Laboratory 3

Variant 4

Class group 105

Group 24

By Vasileios Ntalas and Dimitrios Gkolias

# Genetic Algorithm for Booth Function Optimization

## Introduction

The purpose of this report is to detail the design and implementation of a genetic algorithm (GA) to optimize (minimize) the Booth function, defined by:

$$Booth(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

with the global minimum located at x=1 , y=3 where the function value is 0. In this project, and are constrained to the range [-5, 5].

The GA employs **Roulette Wheel Selection**, **Gaussian Mutation**, and a **Random Interpolation Crossover** operator. The Booth function is a classic test function for numerical optimization, with a global minimum value of 0 at . By using the evolutionary process, the GA iteratively refines candidate solutions to find the coordinates that yield the minimum Booth function value.

## Methodology

A genetic algorithm is a stochastic optimization method inspired by natural selection. It evolves a population of candidate solutions (individuals), each with an associated fitness value, through the operations of **selection**, **crossover**, and **mutation**. The key steps are:

1. **Initialization**: Generate an initial population of size within the search space.
2. **Selection**: Choose individuals based on fitness for reproduction.
3. **Crossover**: Recombine pairs of individuals to produce new offspring.
4. **Mutation**: Introduce random perturbations in offspring to maintain diversity.
5. **Replacement**: Form the next generation population from offspring.

6. **Termination**: Repeat until a stopping criterion (e.g., a fixed number of generations) is met.

Each individual in the population is a vector [x,y]. We randomly initialize each dimension in [-5, 5], ensuring a diverse range of starting points.

## Genetic Algorithm Approach

### Population Initialization

Generate a set of random solutions (x,y) within [−5,5] × [−5,5]. Each individual can be viewed as a “gene” containing two floating-point values for xxx and y.

### Evaluation (Fitness Calculation)

We interpret lower Booth function values as “better.” However, GAs usually maximize fitness, so we define:

$$fitness(x,y) = \frac{1}{1+f(x,y)} \; ensuring\ lower\ f \rightarrow higher\ fitness$$

### Selection (Roulette Wheel)

- Compute each individual’s fitness relative to the total fitness in the population.
- Pick individuals stochastically, where higher-fitness solutions are more likely (though not guaranteed) to be chosen.

### Crossover (Random Interpolation)

- Pairs of parents produce two children using an **interpolation factor $\boldsymbol{\alpha \in [0,1]}$**
- $Child_1 = \alpha p_1 + (1+\alpha) p_2$, $Child_2 = \alpha p_2 + (1+\alpha) p_1$ ,
- Only a fraction (crossover_rate) of parent pairs undergo crossover, promoting both diversity and retention of existing solutions.

### Mutation (Gaussian Noise)

- With probability mutation_rate, add a small random offset Δ
- Restrict (clip) mutated values back into [−5,5] so individuals remain valid.

### Iterate

- Replacement strategy: The new population fully **replaces** the old one.
- Continue for `num_generations`. Track and log the best individual (highest fitness) each generation to monitor convergence.

## Implementation Highlights

Below is a summary of the key parts in the **Python code**. Only the *essential logic* is explained here for clarity.

**Class GeneticAlgorithm**
The class encapsulates all GA-related steps:

```
    ga = GeneticAlgorithm(
        population_size=100,
        mutation_rate=0.3,
        mutation_strength=0.6,
        crossover_rate=0.5,
        num_generations=100
    )
```

These parameters determine population size, mutation probability, mutation magnitude, fraction of crossover pairs, and total generations to run.

**Fitness Calculation**

```
def evaluate_population(self, population: np.ndarray) -> np.ndarray:
        """
        Compute the 'fitness' of each individual.
        Since we want to MINIMIZE f(x,y),
        define fitness = 1.0 / (1.0 + f(x,y)) so that
        lower cost => higher fitness.
        """
        fitness_values = []
        for (x, y) in population:
            cost = self.fn(x, y)
            fitness = 1.0 / (1.0 + cost)
            fitness_values.append(fitness)
        return np.array(fitness_values)
```

Maps cost to fitness so the GA can treat low-cost solutions as “higher fitness.”

**Selection (Roulette Wheel)**

```
def selection(self, population: np.ndarray, fitnesses: np.ndarray) -> np.ndarray:
        """
        Roulette Wheel Selection:
          - Convert fitness array to probabilities.
          - Use random draws to select 'parents' for reproduction.
        """
        new_population = []
        total_fitness = np.sum(fitnesses)
        if total_fitness == 0:
```

```python
            # If total_fitness is zero (edge case), pick uniformly.
            probabilities = np.ones_like(fitnesses) / len(fitnesses)
        else:
            probabilities = fitnesses / total_fitness

        for _ in range(self.population_size):
            r = random.random()
            cumulative = 0.0
            for i, p in enumerate(probabilities):
                cumulative += p
                if cumulative >= r:
                    new_population.append(population[i])
                    break

        return np.array(new_population)
```

Accumulates probabilities so that fitter individuals have a larger “slice of the wheel.”

**Crossover (Random Interpolation)**

```python
def crossover(self, parents: np.ndarray) -> np.ndarray:
        """
        Random Interpolation Crossover:
          - For pairs of parents, produce 2 offspring via interpolation.
          - We only do crossover for a subset of the population
            determined by crossover_rate (fraction of pairs).
        """
        offspring = parents.copy()

        # Number of pairs to actually perform crossover on:
        num_pairs = int(self.crossover_rate * (len(parents) // 2))
        indices = np.arange(len(parents))
        np.random.shuffle(indices)

        for i in range(num_pairs):
            idx1 = indices[2*i]
            idx2 = indices[2*i + 1]
            p1 = parents[idx1]
            p2 = parents[idx2]

            alpha = random.random()  # interpolation factor in [0,1]

            # Child 1
            c1 = alpha * p1 + (1.0 - alpha) * p2
            # Child 2
            c2 = alpha * p2 + (1.0 - alpha) * p1
            offspring[idx1] = c1
            offspring[idx2] = c2

        return offspring
```
We only apply crossover to some fraction of parent pairs, ensuring a balance of exploration and exploitation.

**Mutation**

```
def mutate(self, population: np.ndarray) -> np.ndarray:
        """
        Gaussian Mutation:
          - With probability mutation_rate, add N(0, mutation_strength) to x and
y.
          - Clip values to [x_bounds, y_bounds].
        """
        for i in range(len(population)):
            if random.random() < self.mutation_rate:
                dx = random.gauss(0, self.mutation_strength)
                dy = random.gauss(0, self.mutation_strength)
                new_x = population[i][0] + dx
                new_y = population[i][1] + dy
                # Clip to valid domain
                new_x = max(self.x_bounds[0], min(self.x_bounds[1], new_x))
                new_y = max(self.y_bounds[0], min(self.y_bounds[1], new_y))
                population[i] = (new_x, new_y)
        return population
```
This ensures continuous variation in the population. Clipping enforces the feasible domain.

## Experiments

### Generic Parameter Sets

We tried 4 different genetic algorithm parameters combinations, as shown in the table. The population size and the number of gens was kept relatively high as we want to focus on the effect of the other parameters mostly and how their combination affects the result.

| Case | Pop. Size | Mut. Rate | Mut. Str. | Cross Rate | Gens |
|---|---|---|---|---|---|
| **1** | 100 | 0.1 | 0.5 | 0.5 | 100 |
| **2** | 200 | 0.3 | 1.0 | 0.6 | 100 |
| **3** | 300 | 0.7 | 2.0 | 0.7 | 100 |
| **4** | 200 | 1.0 | 3.0 | 0.8 | 100 |

| Case | Best(x,y) | Best Fitness |
|---|---|---|
| **1** | 1.00025689<br>2.99862599 | 11.877398130350516 |
| **2** | 0.99860593<br>3.00071079 | 12.35315688400215 |
| **3** | 0.99496749<br>3.00224813 | 9.698301462984935 |
| **4** | 1.09406362<br>2.85972649 | 3.3314497790250925 |

We can see that the second case is the most ideal one since it has the highest fitness and also being closest to the minimum.

## Randomness in generic algorithm

So we implement more experiments using that same set of parameters as before

| Seed | Best(x,y) | Best Fitness |
|---|---|---|
| **1** | 1.00215319<br>2.99743499 | 11.3394956956333 |
| **2** | 1.00037679<br>2.99952707 | 14.725305750195218 |
| **3** | 1.00000821<br>3.00005038 | 17.929708509118644 |
| **4** | 0.99766613<br>3.00361348 | 10.594511874516963 |

$$Mean(Average\ fitness) = \frac{11.3395 + 14.7253 + 17.9297 + 10.5945}{4} = 13.6478$$

With a standard deviation of <u>3.1424</u>

We can clearly see that seed 3 is the closest to the solution.

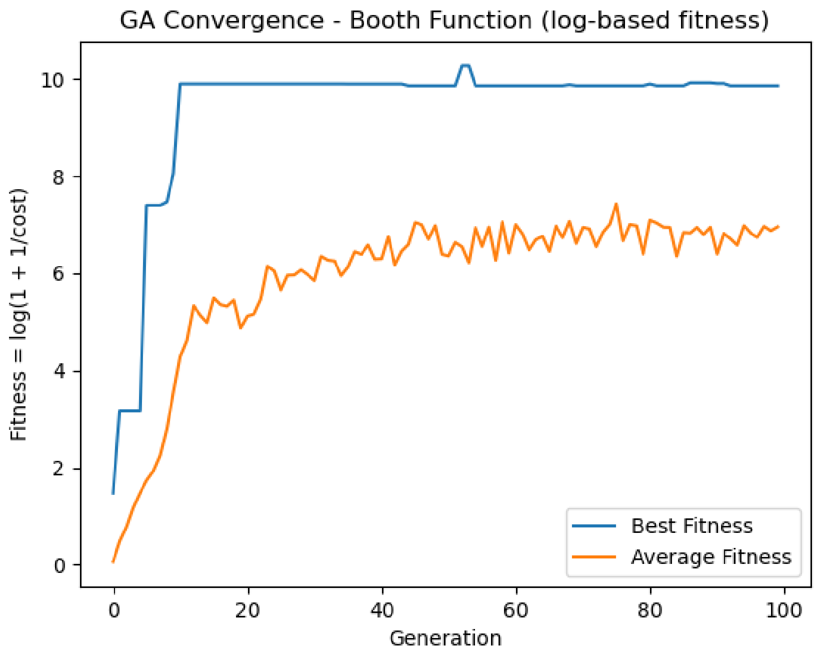
Now we will gradually decrease the population and see how the algorithm reacts. What we expect to happen is for the algorithm to worsen, as there is less chance for the population to find the minimum.

| Decreased Population of 2nd | Best(x,y) | Best Fitness |
|---|---|---|
| **50%** | 1.00019537<br>2.99907608 | 12.711939710892931 |
| **25%** | 1.0033375<br>2.99524651 | 10.083741592016173 |
| **10%** | 1.00180565<br>2.99858733 | 7.999397965133113 |

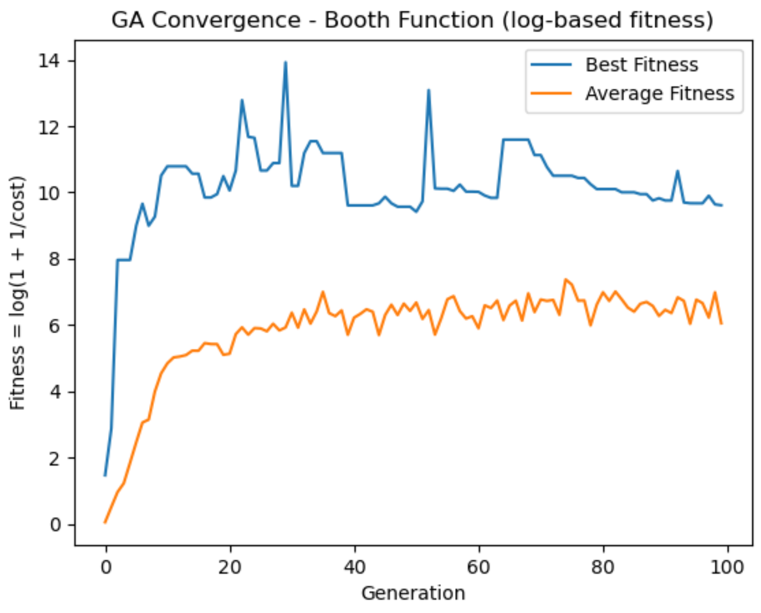
| Crossover Rate | Avg Best(x,y) | Avg Best Fitness |
|---|---|---|
| **0.1** | 1.00395249<br>2.99464946 | 9.86300095111409 |
| **0.5** | 1.003733<br>2.99989859 | 9.61538437409576 |
| **0.7** | 0.99931048<br>3.00048763 | 13.947621830669814 |
| **0.9** | 0.99876988<br>2.99991363 | 11.680970135925346 |

To rule out random effects we went through each crossover rate with multiple values of seeds.

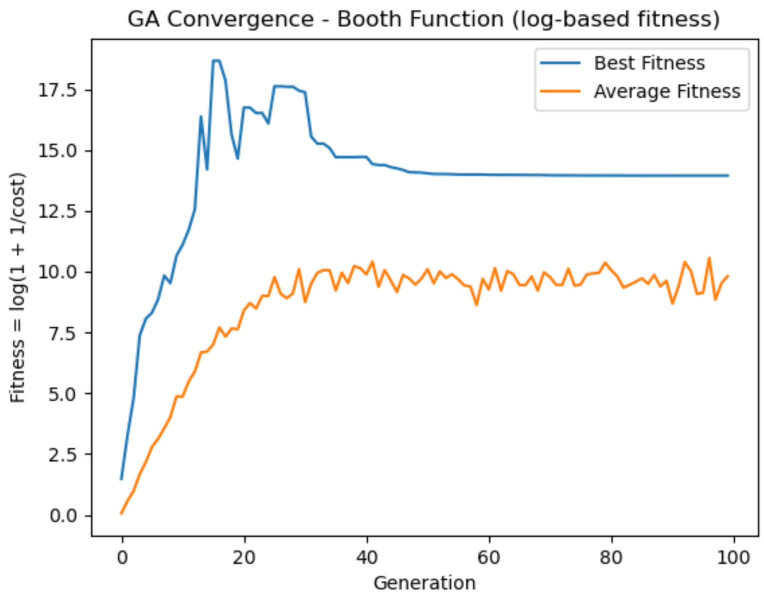
***Crossover Rate=0.1***



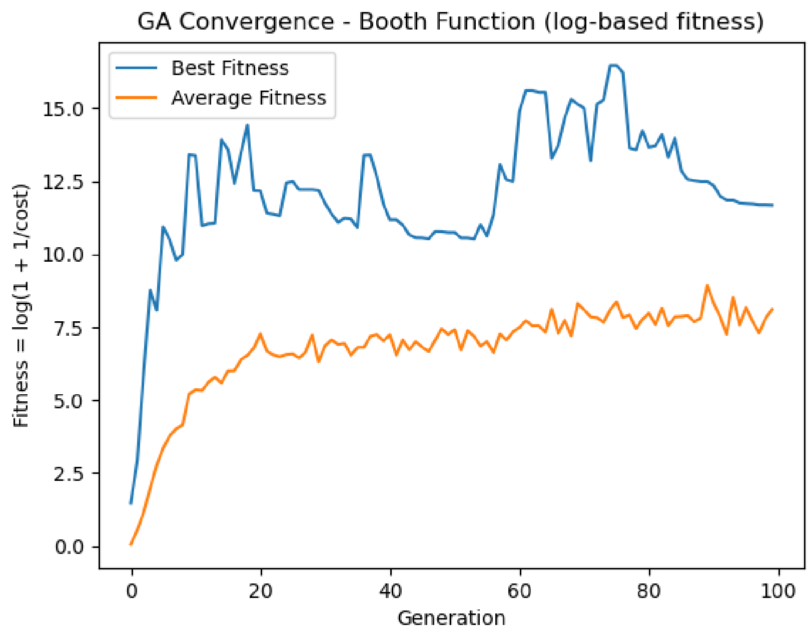
***Crossover Rate=0.5***



***Crossover Rate=0.7***



***Crossover Rate= 0.9***



**Crossover rate 0.7** gave the best results. It balanced creating new solutions and keeping the good ones, helping the algorithm find the best answer.

**Crossover rate 0.1** was more stable but improved slowly and didn't reach the best solutions.

**Crossover rate 0.9** changed the population too much, which made it harder for the algorithm to keep good solutions and slowed down progress.

## Mutation and the Convergence

Now we're going to analyze how mutation settings affect the convergence of the genetic algorithm. Again we are running across multiple seeds.

| Case | Mutation Rate | Mutation Strength | Avg Best(x,y) | Avg Best Fitness |
|---|---|---|---|---|
| **1** | 0.1 | 0.5 | 1.00070575 2.99984336 | 13.268120780005292 |

| | | | | |
|---|---|---|---|---|
| **2** | 0.3 | 1.0 | 0.99876988 2.99991363 | 11.680970135925346 |
| **3** | 0.7 | 1.5 | 1.00002881 2.99994226 | 18.706936565690008 |
| **4** | 1.0 | 2.0 | 0.31480994 3.53154997 | 0.7799678054229939 |

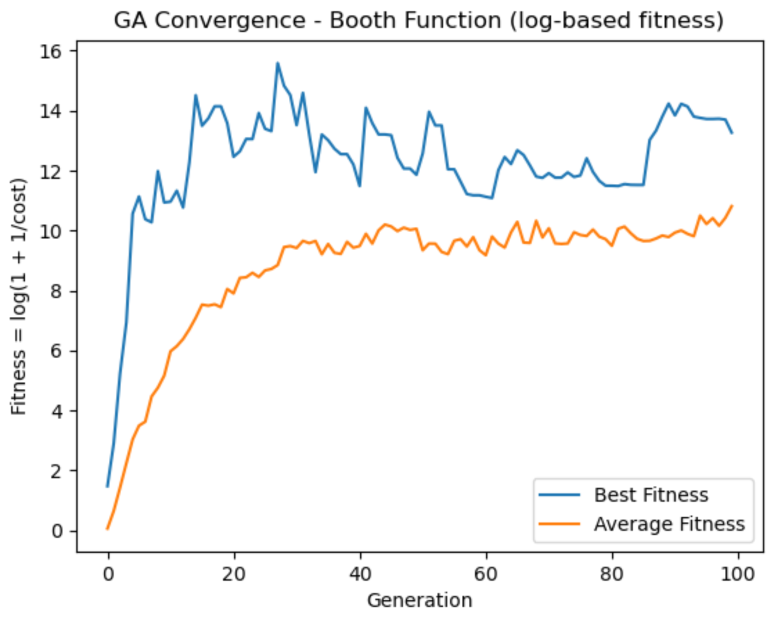
**Moderate to moderately high mutation (0.3–0.7)** offered the best balance between exploration and convergence.

**Too little mutation** can lead to slow or stuck convergence, while

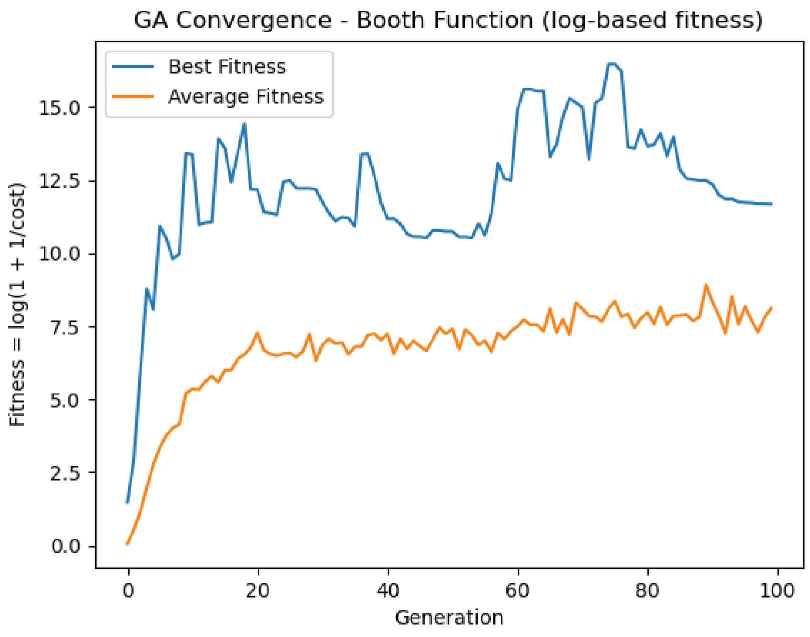
**Too much mutation** disrupts progress and leads to random search behavior.

The best results were achieved in **Case 3**, showing that carefully tuned higher mutation levels can significantly improve GA performance — as long as they don't overpower selection and crossover.

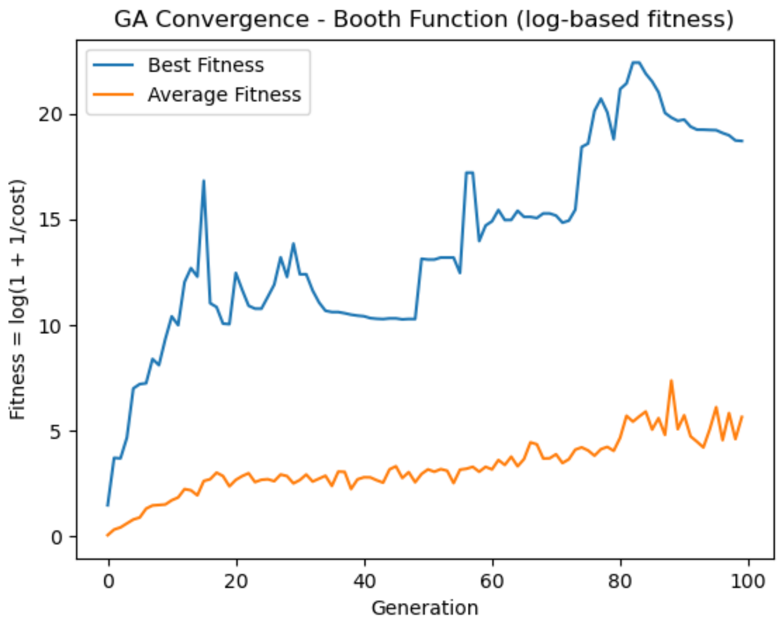
***Case 1***



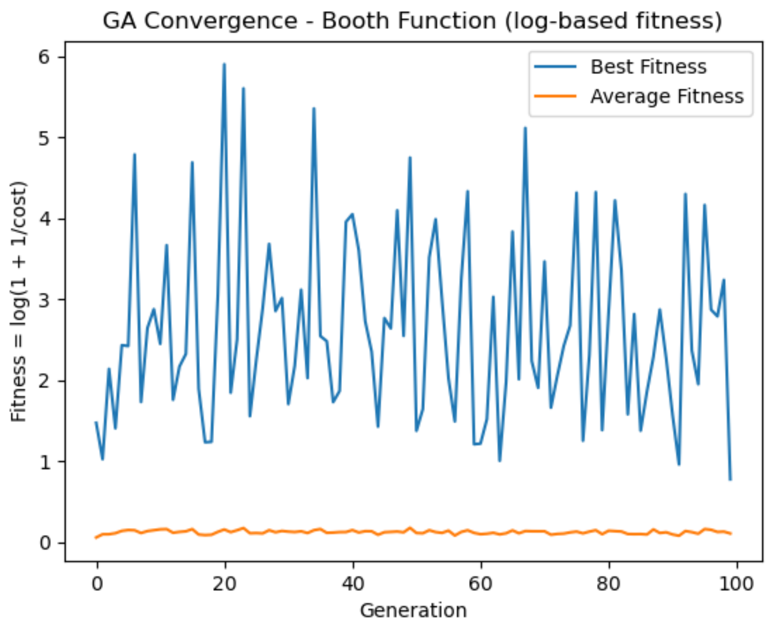
***Case 2***


GA Convergence - Booth Function (log-based fitness)
Best Fitness
Average Fitness
Fitness = log(1 + 1/cost)
Generation
0.0
2.5
5.0
7.5
10.0
12.5
15.0
0
20
40
60
80
100

***Case 3***


GA Convergence - Booth Function (log-based fitness)
Best Fitness
Average Fitness
Fitness = log(1 + 1/cost)
Generation
0
5
10
15
20
0
20
40
60
80
100

***Case 4***



## Discussion

### ***Effect of Parameters***

- **Population Size**: Larger populations foster diversity but require more evaluations per generation.
- **Mutation Rate & Strength**:

  Too small → population may converge prematurely.

  Too large → solutions become chaotic, losing good solutions.

- **Crossover Rate**:

  High crossover → more mixing of genetic material, leading to faster exploration.

  Low crossover → slower exploration, but less risk of overwriting promising genes.

### ***Convergence***

- The Booth function has a single global minimum, so the GA generally finds (1,3) or a very close approximation.
- In repeated runs, small deviations from the exact global minimum can occur due to random processes.

### ***Advantages***

- No need for gradients (works for black-box or noisy functions).
- The interplay of selection, crossover, and mutation often avoids local minima.

***Disadvantages***

- GAs can be slower than direct gradient-based methods for simpler functions.
- Requires parameter tuning (mutation/crossover rate, population size, etc.).

## Conclusion

By applying **roulette wheel selection**, **random interpolation crossover**, and **Gaussian mutation** to the **Booth function**, we have demonstrated that a GA effectively converges toward the global minimum at (1,3). The code is written so that each operation—population initialization, selection, crossover, mutation—is clear and modular.

Key takeaways:

- **Genetic Algorithms** are flexible, gradient-free optimizers suitable for a range of problems.
- **Parameter Tuning** is important for robust performance and avoidance of premature convergence.
- **Plotting best and average fitness** reveals how quickly the population evolves and how effectively it zeroes in on the global optimum.

This project highlights both the theoretical concepts (fitness proportionate selection, blending crossover, Gaussian-based mutation) and the practical code implementation to confirm their effectiveness on a well-known function.