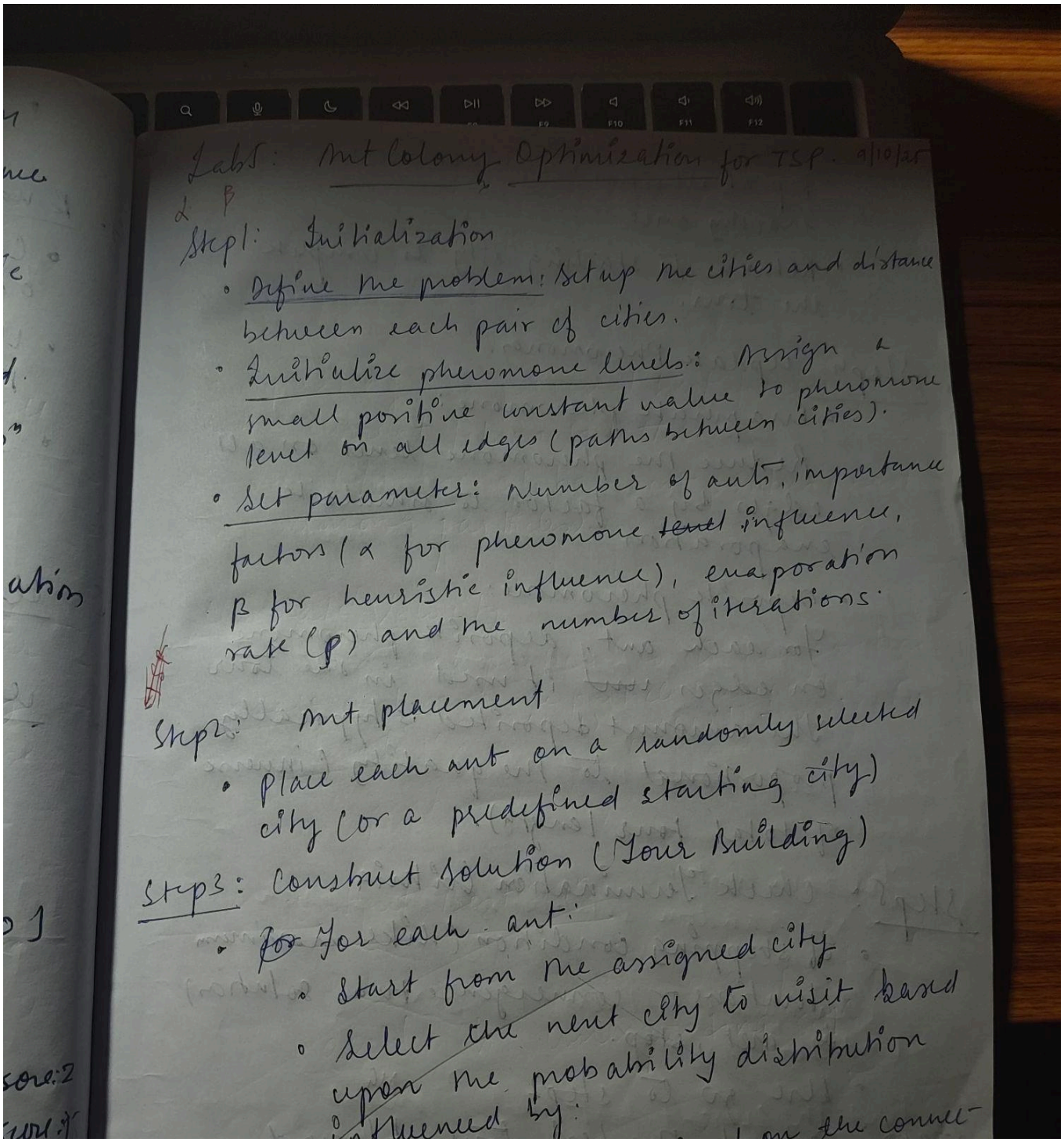
Output :

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

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:



Lab 5: Ant Colony Optimization for TSP. 9/10/25

α β

Step 1: Initialization

- Define the problem: Setup the cities and distance between each pair of cities.
- Initialize pheromone levels: Assign a small positive constant value to pheromone level on all edges (paths between cities).
- Set parameter: Number of ants, importance factors (α for pheromone ~~level~~ influence, β for heuristic influence), evaporation rate (ρ) and the number of iterations.

Step 2: Ant placement

- Place each ant on a randomly selected city (or a predefined starting city)

Step 3: Construct Solution (Tour Building)

- ~~for~~ For each ant:
  - Start from the assigned city
  - Select the next city to visit based upon the probability distribution influenced by:

- Repeat all cities until all are visited exactly once.
- Return to starting city to complete the tour.

Step4: Update Pheromones.

- Evaporate pheromone:
  Reduce the pheromone level on all edges by a factor to simulate evaporation
- Deposit pheromone:
  For each ant, deposit pheromone on edges that it used in the tour. The amount deposited is typically proportional to the quality (inverse of total tour length).

Step5: Check Termination Criteria

- If stopping condition (like maximum iterations or convergence to a solution) is met, stop.
- Else go to step2

MG.

Step6: Output the best tour found

- Return the best (shortest) tour found by the ants during iteration

Code:

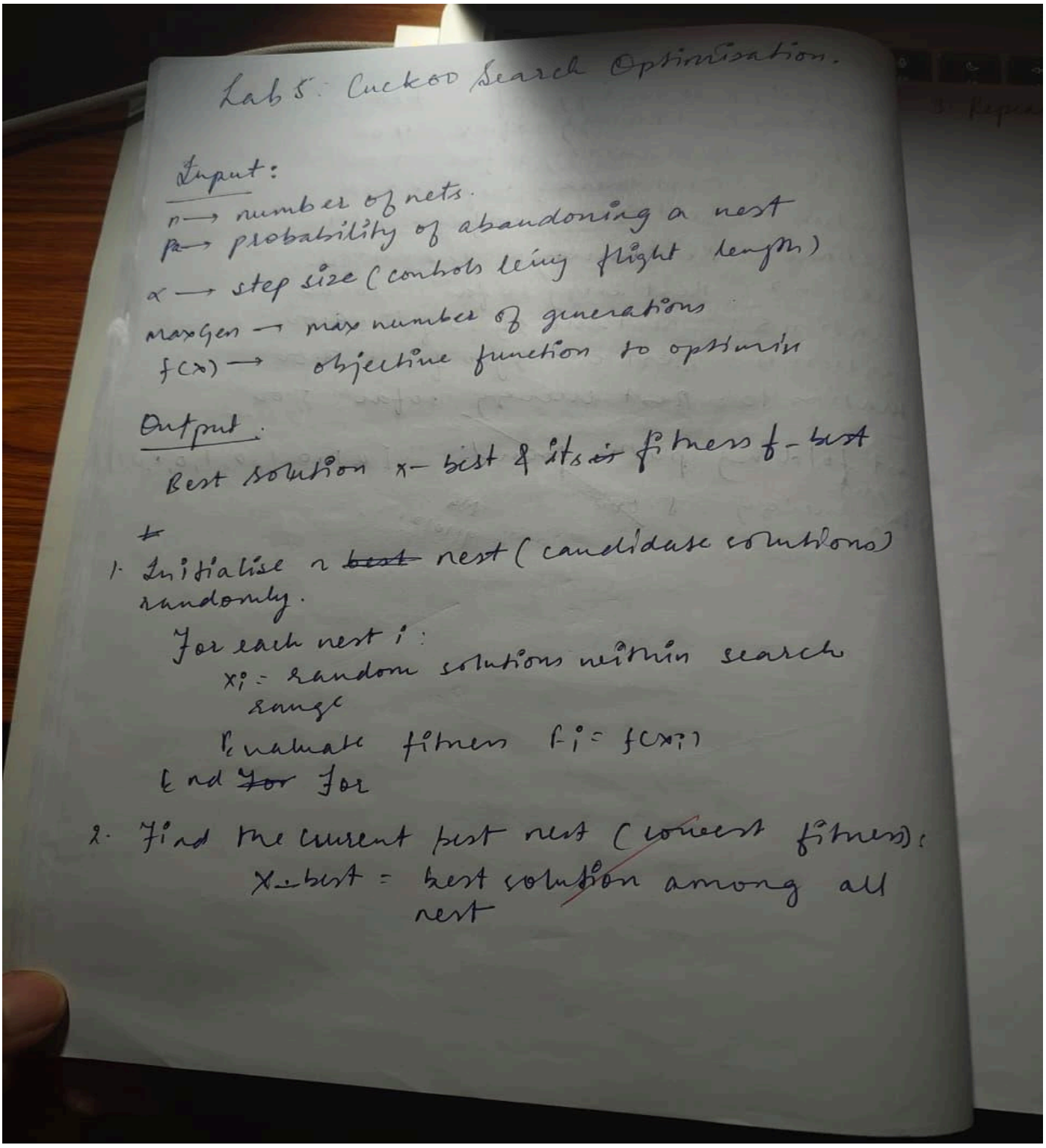
```
import numpy as np
import random

   class AntColony:
       def _init_(self, cities, num_ants=10, alpha=1.0, beta=2.0, rho=0.5,
   iterations=100):
           self.cities = cities
           self.num_ants = num_ants
           self.alpha = alpha
           self.beta = beta
           self.rho = rho
           self.iterations = iterations
           self.num_cities = len(cities)
           self.pheromone = np.ones((self.num_cities, self.num_cities))
           self.distance = self.calculate_distances()
       def calculate_distances(self):
           distances = np.zeros((self.num_cities, self.num_cities))
           for i in range(self.num_cities):
               for j in range(i + 1, self.num_cities):
                   distances[i][j] = distances[j][i] =
   np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
           return distances
       def select_next_city(self, current_city, visited):
           probabilities = []
           for next_city in range(self.num_cities):
               if next_city not in visited:
                   pheromone = self.pheromone[current_city][next_city] **
   self.alpha self.beta                                    heuristic = (1 /
                                                           self.distance[current_city][next_ci
                                                           ty]) **
                                                           probabilities.append(pheromone *
                                                           heuristic)
                                                   else:
                                                           probabilities.append(0)
           total = sum(probabilities)
           probabilities = [p / total for p in probabilities]
           return np.random.choice(range(self.num_cities), p=probabilities)
       def construct_solution(self):
           for _ in range(self.num_ants):
               visited = [0]
               current_city = 0
               for _ in range(1, self.num_cities):
                   next_city = self.select_next_city(current_city, visited)
                   visited.append(next_city)
                   current_city = next_city
               visited.append(0) # Return to starting city
```

```
yield visited
    def update_pheromones(self, solutions): self.pheromone
        *= (1 - self.rho) # Evaporation for solution in
        solutions:
length = self.calculate_tour_length(solution) pheromone_deposit = 1 / length
            for i in range(len(solution) - 1):
                self.pheromone[solution[i]][solution[i + 1]] +=
pheromone_deposit
def calculate_tour_length(self, solution):
        return sum(self.distance[solution[i]][solution[i + 1]] for i in
range(len(solution) - 1))

    def run(self): best_solution =
        None
best_length = float('inf')
for _ in range(self.iterations):
solutions = list(self.construct_solution()) self.update_pheromones(solutions)
for solution in solutions:
length = self.calculate_tour_length(solution) if length <
                best_length:
best_length = length best_solution = solution
          return best_solution, best_length
   cities = [(0, 0), (1, 2), (2, 1), (4, 4), (2, 4)]
aco = AntColony(cities)
best_route, best_distance = aco.run()
print("Best Route:", best_route) print("Best
Distance:", best_distance)

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

Program 4

Cuckoo Search (CS)

Algorithm:



3. Repeat until maximum generations:
For each cuckoo i:
Generate a new soln by Lévy flight:

$new\_x = x_i + \alpha \times levy(x)$

Evaluate fitness of new solution:
$f_{new} = f(new\_x)$.
Randomly choose another nest j:
If $f_{new} < f_j$;
Replace nest j with new_x.
End If.
End For.

A fraction (Pa) of worst nest are abandoned:
For each nest:
If rand() < Pa:
Replace with a new random soln.
End if
End for
Keep the best nest found so far:
If any new soln is better than x_best:
Update x_best & f_best

MG
16/10/25.

Code:

```
import numpy as np
import math
# Objective function to optimize (example: Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Lévy Flight distribution
def levy_flight(beta=1.5, size=1):
    sigma_u = (math.gamma(1 + beta) * np.sin(np.pi * beta / 2) /
               math.gamma((1 + beta) / 2) * beta * (2 ** ((beta - 1) /
2)))**(1 / beta)
    u = np.random.normal(0, sigma_u, size)
    v = np.random.normal(0, 1, size)
    step = u / (np.abs(v) ** (1 / beta))
    return step

# Cuckoo Search Algorithm
def cuckoo_search(objective_function, dim, lower_bound, upper_bound,
num_nests=25, max_iter=100, pa=0.25):
    # Initialize nests with random solutions within bounds
    nests = np.random.rand(num_nests, dim) * (upper_bound - lower_bound) +
lower_bound
    fitness = np.apply_along_axis(objective_function, 1, nests)

    # Initialize the best solution
    best_nest_idx = np.argmin(fitness)
    best_nest = nests[best_nest_idx]
    best_fitness = fitness[best_nest_idx]

    # Iterate for a fixed number of generations or until convergence
    for iteration in range(max_iter):
        for i in range(num_nests):
            # Generate a new solution using Lévy flight
            step = levy_flight(size=dim)
            new_nest = nests[i] + 0.01 * step
            new_nest = np.clip(new_nest, lower_bound, upper_bound)

            # Evaluate the new solution
            new_fitness = objective_function(new_nest)

            # If the new solution is better, replace the old solution
```

```
            if new_fitness < fitness[i]:
                nests[i] = new_nest fitness[i]
                = new_fitness

# Abandon the worst nests for i in
        range(num_nests):
if np.random.rand() < pa: # Probability to abandon
nests[i] = np.random.rand(dim) * (upper_bound - lower_bound)
+ lower_bound
fitness[i] = objective_function(nests[i])

# Find the current best nest best_nest_idx =
        np.argmin(fitness) best_nest =
        nests[best_nest_idx] best_fitness =
        fitness[best_nest_idx]

        # print(f"Iteration {iteration+1}, Best Fitness: {best_fitness}") return

    best_nest, best_fitness

# Example usage of Cuckoo Search

# Define the problem dimensions and bounds dim
= 5 # Dimension of the solution space
lower_bound = -5 # Lower bound of the search space
upper_bound = 5 # Upper bound of the search space

# Run Cuckoo Search
best_solution, best_fitness = cuckoo_search(objective_function, dim, lower_bound,
upper_bound, num_nests=25, max_iter=100, pa=0.25)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

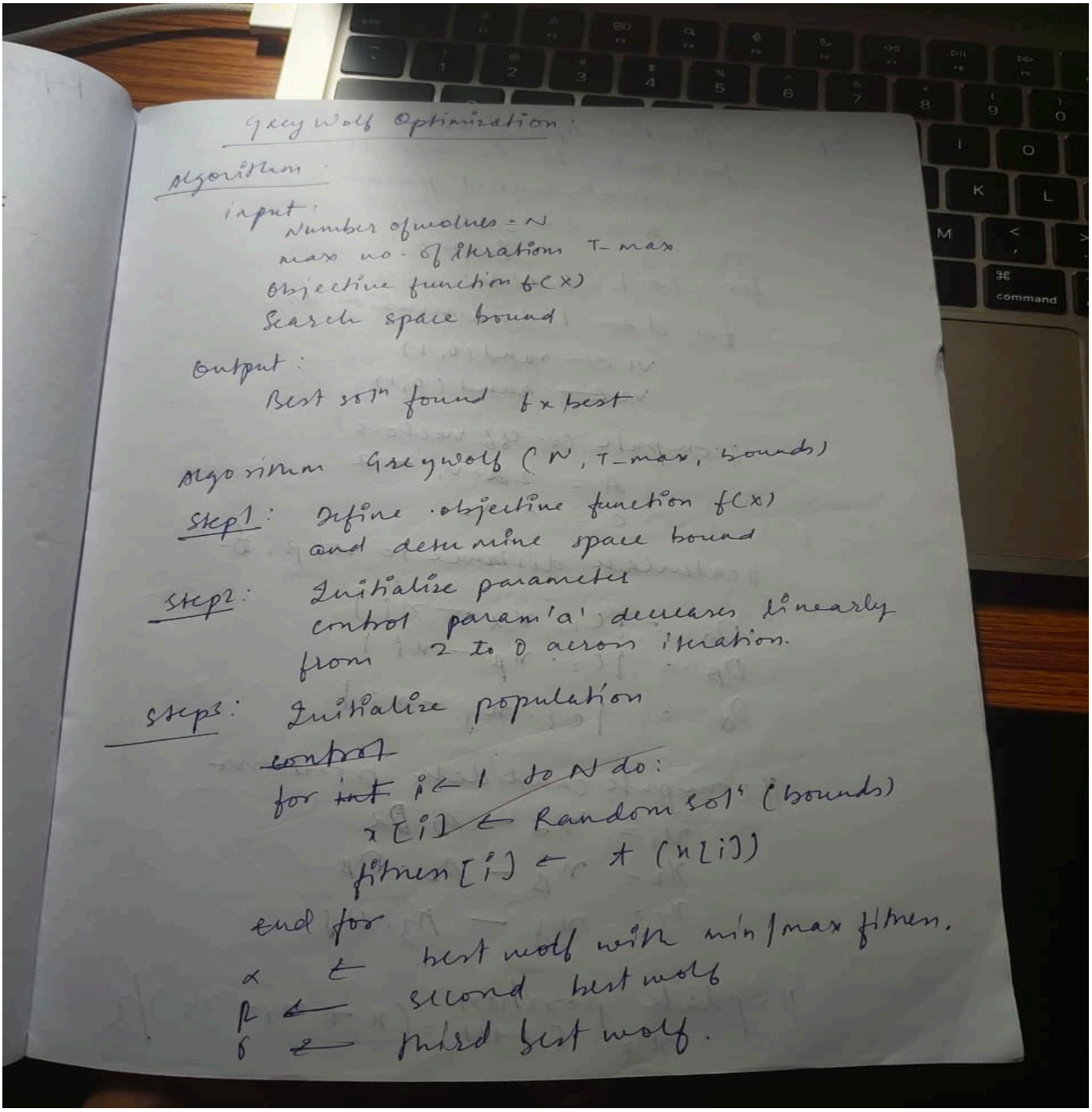
Output :

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

Program 5

Grey Wolf Optimizer (GWO)

Algorithm:



Step 4: Iterative optimization

for $t \leftarrow 1$ to T_{max} do:

// update control param

$$a = 2 - \frac{2t}{T_{max}}$$

for $i \leftarrow 1$ to N do:

for $d \leftarrow 1$ do dimension do:

$v_1 \leftarrow$ rand(0, 1)

$v_2 \leftarrow$ rand(0, 1)

// compute co-eff vectors

$$A = 2a \cdot v_1 - a$$

$$C = 2 \cdot v_2$$

// calculate distance to α, β, γ

$$D_\alpha = |C_1 \cdot x_\alpha - x_i|$$

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

$$D_\gamma = |C_3 \cdot x_\gamma - x_i|$$

// compute candidate positions

$$x_1 = x_\alpha - A_1 \cdot D_\alpha$$

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

$$x_3 = x_\gamma - A_3 \cdot D_\gamma$$

// update position

$$x_{i,d}(t+1) = (x_1 + x_2 + x_3)/3$$

Code:

```
import numpy as np

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

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

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

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

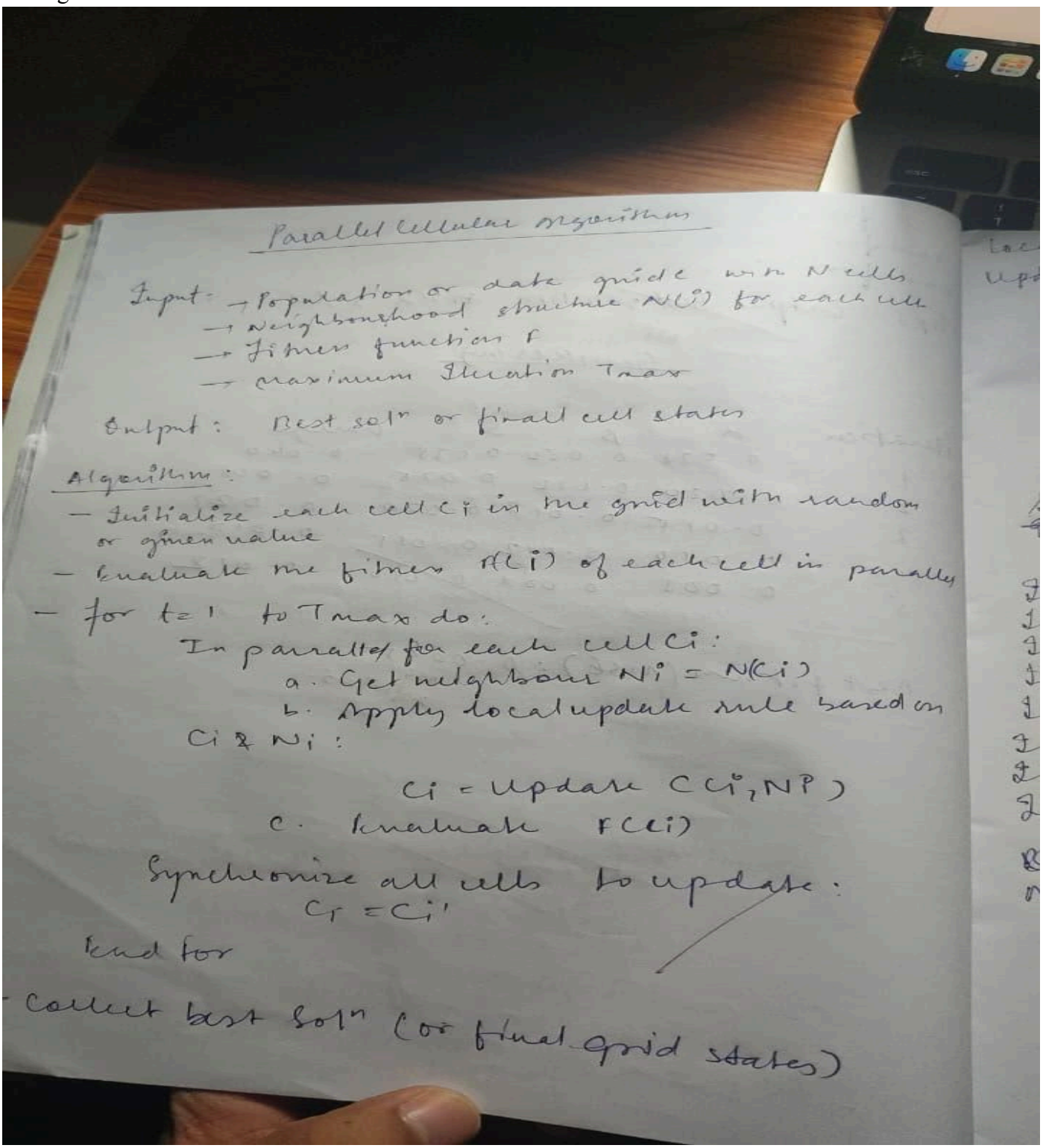
Output :

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

Program 6

Parallel Cellular Algorithms and Programs

Algorithm:

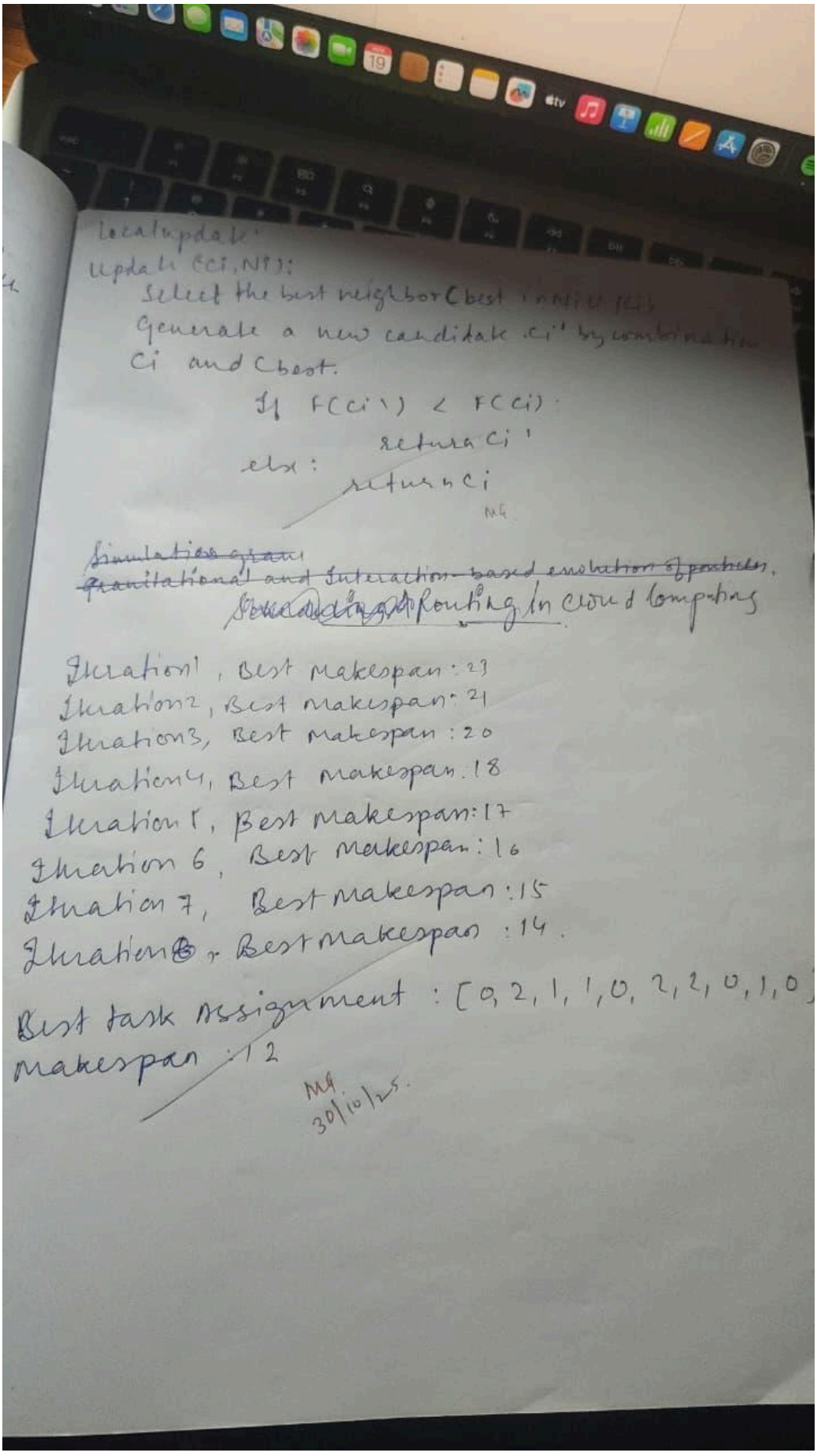


Parallel Cellular Algorithms

Input: → Population or data grid with N cells
→ Neighborhood structure N(i) for each cell
→ Fitness function f
→ maximum Iteration Tmax

Output: Best soln or final cell states

Algorithm:
- Initialize each cell Ci in the grid with random or given value
- Evaluate the fitness f(i) of each cell in parallel
- for t=1 to Tmax do:
  In parallel for each cell Ci:
  a. Get neighbours Ni = N(Ci)
  b. Apply local update rule based on Ci & Ni:
  Ci = Update (Ci, Ni)
  c. Evaluate f(Ci)
  Synchronize all cells to update:
  Ci = Ci'
  End for
- Collect best soln (or final grid states)



localupdate:
Update(Ci, NT):
Select the best neighbor Cbest ∈ N(Ci)
Generate a new candidate Ci' by combining
Ci and Cbest.
If F(Ci') < F(Ci):
return Ci'
else:
return Ci
MG

~~Simulation of ant~~
~~Gravitational and Interaction-based evolution of particles.~~
~~Scheduling~~ Routing in cloud computing

Iteration1, Best makespan: 23
Iteration2, Best makespan: 21
Iteration3, Best makespan: 20
Iteration4, Best makespan. 18
Iteration 5, Best makespan: 17
Iteration 6, Best makespan: 16
Iteration 7, Best makespan: 15
Iteration 8, Best makespan: 14.

Best task Assignment: [0, 2, 1, 1, 0, 2, 2, 0, 1, 0]
makespan: 12

MG
30/10/25.

Code:

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

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

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

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

def fitness(cell):
return rastrigin(cell)

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

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

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

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

parallel_cellular_algorithm()
```

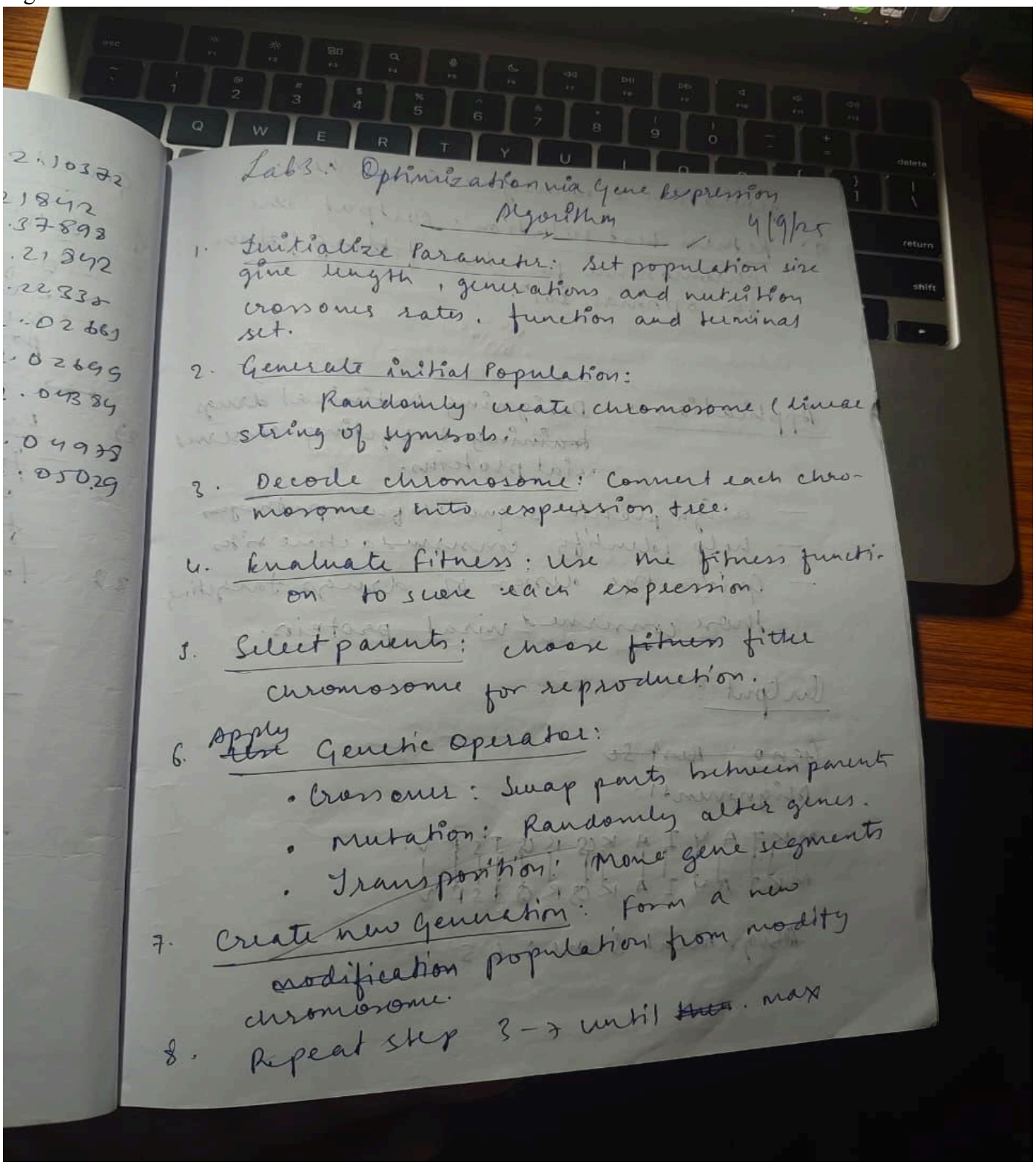
Output:

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

Program 7

Optimization via Gene Expression Algorithms

Algorithm:



Lab 3: Optimization via Gene Expression Algorithm 4/9/25

1. Initialize Parameter: Set population size gene length, generations and mutation crossovers rates. function and terminal set.
2. Generate initial Population:
   Randomly create chromosome (linear string of symbols.
3. Decode chromosome: Convert each chromosome into expression tree.
4. Evaluate fitness: Use the fitness function to score each expression.
5. Select parents: choose fitter chromosome for reproduction.
6. Apply Genetic Operator:
   - Crossover: Swap parts between parent
   - Mutation: Randomly alter genes.
   - Transposition: Move gene segments
7. Create new Generation: Form a new modification population from modify chromosome.
8. Repeat step 3 → until max

generation as convergence is reached.

9: Return best solution: Output the chromosome for best ~~fitting~~ fitness for optimal soln.

MG 4/9/25

Application: Designing ~~Drug~~ antiviral drugs ~~training~~ targeting conserved viral proteins.

- align protein sequences from pathogens
- help identify conserved active sites
- Guides the design of drugs targeting those conserved viral proteins.

Output:

~~Geno: Best Sc~~

Alignment:

```
M K T A Y I A K Q R Q I S F V.
| | | | | | | | | | | | | |
M K T A Y I A K Q R Q D S F V
```

Alignment Score: 68.

Code:

```
import numpy as np

# Define the mathematical function to optimize (example: minimize f(x) = x^2)
def optimization_function(x):
    return np.sum(x**2) # Modify this for other functions to optimize

# Parameters
POPULATION_SIZE = 50 # Number of individuals
GENE_LENGTH = 5       # Number of genes (dimensions of the problem)
MUTATION_RATE = 0.1   # Probability of mutation
CROSSOVER_RATE = 0.7 # Probability of crossover
GENERATIONS = 100     # Number of generations
SEARCH_SPACE = (-10, 10) # Range of values for genes

# Initialize Population
def initialize_population():
    return np.random.uniform(SEARCH_SPACE[0], SEARCH_SPACE[1],
(POPULATION_SIZE, GENE_LENGTH))

# Evaluate Fitness (lower is better for minimization)
def evaluate_fitness(population):
    fitness = np.array([optimization_function(ind) for ind in population])
    return fitness

# Selection (Roulette Wheel Selection)
def select_parents(population, fitness):
    # Convert fitness to probabilities (lower fitness is better)
    inverted_fitness = 1 / (fitness + 1e-6) # Avoid division by zero
    selection_prob = inverted_fitness / np.sum(inverted_fitness)
    selected_indices = np.random.choice(np.arange(POPULATION_SIZE),
size=POPULATION_SIZE, p=selection_prob)
    return population[selected_indices]
```

```
# Crossover (Blend Crossover)
def crossover(parents):
    offspring = np.empty_like(parents) for
    i in range(0, POPULATION_SIZE, 2):
        p1, p2 = parents[i], parents[i+1]
        if np.random.rand() < CROSSOVER_RATE:
            alpha = np.random.rand() # Blending factor
            offspring[i] = alpha * p1 + (1 - alpha) * p2
            offspring[i+1] = alpha * p2 + (1 - alpha) * p1
        else:
            offspring[i], offspring[i+1] = p1, p2
    return offspring

# Mutation (Random Perturbation)
def mutate(offspring):
    for i in range(POPULATION_SIZE):
        if np.random.rand() < MUTATION_RATE:
            mutation_point = np.random.randint(0, GENE_LENGTH)
            offspring[i][mutation_point] += np.random.uniform(-1, 1)
            # Keep within search space
            offspring[i][mutation_point] =
np.clip(offspring[i][mutation_point], SEARCH_SPACE[0], SEARCH_SPACE[1])
    return offspring

# Gene Expression (Translate Genetic Code into Solutions)
def gene_expression(genes):
    # In this simple example, the genes directly represent the solution
    return genes

# Main Algorithm
def gene_expression_algorithm():
    # Initialize population
    population = initialize_population()
    best_solution = None
    best_fitness = float('inf')

    # Iterate through generations
    for generation in range(GENERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
```

Output:

```
Generation 2: Best Fitness = 11.641082640808637
…
```

```
Generation 99: Best Fitness = 0.02233046748484963
Generation 100: Best Fitness = 0.02233046748484963

Optimal Solution Found:
 Best Solution: [ 0.07226226 -0.11854791 0.03245473 -0.01236219 0.04299877]
 Best Fitness: 0.0223304674848496
```