Laboratory 3

Variant 5

Group 5

By Jan Szachno and Aleksandra Głogowska

---

1. Introduction

The task is to write a Python program to solve a given 2D function using a basic genetic algorithm, with initialization of x and y in the range [-5,5]. In our case, the given function is the Stybliński-Tang function, and it is depicted in Figure 1.1. The genetic algorithm that we have to use is the Rank Selection algorithm. For mutation of the next generations, we have to use the Gaussian operator, and for the crossover, random interpolation. We have been given a Python template with a set of parameters that have to be used in the algorithm. The parameters are population size, mutation rate, mutation strength, crossover rate, and number of generations.

$$f(x,y) = \frac{1}{2}\left(\frac{x^4}{2} - 16x^2 + 5x + \frac{y^4}{2} - 16y^2 + 5y\right)$$

Figure 1.1. - Stybliński-Tang function.

2. Algorithm Description

The genetic algorithm begins by creating a population of potential solutions to a problem. Each solution is evaluated for its fitness. Through selection, individuals with higher fitness are chosen to mix their ‘’traits’’ with other solutions and pass them to the next generations. Crossover combines traits from selected individuals to create new solutions. Mutation introduces random changes to maintain genetic diversity. This iterative process continues for a specified number of generations.

The goal of the algorithm is to find a value of the Stybliński-Tang function that will indicate the minimum of the function. We are using a genetic algorithm, which is called the Rank Selection. Other techniques used are the Gaussian operator and the crossover interpolation.

The program starts with the selection of the parameters. Population size is a number that indicates how many points (x, y) we are primarily randomly choosing. The mutation rate is the parameter that determines the probability of a mutation occurring for each individual in the population during each generation. Later in the algorithm, we choose a random number from 0 to 1, and if the mutation rate is higher than the random number, the mutation strength number is applied as a mutation to a particular point. The mutation strength is also inputted as a parameter. In genetic algorithms, the crossover rate (also known as the crossover probability) is a parameter that determines the frequency with which crossover operations are performed during the evolution of the population. Crossover is a genetic operator that combines the genetic information of two parent individuals to generate new offspring. It's a crucial mechanism for introducing variation into the population, which, in turn, aids in the exploration of the solution space. The crossover rate is typically expressed as a fraction or a percentage. For example, a crossover rate of 0.7 or 70% means that, on average, 70% of the selected pairs of individuals will undergo crossover to produce offspring, while the remaining 30% will be copied into the next generation unchanged. The number of generations indicates how many crossovers there will be. The program's randomness is based on the seed parameter.

After the selection of appropriate values for the parameters, first, we initialize the population, which in other words, is picking the number of individual solutions of points (x, y). The coordinates of each candidate solution are randomly sampled on the 2D plane in the range [-5,5] using the uniform distribution.

Now, we are calling a function that will generate the “best solutions,” “best fitness values,” and “average fitness values.” The “best solutions” are the (x, y) candidate solutions within each generation that achieve the lowest fitness values. The “best fitness values” are the calculated values of the function Styblíński-Tang based on the “best solutions.” The “average fitness values” are the average values of our given function of every point in each generation. The algorithm is going through each generation. It takes every individual point and evaluates the fitness of the population by substituting (x, y) values in the given function. Then, it ranks each individual based on their fitness. The individuals with the lowest values (the ones that could be the closest to the minimum of the function) get the best rating - the highest assigned probability. After that, the algorithm selects the same number of points as the population number, but it allows for duplicates of the individuals in the selection. The probabilities of each candidate value, assigned before, impact the selection of individuals. The new population gets selected this way, and it overwrites the old population.

Now, the algorithm takes the new population, and it implements the crossover mechanism with random interpolation over the parent to create the offspring. In our case, the algorithm makes parent pairs randomly, based on the random number generator and the seed, preset as a parameter. For every already shuffled parent-pair, it generates a number in the range [0,1). If the newly generated number is lower than the crossover rate parameter, random interpolation is applied to both selected parents. The random interpolation formulas are depicted in Figure 2.1. Where o refers to offspring, $p1, p2$ to parents; and α is a random number. If the crossover rate is lower than the generated number, the parents are passed as offspring.

$$x_o = \alpha \cdot x_{p1} + (1 - \alpha) \cdot x_{p2},$$
$$y_o = \alpha \cdot y_{p1} + (1 - \alpha) \cdot y_{p2}$$

Figure 2.1. - Random Interpolation formulas.

Next, the algorithm mutates the current population. It takes individuals one by one and generates a random number in the range [0,1). If the number is lower than the mutation rate parameter, it adds a “noise” to the individual’s value. The “noise” is randomized based on Gaussian distribution. The whole described process is repeated until the number of generations requirement is checked.

3. Experiments and Conclusions

For the report, we ran our code, and we documented the results of tasks 1-4, which consisted of doing different experiments on the code. The “fitness” result that has been presented in every table for the sake of the experiments has been calculated in the following way. We checked the minimal value of the Styblíński-Tang function, and we found out that it is equal to -78.3323314075428 for x and y equal to -2.903534. To be able to visualize the changes using the logarithmic scale, and to draw conclusions in the best way possible, we took the absolute values of the known global minimum subtracted from the values of our solutions. Knowing that the lowest “fitness” values indicated the most correct solutions, we described the conclusions.

TASK 1

Finding genetic algorithm parameters. Run the algorithm with different parameters (population size, mutation rate and strength, crossover rate, and number of generations) until you find a set of parameters that obtains a good fitness value. Present the tested combinations in a table alongside the solutions obtained by the algorithm.

We ran the algorithm with different parameters, trying to find the best combination. All of the experiments had seed 42. The parameters tested are presented in Table 3.1.

Population size	Mutation rate	Mutation strength	Crossover rate	Number of generations	Fitness
100	0.5	0.5	0.5	100	6.64E-03
100	0.5	0.5	0.5	1000	8.85E-03
1000	0.5	0.5	0.5	100	2.82E-03
100	0.1	0.5	0.5	100	5.68E-14
100	0.5	0.1	0.5	100	3.54E-04
100	0.5	0.5	0.1	100	3.86E-03
1000	0.1	0.1	0.1	100	5.68E-14
1000	0.1	0.1	0.1	1000	5.68E-14
1000	0.1	0.1	0.1	200	5.68E-14

Table 3.1. - Table with results for task 1.

From Table 3.1, we can draw some initial observations. From our first experiment we can see that the mutation rate is more significant than mutation strength, and changing its value has more impact on the final result. We observed a similar relation with population size and number of generations. From our experiment, we can see that increasing the population size has a better impact on the final results than increasing the number of generations.

TASK 2

Randomness in genetic algorithm. Run the algorithm with the best parameters found in point 1. using 5 different random seeds. Report the best solution across all seeds and its fitness value, and the average fitness value and its standard deviation across all seeds. Rerun the algorithm with decreasing population size (eg. use around 50%, 25%, and 10% of the original population), and present the results in a table.

For this task, we tested the impact of different seeds on the algorithm. For each seed value, we adjusted the population parameter and checked the results. The experiments for this task can be seen in Table 3.2. For every experiment, mutation rate = 0.1, mutation strength = 0.1, crossover rate = 0.1, and number of generations = 200.

Seed	**Population**	**Fitness**
100	1000	5.68E-14
	500	2.93E-12
	250	2.40E-07
	100	2.70E-05
512	1000	5.68E-14
	500	7.02E-07
	250	1.46E-09
	100	1.11E-04
8329	1000	3.90E-07
	500	-5.68E-14
	250	3.42E-06
	100	1.18E-04
123456789	1000	5.68E-14
	500	2.01E-08
	250	7.93E-07
	100	5.56E-07
112233	1000	5.68E-14
	500	3.55E-13
	250	1.72E-07

	100	1.18E-05

Table 3.2. - Results of experiments for task 2.

This experiment has shown us that parameters selected for a certain seed do not always produce great results in all scenarios. Those observations were easier to observe for smaller population sizes since the initial population was less diverse. For example, for the smallest population of 100, the difference between the worst and best case was three orders of magnitude.

TASK 3

Crossover impact. Run the algorithm a few times, changing the crossover rate. Plot the best and average fitness across the generations. Report the results averaged across more than one seed.

For this task, we have only been changing the crossover rate parameter for every experiment. Other parameters are seed = 42, population size = 1000, mutation rate = 0.1, mutation strength = 0.1, and number of generations = 200. The results of every experiment are shown in Table 3.3.

Crossover rate	**Fitness**
0.1	5.68E-14
0.25	5.68E-14
0.5	5.68E-14
0.6	5.68E-14
0.7	5.68E-14
0.75	1.73E-08
0.725	2.84E-14
0.775	4.26E-14
0.8	4.26E-14
0.85	8.92E-08
0.9	1.22E-08
1.0	1.36E-07

Table 3.3. - Results of the experiments for task 3.

For the reason of observation of the impact of the crossover rate on the algorithm, we plotted four graphs with different crossover rates, with the same parameters as the experiments from Table 3.3. The plots are depicted in Figures 3.4-3.7.

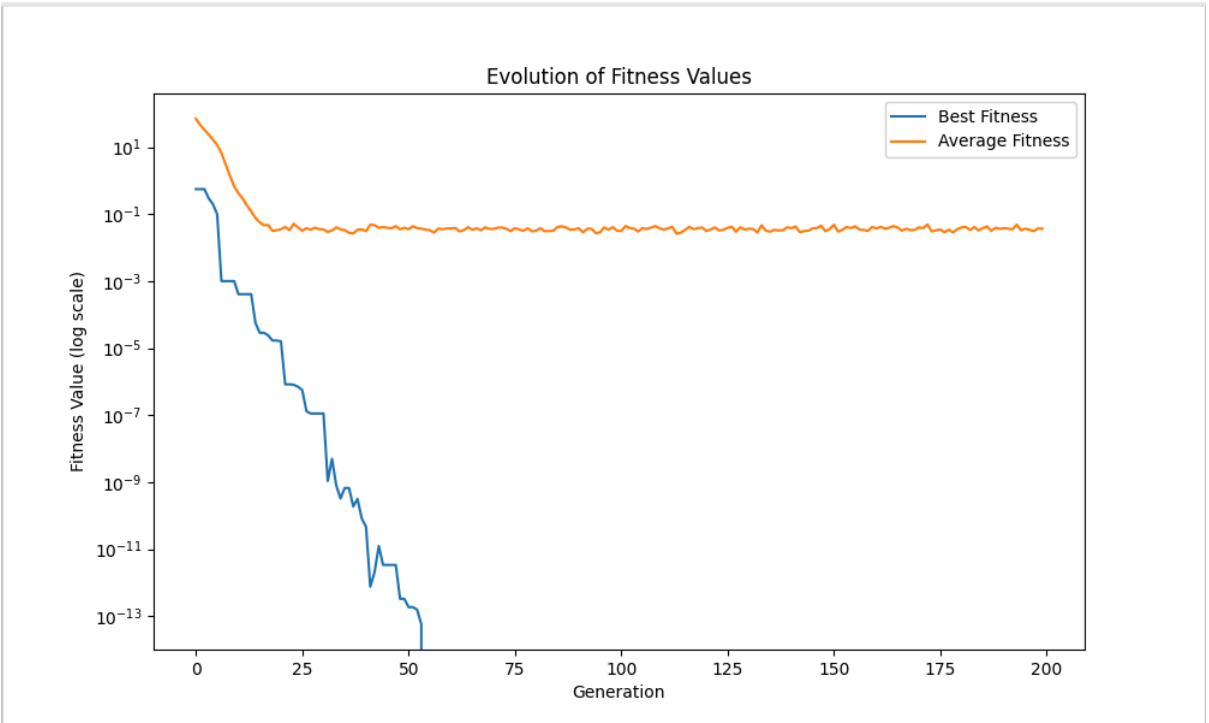

Figure 3.4. - Plot with the best and average fitness across the generations, with crossover rate equal to 0.1.

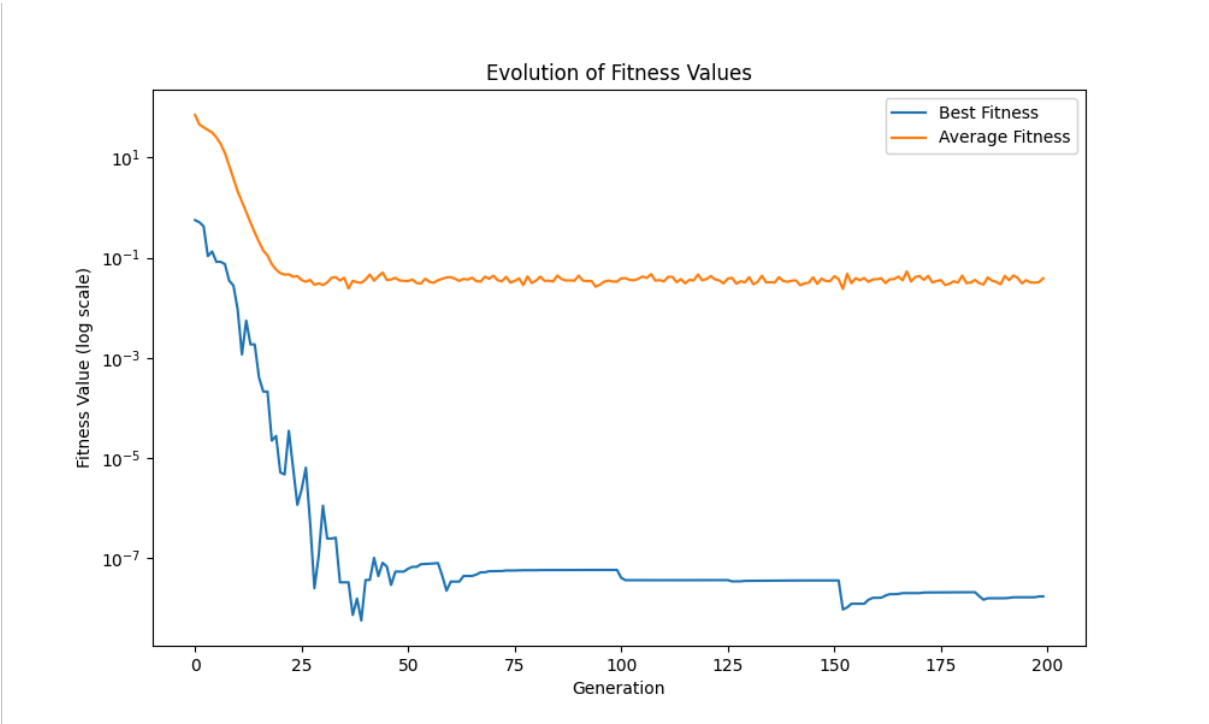

Figure 3.5. - Plot with the best and average fitness across the generations, with crossover rate equal to 0.75.

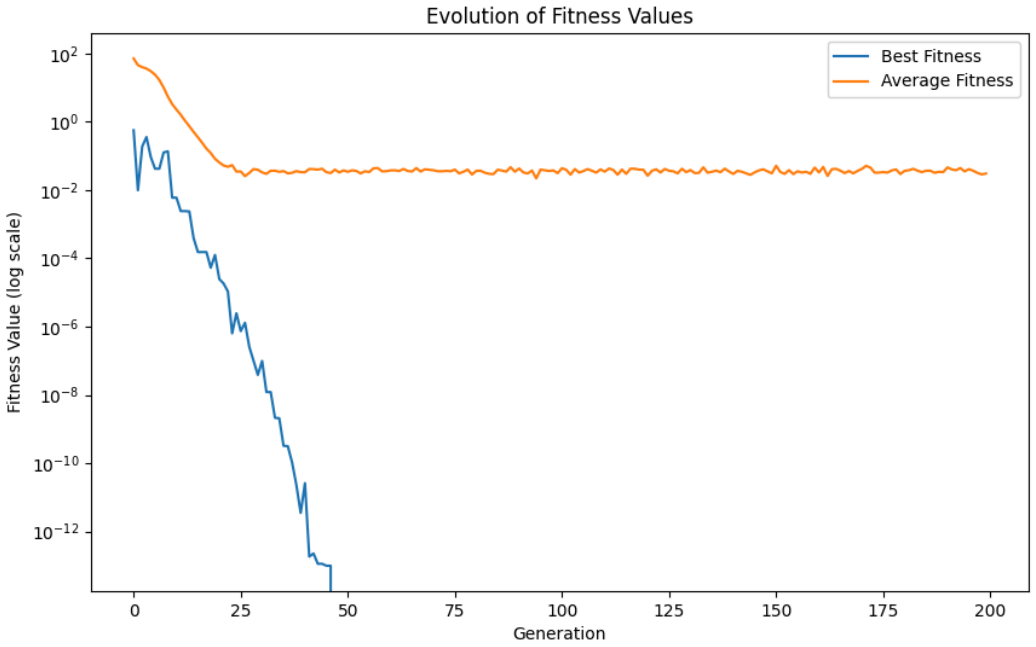

Figure 3.6. - Plot with the best and average fitness across the generations, with crossover rate equal to 0.775.

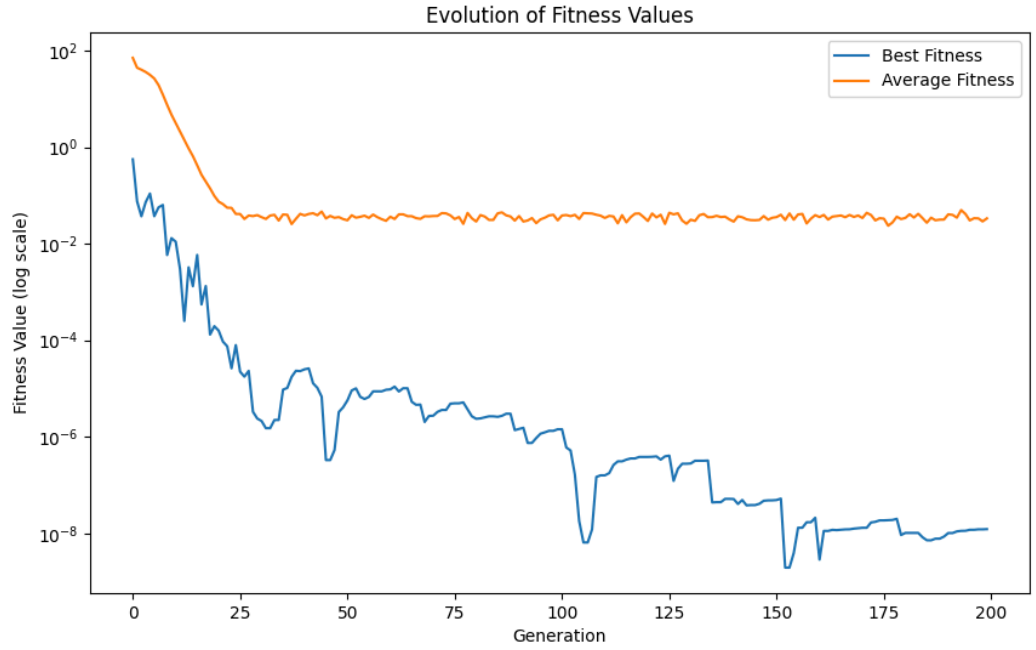

Figure 3.7. - Plot with the best and average fitness across the generations, with crossover rate equal to 0.9.

The selection of an appropriate crossover rate is important because it affects the balance between exploration (searching new areas of the solution space) and exploitation (refining the current known solutions). A high crossover rate encourages exploration by generating a diverse set of solutions, while a lower crossover rate may lead to faster convergence by

focusing on the exploitation of existing solutions. Finding the right balance is crucial for the effectiveness of the genetic algorithm.

TASK 4

Mutation and the convergence. Run the algorithm increasing the mutation rate and mutation strength. Plot the best and average fitness across the generations. Report the results averaged across more than one seed.

For the experiments, we set parameters such as the seed = 42, population size = 1000, crossover rate = 0.1, and number of generations = 200. The first part of the experiment consisted of changing the mutation rate, and the second was mutation strength. The results of these experiments are in Table 3.8. and 3.9.

Mutation rate	**Mutation strength**	**Fitness**
0.1	0.1	5.68E-14
0.25	0.1	5.68E-14
0.3	0.1	4.45E-11
0.35	0.1	2.84E-14
0.4	0.1	1.73E-10
0.45	0.1	9.18E-07
0.5	0.1	5.24E-05
0.6	0.1	4.51E-05
0.7	0.1	1.51E-03
0.8	0.1	1.51E-04
0.9	0.1	9.19E-04
1	0.1	7.74E-04

Table 3.8. - Results of the experiments for task 4, with changing mutation rates.

For the reason of observation of the impact of the mutation rate on the algorithm, we plotted four graphs with different mutation rates, with the same parameters as the experiments from Table 3.8. The plots are depicted in Figures 3.10-3.13. After that, we plotted graphs with changing mutation strengths, with the same parameters as the experiments from Table 3.9. The plots are pictured in Figures 3.14-3.17.

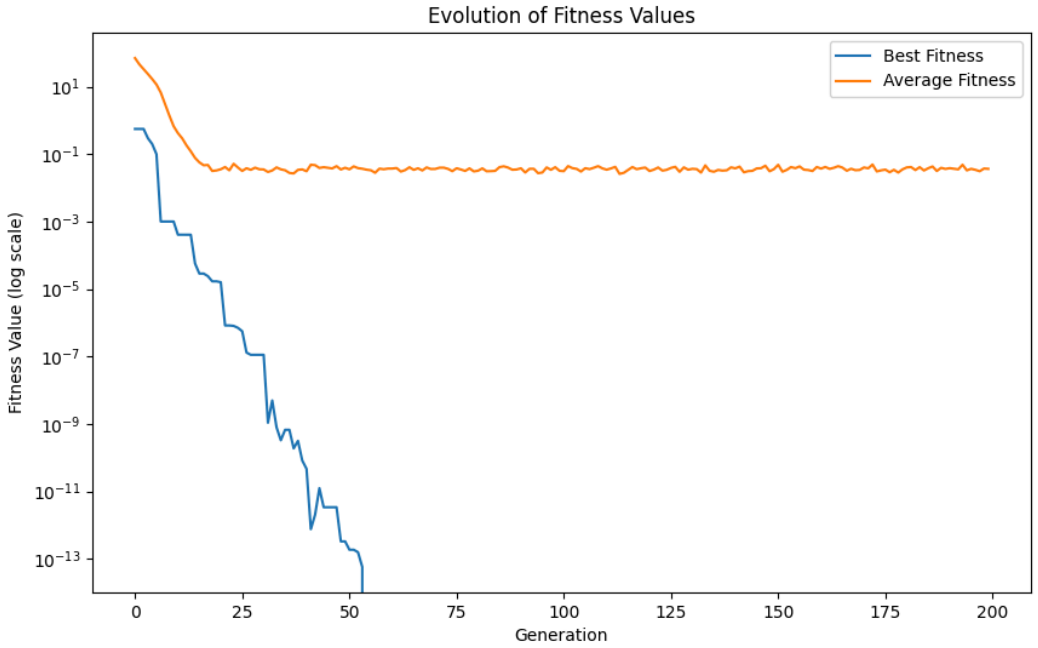

Figure 3.10. - Plot with the best and average fitness across the generations, with mutation rate equal to 0.1.

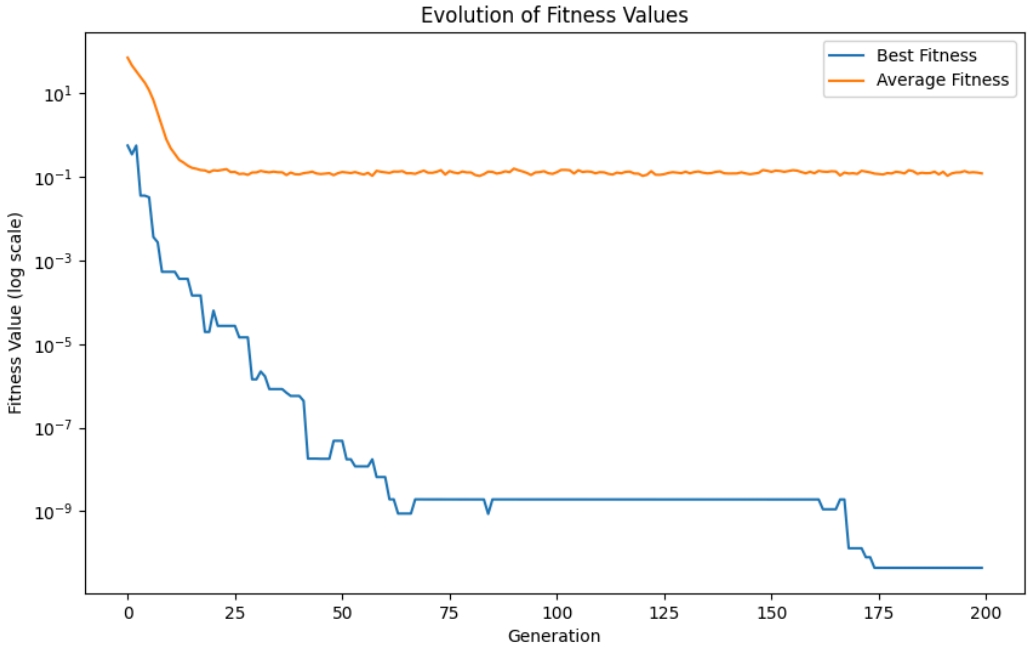

Figure 3.11. - Plot with the best and average fitness across the generations, with mutation rate equal to 0.3.

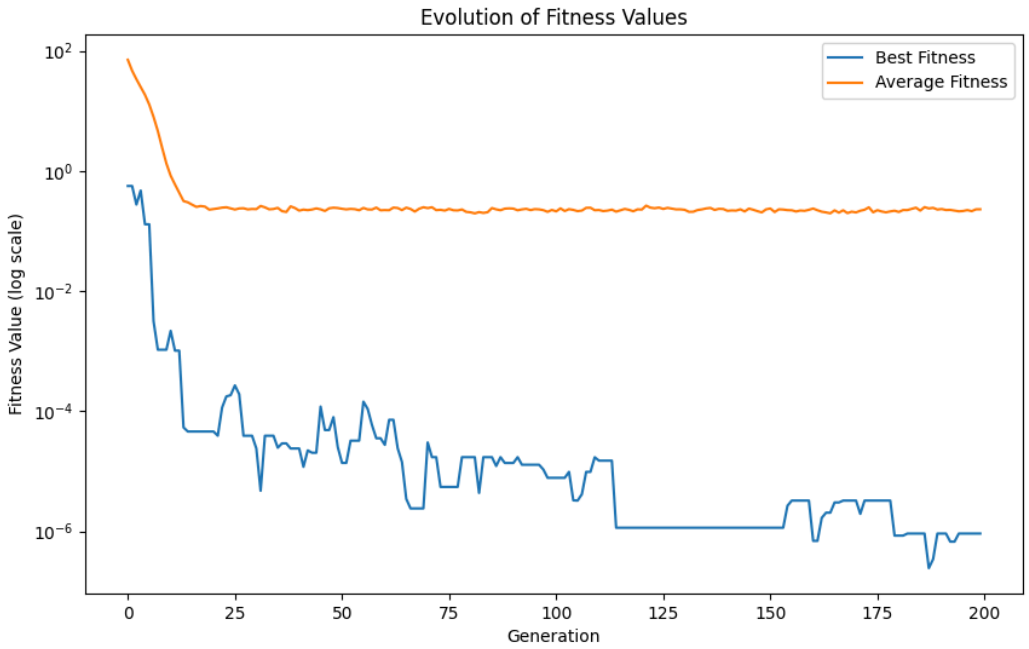

Figure 3.12. - Plot with the best and average fitness across the generations, with mutation rate equal to 0.45.

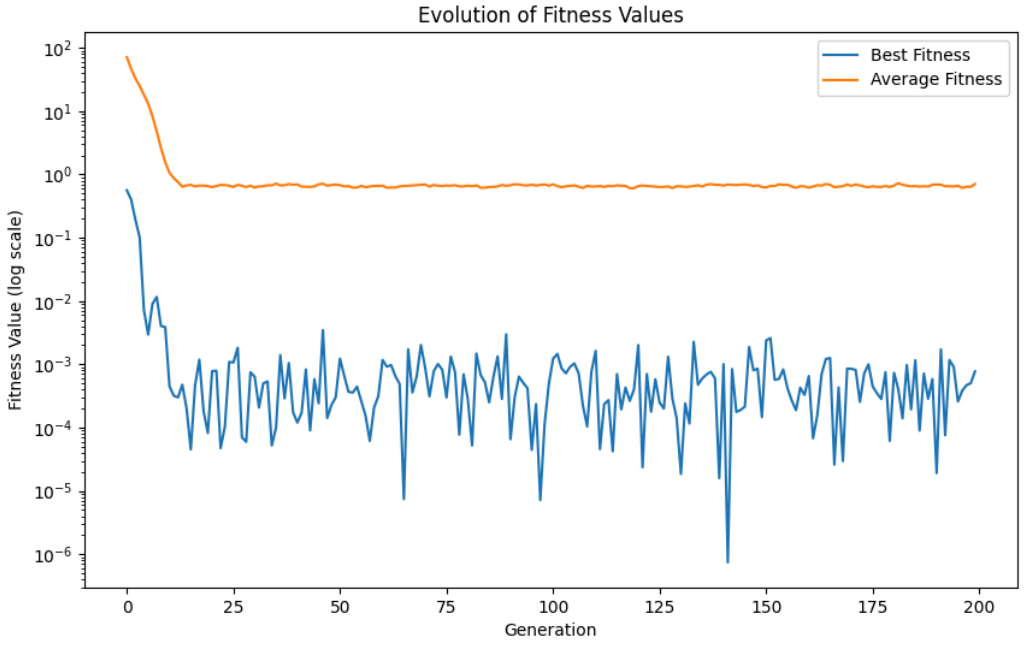

Figure 3.13. - Plot with the best and average fitness across the generations, with mutation rate equal to 1.

Mutation rate	Mutation strength	Fitness
0.25	0.1	5.68E-14
0.25	0.25	5.68E-14
0.25	0.5	4.97E-13
0.25	0.7	1.14E-10
0.25	0.75	-5.68E-14
0.25	0.725	7.89E-11
0.25	0.775	5.28E-07
0.25	0.8	-5.68E-14
0.25	0.85	1.78E-08
0.25	0.9	-5.68E-14
0.25	1.0	2.56E-13

Table 3.9. - Results of the experiments for task 4, with changing mutation strengths.

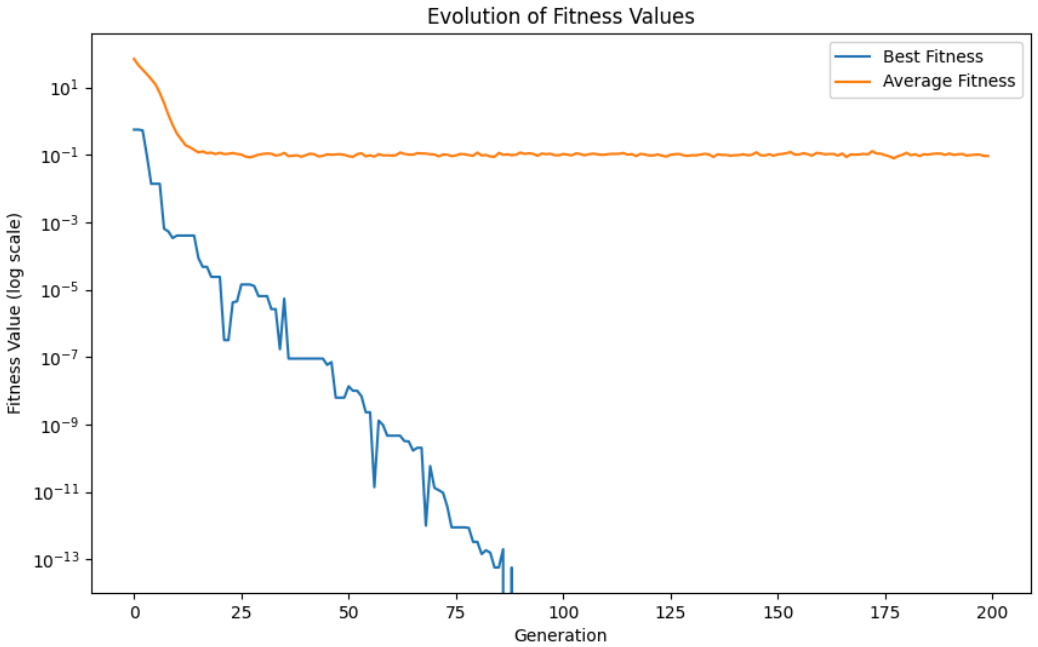

Figure 3.14. - Plot with the best and average fitness across the generations, with mutation strength equal to 0.1.

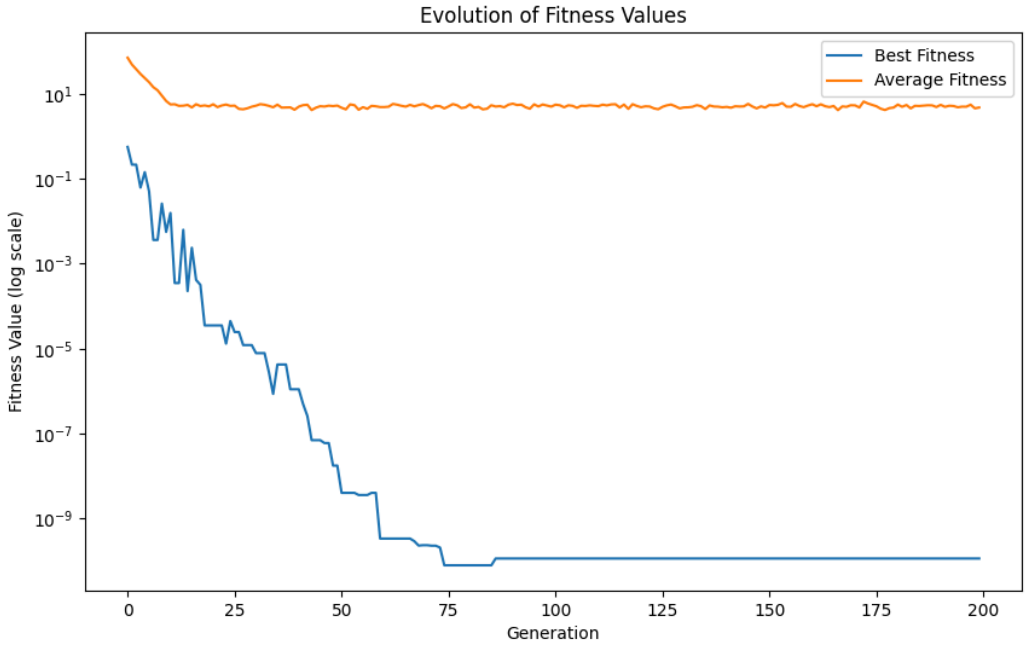

Figure 3.15. - Plot with the best and average fitness across the generations, with mutation strength equal to 0.7.

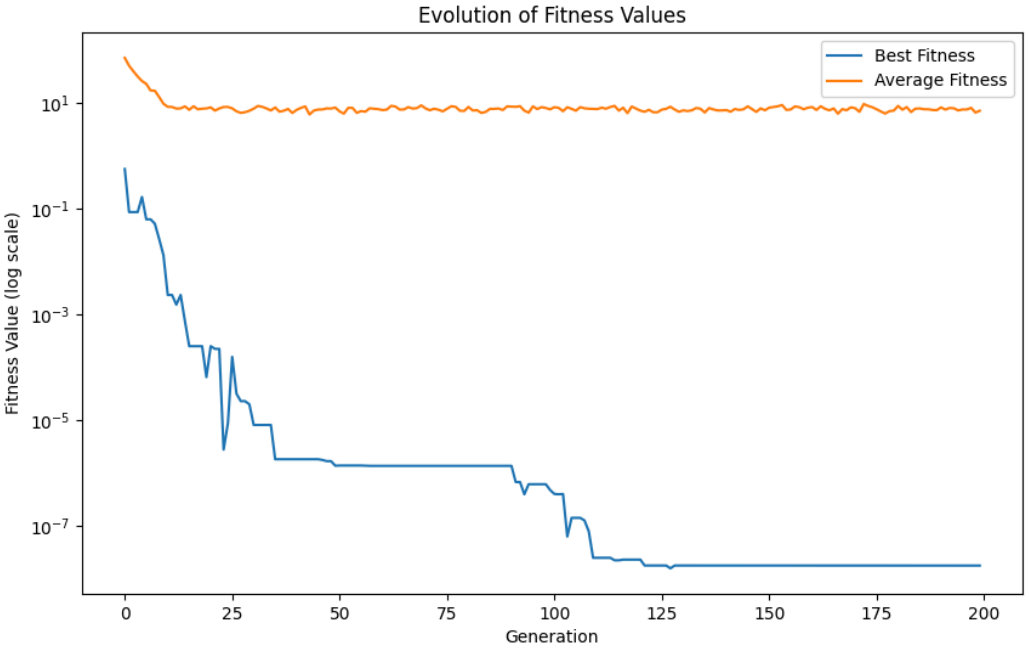

Figure 3.16. - Plot with the best and average fitness across the generations, with mutation strength equal to 0.85.

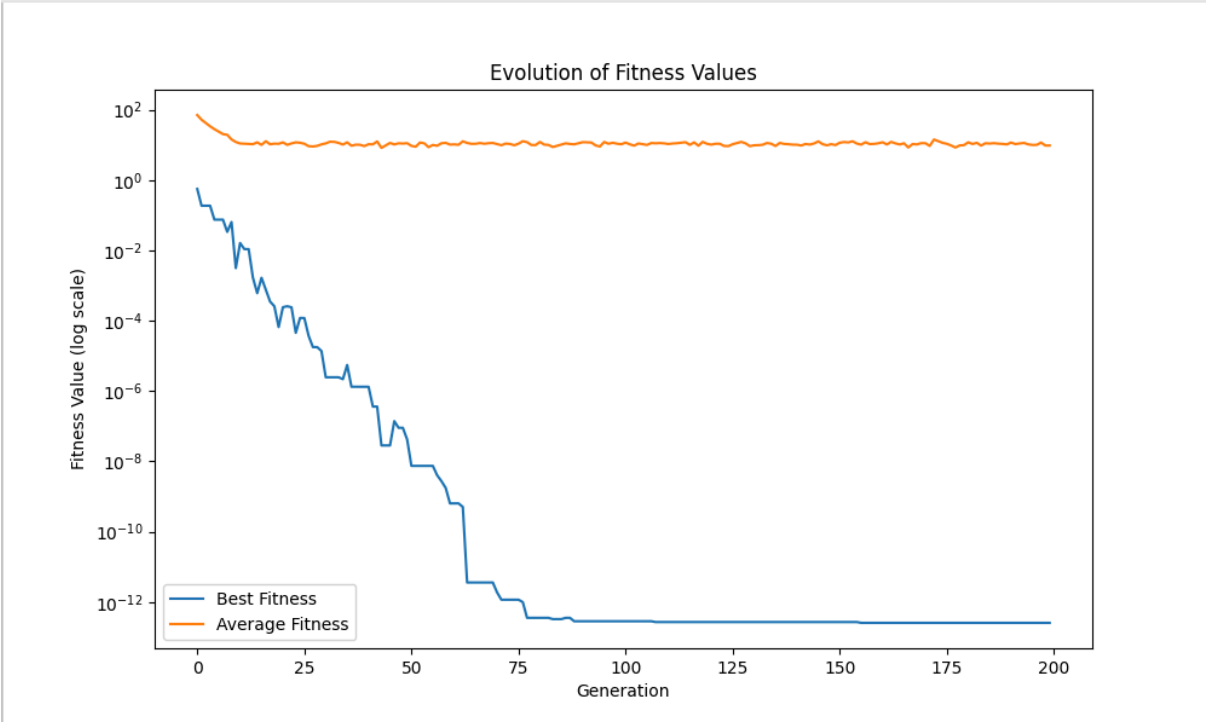

Figure 3.17. - Plot with the best and average fitness across the generations, with mutation strength equal to 1.

The mutation rate determines the probability of a mutation occurring for a given individual within the population during each generation. A higher mutation rate means more individuals will undergo mutation, leading to a more diverse population. This increased diversity can help explore a larger portion of the search space, which is beneficial for avoiding getting trapped in a local optimum and finding better solutions. However, suppose the mutation rate is too high. In that case, it may lead to excessive exploration, with a lack of exploitation, slowing down the convergence and increasing the amount of generations needed for convergence. On the other hand, a low mutation rate may result in a lack of variety in the population, leading to premature convergence and trapping the algorithm in local optima.

Mutation strength refers to the magnitude of change caused by a mutation. It determines how much an individual's parameters are altered when it undergoes mutation. Strong mutations can lead to more significant changes in the population, potentially enabling rapid search space exploration. This can be advantageous in evading local optima and finding unexplored solutions. On the contrary, overly strong mutations may introduce too much disruption, making it difficult for the algorithm to converge to optimal or near-optimal solutions. It can also lead to the loss of suitable parameters obtained during evolution. Weak mutations may result in incremental changes that are insufficient to explore the search space effectively, potentially slowing down the algorithm's convergence.

During our experiment, we observed the impact of both mutation and mutation strength on the algorithm's convergence. As expected previously, for this particular experiment, changing the mutation rate results in more changes than changing the mutation strength. We also see that the average fitness value is less prone to instability caused by the mutation than the best fitness value for each generation, which can fluctuate highly for higher mutation rates.