VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Rushil M (1BM22CS225)

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
Sep-2024 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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Rushil M (1BM22CS225),** 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.

Swathi S Assistant Professor Department of CSE, BMSCE	Dr. Kavita Professor & HOD Department of CSE, BMSCE

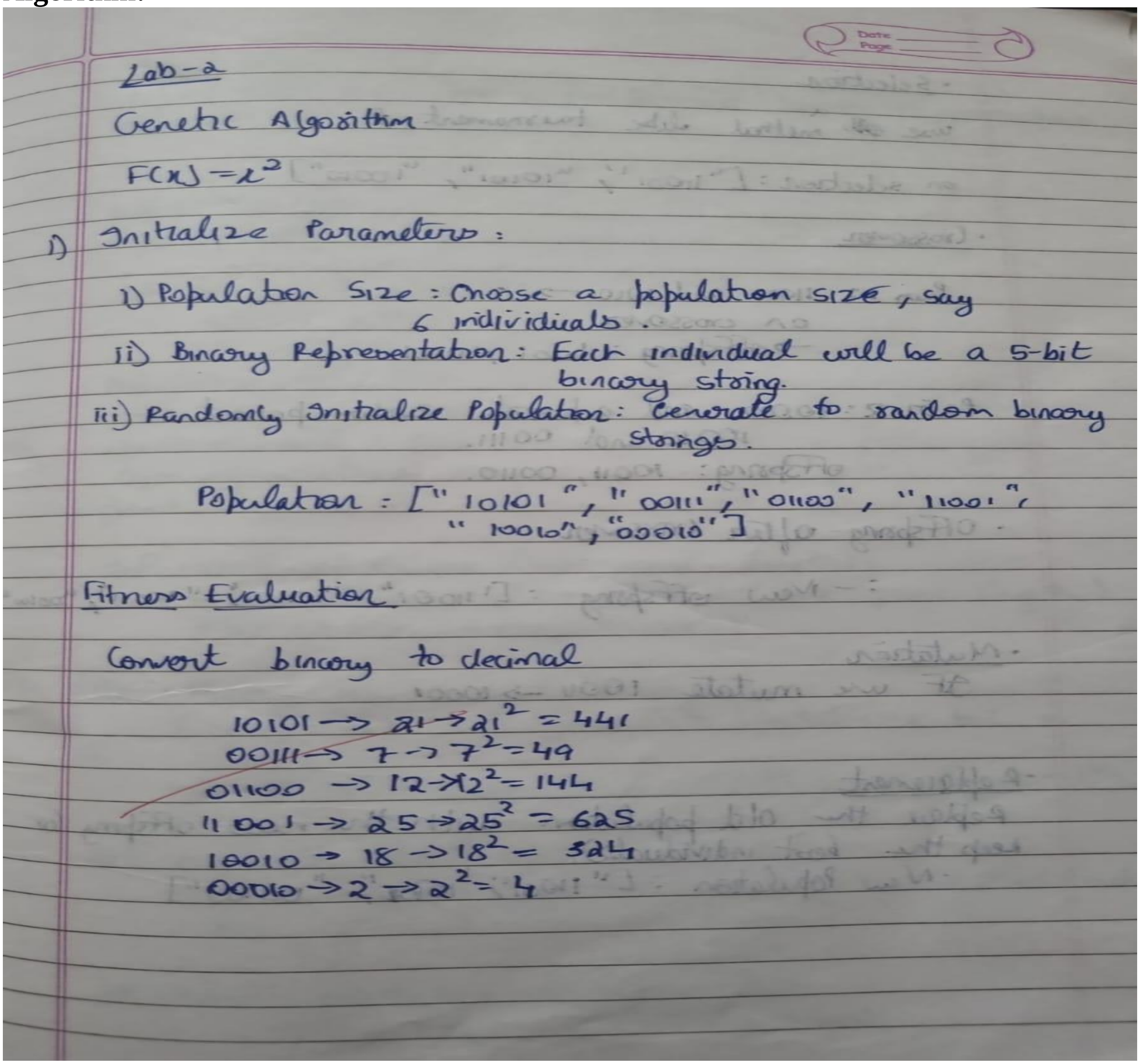
Index

Github Link:
https://github.com/Rushil0707/BISLab

1. Genetic Algorithm

A **genetic algorithm (GA)** is a search heuristic inspired by the process of natural selection and genetics. It is used to solve optimization and search problems. The algorithm simulates the process of natural evolution, where the fittest individuals are selected to reproduce and pass their genes to the next generation, leading to the gradual improvement of solutions.

Algorithm:



Lab-2

Genetic Algorithm

$F(x) = x^2$

1) Initialize Parameters:

i) Population Size: Choose a population size, say 6 individuals.

ii) Binary Representation: Each individual will be a 5-bit binary string.

iii) Randomly Initialize Population: Generate 6 random binary strings.

Population: ["10101", "00111", "01100", "11001", "10010", "00010"]

Fitness Evaluation

Convert binary to decimal

$10101 \rightarrow 21 \rightarrow 21^2 = 441$

$00111 \rightarrow 7 \rightarrow 7^2 = 49$

$01100 \rightarrow 12 \rightarrow 12^2 = 144$

$11001 \rightarrow 25 \rightarrow 25^2 = 625$

$10010 \rightarrow 18 \rightarrow 18^2 = 324$

$00010 \rightarrow 2 \rightarrow 2^2 = 4$

Iteration

Repeat steps 2-6 for a predetermined number of g
or until convergence

```
import random
import numpy as np

def fitness_function(x):
    return x ** 2

population_size = 10
mutation_rate = 0.01
crossover_rate = 0.8
num_generations = 100
gene_length = 10

def create_population(size, gene_length):
    return [np.random.randint(0, 2, gene
    tolist() for _ in range(size)]

def binary_to_decimal(binary):
    binary_str = "".join(str(bit) for
    binary)
    return int(binary_str, 2) / (2 ** gene_

def evaluate_populations(population):
    return [fitness_function(binary_to_d
    for individual in population]

def select(population, fitness_score):
        total_fitness = sum(fitness_scores)
```

Code:

```
import random
import numpy as np

def fitness_function(x):
    return x**2

population_size = 10
mutation_rate = 0.01
crossover_rate = 0.8
num_generations = 10
gene_length = 10

def create_population(size, gene_length):
    return [np.random.randint(0, 2, gene_length).tolist() for _ in range(size)]

def binary_to_decimal(binary):
    binary_str = ''.join(str(bit) for bit in binary)
    return int(binary_str, 2) / ((2**gene_length) - 1) * 10 - 5

def evaluate_population(population):
    return [fitness_function(binary_to_decimal(individual)) for individual in population]

def select(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [fitness / total_fitness for fitness in fitness_scores]
    return population[np.random.choice(range(len(population)), p=selection_probs)]

def mutate(individual):
    for i in range(gene_length):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]
    return individual

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        point = random.randint(1, gene_length - 1)
        return parent1[:point] + parent2[point:], parent2[:point] + parent1[point:]
    return parent1, parent2

def genetic_algorithm():
    population = create_population(population_size, gene_length)

    for generation in range(num_generations):
        fitness_scores = evaluate_population(population)
        best_fitness = max(fitness_scores)
```

```
        best_individual = population[fitness_scores.index(best_fitness)]

        print(f"Generation {generation}: Best Fitness = {best_fitness:.4f}")

        new_population = []
        while len(new_population) < population_size:
            parent1 = select(population, fitness_scores)
            parent2 = select(population, fitness_scores)
            offspring = crossover(parent1, parent2)
            new_population.extend([mutate(child) for child in offspring])

        population = new_population[:population_size]
        best_fitness = max(fitness_scores)
        best_individual = population[fitness_scores.index(best_fitness)]

    best_solution = binary_to_decimal(best_individual)
    print(f"Best Solution: {best_solution}")

# Run the genetic algorithm
genetic_algorithm()
```

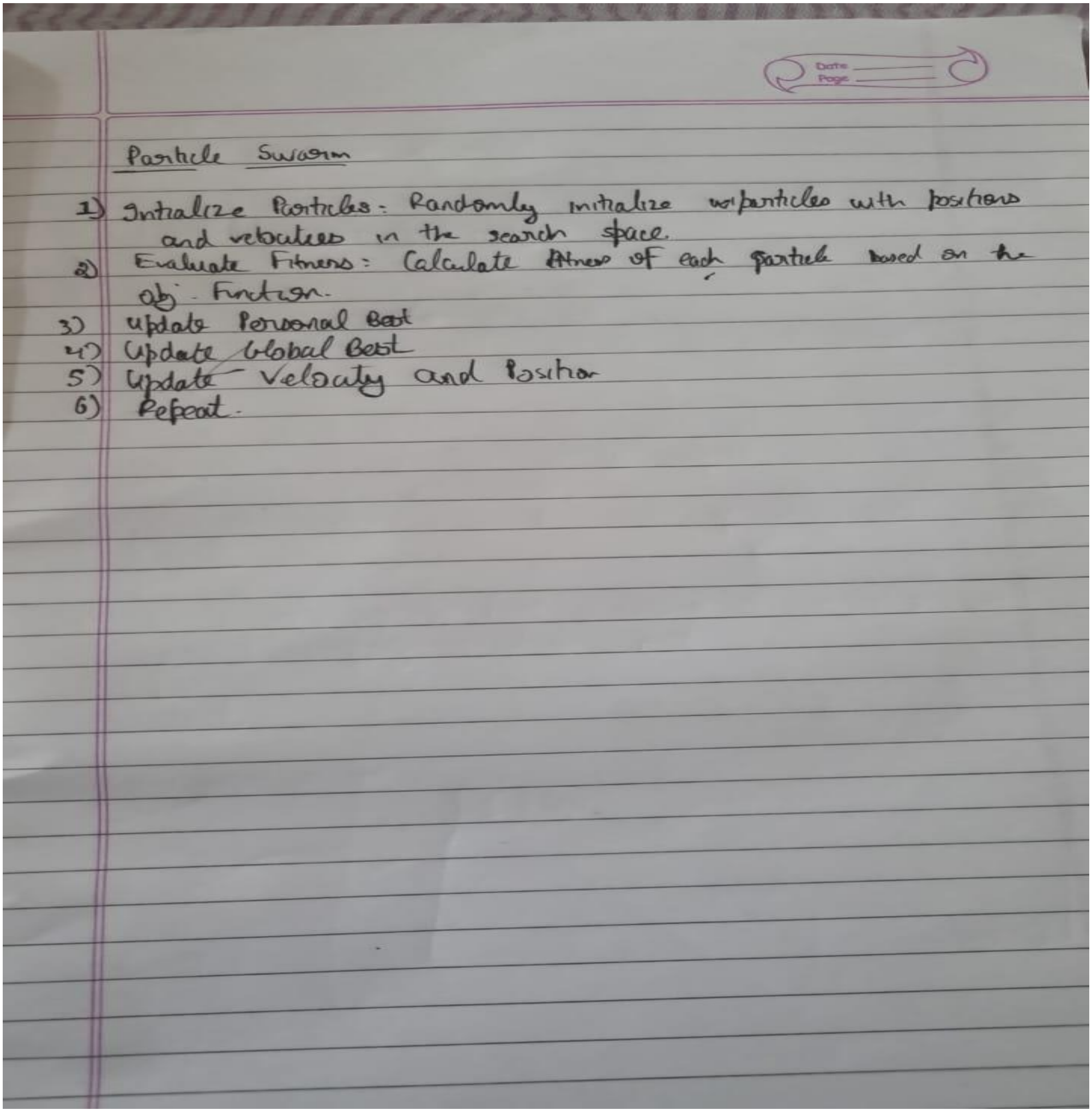
Output:

```
Generation 1: Best Fitness = 92.9885
Generation 2: Best Fitness = 70.8814
Generation 3: Best Fitness = 76.0476
Generation 4: Best Fitness = 76.0476
Generation 5: Best Fitness = 70.7608
Generation 6: Best Fitness = 70.7328
Generation 7: Best Fitness = 70.7328
Generation 8: Best Fitness = 70.6885
Generation 9: Best Fitness = 76.1728
Best fitness found: 72.0534
```

2. Particle Swarm Optimisation for function Optimisation

Particle Swarm Optimization (PSO) is a heuristic optimization algorithm inspired by the social behavior of birds flocking or fish schooling. It is used to find optimal solutions by mimicking the collective behavior of a swarm of particles in a search space.

Algorithm:



Particle Swarm

1) Initialize Particles: Randomly initialize w/particles with positions and velocities in the search space.
2) Evaluate Fitness: Calculate fitness of each particle based on the obj. function.
3) update Personal Best
4) Update Global Best
5) Update Velocity and Position
6) Repeat.

Code:

```
import numpy as np
import multiprocessing

def fitness_function(matrix):
    return np.var(matrix)

def update_cell(i, j, matrix, size):
    neighbors = []
    for di in range(-1, 2):
        for dj in range(-1, 2):
            ni, nj = (i + di) % size, (j + dj) % size
            neighbors.append(matrix[ni, nj])
    return np.mean(neighbors)

def parallel_cell_update(matrix, size):
    pool = multiprocessing.Pool(processes=multiprocessing.cpu_count())
    result = []
    for i in range(size):
        for j in range(size):
            result.append(pool.apply_async(update_cell, (i, j, matrix, size)))
    pool.close()
    pool.join()

    new_matrix = np.array([r.get() for r in result]).reshape(matrix.shape)
    return new_matrix

def cellular_optimization(matrix, max_iter=10):
    size = matrix.shape[0]
    for t in range(max_iter):
        matrix = parallel_cell_update(matrix, size)
        print(f"Iteration {t + 1}: Matrix:\n{matrix}")
    return matrix

matrix = np.random.rand(5, 5)
optimized_matrix = cellular_optimization(matrix)
print("Optimized Matrix:\n", optimized_matrix)
```

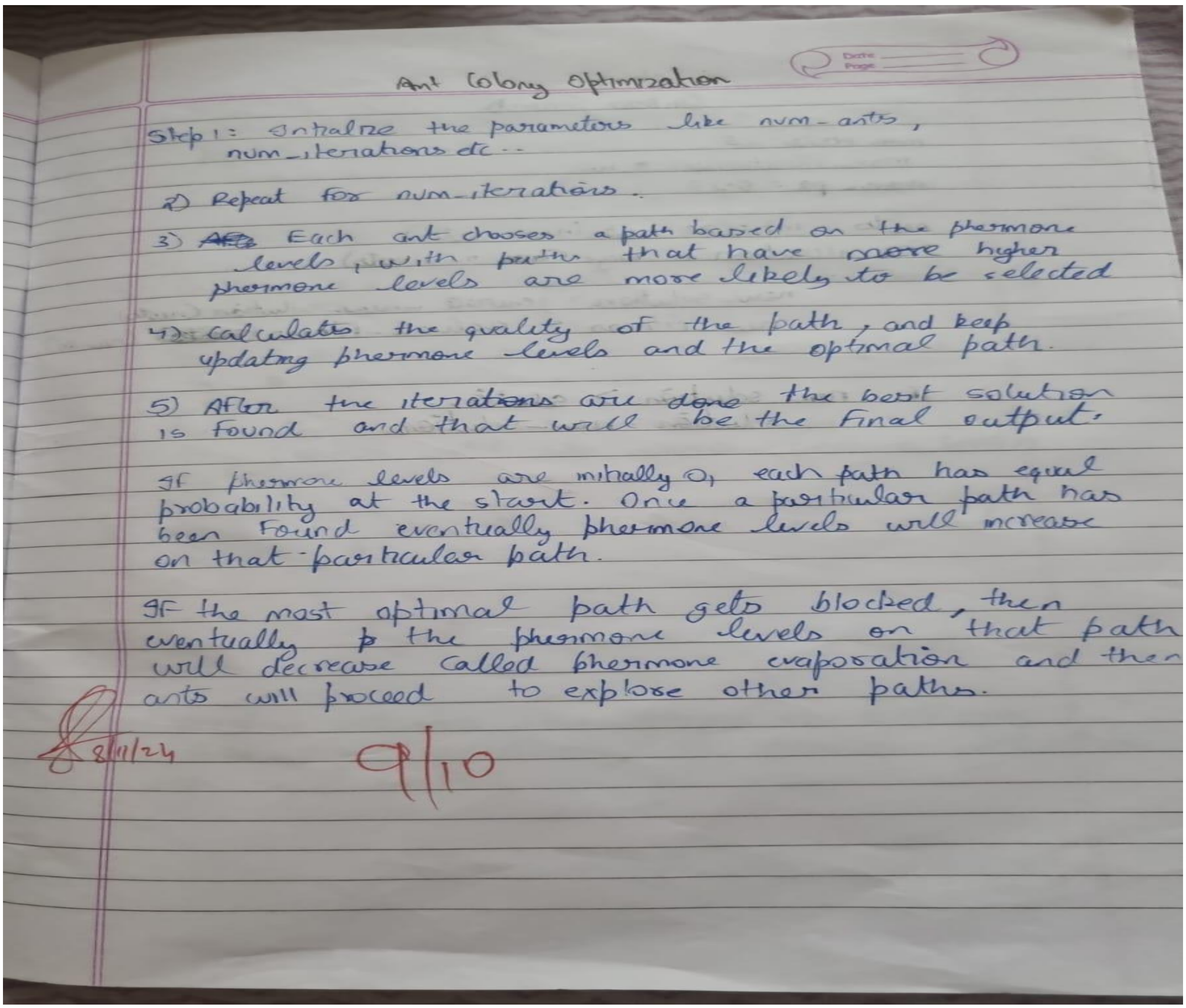
Output:

```
Optimized Matrix:
 [[0.46875618 0.46865932 0.46854258 0.46856729 0.46869931]
 [0.46865422 0.46855803 0.46844042 0.46846392 0.46859606]
 [0.46857991 0.46848458 0.46836626 0.46838846 0.46852051]
 [0.46863597 0.46854049 0.46842259 0.4684452  0.46857708]
 [0.46874493 0.46864851 0.46853158 0.46855574 0.4686876 ]]
```

3. Ant Colony Optimisation

Ants in nature deposit pheromones on their paths as they move. The intensity of the pheromone on a path influences the probability that other ants will choose that path. Over time, the pheromone trails strengthen on paths that are frequently used and weak on less frequently used ones. This behavior leads to the discovery of the shortest or optimal path between the ant colony and a food source. ACO mimics this process to solve various optimization problems, like the traveling salesman problem (TSP), vehicle routing problems, and others.

Algorithm:



Ant Colony Optimization

Step 1: Initialize the parameters like num_ants, num_iterations etc..

2) Repeat for num_iterations.

3) ~~A~~ Each ant chooses a path based on the phermone levels, with paths that have ~~more~~ higher phermone levels are more likely to be selected

4) Calculate the quality of the path, and keep updating phermone levels and the optimal path.

5) After the iterations are done the best solution is found and that will be the Final output.

If phermone levels are initially 0, each path has equal probability at the start. Once a particular path has been Found eventually phermone levels will increase on that particular path.

If the most optimal path gets blocked, then eventually ~~p~~ the phermone levels on that path will decrease called phermone evaporation and then ants will proceed to explore other paths.

8/11/24

9/10

Code:

```
import numpy as np
import random

# Parameters
num_ants = 10
num_iterations = 100
alpha = 1  # Importance of pheromone
beta = 2   # Importance of heuristic information
evaporation_rate = 0.5
pheromone_constant = 1.0
num_nodes = 20  # Number of nodes or elements in the problem space

# Initialize pheromone matrix to zero
pheromone_matrix = np.zeros((num_nodes, num_nodes))  # Explicitly set to 0

# Heuristic information (problem-specific, this should be adapted)
def heuristic_info(i, j):
    # Placeholder heuristic: in a real problem, replace this with problem-specific values
    return 1.0 / (abs(i - j) + 1e-10)

# Initialize ants' paths and lengths
def initialize_ants(num_ants, num_nodes):
    return [random.sample(range(num_nodes), num_nodes) for _ in range(num_ants)]

# Evaluate the fitness of a path (problem-specific)
def fitness_function(path):
    # Placeholder fitness function: in a real problem, define the cost/fitness of a path
    return sum(abs(path[i] - path[i+1]) for i in range(len(path) - 1))

# Update pheromones
def update_pheromones(pheromone_matrix, ants, fitnesses):
    # Evaporate pheromones
    pheromone_matrix *= (1 - evaporation_rate)

    # Deposit new pheromones based on the fitness of each ant's path
    for ant, fitness in zip(ants, fitnesses):
        for i in range(len(ant) - 1):
            pheromone_matrix[ant[i]][ant[i+1]] += pheromone_constant / (fitness + 1e-10)

# Ant decision rule: choose the next node based on pheromone and heuristic
def choose_next_node(current_node, visited, pheromone_matrix, alpha, beta):
    probabilities = []
    for next_node in range(num_nodes):
        if next_node not in visited:
            pheromone = (pheromone_matrix[current_node][next_node] + 1e-10) ** alpha
```

```
            heuristic = heuristic_info(current_node, next_node) ** beta
            probabilities.append((next_node, pheromone * heuristic))
        else:
            probabilities.append((next_node, 0))

    total = sum(prob for _, prob in probabilities)
    probabilities = [(node, prob / total) if total > 0 else (node, 0) for node, prob in probabilities]
    chosen_node = random.choices([node for node, _ in probabilities], weights=[prob for _, prob in
probabilities])[0]
    return chosen_node

# Main ACO function
def ant_colony_optimization():
    best_path = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        # Generate paths for all ants
        ants = []
        for _ in range(num_ants):
            path = []
            visited = set()
            current_node = random.randint(0, num_nodes - 1)
            path.append(current_node)
            visited.add(current_node)

            while len(path) < num_nodes:
                next_node = choose_next_node(current_node, visited, pheromone_matrix, alpha, beta)
                path.append(next_node)
                visited.add(next_node)
                current_node = next_node

            ants.append(path)

        # Evaluate paths and update best solution
        fitnesses = [fitness_function(path) for path in ants]
        for path, fitness in zip(ants, fitnesses):
            if fitness < best_fitness:
                best_fitness = fitness
                best_path = path

        # Update pheromones
        update_pheromones(pheromone_matrix, ants, fitnesses)

        # Print iteration details
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")
```

```
    print(f"Best Path: {best_path}")
    print(f"Best Fitness: {best_fitness}")

# Run the ACO algorithm
ant_colony_optimization()
```

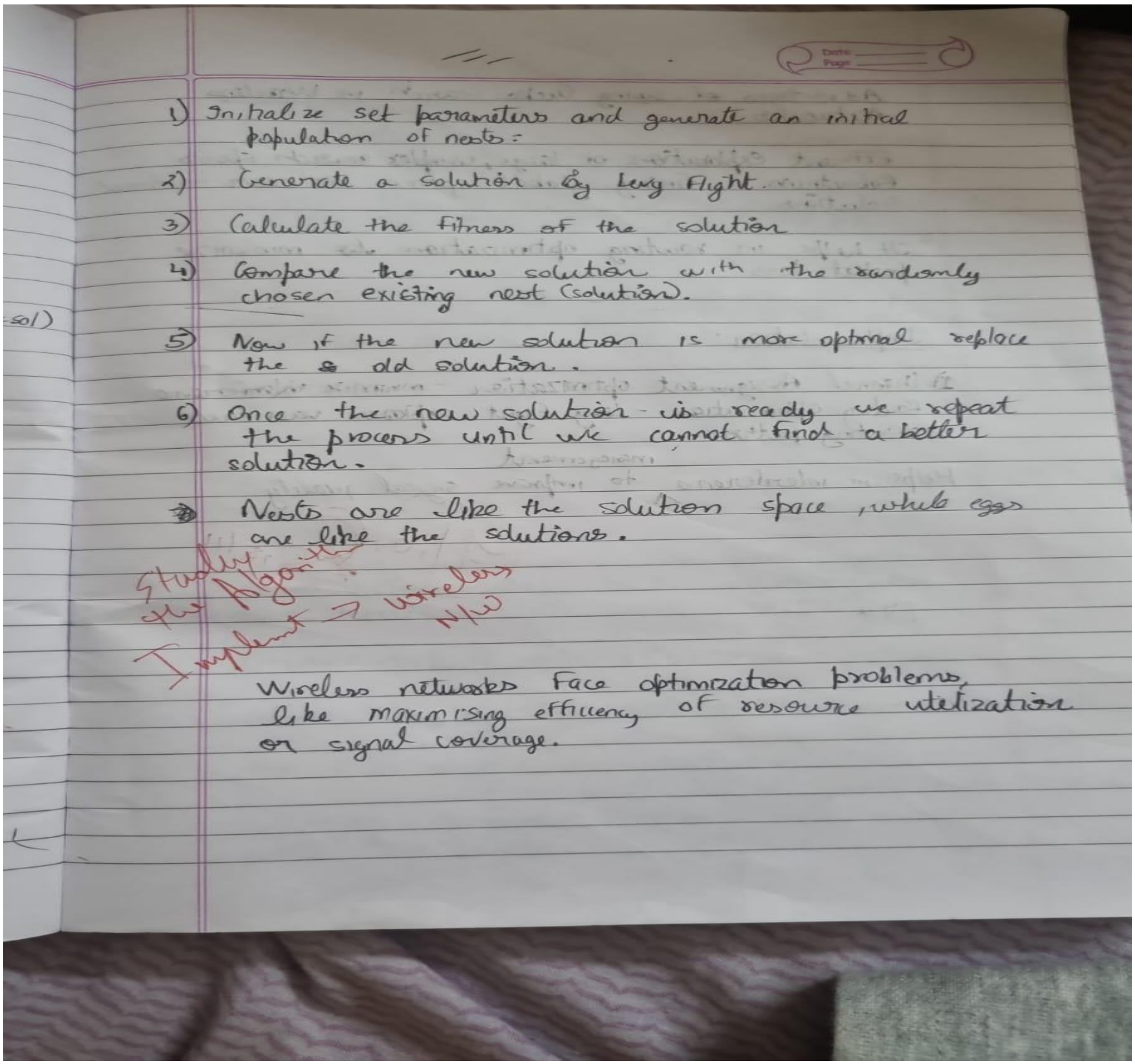
Output:

```
Iteration 99: Best Fitness = 19
Iteration 100: Best Fitness = 19
Best Path: [19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Best Fitness: 19
```

4. Cuckoo Search(CS)

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour 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:



1) Initialize set parameters and generate an initial population of nests:

2) Generate a solution by Levy Flight

3) Calculate the fitness of the solution

4) Compare the new solution with the randomly chosen existing nest (solution).

5) Now if the new solution is more optimal replace the ~~s~~ old solution.

6) Once the new solution is ready we repeat the process until we cannot find a better solution.

Nests are like the solution space, while eggs are like the solutions.

Study the Algorithm

Implement → wireless N/W

Wireless networks face optimization problems, like maximising efficiency of resource utilization or signal coverage.

Code:

```
import numpy as np

# Objective Function - To be customized according to the specific network problem
def objective_function(x):
    # Example: Optimizing base station locations and transmission power
    # Assuming x[0], x[1] are base station coordinates and x[2] is transmission power
    base_station_x = x[0]
    base_station_y = x[1]
    transmission_power = x[2]

    # Example: Calculate signal strength, coverage, or other parameters for the network
    # This is just a placeholder for the actual network performance evaluation
    coverage = (base_station_x ** 2 + base_station_y ** 2) ** 0.5  # Distance from origin
    efficiency = transmission_power / (1 + coverage)  # Just a sample efficiency calculation

    return -efficiency  # We are minimizing the negative of efficiency (to maximize efficiency)

# Levy Flight - Helps in the exploration of new solutions
def levy_flight(Lambda, dim):
    beta = 3 / 2
    sigma = ((gamma(1 + beta) * np.sin(np.pi * beta / 2)) /
             (gamma((1 + beta) / 2) * beta * 2**((beta - 1) / 2)))**(1 / beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.abs(v)**(1 / beta)
    return Lambda * step

# Cuckoo Search Algorithm
def cuckoo_search(objective_function, n_nests, max_iter, dim, lower_bound, upper_bound):
    # Step 1: Initialize the nests (solutions)
    nests = np.random.uniform(low=lower_bound, high=upper_bound, size=(n_nests, dim))
    fitness = np.array([objective_function(nest) for nest in nests])

    # Step 2: Find the best solution
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Step 3: Iteration (Search Process)
    for iter in range(max_iter):
        # Generate new solutions by Levy flight
        new_nests = nests + levy_flight(0.01, dim)

        # Apply boundary conditions (clamp to bounds)
        new_nests = np.clip(new_nests, lower_bound, upper_bound)
```

```
        # Evaluate the fitness of new solutions
        new_fitness = np.array([objective_function(nest) for nest in new_nests])

        # Find the best solution so far
        better_nests = new_fitness < fitness
        nests[better_nests] = new_nests[better_nests]
        fitness[better_nests] = new_fitness[better_nests]

        # Find the best solution overall
        current_best_fitness = np.min(fitness)
        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[np.argmin(fitness)]

        print(f"Iteration {iter+1}: Best Fitness = {best_fitness}")

    return best_nest, best_fitness

# Parameters
n_nests = 20            # Number of nests (solutions)
max_iter = 100          # Number of iterations
dim = 3                 # Dimension (e.g., x, y coordinates, and transmission power)
lower_bound = np.array([0, 0, 0])   # Lower bounds of the variables
upper_bound = np.array([100, 100, 10])  # Upper bounds of the variables

# Run Cuckoo Search
best_solution, best_score = cuckoo_search(objective_function, n_nests, max_iter, dim, lower_bound,
upper_bound)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness (Efficiency): {-best_score}")
```

Output:

```
Iteration 94: Best Fitness = -0.10737431217815656
Iteration 95: Best Fitness = -0.10737431217815656
Iteration 96: Best Fitness = -0.10737431217815656
Iteration 97: Best Fitness = -0.10737431217815656
Iteration 98: Best Fitness = -0.1073793356068069
Iteration 99: Best Fitness = -0.1073793356068069
Iteration 100: Best Fitness = -0.10738994879338912
Best Solution: [61.87286198 68.2464836  10.        ]
Best Fitness (Efficiency): 0.10738994879338912
```

5. Grey Wolf Optimiser:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour 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:

22/11/24

Grey Wolf Optimization

Initialization = start with a random group of solutions

Ranking : Identify the best solution (alpha), the second best (beta) and then delta, gamma the Followers.

keep iterating : keep iterating for the pre-defined num_of_iterations, and the final ranking will be reached.

After all the iterations are done the alpha will be the most optimal solution Followed by, beta, ~~gamma~~ delta, gamma.

The algorithm stops when it finds a sufficiently good solution or reaches the max iterations.

Implentation in Image processing + Disadv

```
num_wolves = 20
num_iterations = 50

Fitness = evaluate_Fitness (wolves)

alpha, beta, delta = ~~rank~~ (v. rank_wolves (wolves

For i in range (num_iterations)
        wolF = update_position (wolF, alpha, bet
   Fitness = evaluate_Fitness (wolF)
print( "Best solution is:", alpha)
```

Code:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Objective function: Otsu's Thresholding for Image Segmentation
def otsu_variance(threshold, histogram, total_pixels):
    background_weight = np.sum(histogram[:threshold])
    foreground_weight = np.sum(histogram[threshold:])
    if background_weight == 0 or foreground_weight == 0:
        return float('inf')  # Avoid division by zero
    background_mean = np.sum(np.arange(threshold) * histogram[:threshold]) / background_weight
    foreground_mean = np.sum(np.arange(threshold, 256) * histogram[threshold:]) / foreground_weight
    between_class_variance = background_weight * foreground_weight * (background_mean - 
foreground_mean) ** 2
    return -between_class_variance  # Minimize negative of variance

# Grey Wolf Optimizer
def grey_wolf_optimizer(histogram, total_pixels, max_iter=50, population_size=10):
    dim = 1  # Only optimizing threshold
    alpha_pos, beta_pos, delta_pos = None, None, None
    alpha_score, beta_score, delta_score = float('inf'), float('inf'), float('inf')
    wolves = np.random.randint(0, 256, (population_size, dim))

    a = 2  # Control parameter
    for iteration in range(max_iter):
        for i in range(population_size):
            fitness = otsu_variance(wolves[i][0], histogram, total_pixels)
            if fitness < alpha_score:
                alpha_score, beta_score, delta_score = fitness, alpha_score, beta_score
                alpha_pos, beta_pos, delta_pos = wolves[i], alpha_pos, beta_pos
            elif fitness < beta_score:
                beta_score, delta_score = fitness, beta_score
                beta_pos, delta_pos = wolves[i], beta_pos
            elif fitness < delta_score:
                delta_score = fitness
                delta_pos = wolves[i]

        # Update positions
        for i in range(population_size):
            for d in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
```

```
            D_alpha = abs(C1 * alpha_pos[d] - wolves[i][d])
            X1 = alpha_pos[d] - A1 * D_alpha

            r1, r2 = np.random.rand(), np.random.rand()
            A2, C2 = 2 * a * r1 - a, 2 * r2
            D_beta = abs(C2 * beta_pos[d] - wolves[i][d])
            X2 = beta_pos[d] - A2 * D_beta

            r1, r2 = np.random.rand(), np.random.rand()
            A3, C3 = 2 * a * r1 - a, 2 * r2
            D_delta = abs(C3 * delta_pos[d] - wolves[i][d])
            X3 = delta_pos[d] - A3 * D_delta

            wolves[i][d] = np.clip((X1 + X2 + X3) / 3, 0, 255)

        a -= 2 / max_iter  # Linearly decrease a

    return int(alpha_pos[0])  # Return optimal threshold

# Main function
if __name__ == "__main__":
    # Load and preprocess image
    img = cv2.imread("/content/design_resolution_original.jpg", 0)  # Grayscale image
    histogram, _ = np.histogram(img.ravel(), bins=256, range=(0, 256))
    total_pixels = img.size

    # Run GWO
    optimal_threshold = grey_wolf_optimizer(histogram, total_pixels)
    print("Optimal Threshold:", optimal_threshold)

    # Apply threshold
    _, segmented_img = cv2.threshold(img, optimal_threshold, 255, cv2.THRESH_BINARY)

    # Display results
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(img, cmap="gray")
    plt.subplot(1, 2, 2)
    plt.title("Segmented Image")
    plt.imshow(segmented_img, cmap="gray")
    plt.show()
```

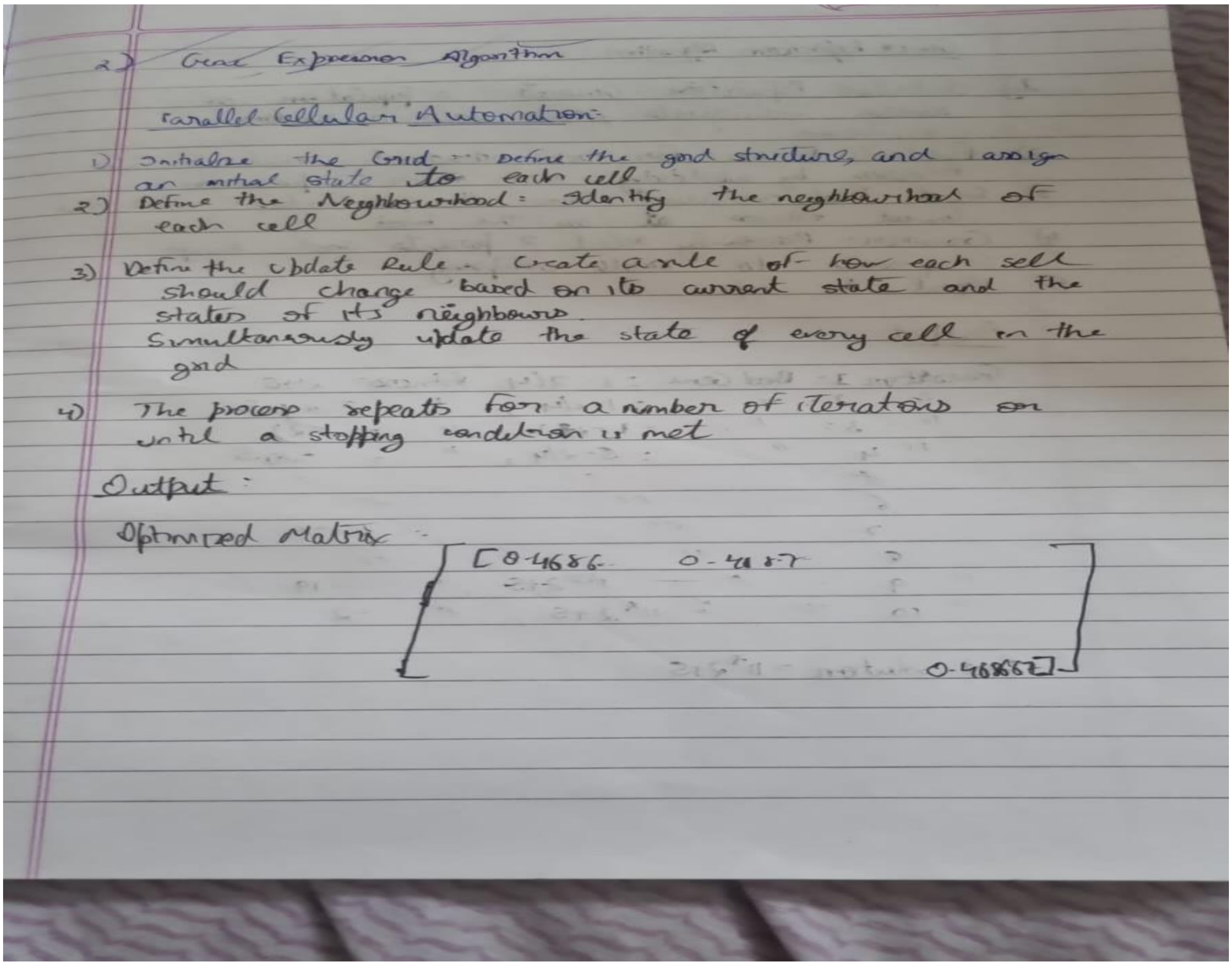
Output:

```
Optimal solution: [[0.0569084]]
Fitness of optimal solution: [1.11441792]
```

6. Parallel Cellular Algorithms and Programs:

The Parallel Cell Algorithm is a computational method used for solving problems that involve large datasets, spatial partitioning, or simulations where a domain is divided into smaller "cells" that can be processed independently or semi-independently in parallel. It is commonly applied in scientific computing, numerical simulations, and artificial intelligence, where computational efficiency is crucial.

Algorithm:



2) Gene Expression Algorithm

Parallel Cellular Automation:

1) Initialize the Grid - Define the grid structure, and assign an initial state to each cell.
2) Define the Neighbourhood: Identify the neighbourhood of each cell
3) Define the Update Rule - Create a rule of how each cell should change based on its current state and the states of its neighbours.
Simultaneously update the state of every cell in the grid
4) The process repeats for a number of iterations or until a stopping condition is met

Output:

Optimized Matrix:

[[0.4686 0.4187 ...

... 0.46866]]

Code:

```
import numpy as np
import multiprocessing

def fitness_function(matrix):
    return np.var(matrix)

def update_cell(i, j, matrix, size):
    neighbors = []
    for di in range(-1, 2):
        for dj in range(-1, 2):
            ni, nj = (i + di) % size, (j + dj) % size
            neighbors.append(matrix[ni, nj])
    return np.mean(neighbors)

def parallel_cell_update(matrix, size):
    pool =
multiprocessing.Pool(processes=multiprocessi
ng.cpu_count())
    result = []
    for i in range(size):
        for j in range(size):

result.append(pool.apply_async(update_cell,
(i, j, matrix, size)))
    pool.close()
    pool.join()

    new_matrix = np.array([r.get() for r in
result]).reshape(matrix.shape)
    return new_matrix

def cellular_optimization(matrix,
max_iter=10):
    size = matrix.shape[0]
    for t in range(max_iter):
        matrix = parallel_cell_update(matrix,
size)
        print(f"Iteration {t + 1}:
Matrix:\n{matrix}")
    return matrix

matrix = np.random.rand(5, 5)
optimized_matrix =
cellular_optimization(matrix)
print("Optimized Matrix:\n",
optimized_matrix)
```

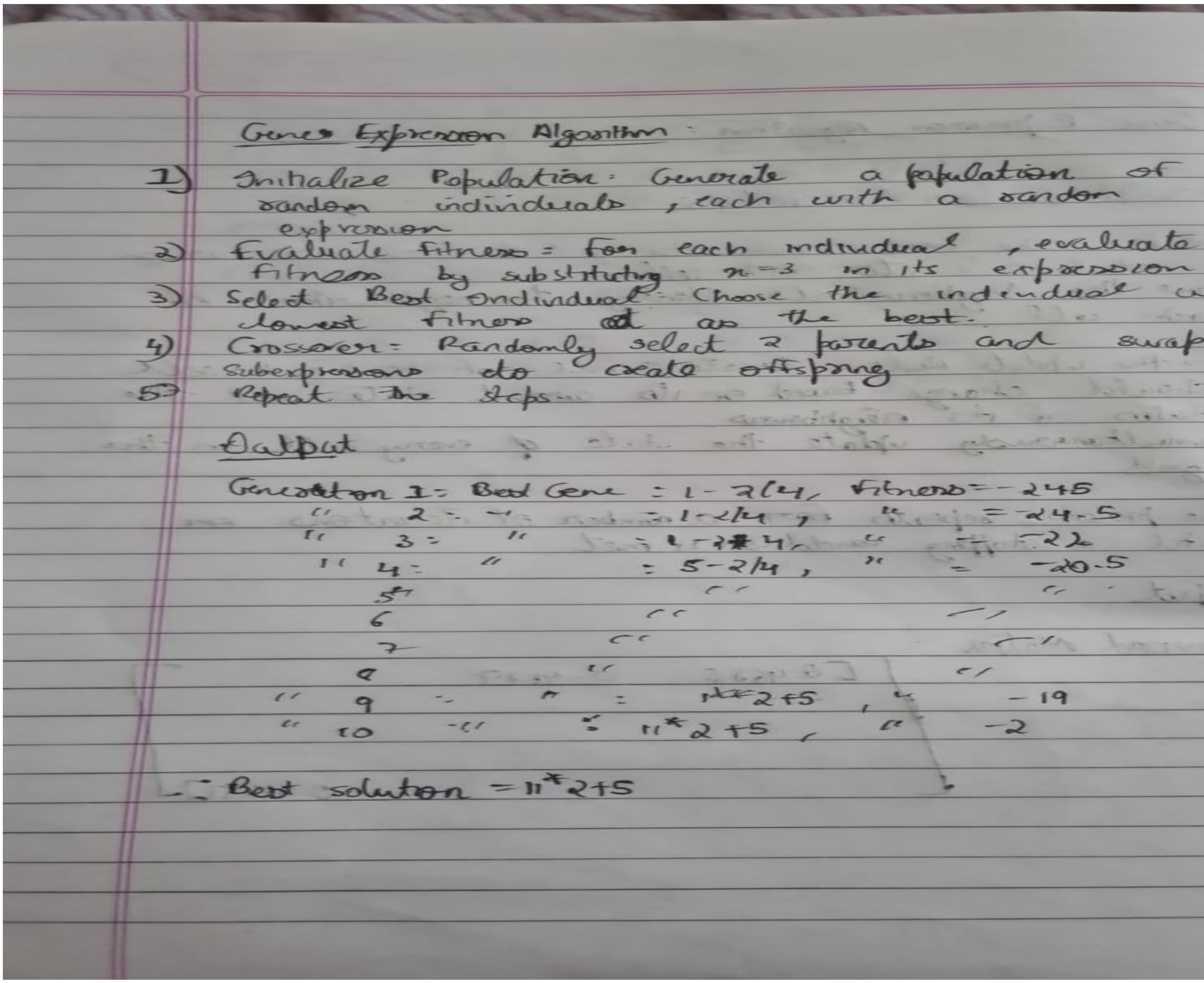
Output:

```
Optimized Matrix:
 [[0.46875618 0.46865932 0.46854258 0.46856729 0.46869931]
 [0.46865422 0.46855803 0.46844042 0.46846392 0.46859606]
 [0.46857991 0.46848458 0.46836626 0.46838846 0.46852051]
 [0.46863597 0.46854049 0.46842259 0.4684452  0.46857708]
 [0.46874493 0.46864851 0.46853158 0.46855574 0.4686876 ]]
```

7. Gene Expression Algorithms(GEA):

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:



Gene Expression Algorithm:

1) Initialize Population: Generate a population of random individuals, each with a random expression
2) Evaluate Fitness: For each individual, evaluate fitness by substituting $x=3$ in its expression
3) Select Best Individual: Choose the individual with lowest fitness as the best.
4) Crossover: Randomly select 2 parents and swap subexpressions to create offspring
5) Repeat the steps.

Output

Generation 1: Best Gene = 1-2/4, Fitness = -24.5
" 2: " = 1-2/4, " = -24.5
" 3: " = 1-2*4, " = -22
" 4: " = 5-2/4, " = -20.5
5 " "
6 " "
7 " "
8 " "
" 9: " = 11*2+5, " -19
" 10: " = 11*2+5, " -2

∴ Best solution = 11*2+5

Code:

```
import numpy as np

# Define the target function
def target_function(x):
    return x**2  # The function to optimize

# Define the fitness function
def fitness_function(expression, target, x_value):
    """
    Evaluate the fitness of an expression.
    :param expression: The candidate solution (as a string).
    :param target: Target output to achieve (e.g., x^2).
    :param x_value: The value of x to plug into the function.
    :return: Fitness value (higher is better).
    """
    try:
        result = eval(expression)  # Evaluate the expression
        return -abs(result - target_function(x_value))  # Closer to x^2, better the fitness
    except:
        return float('-inf')  # Invalid expressions get very low fitness

# Gene Expression Algorithm
class GeneExpressionAlgorithm:
    def __init__(self, population_size, gene_length, target, generations, mutation_rate, x_value):
        self.population_size = population_size
        self.gene_length = gene_length
        self.target = target
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.x_value = x_value
        self.operators = ['+', '-', '*', '/', '**']
        self.variables = ['x']
        self.constants = ['1', '2', '3', '4', '5']
        self.population = self._initialize_population()

    def _initialize_population(self):
        """
        Generate a random initial population.
        """
        population = []
        for _ in range(self.population_size):
            gene = ''.join(
```

```
                np.random.choice(self.variables + self.operators + self.constants, self.gene_length)
            )
            population.append(gene)
        return population

    def _mutate(self, gene):
        """
        Apply random mutation to a gene.
        """
        gene = list(gene)
        for i in range(len(gene)):
            if np.random.rand() < self.mutation_rate:
                gene[i] = np.random.choice(self.variables + self.operators + self.constants)
        return ''.join(gene)

    def evolve(self):
        """
        Evolve the population to optimize the function.
        """
        for generation in range(self.generations):
            # Evaluate fitness for each gene in the population
            fitness = [fitness_function(gene, self.target, self.x_value) for gene in self.population]

            # Select the best-performing genes
            sorted_indices = np.argsort(fitness)[::-1]  # Descending sort
            self.population = [self.population[i] for i in sorted_indices[:self.population_size // 2]]

            # Generate offspring by mutating the best genes
            offspring = [self._mutate(gene) for gene in self.population]
            self.population += offspring

            # Print the best gene of the generation
            print(f"Generation {generation + 1}: Best Gene = {self.population[0]}, Fitness =
{fitness[sorted_indices[0]]}")

        # Return the best solution
        best_gene = self.population[0]
        return best_gene

# Main Execution
if __name__ == "__main__":
    # Parameters
    population_size = 20
    gene_length = 5
    target = 25  # Target value of x^2 for x=5
    x_value = 5  # Use x = 5 for optimization
```

```
    generations = 10
    mutation_rate = 0.2

    # Initialize and run the algorithm
    gep = GeneExpressionAlgorithm(population_size, gene_length, target, generations, mutation_rate, x_value)
    best_solution = gep.evolve()
    print(f"Best Solution: {best_solution}")
```

Output:

```
Generation 1: Best Gene = 1-2/4, Fitness = -24.5
Generation 2: Best Gene = 1-2/4, Fitness = -24.5
Generation 3: Best Gene = 1-2+4, Fitness = -22
Generation 4: Best Gene = 5-2/4, Fitness = -20.5
Generation 5: Best Gene = 5-2/4, Fitness = -20.5
Generation 6: Best Gene = 5-2/4, Fitness = -20.5
Generation 7: Best Gene = 5-2/4, Fitness = -20.5
Generation 8: Best Gene = 5-2/4, Fitness = -20.5
Generation 9: Best Gene = 1**2+5, Fitness = -19
Generation 10: Best Gene = 11*2+5, Fitness = -2
Best Solution: 11*2+5
```