VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SIDHVIN VIDYADHAR BURLI(1BM21CS211)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Bio Inspired Systems (23CS5BSBIS)" carried out by **Sidhvin Vidyadhar Burli(1BM21CS211),** 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.

Syed Akram Assistant Professor Department of CSE, BMSCE	**Dr. Jyothi S Nayak** Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	03/10/24	GENETIC ALGORITHM	4-9
2	24/10/24	PARTICLE SWARM OPTIMIZATION	10-14
3	07/11/24	ANT COLONY OPTIMIZATION	15-21
4	21/11/24	CUCKOO SEARCH ALGORITHM	22-24
5	28/11/24	GREY WOLF OPTIMIZATION	25-28
6	12/12/24	PARALLEL CELLULAR ALGORITHM	29-34
7	12/12/24	GENE EXPRESSION ALGORITHM	35-39

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

Program 1

Genetic Algorithm for Optimization Problems

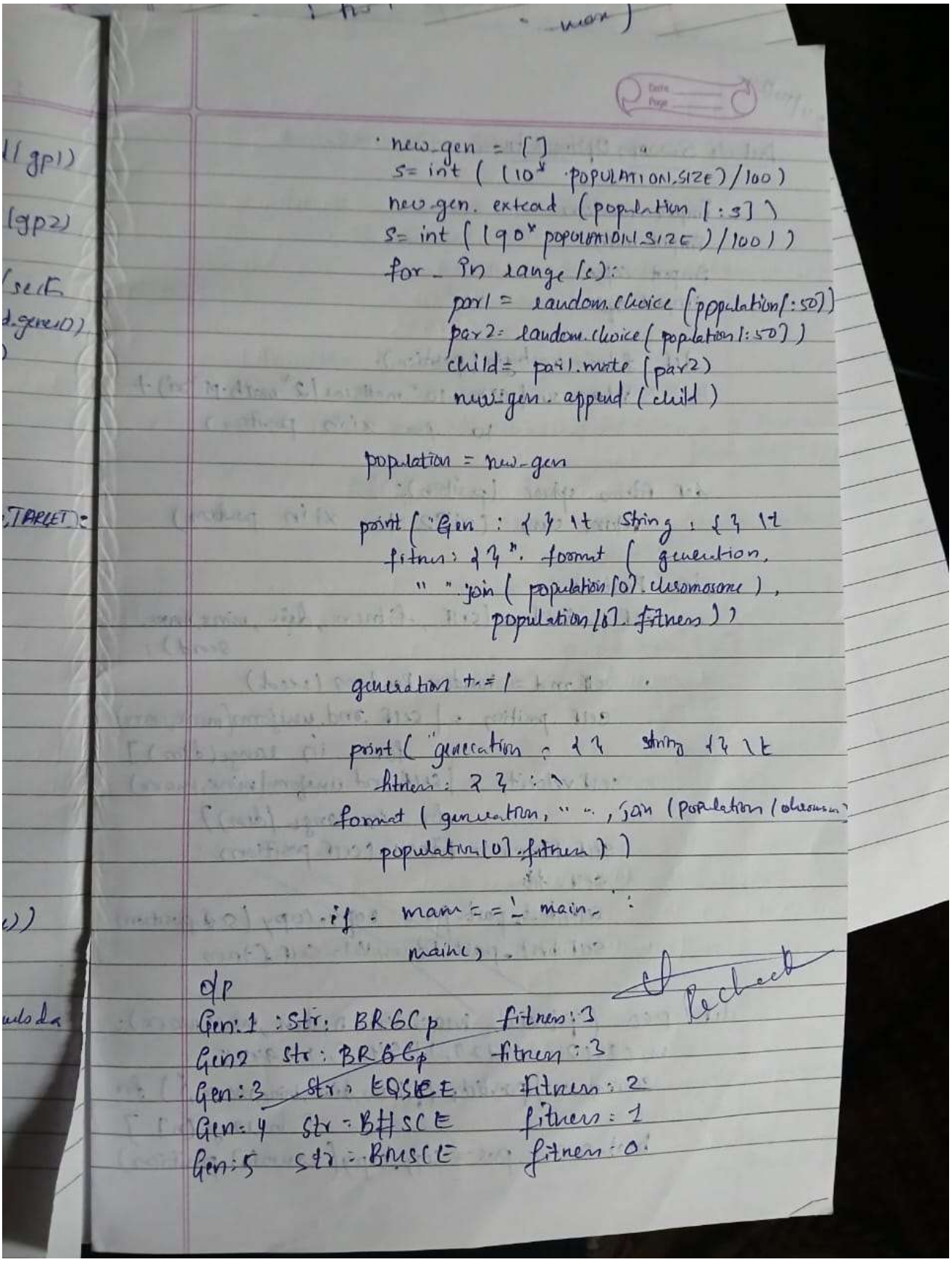
Algorithm:

1 Genetic Algorithms 3/10/24

```
import random

POPULATION_SIZE = 100

GENES = '''abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKL
        MNOPQRSTUVWXYZ 1234567890, .:! #&/()
        {[]} "'

TARGET = "bmsce"

class Individual (object):
    def __init__ (self, chromosome)
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes (self):
        global GENES
        gene = random.choice (GENES)
        return gene

    @classmethod
    def create_gnome (self):
        global TARGET
        gnome_len = len (TARGET)
        return [self.mutated_genes() for _ in range
                          (gnome_len) ]

    def mate (self, par2 ):
        child_chromosome = [ ]
        for gp1, gp2 in zip( self.chromosome, par2.
                                 chromosome ):
            prob = random. random()
```

```
            if prob < 0.45:
                child_chromosome.append(gp1)
            if prob< 0.9 :
                child_chromosome.append(gp2)
            else:
                child_chromosome.append(self.
                                        mutated_gene())
        return Individual(child_chromosome)

    def cal_fitness(self):
        global TARGET
        fitness = 0
        for gs,gt in zip( self.chromosome,TARGET):
            if gs!=gt : fitness+=1
        return fitness

def main():
    global POPULATION_SIZE
    generation =1
    found = False
    population = []

    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

    while not found:
        population = sorted(population , key=lambda
                            x: x.fitness)
        if population[0].fitness <=0 :
            found = true :
            break.
```

o/p
Gen
Gen
Gen
Gen
Gen:



```
new_gen = []
s = int ((10 * POPULATION_SIZE)/100)
new_gen.extend(population[:s])
s = int ((90 * POPULATION_SIZE)/100))
for _ in range(s):
    par1 = random.choice(population[:50])
    par2 = random.choice(population[:50])
    child = par1.mate(par2)
    new_gen.append(child)

population = new_gen

print("Gen : {} \t String : {} \t
    fitness : {}".format(generation,
    "".join(population[0].chromosome),
    population[0].fitness))

generation += 1

print("generation : {} \t string {} \t
    fitness : {} : \
format(generation, "".join(population[0].chromosome),
    population[0].fitness))

if __name__ == '__main__':
    main()
```

o/p

Gen: 1 : str: BRGCp fitness: 3

Gen2 str: BRGGp fitness: 3

Gen: 3 str: EQSKE fitness: 2

Gen: 4 str: BHSCE fitness: 1

Gen: 5 str: BMSCE fitness: 0

Recheck

Code:

```
import random

POPULATION_SIZE = 100

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

TARGET = "I love GeeksforGeeks"

class Individual(object):
    '''
    Class representing individual in population
    '''
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        '''
        create random genes for mutation
        '''
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        '''
        create chromosome or string of genes
        '''
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        '''
        Perform mating and produce new offspring
        '''

        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            prob = random.random()

            if prob < 0.45:
```

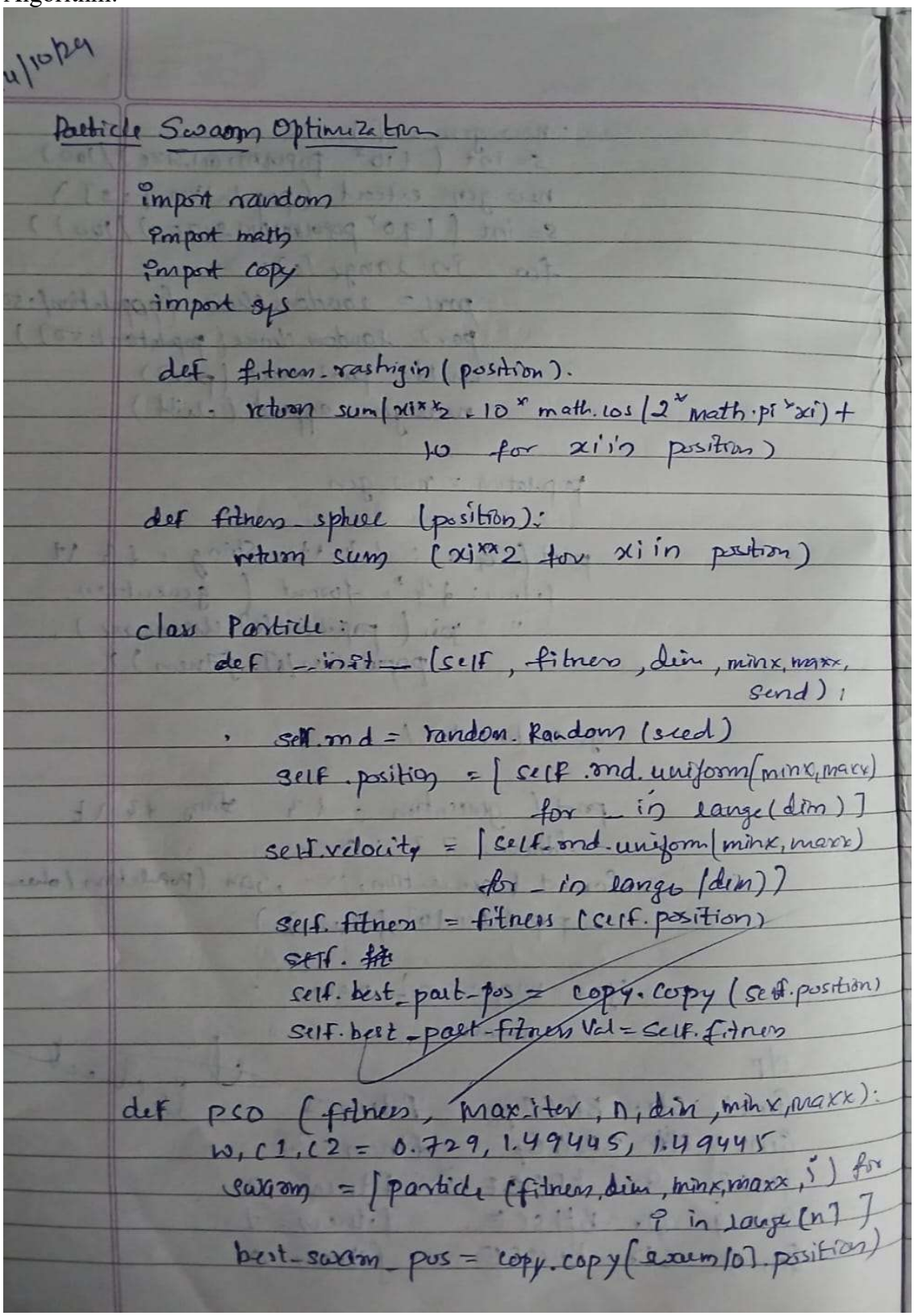
```
                        child_chromosome.append(gp1)

                    elif prob < 0.90:
                        child_chromosome.append(gp2)

                    else:
                        child_chromosome.append(self.mutated_genes())

                return Individual(child_chromosome)

        def cal_fitness(self):
                '''
                Calculate fitness score, it is the number of
                characters in string which differ from target
                string.
                '''
                global TARGET
                fitness = 0
                for gs, gt in zip(self.chromosome, TARGET):
                        if gs != gt: fitness+= 1
                return fitness

def main():
        global POPULATION_SIZE

        generation = 1

        found = False
        population = []

        for _ in range(POPULATION_SIZE):
                        gnome = Individual.create_gnome()
                        population.append(Individual(gnome))

        while not found:

                population = sorted(population, key = lambda x:x.fitness)

                if population[0].fitness <= 0:
                        found = True
                        break

                new_generation = []

                s = int((10*POPULATION_SIZE)/100)
                new_generation.extend(population[:s])
```

```
            s = int((90*POPULATION_SIZE)/100)
            for _ in range(s):
                parent1 = random.choice(population[:50])
                parent2 = random.choice(population[:50])
                child = parent1.mate(parent2)
                new_generation.append(child)

            population = new_generation

            print("Generation: {}\tString: {}\tFitness: {}".
                format(generation,
                "".join(population[0].chromosome),
                population[0].fitness))

            generation += 1


        print("Generation: {}\tString: {}\tFitness: {}".
            format(generation,
            "".join(population[0].chromosome),
            population[0].fitness))

if __name__ == '__main__':
        main()
```

Program 2

PARTICLE SWARM OPTIMIZATION

Algorithm:



```
4/10/24

Particle Swarm Optimization

import random
import math
import copy
import sys

def fitness_rastrigin(position):
    return sum([xi**2 - 10*math.cos(2*math.pi*xi) + 10 for xi in position])

def fitness_sphere(position):
    return sum([xi**2 for xi in position])

class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [self.rnd.uniform(minx, maxx) for _ in range(dim)]
        self.velocity = [self.rnd.uniform(minx, maxx) for _ in range(dim)]
        self.fitness = fitness(self.position)
        self.best_part_pos = copy.copy(self.position)
        self.best_part_fitness_val = self.fitness

def pso(fitness, max_iter, n, dim, minx, maxx):
    w, c1, c2 = 0.729, 1.49445, 1.49445
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]
    best_swarm_pos = copy.copy(swarm[0].position)
```

```
best_swarm_fitnessVal = min(p.fitness for p in swarm)
for p in swarm:
    if p.fitness < best_swarm_fitnessVal:
        best_swarm_fitnessVal = p.fitness
        best_swarm_pos = copy.copy(p.position)

for iter in range(max_iter):
    if iter % 10 == 0 and iter > 1:
        print(f"Iter = {iter} best fitness = {best_swarm_fitnessVal:.3f}")

    for p in swarm:
        r1, r2 = random.random(), random.random()
        for k in range(dim):
            p.velocity[k] = (w * p.velocity[k] +
                c1 * r1 * (p.best_part_pos[k] -
                p.position[k]) +
            (c2 * r2 * (best_swarm.pos[k] - p.pos[k])

            p.velocity[k] = max(min(p.velocity[k]
                maxx), minx)

        for k in range(dim):
            p.position[k] += p.velocity[k]

        p.fitness = fitness(p.position)

        if p.fitness < p.best_part_fitnessVal:
            p.best_part_fitnessVal = p.fitness
            p.best_part_pos = copy.copy(p.position)
        if p.fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = p.fitness
            best_swarm_pos = copy.copy(p.position)
```

```
    return best_score, best_pos

print("Begin PSO")
dim = 3
fitness = fitness_rastrigin
num_particles, max_iter = 10, 50

best_position = pso(fitness, max_iter, num_particles, dim,
                    -50, 50)

print("\nBest solution found:", ["%.6f" % pos
                                 for pos in best_position])
print("Fitness of best soln = %.6f" %
                    fitness(best_position))
```

Output

Begin PSO

Itr = 10 best fitness = ~~35.954~~ 17.294
Itr = 20 best fitness = 17.294
Itr = 30 best fitness = 6.193
Itr = 40 best fitness = 2.199
Itr = 50 best fitness = 1.830

Best solution found: [0.013356, 0.058027, -0.968403]

fitness of Best soln: 1810.272

Code:

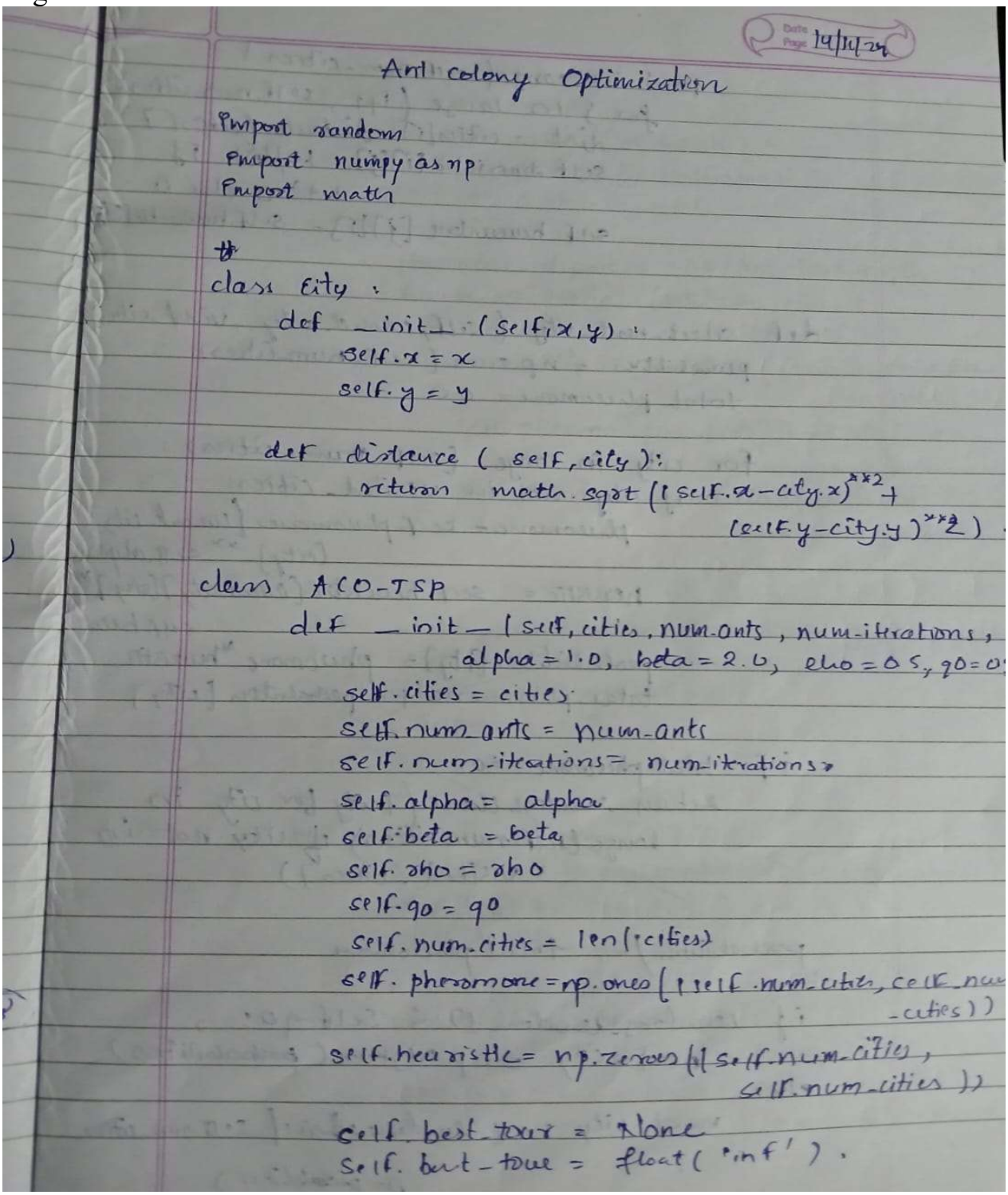
```
import random
import math
import copy
import sys

def fitness_rastrigin(position):
    return sum((xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10 for xi in position)

def fitness_sphere(position):
    return sum(xi * xi for xi in position)

class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
        self.velocity = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
        self.best_part_pos = self.position[:]
        self.fitness = fitness(self.position)
        self.best_part_fitnessVal = self.fitness

def pso(fitness, max_iter, n, dim, minx, maxx):
    w, c1, c2 = 0.729, 1.49445, 1.49445
    rnd = random.Random(0)
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

    best_swarm_pos, best_swarm_fitnessVal = [0.0] * dim, sys.float_info.max
    for p in swarm:
        if p.fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = p.fitness
            best_swarm_pos = p.position[:]

    for Iter in range(max_iter):
        if Iter % 10 == 0 and Iter > 1:
            print(f"Iter = {Iter} best fitness = {best_swarm_fitnessVal:.3f}")

        for p in swarm:
            for k in range(dim):
                r1, r2 = rnd.random(), rnd.random()
                p.velocity[k] = w * p.velocity[k] + c1 * r1 * (p.best_part_pos[k] - p.position[k]) + c2 * r2 *
(best_swarm_pos[k] - p.position[k])
                p.velocity[k] = max(min(p.velocity[k], maxx), minx)

            p.position = [p.position[k] + p.velocity[k] for k in range(dim)]
            p.fitness = fitness(p.position)

            if p.fitness < p.best_part_fitnessVal:
```

```
            p.best_part_fitnessVal = p.fitness
            p.best_part_pos = p.position[:]

        if p.fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = p.fitness
            best_swarm_pos = p.position[:]

    return best_swarm_pos

def run_pso(fitness, dim, minx, maxx):
    print(f"Goal is to minimize the function in {dim} variables")
    print(f"Function has known min = 0.0 at ({', '.join(['0'] * (dim - 1))}, 0)")

    num_particles, max_iter = 50, 100
    best_position = pso(fitness, max_iter, num_particles, dim, minx, maxx)

    print(f"Best solution found: {', '.join([f'{x:.6f}' for x in best_position])}")
    print(f"Fitness of best solution = {fitness(best_position):.6f}\n")

print("\nBegin PSO for Rastrigin function\n")
run_pso(fitness_rastrigin, 3, -10.0, 10.0)

print("\nBegin PSO for Sphere function\n")
run_pso(fitness_sphere, 3, -10.0, 10.0)
```

PROGRAM 3

ANT COLONY OPTIMIZATION

Algorithm:



Ant colony Optimization

```
import random
import numpy as np
import math

#
class City :
    def __init__ (self,x,y) :
        self.x = x
        self.y = y

    def distance ( self, city ):
        return math.sqrt ((self.x-city.x)**2 +
                          (self.y-city.y)**2 )

class ACO-TSP
    def __init__ ( self, cities, num_ants , num_iterations ,
                   alpha=1.0, beta=2.0, eho=0.5, q0=0
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.nu
                                  -cities))
        self.heuristic = np.zeroes((self.num_cities,
                                    self.num_cities))
        self.best_tour = None
        self.best_tour = float('inf').
```

```
for i in range(self.num_cities):
    for j in range(i+1, self.num_cities):
        dist = cities[i].distance(cities[j])
        self.heuristic[i][j] = 1.0/dist if
                               dist!=0 else 0
        self.heuristic[j][i] = self.heuristic[i][j]


def select_next_city(self, current_city, visited_cities):
    probabilities = np.zeros(self.num_cities)
    total_pheromone = 0.0

    for city in range(self.num_cities):
        if city not in visited_cities:
            pheromone = self.pheromone[current_city]
                        [city] ** self.alpha
            heuristic = self.heuristic[current_city][city] **
                                        self.beta
            probabilities[city] = pheromone * heuristic
            total_pheromone += probabilities[city]

    if total_pheromone == 0:
        return random.choice([city for city in
            range(self.num_cities) if city not in
                visited_cities])

    probabilities /= total_pheromone

    if random.random() < self.q0:
        next_city = np.argmax(probabilities)
    else:
        next_city = np.random.choice(self.num_cities,
                                     p=probabilities)
```

```
        return next_city

    def update_pheromone(self, ants):
        self.pheromone *= (1 - self.rho)

        for ants in ants:
            pheromone_deposit = 1.0 / ant.tour_length
            for i in range(self.num_cities):
                current_city = ant.tour[i]
                next_city = ant.tour[(i+1) % self.num_cities]

                self.pheromone[current_city][next_city] +=
                                        pheromone_deposit
                self.pheromone[next_city][current_city] +=
                                        pheromone_deposit

class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.num_cities = num_cities
        self.aco_tsp = aco_tsp
        self.tour = []
        self.tour_length = 0.0

    def construct_solution(self):
        start_city = random.randint(0, self.num_cities-1)
        self.tour = [start_city]
        visited_cities = set(self.tour)

        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_tsp.select_next_city(
                                current_city, visited_cities)
```

```
        self.tour.append(next_city)
        visited_cities.add(next_city)
        self.tour_length += self.aco_tsp.cities[current_city].
            distance(self.aco_tsp.cities[next_city])
        current_city = next_city

    self.tour_length += self.aco_tsp.cities[self.tour[-1]].
        distance(self.aco_tsp.cities[self.tour[0]])

if __name__ == "__main__":

    cities = [City(0,0), City(1,3), City(4,3), City(6,1),
              City(3,0)]

    aco = ACO_TSP(cities=cities, num_ants=10, 50, 10, 2.0, 0.5,
                  0.9)

    best_tour, best_tour_length = aco.run()

    print("Best tour:", best_tour)

    print("Best tour length:", best_tour_length)
```

Output :

Best tour : [4, 0, 1, 2, 3]
Best tour length : 15.1529

Code:

```
import random
import numpy as np
import math

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return math.sqrt((self.x - city.x)**2 + (self.y - city.y)**2)

class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations, alpha=1.0, beta=2.0, rho=0.5,
q0=0.9):
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))
        self.heuristic = np.zeros((self.num_cities, self.num_cities))  (inverse of distance)
        self.best_tour = None
        self.best_tour_length = float('inf')

        for i in range(self.num_cities):
            for j in range(i + 1, self.num_cities):
                dist = cities[i].distance(cities[j])
                self.heuristic[i][j] = 1.0 / dist if dist != 0 else 0
                self.heuristic[j][i] = self.heuristic[i][j]

    def select_next_city(self, current_city, visited_cities):
        probabilities = np.zeros(self.num_cities)
        total_pheromone = 0.0

        for city in range(self.num_cities):
```

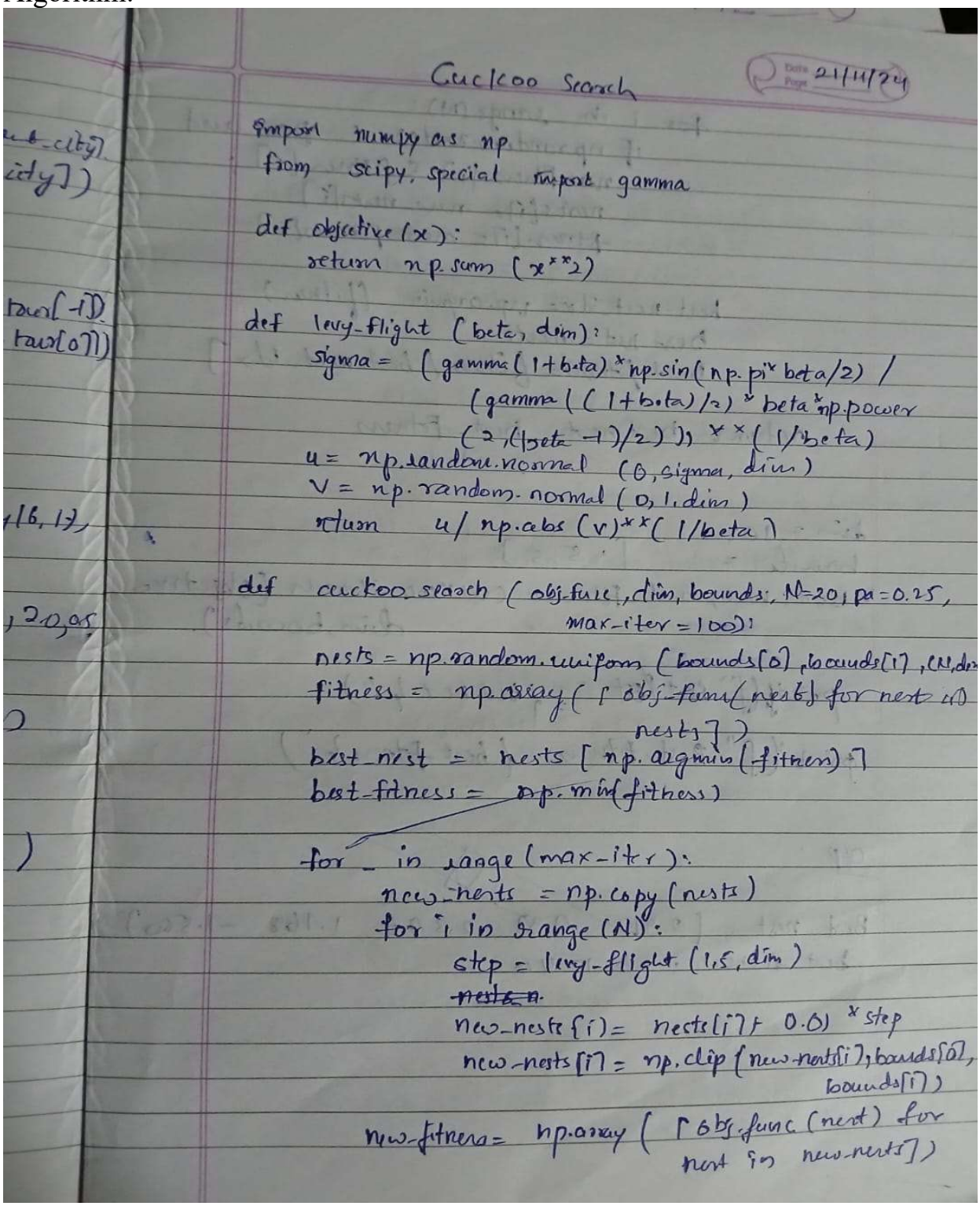
```
            if city not in visited_cities:
                pheromone = self.pheromone[current_city][city] ** self.alpha
                heuristic = self.heuristic[current_city][city] ** self.beta
                probabilities[city] = pheromone * heuristic
                total_pheromone += probabilities[city]

        if total_pheromone == 0:
            return random.choice([city for city in range(self.num_cities) if city not in
visited_cities])

        probabilities /= total_pheromone

        if random.random() < self.q0:
            next_city = np.argmax(probabilities)
        else:
            next_city = np.random.choice(self.num_cities, p=probabilities)

        return next_city

    def update_pheromone(self, ants):
        self.pheromone *= (1 - self.rho)

        for ant in ants:
            pheromone_deposit = 1.0 / ant.tour_length
            for i in range(self.num_cities):
                current_city = ant.tour[i]
                next_city = ant.tour[(i + 1) % self.num_cities]
                self.pheromone[current_city][next_city] += pheromone_deposit
                self.pheromone[next_city][current_city] += pheromone_deposit

    def run(self):
        for iteration in range(self.num_iterations):
            ants = [Ant(self.num_cities, self) for _ in range(self.num_ants)]
            for ant in ants:
                ant.construct_solution()

            self.update_pheromone(ants)


            for ant in ants:
```

```
            if ant.tour_length < self.best_tour_length:
                self.best_tour_length = ant.tour_length
                self.best_tour = ant.tour

        print(f"Iteration {iteration + 1}/{self.num_iterations}: Best Tour Length =
{self.best_tour_length}")

    return self.best_tour, self.best_tour_length

class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.num_cities = num_cities
        self.aco_tsp = aco_tsp
        self.tour = []
        self.tour_length = 0.0

    def construct_solution(self):
        start_city = random.randint(0, self.num_cities - 1)
        self.tour = [start_city]
        self.tour_length = 0.0
        visited_cities = set(self.tour)
        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_tsp.select_next_city(current_city, visited_cities)
            self.tour.append(next_city)
            visited_cities.add(next_city)
            self.tour_length +=
self.aco_tsp.cities[current_city].distance(self.aco_tsp.cities[next_city])
            current_city = next_city
        self.tour_length += self.aco_tsp.cities[self.tour[-
1]].distance(self.aco_tsp.cities[self.tour[0]])

if __name__ == "__main__":
    cities = [City(0, 0), City(1, 3), City(4, 3), City(6, 1), City(3, 0)]
    aco = ACO_TSP(cities=cities, num_ants=10, num_iterations=100, alpha=1.0,
beta=2.0, rho=0.5, q0=0.9)
    best_tour, best_tour_length = aco.run()

    print("\nBest tour found:", best_tour)
    print("Best tour length:", best_tour_length)
```

PROGRAM 4

CUCKOO SEARCH ALGORITHM

Algorithm:



Cuckoo Search

Date Page 21/11/24

```
import numpy as np
from scipy.special import gamma

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

def levy_flight(beta, dim):
    sigma = (gamma(1+beta) * np.sin(np.pi*beta/2) /
             (gamma((1+beta)/2) * beta * np.power
              (2,((beta-1)/2)))) ** (1/beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    return u / np.abs(v)**(1/beta)

def cuckoo_search(obj_func, dim, bounds, N=20, pa=0.25,
                  max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1], (N, dim
    fitness = np.array([obj_func(nest) for nest in
                        nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for _ in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(N):
            step = levy_flight(1.5, dim)
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i], bounds[0],
                                   bounds[1])
        new_fitness = np.array([obj_func(nest) for
                                nest in new_nests])
```

```
for i in range(N):
    if np.random.rand() < pa and
        new_fitness[i] < fitness[i]:
        nests[i] = new_nests[i]
        fitness[i] = new_fitness[i]

best_nest_idx = np.argmin(fitness)
Best_nest = nests
best_fitness = fitness[best_nest_idx]

return best_nest, best_fitness

dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective,
                                        dim, bounds)

print(f"Best Nest : {best_nest}");
print(f"Best fitness: {best_fitness}")
```

O/P :

Best nest : [2.519, 0.81 2.396 1.168 -1.560]
best fitness : 32.1240

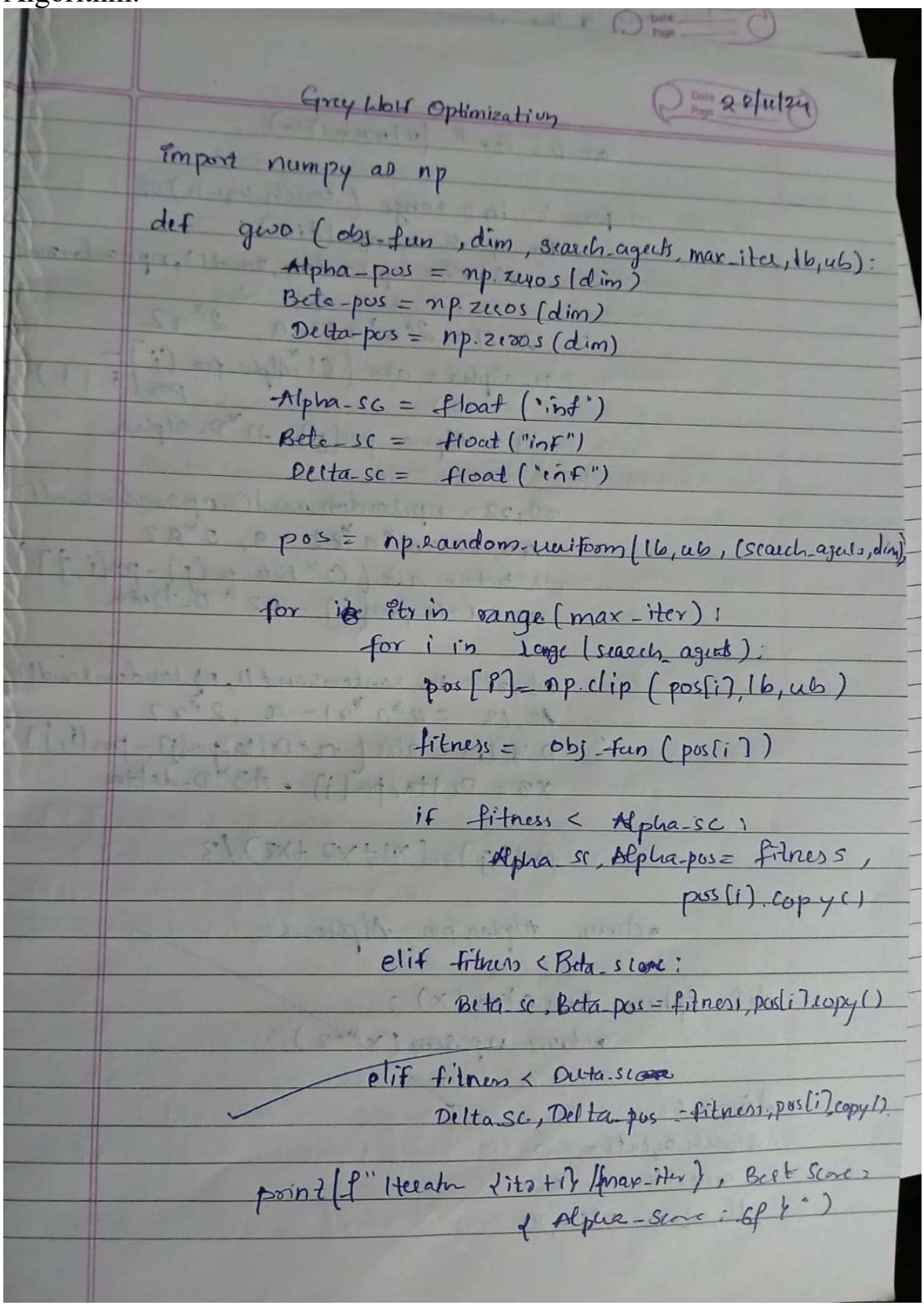
Code:

```
import numpy as np
from scipy.special import gamma

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

def levy_flight(beta, dim):
    sigma = (gamma(1+beta)*np.sin(np.pi*beta/2) /
             (gamma((1+beta)/2)*beta*np.power(2, (beta-1)/2)))**(1/beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    return u / np.abs(v)**(1/beta)

def cuckoo_search(obj_func, dim, bounds, N=20, pa=0.25, max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1], (N, dim))
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)
    for _ in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(N):
            step = levy_flight(1.5, dim)  # Lévy exponent 1.5
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])
        new_fitness = np.array([obj_func(nest) for nest in new_nests])
        for i in range(N):
            if np.random.rand() < pa and new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]
        best_nest_idx = np.argmin(fitness)
        best_nest = nests[best_nest_idx]
        best_fitness = fitness[best_nest_idx]
    return best_nest, best_fitness
dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective, dim, bounds)

print(f"Best Nest: {best_nest}")
print(f"Best Fitness: {best_fitness}")
```

PROGRAM 5

GREY WOLF OPTIMIZATION

Algorithm:



Grey Wolf Optimization

20/11/24

```
import numpy as np

def gwo(obj_fun, dim, search_agents, max_iter, lb, ub):
    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)

    Alpha_sc = float("inf")
    Beta_sc = float("inf")
    Delta_sc = float("inf")

    pos = np.random.uniform(lb, ub, (search_agents, dim))

    for itr in range(max_iter):
        for i in range(search_agents):
            pos[i] = np.clip(pos[i], lb, ub)

            fitness = obj_fun(pos[i])

            if fitness < Alpha_sc:
                Alpha_sc, Alpha_pos = fitness,
                                      pos[i].copy()

            elif fitness < Beta_sc:
                Beta_sc, Beta_pos = fitness, pos[i].copy()

            elif fitness < Delta_sc
                Delta_sc, Delta_pos = fitness, pos[i].copy()

    print(f"Iteration {itr+1}/{max_iter}, Best Score:
          {Alpha_sc:.6f}")
```

```
a= 2 - itr * (2/max_iter)

for i in range (search_agents):
    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] -
                                pos(i,j])
        X1 = Alpha_pos[j] - A1*D_alpha

        r1,r2 = np.random.rand(), np.random.rand()
        A2, C2 = 2*a*r1 - a, 2*a2
        D_beta = abs ( C2* Beta_pos[j] - pos[i,j])
        X2 = Beta_pos[j] - A2 * D_Delta

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

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

    return Alpha_pos, Alpha_sc

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

dim = 5
search_agents = 50
max_iter = 50
lb,ub = -10,10
```

best_pos, best_sc = gwo(sphere_function, dim, search_agents, max_iter, lbs, ubs)

print("Best Position : ", best_pos)
print("Best Score : ", best_sc)

Output

Best Score : 4.7615e-09

Best Pos : [-2.9929e-05, -2.911e-05, +2.9204e-05, -3.1702e-05]

Code:

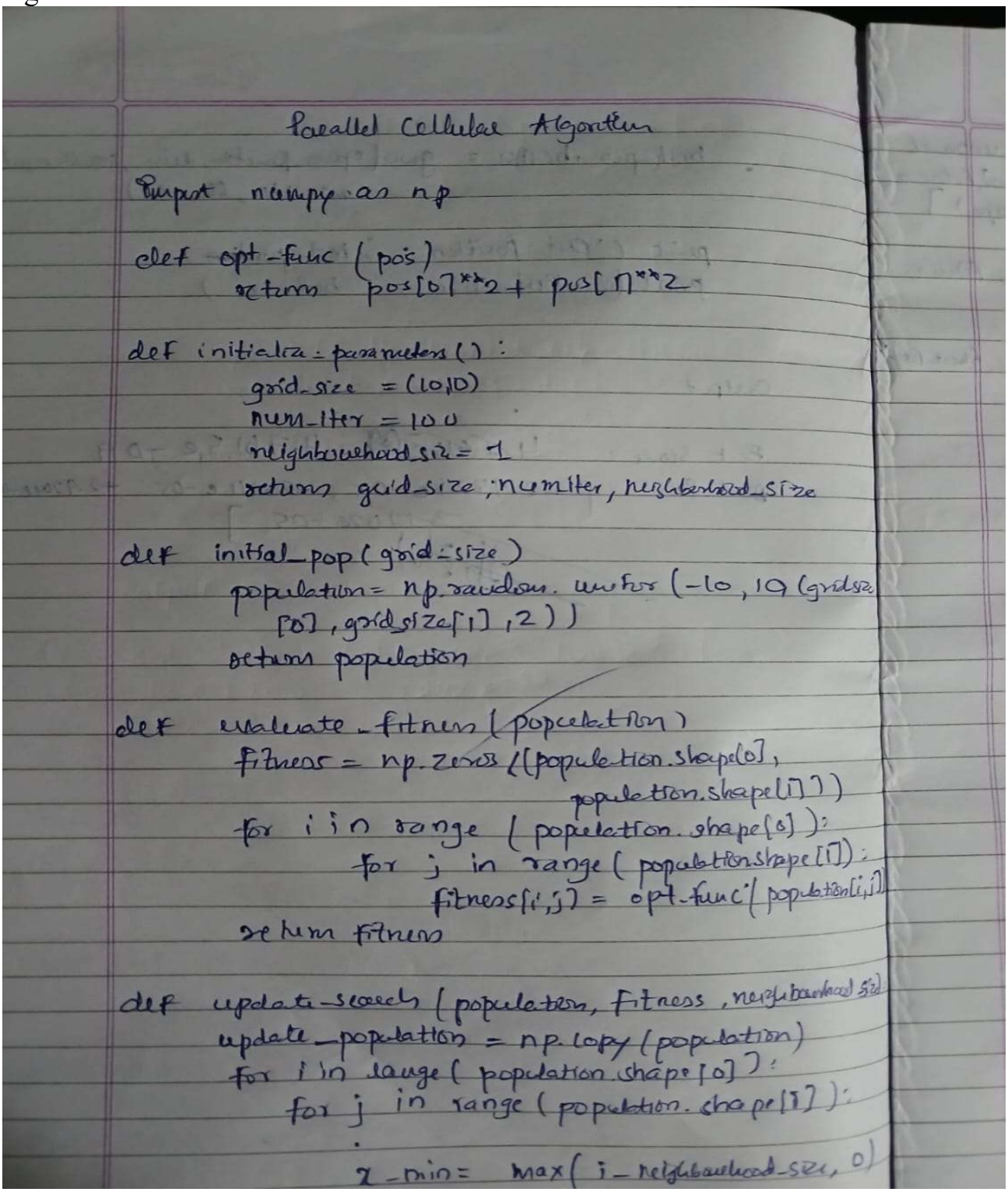
```
import numpy as np

def gwo(obj_function, dim, search_agents, max_iter, lb, ub):
    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, (search_agents, dim))

    for iteration in range(max_iter):
        for i in range(search_agents):
            positions[i] = np.clip(positions[i], lb, ub)

            fitness = obj_function(positions[i])

            if fitness < Alpha_score:
                Alpha_score, Alpha_pos = fitness, positions[i].copy()
            elif fitness < Beta_score:
                Beta_score, Beta_pos = fitness, positions[i].copy()
            elif fitness < Delta_score:
                Delta_score, Delta_pos = fitness, positions[i].copy()

        print(f"Iteration {iteration + 1}/{max_iter}, Best Score: {Alpha_score:.6f}")

        a = 2 - iteration * (2 / max_iter)  # Linearly decreases from 2 to 0

        for i in range(search_agents):
            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

    return Alpha_pos, Alpha_score

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

dim = 5
search_agents = 30
max_iter = 50
lb, ub = -10, 10

best_position, best_score = gwo(sphere_function, dim, search_agents, max_iter, lb, ub)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

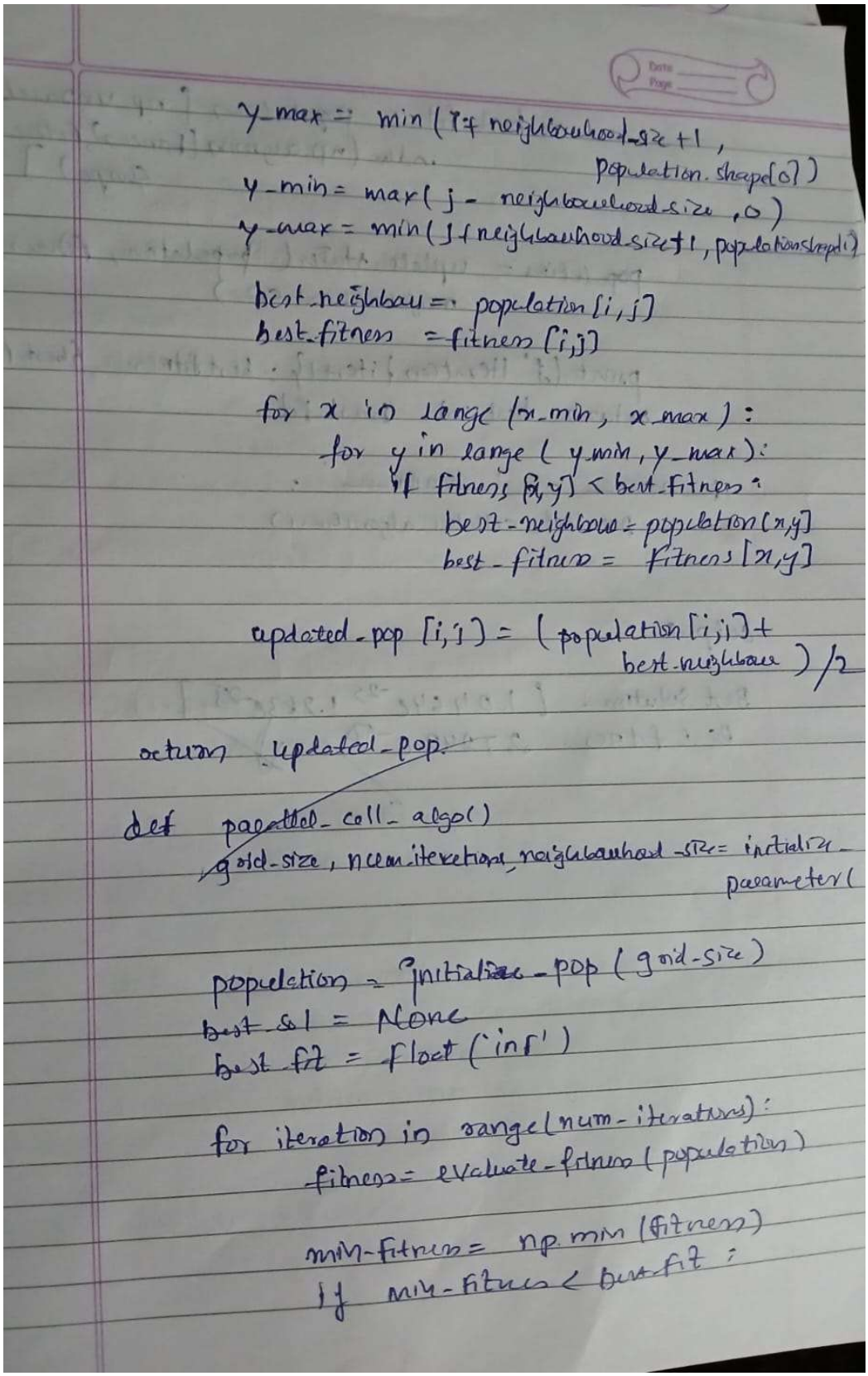
PROGRAM 6

PARALLEL CELLULAR ALGORITHM

Algorithm:



```
Parallel Cellular Algorithm

import numpy as np

def opt_func (pos)
    return pos[0]**2 + pos[1]**2

def initialize_parameters():
    grid_size = (10,10)
    num_iter = 100
    neighbourhood_size = 1
    return grid_size, num_iter, neighbourhood_size

def initial_pop(grid_size)
    population = np.random.uniform(-10, 10, (grid_size
        [0], grid_size[1], 2))
    return population

def evaluate_fitness(population)
    fitness = np.zeros((population.shape[0],
                        population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i,j] = opt_func(population[i,j]
    return fitness

def update_search(population, fitness, neighbourhood_size)
    update_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):

            x_min = max(i - neighbourhood_size, 0)
```



```
y_max = min(i + neighbourhood_size + 1,
                    population.shape[0])
y_min = max(j - neighbourhood_size, 0)
y_max = min(j + neighbourhood_size + 1, population.shape[1])

best_neighbour = population[i, j]
best_fitness = fitness[i, j]

for x in range(x_min, x_max):
    for y in range(y_min, y_max):
        if fitness[x, y] < best_fitness:
            best_neighbour = population[x, y]
            best_fitness = fitness[x, y]

updated_pop[i, j] = (population[i, j] +
                     best_neighbour) / 2

return updated_pop

def parallel_coll_algo()
    grid_size, num_iterations, neighbourhood_size = initialize_
                                                    parameter(

population = initialize_pop(grid_size)
best_sol = None
best_fit = float('inf')

for iteration in range(num_iterations):
    fitness = evaluate_fitness(population)

    min_fitness = np.min(fitness)
    if min_fitness < best_fit:
```

```
        best_fit = population[np.unravel_
                   index(np.argmin(fitness), fitness.
                                   shape)]

    population = update_state(population, fitness,
                              neighborhood_size)

    print(f"Iteration {iter+1}: best fitness: {best_fit}")
return best_sol, best_fit

if __name__ == "__main__":
    parallel_cellular_algorithm()
```

Output

Best Solution : $[1.0464e^{-25}\ 1.263e^{-25}]$

Best fitness : $2.7242e^{-50}$

Code:

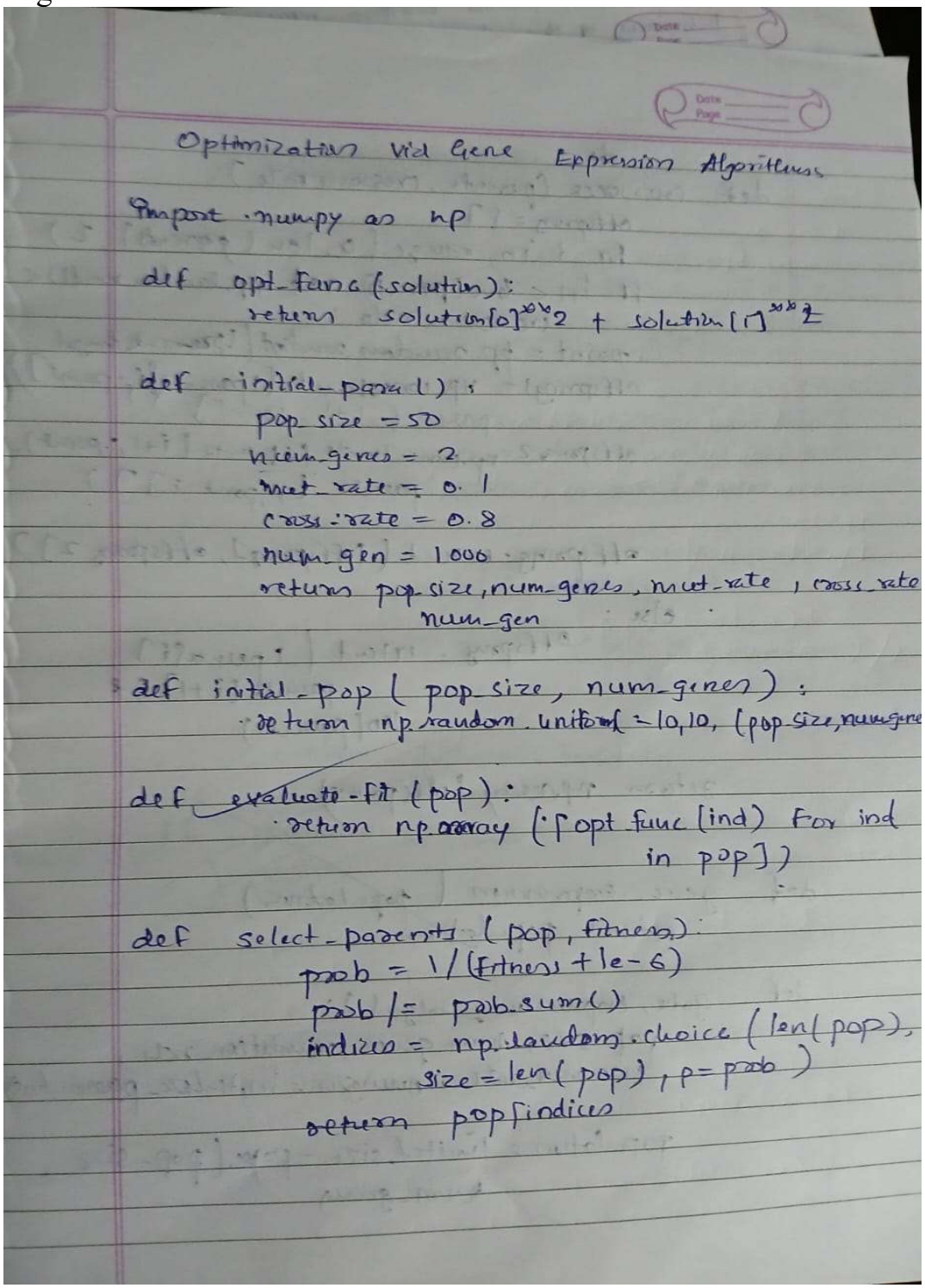
```
import numpy as np

def optimization_function(position):
    return position[0]*2 + position[1]*2

def initialize_parameters():
    grid_size = (10, 10)
    num_iterations = 100
    neighborhood_size = 1
    return grid_size, num_iterations, neighborhood_size

def initialize_population(grid_size):
    population = np.random.uniform(-10, 10, (grid_size[0], grid_size[1], 2))
    return population

def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

def update_states(population, fitness, neighborhood_size):
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            x_min = max(i - neighborhood_size, 0)
            x_max = min(i + neighborhood_size + 1, population.shape[0])
            y_min = max(j - neighborhood_size, 0)
            y_max = min(j + neighborhood_size + 1, population.shape[1])

            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]

            for x in range(x_min, x_max):
                for y in range(y_min, y_max):
                    if fitness[x, y] < best_fitness:
                        best_neighbor = population[x, y]
                        best_fitness = fitness[x, y]
```

```
            updated_population[i, j] = (population[i, j] + best_neighbor) / 2

    return updated_population


def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size = initialize_parameters()

    population = initialize_population(grid_size)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)

        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness),
fitness.shape)]

        population = update_states(population, fitness, neighborhood_size)

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

if _name_ == "_main_":
    parallel_cellular_algorithm()
```

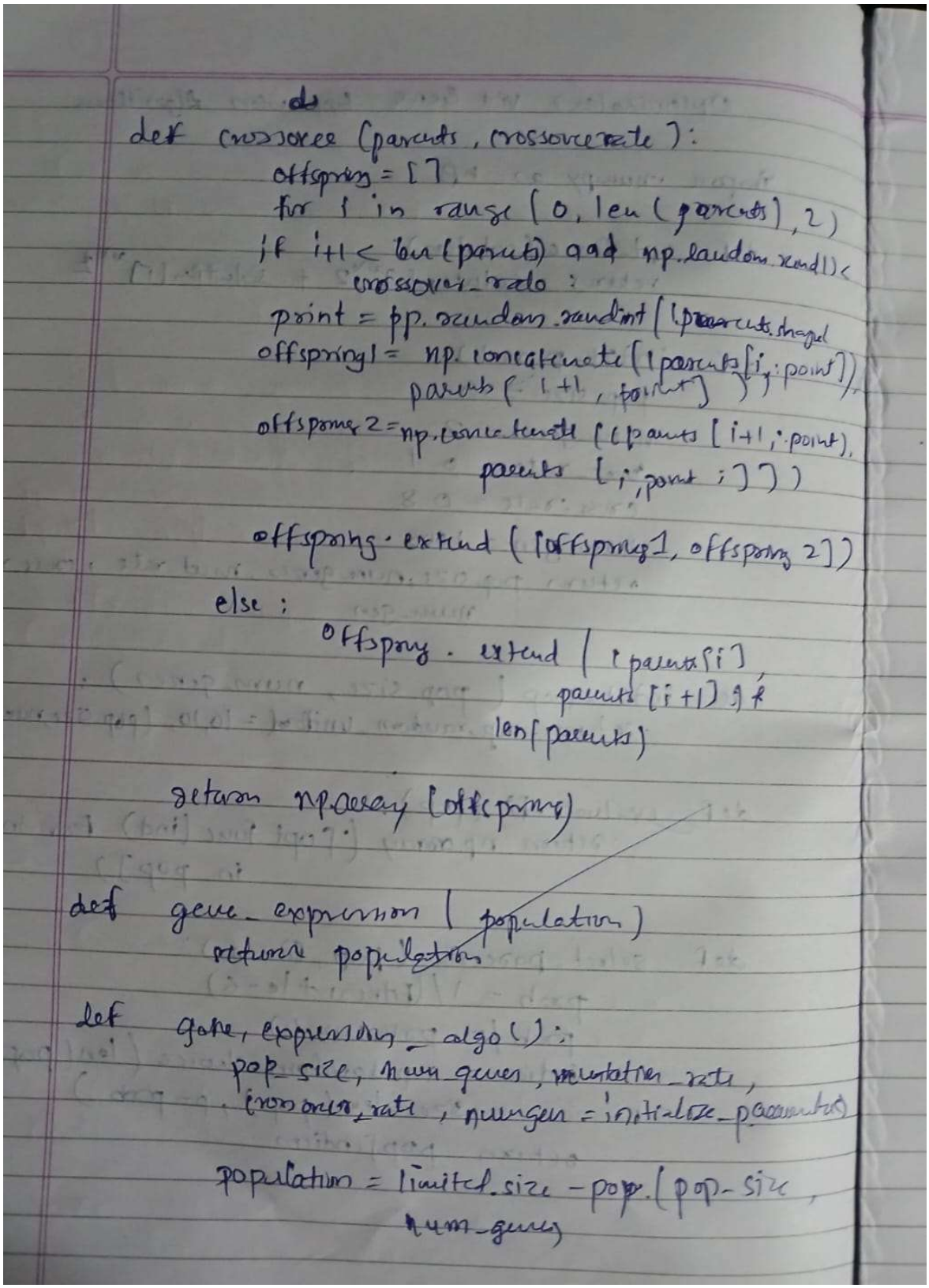
PROGRAM 7

GENE EXPRESSION ALGORITHM

Algorithm:



```
Optimization via Gene Expression Algorithms

import numpy as np

def opt_func(solution):
    return solution[0]**2 + solution[1]**2

def initial_para():
    pop_size = 50
    num_genes = 2
    mut_rate = 0.1
    cross_rate = 0.8
    num_gen = 1000
    return pop_size, num_genes, mut_rate, cross_rate
           num_gen

def initial_pop(pop_size, num_genes):
    return np.random.uniform(-10,10,(pop_size,num_genes

def evaluate_fit(pop):
    return np.array([opt_func(ind) for ind
                     in pop])

def select_parents(pop, fitness):
    prob = 1/(fitness + 1e-6)
    prob /= prob.sum()
    indices = np.random.choice(len(pop),
              size=len(pop), p=prob)
    return pop[indices]
```



```
def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i+1 < len(parents) and np.random.rand() < crossover_rate:
            point = np.random.randint(1, parents.shape[1])
            offspring1 = np.concatenate((parents[i, :point],
                                         parents[i+1, point:]))
            offspring2 = np.concatenate((parents[i+1, :point],
                                         parents[i, point:]))
            offspring.extend([offspring1, offspring2])
        else:
            offspring.extend([parents[i],
                              parents[i+1]][:len(parents)])
    return np.array(offspring)

def gene_expression(population):
    return population

def gene_expression_algo():
    pop_size, num_genes, mutation_rate,
    crossover_rate, num_gen = initialize_parameters()

    population = limited_size - pop.(pop_size,
                                     num_genes)
```

```
best_sol = None
best_fit = float('inf')

for gen in range(num_gen):
    fitness = evaluate_fitness(population)
    fit_idx = np.argmin(fitness)
    if fitness[fit_idx] < best_fitness:
        best_fitness = fitness[fit_idx]
        best_sol = pop[fit_idx]
    parents = select_parents(pop, fitness)
    offspring = crossover(parents, crossover_
    pop = gene_expression(pop)
    print('Generation {gen+1}')

print('Best : {best_sol} Best Fitness =
                {best_fitness}')

return best_sol, best_fit

if __name__ == "__main__":
    gene_express_algo()
```

Output

```
Best sol :  [-.9039  -9.7681]
Best fit :  -35.744,
```

Code:

```
import numpy as np

def optimization_function(solution):
    return solution[0]*2 + solution[1]*2

def initialize_parameters():
    population_size = 50
    num_genes = 2
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes, mutation_rate, crossover_rate, num_generations

def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size, num_genes))

def evaluate_fitness(population):
    return np.array([optimization_function(ind) for ind in population])

def select_parents(population, fitness):
    probabilities = 1 / (fitness + 1e-6)
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population), size=len(population), p=probabilities)
    return population[indices]

def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents) and np.random.rand() < crossover_rate:
            point = np.random.randint(1, parents.shape[1])
            offspring1 = np.concatenate((parents[i, :point], parents[i + 1, point:]))
            offspring2 = np.concatenate((parents[i + 1, :point], parents[i, point:]))
            offspring.extend([offspring1, offspring2])
        else:
            offspring.extend([parents[i], parents[i + 1] if i + 1 < len(parents) else
parents[i]])
    return np.array(offspring)

def mutate(offspring, mutation_rate):
```

```
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene = np.random.randint(individual.size)
            individual[gene] += np.random.normal(0, 1)

def gene_expression(population):
    return population

def gene_expression_algorithm():
    population_size, num_genes, mutation_rate, crossover_rate, num_generations =
initialize_parameters()

    population = initialize_population(population_size, num_genes)

    best_solution = None
    best_fitness = float('inf')

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

        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        parents = select_parents(population, fitness)
        offspring = crossover(parents, crossover_rate
        population = mutate(offspring, mutation_rate)
        population = gene_expression(population)
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")
    print(f"Best Solution: {best_solution}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

if _name_ == "_main_":
    gene_expression_algorithm()
```