

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems

(23CS5BSBIS) *Submitted by*

Sinchana(1BM23CS329)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Jan-2026

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **Sinchana(1BM23CS329)**, 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.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	29/08/25	Genetic Algorithms	1
2	29/08/25	Particle Swarm Optimization	7
3	10/10/25	Ant Colony Optimization	11
4	17/10/25	Cuckoo Search	15
5	17/10/25	Grey Wolf Optimizer	20
6	07/11/25	Parallel Cellular Algorithms	25
7	29/08/25	Gene Expression Algorithm	28

Github Link:

[sinchana329/BISLAB](https://github.com/sinchana329/BISLAB)

22/08/25

Bafna Gold

Date:

Page:

Algorithms

Application of algorithms

① Genetic algorithm for optimization problem.

1. Engineering design & optimization
2. Artificial intelligence & machine learning
3. Finance & economics
4. robotics

② particle swarm optimization for function optimization

PSO is a metaheuristic optimization algorithm inspired by the social behaviour of bird flocking or fish schooling.

Application

- ① training neural networks
- ② parameter optimization
- ③ Image processing

③ Ant colony optimization for the travelling salesman

problems

ACO is widely applied to the travelling salesman problem because it mimics natural foraging behaviour to find optimal paths.

Application

- ① Dynamic environment
- ② Combinational optimization
- ③ Improved algorithm.

④ cuckoo search (CS)

It is applied in the fields like engineering optimization, data science, CS to solve

Complex optimization problem by mimicking
beehive

② Wind energy forecasting

③ optimal power flow

④ structural & Design optimization.

⑤ Grey wolf optimizer (GWO)

applied across diverse field including
machine learning, engineering power system

AP

ML

Robotics

& UAVs

Bioinformatics & medicine.

⑥ parallel cellular algorithm & program

In various field due to their inherent
parallelism & ability to model complex sys.

A

modelling biological system.

Genetic algorithm

Image processing & Cryptograph.

⑦ optimization via Gene Expression algorithm

It is a evolutionary algorithm that combines
aspects of genetic algorithm.

A

time series prediction

optimization of engineering design

Control system design

Program 1

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

Algorithm:

29/08/21

Bafna Gold

Date: Page:

Genetic Algorithm : 5 main phases - initialization

fitness assignment

Selection

Crossover

termination

steps

1. selecting encoding techniques 0 to 31

2. Select the initial population.

SI no	Initial population	X Value	fitness $f(x)=x^2$	probability $f(x)/\sum f(x)$	prob	Expect	Actual
1	01100	12	144	0.124	12.4	0.42	1
2	11001	25	625	0.541	54	2.1	2
3	00101	5	25	0.0216	2	0.085	0
4	10011	9	361	0.3125	3	1.25	1

Sum 1155

avg 288.75

max 625

(3) select mating pool

string no	mating pool	crossover	offspring off	x value	fitness $f(x)=x^2$
1	01100		01101	13	169
2	11001	4	11000	24	576
3	00101	1	01011	27	729
4	10011	2	10001	17	289

Sum 0-1

avg 1-0

max

5) mutation:

Crossover: Random 4 and 3

max value $\rightarrow 29$

5) mutation:

String no	offspring after crossover	mutation chromosome for offspring	offspring after mutation	x value	fitness $f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11001	100000	11001	27	729
4	10001	00101	10100	20	400

$$\text{Sum} = 2546$$

$$\text{avg} = 630.5$$

$$\text{max} = 841$$

2

Code:

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

# 1. Define the function to optimize
def func(x):
    return x * x # Adjusted for new range

# 2. Parameters
POP_SIZE = 50
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.8
GENERATIONS = 10
CHROMOSOME_LENGTH = 10 # Now allows x in [0, 1023]

# 3. Decode chromosome to integer
def decode(chromosome):
    return int("".join(str(bit) for bit in chromosome), 2)

# 4. Create initial population
def create_population():
    return np.random.randint(2, size=(POP_SIZE, CHROMOSOME_LENGTH))

# 5. Evaluate fitness
def evaluate_fitness(population):
    decoded = np.array([decode(chrom) for chrom in population])
    fitness = func(decoded)
    return fitness

# 6. Selection (Roulette Wheel)
def select(population, fitness):
    min_fitness = np.min(fitness)
    if min_fitness < 0:
        fitness = fitness - min_fitness + 1e-6
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    indices = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, p=probabilities)
    return population[indices]

# 7. Crossover (Single-point)
def crossover(population):
    new_population = []
    for i in range(0, POP_SIZE, 2):
        parent1 = population[i]
```

```

parent2 = population[(i + 1) % POP_SIZE]
if np.random.rand() < CROSSOVER_RATE:
    point = np.random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = np.concatenate([parent1[:point], parent2[point:]])
    child2 = np.concatenate([parent2[:point], parent1[point:]])
    new_population.extend([child1, child2])
else:
    new_population.extend([parent1, parent2])
return np.array(new_population)

# 8. Mutation
def mutate(population):
    for i in range(POP_SIZE):
        for j in range(CHROMOSOME_LENGTH):
            if np.random.rand() < MUTATION_RATE:
                population[i, j] = 1 - population[i, j]
    return population

# 9. Main GA loop
def genetic_algorithm():
    population = create_population()
    best_solution = None
    best_fitness = -np.inf
    best_fitness_list = []

    for generation in range(GENERATIONS):
        fitness = evaluate_fitness(population)
        max_idx = np.argmax(fitness)
        current_best_fitness = fitness[max_idx]
        current_best_solution = decode(population[max_idx])

        # Update global best
        print(f"Generation {generation + 1}: x = {current_best_solution}, f(x) = {current_best_fitness:.4f}")

        best_fitness_list.append(current_best_fitness)

    # Elitism
    elite = population[max_idx].copy()

    # GA steps
    population = select(population, fitness)
    population = crossover(population)
    population = mutate(population)

```

```

# Preserve elite
population[np.random.randint(POP_SIZE)] = elite

# Plot fitness over generations
plt.figure(figsize=(10, 5))
plt.plot(range(1, GENERATIONS + 1), best_fitness_list, label='Best Fitness')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.title('Best Fitness Over Generations')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

return current_best_solution, current_best_fitness

# Run the GA
best_x, best_val = genetic_algorithm()
print(f"\nFinal Best Solution: x = {best_x}, f(x) = {best_val:.4f}")

```

Output:

```

Generation 1: x = 991, f(x) = 982081.0000
Generation 2: x = 991, f(x) = 982081.0000
Generation 3: x = 991, f(x) = 982081.0000
Generation 4: x = 1008, f(x) = 1016064.0000
Generation 5: x = 1008, f(x) = 1016064.0000
Generation 6: x = 1008, f(x) = 1016064.0000
Generation 7: x = 1008, f(x) = 1016064.0000
Generation 8: x = 1012, f(x) = 1024144.0000
Generation 9: x = 1012, f(x) = 1024144.0000
Generation 10: x = 1014, f(x) = 1028196.0000

```

Final Best Solution: x = 1014, f(x) = 1028196.0000

Program 2

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

Algorithm:

lab 3

particle swarm optimization (PSO)

Pseudo code

- 1 $p = \text{particle initialization}()$
- 2 for $i = 1$ to max
- 3 for each particle p in P do
 $f_p = f(p)$
- 4 if f_p is better than $f(p_{\text{best}})$
 $p_{\text{best}} = p$
- end
- end
- 5 $g_{\text{best}} = \text{best p trap}$
- 6 for each particle p in P do
 $v_i^{t+1} = v_i^t + c_1 v_i^t (p_{\text{best}}^t - p_i^t) + c_2 v_i^t (g_{\text{best}} - p_i^t)$
- 7 $p_i^{t+1} = p_i^t + v_i^{t+1}$
- end
- end

eg: Iteration 1

$$f(x, y) = x^2 + y^2$$

$$(x, y) \text{ minimization } (w) = 0.3$$

Value of cognitive + social constants

$$c_1 = 2, c_2 = 2$$

Initial solution are set to 1000

$$p_i \text{ fitness value} = 1^2 + 1^2 = 2$$

particle no	initial pos	pos x	pos y	vector x	to y	best soln	Best pos x	y	fitness value
P1 P1	1	1	0	0	0	1000	-	-	2
P2 P2	-1	1	0	0	0	1000	-	-	2
P3 P3	0.5	-0.5	0	0	0	1000	-	-	0.5
P4 P4	1	-1	0	0	0	1000	-	-	2
P5	0.25	0.25	0	0	0	1000	-	-	0.125

Iteration 2.

pno	initial pos		velocity		best soln	Best pos		fitness value
	x	y	x	y		x	y	
P1	1	1	-0.25	-0.75	2	1	1	2
P2	-1	1	1.25	-0.75	2	-1	1	2
P3	0.5	-0.5	-0.25	0.75	0.5	0.5	-0.5	0.5
P4	1	-1	-0.75	1.25	2	1	-1	2
P5	0.25	0.25	0	0	0.125	0.25	0.25	0.125

Code:

```
import random
from math import sqrt
```

```
c1, c2 = 1, 1
```

```
def fitness(x):
    return -x**2 + 5*x + 20
```

```
def init():
    n = int(input("Enter no. of particles: "))
    v = [0 for i in range(n)]
    x = list(map(float, input("Enter positions of particles:").split()))
    p = x.copy()
    fp = [fitness(xi) for xi in x]
    return n, v, fp, p, x
```

```
def find(n, fp, p):
    max_fitness = float('-inf')
    pos = -1
    for i in range(n):
        if fp[i] > max_fitness:
            max_fitness = fp[i]
            pos = i
    return pos
```

```
def update(n, v, fp, p, x, max_pos):
    r1, r2 = sqrt(random.random()), sqrt(random.random())

    for i in range(n):
        v[i] = v[i] + c1 * r1 * (p[i] - x[i]) + c2 * r2 * (p[max_pos] - x[i])
        x[i] = x[i] + v[i]

    for i in range(n):
        fp[i] = fitness(x[i])
        if fp[i] > fitness(p[i]):
            p[i] = x[i]
```

```
def print_state(v, fp, p, x):
    print(f"
```

```

n, v, fp, p, x = init()
print_state(v, fp, p, x)
max_pos = find(n, fp, p)
gbest = p[max_pos]

while True:
    update(n, v, fp, p, x, max_pos)
    max_pos = find(n, fp, p)
    if fitness(gbest) == fitness(p[max_pos]):
        break
    print_state(v, fp, p, x)
    gbest = p[max_pos]

print(f'Global Best Solution: {gbest} with fitness: {fitness(gbest)}')

```

Output:

```

Enter no. of particles: 5
Enter positions of particles:1 -1 0.5 1 0.25

```

```

[1.0, -1.0, 0.5, 1.0, 0.25]
[1.0, -1.0, 0.5, 1.0, 0.25]
[0, 0, 0, 0, 0]
[24.0, 14.0, 22.25, 24.0, 21.1875]

```

Global Best Solution: 1.0 with fitness: 24.0

Program 3

The foraging behavior of ants has inspired the development of optimization algorithms that can solve

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

Algorithm:

11

lab 4

4. Ant colony optimization (ACO) for traveling salesman problem.

Pseudocode

```

Initialize pheromones
For iteration = 1 to max_iterations:
    For each ant:
        build a tour using pheromones and heuristic info
        calculate tour length
    End for
    return best tour

```

algorithm

```

Initialize  $t = 1$  to  $t_{max}$ 
For  $t = 1$  to  $t_{max}$ :
    For each ant  $k$ :

$$P_{ij}^k = \frac{[\tau_{ij}]^\alpha + [\eta_{ij}]^\beta}{\sum_{l \in N_i} ([\tau_{il}]^\alpha + [\eta_{il}]^\beta)}$$

        where  $\eta_{ij} = 1/d_{ij}$ 
        compute route length  $L_k$ 
    End for
    update pheromone:

$$\tau_{ij} = (1 - \rho) \tau_{ij} + \sum_k \Delta \tau_{ij}^k$$

    where  $\Delta \tau_{ij}^k = Q / L_k$  if ant  $k$  used edge  $i-j$ , else 0
Return best route

```

application: vehicle routing problem

output Attraction	best route	length	Phenomenon	edge 0 2 3
1	0 → 1 → 2 → 3 → 0	14	0.1214	0.05
2	0 → 3 → 2 → 1 → 0	14	0.06	0.1321
3	1 → 0 → 3 → 2 → 1	14	0.09	0.1

10/10/25

Code:

```
import numpy as np
import random

def initialize_pheromone(num_cities, initial_pheromone=1.0):
    return np.ones((num_cities, num_cities)) * initial_pheromone

def calculate_probabilities(pheromone, distances, visited, alpha=1,
beta=2):
    pheromone = np.copy(pheromone)
    pheromone[list(visited)] = 0 # zero out visited cities

    heuristic = 1 / (distances + 1e-10) # inverse of distance
    heuristic[list(visited)] = 0

    prob = (pheromone ** alpha) * (heuristic ** beta)
    total = np.sum(prob)
    if total == 0:
        # If no options (all visited), choose randomly among unvisited
        choices = [i for i in range(len(distances)) if i not in visited]
        return choices, None
    prob = prob / total
    return range(len(distances)), prob

def select_next_city(probabilities, cities):
    if probabilities is None:
        return random.choice(cities)
    return np.random.choice(cities, p=probabilities)

def path_length(path, distances):
    length = 0
    for i in range(len(path)):
        length += distances[path[i-1]][path[i]]
    return length

def ant_colony_optimization(distances, n_ants=5, n_iterations=50, decay=0.5, alpha=1,
beta=2):
    num_cities = len(distances)
    pheromone = initialize_pheromone(num_cities)
    best_path = None
    best_length = float('inf')

    for iteration in range(n_iterations):
        all_paths = []
        for _ in range(n_ants):
            path = [0] # start at city 0
            visited = set(path)
```

```

for _ in range(num_cities - 1):
    current_city = path[-1]
    cities, probabilities = calculate_probabilities(pheromone[current_city], distances[current_city],
    visited, alpha, beta)
    next_city = select_next_city(probabilities, cities)
    path.append(next_city)
    visited.add(next_city)

length = path_length(path, distances)
all_paths.append((path, length))

if length < best_length:
    best_length = length
    best_path = path

# Evaporate pheromone
pheromone *= (1 - decay)

# Deposit pheromone proportional to path quality
for path, length in all_paths:
    deposit = 1 / length
    for i in range(len(path)):
        pheromone[path[i-1]][path[i]] += deposit

return best_path, best_length

# Example usage
if __name__ == "__main__":
    distances = np.array([
        [np.inf, 2, 2, 5, 7],
        [2, np.inf, 4, 8, 2],
        [2, 4, np.inf, 1, 3],
        [5, 8, 1, np.inf, 2],
        [7, 2, 3, 2, np.inf]
    ])

    best_path, best_length = ant_colony_optimization(distances)
    print(f"Best path: {[int(city) for city in best_path]} with length: {best_length:.2f}")

```

Output:

Best path: [0, 2, 3, 4, 1] with length: 9.00

Program 4

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:

Lab 5

Cuckoo Searching Algorithm

- ① set the initial value of the host nest size $p.c(0,1)$ & maximum number of iteration max.
- ② set $t=0$ & counter initialization
3. for $(i=1; i \leq n)$ do
 4. generate initial population or n host x_i
 5. evaluate fitness function $f(x_i, t)$
 6. end for
- 7 generate
 - evaluate fitness function
 - Choose a host x_i among n
 - replace the solution x_i with the solution x_{i+1}
 - end if.

Application.

Q. minimize $f(x) = x^2$

global minimum at $x=0$ $f(0)=0$

step 1

$$x_1 = 4$$

$$x_2 = -3$$

$$x_3 = 6$$

$$f(x_1) = 16$$

$$f(x_2) = 9$$

$$f(x_3) = 36$$

best soln

$$x_1 = 3$$

fitness = 9.

step 2 :

$$x_{new} = x_{old} + Levy(x)$$

$$L=1$$

$$x_{1, new} = 4 + (-2) = 2$$

$$f(2) = 4$$

$$x_{2, new} = -3 + (7) = -2$$

$$f(-2) = 4$$

$$x_{3, new} = -6 + (4) = -2$$

best soly $x=2$ $f(2)=4$.

step 3

evaluate & select best.

Nest 1 $= 2$ (fitness 4)

Nest 2 $= -2$ (fitness 4)

Nest 3 $= 2$ (fitness 4)

step 4

$p_a = 0.25$

$x_2 = -1 \rightarrow f(-1) = 1$

best solution now $x = -1$, $f(x) = 1$

step 5

$x = -0.5 \rightarrow f(x) = 0.25$

evary converges near $x=0$ $f(x)=0$

Code:

```
import numpy as np
import math

def knapsack_fitness(solution, values, weights, capacity):
    """Calculate fitness: total value if weight within capacity, else zero."""
    total_weight = np.sum(solution * weights)
    if total_weight > capacity:
        return 0 # Penalize overweight solutions
    return np.sum(solution * values)

def levy_flight(Lambda, size):
    """Generate Levy flight steps."""
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) / (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.normal(0, sigma, size)
    v = np.random.normal(0, 1, size)
    step = u / (np.abs(v) ** (1 / Lambda))
    return step

def sigmoid(x):
    """Sigmoid function for mapping continuous to probability."""
    return 1 / (1 + np.exp(-x))

def cuckoo_search_knapsack(values, weights, capacity, n_nests=25, max_iter=100, pa=0.25):
    """
    Cuckoo Search for 0/1 Knapsack Problem.

    Args:
    values: numpy array of item values
    weights: numpy array of item weights
    capacity: max capacity of knapsack
    n_nests: number of nests (population size)
    max_iter: max iterations
    pa: probability of abandoning nests

    Returns:
    best_solution: binary numpy array with item selection
    best_fitness: total value of best_solution
    """
```

"""

```
n_items = len(values)
nests = np.random.randint(0, 2, size=(n_nests, n_items))
fitness = np.array([knapsack_fitness(n, values, weights, capacity) for n in nests])

best_idx = np.argmax(fitness)
best_solution = nests[best_idx].copy()
best_fitness = fitness[best_idx]
```

```
Lambda = 1.5 # Levy flight exponent
```

```
for iteration in range(max_iter):
    for i in range(n_nests):
        step = levy_flight(Lambda, n_items)
        current = nests[i].astype(float)
        new_solution_cont = current + step
        probs = sigmoid(new_solution_cont)
        new_solution_bin = (probs > 0.5).astype(int)
```

```
new_fitness = knapsack_fitness(new_solution_bin, values, weights, capacity)
```

```
# Greedy selection
```

```
if new_fitness > fitness[i]:
    nests[i] = new_solution_bin
    fitness[i] = new_fitness
```

```
if new_fitness > best_fitness:
    best_fitness = new_fitness
    best_solution = new_solution_bin.copy()
```

```
# Abandon worst nests with probability pa
```

```
n_abandon = int(pa * n_nests)
if n_abandon > 0:
    abandon_indices = np.random.choice(n_nests, n_abandon, replace=False)
    for idx in abandon_indices:
        nests[idx] = np.random.randint(0, 2, n_items)
        fitness[idx] = knapsack_fitness(nests[idx], values, weights, capacity)
```

```
# Update global best after abandonment
```

```

current_best_idx = np.argmax(fitness)
if fitness[current_best_idx] > best_fitness:
    best_fitness = fitness[current_best_idx]

best_solution = nests[current_best_idx].copy()

# Print progress: every 10 iterations and first iteration
if iteration == 0 or (iteration + 1) % 10 == 0:
    print(f"Iteration {iteration + 1}/{max_iter}, Best Fitness: {best_fitness}")

return best_solution, best_fitness

if __name__ == "__main__":
    # Example knapsack problem
    values = np.array([60, 100, 120, 80, 30])
    weights = np.array([10, 20, 30, 40, 50])
    capacity = 100

    best_sol, best_val = cuckoo_search_knapsack(values, weights, capacity, n_nests=30,
max_iter=100, pa=0.25)

    print("\nBest solution found:")
    print(best_sol)
    print("Total value:", best_val)
    print("Total weight:", np.sum(best_sol * weights))

```

Output:

```

Iteration 1/100, Best Fitness: 360
Iteration 10/100, Best Fitness: 360
Iteration 20/100, Best Fitness: 360
Iteration 30/100, Best Fitness: 360
Iteration 40/100, Best Fitness: 360
Iteration 50/100, Best Fitness: 360
Iteration 60/100, Best Fitness: 360
Iteration 70/100, Best Fitness: 360
Iteration 80/100, Best Fitness: 360
Iteration 90/100, Best Fitness: 360
Iteration 100/100, Best Fitness: 360

```

Best solution found:

[1 1 1 1 0]

Total value: 360

Total weight: 100

19

Program 5

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:

6

Grey wolf optimizer (GWO)

Algorithm.

1. Initialize population $x_i, i=1, 2, 3, \dots, N$ randomly
2. evaluate fitness of each x_i
3. identify α (best), β (second best) & δ (third best).

4. while iteration $<$ max-iter :

for each wolf x_i :

update coefficients A & C :

$$D\alpha = |C_1 * x_\alpha - x_i|$$

$$D\beta = |C_2 * x_\beta - x_i|$$

$$D\delta = |C_3 * x_\delta - x_i|$$

$$x_1 = x_i - A_1 * D\alpha$$

$$x_2 = x_i - A_2 * D\beta$$

$$x_3 = x_i - A_3 * D\delta$$

$$x_i(\text{new}) = (x_1 + x_2 + x_3) / 3$$

End for

update fitness and α, β, δ

Decrease a linearly from 2 to 0

End while.

return x_α as best solution.

1. Example

$$f(x) = x^2$$

$$x = 0$$

$$N = 4$$

$$\text{dim} = 1$$

$$\text{max} = 5$$

Step 1 :

lets randomly ini 4.

wolf	x_i	$f(x_i) = x_i^2$
1	-2.5	6.25
2	3.0	9.00
3	-1.0	1.00
4	2.0	4.00

Step 2 : α, β, δ

Ranks	wolf	position	fitness
α	3	-1.0	1.00
β	4	2.0	4.00
δ	1	-2.5	6.25

Generate random number

then calculate

$$A = 2 * r_1 - \alpha = 2 * 0.6 * 0.5 - 1.0 = -0.4$$

$$C_1 = 2 * r_2 = 1.4$$

compute distance to α, β, δ

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

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

$$D_\delta = |C_3 * x_\delta - x_i|$$

compute new position

$$x_1 = x_\alpha - A_1 * D_\alpha = -1.0$$

$$x_2 = x_\beta - A_2 * D_\beta = 2.0$$

$$x_3 = x_\delta - A_3 * D_\delta = -2.5$$

new best

$$\alpha = 0.014, (x=0.5, R=0.25)$$

$$\beta = 0.043 (x=1.0, R=1.0)$$

$$\delta = 0.014 (x=2.0, R=0.0)$$

Code:

```
import numpy as np

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

class GreyWolfOptimizer:
    def __init__(self, obj_func, n_wolves, dim, max_iter, lb=-10, ub=10):
        self.obj_func = obj_func
        self.n_wolves = n_wolves
        self.dim = dim
        self.max_iter = max_iter
        self.lb = lb
        self.ub = ub

        self.positions = np.random.uniform(self.lb, self.ub, (self.n_wolves, self.dim))

        self.alpha_pos = np.zeros(self.dim)
        self.alpha_score = float('inf')

        self.beta_pos = np.zeros(self.dim)
        self.beta_score = float('inf')

        self.delta_pos = np.zeros(self.dim)
        self.delta_score = float('inf')

    def optimize(self):
        for iter in range(self.max_iter):
            for i in range(self.n_wolves):
                self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)
                fitness = self.obj_func(self.positions[i])

                if fitness < self.alpha_score:
                    self.alpha_score = fitness
                    self.alpha_pos = self.positions[i].copy()
                elif fitness < self.beta_score:
                    self.beta_score = fitness
                    self.beta_pos = self.positions[i].copy()
                elif fitness < self.delta_score:
```

```

self.delta_score = fitness
self.delta_pos = self.positions[i].copy()

a = 2 - iter * (2 / self.max_iter)

for i in range(self.n_wolves):
    for j in range(self.dim):
        r1 = np.random.rand()
        r2 = np.random.rand()
        A1 = 2 * a * r1 - a
        C1 = 2 * r2
        D_alpha = abs(C1 * self.alpha_pos[j] - self.positions[i, j])
        X1 = self.alpha_pos[j] - A1 * D_alpha

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

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

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

    return self.alpha_pos, self.alpha_score

if __name__ == "__main__":
    # Take inputs from user
    n_wolves = int(input("Enter number of wolves: "))
    dim = int(input("Enter number of dimensions: "))
    max_iter = int(input("Enter max iterations: "))

```

```
gwo = GreyWolfOptimizer(obj_func=sphere, n_wolves=n_wolves, dim=dim, max_iter=max_iter)
best_pos, best_score = gwo.optimize()
```

23

```
print(f"Best Position: {best_pos}")
print(f"Best Score: {best_score}")
```

Output:

Enter number of wolves: 5

Enter number of dimensions: 4

Enter max iterations: 10

Best Position: [2.16579979 2.34635848 -1.00125355 0.49522175]

Best Score: 11.443840087181393

Program 6

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

Algorithm:

¶

Parallel Cellular Algorithm

Algorithm steps ..

1. Initialize the grid with starting states
2. for each time step $t=1$ to T
 - a) the parallel for all cells (i, j) :
Read neighbour states
Apply rule : $c_{new}(i, j) = f(c(i, j), \text{neighbors})$
 - b) synchronize and copy $c = c_{new}$
- 3) output the final grid

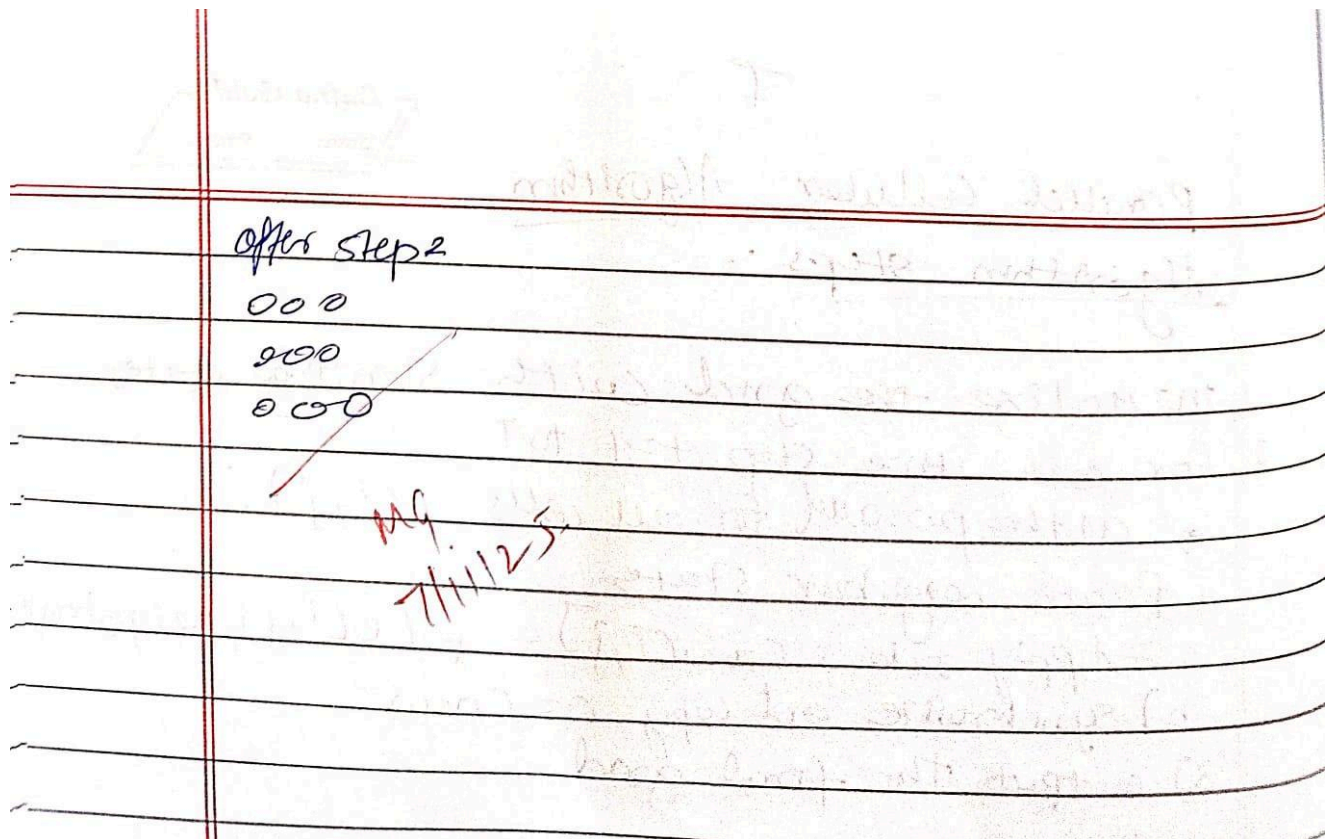
Application

Finding the majority value in a grid

initial grid

0 1 0
1 0 1
0 1 0

cell (i, j)	Current	# of neighbor 1s	new value
1,1	0	3	0
1,2	1	2	0
1,3	0	3	0
2,1	1	2	0
2,2	0	4	1
2,3	1	2	0
3,1	0	3	0
3,2	1	2	0
3,3	0	3	0



Code:

```
import numpy as np

# Initialize
grid = np.random.uniform(low=-10, high=10, size=(10,
10)) num_iterations = 100

# Define fitness function
def fitness_function(x):
    return x**2 - 4*x + 4

# Iterate
for iteration in range(num_iterations):
    new_grid = np.zeros_like(grid)
    for r in range(grid.shape[0]):
        for c in range(grid.shape[1]):
            neighbor_values = []
            for dr in [-1, 0, 1]:
                for dc in [-1, 0, 1]:
                    nr = (r + dr) % grid.shape[0]
```

```

nc = (c + dc) % grid.shape[1]
neighbor_values.append(grid[nr, nc])
# Update to average of neighbor values (per algorithm spec)
new_grid[r, c] = np.mean(neighbor_values)
grid = new_grid.copy()

# Find best solution
fitness_values = fitness_function(grid)
best_fitness_overall = np.min(fitness_values)
best_x_overall = grid[np.unravel_index(np.argmin(fitness_values), grid.shape)]

# Verbose Output
print("=== Parallel Cellular Algorithm Results ===")
print(f"Total iterations performed: {num_iterations}")
print(f"Best x value found: {best_x_overall:.6f}")
print(f"Corresponding fitness (minimum f(x)): {best_fitness_overall:.6f}")
print("Algorithm converged toward  $x \approx 2$ , where  $f(x) = 0$  (expected optimum).")

```

26

Output:

```

=== Parallel Cellular Algorithm Results ===
Total iterations performed: 100
Best x value found: 0.317779
Corresponding fitness (minimum f(x)): 2.829867
Algorithm converged toward  $x \approx 2$ , where  $f(x) = 0$  (expected optimum).

```

Program 7

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

solving complex optimization problems in various domains, including engineering, data analysis, and machine learning

Algorithm:

optimization via Genetic Expression algorithms.

$$f(x) = x^2$$

begin

define fitness: $\text{fitness}(x) = x^2$

Set parameters:

$p_c = 0.6$, $G = 5$,

$m = 0.1$, $c = 0.7$

Chromosome length = 5 bits.

Create initial population of 6 random 5 bit strings

for each generation (1 to 5):

calculate fitness for each individual

Select parents (e.g. tournament)

Apply crossover (probability 0.7)

Apply mutation (p 0.1 per bit)

replace population with new offspring

track best individual

Output best solution (decode binary to x & fitness)

End.

output:

G 1 : Best individual = 11011 ($x=27$), $F=729$

G 2 : BI = 11011, $x=27$, $F=729$

G 3 : BI = 11111, $x=31$, $F=961$

G 4 : BI = 11111, $x=31$, $F=961$

G 5 : BI = 11111, $x=31$

Code:

```
import random
import math

# Example:  $f(x) = x * \sin(10 * \pi * x) + 2$ 
def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2

POPULATION_SIZE = 6
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
GENERATIONS = 20
DOMAIN = (-1, 2)

def random_gene():
    return random.uniform(DOMAIN[0], DOMAIN[1])

def create_chromosome():
    return [random_gene() for _ in range(GENE_LENGTH)]

def initialize_population(size):
    return [create_chromosome() for _ in range(size)]

def evaluate_population(population):
    return [fitness_function(express_gene(chrom)) for chrom in population]

def express_gene(chromosome):
    return sum(chromosome) / len(chromosome)

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual
    return random.choice(population)
```

```

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENE_LENGTH - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1[:], parent2[:]

def mutate(chromosome):
    new_chromosome = []
    for gene in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome.append(random_gene())
        else:
            new_chromosome.append(gene)
    return new_chromosome

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
    best_solution = None
    best_fitness = float("-inf")

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, chrom in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = chrom[:]

    print(f"Generation {generation+1}: Best Fitness = {best_fitness:.4f}, Best x
    = {express_gene(best_solution):.4f}")

    new_population = []
    while len(new_population) < POPULATION_SIZE:
        parent1 = select(population, fitnesses)
        parent2 = select(population, fitnesses)
        offspring1, offspring2 = crossover(parent1, parent2)

```

```

offspring1 = mutate(offspring1)
offspring2 = mutate(offspring2)
new_population.extend([offspring1, offspring2])

population = new_population[:POPULATION_SIZE]

print("\nBest solution found:")
print(f'Genes: {best_solution}')
x_value = express_gene(best_solution)
print(f'x = {x_value:.4f}')
print(f'f(x) = {fitness_function(x_value):.4f}')

if __name__ == "__main__":
    gene_expression_algorithm()

```

output:

```

Generation 1: Best Fitness = 2.4347, Best x = 0.6245
Generation 2: Best Fitness = 2.4827, Best x = 0.6746
Generation 3: Best Fitness = 2.4827, Best x = 0.6746
Generation 4: Best Fitness = 2.4827, Best x = 0.6746
Generation 5: Best Fitness = 2.4827, Best x = 0.6746
Generation 6: Best Fitness = 2.4827, Best x = 0.6746
Generation 7: Best Fitness = 2.4827, Best x = 0.6746
Generation 8: Best Fitness = 2.4827, Best x = 0.6746
Generation 9: Best Fitness = 2.4827, Best x = 0.6746
Generation 10: Best Fitness = 2.4827, Best x = 0.6746
Generation 11: Best Fitness = 2.4827, Best x = 0.6746
Generation 12: Best Fitness = 2.4827, Best x = 0.6746
Generation 13: Best Fitness = 2.4827, Best x = 0.6746
Generation 14: Best Fitness = 2.4827, Best x = 0.6746
Generation 15: Best Fitness = 2.5073, Best x = 0.6728
Generation 16: Best Fitness = 2.5073, Best x = 0.6728
Generation 17: Best Fitness = 2.5073, Best x = 0.6728
Generation 18: Best Fitness = 2.5315, Best x = 0.6318
Generation 19: Best Fitness = 2.5315, Best x = 0.6318
Generation 20: Best Fitness = 2.5315, Best x = 0.6318

```

Best solution found:

Genes: [-0.9341914889787352, 1.582236333230926, 0.5195878130862375,

1.8961703080811958, 1.9026923622619076, -0.42906418830093207,
-0.5325680984167858, 1.8332299106440781, - 0.369575018958584, 0.8496492245933607]

$x = 0.6318$

$f(x) = 2.5315$