VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SOHAN T SANJEEV (1BM23CS421)

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
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

CERTIFICATE

This is to certify that the Lab work entitled "Bio Inspired Systems (23CS5BSBIS)" carried out by **SOHAN T SANJEEV (1BM23CS421),** 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.

Saritha AN Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	3-10-24	Implementation of Genetic Algorithms using Optimization	4-7
2	25-10-24	Implementation of Particle Swarm Optimization for Function Optimization	8-11
3	7-11-24	Implementation of Ant Colony Algorithm	12-15
4	14-11-24	Implementation of Cuckoo Search Algorithm	16-21
5	21-11-24	Implementation of Grey Wolf Search Optimization Algorithm	22-25
6	28-11-24	Implementation of Parallel Cellular Algorithms and Programs	26-30
7	18-12-24	Implementation of Gene Expression Algorithms (GEA)	31-36

GitHub Link:

https://github.com/SohanTbmsce/5E---BIS

Program 1

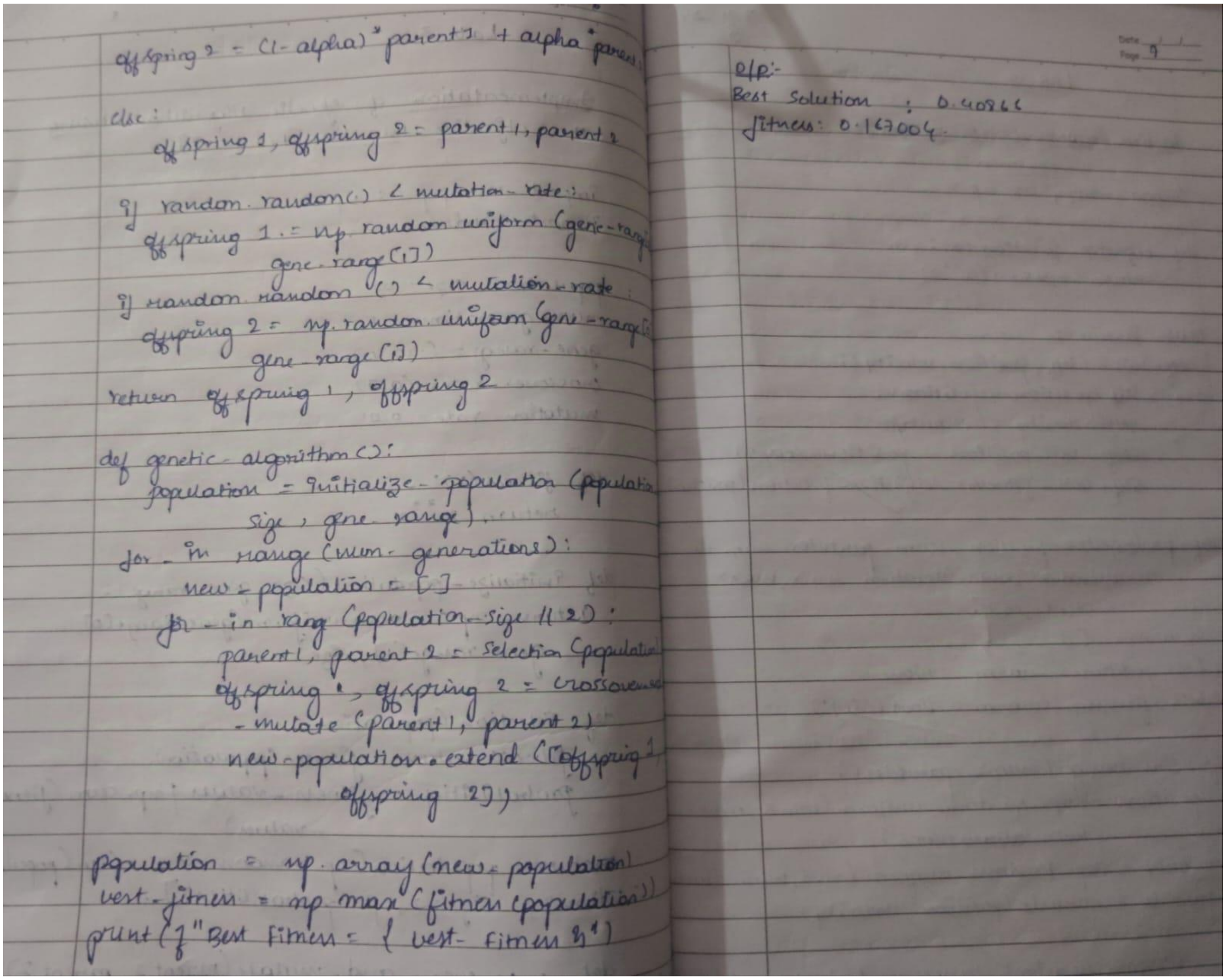
Implementation of Genetic Algorithms using Optimization

Algorithm:

Lab-3

Implementation of Genetic Algorithms using the objective Function [f(x) = x^2]

```
import numpy as np
import random

population_size = 100
num_generations = 50
gene_range = (-10, 10)
crossover_rate = (0.7)
mutation_rate = 0.01

def fitness(x):
    return x ** 2

def initialize_population(size, gene_range):
    return np.random.uniform(gene_range[0],
        gene_range[1], size)

def selection(population):
    fitness_values = fitness(population)
    probabilities = fitness_values / np.sum(fitness
                    _values)
return population[np.random.choice(len(popula
tion), size=2, p=probabilities)]

def crossover_and_mutate(parent1, parent2):
    if random.random() < crossover_rate:
        alpha = random.random()
        offspring1 = alpha * parent1 + (1-alpha)
                     * parent 2
```



offspring 2 = (1-alpha) * parent 1 + alpha * parent
else:
offspring 1, offspring 2 = parent 1, parent 2
if random.random() < mutation_rate:
offspring 1 = np.random.uniform(gene_range
gene_range[1])
if random.random() < mutation_rate
offspring 2 = np.random.uniform(gene_range[
gene_range[1])
return offspring 1, offspring 2

def genetic_algorithm():
population = initialize_population(population
size, gene_range).
for _ in range(num_generations):
new_population = []
for _ in rang(population_size // 2):
parent 1, parent 2 = selection(population
offspring 1, offspring 2 = crossover
_mutate(parent 1, parent 2)
new_population.extend([offspring 1,
offspring 2])

population = np.array(new_population)
best_fitness = np.max(fitness(population))
print(f"Best fitness = {best_fitness}")

o/p:-
Best Solution : 0.40866
fitness: 0.167004

Code:

```
#LAB-3:GENETIC ALGORITHM USING OPTIMIZATION
import numpy as np
import random

population_size = 50

num_generations = 50
gene_range = (-10, 10)
crossover_rate = 0.7
mutation_rate = 0.01


def fitness(x):
    return x ** 2
```

```python
def initialize_population(size, gene_range):
    return np.random.uniform(gene_range[0], gene_range[1], size)


def selection(population):
    fitness_values = fitness(population)
    probabilities = fitness_values / np.sum(fitness_values)
    return population[np.random.choice(len(population), size=2,
p=probabilities)]


def crossover_and_mutate(parent1, parent2):
    if random.random() < crossover_rate:
        alpha = random.random()
        offspring1 = alpha * parent1 + (1 - alpha) * parent2
        offspring2 = (1 - alpha) * parent1 + alpha * parent2
    else:
        offspring1, offspring2 = parent1, parent2

    if random.random() < mutation_rate:
        offspring1 = np.random.uniform(gene_range[0], gene_range[1])
    if random.random() < mutation_rate:
        offspring2 = np.random.uniform(gene_range[0], gene_range[1])

    return offspring1, offspring2

def genetic_algorithm():
    population = initialize_population(population_size, gene_range)

    for _ in range(num_generations):
        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = selection(population)
            offspring1, offspring2 = crossover_and_mutate(parent1, parent2)
            new_population.extend([offspring1, offspring2])

        population = np.array(new_population)
        best_fitness = np.max(fitness(population))
        print(f"Best Fitness = {best_fitness}")

    return population[np.argmax(fitness(population))]


best_solution = genetic_algorithm()print(f"The best solution found: x =
{best_solution}, f(x) = {fitness(best_solution)}")
```

Output:

```
Best Fitness = 96.41554825031078
Best Fitness = 96.41554825031078
Best Fitness = 93.77197105788775
Best Fitness = 93.77197105788775
Best Fitness = 89.75890201273582
Best Fitness = 89.75890201273582
Best Fitness = 89.75890201273582
Best Fitness = 86.28500889461726
Best Fitness = 85.46826382132502
Best Fitness = 85.46826382132502
Best Fitness = 83.92324509711497
Best Fitness = 82.56253766252124
Best Fitness = 81.54296033544576
Best Fitness = 80.93166255477712
Best Fitness = 80.93166255477712
Best Fitness = 98.63638625600349
Best Fitness = 95.71847971648907
Best Fitness = 95.71847971648907
Best Fitness = 78.67688660267774
Best Fitness = 78.42074885686151
Best Fitness = 78.40955357287604
Best Fitness = 78.2163163886333
Best Fitness = 78.2163163886333
Best Fitness = 77.87139603278503
Best Fitness = 77.83602785214866
Best Fitness = 77.73510449497564
Best Fitness = 77.5393503439129
Best Fitness = 80.56823420832076
Best Fitness = 77.46782583009356
Best Fitness = 77.46782583009356
Best Fitness = 77.46782583009356
Best Fitness = 77.4087588279005
Best Fitness = 77.38033782912058
Best Fitness = 77.38033782912058
Best Fitness = 77.37443058909962
Best Fitness = 77.36238137273801
Best Fitness = 77.35009480792148
Best Fitness = 77.33394495335988
Best Fitness = 77.33246445407919
Best Fitness = 77.33246445407919
Best Fitness = 77.32984862750179
Best Fitness = 77.30828605125325
Best Fitness = 77.26925723673783
Best Fitness = 77.17293761811123
Best Fitness = 77.13725577017766
Best Fitness = 77.13725577017766
Best Fitness = 77.1084152222153
Best Fitness = 77.1084152222153
Best Fitness = 76.9472934808921
Best Fitness = 76.93707640654297
The best solution found: x = 8.771378250112292, f(x) = 76.93707640654297
```

Program 2

Implementation of Particle Swarm Optimization for Function Optimization

Algorithm:

Page 2 10/24

LAB - 2

Particle Swarm optimization for Function Optimization

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

Applications of PSO:

Particle Swarm optimization has a wide range of Application across various fields, including:-

1) Engineering Design: optimization of structural designs, component layouts and resource allocation.
2) Artificial Intelligence: Feature selection, training of neural networks and optimization of algorithms
3) Robotics: path planning and multi-robot coordination
4) Finance: Portfolio optimization and risk assessment
5) Image processing: Edge detection and Image segmentation

Telecommunication: Network design and frequency allocation.

Healthcare: Intelligent diagnosis, disease detection and classification, Medical image segmentation

Page 3

Algorithm steps:

Step 1:- Define the problem (Mathematical Function):

→ Identify a mathematical function $f(x)$ that you want to optimize (eg, minimize or maximize)

→ common choice include functions like Rastrigin function or the sphere function

Step 2: Initialize parameters:

→ Set parameters such as:

(•) Number of particles 'n'

(•) Inertia weight w (controls exploration vs exploitation)

(•) Cognitive coefficient c_1 (influences how much a particle is attracted to its own best position).

(•) Social coefficient c_2 (influences how much a particle is attracted to the global best position)

Step 3:- Initialize particles:

(•) generate an initial population of particles with random positions and velocities in the solution space

Step 4:- Evaluate fitness:- calculate the fitness of each particles with random position and velocities in the solution space

Step 5:- Update Velocities and position:

→ update the velocity v_i and position x_i of each particle using the formula: $v_i = w \cdot v_i + c_1 \cdot r_1 \cdot (pbest_i - x_i) + c_2 \cdot r_2 (gbest - x_i)$ $x_i = x_i + v_i$

→ Here, pbest i is the personal best solution of particle i, gbest is the global best position among all

Date
Page 4

particles and r_1 are random numbers betw[een]
0 and 1.

6. Iterate: Repeat the evaluation, updating [and] position adjustment for a set of number ite[rations] or until a convergence criterion is met (e.g. [no] significant improvement in the best solution).

7. O/P the best solution:
→ After completing the iteration, output the best solution found, including its position and fitness value.

3/10/24

Code:

```
#LAB-4:PARTICLE SWARM OPTIMIZATION
import numpy as np

# Objective function (example: Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Particle class
class Particle:
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity
        self.best_position = position.copy()
        self.best_fitness = objective_function(position)

# Particle Swarm Optimization (PSO) function
def pso(objective_function, num_particles, num_dimensions, max_iter, minx,
maxx, w, c1, c2):
    # Initialize the swarm
    swarm = []
    best_position_swarm = None
    best_fitness_swarm = float('inf')

    # Create particles
    for _ in range(num_particles):
        position = np.random.uniform(minx, maxx, num_dimensions)
        velocity = np.random.uniform(-1, 1, num_dimensions)
        particle = Particle(position, velocity)

        # Update global best if this particle has a better fitness
        if particle.best_fitness < best_fitness_swarm:
            best_fitness_swarm = particle.best_fitness
            best_position_swarm = particle.best_position.copy()

        swarm.append(particle)

    # PSO main loop
    for iter in range(max_iter):
        for particle in swarm:
            # Calculate random factors
            r1, r2 = np.random.rand(), np.random.rand()

            # Update velocity
            inertia = w * particle.velocity
```

```
            cognitive = c1 * r1 * (particle.best_position - particle.position)
            social = c2 * r2 * (best_position_swarm - particle.position)
            particle.velocity = inertia + cognitive + social

            # Update position
            particle.position += particle.velocity

            # Apply position boundaries
            particle.position = np.clip(particle.position, minx, maxx)

            # Evaluate fitness of the new position
            fitness = objective_function(particle.position)

            # Update the particle's best fitness and position if needed
            if fitness < particle.best_fitness:
                particle.best_fitness = fitness
                particle.best_position = particle.position.copy()

            # Update global best fitness and position if needed
            if fitness < best_fitness_swarm:
                best_fitness_swarm = fitness
                best_position_swarm = particle.position.copy()

    # Return the best particle of the swarm
    return best_position_swarm, best_fitness_swarm

# PSO Hyperparameters
num_particles = 30
num_dimensions = 2
max_iter = 100
minx, maxx = -10, 10
w = 0.5      # Inertia weight
c1 = 1.5     # Cognitive (particle's own best) weight
c2 = 1.5     # Social (swarm's best) weight

# Run PSO
best_position, best_fitness = pso(objective_function, num_particles,
num_dimensions, max_iter, minx, maxx, w, c1, c2)

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

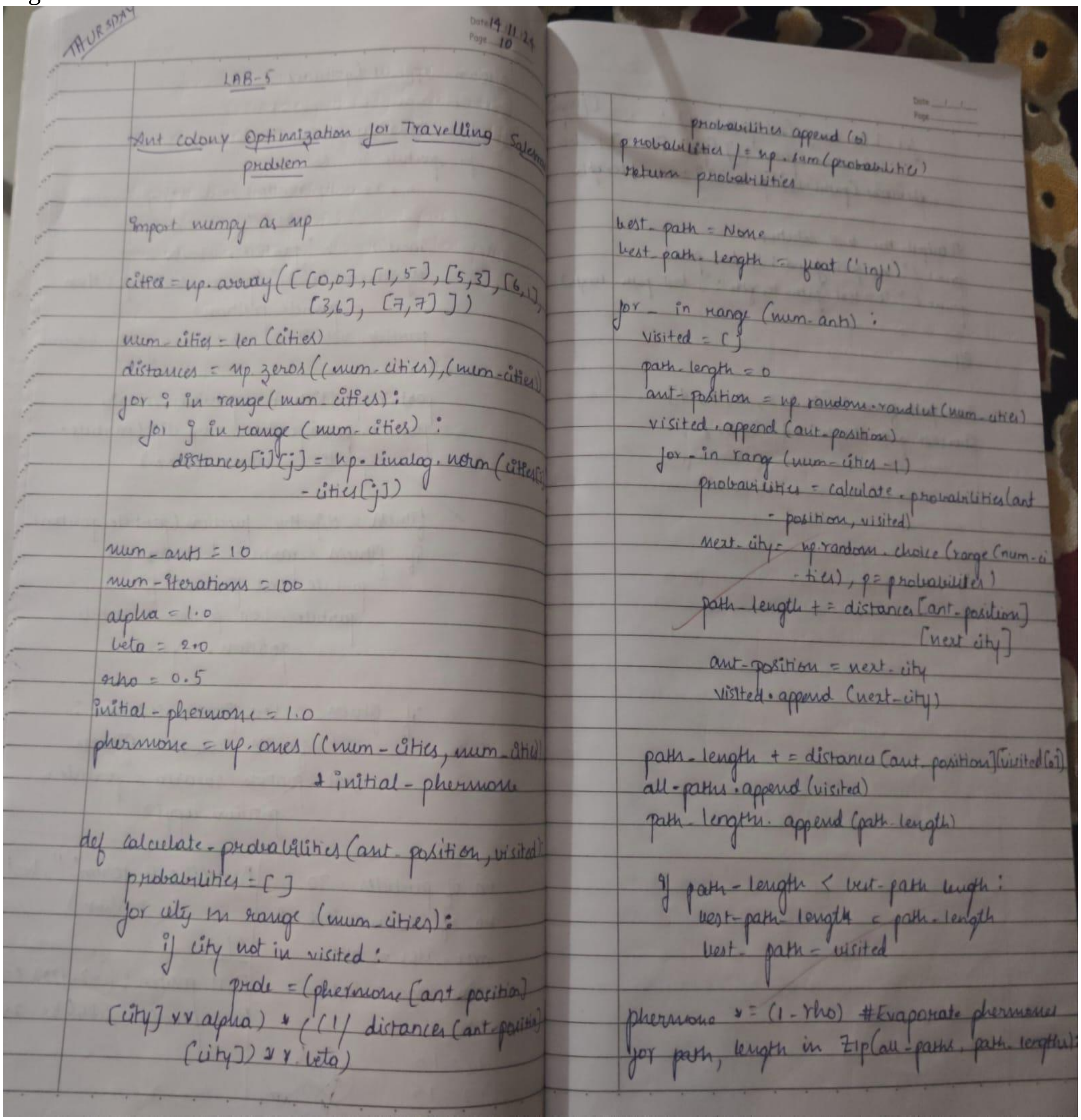
Output:

```
Best Position: [-7.11248266e-13 -6.05931834e-13]BestFitnesss8.730274834800181e
```

Program 3

Implementation of Ant Colony Algorithm

Algorithm:



THURSDAY

Date 14.11.24
Page 10

LAB-5

Ant colony Optimization for Travelling Salesman problem

```
import numpy as np

cities = np.array([[0,0],[1,5],[5,3],[6,1],
                   [3,6], [7,7]])
num_cities = len(cities)
distances = np.zeros((num_cities),(num_cities))
for i in range(num_cities):
    for j in range(num_cities):
        distances[i][j] = np.linalg.norm(cities[i]
                          - cities[j])

num_ants = 10
num_iterations = 100
alpha = 1.0
beta = 2.0
rho = 0.5
initial_pheromone = 1.0
pheromone = np.ones((num_cities, num_cities))
                    * initial_pheromone

def calculate_probabilities(ant_position, visited):
    probabilities = []
    for city in range(num_cities):
        if city not in visited:
            prob = (pheromone[ant_position]
    [city] ** alpha) * ((1/ distances[ant_position]
        [city]) ** beta)
            probabilities.append(0)
    probabilities /= np.sum(probabilities)
    return probabilities

best_path = None
best_path_length = float('inf')

for _ in range(num_ants):
    visited = []
    path_length = 0
    ant_position = np.random.randint(num_cities)
    visited.append(ant_position)
    for _ in range(num_cities - 1)
        probabilities = calculate_probabilities(ant
                          _position, visited)
        next_city = np.random.choice(range(num_ci
                      ties), p=probabilities)
        path_length += distances[ant_position]
                                [next_city]
        ant_position = next_city
        visited.append(next_city)

    path_length += distances[ant_position][visited[0]]
    all_paths.append(visited)
    path_lengths.append(path_length)

    if path_length < best_path_length:
        best_path_length = path_length
        best_path = visited

pheromone *= (1 - rho) #Evaporate pheromones
for path, length in zip(all_paths, path_lengths):
```

Date / /
Page

```
for i in range(len(path) -1) :
        phermone[path[i]][path[i+1]] += 1.
phermone[path[-1]][path[0]] += 1.0 / length

# output the best solution
print (" best path found : ", best_path)
print (" shortest path length :", best_path_length
```

o/p:

best_path found : [5, 2, 3, 0, 1, 4]
Shortest path length : 24.2491

14.11

Code:

```
 #LAB-5:ANT COLONY ORGANIZATION

import numpy as np

cities = np.array([
    [0, 0], [1, 5], [5, 3], [6, 1], [3, 6], [7, 7]
])

num_cities = len(cities)
distances = np.zeros((num_cities, num_cities))
for i in range(num_cities):
    for j in range(num_cities):
        distances[i][j] = np.linalg.norm(cities[i] - cities[j])

num_ants = 10
num_iterations = 100
alpha = 1.0
beta = 2.0
rho = 0.5
initial_pheromone = 1.0
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

def calculate_probabilities(ant_position, visited):
    probabilities = []
    for city in range(num_cities):
        if city not in visited:
            prob = (pheromone[ant_position][city] ** alpha) * ((1 /
distances[ant_position][city]) ** beta)
            probabilities.append(prob)
        else:
            probabilities.append(0)
    probabilities /= np.sum(probabilities)
    return probabilities

best_path = None
best_path_length = float('inf')

for _ in range(num_iterations):
    all_paths = []
    path_lengths = []

    for ant in range(num_ants):
        visited = []
```

```
        path_length = 0
        ant_position = np.random.randint(num_cities)  # Start at a random city
        visited.append(ant_position)

        for _ in range(num_cities - 1):
            probabilities = calculate_probabilities(ant_position, visited)
            next_city = np.random.choice(range(num_cities), p=probabilities)
            path_length += distances[ant_position][next_city]
            ant_position = next_city
            visited.append(next_city)

        path_length += distances[ant_position][visited[0]]  # Return to start
        all_paths.append(visited)
        path_lengths.append(path_length)

        # Update best path
        if path_length < best_path_length:
            best_path_length = path_length
            best_path = visited


    pheromone *= (1 - rho)
    for path, length in zip(all_paths, path_lengths):
        for i in range(len(path) - 1):
            pheromone[path[i]][path[i+1]] += 1.0 / length
        pheromone[path[-1]][path[0]] += 1.0 / length  # Complete the cycle


print("Best path found:", best_path)
print("Shortest path length:", best_path_length)
```

Output:

```
Best path found: [5, 2, 3, 0, 1, 4]
Shortest path length: 24.249159579507822
```

Program 4

Implementation of Cuckoo Search Algorithm

Algorithm:

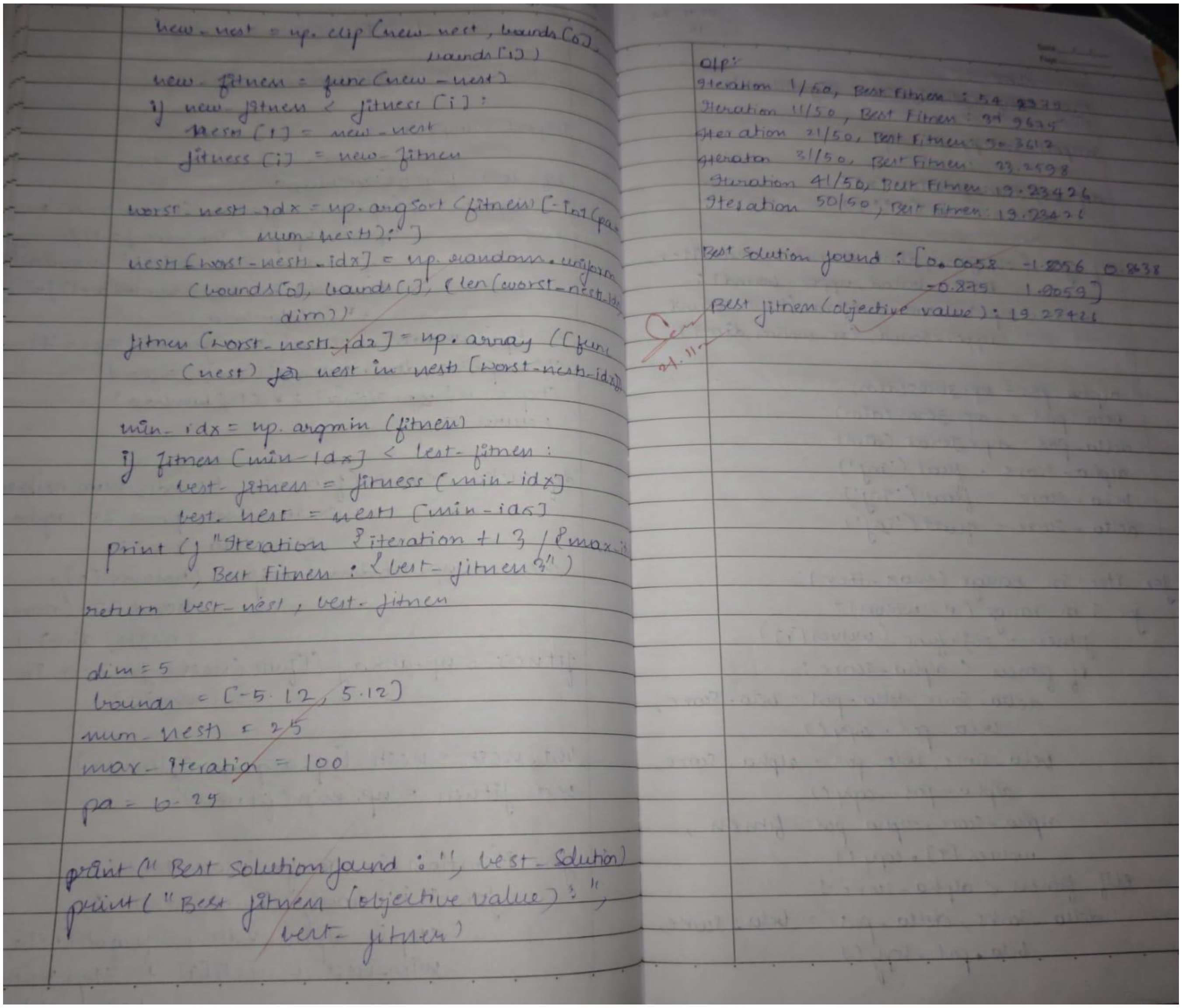
THURSDAY

Date 21/11/24
Page 13

LAB-6

Cuckoo Search Algorithm

```
import numpy as np

def levy_flight (lambda, d):
    sigma_u = (np.math.gamma (1+ lambda) *
              np.sin (np.pi * lambda / 2) /
              np.math.gamma ((1+ lambda) / 2)
              * lambda * 2 ** ((lambda - 1) / 2)))
              ** (1 / lambda)
    u = np.random.normal (0, sigma_u, size = d)
    v = np.random.normal (0, 1, size = d)
    step = u / np.abs(v) ** (1 / lambda)
    return step

def cuckoo_search (fun, dim, bounds, num_nests=25,
                   max_iteration, pa=0.25, alpha
                   = 0.01, beta = 1.5):
    nests = np.random.uniform (bounds [0],
                               bounds[1], (num_
                               nests, dim))
    fitness = np.array ([func (nest) for nest in
                         nests])

    best_nest = nests [np.argmin (fitness)]
    best_fitness = np.min (fitness)

    for iteration in range (max_iter):
        for i in range (num_nests):
            step = alpha * levy_flight (beta, dim)
            num_nest = nests[i] + step * (nests[i]
                       - best_nest)
```



new_nest = np.clip(new_nest, bounds[0], bounds[1])
new_fitness = func(new_nest)
if new_fitness < fitness[i]:
nests[i] = new_nest
fitness[i] = new_fitness

worst_nest_idx = np.argsort(fitness)[-int(pa * num_nests):]
nests[worst_nest_idx] = np.random.uniform(bounds[0], bounds[1], (len(worst_nest_idx), dim))
fitness[worst_nest_idx] = np.array([func(nest) for nest in nests[worst_nest_idx]])

min_idx = np.argmin(fitness)
if fitness[min_idx] < best_fitness:
best_fitness = fitness[min_idx]
best_nest = nests[min_idx]
print(f"Iteration {iteration+1}/{max_iter}, Best Fitness: {best_fitness}")
return best_nest, best_fitness

dim = 5
bounds = [-5.12, 5.12]
num_nests = 25
max_iteration = 100
pa = 0.25

print("Best Solution found:", best_solution)
print("Best fitness (objective value):", best_fitness)

O/P:
Iteration 1/50, Best Fitness: 54.2279
Iteration 11/50, Best Fitness: 39.9635
Iteration 21/50, Best Fitness: 30.3612
Iteration 31/50, Best Fitness: 23.2598
Iteration 41/50, Best Fitness: 19.23426
Iteration 50/50, Best Fitness: 19.23426

Best Solution found: [0.0058 -1.8056 0.8638 -0.875 1.9059]
Best fitness (objective value): 19.23426

Code:

```
#LAB-6:CUCKOO SEARCH ALGORITHM
import numpy as np

# Define the Rastrigin function (used for optimization problems)
def rastrigin(x):
    A = 10
    return A * len(x) + sum(x**2 - A * np.cos(2 * np.pi * x))

# Levy Flight function
def levy_flight(Lambda, d):
    # Generate Lévy flights
```

```
    sigma_u = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (np.math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) /
2)))**(1 / Lambda)
    u = np.random.normal(0, sigma_u, size=d)
    v = np.random.normal(0, 1, size=d)
    step = u / np.abs(v)**(1 / Lambda)
    return step

# Cuckoo Search Algorithm
def cuckoo_search(func, dim, bounds, num_nests=25, max_iter=100, pa=0.25,
alpha=0.01, beta=1.5):
    # Initial population (random solutions within bounds)
    nests = np.random.uniform(bounds[0], bounds[1], (num_nests, dim))
    fitness = np.array([func(nest) for nest in nests])

    # Best nest found so far
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Main loop
    for iteration in range(max_iter):
        # Generate new solutions using Lévy flights
        for i in range(num_nests):
            step = alpha * levy_flight(beta, dim)
            new_nest = nests[i] + step * (nests[i] - best_nest)

            # Bound checking (to ensure new nests are within bounds)
            new_nest = np.clip(new_nest, bounds[0], bounds[1])

            # Calculate fitness of the new nest
            new_fitness = func(new_nest)

            # If new solution is better, replace old nest
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon the worst nests and replace them with new random ones
        worst_nests_idx = np.argsort(fitness)[-int(pa * num_nests):]
        nests[worst_nests_idx] = np.random.uniform(bounds[0], bounds[1],
(len(worst_nests_idx), dim))
        fitness[worst_nests_idx] = np.array([func(nest) for nest in
nests[worst_nests_idx]])

        # Update the best nest
```

```
        min_idx = np.argmin(fitness)
        if fitness[min_idx] < best_fitness:
            best_fitness = fitness[min_idx]
            best_nest = nests[min_idx]

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

    return best_nest, best_fitness

# Define problem bounds and parameters
dim = 5  # Problem dimensionality (e.g., 5-dimensional problem)
bounds = [-5.12, 5.12]  # Bounds of the search space (for Rastrigin function)
num_nests = 25  # Number of nests (solutions)
max_iter = 100  # Maximum number of iterations
pa = 0.25  # Probability of discovering a nest (fraction of worst nests to
abandon)

# Run the Cuckoo Search algorithm
best_solution, best_fitness = cuckoo_search(rastrigin, dim, bounds, num_nests,
max_iter, pa)

print("Best solution found: ", best_solution)
print("Best fitness (objective value): ", best_fitness)
```

Output:

```
<ipython-input-1-48ab2d5bdd0a>:11: DeprecationWarning: `np.math` is a deprecated alias for the standard library `math`
module (Deprecated Numpy 1.25). Replace usages of `np.math` with `math`
  sigma_u = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
<ipython-input-1-48ab2d5bdd0a>:12: DeprecationWarning: `np.math` is a deprecated alias for the standard library `math`
module (Deprecated Numpy 1.25). Replace usages of `np.math` with `math`
  (np.math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
Iteration 1/100, Best Fitness: 34.10283829263206
Iteration 2/100, Best Fitness: 34.10283829263206
Iteration 3/100, Best Fitness: 34.10283829263206
Iteration 4/100, Best Fitness: 34.10283829263206
Iteration 5/100, Best Fitness: 34.10283829263206
Iteration 6/100, Best Fitness: 34.10283829263206
Iteration 7/100, Best Fitness: 34.10283829263206
Iteration 8/100, Best Fitness: 34.10283829263206
Iteration 9/100, Best Fitness: 34.10283829263206
Iteration 10/100, Best Fitness: 34.10283829263206
Iteration 11/100, Best Fitness: 34.10283829263206
Iteration 12/100, Best Fitness: 34.10283829263206
Iteration 13/100, Best Fitness: 34.10283829263206
Iteration 14/100, Best Fitness: 34.10283829263206
Iteration 15/100, Best Fitness: 34.10283829263206
Iteration 16/100, Best Fitness: 34.10283829263206
```

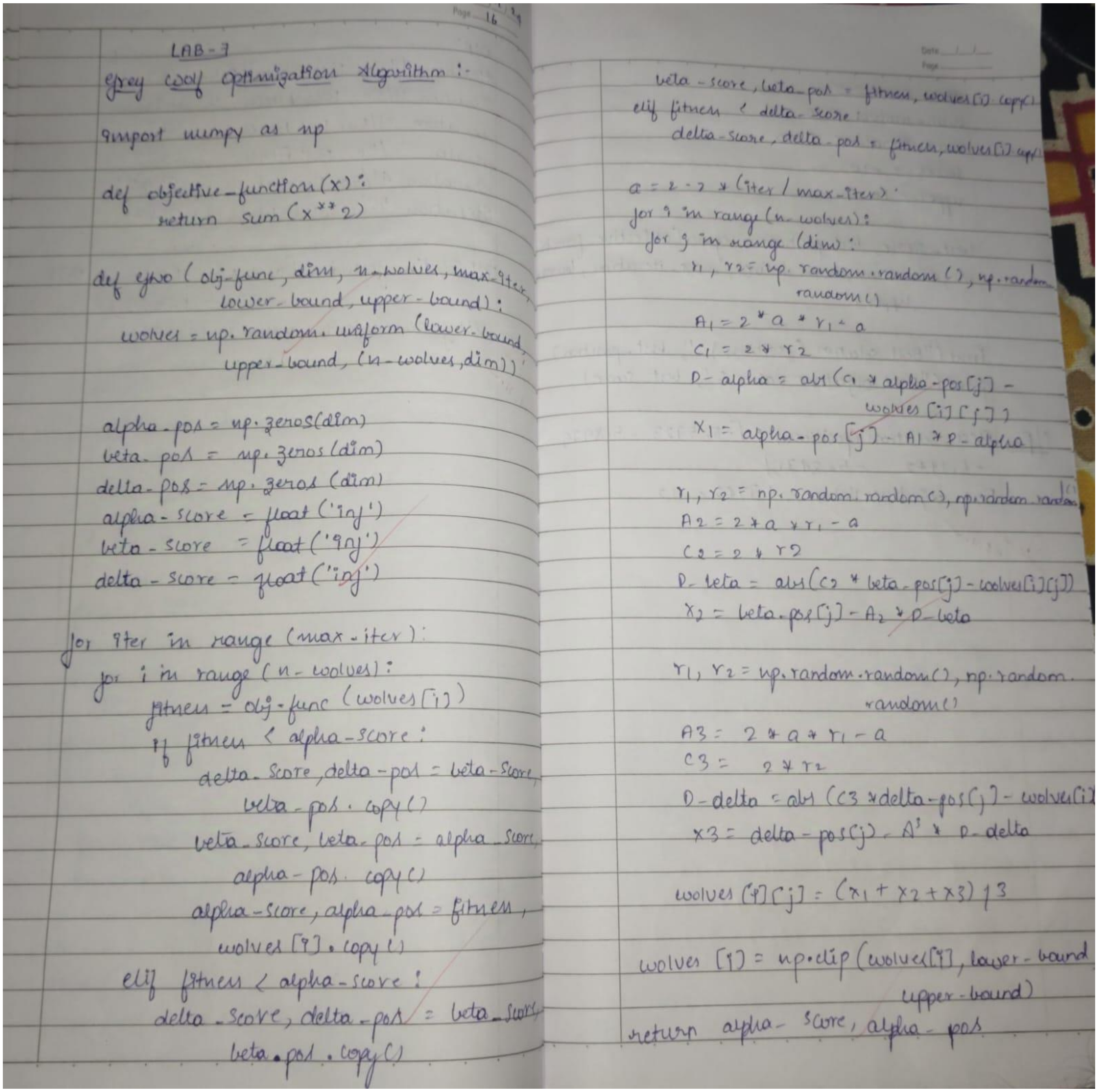
```
Iteration 17/100, Best Fitness: 34.10283829263206
Iteration 18/100, Best Fitness: 34.10283829263206
Iteration 19/100, Best Fitness: 34.10283829263206
Iteration 20/100, Best Fitness: 34.10283829263206
Iteration 21/100, Best Fitness: 34.10283829263206
Iteration 22/100, Best Fitness: 34.10283829263206
Iteration 23/100, Best Fitness: 34.10283829263206
Iteration 24/100, Best Fitness: 34.10283829263206
Iteration 25/100, Best Fitness: 34.10283829263206
Iteration 26/100, Best Fitness: 29.34980262200628
Iteration 27/100, Best Fitness: 29.34980262200628
Iteration 28/100, Best Fitness: 29.34980262200628
Iteration 29/100, Best Fitness: 23.133784372306057
Iteration 30/100, Best Fitness: 23.133784372306057
Iteration 31/100, Best Fitness: 23.133784372306057
Iteration 32/100, Best Fitness: 23.133784372306057
Iteration 33/100, Best Fitness: 23.133784372306057
Iteration 34/100, Best Fitness: 23.133784372306057
Iteration 35/100, Best Fitness: 23.133784372306057
Iteration 36/100, Best Fitness: 23.133784372306057
Iteration 37/100, Best Fitness: 23.133784372306057
Iteration 38/100, Best Fitness: 22.800741517549888
Iteration 39/100, Best Fitness: 22.800741517549888
Iteration 40/100, Best Fitness: 22.800741517549888
Iteration 41/100, Best Fitness: 22.800741517549888
Iteration 42/100, Best Fitness: 22.800741517549888
Iteration 43/100, Best Fitness: 21.978739473969867
Iteration 44/100, Best Fitness: 21.331310300542093
Iteration 45/100, Best Fitness: 21.134160680988344
Iteration 46/100, Best Fitness: 21.134160680988344
Iteration 47/100, Best Fitness: 21.112031250864927
Iteration 48/100, Best Fitness: 20.22783836901366
Iteration 49/100, Best Fitness: 20.22783836901366
Iteration 50/100, Best Fitness: 20.165109724434036
Iteration 51/100, Best Fitness: 20.165109724434036
Iteration 52/100, Best Fitness: 18.992313289468903
Iteration 53/100, Best Fitness: 18.992313289468903
Iteration 54/100, Best Fitness: 18.992313289468903
Iteration 55/100, Best Fitness: 18.992313289468903
Iteration 56/100, Best Fitness: 18.992313289468903
Iteration 57/100, Best Fitness: 18.992313289468903
Iteration 58/100, Best Fitness: 18.992313289468903
Iteration 59/100, Best Fitness: 16.345713901046942
Iteration 60/100, Best Fitness: 16.345713901046942
Iteration 61/100, Best Fitness: 16.345713901046942
Iteration 62/100, Best Fitness: 16.345713901046942
Iteration 63/100, Best Fitness: 16.345713901046942
Iteration 64/100, Best Fitness: 16.345713901046942
Iteration 65/100, Best Fitness: 16.345713901046942
Iteration 66/100, Best Fitness: 16.345713901046942
Iteration 67/100, Best Fitness: 16.345713901046942
Iteration 68/100, Best Fitness: 16.345713901046942
Iteration 69/100, Best Fitness: 16.345713901046942
Iteration 70/100, Best Fitness: 16.345713901046942
Iteration 71/100, Best Fitness: 16.126755706184014
Iteration 72/100, Best Fitness: 15.66247782758252
```

```
Iteration 73/100, Best Fitness: 15.66247782758252
Iteration 74/100, Best Fitness: 15.213018385737115
Iteration 75/100, Best Fitness: 15.213018385737115
Iteration 76/100, Best Fitness: 15.213018385737115
Iteration 77/100, Best Fitness: 14.351369913536452
Iteration 78/100, Best Fitness: 14.16685937126772
Iteration 79/100, Best Fitness: 14.16685937126772
Iteration 80/100, Best Fitness: 14.16685937126772
Iteration 81/100, Best Fitness: 14.16685937126772
Iteration 82/100, Best Fitness: 13.214801041894482
Iteration 83/100, Best Fitness: 13.214801041894482
Iteration 84/100, Best Fitness: 13.214801041894482
Iteration 85/100, Best Fitness: 13.214801041894482
Iteration 86/100, Best Fitness: 13.214801041894482
Iteration 87/100, Best Fitness: 12.893913779918272
Iteration 88/100, Best Fitness: 12.893913779918272
Iteration 89/100, Best Fitness: 12.893913779918272
Iteration 90/100, Best Fitness: 12.893913779918272
Iteration 91/100, Best Fitness: 12.893913779918272
Iteration 92/100, Best Fitness: 12.893913779918272
Iteration 93/100, Best Fitness: 12.705031336984696
Iteration 94/100, Best Fitness: 12.705031336984696
Iteration 95/100, Best Fitness: 12.705031336984696
Iteration 96/100, Best Fitness: 12.705031336984696
Iteration 97/100, Best Fitness: 12.705031336984696
Iteration 98/100, Best Fitness: 12.705031336984696
Iteration 99/100, Best Fitness: 12.705031336984696
Iteration 100/100, Best Fitness: 12.705031336984696
Best solution found:  [-1.92879105  1.96608788 -1.02863024 -0.10117856  0.06084245]
Best fitness (objective value):  12.705031336984696
```

Program 5

Implementation of Grey Wolf Search Optimization Algorithm

Algorithm:



Page 16

LAB-7

Grey wolf optimization Algorithm :-

```
import numpy as np

def objective_function(x):
        return sum(x**2)

def gwo(obj_func, dim, n_wolves, max_iter,
             lower_bound, upper_bound):
    wolves = np.random.uniform(lower_bound,
             upper_bound, (n_wolves, dim))

alpha_pos = np.zeros(dim)
beta_pos = np.zeros(dim)
delta_pos = np.zeros(dim)
alpha_score = float('inf')
beta_score = float('inf')
delta_score = float('inf')

for iter in range(max_iter):
    for i in range(n_wolves):
        fitness = obj_func(wolves[i])
        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 < alpha_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 - 2 * (iter / max_iter)
    for i in range(n_wolves):
      for j in range(dim):
          r1, r2 = np.random.random(), np.random.
                        random()
          A1 = 2 * a * r1 - a
          C1 = 2 * r2
          D_alpha = abs(C1 * alpha_pos[j] -
                        wolves[i][j])
          X1 = alpha_pos[j] - A1 * D_alpha

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

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

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

      wolves[i] = np.clip(wolves[i], lower_bound
                        upper_bound)
return alpha_score, alpha_pos
```

Date / /
Page

dimension = 5
num-wolves = 10
max-iterations = 100
lower = -10
upper = 10

best-score, best-position = gwo(objective-function, dimension, num-wolves, max-iterations, lower, upper)

Print("Best solution (position):", best-position)
print("Best objective value :", best-score)

O/P:- Best Solution (position) : [-5.5723 -5.8929 -6.1442 -5.8431]
Best objective value : 1.7742e-12

28.11.

Code:

```
import numpy as np

# Define the function to optimize (objective function)
def objective_function(x):
    return sum(x**2)  # Example: minimize the sum of squares of elements

# GWO algorithm implementation
def GWO(obj_func, dim, n_wolves, max_iter, lower_bound, upper_bound):
    # Initialize the positions of wolves
    wolves = np.random.uniform(lower_bound, upper_bound, (n_wolves, dim))

    # Initialize alpha, beta, delta wolves (best, second-best, third-best
solutions)
    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)
    alpha_score = float('inf')  # Lowest value of the objective function
    beta_score = float('inf')
    delta_score = float('inf')

    # Main iteration loop
    for iter in range(max_iter):
        # Evaluate the fitness of each wolf
        for i in range(n_wolves):
            fitness = obj_func(wolves[i])
            # Update alpha, beta, delta based on 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()

        # Update the position of each wolf
        a = 2 - 2 * (iter / max_iter)  # Linearly decreasing parameter
        for i in range(n_wolves):
            for j in range(dim):
```

```
                r1, r2 = np.random.random(), np.random.random()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
                X1 = alpha_pos[j] - A1 * D_alpha

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

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

                # Update wolf position
                wolves[i][j] = (X1 + X2 + X3) / 3

            # Ensure wolves stay within bounds
            wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)

    return alpha_score, alpha_pos  # Return the best solution

# Problem definition
dimension = 5
num_wolves = 10
max_iterations = 100
lower = -10
upper = 10

# Run the GWO algorithm
best_score, best_position = GWO(objective_function, dimension, num_wolves,
max_iterations, lower, upper)

print("Best Solution (Position):", best_position)
print("Best Objective Value:", best_score)
```

Output:

```
Best Solution (Position): [-5.57231784e-07 -5.89896795e-07 -6.14429041e-07 -
5.84311808e-07
 -6.29901822e-07]Best Objective Value: 1.7742051303428747e-12
```

Program 6

Implementation of Parallel Cellular Algorithms and Programs

Algorithm:

~~lab~~ Program 6

Parallel Cellular Algorithm

```
import numpy as np

def fitness_function(x, y):
    return -x**2 - y**2 + 10*x + 8*y

grid_size=(5,5)
num_Iteration= 5
search_space= (-10,10)
neighborhood_size=1

population = np.random.uniform(search_space[0],
search_space[1], (*grid_size, 2))

def get_best_neighbor(population, x, y, neighborhood_size):
    x_min, x_max = max(0, x-neighborhood_size),
                   min(grid[0], x+neighborhood_size+1)
    y_min, y_max = max(0, y-neighborhood),
                   min(grid[1], y+neighborhood_size+1)
    neighborhood = population[x_min:x_max, y_min:
                   y_max].reshape(-1,2)
    return max(neighborhood, key=lambda ind:
        fitness_function(ind[0], ind[1]))

for iteration in range(num_iteration):
    new_population = np.copy(population)
    for x in range(grid_size[0]):
        for y in range(grid_size[1]):
            best_neighbor = get_best_neighbor(population,
                            x, y, neighborhood_size)
```

Date / /
Page

```
new_population[x, y] = np.clip(best_neighbour +
                       mutation, search_space[0],
                       search_space[1])
population = new_population

    fitness = np.array([[fitness_function(
              ind[0], ind[1]) for ind in row]
              for row in population])
    best_idx = np.unravel_index(fitness.argmax(),
               fitness.shape)
    print(f"Iteration {iteration+1}: Best fitness=
          {fitness[best_idx]:.2f}, Best Cell=
          {population[best_idx]}")

# output final best solution
best_idx = np.unravel_index(fitness.argmax(),
           fitness.shape)
print(f"\nBest solution: {x, y} = {population[best
      _idx]}, fitness {best_idx]:.2f}")
```

output

Iteration 1: Best fitness= 40.60, Best cell=
[4.78053407 4.59665832]
Iteration 2: Best fitness= 40.99, Best cell=
[5.070018 3.90911838]
Iteration 3: Best fitness = 40.98, Best cell
[4.86433292 3.93221728]
Iteration 4: Best fitness = 40.97, Best cell
= [5.05377386 3.84260916]
Iteration 5: Best fitness = 40.98,
Best cell = [5.12305527 4.05919837]

Code:

```
import numpy as np

# Define the fitness function
def fitness_function(x):
    return -x**2 + 5*x + 6  # Example function to maximize

# Parameters
grid_size = 5  # Define a 5x5 grid
num_particles = grid_size ** 2
num_iterations = 50
search_space = (-10, 10)
inertia_weight = 0.5
personal_influence = 1.5
neighbor_influence = 1.5

# Initialize particle positions and velocities
positions = np.random.uniform(search_space[0], search_space[1], num_particles)
velocities = np.random.uniform(-1, 1, num_particles)

# Store personal best positions and their fitness values
personal_best_positions = np.copy(positions)
personal_best_fitness = np.array([fitness_function(pos) for pos in positions])

# Define a 2D grid to simulate cellular automata
grid = positions.reshape((grid_size, grid_size))

# Main PCA loop
for iteration in range(num_iterations):
    # Update particles
    for i in range(grid_size):
        for j in range(grid_size):
            # Get the current particle index
            particle_idx = i * grid_size + j

            # Get neighbors in the grid (using periodic boundary conditions)
            neighbors = [
                ((i-1) % grid_size, j),  # Up
                ((i+1) % grid_size, j),  # Down
                (i, (j-1) % grid_size),  # Left
                (i, (j+1) % grid_size),  # Right
            ]

            # Find the best neighbor
```

```
            best_neighbor_fitness = -np.inf
            best_neighbor_position = positions[particle_idx]
            for ni, nj in neighbors:
                neighbor_idx = ni * grid_size + nj
                neighbor_fitness = fitness_function(positions[neighbor_idx])
                if neighbor_fitness > best_neighbor_fitness:
                    best_neighbor_fitness = neighbor_fitness
                    best_neighbor_position = positions[neighbor_idx]

            # Update velocity using the PSO formula
            r1, r2 = np.random.rand(), np.random.rand()
            velocities[particle_idx] = (
                inertia_weight * velocities[particle_idx] +
                personal_influence * r1 *
(personal_best_positions[particle_idx] - positions[particle_idx]) +
                neighbor_influence * r2 * (best_neighbor_position -
positions[particle_idx])
            )

            # Update position
            positions[particle_idx] += velocities[particle_idx]
            # Clip position to search space
            positions[particle_idx] = np.clip(positions[particle_idx],
search_space[0], search_space[1])

    # Update personal bests
    for i in range(num_particles):
        fitness = fitness_function(positions[i])
        if fitness > personal_best_fitness[i]:
            personal_best_fitness[i] = fitness
            personal_best_positions[i] = positions[i]

    # Track and print the best fitness in this iteration
    best_fitness = personal_best_fitness.max()
    best_position = personal_best_positions[personal_best_fitness.argmax()]
    print(f"Iteration {iteration+1}: Best Fitness = {best_fitness:.2f}, Best
Position = {best_position:.2f}")

# Output the best solution found
print(f"\nBest solution: x = {best_position:.2f}, Fitness =
{best_fitness:.2f}")
```

Output:

```
Iteration 1: Best Fitness = 12.25, Best Position = 2.50
Iteration 2: Best Fitness = 12.25, Best Position = 2.50
Iteration 3: Best Fitness = 12.25, Best Position = 2.50
```

```
Iteration 4: Best Fitness = 12.25, Best Position = 2.50
Iteration 5: Best Fitness = 12.25, Best Position = 2.50
Iteration 6: Best Fitness = 12.25, Best Position = 2.50
Iteration 7: Best Fitness = 12.25, Best Position = 2.50
Iteration 8: Best Fitness = 12.25, Best Position = 2.50
Iteration 9: Best Fitness = 12.25, Best Position = 2.50
Iteration 10: Best Fitness = 12.25, Best Position = 2.50
Iteration 11: Best Fitness = 12.25, Best Position = 2.50
Iteration 12: Best Fitness = 12.25, Best Position = 2.50
Iteration 13: Best Fitness = 12.25, Best Position = 2.50
Iteration 14: Best Fitness = 12.25, Best Position = 2.50
Iteration 15: Best Fitness = 12.25, Best Position = 2.50
Iteration 16: Best Fitness = 12.25, Best Position = 2.50
Iteration 17: Best Fitness = 12.25, Best Position = 2.50
Iteration 18: Best Fitness = 12.25, Best Position = 2.50
Iteration 19: Best Fitness = 12.25, Best Position = 2.50
Iteration 20: Best Fitness = 12.25, Best Position = 2.50
Iteration 21: Best Fitness = 12.25, Best Position = 2.50
Iteration 22: Best Fitness = 12.25, Best Position = 2.50
Iteration 23: Best Fitness = 12.25, Best Position = 2.50
Iteration 24: Best Fitness = 12.25, Best Position = 2.50
Iteration 25: Best Fitness = 12.25, Best Position = 2.50
Iteration 26: Best Fitness = 12.25, Best Position = 2.50
Iteration 27: Best Fitness = 12.25, Best Position = 2.50
Iteration 28: Best Fitness = 12.25, Best Position = 2.50
Iteration 29: Best Fitness = 12.25, Best Position = 2.50
Iteration 30: Best Fitness = 12.25, Best Position = 2.50
Iteration 31: Best Fitness = 12.25, Best Position = 2.50
Iteration 32: Best Fitness = 12.25, Best Position = 2.50
Iteration 33: Best Fitness = 12.25, Best Position = 2.50
Iteration 34: Best Fitness = 12.25, Best Position = 2.50
Iteration 35: Best Fitness = 12.25, Best Position = 2.50
Iteration 36: Best Fitness = 12.25, Best Position = 2.50
Iteration 37: Best Fitness = 12.25, Best Position = 2.50
Iteration 38: Best Fitness = 12.25, Best Position = 2.50
Iteration 39: Best Fitness = 12.25, Best Position = 2.50
Iteration 40: Best Fitness = 12.25, Best Position = 2.50
Iteration 41: Best Fitness = 12.25, Best Position = 2.50
Iteration 42: Best Fitness = 12.25, Best Position = 2.50
Iteration 43: Best Fitness = 12.25, Best Position = 2.50
Iteration 44: Best Fitness = 12.25, Best Position = 2.50
Iteration 45: Best Fitness = 12.25, Best Position = 2.50
Iteration 46: Best Fitness = 12.25, Best Position = 2.50
Iteration 47: Best Fitness = 12.25, Best Position = 2.50
Iteration 48: Best Fitness = 12.25, Best Position = 2.50
Iteration 49: Best Fitness = 12.25, Best Position = 2.50
Iteration 50: Best Fitness = 12.25, Best Position = 2.50

Best solution: x = 2.50, Fitness = 12.25
```

Program 7

Implementation of Gene Expression Algorithms (GEA)

Algorithm:

Expression
Program Genetic Algorithm

```
import numpy as np

def fitness_function(x):
    return -x**2 + 5*x+6

population_size =10
mutation_rate =0.1
crossover_rate =0.8
num_generations =50
search_space = (-10, 10)

population= np.random.uniform(search_space
[0], search_space[1], population_size)

for generation in range(num_generations):
    fitness= np.array([fitness_function(ind)
            for ind in population])
    adjusted_fitness= fitness-fitness.min()+1
    probabilities= adjusted_fitness/adjusted_
                fitness.sum()
    selected = np.random.choice(population,
        size=population_size, p=probabilities)

    next_population=[]
    for i in range(0, population_size, 2):
        parent1 = selected[i]
        parent2 = selected[(i+1) % population_size]
        if np.random.rand() < crossover_rate:
            offspring1 = (parent1+parent2)/2
            offspring2 = (parent2+parent1)/2
        else:
```

Date ___/___/___
Page ________

```
offspring + = np.random.uniform(-1,1)
if np.random.rand() < mutation_rate
else 0
offspring + = np.random.uniform(-1,1)
if np.random.rand() < mutation_rate
else 0

best_idx = fitness.argmax()
print(f"Generation {generation+1}:
    Best fitness = {fitness[best_idx]:.2f},
    Best Individual = {population[best_idx]
    :.2f}")

final_fitness = np.array([fitness_function(ind)
                for ind in population])
best_idx = final_fitness.argmax()
print(f"\n Best Solution: x = {population[
    best_idx]:.2f}, fitness = {final_fitness
    [best_idx]:.2f}")
```

output

Best solution x = 2.48, fitness = 12.20

Code:

```
import numpy as np

# Define the function to optimize
def fitness_function(x, y):
    return -x**2 - y**2 + 10*x + 8*y  # Example function to maximize

# Parameters
population_size = 20
num_genes = 2  # Each sequence represents (x, y)
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.8
search_space = (-10, 10)

# Gene Expression Function (maps genes to variables)
def express_genes(genes):
    return genes  # In this case, genes directly represent (x, y)

# Initialize population with random genetic sequences
def initialize_population(size, num_genes, search_space):
    return np.random.uniform(search_space[0], search_space[1], (size,
num_genes))

# Fitness Evaluation
def evaluate_fitness(population):
    return np.array([fitness_function(ind[0], ind[1]) for ind in population])

# Selection (Roulette Wheel Selection)
def select_parents(population, fitness):
    probabilities = fitness - fitness.min() + 1  # Make fitness non-negative
    probabilities /= probabilities.sum()
    parent_indices = np.random.choice(len(population), size=len(population),
p=probabilities)
    return population[parent_indices]

# Crossover
def perform_crossover(parent1, parent2, crossover_rate):
    if np.random.rand() < crossover_rate:
        crossover_point = np.random.randint(1, num_genes)
        offspring1 = np.concatenate((parent1[:crossover_point],
parent2[crossover_point:]))
```

```
        offspring2 = np.concatenate((parent2[:crossover_point],
parent1[crossover_point:]))
        return offspring1, offspring2
    return parent1, parent2

# Mutation
def mutate(offspring, mutation_rate, search_space):
    for gene_index in range(len(offspring)):
        if np.random.rand() < mutation_rate:
            offspring[gene_index] += np.random.uniform(-1, 1)
            offspring[gene_index] = np.clip(offspring[gene_index],
search_space[0], search_space[1])
    return offspring

# Main Gene Expression Algorithm
def gene_expression_algorithm():
    population = initialize_population(population_size, num_genes,
search_space)

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

        # Selection
        parents = select_parents(population, fitness)

        # Generate offspring
        offspring = []
        for i in range(0, len(parents), 2):
            parent1 = parents[i]
            parent2 = parents[(i + 1) % len(parents)]
            child1, child2 = perform_crossover(parent1, parent2,
crossover_rate)
            offspring.append(mutate(child1, mutation_rate, search_space))
            offspring.append(mutate(child2, mutation_rate, search_space))

        # Update population
        population = np.array(offspring)

        # Track best solution
        best_fitness = fitness.max()
        best_individual = population[np.argmax(fitness)]
        print(f"Generation {generation+1}: Best Fitness = {best_fitness:.2f},
Best Individual = {best_individual}")
```

```python
    # Final best solution
    fitness = evaluate_fitness(population)
    best_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_fitness = fitness[best_index]
    print(f"\nBest Solution: (x, y) = {best_solution}, Fitness =
{best_fitness:.2f}")

# Run the algorithm
gene_expression_algorithm()
```

Output:

```
Generation 1: Best Fitness = 39.32, Best Individual = [6.4443493  6.05169415]
Generation 2: Best Fitness = 38.87, Best Individual = [5.11149113 6.05169415]
Generation 3: Best Fitness = 36.78, Best Individual = [8.82145888 5.21351385]
Generation 4: Best Fitness = 40.27, Best Individual = [3.90027534 0.9369009 ]
Generation 5: Best Fitness = 40.27, Best Individual = [7.15955866 6.05169415]
Generation 6: Best Fitness = 39.51, Best Individual = [5.11149113 6.05169415]
Generation 7: Best Fitness = 39.42, Best Individual = [5.85288378 5.85293052]
Generation 8: Best Fitness = 39.42, Best Individual = [7.15955866 5.21351385]
Generation 9: Best Fitness = 39.42, Best Individual = [5.20227392 5.2437933 ]
Generation 10: Best Fitness = 39.41, Best Individual = [5.65804167
5.21351385]Generation 11: Best Fitness = 39.49, Best Individual = [4.90859442
5.21351385]Generation 12: Best Fitness = 39.52, Best Individual = [4.90859442
5.21351385]Generation 13: Best Fitness = 39.52, Best Individual = [5.85288378
4.30115549]Generation 14: Best Fitness = 40.18, Best Individual = [5.85288378
5.21351385Generation 15: Best Fitness = 39.89, Best Individual = [5.65804167
5.04958577]Generation 16: Best Fitness = 39.89, Best Individual = [5.85288378
5.21351385]Generation 17: Best Fitness = 39.89, Best Individual = [4.90859442
5.04958577]Generation 18: Best Fitness = 39.89, Best Individual = [5.85288378
5.04958577]Generation 19: Best Fitness = 39.95, Best Individual = [5.54882566
5.04958577]Generation 20: Best Fitness = 40.48, Best Individual = [5.54882566
5.04958577]Generation 21: Best Fitness = 40.54, Best Individual = [5.65804167
5.04958577]Generation 22: Best Fitness = 40.41, Best Individual = [5.54882566
5.43490292]
Generation 23: Best Fitness = 39.91, Best Individual = [5.65804167
5.21351385]Generation 24: Best Fitness = 40.19, Best Individual = [5.79364332
4.8861793 ]Generation 25: Best Fitness = 40.70, Best Individual = [5.54882566
5.43490292]Generation 26: Best Fitness = 40.80, Best Individual = [4.63864695
5.04958577]Generation 27: Best Fitness = 40.70, Best Individual = [5.44084835
4.0483479 ]Generation 28: Best Fitness = 40.80, Best Individual = [5.16742759
6.11768563]Generation 29: Best Fitness = 40.74, Best Individual = [5.93615991
4.0483479 ]Generation 30: Best Fitness = 40.82, Best Individual = [5.44084835
4.8861793 ]Generation 31: Best Fitness = 40.99, Best Individual = [5.91045538
3.88969771]Generation 32: Best Fitness = 40.98, Best Individual = [5.34471628
3.88969771]Generation 33: Best Fitness = 40.88, Best Individual = [5.44084835
4.8861793 ]Generation 34: Best Fitness = 40.96, Best Individual = [4.89894219
```

```
3.82584918]Generation 35: Best Fitness = 40.96, Best Individual = [5.83107111
3.82584918]
Generation 36: Best Fitness = 40.96, Best Individual = [4.53974585 3.8960765 ]
Generation 37: Best Fitness = 40.96, Best Individual = [5.41385951 3.26677091]
Generation 38: Best Fitness = 40.98, Best Individual = [5.92663004 3.88969771]
Generation 39: Best Fitness = 40.98, Best Individual = [4.89894219 4.45997236]
Generation 40: Best Fitness = 40.99, Best Individual = [4.89894219 4.25553679]
Generation 41: Best Fitness = 40.98, Best Individual = [5.41385951 3.67520291]
Generation 42: Best Fitness = 40.96, Best Individual = [4.35953253 3.20848384]
Generation 43: Best Fitness = 40.99, Best Individual = [5.41385951 3.82584918]
Generation 44: Best Fitness = 40.99, Best Individual = [5.41385951 3.26677091]
Generation 45: Best Fitness = 40.91, Best Individual = [5.41385951 4.18141689]
Generation 46: Best Fitness = 40.99, Best Individual = [4.53277483 4.14149616]
Generation 47: Best Fitness = 40.97, Best Individual = [5.55743837 3.32829027]
Generation 48: Best Fitness = 40.91, Best Individual = [4.58377247 3.26677091]
Generation 49: Best Fitness = 40.87, Best Individual = [4.58377247 3.26677091]
Generation 50: Best Fitness = 40.97, Best Individual = [5.00052736 3.82584918]

Best Solution: (x, y) = [5.00052736 3.82584918], Fitness = 40.97
```