VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Prerana P Jois (1BM23CS250)

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 **Prerana P Jois (1BM23CS250),** 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.

Dr. Namratha M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/preranapjois/BIS_LAB.git

Program 1:
Genetic algorithm

Genetic algorithm

Algorithm:

Genetic-algo (S)

input (i): S → set of blocks
output: superstring of set

initialization:

initialize the population, t_0 ← generation

Eval fitness (S, p_t).

while termination conditions not met

do {
select individuals from P_t
Recombine individuals
Mutate individuals
EVALUATE-FITNESS-MA (S, modified individuals)

P_{t+1} ← newly created individuals
t ← t + 1

return (superstring derived from best individual in P)

procedure EVALUATE-FITNESS GA(S, P)

S ← set of blocks
P - population of individuals

for each individual $i \in P$

do
general derived string $s(i)$
$m \leftarrow$ all blocks from S that are not covered by $s(i)$

$s'(i) \leftarrow$ concatenation of $s(i)$ & m
$fitness(i) \leftarrow \frac{1}{\|s'(i)\|^2}$

crossover - individuals (S, P_1, P_2)

New [ ];

for i in range(P_1 . length);

P = random. random( )
if P<0.5
New [i] < P_1[i];
elif P< 0.9

Date
Page

New [i] = P_2 [i];
else
New [i] = mutate (S)
}
return New;

Steps :

(1) Initialize population

(2) Evaluate fitness

(3) while termination not reached :

* Select best individuals

* Apply crossover & mutation

* Evaluate new individuals

* Replace old population with new one.

— Eveuli in Neural N/w

Code :

```
import numpy as np

inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
expected = np.array([[0], [1], [1], [0]])

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

def forward_pass(weights, x):
    w1 = weights[:4].reshape((2, 2))
    b1 = weights[4:6]
    w2 = weights[6:8].reshape((2, 1))
    b2 = weights[8]
    z1 = np.dot(x, w1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, w2) + b2
    a2 = sigmoid(z2)
    return a2

def fitness(weights):
    predictions = forward_pass(weights, inputs)
    error = np.mean((predictions - expected) ** 2)
    return -error

pop_size = 100
num_weights = 9
generations = 200
mutation_rate = 0.1

population = np.random.uniform(-1, 1, (pop_size, num_weights))

for generation in range(generations):
    fitness_scores = np.array([fitness(ind) for ind in population])
    sorted_idx = np.argsort(fitness_scores)[::-1]
    population = population[sorted_idx]
    fitness_scores = fitness_scores[sorted_idx]
```

```
    top_n = int(0.2 * pop_size)
    parents = population[:top_n]
    children = []

    while len(children) < pop_size - top_n:
        p1, p2 = parents[np.random.randint(0, top_n, 2)]
        crossover = np.random.randint(1, num_weights)
        child = np.concatenate((p1[:crossover], p2[crossover:]))
        if np.random.rand() < mutation_rate:
            mutation_point = np.random.randint(num_weights)
            child[mutation_point] += np.random.uniform(-0.5, 0.5)
        children.append(child)

    population = np.vstack((parents, children))

    if generation % 10 == 0 or generation == generations - 1:
        print(f"Generation {generation+1} | Best fitness: {fitness_scores[0]:.4f}")

best_weights = population[0]
print("\nFinal predictions:")
preds = forward_pass(best_weights, inputs)
for i in range(len(inputs)):
    print(f"Input: {inputs[i]} => Predicted: {preds[i][0]:.4f} | Expected: {expected[i][0]}")

print("\nDataset: XOR dataset")
```

Output:

Generation 1 | Best fitness: -0.2497

Generation 11 | Best fitness: -0.2430

Generation 21 | Best fitness: -0.2329

Generation 31 | Best fitness: -0.2196

Generation 41 | Best fitness: -0.2029

Generation 51 | Best fitness: -0.1898

Generation 61 | Best fitness: -0.1626

Generation 71 | Best fitness: -0.1423

Generation 81 | Best fitness: -0.1184

Generation 91 | Best fitness: -0.0898

Generation 101 | Best fitness: -0.0751

Generation 111 | Best fitness: -0.0538

Generation 121 | Best fitness: -0.0463

Generation 131 | Best fitness: -0.0312

Generation 141 | Best fitness: -0.0221

Generation 151 | Best fitness: -0.0142

Generation 161 | Best fitness: -0.0098

Generation 171 | Best fitness: -0.0060

Generation 181 | Best fitness: -0.0036

Generation 191 | Best fitness: -0.0021

Generation 200 | Best fitness: -0.0012

Final predictions:

Input: [0 0] => Predicted: 0.0440 | Expected: 0

Input: [0 1] => Predicted: 0.9750 | Expected: 1

Input: [1 0] => Predicted: 0.9643 | Expected: 1

Input: [1 1] => Predicted: 0.0331 | Expected: 0

Dataset: XOR dataset

This is a classic non-linear classification problem (cannot be solved by a single perceptron; needs hidden layers or non-linear models).

Program 2:
Gene Expression algorithm

Page

Gene expression

Input:
Population size (N)
Number of generation (G)
Mutation probability (Pm)
Crossover probability (Pc)
Transposition probability (Pt)
Problem - specific fitness function F(

Initialization
Population ← Generate random chrom
for each chromosome C in population do
Expression ← Decode (C) // convert gen->ex
Fitness (C) ← F(Expression)
End for

For gen = 1 to G do
New population ← φ

While size (New Population) < N do
Parent 1 ← Select (population)
Parent 2 ← Select (Population)

With Probability Pc:
(Child1, child2) ← crossover
(Parent1, Parent2)

```
otherwise:
    child1 ← copy (Parent1)
    child2 ← copy (Parent2)

with Probability Pm:
        child1 ← mutate (child1)

with probability Pm:
        child2 ← mutate (child2)

with probability Pt:
        child1 ← Transpose (child1)
with probability Pt:
        child2 ← Transpose (child2)

// evaluate fitness

Expression1 ← Decode (child1)
Expression2 ← Decode (child2)

Fitness (child1) ← F(Expression1)
Fitness (child2) ← F(Expression2)

Add child1, child2 to NewPopulation
End while

// Replacement
Population ← select Best (Newpopulation, N)
End For
```

```
Best ← chromosome with highest
        fitness in population

BestExpression ← Decode (Best)
Return BestExpression
End.
```

- Execute in the field of IP.
- Valuate the results observed & point out the difference of GA & GE on IP.

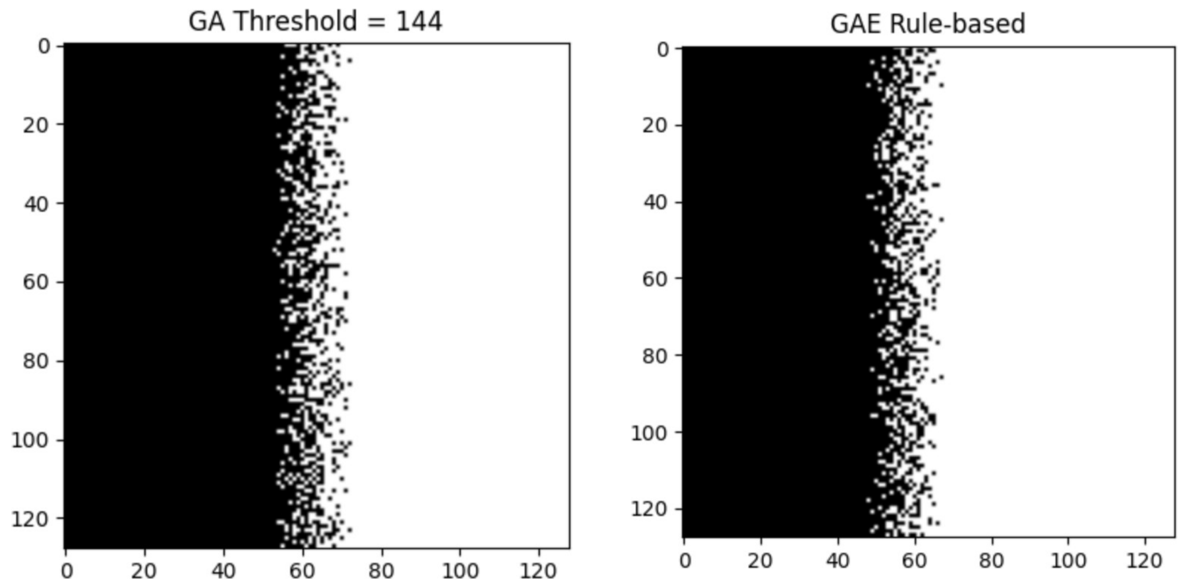
Code:

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

rows, cols = 128, 128
x = np.tile(np.linspace(0, 255, cols, dtype=np.uint8), (rows, 1))
noise = np.random.randint(0, 40, (rows, cols), dtype=np.uint8)
img = cv2.add(x, noise)

def fitness(threshold):
    foreground = img[img > threshold]
    background = img[img <= threshold]
    if len(foreground) == 0 or len(background) == 0:
        return 0
    w0 = len(background) / (rows * cols)
    w1 = len(foreground) / (rows * cols)
    m0 = np.mean(background)
    m1 = np.mean(foreground)
    return w0 * w1 * ((m0 - m1) ** 2)

pop_size = 30
generations = 40
population = np.random.randint(0, 255, size=pop_size)

for g in range(generations):
    scores = np.array([fitness(t) for t in population])
    best_idx = np.argmax(scores)
    best_threshold = population[best_idx]
    print(f"Generation {g+1} | Best threshold: {best_threshold}")
    sorted_idx = np.argsort(scores)[::-1]
    parents = population[sorted_idx[:pop_size // 2]]
    children = []
    while len(children) < pop_size - len(parents):
        p1, p2 = random.sample(list(parents), 2)
        child = (p1 + p2) // 2
        if random.random() < 0.3:
            child += random.randint(-10, 10)
        child = np.clip(child, 0, 255)
        children.append(child)
    population = np.concatenate((parents, children))

best_threshold = int(best_threshold)
print("\nGA Result → Best threshold found =", best_threshold)
```

```
_, ga_result = cv2.threshold(img, best_threshold, 255, cv2.THRESH_BINARY)

rule_threshold = int(0.65 * np.mean(img) + 40)
_, rule_result = cv2.threshold(img, rule_threshold, 255, cv2.THRESH_BINARY)
print("Rule-based threshold result =", rule_threshold)

plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.imshow(ga_result, cmap='gray')
plt.title(f"GA Threshold = {best_threshold}")
plt.subplot(2, 1, 2)
plt.imshow(rule_result, cmap='gray')
plt.title("GAE Rule-based")
plt.tight_layout()
plt.show()

print("\n* GA Result → Best threshold found =", best_threshold)
print("* GAE Rule-based result =", rule_threshold)
print("1) Original: synthetic gradient + noise")
print("2) GA thresholding result =", best_threshold)
```

Output:

Generation 1 | Best threshold: 146 Generation 2 | Best threshold: 144 Generation 3 | Best threshold: 144 Generation 4 | Best threshold: 144 Generation 5 | Best threshold: 144 Generation 6 | Best threshold: 144 Generation 7 | Best threshold: 144 Generation 8 | Best threshold: 144 Generation 9 | Best threshold: 144 Generation 10 | Best threshold: 144 Generation 11 | Best threshold: 144 Generation 12 | Best threshold: 144 Generation 13 | Best threshold: 144 Generation 14 | Best threshold: 144 Generation 15 | Best threshold: 144 Generation 16 | Best threshold: 144 Generation 17 | Best threshold: 144 Generation 18 | Best threshold: 144 Generation 19 | Best threshold: 144 Generation 20 | Best threshold: 144 Generation 21 | Best threshold: 144 Generation 22 | Best threshold: 144 Generation 23 | Best threshold: 144 Generation 24 | Best threshold: 144 Generation 25 | Best threshold: 144 Generation 26 | Best threshold: 144 Generation 27 | Best threshold: 144 Generation 28 | Best threshold: 144 Generation 29 | Best threshold: 144 Generation 30 | Best threshold: 144 Generation 31 | Best threshold: 144 Generation 32 | Best threshold: 144 Generation 33 | Best threshold: 144 Generation 34 | Best threshold: 144 Generation 35 | Best threshold: 144 Generation 36 | Best threshold: 144 Generation 37 | Best threshold: 144 Generation 38 | Best threshold: 144 Generation 39 | Best threshold: 144 Generation 40 | Best threshold: 144 GA Result → Best threshold found = 144 Rule-based threshold result = 134



* GA Result → Best threshold found = 144
* GAE Rule-based result = 134
1) Original: synthetic gradient + noise
2) GA thresholding result = 144

Program 3:
Particle swan optimization

Particle Swan optimization

Algorithm PSO

Input:

```
    N
    MaxIter      // no of ~~particles~~, iterations
    w, c1, c2    // PSO parameters
    xmin, xmax
    F(x)
```

output:

```
    gbest
```

```
Begin
  // Step 1: Initialize particles
  For i = 1 to N do
     Position [i] <- random (xmin, xmax)
     velocity [i] <- random (-|xmax - xmin|,
        |xmax - xmin|)

     Pbest [i] <- Position [i]
     Fitness_Pbest [i] <- F (Position [i])
  EndFor
  gbest <- best of Pbest
```

Date ______
Page ______

//Step 2: Iteration

```
For iter = 1 to MaxIter do
    For i = 1 to N do
        //update velocity
        velocity [i] ← w* velocity [i]
                          +
        c1 * rand () * (Pbest [i] - Position[i])
                          +
        c2* rand()* (gbest - Position [i])
            //update position
            Position [i] ← Position [i] +
        velocity [i]
            //Boundary check
            If Position [i] < xmin then
        Position [i] ← xmin
            If position [i] > xmax then
        Position [i] ← xmax.
            //Evaluate fitness
diff b/w gbest  Fitness ← F(Position [i])
& pbest         //update personal best
Extent in       If fitness better Than
the fed         Fitness_pbest [i] then
of Energy optimiz  Pbest [i] ← Position [i]
                Fitness_Pbest [i] ← Fitness
            EndIf
        End for
    End for

    Return gbest
End.
```

Code :

```
import numpy as np
def energy_function(position):
    x, y = position
    return (x**2 + y**2) + 10 * np.sin(x) * np.sin(y)

num_particles = 30
dimensions = 2
iterations = 100

w = 0.7
c1 = 1.5
c2 = 1.5
positions = np.random.uniform(-10, 10, (num_particles, dimensions))
velocities = np.random.uniform(-1, 1, (num_particles, dimensions))
pbest_positions = np.copy(positions)
pbest_scores = np.array([energy_function(p) for p in positions])

gbest_position = pbest_positions[np.argmin(pbest_scores)]
gbest_score = np.min(pbest_scores)

for t in range(iterations):
    for i in range(num_particles):
        fitness = energy_function(positions[i])
        if fitness < pbest_scores[i]:
            pbest_scores[i] = fitness
            pbest_positions[i] = positions[i]

    best_particle = np.argmin(pbest_scores)
    if pbest_scores[best_particle] < gbest_score:
        gbest_score = pbest_scores[best_particle]
        gbest_position = pbest_positions[best_particle]
    r1, r2 = np.random.rand(), np.random.rand()
    for i in range(num_particles):
        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (pbest_positions[i] - positions[i]) +
                         c2 * r2 * (gbest_position - positions[i]))
        positions[i] += velocities[i]

    print(f"Iteration {t+1}: Best Fitness = {gbest_score:.12e}")
print("\nBest Solution Found:")
print(gbest_position)
print("\nBest Objective Function Value:")
print(gbest_score)
```

Output:

Iteration 1: Best Fitness = -2.569029538935e-01
Iteration 2: Best Fitness = -4.739972162755e+00
Iteration 3: Best Fitness = -4.895159615348e+00
Iteration 4: Best Fitness = -4.895159615348e+00
Iteration 5: Best Fitness = -4.980324627255e+00
Iteration 6: Best Fitness = -5.226913277912e+00
Iteration 7: Best Fitness = -5.226913277912e+00
Iteration 8: Best Fitness = -5.391303100542e+00
Iteration 9: Best Fitness = -5.391303100542e+00
Iteration 10: Best Fitness = -5.391303100542e+00
Iteration 11: Best Fitness = -5.866209539629e+00
Iteration 12: Best Fitness = -5.899624748247e+00
Iteration 13: Best Fitness = -5.899624748247e+00
Iteration 14: Best Fitness = -5.899624748247e+00
Iteration 15: Best Fitness = -5.899624748247e+00
Iteration 16: Best Fitness = -5.899624748247e+00
Iteration 17: Best Fitness = -5.903361933188e+00
Iteration 18: Best Fitness = -5.904170911460e+00
Iteration 19: Best Fitness = -5.904170911460e+00
Iteration 20: Best Fitness = -5.904170911460e+00
Iteration 21: Best Fitness = -5.904470092450e+00
Iteration 22: Best Fitness = -5.904470092450e+00
Iteration 23: Best Fitness = -5.904470092450e+00
Iteration 24: Best Fitness = -5.904470092450e+00
Iteration 25: Best Fitness = -5.904473550935e+00
Iteration 26: Best Fitness = -5.904473550935e+00
Iteration 27: Best Fitness = -5.904473550935e+00
Iteration 28: Best Fitness = -5.904473550935e+00
Iteration 29: Best Fitness = -5.904473550935e+00
Iteration 30: Best Fitness = -5.904473550935e+00
Iteration 31: Best Fitness = -5.904473550935e+00
Iteration 32: Best Fitness = -5.904473550935e+00
Iteration 33: Best Fitness = -5.904473550935e+00
Iteration 34: Best Fitness = -5.904473550935e+00
Iteration 35: Best Fitness = -5.904473550935e+00
Iteration 36: Best Fitness = -5.904473550935e+00
Iteration 37: Best Fitness = -5.904490733230e+00
Iteration 38: Best Fitness = -5.904490733230e+00
Iteration 39: Best Fitness = -5.904490765753e+00
Iteration 40: Best Fitness = -5.904490765753e+00
Iteration 41: Best Fitness = -5.904490765753e+00
Iteration 42: Best Fitness = -5.904490765753e+00
Iteration 43: Best Fitness = -5.904490765753e+00

Iteration 44: Best Fitness = -5.904490791690e+00
Iteration 45: Best Fitness = -5.904490791690e+00
Iteration 46: Best Fitness = -5.904490791690e+00
Iteration 47: Best Fitness = -5.904491355488e+00
Iteration 48: Best Fitness = -5.904491355488e+00
Iteration 49: Best Fitness = -5.904491355488e+00
Iteration 50: Best Fitness = -5.904491355488e+00
Iteration 51: Best Fitness = -5.904491355488e+00
Iteration 52: Best Fitness = -5.904491355488e+00
Iteration 53: Best Fitness = -5.904491355488e+00
Iteration 54: Best Fitness = -5.904491366230e+00
Iteration 55: Best Fitness = -5.904491366230e+00
Iteration 56: Best Fitness = -5.904491507256e+00
Iteration 57: Best Fitness = -5.904491507256e+00
Iteration 58: Best Fitness = -5.904491547815e+00
Iteration 59: Best Fitness = -5.904491547815e+00
Iteration 60: Best Fitness = -5.904491547815e+00
Iteration 61: Best Fitness = -5.904491547815e+00
Iteration 62: Best Fitness = -5.904491547815e+00
Iteration 63: Best Fitness = -5.904491547815e+00
Iteration 64: Best Fitness = -5.904491547815e+00
Iteration 65: Best Fitness = -5.904491560085e+00
Iteration 66: Best Fitness = -5.904491560085e+00
Iteration 67: Best Fitness = -5.904491560085e+00
Iteration 68: Best Fitness = -5.904491560085e+00
Iteration 69: Best Fitness = -5.904491560085e+00
Iteration 70: Best Fitness = -5.904491560085e+00
Iteration 71: Best Fitness = -5.904491560085e+00
Iteration 72: Best Fitness = -5.904491560085e+00
Iteration 73: Best Fitness = -5.904491560085e+00
Iteration 74: Best Fitness = -5.904491560085e+00
Iteration 75: Best Fitness = -5.904491646024e+00
Iteration 76: Best Fitness = -5.904491663358e+00
Iteration 77: Best Fitness = -5.904491665196e+00
Iteration 78: Best Fitness = -5.904491665196e+00
Iteration 79: Best Fitness = -5.904491670423e+00
Iteration 80: Best Fitness = -5.904491670423e+00
Iteration 81: Best Fitness = -5.904491670423e+00
Iteration 82: Best Fitness = -5.904491670423e+00
Iteration 83: Best Fitness = -5.904491670423e+00
Iteration 84: Best Fitness = -5.904491670423e+00
Iteration 85: Best Fitness = -5.904491670423e+00
Iteration 86: Best Fitness = -5.904491670423e+00
Iteration 87: Best Fitness = -5.904491670423e+00
Iteration 88: Best Fitness = -5.904491670423e+00
Iteration 89: Best Fitness = -5.904491670423e+00

```
Iteration 90: Best Fitness = -5.904491670423e+00
Iteration 91: Best Fitness = -5.904491670423e+00
Iteration 92: Best Fitness = -5.904491670423e+00
Iteration 93: Best Fitness = -5.904491670423e+00
Iteration 94: Best Fitness = -5.904491670610e+00
Iteration 95: Best Fitness = -5.904491670610e+00
Iteration 96: Best Fitness = -5.904491670610e+00
Iteration 97: Best Fitness = -5.904491670610e+00
Iteration 98: Best Fitness = -5.904491670610e+00
Iteration 99: Best Fitness = -5.904491670610e+00
Iteration 100: Best Fitness = -5.904491670610e+00

Best Solution Found: [-1.29786271 1.29787174]
Best Objective Function Value: -5.904491670609598
```

Program 4:
Ant colony optimization for the travelling salesman problem

Date
Page

Ant colony optimization for the travelling salesman problem

Algorithm:

(1) Initialyze pher

(1) Input: number of cities N, cost function cost(i, j) (or distance matrix), parameters α, β, ρ (evaporation), Q, n_ants, n_iters

Initialize pheromone values τ on all solution components
Set parameters α (pheromone influence), β (heuristic influence), evaporation rate ρ, number of ants m

while stopping criteria not met:
for each ant k in 1 to m:
initialize an empty solution S_k
while solution S_k is incomplete:
Select next solution component c
based on probability proportional to:

[τ(c)]^α * [η(c)]^β

where η(c) is heuristic desirability of component c add component c to solution S_k evaluate the quality of solutions

for each solution component c:
evaporate pheromone:
$\tau(c) = (1-\rho) * \tau(c)$

for each ~~solution~~ ant k:
deposit pheromone on components in solution S_k:

$\tau(c) = \tau(c) + \Delta\tau\_k(c)$
amount $\Delta\tau\_k(c)$ depends on quality of solution S_k

return the best solution found

[Note:
α → Pheromone influence
β → Heuristic influence
ρ → Evaporation rate
m → number of ants

us Routi
Mgmt in Traffic

$\Delta\tau_k(c)$ → Reward signal
(that guides future ants towards better solution)

$\Delta\tau(i,j)$ = ~~The amount of pheromone deposited~~
$\Delta\tau$ → change in pheromone]

Code:

```
import numpy as np
import random

num_stops = 6
np.random.seed(42)

distances = np.random.randint(10, 100, size=(num_stops, num_stops))
for i in range(num_stops):
    distances[i][i] = 0
    for j in range(i + 1, num_stops):
        distances[j][i] = distances[i][j]

num_ants = 10
num_iterations = 100
alpha = 1.0
beta = 5.0
evaporation_rate = 0.5
Q = 100

pheromone = np.ones((num_stops, num_stops))

def route_distance(route):
    total = 0
    for i in range(len(route) - 1):
        total += distances[route[i]][route[i + 1]]
    total += distances[route[-1]][route[0]]
    return total

best_route = None
best_distance = float("inf")

for iteration in range(num_iterations):
    all_routes = []
    all_distances = []

    for ant in range(num_ants):
        route = [random.randint(0, num_stops - 1)]
        while len(route) < num_stops:
            i = route[-1]
            unvisited = [j for j in range(num_stops) if j not in route]

            probabilities = []
            for j in unvisited:
```

```
            pheromone_factor = pheromone[i][j] ** alpha
            distance_factor = (1.0 / (distances[i][j] + 1e-10)) ** beta
            probabilities.append(pheromone_factor * distance_factor)

        probabilities = np.array(probabilities)
        probabilities /= probabilities.sum()

        next_stop = np.random.choice(unvisited, p=probabilities)
        route.append(next_stop)

    total_distance = route_distance(route)
    all_routes.append(route)
    all_distances.append(total_distance)

    if total_distance < best_distance:
        best_distance = total_distance
        best_route = route

  pheromone *= (1 - evaporation_rate)
  for r, dist in zip(all_routes, all_distances):
    for i in range(len(r) - 1):
        pheromone[r[i]][r[i + 1]] += Q / dist
    pheromone[r[-1]][r[0]] += Q / dist

  print(f"Iteration {iteration+1}: Best Distance (Traffic-Aware) = {best_distance:.4f}")

print("\nOptimal Bus Route Found:")
print(" → ".join(str(stop) for stop in best_route))
print(f"Optimal Total Travel Distance (with traffic): {best_distance:.4f}")
```

Output:

```
Iteration 1: Best Distance (Traffic-Aware) = 234.0000
Iteration 2: Best Distance (Traffic-Aware) = 234.0000
Iteration 3: Best Distance (Traffic-Aware) = 234.0000
Iteration 4: Best Distance (Traffic-Aware) = 234.0000
Iteration 5: Best Distance (Traffic-Aware) = 234.0000
Iteration 6: Best Distance (Traffic-Aware) = 234.0000
Iteration 7: Best Distance (Traffic-Aware) = 234.0000
Iteration 8: Best Distance (Traffic-Aware) = 234.0000
Iteration 9: Best Distance (Traffic-Aware) = 234.0000
Iteration 10: Best Distance (Traffic-Aware) = 234.0000
Iteration 11: Best Distance (Traffic-Aware) = 234.0000
Iteration 12: Best Distance (Traffic-Aware) = 234.0000
Iteration 13: Best Distance (Traffic-Aware) = 234.0000
```

Iteration 14: Best Distance (Traffic-Aware) = 234.0000
Iteration 15: Best Distance (Traffic-Aware) = 234.0000
Iteration 16: Best Distance (Traffic-Aware) = 234.0000
Iteration 17: Best Distance (Traffic-Aware) = 234.0000
Iteration 18: Best Distance (Traffic-Aware) = 234.0000
Iteration 19: Best Distance (Traffic-Aware) = 234.0000
Iteration 20: Best Distance (Traffic-Aware) = 234.0000
Iteration 21: Best Distance (Traffic-Aware) = 234.0000
Iteration 22: Best Distance (Traffic-Aware) = 234.0000
Iteration 23: Best Distance (Traffic-Aware) = 234.0000
Iteration 24: Best Distance (Traffic-Aware) = 234.0000
Iteration 25: Best Distance (Traffic-Aware) = 234.0000
Iteration 26: Best Distance (Traffic-Aware) = 234.0000
Iteration 27: Best Distance (Traffic-Aware) = 234.0000
Iteration 28: Best Distance (Traffic-Aware) = 234.0000
Iteration 29: Best Distance (Traffic-Aware) = 234.0000
Iteration 30: Best Distance (Traffic-Aware) = 234.0000
Iteration 31: Best Distance (Traffic-Aware) = 234.0000
Iteration 32: Best Distance (Traffic-Aware) = 234.0000
Iteration 33: Best Distance (Traffic-Aware) = 234.0000
Iteration 34: Best Distance (Traffic-Aware) = 234.0000
Iteration 35: Best Distance (Traffic-Aware) = 234.0000
Iteration 36: Best Distance (Traffic-Aware) = 234.0000
Iteration 37: Best Distance (Traffic-Aware) = 234.0000
Iteration 38: Best Distance (Traffic-Aware) = 234.0000
Iteration 39: Best Distance (Traffic-Aware) = 234.0000
Iteration 40: Best Distance (Traffic-Aware) = 234.0000
Iteration 41: Best Distance (Traffic-Aware) = 234.0000
Iteration 42: Best Distance (Traffic-Aware) = 234.0000
Iteration 43: Best Distance (Traffic-Aware) = 234.0000
Iteration 44: Best Distance (Traffic-Aware) = 234.0000
Iteration 45: Best Distance (Traffic-Aware) = 234.0000
Iteration 46: Best Distance (Traffic-Aware) = 234.0000
Iteration 47: Best Distance (Traffic-Aware) = 234.0000
Iteration 48: Best Distance (Traffic-Aware) = 234.0000
Iteration 49: Best Distance (Traffic-Aware) = 234.0000
Iteration 50: Best Distance (Traffic-Aware) = 234.0000
Iteration 51: Best Distance (Traffic-Aware) = 234.0000
Iteration 52: Best Distance (Traffic-Aware) = 234.0000
Iteration 53: Best Distance (Traffic-Aware) = 234.0000
Iteration 54: Best Distance (Traffic-Aware) = 234.0000
Iteration 55: Best Distance (Traffic-Aware) = 234.0000
Iteration 56: Best Distance (Traffic-Aware) = 234.0000
Iteration 57: Best Distance (Traffic-Aware) = 234.0000
Iteration 58: Best Distance (Traffic-Aware) = 234.0000
Iteration 59: Best Distance (Traffic-Aware) = 234.0000

Iteration 60: Best Distance (Traffic-Aware) = 234.0000
Iteration 61: Best Distance (Traffic-Aware) = 234.0000
Iteration 62: Best Distance (Traffic-Aware) = 234.0000
Iteration 63: Best Distance (Traffic-Aware) = 234.0000
Iteration 64: Best Distance (Traffic-Aware) = 234.0000
Iteration 65: Best Distance (Traffic-Aware) = 234.0000
Iteration 66: Best Distance (Traffic-Aware) = 234.0000
Iteration 67: Best Distance (Traffic-Aware) = 234.0000
Iteration 68: Best Distance (Traffic-Aware) = 234.0000
Iteration 69: Best Distance (Traffic-Aware) = 234.0000
Iteration 70: Best Distance (Traffic-Aware) = 234.0000
Iteration 71: Best Distance (Traffic-Aware) = 234.0000
Iteration 72: Best Distance (Traffic-Aware) = 234.0000
Iteration 73: Best Distance (Traffic-Aware) = 234.0000
Iteration 74: Best Distance (Traffic-Aware) = 234.0000
Iteration 75: Best Distance (Traffic-Aware) = 234.0000
Iteration 76: Best Distance (Traffic-Aware) = 234.0000
Iteration 77: Best Distance (Traffic-Aware) = 234.0000
Iteration 78: Best Distance (Traffic-Aware) = 234.0000
Iteration 79: Best Distance (Traffic-Aware) = 234.0000
Iteration 80: Best Distance (Traffic-Aware) = 234.0000
Iteration 81: Best Distance (Traffic-Aware) = 234.0000
Iteration 82: Best Distance (Traffic-Aware) = 234.0000
Iteration 83: Best Distance (Traffic-Aware) = 234.0000
Iteration 84: Best Distance (Traffic-Aware) = 234.0000
Iteration 85: Best Distance (Traffic-Aware) = 234.0000
Iteration 86: Best Distance (Traffic-Aware) = 234.0000
Iteration 87: Best Distance (Traffic-Aware) = 234.0000
Iteration 88: Best Distance (Traffic-Aware) = 234.0000
Iteration 89: Best Distance (Traffic-Aware) = 234.0000
Iteration 90: Best Distance (Traffic-Aware) = 234.0000
Iteration 91: Best Distance (Traffic-Aware) = 234.0000
Iteration 92: Best Distance (Traffic-Aware) = 234.0000
Iteration 93: Best Distance (Traffic-Aware) = 234.0000
Iteration 94: Best Distance (Traffic-Aware) = 234.0000
Iteration 95: Best Distance (Traffic-Aware) = 234.0000
Iteration 96: Best Distance (Traffic-Aware) = 234.0000
Iteration 97: Best Distance (Traffic-Aware) = 234.0000
Iteration 98: Best Distance (Traffic-Aware) = 234.0000
Iteration 99: Best Distance (Traffic-Aware) = 234.0000
Iteration 100: Best Distance (Traffic-Aware) = 234.0000

Optimal Bus Route Found:
0 → 1 → 5 → 2 → 4 → 3
Optimal Total Travel Distance (with traffic): 234.0000

Program 5:
Cuckoo Search algorithm

Page

Cuckoo Search algorithm

(1) Initialize population (nests):

For $i=1$ to n:
Initialize nest i with a random solution x_i within search bounds
Evaluate fitness $F_i = f(x_i)$
End for

(2) Find the current best solution:

x_best = nest with minimum (or maximum) fitness value

(3) Repeat until stopping criterion (MaxIter or convergence):

For $t=1$ to MaxIter:

For each cuckoo ($i=1$ to n):
Generate a new solution xi_new by Levy flight:
xi_new = x_i + α * Le'vy(λ)
Evaluate fitness Fi_new = f(xi_new)
Randomly choose a nest j with ($j \neq i$)

If Fi_new < Fj:
Replace nestj with xj_new
End if
End for

For each nest i:
with probability pa:
Replace xi with a new random solution
End if
End for

Find x_best = nest with best fitness

End for.

Optimiz^n in Electrical grid

Code:

```
import numpy as np
from math import gamma, sin, pi

def electrical_grid_loss(x):
    x = np.array(x)
    return np.sum(x**2 - 10 * np.cos(2 * np.pi * x) + 10)

def cuckoo_search(obj_func, n=15, d=2, lb=-5, ub=5, pa=0.25, iterations=100):
    beta = 1.5
    sigma = (gamma(1 + beta) * sin(pi * beta / 2) /
             (gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))) ** (1 / beta)

    nests = np.random.uniform(lb, ub, (n, d))
```

```
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for t in range(iterations):
        new_nests = np.copy(nests)
        for i in range(n):
            step = np.random.normal(0, sigma, size=d) / (abs(np.random.normal(0, 1, size=d)) ** (1 /
beta))    new_nest = nests[i] + 0.01 * step * (nests[i] - best_nest)
            new_nest = np.clip(new_nest, lb, ub)
            new_nests[i] = new_nest

        new_fitness = np.array([obj_func(nest) for nest in new_nests])

        for i in range(n):
            if new_fitness[i] < fitness[i]:
                fitness[i] = new_fitness[i]
                nests[i] = new_nests[i]

        abandon_mask = np.random.rand(n, d) < pa
        random_nests = np.random.uniform(lb, ub, (n, d))
        nests = np.where(abandon_mask, random_nests, nests)
        fitness = np.array([obj_func(nest) for nest in nests])

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

        print(f"Iteration {t+1}: Best Grid Loss = {best_fitness:.10e}")

    return best_nest, best_fitness

best_solution, best_value = cuckoo_search(electrical_grid_loss, n=20, d=3, lb=-5, ub=5,
iterations=100)

print("\nOptimal Electrical Grid Configuration:")
print(best_solution)
print("\nMinimum Power Loss (Objective Value):")
print(best_value)
```

Output:

```
Iteration 1: Best Grid Loss = 1.1953471974e+01
Iteration 2: Best Grid Loss = 1.1953471974e+01
Iteration 3: Best Grid Loss = 1.1953471974e+01
Iteration 4: Best Grid Loss = 5.2025691754e+00
Iteration 5: Best Grid Loss = 5.2025691754e+00
Iteration 6: Best Grid Loss = 5.2025691754e+00
Iteration 7: Best Grid Loss = 5.2025691754e+00
Iteration 8: Best Grid Loss = 5.2025691754e+00
Iteration 9: Best Grid Loss = 5.2025691754e+00
Iteration 10: Best Grid Loss = 5.2025691754e+00
Iteration 11: Best Grid Loss = 5.2025691754e+00
Iteration 12: Best Grid Loss = 5.2025691754e+00
Iteration 13: Best Grid Loss = 5.2025691754e+00
Iteration 14: Best Grid Loss = 5.2025691754e+00
Iteration 15: Best Grid Loss = 5.2025691754e+00
Iteration 16: Best Grid Loss = 5.2025691754e+00
Iteration 17: Best Grid Loss = 5.2025691754e+00
Iteration 18: Best Grid Loss = 5.2025691754e+00
Iteration 19: Best Grid Loss = 5.2025691754e+00
Iteration 20: Best Grid Loss = 5.2025691754e+00
Iteration 21: Best Grid Loss = 5.2025691754e+00
Iteration 22: Best Grid Loss = 5.2025691754e+00
Iteration 23: Best Grid Loss = 5.2025691754e+00
Iteration 24: Best Grid Loss = 5.2025691754e+00
Iteration 25: Best Grid Loss = 5.2025691754e+00
Iteration 26: Best Grid Loss = 5.2025691754e+00
Iteration 27: Best Grid Loss = 5.2025691754e+00
Iteration 28: Best Grid Loss = 5.2025691754e+00
Iteration 29: Best Grid Loss = 5.2025691754e+00
Iteration 30: Best Grid Loss = 5.2025691754e+00
Iteration 31: Best Grid Loss = 5.2025691754e+00
Iteration 32: Best Grid Loss = 5.2025691754e+00
Iteration 33: Best Grid Loss = 5.2025691754e+00
Iteration 34: Best Grid Loss = 5.2025691754e+00
Iteration 35: Best Grid Loss = 5.2025691754e+00
Iteration 36: Best Grid Loss = 5.2025691754e+00
Iteration 37: Best Grid Loss = 5.2025691754e+00
Iteration 38: Best Grid Loss = 5.2025691754e+00
Iteration 39: Best Grid Loss = 5.2025691754e+00
Iteration 40: Best Grid Loss = 5.2025691754e+00
Iteration 41: Best Grid Loss = 5.2025691754e+00
Iteration 42: Best Grid Loss = 5.2025691754e+00
Iteration 43: Best Grid Loss = 5.2025691754e+00
Iteration 44: Best Grid Loss = 5.2025691754e+00
```

Iteration 45: Best Grid Loss = 5.2025691754e+00
Iteration 46: Best Grid Loss = 5.2025691754e+00
Iteration 47: Best Grid Loss = 5.2025691754e+00
Iteration 48: Best Grid Loss = 5.2025691754e+00
Iteration 49: Best Grid Loss = 5.2025691754e+00
Iteration 50: Best Grid Loss = 5.2025691754e+00
Iteration 51: Best Grid Loss = 5.2025691754e+00
Iteration 52: Best Grid Loss = 5.2025691754e+00
Iteration 53: Best Grid Loss = 5.2025691754e+00
Iteration 54: Best Grid Loss = 5.2025691754e+00
Iteration 55: Best Grid Loss = 5.2025691754e+00
Iteration 56: Best Grid Loss = 5.2025691754e+00
Iteration 57: Best Grid Loss = 5.2025691754e+00
Iteration 58: Best Grid Loss = 5.2025691754e+00
Iteration 59: Best Grid Loss = 5.2025691754e+00
Iteration 60: Best Grid Loss = 5.2025691754e+00
Iteration 61: Best Grid Loss = 5.2025691754e+00
Iteration 62: Best Grid Loss = 5.2025691754e+00
Iteration 63: Best Grid Loss = 5.2025691754e+00
Iteration 64: Best Grid Loss = 5.2025691754e+00
Iteration 65: Best Grid Loss = 5.2025691754e+00
Iteration 66: Best Grid Loss = 5.2025691754e+00
Iteration 67: Best Grid Loss = 5.2025691754e+00
Iteration 68: Best Grid Loss = 5.2025691754e+00
Iteration 69: Best Grid Loss = 5.2025691754e+00
Iteration 70: Best Grid Loss = 5.2025691754e+00
Iteration 71: Best Grid Loss = 5.2025691754e+00
Iteration 72: Best Grid Loss = 5.2025691754e+00
Iteration 73: Best Grid Loss = 5.2025691754e+00
Iteration 74: Best Grid Loss = 5.2025691754e+00
Iteration 75: Best Grid Loss = 5.2025691754e+00
Iteration 76: Best Grid Loss = 5.2025691754e+00
Iteration 77: Best Grid Loss = 5.2025691754e+00
Iteration 78: Best Grid Loss = 5.2025691754e+00
Iteration 79: Best Grid Loss = 5.2025691754e+00
Iteration 80: Best Grid Loss = 5.2025691754e+00
Iteration 81: Best Grid Loss = 5.2025691754e+00
Iteration 82: Best Grid Loss = 5.2025691754e+00
Iteration 83: Best Grid Loss = 5.2025691754e+00
Iteration 84: Best Grid Loss = 5.2025691754e+00
Iteration 85: Best Grid Loss = 5.2025691754e+00
Iteration 86: Best Grid Loss = 5.2025691754e+00
Iteration 87: Best Grid Loss = 5.2025691754e+00
Iteration 88: Best Grid Loss = 5.2025691754e+00
Iteration 89: Best Grid Loss = 5.2025691754e+00
Iteration 90: Best Grid Loss = 5.2025691754e+00

Iteration 91: Best Grid Loss = 5.2025691754e+00
Iteration 92: Best Grid Loss = 5.2025691754e+00
Iteration 93: Best Grid Loss = 5.2025691754e+00
Iteration 94: Best Grid Loss = 5.2025691754e+00
Iteration 95: Best Grid Loss = 5.2025691754e+00
Iteration 96: Best Grid Loss = 5.2025691754e+00
Iteration 97: Best Grid Loss = 5.2025691754e+00
Iteration 98: Best Grid Loss = 5.2025691754e+00
Iteration 99: Best Grid Loss = 5.2025691754e+00
Iteration 100: Best Grid Loss = 5.2025691754e+00

Optimal Electrical Grid Configuration:
[-1.9760426   0.07752152 -0.00891342]

Minimum Power Loss (Objective Value):
5.202569175412648

Program 6:
Grey wolf optimization algorithm

Page

Grey wolf optimization

Initialize population of grey wolves
x_i ($i = 1, 2, \ldots, N$)

Initialize a, A, C
Evaluate fitness of each wolf

Identify alpha (best), beta (2nd best),
detla (3rd best)

t = 0
while t < max_iterations
a = 2 * (1 - t / max_iterations)

for each wolf i = 1 ... N:
r1 = rand (0,1)

1P

Foreground
&
Background image diffunation

A1 = 2 * a * r1 - a
C1 = 2 * r2
D_alpha = abs(C1 * x_alpha - x_i)
X1 = x_alpha - A1 * D_alpha

r1 = rand(0, 1);

r2 = rand (0, 1)
A2 = 2 * a * r1 - a
C2 = 2 * r2
D_beta = abs(C2 * x_beta - x_i)
x2 = x_beta - A2 * D_beta

r1 = rand (0, 1)
r2 = rand (0, 1)
A3 = 2 * a * r1 - a
C3 = 2 * r2
D_delta = abs(C3 * x_delta - x_i)
x3 = x_delta - A3 * D_delta

x_i_new = (x1 + x2 + x3) / 3
x_i_new = clip_to_bounds (x_i_new,
lower_bound, upper_bound)

end for

Evaluate fitness(x_i) for all wolves
(using x_i_new)

update alpha, beta, delta based
on new fitness values
t = t + 1

Return alpha (best solution found)

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files

print("       Upload an image (JPG/PNG):")
uploaded = files.upload()
filename = list(uploaded.keys())[0]
img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
img = cv2.resize(img, (256, 256))

def         fitness(threshold):
    threshold = int(threshold)
    foreground = img[img >= threshold]
    background = img[img < threshold]
    if len(foreground) == 0 or len(background) == 0:
        return 0
    w0 = len(background) / img.size
    w1 = len(foreground) / img.size
    u0 = np.mean(background)
    u1    =    np.mean(foreground)
    return w0 * w1 * (u0 - u1)**2

def grey_wolf_optimization(fitness_func, lb, ub, dim=1, n_wolves=20, iterations=50):
    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')
    positions = np.random.uniform(lb, ub, (n_wolves, dim))

    for t in range(iterations):
        a = 2 - t * (2 / iterations)
        for i in range(n_wolves):
            score = fitness_func(positions[i])
            if score > alpha_score:
                alpha_score = score
                alpha_pos = positions[i].copy()
            elif score > beta_score:
                beta_score = score
                beta_pos = positions[i].copy()
            elif score > delta_score:
                delta_score = score
```

```
                delta_pos = positions[i].copy()
        for i in range(n_wolves):
            for j 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[j] - positions[i][j])
                X1 = alpha_pos[j] - 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[j] - positions[i][j])
                X2 = beta_pos[j] - 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[j] - positions[i][j])
                X3 = delta_pos[j] - A3 * D_delta

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

        positions = np.clip(positions, lb, ub)
        print(f"Iteration {t+1}: Best Fitness = {alpha_score:.5f}, Threshold = {alpha_pos[0]:.2f}")

    return int(alpha_pos[0])

best_threshold = grey_wolf_optimization(fitness, lb=0, ub=255, dim=1, n_wolves=20,
iterations=50)
print(f"\nOptimal Threshold Found by GWO: {best_threshold}")

_, foreground = cv2.threshold(img, best_threshold, 255, cv2.THRESH_BINARY)
background = cv2.bitwise_not(foreground)

plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.imshow(img, cmap='gray')
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(foreground, cmap='gray')
plt.title("Foreground (GWO)")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(background, cmap='gray')
plt.title("Background")
```

```
plt.axis('off')

plt.show()
```

Output:

Upload an image (JPG/PNG):

```
Screenshot 2025-11-13 at 21.35.18.png(image/png) - 902667 bytes, last modified: n/a - 100% done
 Saving Screenshot 2025-11-13 at 21.35.18.png to Screenshot 2025-11-13 at 21.35.18.png
Iteration 1: Best Fitness = 354.43253, Threshold = 137.21
Iteration 2: Best Fitness = 379.28291, Threshold = 118.82
Iteration 3: Best Fitness = 380.05984, Threshold = 121.81
Iteration 4: Best Fitness = 380.05984, Threshold = 121.81
Iteration 5: Best Fitness = 380.05984, Threshold = 121.81
Iteration 6: Best Fitness = 380.05984, Threshold = 121.81
Iteration 7: Best Fitness = 380.05984, Threshold = 121.81
Iteration 8: Best Fitness = 380.05984, Threshold = 121.81
Iteration 9: Best Fitness = 380.05984, Threshold = 121.81
Iteration 10: Best Fitness = 380.05984, Threshold = 121.81
Iteration 11: Best Fitness = 380.05984, Threshold = 121.81
Iteration 12: Best Fitness = 380.05984, Threshold = 121.81
Iteration 13: Best Fitness = 380.05984, Threshold = 121.81
Iteration 14: Best Fitness = 380.05984, Threshold = 121.81
Iteration 15: Best Fitness = 380.05984, Threshold = 121.81
Iteration 16: Best Fitness = 380.05984, Threshold = 121.81
Iteration 17: Best Fitness = 380.05984, Threshold = 121.81
Iteration 18: Best Fitness = 380.05984, Threshold = 121.81
Iteration 19: Best Fitness = 380.05984, Threshold = 121.81
Iteration 20: Best Fitness = 380.05984, Threshold = 121.81
Iteration 21: Best Fitness = 380.05984, Threshold = 121.81
Iteration 22: Best Fitness = 380.05984, Threshold = 121.81
Iteration 23: Best Fitness = 380.05984, Threshold = 121.81
/tmp/ipython-input-1969087285.py:21: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
  threshold = int(threshold)
Iteration 24: Best Fitness = 380.05984, Threshold = 121.81
Iteration 25: Best Fitness = 380.05984, Threshold = 121.81
Iteration 26: Best Fitness = 380.05984, Threshold = 121.81
Iteration 27: Best Fitness = 380.05984, Threshold = 121.81
Iteration 28: Best Fitness = 380.05984, Threshold = 121.81
Iteration 29: Best Fitness = 380.05984, Threshold = 121.81
```

```
Optimal Threshold Found by GWO: 121
```

Program 7:
Paralell cellular algorithm

Parallel cellular algorithm

Begin

Step1: Define objective function $f(x)$

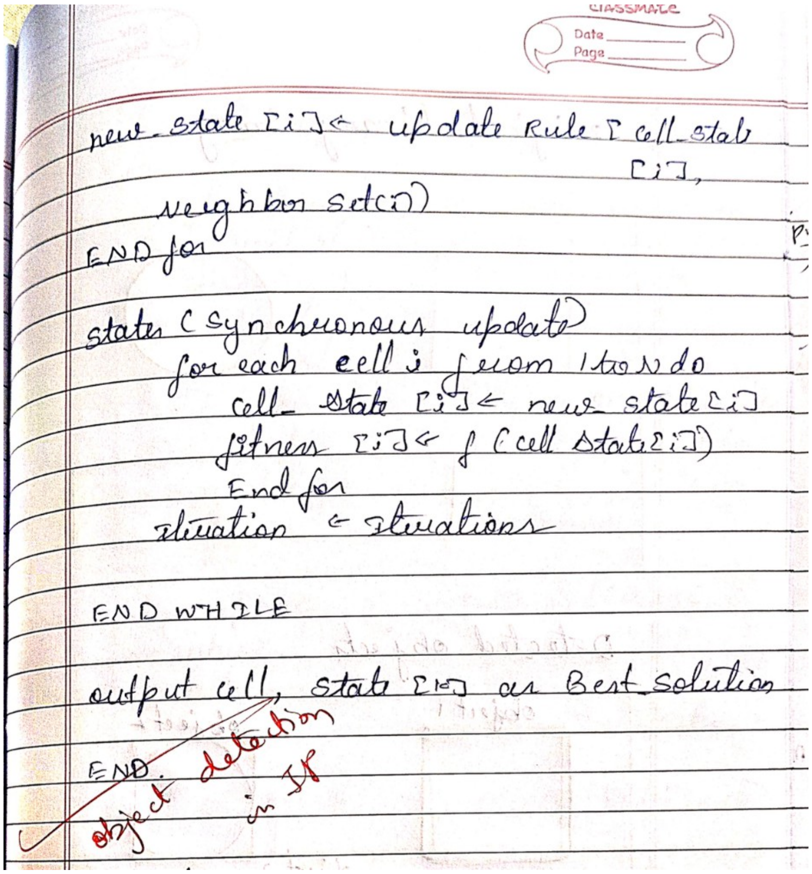
Step 2: Initialize parameters
Set number of cells ← N
Set max_iterations ← T
Define grid Structure and neighbourhood pattern

Step 3: Initialize population
for each cell i from 1 to N do
Initialize cell State [i] with a random
Solution
END FOR

Step4: Evaluate fitness
For each cell i from 1 to N do
fitness [i] ← f [cell State[i]
END for

Iteration ← 0

for each cell i from 1 to N do i
PARALLEL
Identity NeighborSet(i)



new_state [i] ← update Rule [cell_state
[i],
neighbor set[i])
END for

states (Synchronous update)
for each cell i from 1 to N do
cell_ State [i] ← new_ state[i]
fitness [i] ← f (cell State[i])
End for
iteration ← iterations

END WHILE

output cell, state [k] as Best solution

END.

object detection
in IP

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files

# Upload
print("    Upload an image:")
uploaded = files.upload()
filename = list(uploaded.keys())[0]

# Preprocess
img = cv2.imread(filename, 0)
img = cv2.resize(img, (256, 256))
blur = cv2.GaussianBlur(img, (5, 5), 0)
binary = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.OTSU)[1]
grid = (binary // 255).astype(np.uint8)
```

```
# PCA update
def pca_update(grid):
    kernel = np.array([[1,1,1],[1,0,1],[1,1,1]])
    neighbors = cv2.filter2D(grid, -1, kernel)
    return np.where(neighbors >= 4, 1, 0)

# PCA iterations
for _ in range(10):
    grid = pca_update(grid)

# Detect objects
num_labels, labels, stats, _ = cv2.connectedComponentsWithStats((grid*255).astype("uint8"), 8)

detected = np.zeros_like(grid)
for i in range(1, num_labels):
    if stats[i][4] > 30:
        detected[labels == i] = 255

# Display
plt.figure(figsize=(12,4))
plt.subplot(131); plt.imshow(img,cmap='gray'); plt.title("Original"); plt.axis('off')
plt.subplot(132); plt.imshow(grid,cmap='gray'); plt.title("After PCA Evolution"); plt.axis('off')
plt.subplot(133); plt.imshow(detected,cmap='gray'); plt.title("Detected Objects"); plt.axis('off')
plt.show()
```

Output: