- Genetic Algorithm for optimization Problem
Genetic Algorithm are optimization techniques inspired by principles of natural selection & genetics. They iteratively evolve a population of candidate solutions using selection, crossover & mutation to find near optimal solutions

Uses:
- Optimization: Finding optimal solutions
- Search Problem: Exploring large sol^n spaces
- Machine Learning: Tuning hyperparameters

Application Fields:
1) Scheduling: Job/resources scheduling
2) Routing: Vehicle and network routing
3) Game Development: AI behaviour and strategy
4) Finance: Portfolio Optimization
5) Robotics: Path planning and control
6) Engineering: Structural and circuit design

Optimization techniques:
- Fitness Function design - Effective evaluat^n of solution
- Parameter Tuning: Adjusting key algorithm parameters
- Hybrid Approaches: Combining with other optimiz - ation methods
- Elitism: Preserving top sol^n across generat^n

- Particle Swarm Optimization (PSO) for Function Optimization

PSO is inspired by social behaviour in Nature where a group of candidate solutions collaboratively searches for the best sol^n in the solution space.

- Uses:
- Optimization: Finding optimal sol^n
- Funct^n optimizat^n: Minimizing or maximizing funcs
- Machine learning: Tuning model parameters
- Application Feilds:
1) Engineering: Design and control Optimizat^n
2) Finance: Portfolio and risk optimizat^n
3) Robotics: path planning & sensor optimizat^n
4) Telecomunications: network resource allocat^n
5) Data Mining: Feature select^n & clustering
- Optimization Techniques
1) Velocity update: Adjusting movement based on personal and global bests
2) Parameter Tuning: Optimization inteia and coefficients for better convergence
3) Hybrid Approaches: Combining with other methods for enhanced performance
4) Adaptive PSO: Dynamic adjustment of parameters during iterations.

. Ant Colony Optimization (ACO) for Traveling Salesman
ACO is a metaheuristic inspired by ants foraging behaviour, used to find the shortest route for the Traveling Salesman Problem

Uses:
. Path optimization: Finding the shortest route visiting each city once
. Combinatorial optimizat°: Solving routing and scheduling issues

Application Fields:
1) Logistics: Delivery route planning
2) Telecommunications: Network routing
3) Robotics: Path planning
4) Tourism: Travel itinerary optimization
5) Manufacturing: Production scheduling

. Optimization Techniques:
1) Pheromon update: Adjusting pheromon levels based on solution quality
2) Heuristic informat°: Combining pheromones with distance heuristics
3) Parameter Tuning: optimizing evaporat° rates and ant no.s
4) Hybrid ACO: Combining with local search methods

- Cukoo Search (CS)

It is a natural inspired optimizat° algorithm modeled on the parasitic behaviour of certain cuckoo species that lay thier eggs in the nests of other birds. The algorithm uses levy flights to perform a random walk to explore the search space effectively.

- Uses:
  - Global optimizat° problems
  - Function optimizat°
  - Multi-objective optimizat°
- Application Feilds
  - Engineering design
  - Image processing
  - Feature select° in machine learning
  - Wireless sensor networks
- Optimizat° Technique
  - Hybridizat° with other algorithms (eg. Particles Swarm optimization, Genetic Algorithm)
  - Parameter tuning for step size (levy flight) and population size
- Hybridizat° with other algorithms
- Parameter tuning for stepsize and populat° size
- Dynamic control of search balance between explorat° and exploitation.

- Grey Wolf Optimizer (GWO)

GWO mimics the social hierachy and hunting behaviour of grey wolves. The wolves are divided into four categories: alpha, beta, delta and omega. The optimizat° process is guided by the top wolves, simulating the packs encircling and attacking strategies.

- Uses:
  - optimizat° of complex functions
  - Multi-objective optimizat°
  - Constrained and unconstrained optimizat°
- Applicat° Feilds:
  - Engineering design optimizat°
  - Feature select° in machine learning
  - Power systems optimizat°
  - Feature select° in ML
  - Power systems Optimizat°
- Optimization Techniques
  - Dynamic tuning of parameters (α, β, δ leadership parameters)
  - Hybridizat° with other metaheuristics
  - Balancing exploration (global search) and exploitation (local search)

- Parallel Cellular Algorithms (PCA)

PCA extend the concept of Cellular Automata by including parallelism in computation. The algorithm operates on a grid where cells represent the solution space, and each cell interacts only with its local neighbor-hood, leading to a decentralized approach to optimization.

- Uses:
- Parallel and distributed optimization
- Multi agent systems
- Local search optimization
- Multi-agent system
- Local search optimization
- Application Fields:
- Evolutionary algorithms
- Multi core computing
- Genetic algorithms for large scale problems
- Traffic simulation and control
- Optimization Technique:
- Parallelization across multiple processors or cores
- Dynamic load balancing to optimize resource utilization
- Hybridization with other parallel algorithms (eg. Parallel Genetic Algorithm)

- Gene Expression Programming (GEP)

Gene Expression Programming (GEP) is an evolutionary algorithm that encodes potential solutions as linear chromosomes, which are then expressed as nonlinear structures to solve complex problems. It is inspired by genetic programming and uses a genetic algorithm approach for evolution.

- Uses:
- Symbolic regression
- Function approximation
- Classification and prediction
- Application Fields:
- Financial modeling
- Control system
- Bioinformatics
- Robotics (evolution of controller)
- Optimization Technique
- Fitness function optimization (e.g., dynamic adaptation of selection pressure)
- Mutation and crossover parameter tuning
- Hybridization with neural networks or other evolutionary algorithms.

9th oct

Genetic Algorithm

It is a optimization technique based on natural selection and evolution. It is done with the help of crossover we generally use MATLAB alongside example such as Travelling Salesman Problem.
It explains history, current uses & future potential of genetic algorithm in various fields

1) Components, structure & Terminology of genetic algorithms

Key components of genetic algorithm

1) Fitness Function - measure how well a chromosome solves problem
2) Population of chromosomes - a set of candidates each represented by an array of parameter values
   chromosome = [p1, p2, ... pNpar]
3) Selection - chooses which chromosome reproduces based on fitness score calculated as

$$P(C_{G3}) = \left| \frac{f(C_{G3})}{\sum_{i=1}^{N_{pop}} f(C_i)} \right|$$

4) Crossover - Combines two parent chromosomes to produce offspring
   Ex: parent chromosomes are
   1101011106 1000 & 0101110 1010010
   are crossed over after 4th bit
   0101011100 1000 & 1101110 1010010

5) Mutation - Introduces random change to chromosome to maintain diversity

From initial popul^n chromosomes are selected based on fitness function followed by crossover & Mutat^n to create a new generat^n and the cycle continues until a satisfactory sol^n is found

2) Preliminary Example

Given 3 examples demonstrat GA of increasing complexity

2.1) Maximizing a func^n of one variable

Maximize the func^n $f(x) = \frac{-x^2}{10} + 3x$ for x in range [0, 31]

Assume x as a chromosome and is represented as 5 bits 0 to 31 i.e 00000 to 11111, generate random popul^n evaluate their fitness and use probability function

Pair of chromosome crossover to produce offspring

After one generat^n both max & avg fitness values of popul^n improved illustrating GA's ability to evolve towards fitter sol^n

2.2) Number of 1's in a String

Maximiz

16th oct

Program - 1

1) Genetic algorithm

```
import random
def fitness_function(x):
    return x**2
def generate_population(pop_size, min_x, max_x):
    return [generate_individual(min_x, max_x)
    for _ in range(pop_size)]
def selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [fitness / total_fitness for
fitness in fitness_scores]
    return random.choices(population, weights=
selection_probs, k=2)

def crossover(parent1, parent2):
    alpha = random.uniform(0, 1)
    child1 = alpha * parent1 + (1-alpha) * parent2
    child2 = alpha * parent2 + (1-alpha) * parent1
return child1, child2

def mutation(individual, mutate_rate, min_x, max_x):
    if random.random() < mutate_rate:
        return generate_individual(min_x, max_x)
    return individual

def genetic_algorithm(pop_size, min_x, max_x, gen
_ations, mutation_rate):
```

```
population = generate_population(pop_size, min_x, max_x)
for generation in range(generations):
    fitness_scores = [fitness_function(ind) for ind in population]
    new_population = []
    for _ in range(pop_size//2):
        parent1, parent2 = selection(population, fitness_scores)
        child1, child2 = crossover(parent1, parent2)
        child1 = mutation(child1, mutation_rate, min_x, max_x)
        new_population.extend([child1, child2])
    population = new_population
    best_individual = max(population, key=fitness_function)
    print(f"Generation {generation+1}: Best solution = {best_individual}, Fitness = {fitness_function(best_individual)}")
return max(population, key=fitness_func)
```

```
population_size = 20
min_value = -10
max_value = 10
num_generations = 50
mutation_probability = 0.1
best_solution = genetic_algorithm(population_size, min_value,
```

mean_value, num_generations, mutation_probability)
print(f"Best solution found: {best_sol}, Fitness: {fitness_function(best_sol)}")

Output: Generation 1: Best Solution = 9.89804... Fitness = 97.97138...
Generation 50: Best Solution = 9.89804... Fitness = 97.97138...
Best Solution found: 9.89804..., Fitness: 97.97138...

23rd oct

Program - 2

2) Particle Swarm optimization

```
import random
def fitness_function(x):
    return x**2

class Particle:
    def __init__(self, min_x, max_x):
        self.position = random.uniform(min_x, max_x)
        self.velocity = random.uniform(-1, 1)
        self.best_posit = self.position
        self.best_fitness = fitness_func(self.posit)

def update_velocity(self, global_best_posit, inertia_coefficient, cognitive_coefficient, social_coeffcient):
    r1, r2 = random.random(), random.random()
    cognitive_velocity = cognitive_coefficient * r1 * (self.best_posit - self.posit)
    social_velocity = social_coefficient * r2 * (global_best_posit - self.posit)
    self.velocity = (inertia_coefficient * self.velocity) + cognitive_velocity + social_velocity

def update_position(self, min_x, max_x):
    self.posit += self.velocity
    self.posit = max(min_x, min(self.
```

posit^n, max_x))
fitness = fitness_func^n (self.posit^n)
if fitness < self.best_fitness:
self.best_fitness = fitness
self.best_posit^n = self.posit^n

def particle_swarm_optimizer^n (pop_size, min_x, max_x, generations, inertia_wt, cognitive_coefficient, social_coefficient):
swarm = [Particle(min_x, max_x) for _ in range(pop_size)]

global_best_posit^n = swarm[0].best_posit^n
global_best_fitness = swarm[0].best_fitness

for generat^n in range(generations):
for particle in swarm:
if particle.best_fitness < global_best_fitness:
global_best_fitness = particle.best_fitness
global_best_posit^n = particle.best_posit^n

particle.update_velocity(global_best_posit^n, inertia_wt, cognitive_coefficient, social_coefficient)
particle.update_position(min_x, max_x)

print(f"Generat^n {generat^n+1}: Best sol^n = {global_best_posit^n}, Fitness = {global_

best_fitness 3")
return global_best_posit^n

population_size = 30
min_value = -10
max_value = 10
num_generat^n = 50
inertia_wt = 0.5
cognitive_coefficient = 1.5
social_coefficient = 1.5

best_sol^n = particle_swarm_optimizat^n (popul_at^n_size, min_val, max_val, num_generations, inertia_wt, cognitive_coefficient, social_coefficient)
print("Best sol^n found: {best_sol^n}
Fitness: {Fitness_func^n(best_sol^n)}")

output:
generat^n 1: Best sol^n = 0.004891... Fitness = 2.3928e-05
generat^n 50: Best sol^n = 3.7754 x 10^-9 Fitness = 1.4254 x 10^-17
Best sol^n found = 3.7754 x 10^-9 Fitness = 1.425 4x10^-17

23/10/24

Program 3

3) Ant Colony Optimization for Travelling Salesman.

```
import numpy as np
import random
class AntColony:
    def __init__(self, cities, num_ants=10, alpha=1.0, beta=2.0, rho=0.5, iterations=100):
        self.cities = cities
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.iterations = iterations
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))
        self.distance = self.calculate_dist()

def calculate_distances(self):
    distance = np.zeros((self.num_cities, self.num_cities))
    for i in range(self.num_cities):
        for j in range(i+1, self.num_cities):
            distance[i][j] = distances[j][i] = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
    return distance
```

```
def select_next_city(self, current_city, visited):
    probabilities = []
    for next_city in range(self.num_cities):
        if next_city not in visited:
            pheromone = self.pheromone
[current_city][next_city] ** self.alpha
            heuristic = (1 / self.distance[curre
_nt_city][next_city]) ** self.beta
            probabilities.append(pheromone *
heuristic)
    else:
        probabilities.append(0)
    total = sum(probabilities)
    probabilities = [p / total for p in probabiliti]
    return np.random.choice(range(self.num_
cities), p=probabilities)

def construct_sol(self):
    for _ in range(self.num_ants):
        visited = [0]
        current_city = 0
        for _ in range(1, self.num_cities):
            next_city = self.select_next
_city(current_city, visited)
            visited.append(next_city)
            current_city = next_city
        visited.append(0)
        yield visited
```

```
def update_pheromones(self, solutions):
    self.pheromone *= (1 - self.rho)
    for sol in sols:
        length = self.calculate_tour_length(sol)
        pheromone_deposit = 1 / length
        for i in range(len(sol) - 1):
            self.pheromone[sol[i]][sol[i+1]] += pheromone_deposit

def calculate_tour_length(self, sol):
    return sum(self.distance[sol[i]][sol[i+1]] for i in range(len(sol)-1))

def run(self):
    best_sol = None
    best_length = float('inf')

    for _ in range(self.iterations):
        sols = [self.construct_sol() for _ in range(self.num_ants)]
        self.update_pheromones(sols)

        for sol in sols:
            length = self.calculate_tour_length(sol)
            if length < best_length:
                best_length = length
                best_sol = sol
    return best_sol, best_length
```

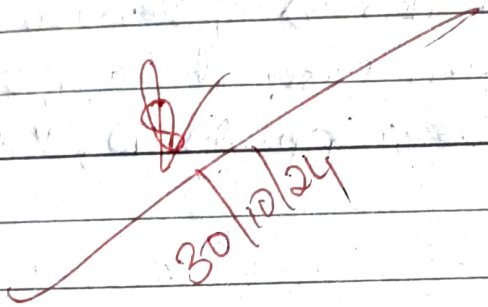
```
cities = [(0,0), (1, 2), (2, 1), (3, 3), (4, 2)]
aco = AntColony(cities)
best_route, best_distance = aco.run()

print("Best route:", best_route)
print("Best Distance:", best_distance)
```

Output:

Best route [0, 4, 3, 4, 2, 0]

Best distance: 10.3584



4) Cuckoo search algorithm

```
import numpy as np
from scipy.special import gamma
def objective_fun(x):
    return np.sum(x**2)
def levy_flight(d, size=1):
    sigma_u = np.power(gamma(1+d)*np.sin(np.pi*d/2)/gamma((1+d)/2)*d*2**((d-1)/2)), 1/d)
    u = np.random.normal(0, sigma_u, size)
    v = np.random.normal(0, 1, size)
    step = u/np.power(np.abs(v), 1/d)
    return step
def cuckoo_search(objective_fun, n_nests=25, max_iter=50, pa=0.25):
    nests = np.random.uniform(low=-5, high=5, size=(n_nests, 2))
    fitness = np.apply_along_axis(objective_function, 1, nests)
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)
    for iteration in range(max_iter):
        for i in range(n_nests):
            new_nest = nests[i] + levy_flight(size=2)
```

```
new_fitness = objective_function(new_nest)
if new_fitness < fitness[i]:
    nests[i] = new_nest
    fitness[i] = new_fitness
abandon = np.random.rand(n_nests) < pa
nests[abandon] = np.random.uniform(low=-5, high=5, size=(np.sum(abandon), 2))
current_best_nest = nests[np.argmin(fitness)]
current_best_fitness = np.min(fitness)
if current_best_fitness < best_fitness:
    best_nest = current_best_nest
    best_fitness = current_best_fitness
print(f"Iteration {iteration+1}, Best fitness: {best_fitness}")
return best_nest, best_fitness
n_nests = 25
max_iter = 50
pa = 0.25
best_solution, best_value = cuckoo_search(objective_function, n_nests, max_iter, pa)
print(f"\n Best sol": {best_sol}")
print(f" Best fitness value : {best_value}")
```

output:

Iteration 1: Best Fitness: 0.5897068...

Iterat^n 2: Best Fitness: 0.4673083...

⋮

Iteration 50: Best Fitness: 0.0001617114239...

Best solution: [-0.5115321, -3.0542364]

Best Fitness value: 0.0001617114239...

8/11/24

5) Grey Wolf Optimizer

Algorithm:

1) Define obj func^n $f(x) = x^2$
2) Initialize n - wolves, n - iterations, dim & boundaries
3) Initialize wolves by randomly placing wolves within search space. Calculate fitness of each wolf using obj func^n
4) Identify leadership hierarchy & rank
   Alpha wolf : Best sol^n
   Beta wolf : 2nd best
   Delta wolf : 3rd
5) Update wolves position:
   For each wolf:
   Calc distance D to α, β, δ wolves.
   $D_\alpha = |\gamma_1 \alpha - \text{wolf pos}^n|$
   Update posit^n based on α, β, δ influences
   $$x_{new} = \frac{X_\alpha + X_\beta + X_\delta}{3}$$
   Decrease parameter a linearly from 2 to 0 over iterations Vs exploration
6) Make sure wolf remains within bounds & recalculate fitness of α, β, δ wolves based on new hierarchy
7) Perform position updates & recalculate
8) Return alpha wolf posit^n & fitness as optimal sol^n

Code:

```
import numpy as np
def obj_fun(x):
    return np.sum(x**2)

def gwo(obj_func, dim, solves, iters, lb, ub):
    pos = np.random.uniform(low=lb, high=ub, size=(solves, dim))
    a_pos, b_pos, d_pos = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    a_score, b_score, d_score = float("inf"), float("inf"), float("inf")

    for t in range(iters):
        for i in range(solves):
            fit = obj_func(pos[i])
            if fit < a_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = a_score, b_pos.copy()
                a_score, a_pos = fit, pos[i].copy()
            elif fit < b_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = fit, pos[i].copy()
            elif fit < d_score:
                d_score, d_pos = fit, pos[i].copy()
        a = 2 - t * (2/iters)
        for i in range(solves):
            for j in range(dim)
```

r1, r2 = np.random.rand(), np.random.rand()

A1, C1 = 2*a*r1 - a, 2*r2

D_a = abs(C1*a - pos[i,j])

X1 = a - pos[i,j] - A1*D_a

r1, r2 = np.random.rand(), np.random.rand()

A2, C2 = 2*a*r1 - a, 2*r2

D_b = abs(C2*b - pos[i,j])

X2 = b - pos[i,j] - A2*D_b

r1, r2 = np.random.rand(), np.random.rand()

A3, C3 = 2*a*r1 - a, 2*r2

D_d = abs(C3*d - pos[i,j])

X3 = d - pos[i,j] - A3*D_d

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

pos[i] = np.clip(pos[i], lb, ub)

print(f"Iter {t+1}/{its} Best score {a_score} Best pos: {a_pos}")

return a_score, a_pos

dim = 5

worst = 20

its = 50

lb = -10

ub = 10

best_score, best_pos = gwo(obj_func, dim, worst, its, lb, ub)

print ("\n Final Best score :", best_score)
print ("Final Best Pos :", best_pos)

output: Iter 1/50, Best Score : 8.083
Best Position : [1.381 -0.928 -1.662
1.494 0.624]

:

50 iterations
Final Best Score : $8.904 e^{-18}$
Final Best pos : [$1.2 e^{-05}$ $-1.6 e^{-05}$
$-4.72 e^{-06}$ $1.44 e^{-05}$ $-1.5 e^{-05}$]

8/11/24

6) Parallel Cellular Optimization

```
import numpy as np
import random
def objective_func(x):
    return np.sum(x**2)
def initialize_grid(grid_size, dim, bounds):
    return np.random.uniform(bounds[0],
bounds[1], (grid_size, grid_size, dim))
def evaluate_grid(grid, objective_func):
    fitness = np.zeros((grid.shape[0], grid.shape[1]))
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            fitness[i,j] = objective_func(grid[i,j])
    return fitness
def select_best_neighbour(grid, fitness, x, y):
    neighbours = [((x-1) % grid.shape[0], y),
((x+1) % grid.shape[0], y), (x, (y-1) %
grid.shape[1]), (x, (y+1) % grid.
shape[1])]
best_pos = min(neighbours, key=lambda
pos: fitness[pos[0], pos[1]])
    return grid[best_pos[0], best_pos[1]]

def crossover(parent1, parent2):
    alpha = np.random.rand()
    return alpha * parent1 + (1-alpha)
* parent2
```

```
def mutate(individual, bounds, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += np.random.uniform(-1, 1)
    individual[i] = np.clip(individual[i], bounds[0], bounds[1])
    return individual

def parallel_cellular_ga(objective_func, grid_size=5, dim=2, bounds=(-5, 5),
num_iter=100, mutation_rate=0.1):
    grid = initialize_grid(grid_size, dim, bounds)
    fitness = evaluate_grid(grid, obj_func)
for iteration in range(num_iter):
    new_grid = np.copy(grid)
    for i in range(grid_size):
        for j in range(grid_size):
            parent1 = grid[i,j],
            parent2 = select_best_neighbor
(grid, fitness, i, j)
    offspring = crossover(parent1, parent2)
    offspring = mutate(offspring, bounds,
mutation_rate)
offspring_fitness = objective_func(offspring)
    if offspring_fitness < fitness[i,j]:
```

new_grid[i, j] = offspring
fitness[i, j] = offspring.fitness
grid = new_grid

best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
best_fitness = fitness[best_position]
print(f"Iteration {iteration+1}: Best Fitness = {best_fitness}")
best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
return grid[best_position[0], best_position[1]], fitness[best_position]

grid_size = 5
dim = 2
bounds = (-5, 5)
num_iter = 50
mutation_rate = 0.1

best_sol, best_fitness = parallel_cellular_ga(objective_func, grid_size, dim, bounds, num_iter, mutation_rate)

print(f"\nBest solution: {best_sol}")
print(f"Best fitness: {best_fitness}")

Output: Iteration 1: Best fitness = 0.273339

Iteration 50: Best Fitness = 4.7740

Best Sol: [9.719568e-05, 6.04789e-05]

Best fitness = 6.29642 e-06

7) Gene expression and optimization

```
import random
import math

Population_size = 50
gene_length = 30
generations = 100
mutation_Rate = 0.05
crossover_rate = 0.7

terminals = ['x', '1', '2', '3', '4', '5']
functions = ['+', '-', '*', '/', 'sin', 'cos']
def cost_function(x):
    return x**2 - 10 * math.sin(2*x)
class GeneExpression:
    def __init__(self):
        self.gene = self.random_gene()
        self.cached_fitness = None
    def random_gene(self):
        return [random.choice(Terminals + Functions)
for _ in range(Gene_length)]
    def decode_gene(self, x):
        stack = []
        for token in self.gene:
            if token in terminals:
                stack.append(float(x) if token == 'x'
```

```
            else float(token))
        elif token in functions:
            if len(stack) >= 1 and token in ('sin', 'cos'):
                arg = stack.pop()
                stack.append(float(a) if token == 'sin'
else math.cos(arg))
            elif len(stack) >= 2:
                b, a = stack.pop(), stack.pop()
                if token == '+': stack.append(a+b)
                elif token == '-': stack.append(a-b)
                elif token == '*': stack.append(a*b)
                elif token == '/' and b != 0: stack.
append(a/b)
            else:
                return float('inf')
        return stack[0] if len(stack) == 1 else float('in

    def fitness(self, x):
        if self.cached_fitness is None:
            try:
                result = self.decode_gene(x)
                self.cached_fitness = abs(target_function(x) -
            except:
                self.cached_fitness = float('inf')
        return self.cached_fitness
```

```
def selection(population, fitness):
    tournament_size = 3
    candidates = random.sample(list(zip(population,
fitness)), tournament_size)
    return min(candidates, key=lambda x: x[1])[0]

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        point = random.randint(1, gene_length - 1)
        child1 = geneExpression()
        child2 = geneExpression()
        child1.gene = parent1.gene[:point] + parent2.
gene[point:]
        return child1, child2
    return parent1, parent2

def mutate(individual):
    for i in range(gene_length):
        if random.random() < mutation_rate:
            individual.gene[i] = random.choice(term
-nals + functions)

def geneExpression():
    population = [geneExpression() for _ in range
(population_size)]
    x_value = random.uniform(-10, 10)
```

```
for generation in range(generations):
    fitness = [ind.fitness(x_value) for ind in population]
    best_idx = fitness.index(min(fitness))
    print(f"generation {generation}: Best fitness =
{fitness[best_idx]:.5f}")

    new_population = [population[best_idx]]

    while len(new_population) < population_size:
        parent1 = selection(population, fitness)
        parent2 = selection(population, fitness)
        child1, child2 = crossover(parent1, parent2)
        mutate(child1)
        mutate(child2)
        new_population.extend([child1, child2])
    population = new_population

final_fitness = [ind.fitness(x_value) for ind in
    population]
best_idx = final_fitness.index(min(final_fitness))
print("\n optimized solution:")
print(f" Best Gene: {population[best_idx].gene}")
print(f"Best Fitness: {final_fitness[best_idx]:.5f}")

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

Output: generation 0: Best Fitness = inf
generation 4: Best fitness = 37.5664
Generation 13: Best Fitness = 26.06944
generation 22: Best Fitness = 8.88563
generation 24: Best Fitness = 4.40022
generation 26: Best Fitness = 0.35007
generation 29: Best Fitness = 0.11520
Generation 99: Best Fitness = 0.00015

Optimized Solution:
Best Gene: ['-', '5', '*', '4', 'sin', '*', '+', 'x', '5', '*', 'sin', '3', '-', 'x', '4', 'cos', 'f', '*', '-', '4', '5', '2', 'cos', '2', 'sin', 'cos']

Best Fitness = 0.00015