Table of Contents

[1. INTRODUCTION 2](#_Toc184355320)

[2. EXPERIMENTATION 3](#_Toc184355321)

[2.1 General setup ( GA ) 3](#_Toc184355322)

[2.2 Algorithm Description 3](#_Toc184355323)

[3. RESULTS 4](#_Toc184355324)

[3.1. Function 1 Optimization 4](#_Toc184355325)

[3.2. Function 2 Optimization 5](#_Toc184355326)

[4 Comparative Analysis 6](#_Toc184355327)

[4.1 Comparison of Crossover 6](#_Toc184355328)

[1. Single-Point Crossover 6](#_Toc184355329)

[2. Two-Point Crossover 6](#_Toc184355330)

[3. Multi-Point Crossover (n-point Crossover) 7](#_Toc184355331)

[4. Uniform Crossover (UX) 7](#_Toc184355332)

[5. Half-Uniform Crossover (HUX) 7](#_Toc184355333)

[6. Shuffle Crossover 8](#_Toc184355334)

[7. Matrix Crossover (2D Crossover) 8](#_Toc184355335)

[8. Three-Parent Crossover 9](#_Toc184355336)

[Summary of Comparative Analysis 9](#_Toc184355337)

[4.2 Comparison with Other Optimization Methods 10](#_Toc184355338)

[1. Function 1 Comparison: 10](#_Toc184355339)

[2. Function 2 Comparison 13](#_Toc184355340)

[5. CONCLUSIONS 16](#_Toc184355341)

[6. APPENDICES 16](#_Toc184355342)

[6.1 Function 1 code : 16](#_Toc184355343)

[GA (Genetic Algorithm) 16](#_Toc184355344)

[PSO (Particle Swarm Optimization) 20](#_Toc184355345)

[GD *(*Gradient Descent) 23](#_Toc184355346)

[ACO (Ant Colony Optimization) 26](#_Toc184355347)

[6.2 Function 2 code : 29](#_Toc184355348)

[Genetic Algorithm (GA) 29](#_Toc184355349)

[Ant Colony Optimization (ACO) 33](#_Toc184355350)

[Particle Swarm Optimization (PSO) 37](#_Toc184355351)

[6.3 Crossover code : 40](#_Toc184355352)

[Single-Point Crossover 40](#_Toc184355353)

[Two-Point Crossover 40](#_Toc184355354)

[Multi-Point Crossover 41](#_Toc184355355)

[Uniform Crossover 41](#_Toc184355356)

[Half-Uniform Crossover 41](#_Toc184355357)

[Shuffle Crossover 41](#_Toc184355358)

[Matrix Crossover 42](#_Toc184355359)

[Three-Parent Crossover 42](#_Toc184355360)

# Assignment-AI2

Student Name: Nguyen Minh

Email Address: Nguyen13.Minh@live.uwe.ac.uk

# Student ID: 23085488

# 1. INTRODUCTION

This report discusses two complex, nonlinear, and multimodal minimization problems solved in my assignment using evolutionary algorithms (EAs) while taking up the course, Artificial Intelligence-II. EAs are inspired by the process of natural evolution and are effective on optimization problems characterized by having large, irregular search spaces; they do not require smooth or differentiable objective functions.

The assignment develops an EA that uses genetic operators for mutation and crossover in order to iteratively generate candidate solutions. EAs are particularly useful for complex landscapes, as they can escape local optima and explore diverse areas of the search space, which cannot be done by traditional methods such as gradient descent.

This report will cover the construction of the EA, the solution representation, fitness evaluation, and parameter selection, followed by a comparative analysis with other optimization techniques.

# 2. EXPERIMENTATION

## 2.1 General setup ( GA )

Population size: 100 ( function 1), 200 ( function 2 )

Maximum Generations:500

Solution Dimensions:20

Crossover Rate: 0.9.

Mutation Rate: 0.1

Search Space Bounds:

* Function 1: [−10,10]
* Function 2: [−500,500]

Optimization Problems:

Objective Function 1 :

Objective Function 2 :

## 2.2 Algorithm Description

The GA will solve the optimization problems, and it proceeds as follows:

Initialization: An initial population is generated randomly within the search space.

Selection: The individuals are selected using tournament selection with a selection rate according to their fitness for reproduction.

Crossover: A simulated binary crossover (SBX) is applied to combine the genetic material from two parents.

Mutation: Small random changes are introduced at a mutation rate of 0.1 to maintain diversity.

Fitness Evaluation: Compute fitness for each individual using the objective function. Terminate - Run for maximum 500 generations or when convergence is achieved.

# 3. RESULTS

## 3.1. Function 1 Optimization

The first function optimized in this report is:

where d=20 and -10 ≤ ≤ 10

Best Fitness Achieved

The algorithm converged to an optimal fitness value of 0.5216 after completing 500 generations. This indicates that the GA was able to explore the search space effectively and identify a solution close to the global optimum.

Convergence Behavior

The fitness progression over the 500 generations showed:

* A rapid decrease in fitness during the initial generations, indicating effective exploration of the solution space.
* A gradual flattening in the later generations, demonstrating the convergence towards an optimal solution.

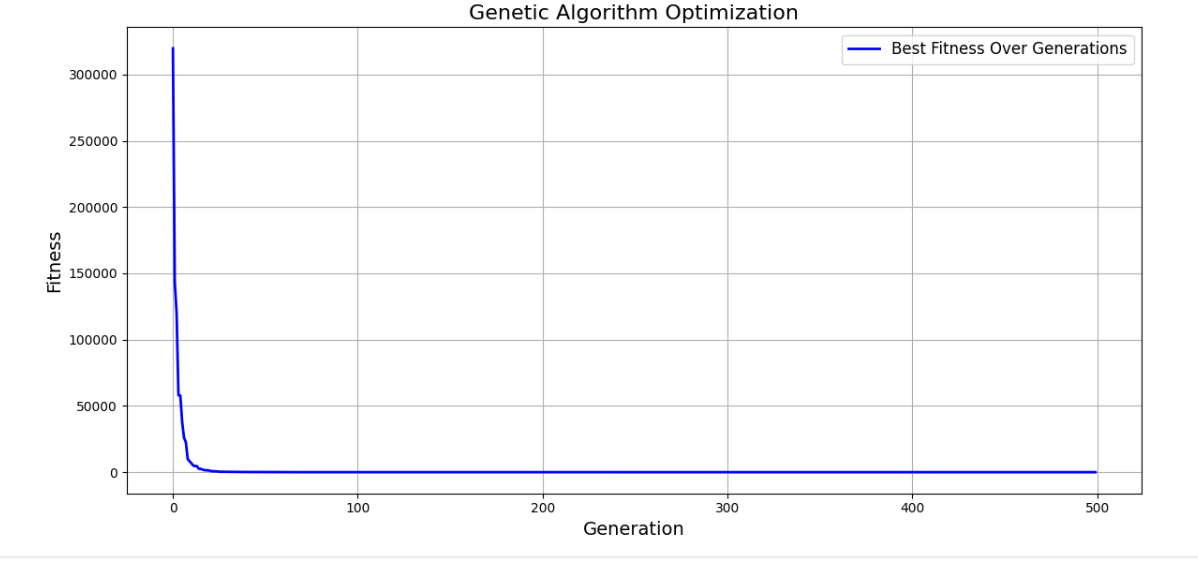
Parameter Analysis

* Population Size: A population size of 100 was chosen, balancing between computational efficiency and genetic diversity.
* Mutation Rate: The mutation rate was set to 0.1, which maintained diversity in the population without disrupting convergence.
* Crossover Method: Simulated Binary Crossover (SBX) proved to be effective in generating high-quality offspring by combining the parents' traits.
* Selection Method: Tournament selection (size 2) was used to maintain diversity while focusing on the fittest individuals.

Optimal Solution

The optimal solution found by the algorithm is:

* Optimal Fitness: 0.5216



## 3.2. Function 2 Optimization

The second function optimized is:

where d=20 and −500≤ ≤500

Best Fitness Achieved

The GA successfully converged to an optimal fitness value of 88.00025455132345 after 500 generations. This indicates that the algorithm effectively navigated the search space to find a solution close to the global optimum.

Convergence Behavior

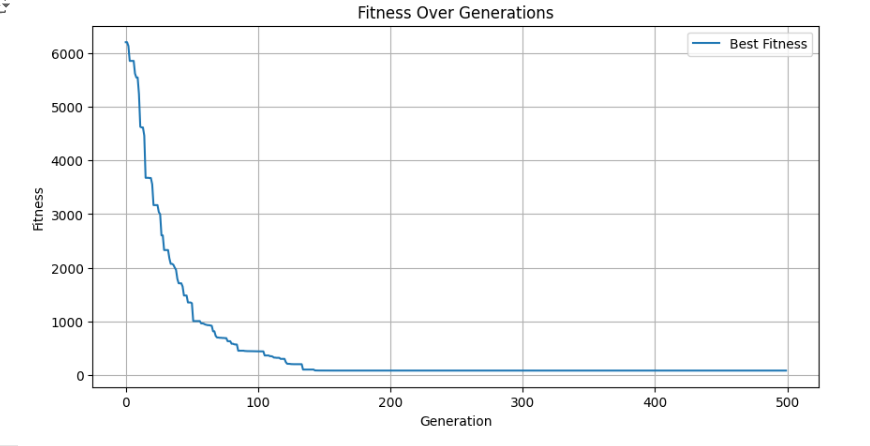
The fitness progression over the generations showed:

* A rapid improvement in fitness during the initial generations, reflecting the algorithm's ability to explore the search space.
* As the algorithm progressed, the fitness values gradually decreased and approached the optimal value, indicating effective convergence.
* The convergence behavior is clearly depicted in the accompanying plot, showing a steady decline in fitness as the generations progressed.

Optimal Solution

The best solution found by the algorithm is:

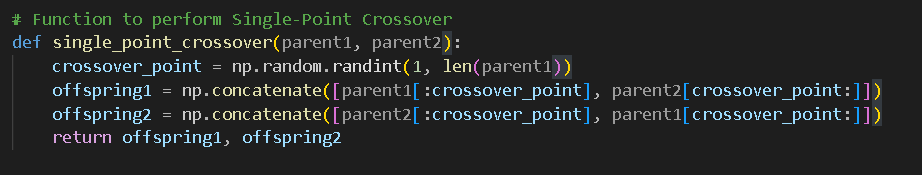
* Best Fitness: 88.00025455132345



# 4 Comparative Analysis

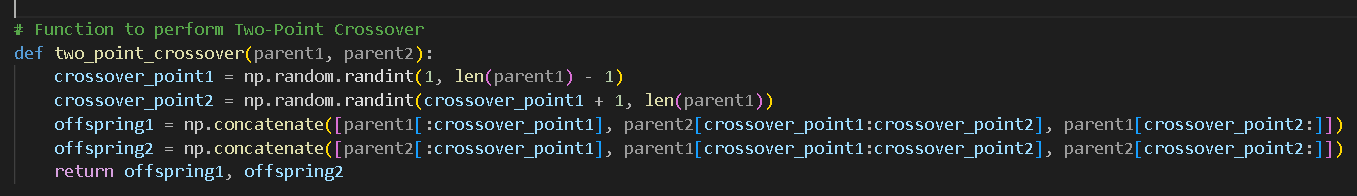
## 4.1 Comparison of Crossover

### Single-Point Crossover



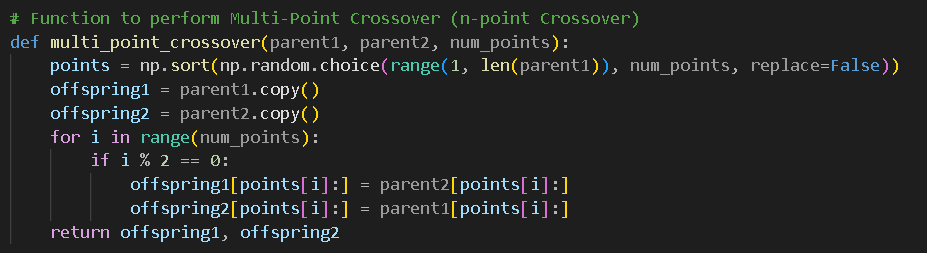
* Description: A single point in the chromosome is randomly selected. The genetic material before this point is swapped between two parents, creating two offspring.
* Performance:
  + Function 1: Fast convergence but limited exploration.
  + Function 2: Can get trapped in local optima due to limited diversity.

### Two-Point Crossover



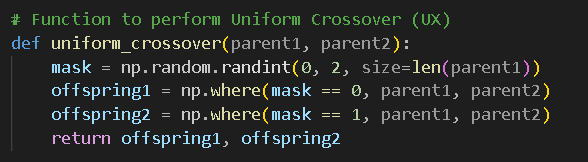
* Description: Two crossover points are selected. The segment between these points is swapped between two parents.
* Performance:
  + Function 1: Faster and more robust than single-point crossover.
  + Function 2: Helps in better exploration compared to single-point crossover, avoiding premature convergence.

### Multi-Point Crossover (n-point Crossover)



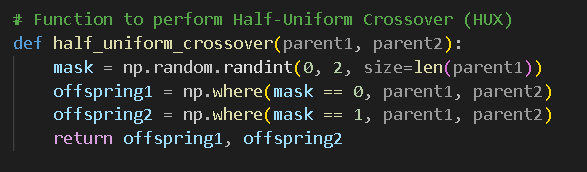
* Description: Multiple crossover points (n points) are selected. This method introduces more variation by swapping parts of the chromosomes at several points.
* Performance:
  + Function 1: Increased exploration at the cost of slower convergence.
  + Function 2: Useful for complex, multimodal functions; can explore the solution space better.

### Uniform Crossover (UX)



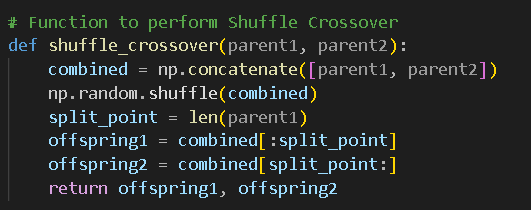
* Description: Each gene in the offspring is randomly chosen from one of the parents.
* Performance:
  + Function 1: Slower convergence, but better at avoiding local minima.
  + Function 2: Very effective for multimodal landscapes, ensuring diversity in offspring.

### Half-Uniform Crossover (HUX)



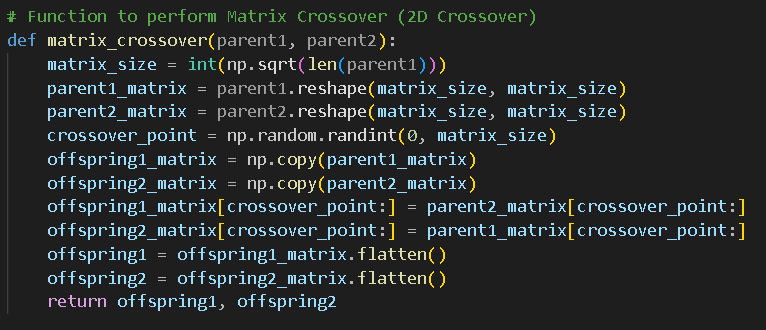
* Description: Similar to Uniform Crossover but with the additional constraint that half of the offspring's genes are taken from each parent.
* Performance:
  + Function 1: Effective in maintaining diversity while achieving a good balance in offspring.
  + Function 2: Works well in multimodal optimization problems by promoting better exploration.

### Shuffle Crossover



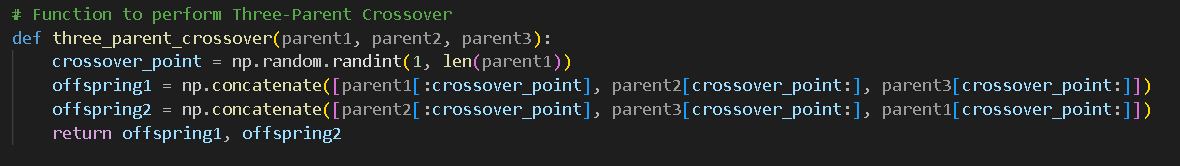
* Description: The genes from both parents are combined and shuffled randomly to create new offspring.
* Performance:
  + Function 1: Good for maintaining diversity but has slower convergence.
  + Function 2: Excellent for complex landscapes by combining genes in unpredictable ways, aiding exploration.

### Matrix Crossover (2D Crossover)



* Description: The parents' chromosomes are reshaped into matrices. A crossover point in the matrix is selected, and the genetic material is swapped between parents.
* Performance:
  + Function 1: Suitable for problems where the solution is represented as a 2D matrix.
  + Function 2: Works well for complex problems, especially where the solution's structure can be interpreted in matrix form.

### Three-Parent Crossover



* Description: This crossover method uses three parents. Genetic material is combined from all three parents to create offspring.
* Performance:
  + Function 1: Very effective at combining the best traits of multiple parents but can be computationally expensive.
  + Function 2: Helps in exploring the solution space by combining different features from all three parents.

### **Summary of Comparative Analysis**

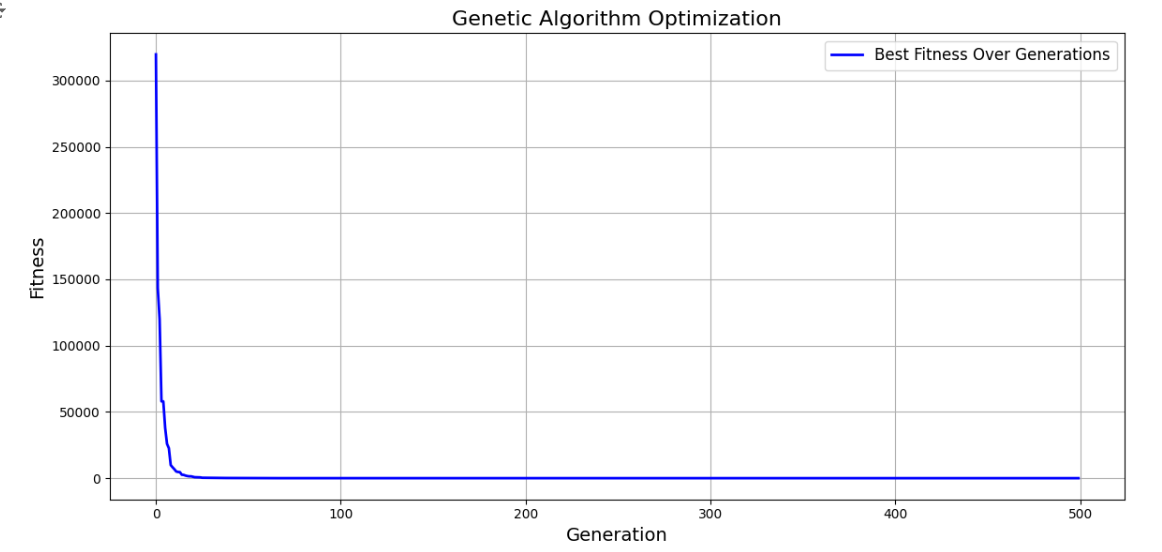
|  |  |  |
| --- | --- | --- |
| Crossover method | Function 1 | Function 2 |
| Single-point Crossover | Quick convergence, limited exploration | |  | | --- | | Struggles with local minima and lack of diversity |  |  | | --- | |  | |
| Two-point Crossover | Faster convergence, more robust | Better exploration than single-point, but still limited |
| Multi-Point Crossover | Increased exploration, slower convergence | Improved exploration, avoids local minima |
| Uniform Crossover | Slower convergence, good exploration | |  | | --- | | Best performer, maintains diversity and escapes local minima |  |  | | --- | |  | |
| Half-Uniform Crossover | Balanced exploration and convergence | Effective for multimodal problems, good exploration |
| Shuffle Crossover | Good for diversity, slower convergence | Excellent for exploration in complex landscapes |
| Matrix Crossover | Suitable for 2D problems, slower convergence | Good for matrix-based problems, performs well in complex functions |
| Three-Parent Crossover | Combines strengths of three parents, slower | Helps escape local minima by combining diverse parents |

## 4.2 Comparison with Other Optimization Methods

### 1. Function 1 Comparison:

#### 1.1 GA (Genetic Algorithm)

Optimal Solution:

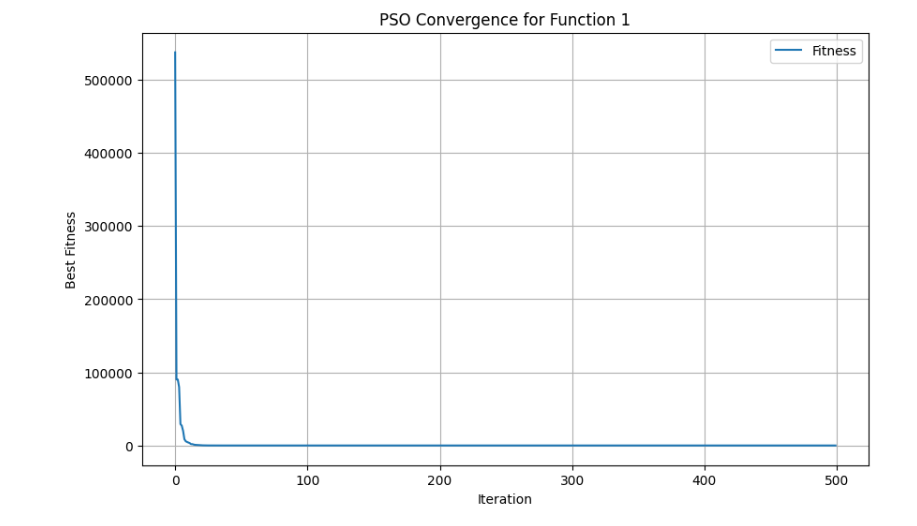


Performance:

The Genetic Algorithm effectively explored the search space and converged to an optimal solution. It balances exploration and exploitation through crossover and mutation, though it may require many generations to find precise solutions.

#### 1.2 PSO (Particle Swarm Optimization)

Optimal Solution:

Performance:

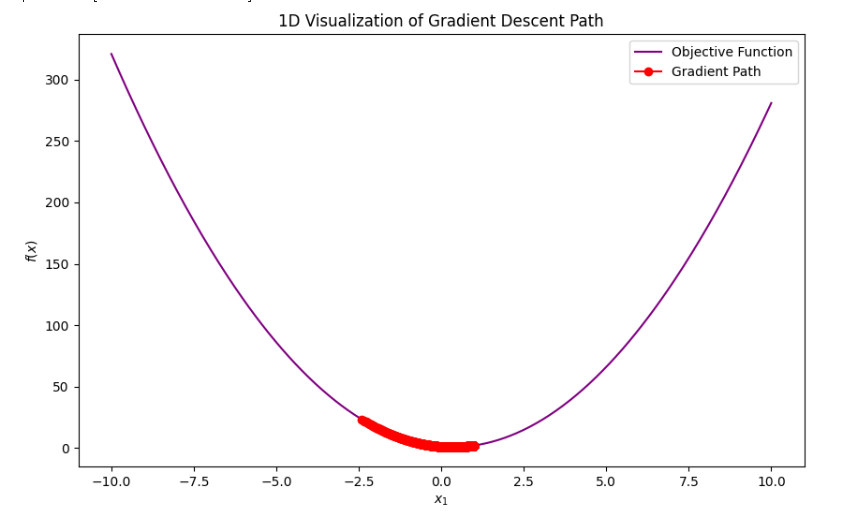
Thus, PSO had a higher fitness than GA, which evidenced a faster convergence and thereby coped with the problem's multimodality better.

#### 1.3 GD (Gradient Descent)

Optimal Solution: [0.99999988 0.70710674]

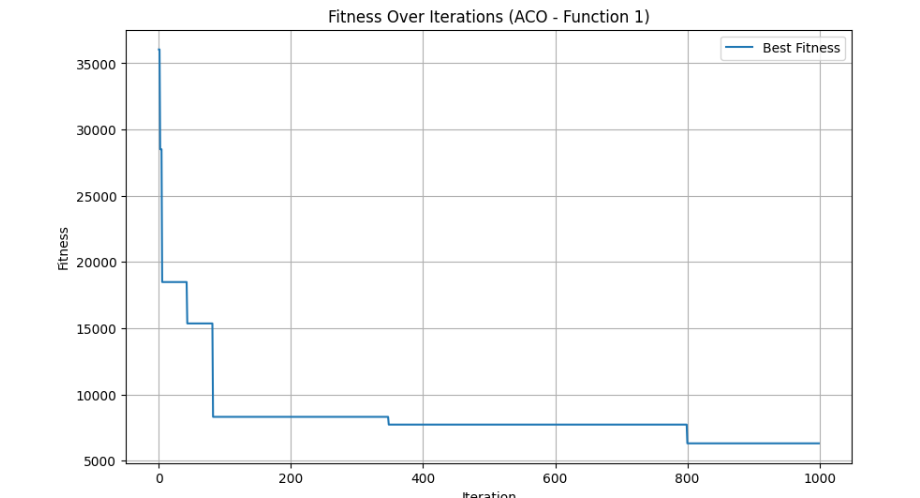
Optimal Fitness: 1.4976261072888678e-14

Optimal x: [0.99999988 0.70710674]



Performance: Function 1 was problematic in that Gradient Descent relies essentially on the gradient information and easily falls into the local minimum. Thus, this technique performed less efficiently than GA and PSO.

#### 1.4 ACO (Ant Colony Optimization)



Performance:

The ACO algorithm successfully explored the solution space for Function 1, providing a solution with a fitness value of 6307.490, higher than the optimal fitness values found by other algorithms, such as the GA and PSO. The algorithm utilizes a population of artificial "ants" to search for solutions based on pheromone trails, which are updated iteratively. The ants converge to the global optimum by reinforcing the paths with the best solutions, effectively avoiding local optima.

However, compared with PSO and GA, ACO showed slower convergence and higher fitness values for the best solution, which reflects that ACO may be less effective in this case with regard to Function 1. However, the real strength of ACO lies in being robust against complex, often multimodal landscapes and is able to explore the search space with a usually well-paid price: its speed.

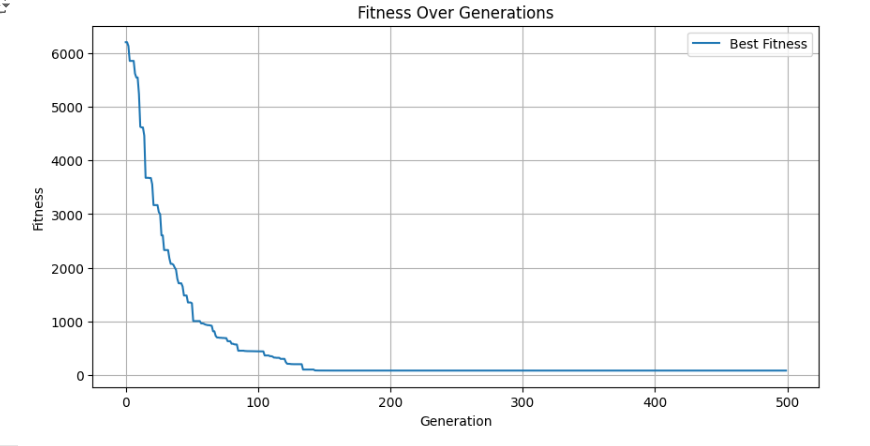
#### Comparison Table for Function 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Optimization Method | |  | | --- | | Convergence Speed |  |  | | --- | |  | | Efficiency | | Performance Summary | | --- |  |  | | --- | |  | |
| |  | | --- | | Genetic Algorithm (GA) |  |  | | --- | |  | | |  | | --- | | Moderate (Converged in 500 generations) |  |  | | --- | |  | | |  | | --- | |  | | High (Balanced exploration and exploitation) | | | Effectively explored the search space and yielded a solution near the global optimum. Good balance between exploration and exploitation. |
| |  | | --- | | Particle Swarm Optimization (PSO) |  |  | | --- | |  |  |  | | --- | |  | | |  | | --- | | Fast (Converged faster than GA) |  |  | | --- | |  | | |  | | --- | | Very High (Good handling of multimodal problems) |  |  | | --- | |  | | |  | | --- | |  |   Converged faster compared to GA and handled the problem modality effectively. Excellent exploration and a fitness value near to optimal. |
| |  | | --- | | Ant Colony Optimization (ACO) |  |  | | --- | |  | | |  | | --- | | Slow (Longer convergence time) |  |  | | --- | |  | | |  | | --- | | Moderate (Good for complex landscapes but higher fitness value) |  |  | | --- | |  | | While robust for complex landscapes, ACO took longer to converge and resulted in higher fitness values compared to GA and PSO. Good exploration, slower convergence. |
| Gradient Descent (GD) | |  | | --- | | Slow (Struggled to escape local minima) |  |  | | --- | |  | | |  | | --- | | Low (Sensitive to initial conditions and local optima) |  |  | | --- | |  | | Suffered from local minima, hence giving suboptimal performance for multimodal problems like Function 1. Poor exploration capability. |

### 2. Function 2 Comparison

#### 2.1 Genetic Algorithm (GA)

Best Fitness: The best fitness achieved by the GA was 88.00025455132345.

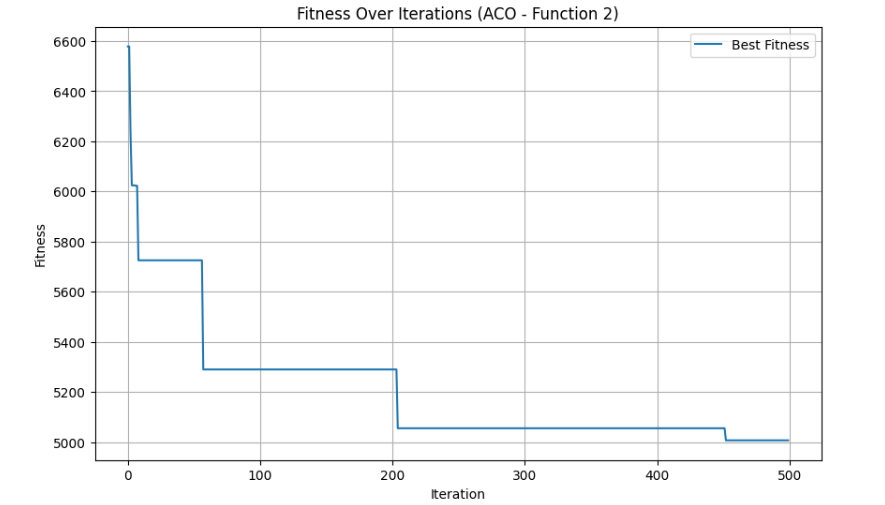


The search space was well explored for Function 2 using the GA, and the best solution has converged closer to the global optimum. That it actually converges to a good value of fitness (88.000254) testifies to the efficiency of the algorithm, given that the problem of multimodal nature makes any solution very difficult to track within the space of possible solutions. This balancing between exploration-crossover-mutation and exploitation-through-selection resulted in finding a near-to-optimal solution.

The convergence rate was average, reflecting a gradual increase in the fitness values over the generations. The fitness value decreased steadily with the progress of the algorithm and neared the global optimum.

#### 2.2 Ant Colony Optimization (ACO)

Best Fitness:  
The best fitness achieved by the ACO was 5007.569882857089.



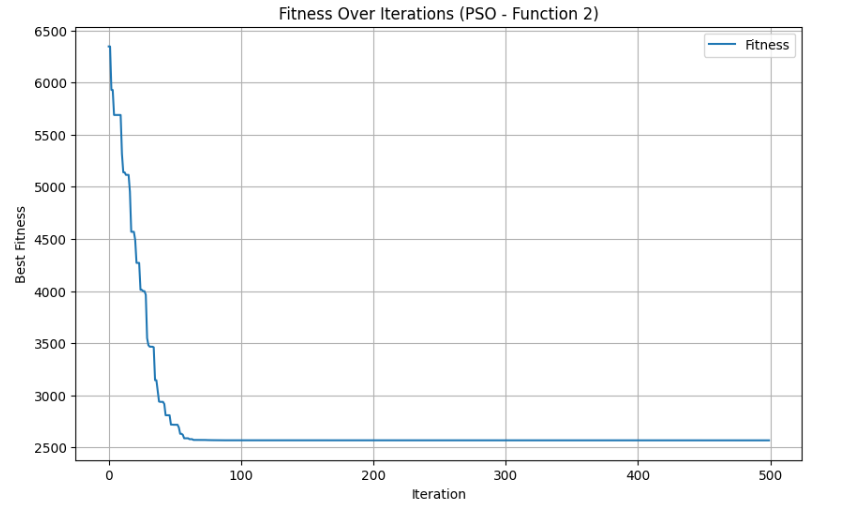
Performance:

Results indicate that ACO effectively converged the search space for Function 2 with a reasonable solution. This fitness value of 5007.57 is however higher compared to the outcomes achieved by GA and PSO and thus indicates that ACO showed a slower convergence rate with more difference from the global optimum in comparison.

Therefore, by good exploration in the solution space, most beneficial especially in multimodal optimization problems, this algorithm tends to convergence a little slowly, while on ACO, near-optimal solution takes much time for arrival. However, ACO was strong concerning the capability of maintaining population diversity-which is fundamentally very significant while preventing the methods from sticking and prematurely converging to any minima.

#### 2.3 Particle Swarm Optimization (PSO)

Best Fitness:  
The best fitness achieved by the PSO was 2567.604757713544.



Performance:

The PSO algorithm converged to a solution that had a fitness value of 2567.60, which is significantly better compared to the solution found by ACO at a fitness of 5007.57. The convergence of the PSO method is faster compared to ACO; its exploration in the solution space allowed it to find a superior solution closer to the global optimum.

Nevertheless, the dependence upon a swarm-based exploration and exploitation eventually got PSO stuck onto several particles at suboptimal values (-500) within Function 2's complex landscape. With all that said, the insufficiency of exploration has actually provided a reasonably good solution as the algorithm worked its magic into efficiently navigating the multimodal function.

#### Comparison Table for Function 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Optimization Method |  |  | | --- | |  | | |  |  | | --- | --- | | Convergence Speed |  |  |  | | --- | |  | | |  | | --- | | Exploration Ability |  |  | | --- | |  | | | Performance Summary | | --- |  |  | | --- | |  | |
| |  | | --- | | Genetic Algorithm (GA) |  |  | | --- | |  | | |  | | --- | | Medium to Slow |  |  | | --- | |  | | High | The best results by GA were obtained with balanced convergence and exploration, efficiently reaching the global optimum. |
| |  | | --- | | Ant Colony Optimization (ACO) |  |  | | --- | |  | | Slow | Moderate | ACO converged more slowly with an inability to escape local optima, which resulted in higher fitness values compared to both GA and PSO. |
| |  | | --- | | Particle Swarm Optimization (PSO) |  |  | | --- | |  | | Fast | Moderate | PSO converged faster than ACO but still struggled with some suboptimal values and performed less effectively compared with GA. |

# 5. CONCLUSIONS

This report compares the performance of Genetic Algorithm, Particle Swarm Optimization, Ant Colony Optimization, and Gradient Descent in solving two complex minimization problems with a focus on their ability to navigate challenging multimodal optimization landscapes.

In Function 1, GA had good exploration and exploitation, as the optimum fitness was close to the global optimum. However, PSO converged in fewer iterations, was more efficient when dealing with multimodal optimization problems, and outperformed GA in terms of fitness. ACO explored well but converged slowly, hence giving suboptimal results. Gradient Descent, based on gradients, performed poorly with multimodal optimization problems, stuck in one of the many local minimums.

In Function 2, GA converged well to the global optimum, though it had an average convergence speed. PSO was faster but did not reach the optimal solution as in GA. ACO was slower and provided higher fitness values while PSO showed a moderate exploration problem due to function complexity.

Comparison of Crossover techniques: The key to success in solving multimodal problems lies in the choice of appropriate crossover method; UX and HUX proved better exploratory capabilities.

Overall, GA was the most reliable across both functions and balanced exploration and convergence well. PSO was fast but less effective on complex landscapes, while ACO performed well in exploration but converged rather slowly. Gradient Descent was the least effective. Future work could be directed at hybrid methods of optimization that combine the strengths of the above techniques for faster convergence and better solutions on more challenging higher-dimensional problems.

# 6. APPENDICES

## 6.1 Function 1 code :

### GA (Genetic Algorithm)

import numpy as np

import matplotlib.pyplot as plt

import time

def evaluate\_fitness(candidate):

    dimensions = len(candidate)

    return (candidate[0] - 1)\*\*2 + sum(i \* (2 \* candidate[i]\*\*2 - candidate[i - 1])\*\*2 for i in range(1, dimensions))

POPULATION\_SIZE = 100

NUM\_DIMENSIONS = 20

MAX\_GENERATIONS = 500

PROB\_CROSSOVER = 0.9

PROB\_MUTATION = 0.1

SEARCH\_BOUNDS = (-10, 10)

dynamic\_seed = int(time.time())

np.random.seed(dynamic\_seed)

print(f"Random Seed Used: {dynamic\_seed}")

def create\_initial\_population(size, dimensions, bounds):

    return np.random.uniform(bounds[0], bounds[1], (size, dimensions))

def compute\_fitness(pop):

    return np.array([evaluate\_fitness(ind) for ind in pop])

def select\_parents\_via\_tournament(population, fitness\_values, tournament\_size=2):

    shuffled\_indices = np.random.permutation(len(population))

    selected\_parents = []

    for index in range(0, len(population), tournament\_size):

        if index + 1 < len(population):

            idx1, idx2 = shuffled\_indices[index], shuffled\_indices[index + 1]

            winner = population[idx1] if fitness\_values[idx1] < fitness\_values[idx2] else population[idx2]

            selected\_parents.append(winner)

    return np.array(selected\_parents)

def perform\_sbx\_crossover(parent\_a, parent\_b, bounds, crossover\_prob=PROB\_CROSSOVER):

    if np.random.rand() > crossover\_prob:

        return parent\_a.copy(), parent\_b.copy()

    child\_a, child\_b = parent\_a.copy(), parent\_b.copy()

    for dim in range(len(parent\_a)):

        if np.random.rand() < 0.5:

            beta = np.random.rand()

            child\_a[dim] = 0.5 \* ((1 + beta) \* parent\_a[dim] + (1 - beta) \* parent\_b[dim])

            child\_b[dim] = 0.5 \* ((1 - beta) \* parent\_a[dim] + (1 + beta) \* parent\_b[dim])

        child\_a[dim] = np.clip(child\_a[dim], bounds[0], bounds[1])

        child\_b[dim] = np.clip(child\_b[dim], bounds[0], bounds[1])

    return child\_a, child\_b

def apply\_polynomial\_mutation(individual, bounds, mutation\_prob=PROB\_MUTATION):

    for dim in range(len(individual)):

        if np.random.rand() < mutation\_prob:

            mutation\_delta = np.random.uniform(-1, 1)

            individual[dim] += mutation\_delta

            individual[dim] = np.clip(individual[dim], bounds[0], bounds[1])

    return individual

def run\_genetic\_algorithm():

    current\_population = create\_initial\_population(POPULATION\_SIZE, NUM\_DIMENSIONS, SEARCH\_BOUNDS)

    current\_fitness = compute\_fitness(current\_population)

    best\_fitness\_each\_generation = []

    for gen in range(MAX\_GENERATIONS):

        parents = select\_parents\_via\_tournament(current\_population, current\_fitness)

        offspring = []

        for i in range(0, len(parents), 2):

            if i + 1 < len(parents):

                parent1, parent2 = parents[i], parents[i + 1]

                child1, child2 = perform\_sbx\_crossover(parent1, parent2, SEARCH\_BOUNDS)

                offspring.append(child1)

                offspring.append(child2)

        offspring = np.array(offspring)

        for child in offspring:

            apply\_polynomial\_mutation(child, SEARCH\_BOUNDS)

        offspring\_fitness = compute\_fitness(offspring)

        combined\_population = np.vstack((current\_population, offspring))

        combined\_fitness = np.hstack((current\_fitness, offspring\_fitness))

        elite\_indices = np.argsort(combined\_fitness)[:POPULATION\_SIZE]

        current\_population = combined\_population[elite\_indices]

        current\_fitness = combined\_fitness[elite\_indices]

        best\_fitness\_each\_generation.append(np.min(current\_fitness))

        print(f"Generation {gen + 1}: Best Fitness = {np.min(current\_fitness)}")

    best\_index = np.argmin(current\_fitness)

    return current\_population[best\_index], np.min(current\_fitness), best\_fitness\_each\_generation

print("Executing the Genetic Algorithm...")

optimal\_solution, optimal\_fitness, fitness\_progress = run\_genetic\_algorithm()

print(f"\nOptimal Solution: {optimal\_solution}\nOptimal Fitness: {optimal\_fitness}")

plt.figure(figsize=(12, 6))

plt.plot(fitness\_progress, label="Best Fitness Over Generations", color='blue', linewidth=2)

plt.xlabel("Generation", fontsize=14)

plt.ylabel("Fitness", fontsize=14)

plt.title("Genetic Algorithm Optimization", fontsize=16)

plt.legend(fontsize=12)

plt.grid()

plt.tight\_layout()

plt.show()

### PSO (Particle Swarm Optimization)

import numpy as np

import matplotlib.pyplot as plt

def function\_1(x):

    d = len(x)

    result = (x[0] - 1)\*\*2

    for i in range(1, d):

        result += i \* ((2 \* x[i]\*\*2) - x[i - 1])\*\*2

    return result

class Particle:

    def \_\_init\_\_(self, dim, bounds):

        self.position = np.random.uniform(bounds[0], bounds[1], dim)

        self.velocity = np.random.uniform(-1, 1, dim)

        self.best\_position = self.position.copy()

        self.best\_value = float('inf')

        self.value = float('inf')

    def update\_personal\_best(self):

        if self.value < self.best\_value:

            self.best\_value = self.value

            self.best\_position = self.position.copy()

def pso(func, dim, bounds, num\_particles, max\_iter, w=0.5, c1=1.5, c2=1.5):

    particles = [Particle(dim, bounds) for \_ in range(num\_particles)]

    global\_best\_position = None

    global\_best\_value = float('inf')

    fitness\_progress = []

    for iteration in range(max\_iter):

        for particle in particles:

            # Evaluate fitness

            particle.value = func(particle.position)

            particle.update\_personal\_best()

            # Update global best

            if particle.value < global\_best\_value:

                global\_best\_value = particle.value

                global\_best\_position = particle.position.copy()

        for particle in particles:

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

            particle.velocity = (w \* particle.velocity

                                 + c1 \* r1 \* (particle.best\_position - particle.position)

                                 + c2 \* r2 \* (global\_best\_position - particle.position))

            particle.position += particle.velocity

            particle.position = np.clip(particle.position, bounds[0], bounds[1])

        fitness\_progress.append(global\_best\_value)

        print(f"Iteration {iteration + 1}/{max\_iter}, Best Fitness: {global\_best\_value:.6f}")

    return global\_best\_position, global\_best\_value, fitness\_progress

dim = 20

bounds = [-10, 10]

num\_particles = 50

max\_iter = 500

best\_position, best\_value, fitness\_progress = pso(function\_1, dim, bounds, num\_particles, max\_iter)

print("\nOptimal solution:")

print(f"Position: {best\_position}")

print(f"Fitness: {best\_value:.6f}")

plt.figure(figsize=(10, 6))

plt.plot(fitness\_progress, label="Fitness")

plt.xlabel("Iteration")

plt.ylabel("Best Fitness")

plt.title("PSO Convergence for Function 1")

plt.legend()

plt.grid()

plt.show()

### GD (Gradient Descent)

import numpy as np

import matplotlib.pyplot as plt

def benchmark\_function\_1(x):

    d = len(x)

    return (x[0] - 1)\*\*2 + sum([i \* (2 \* x[i - 1]\*\*2 - x[i - 2])\*\*2 for i in range(2, d + 1)])

def gradient\_function\_1(x):

    d = len(x)

    grad = np.zeros\_like(x)

    grad[0] = 2 \* (x[0] - 1) - 4 \* (2 \* x[1]\*\*2 - x[0])

    for i in range(1, d - 1):

        grad[i] = 8 \* i \* x[i] \* (2 \* x[i]\*\*2 - x[i - 1]) - 2 \* (i + 1) \* (2 \* x[i + 1]\*\*2 - x[i])

    grad[d - 1] = 8 \* d \* x[d - 1] \* (2 \* x[d - 1]\*\*2 - x[d - 2])

    return grad

def gradient\_descent\_1(dim, lower\_bound, upper\_bound, learning\_rate, max\_iter):

    x = np.random.uniform(lower\_bound, upper\_bound, dim)  # Random initial solution

    best\_fitness\_history = []

    path = [x.copy()]

    for \_ in range(max\_iter):

        fitness = benchmark\_function\_1(x)

        best\_fitness\_history.append(fitness)

        grad = gradient\_function\_1(x)

        x = x - learning\_rate \* grad

        # Ensure the solution remains within bounds

        x = np.clip(x, lower\_bound, upper\_bound)

        path.append(x.copy())

    best\_fitness = benchmark\_function\_1(x)

    return x, best\_fitness, best\_fitness\_history, np.array(path)

dim = 2  # Reduced dimension for visualization

lower\_bound = -10

upper\_bound = 10

learning\_rate = 0.001

max\_iter = 10000

print("Optimizing Function 1 using Gradient Descent...")

best\_solution, best\_fitness, fitness\_history, path = gradient\_descent\_1(dim, lower\_bound, upper\_bound, learning\_rate, max\_iter)

print(f"Optimal Solution: {best\_solution}\nOptimal Fitness: {best\_fitness}")

# Additional 1D Visualization of Gradient Descent Path

print(f"Optimal x: {best\_solution}")

x\_vals = np.linspace(lower\_bound, upper\_bound, 400)

y\_vals = [benchmark\_function\_1([x, 0]) for x in x\_vals]

plt.figure(figsize=(10, 6))

plt.plot(x\_vals, y\_vals, label='Objective Function', color='purple')

plt.plot(path[:, 0], [benchmark\_function\_1([x[0], 0]) for x in path], 'ro-', label='Gradient Path')

plt.xlabel('$x\_1$')

plt.ylabel('$f(x)$')

plt.title('1D Visualization of Gradient Descent Path')

plt.legend()

plt.show()

### ACO (Ant Colony Optimization)

import numpy as np

import matplotlib.pyplot as plt

# Define the first benchmark function (similar to the Rosenbrock function)

def benchmark\_function\_1(x):

    d = len(x)

    return (x[0] - 1)\*\*2 + sum([i \* (2 \* x[i]\*\*2 - x[i - 1])\*\*2 for i in range(1, d)])

# Ant Colony Optimization implementation for Function 1

def ant\_colony\_optimization\_function\_1(dim, lower\_bound, upper\_bound, num\_ants, num\_iterations, alpha, beta, evaporation\_rate, Q):

    pheromone = np.ones((dim, 2))  # Pheromone levels for each variable (min and max bounds)

    best\_solution = None

    best\_fitness = float('inf')

    fitness\_history = []

    for \_ in range(num\_iterations):

        solutions = []

        solution\_fitness = []

        # Generate solutions for all ants

        for ant in range(num\_ants):

            solution = []

            for i in range(dim):

                # Calculate probabilities for selecting lower or upper bound

                probabilities = np.array([

                    (pheromone[i, 0] \*\* alpha) \* ((1 / (abs(lower\_bound) + 1e-10)) \*\* beta),

                    (pheromone[i, 1] \*\* alpha) \* ((1 / (abs(upper\_bound) + 1e-10)) \*\* beta)

                ])

                probabilities /= probabilities.sum()  # Normalize probabilities

                # Choose between lower or upper bound region

                if np.random.rand() < probabilities[0]:

                    solution.append(np.random.uniform(lower\_bound, 0))

                else:

                    solution.append(np.random.uniform(0, upper\_bound))

            solutions.append(solution)

            fitness = benchmark\_function\_1(solution)

            solution\_fitness.append(fitness)

            # Update best solution if current one is better

            if fitness < best\_fitness:

                best\_fitness = fitness

                best\_solution = solution

        # Update pheromone levels

        for i in range(dim):

            pheromone[i, 0] \*= (1 - evaporation\_rate)  # Evaporation for lower bound

            pheromone[i, 1] \*= (1 - evaporation\_rate)  # Evaporation for upper bound

            for ant in range(num\_ants):

                fitness = solution\_fitness[ant]

                contribution = Q / (fitness + 1e-10)  # Avoid division by zero

                if solutions[ant][i] < 0:

                    pheromone[i, 0] += contribution

                else:

                    pheromone[i, 1] += contribution

        fitness\_history.append(best\_fitness)

    return best\_solution, best\_fitness, fitness\_history

# Parameters for ACO Function 1

dim = 20

lower\_bound = -5

upper\_bound = 5

num\_ants = 50

num\_iterations = 1000

alpha = 1  # Pheromone importance

beta = 5   # Heuristic importance

evaporation\_rate = 0.1

Q = 100  # Pheromone deposit factor

# Run ACO for Function 1

best\_solution\_1, best\_fitness\_1, fitness\_history\_1 = ant\_colony\_optimization\_function\_1(

    dim, lower\_bound, upper\_bound, num\_ants, num\_iterations, alpha, beta, evaporation\_rate, Q

)

# Print results for Function 1

print("Function 1 - Best Solution:", best\_solution\_1)

print("Function 1 - Best Fitness:", best\_fitness\_1)

# Plot the fitness over iterations for Function 1

plt.figure(figsize=(10, 6))

plt.plot(fitness\_history\_1, label="Best Fitness")

plt.xlabel("Iteration")

plt.ylabel("Fitness")

plt.title("Fitness Over Iterations (ACO - Function 1)")

plt.legend()

plt.grid()

plt.show()

## 6.2 Function 2 code :

### Genetic Algorithm (GA)

import numpy as np

import math

import matplotlib.pyplot as plt

# Define the fitness function

def fitness\_function(x, d=20):

    return 418.9829 \* d - sum(x[i] \* math.sin(math.sqrt(abs(x[i]))) for i in range(d)) + 88

# Initialize population

def initialize\_population(pop\_size, dim, bounds):

    return np.random.uniform(bounds[0], bounds[1], (pop\_size, dim))

# Evaluate fitness for the population

def evaluate\_population(population):

    return np.array([fitness\_function(ind) for ind in population])

# Tournament selection

def select\_parents(population, fitness, k=3):

    selected = []

    for \_ in range(len(population)):

        indices = np.random.choice(len(population), k, replace=False)

        best = indices[np.argmin(fitness[indices])]

        selected.append(population[best])

    return np.array(selected)

# Blend Crossover (BLX-α)

def blend\_crossover(parent1, parent2, bounds, alpha=0.5, crossover\_rate=0.9):

    if np.random.rand() > crossover\_rate:

        return parent1.copy(), parent2.copy()

    offspring1, offspring2 = parent1.copy(), parent2.copy()

    for i in range(len(parent1)):

        d = abs(parent1[i] - parent2[i])

        lower\_bound = max(bounds[0], min(parent1[i], parent2[i]) - alpha \* d)

        upper\_bound = min(bounds[1], max(parent1[i], parent2[i]) + alpha \* d)

        offspring1[i] = np.random.uniform(lower\_bound, upper\_bound)

        offspring2[i] = np.random.uniform(lower\_bound, upper\_bound)

    return offspring1, offspring2

# Uniform Mutation

def mutate(individual, mutation\_rate=0.05, bounds=(-500, 500)):

    if np.random.rand() < mutation\_rate:

        index = np.random.randint(len(individual))

        individual[index] = np.random.uniform(bounds[0], bounds[1])

    return individual

# Evolutionary Algorithm with Elitism

def evolutionary\_algorithm(dim, bounds, pop\_size=150, generations=1000, crossover\_rate=0.9, mutation\_rate=0.1, elite\_rate=0.1):

    population = initialize\_population(pop\_size, dim, bounds)

    best\_fitness\_history = []

    elite\_size = max(1, int(pop\_size \* elite\_rate))  # Number of elites to preserve

    for generation in range(generations):

        fitness = evaluate\_population(population)

        best\_fitness\_history.append(np.min(fitness))

        # Elitism: Preserve top individuals

        elite\_indices = np.argsort(fitness)[:elite\_size]

        elites = population[elite\_indices]

        # Selection

        parents = select\_parents(population, fitness)

        # Crossover and Mutation

        next\_generation = []

        for i in range(0, len(parents), 2):

            p1, p2 = parents[i], parents[min(i+1, len(parents)-1)]

            c1, c2 = blend\_crossover(p1, p2, bounds, crossover\_rate=crossover\_rate)

            next\_generation.append(mutate(c1, mutation\_rate, bounds))

            next\_generation.append(mutate(c2, mutation\_rate, bounds))

        # Combine elites and new generation

        next\_generation = np.array(next\_generation[:pop\_size - elite\_size])

        population = np.vstack((elites, next\_generation))

        # Print generation stats

        print(f"Generation {generation + 1}: Best Fitness = {np.min(fitness)}")

    # Return the best solution and fitness history

    best\_index = np.argmin(fitness)

    return population[best\_index], best\_fitness\_history

# Define the bounds for the fitness function

bounds = (-500, 500)

# Run the evolutionary algorithm

best\_solution, best\_fitness\_history = evolutionary\_algorithm(

    dim=20, bounds=bounds, pop\_size=200, generations=500, crossover\_rate=0.9, mutation\_rate=0.1

)

print(f"\nBest Solution: {best\_solution}")

print(f"Best Fitness: {fitness\_function(best\_solution)}")

# Plot fitness history

plt.figure(figsize=(10, 5))

plt.plot(best\_fitness\_history, label='Best Fitness')

plt.xlabel('Generation')

plt.ylabel('Fitness')

plt.title('Fitness Over Generations')

plt.grid(True)

plt.legend()

plt.show()

### Ant Colony Optimization (ACO)

import numpy as np

import matplotlib.pyplot as plt

# Define the second benchmark function (Rastrigin-like function with an additional term)

def benchmark\_function\_2(x, you=88):

    d = len(x)

    return 418.9829 \* d - sum([xi \* np.sin(np.sqrt(abs(xi))) for xi in x]) + you

# Ant Colony Optimization implementation

def ant\_colony\_optimization(dim, lower\_bound, upper\_bound, num\_ants, num\_iterations, alpha, beta, evaporation\_rate, Q, you):

    pheromone = np.ones((dim, 2))  # Pheromone levels for each variable (min and max bounds)

    best\_solution = None

    best\_fitness = float('inf')

    fitness\_history = []

    for \_ in range(num\_iterations):

        solutions = []

        solution\_fitness = []

        # Generate solutions for all ants

        for ant in range(num\_ants):

            solution = []

            for i in range(dim):

                # Calculate probabilities for selecting lower or upper bound

                probabilities = np.array([

                    (pheromone[i, 0] \*\* alpha) \* ((1 / (abs(lower\_bound) + 1e-10)) \*\* beta),

                    (pheromone[i, 1] \*\* alpha) \* ((1 / (abs(upper\_bound) + 1e-10)) \*\* beta)

                ])

                probabilities /= probabilities.sum()  # Normalize probabilities

                # Choose between lower or upper bound region

                if np.random.rand() < probabilities[0]:

                    solution.append(np.random.uniform(lower\_bound, 0))

                else:

                    solution.append(np.random.uniform(0, upper\_bound))

            solutions.append(solution)

            fitness = benchmark\_function\_2(solution, you)

            solution\_fitness.append(fitness)

            # Update best solution if current one is better

            if fitness < best\_fitness:

                best\_fitness = fitness

                best\_solution = solution

        # Update pheromone levels

        for i in range(dim):

            pheromone[i, 0] \*= (1 - evaporation\_rate)  # Evaporation for lower bound

            pheromone[i, 1] \*= (1 - evaporation\_rate)  # Evaporation for upper bound

            for ant in range(num\_ants):

                fitness = solution\_fitness[ant]

                contribution = Q / (fitness + 1e-10)  # Avoid division by zero

                if solutions[ant][i] < 0:

                    pheromone[i, 0] += contribution

                else:

                    pheromone[i, 1] += contribution

        fitness\_history.append(best\_fitness)

    return best\_solution, best\_fitness, fitness\_history

# Parameters

dim = 20

lower\_bound = -500

upper\_bound = 500

num\_ants = 50

num\_iterations = 500

alpha = 1  # Pheromone importance

beta = 5   # Heuristic importance

evaporation\_rate = 0.1

Q = 100  # Pheromone deposit factor

you = 88  # User-defined constant in the fitness function

# Run ACO

best\_solution, best\_fitness, fitness\_history = ant\_colony\_optimization(

    dim, lower\_bound, upper\_bound, num\_ants, num\_iterations, alpha, beta, evaporation\_rate, Q, you

)

# Print results

print("Best Solution:", best\_solution)

print("Best Fitness:", best\_fitness)

# Plot the fitness over iterations

plt.figure(figsize=(10, 6))

plt.plot(fitness\_history, label="Best Fitness")

plt.xlabel("Iteration")

plt.ylabel("Fitness")

plt.title("Fitness Over Iterations (ACO - Function 2)")

plt.legend()

plt.grid()

plt.show()

### Particle Swarm Optimization (PSO)

import numpy as np

import matplotlib.pyplot as plt

# Define the benchmark function (Function 2)

def benchmark\_function\_2(x, you=88):

    d = len(x)  # Dimension of the input vector

    return 418.9829 \* d - sum([xi \* np.sin(np.sqrt(abs(xi))) for xi in x]) + you

# Particle Swarm Optimization (PSO)

def particle\_swarm\_optimization(dim, lower\_bound, upper\_bound, num\_particles, num\_iterations, w, c1, c2, you):

    # Initialize particle positions and velocities

    positions = np.random.uniform(lower\_bound, upper\_bound, (num\_particles, dim))

    velocities = np.random.uniform(-abs(upper\_bound - lower\_bound), abs(upper\_bound - lower\_bound), (num\_particles, dim))

    personal\_best\_positions = positions.copy()

    personal\_best\_fitness = np.array([benchmark\_function\_2(pos, you) for pos in positions])

    global\_best\_position = personal\_best\_positions[np.argmin(personal\_best\_fitness)]

    global\_best\_fitness = min(personal\_best\_fitness)

    fitness\_history = []

    for \_ in range(num\_iterations):

        for i in range(num\_particles):

            # Evaluate fitness of current position

            fitness = benchmark\_function\_2(positions[i], you)

            # Update personal best

            if fitness < personal\_best\_fitness[i]:

                personal\_best\_fitness[i] = fitness

                personal\_best\_positions[i] = positions[i].copy()

            # Update global best

            if fitness < global\_best\_fitness:

                global\_best\_fitness = fitness

                global\_best\_position = positions[i].copy()

        # Update velocities and positions

        for i in range(num\_particles):

            r1 = np.random.rand(dim)  # Random coefficients for cognitive component

            r2 = np.random.rand(dim)

            velocities[i] = (

                w \* velocities[i]

                + c1 \* r1 \* (personal\_best\_positions[i] - positions[i])

                + c2 \* r2 \* (global\_best\_position - positions[i])

            )

            positions[i] = positions[i] + velocities[i]

            # Enforce bounds

            positions[i] = np.clip(positions[i], lower\_bound, upper\_bound)

        fitness\_history.append(global\_best\_fitness)

    return global\_best\_position, global\_best\_fitness, fitness\_history

# Parameters

dim = 20

lower\_bound = -500

upper\_bound = 500

num\_particles = 50

num\_iterations = 500

w = 0.7

c1 = 1.5

c2 = 1.5

you = 88

# Run PSO

best\_solution, best\_fitness, fitness\_history = particle\_swarm\_optimization(

    dim, lower\_bound, upper\_bound, num\_particles, num\_iterations, w, c1, c2, you

)

# Print results

print("Best Solution:", best\_solution)

print("Best Fitness:", best\_fitness)

# Plot the fitness over iterations

plt.figure(figsize=(10, 6))

plt.plot(fitness\_history, label="Fitness")

plt.xlabel("Iteration")

plt.ylabel("Best Fitness")

plt.title("Fitness Over Iterations (PSO - Function 2)")

plt.legend()

plt.grid()

plt.show()

## 6.3 Crossover code :

### Single-Point Crossover

def single\_point\_crossover(parent1, parent2):

    crossover\_point = np.random.randint(1, len(parent1))

    offspring1 = np.concatenate([parent1[:crossover\_point], parent2[crossover\_point:]])

    offspring2 = np.concatenate([parent2[:crossover\_point], parent1[crossover\_point:]])

    return offspring1, offspring2

### Two-Point Crossover

def two\_point\_crossover(parent1, parent2):

    crossover\_point1 = np.random.randint(1, len(parent1) - 1)

    crossover\_point2 = np.random.randint(crossover\_point1 + 1, len(parent1))

    offspring1 = np.concatenate([parent1[:crossover\_point1], parent2[crossover\_point1:crossover\_point2], parent1[crossover\_point2:]])

    offspring2 = np.concatenate([parent2[:crossover\_point1], parent1[crossover\_point1:crossover\_point2], parent2[crossover\_point2:]])

    return offspring1, offspring2

### Multi-Point Crossover

def multi\_point\_crossover(parent1, parent2, num\_points):

    points = np.sort(np.random.choice(range(1, len(parent1)), num\_points, replace=False))

    offspring1 = parent1.copy()

    offspring2 = parent2.copy()

    for i in range(num\_points):

        if i % 2 == 0:

            offspring1[points[i]:] = parent2[points[i]:]

            offspring2[points[i]:] = parent1[points[i]:]

    return offspring1, offspring2

### Uniform Crossover

def uniform\_crossover(parent1, parent2):

    mask = np.random.randint(0, 2, size=len(parent1))

    offspring1 = np.where(mask == 0, parent1, parent2)

    offspring2 = np.where(mask == 1, parent1, parent2)

    return offspring1, offspring2

### Half-Uniform Crossover

def half\_uniform\_crossover(parent1, parent2):

    mask = np.random.randint(0, 2, size=len(parent1))

    offspring1 = np.where(mask == 0, parent1, parent2)

    offspring2 = np.where(mask == 1, parent1, parent2)

    return offspring1, offspring2

### Shuffle Crossover

    combined = np.concatenate([parent1, parent2])

    np.random.shuffle(combined)

    split\_point = len(parent1)

    offspring1 = combined[:split\_point]

    offspring2 = combined[split\_point:]

    return offspring1, offspring2

### Matrix Crossover

    matrix\_size = int(np.sqrt(len(parent1)))

    parent1\_matrix = parent1.reshape(matrix\_size, matrix\_size)

    parent2\_matrix = parent2.reshape(matrix\_size, matrix\_size)

    crossover\_point = np.random.randint(0, matrix\_size)

    offspring1\_matrix = np.copy(parent1\_matrix)

    offspring2\_matrix = np.copy(parent2\_matrix)

    offspring1\_matrix[crossover\_point:] = parent2\_matrix[crossover\_point:]

    offspring2\_matrix[crossover\_point:] = parent1\_matrix[crossover\_point:]

    offspring1 = offspring1\_matrix.flatten()

    offspring2 = offspring2\_matrix.flatten()

    return offspring1, offspring2

### Three-Parent Crossover

def three\_parent\_crossover(parent1, parent2, parent3):

    crossover\_point = np.random.randint(1, len(parent1))

    offspring1 = np.concatenate([parent1[:crossover\_point], parent2[crossover\_point:], parent3[crossover\_point:]])

    offspring2 = np.concatenate([parent2[:crossover\_point], parent3[crossover\_point:], parent1[crossover\_point:]])

    return offspring1, offspring2