# Laboratory 3

# Variant 3

# Group 3


# By Krzysztof Kotowski and Juan Manuel Aristizabal Henao


# Introduction

In this exercise, our task is to optimize a certain function (Bukin), by means of evolutionary algorithm, which works like:

1. Create a population of N individuals (each is a set of input values for a certain evaluation function)
2. For M generations:
    a. Assign a value to each individual, by calculating the value using an evaluation function, with individual's set of input values
    b. Choose fraction of fit individuals by making a tournament:
        i. Choose two random individuals
        ii. The one with better evaluation value "survives" and is added to a list of fit individuals
    c. Use the individuals from the fraction as parents of a new generation of individuals:
        i. Choose 2 random individuals as parents
        ii. Create 2 offspring individuals by making their set of input values a combination of parent's set of input values as a synthesis of them along some fraction for first parent and reciprocal for second parent (ex. ¼ of genetic code of first parent and 1-¼=¾ of second parent)
        iii. Create offspring until the new population (of some certain amount) is created
    d. For each newly created individual, introduce random mutations to their "genetic code" (set of input values)
3. Function returns the best-fitting individual's set of input values

Evolutionary algorithms are great for empirically optimising functions, which can be applied to solve a vast amount of real life problems. However, it is important to take an appropriate

population size, so that it is big enough for accurate finding and small enough, so that it can be efficiently calculated using computational resources.

# Implementation

We'll explain our code going in order of function encountering during execution, so starting point is main.py:

## [main()]

```python
if __name__ == "__main__":
    # TODO Experiment 1...
    ga = GeneticAlgorithm(
        population_size=15000,
        mutation_rate=0.05,
        mutation_strength=0.5,
        crossover_rate=0.5,
        num_generations=100,
    )
    best_solutions, fitness_values, avg_fitness_values = ga.evolve(seed=5)
    plot_fitness_values(fitness_values)
    plot_average_fitness_values(avg_fitness_values)
    print("Best solution: ", best_solutions[-1])
```

Here we initialize the GeneticAlgorithm object, so that we can call its evolve function, that will make the whole genetic optimization. It will return values that will be used to visualize how the population evolved during the iterative execution of a genetic algorithm

[GeneticAlgorithm]

```python
class GeneticAlgorithm:
    def __init__(
            self,
            population_size: int,
            mutation_rate: float,
            mutation_strength: float,
            crossover_rate: float,
            num_generations: int,
    ):
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.mutation_strength = mutation_strength
        self.crossover_rate = crossover_rate
        self.num_generations = num_generations
```

Loads required values for genetic evolution:

- Population size
- Mutation rate
- Mutation strength
- Crossover rate
- Number of generations

[GeneticAlgorithm.evolve() Part 1]

```python
    def evolve(self, seed: int) -> ...:
        # Run the genetic algorithm and r
        #   the best fitness for each ger
        set_seed(seed)
```

Initially, the seed python's random number generator is set

[set_seed()]

```python
def set_seed(seed: int) -> None:
    # Set fixed random seed to make the results reproducible
    random.seed(seed)
    np.random.seed(seed)
```

## [GeneticAlgorithm.evolve() Part 2]

```
population = self.initialize_population()
```

At first, the population variable is initialized using initialize_population()

### [GeneticAlgorithm.initialize_population()]

```
def initialize_population(self) -> np.ndarray:
    # Initialize the population and return the result

    # Returns  random uniform distribution of values over [0,1).
    uni_population = np.random.rand(self.population_size, 2)

    # Re-scaling of the population from [0,1) to both [-15, -5] for x and [-3, 3] for y.
    scale_population = np.multiply(uni_population, [10, 6])
    new_population = np.add(scale_population, [-15, -3])
    return new_population
```

Population is initialized as an array of 2D tuples (x,y), where:

- -15 <= x < -5
- -3 <= y < 3

## [GeneticAlgorithm.evolve() Part 3]

```
best_solutions = []
best_fitness_values = []
average_fitness_values = []
```

Arrays for plots in main() are initialized. They will hold values generated in each iteration of evolutionary algorithm

## [GeneticAlgorithm.evolve() Part 4]

```
for generation in tqdm.tqdm(range(self.num_generations)):
```

Evolution algorithm loop. It used “tqdm” object as a wrapper that will display in a console a progress bar as such:

```
PS C:\Users\DELL\OneDrive\Pulpit\Backup laptopa\Presentations and problem sets\S8\EARIN\Labs\lab3\EARIN_Lab3> python .\main.py
 14%|                                                                                | 14/100 [00:04<00:27,  3.08it/s]
```

## [GeneticAlgorithm.evolve() Part 5]

```
fitness_values = self.evaluate_population(population)
best_fitness = np.min(fitness_values).tolist()
average_fitness = np.mean(fitness_values).tolist()
best_solution = population[np.where(fitness_values == best_fitness)[0][0]].tolist()
```

Calls an evaluation function (Bukin), that calculates values of each member of a population.

[evaluate_population()]

```
def evaluate_population(self, population) -> np.ndarray:
    # Evaluate the fitness of the population and return the values f

    evaluation = [bukin_2d(val[0], val[1]) for val in population]
    return np.array(evaluation)
```

For each tuple generated as a population, there is a fitness value generated for it from bukin_2d(: float, : float) function. Returns an array of fitness values in order corresponding to an order of population members

[bukin_2d()]

```
def bukin_2d(x, y):
    """
    Bukin function in 2D.

    Parameters:
        x (float): The x-coordinate.
        y (float): The y-coordinate.

    Returns:
        float: The value of the Bukin function at the given point.
    """
    return 100 * np.sqrt(np.abs(y - 0.01 * x**2)) + 0.01 * np.abs(x + 10)
```

Function provided in task description:

3. Optimize Bukin function using Tournament Selection:

$$f(x, y) = 100\sqrt{|y - 0.01x^2|} + 0.01|x + 10|.$$

Initialize $x$ from range $[-15, 5]$ and $y$ from range $[-3, 3]$.

## [GeneticAlgorithm.evolve() Part 6]

```
parents_for_reproduction = self.selection(population, fitness_values)
```

Selects a list of parents of future offspring (children)

[GeneticAlgorithm.selection()]

```
def selection(self, population, fitness_values) -> np.ndarray:
    # Implement selection mechanism (Tournament Selection) and return the selected

    selected_individuals = np.empty(shape=(0, 2))
    for _ in range(self.population_size // 2):
        while True:
            first_contestant = np.random.randint(len(population))
            second_contestant = np.random.randint(len(population))
            if first_contestant != second_contestant:
                break
```

```
            if fitness_values[first_contestant] <= fitness_values[second_contestant]:
                selected_individual = first_contestant
            else:
                selected_individual = second_contestant

            individual = population[selected_individual]
            selected_individuals = np.append(selected_individuals, [individual], axis=0)

            # population = np.delete(population, first_contestant, axis=0)
            # fitness_values = np.delete(fitness_values, first_contestant)
            #
            # population = np.delete(population, second_contestant, axis=0)
            # fitness_values = np.delete(fitness_values, second_contestant)

        return selected_individuals
```

Selects a number of contestants equal to a half of the population number. Each contestant is chosen by “fighting” with some other randomly chosen contestant by means of comparing fitness values. When the number of “candidates” is selected, the function returns a list of them.

## [GeneticAlgorithm.evolve() Part 7]

```
            offspring = self.crossover(parents_for_reproduction)
```

Creates the future generation of children from parents, using a crossover function

### [GeneticAlgorithm.crossover()]

```
    def crossover(self, parents) -> np.ndarray:
        # Implement the crossover mechanism over the parents and return the offspring

        children = np.empty(shape=(0, 2))
        while len(children) < self.population_size:
            while True:
                first_parent = np.random.randint(len(parents))
                second_parent = np.random.randint(len(parents))
                if first_parent != second_parent:
                    break

            crossover = np.random.random(size=None)
            if crossover < self.crossover_rate:
                alpha = np.random.random(size=None)
                offspring1 = self.rand_interpolation(alpha, parents[first_parent], parents[second_parent])
                offspring2 = self.rand_interpolation(alpha, parents[second_parent], parents[first_parent])
            else:
                offspring1, offspring2 = parents[first_parent], parents[second_parent]

            children = np.append(children, [offspring1, offspring2], axis=0)

        return children
```

Creates a list of children of length equal to the population number. For each pair of randomly chosen parents, it chooses a random crossover alpha value and creates and makes two children out of them. Returns a list of generated children

### [GeneticAlgorithm.rand_interpolation()]

```
    @staticmethod
    def rand_interpolation(alpha: float, parent1, parent2) -> np.ndarray:
        x = alpha * parent1[0] + (1 - alpha) * parent2[0]
        y = alpha * parent1[1] + (1 - alpha) * parent2[1]
        return np.array([x, y])
```

Creates children value tuples as a sum of dissected parents' tuple values

## [GeneticAlgorithm.evolve() Part 8]

```
            population = self.mutate(offspring)
```

Makes random mutations in a newly created population

### [GeneticAlgorithm.mutate()]

```
def mutate(self, individuals: np.ndarray) -> np.ndarray:
    # Implement mutation mechanism over the given individuals and return the results

    for individual in individuals:
        if np.random.random() < self.mutation_rate:
            np.add(individual, self.gaussian_op(), out=individual)
            np.minimum(individual, np.array([-5, 3]), out=individual)
            np.maximum(individual, np.array([-15, -3]), out=individual)

    return individuals
```

For each population member (individual), "tosses a coin" and if the randomly generated value is lower than the mutation rate (for a high number of members, it will amount to about the same fraction of population, as the "percentage value" of mutation rate)

## [GeneticAlgorithm.evolve() Part 9]

```
            best_solutions.append(best_solution)
            best_fitness_values.append(best_fitness)
            average_fitness_values.append(average_fitness)
```

Logs the values for visualisation purposes

## [GeneticAlgorithm.evolve() Part 10 - Final]

```
        return best_solutions, best_fitness_values, average_fitness_values
```

After the loop finishes (all generations simulated), returns the list of values for visualisation

# Experiments

1. **Finding genetic algorithm parameters**:
   The table below depicts 8 experiments run with the specified parameters and their respective results. Experiment 4 is highlighted since it demonstrated the best results. Additionally, the seed value was 5.

| Experiment | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| **Population** | 2000 | 2000 | 2000 | 10000 | 10000 | 10000 | 10000 | 10000 |
| **Mutation Rate** | 0,05 | 0,05 | 0,05 | 0,05 | 0,1 | 0,05 | 0,05 | 0,05 |
| **Mutation strength** | 0,5 | 6 | 6 | 6 | 6 | 6 | 6 | 2 |
| **Crossover rate** | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 | 0,9 | 0,1 | 0,5 |
| **Number of generations** | 100 | 100 | 150 | 150 | 150 | 150 | 150 | 150 |
| **Average fitness values** | 2,9851 | 7,6378 | 7,2086 | 7,4531 | 16,251 | 8,1928 | 7,4701 | 5,6469 |
| **Best Fitness value** | 0,0042 | 0,0021 | 0,0021 | 0,0001 | 0,0023 | 0,2137 | 0,0647 | 0,0060 |
| **Best Solution** | (-9,756 , 0,917) | (-9,786 , 0,958) | (-9,786 , 0,958) | (-10,01 , 1,003) | (-10,23 , 1,048) | (-9,768 , 0,954) | (-7,792 , 0,607) | (-9,400 , 0,884) |

2. **Randomness in genetic algorithm.**

| Seed | 5 | 10101 | 58496 | 123456 | 987654 |
|---|---|---|---|---|---|
| **Average Fitness values** | 7,45307 | 7,55474 | 7,77167 | 7,65442 | 7,71778 |
| **Best Fitness values** | 0,000147 | 0,003948 | 0,002069 | 0,001823 | 0,005444 |
| **Best Solution** | (-10,0147 , 1,00294) | (-9.6051 , 0.92259) | (-9.7931 , 0.95905) | (-10.182 , 1.03679) | (-9.456 , 0.89408) |

The best solution from the 5 runs was **[ -10.0147, 1.00294 ]**, and its fitness value was **1.47e-4**.
The standard deviation (sampled) of the average fitness value across all the tested seeds was: $\sigma$ = **0.12781**
The greatest difference in results, when looking at best fit values, is between seeds 5 and 987654 - in the case of second seed, the difference in value is in the order of multiplying by 50. This deviation shows the extent, to which the algorithm is dependant on randomness.

The table below depicts the results obtained from the experiments done reducing the population but keeping all previous parameters. (e.g. 100%, 50%, 25% and 10%):

| Population | 10000 | 5000 | 2500 | 1000 |
|---|---|---|---|---|
| **Average Fitness Value** | 7,45307 | 8,91761 | 8,11936 | 9,09509 |
| **Best Fitness Value** | 0,000147 | 0,002051 | 0,000926 | 0,001471 |
| **Best Soultion** | (-10,0147 , 1,00294) | (-9.7949 , 0.95941) | (-9.9073 , 0.98156) | (-9.8528 , 0.97079) |

The best fitness value is still found by the full population (e.g., 100%), but the experiment at 25% of the population gave a similar fitness value. In addition, it was noticed that the average fitness value increased as the population decreased.

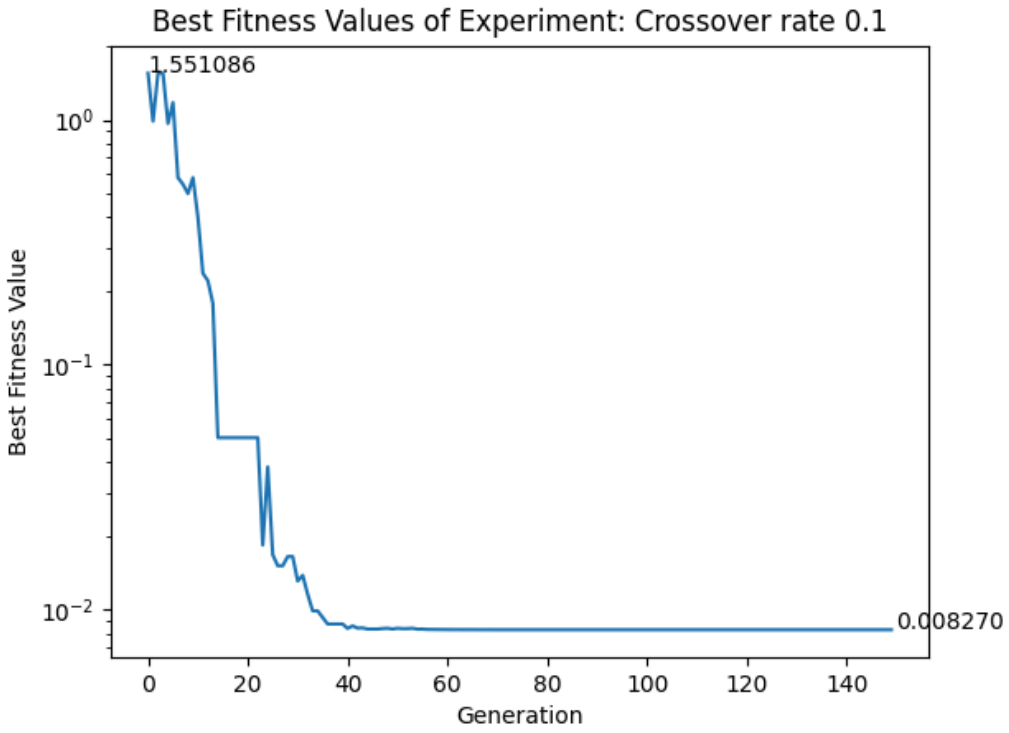
### 3. Crossover impact

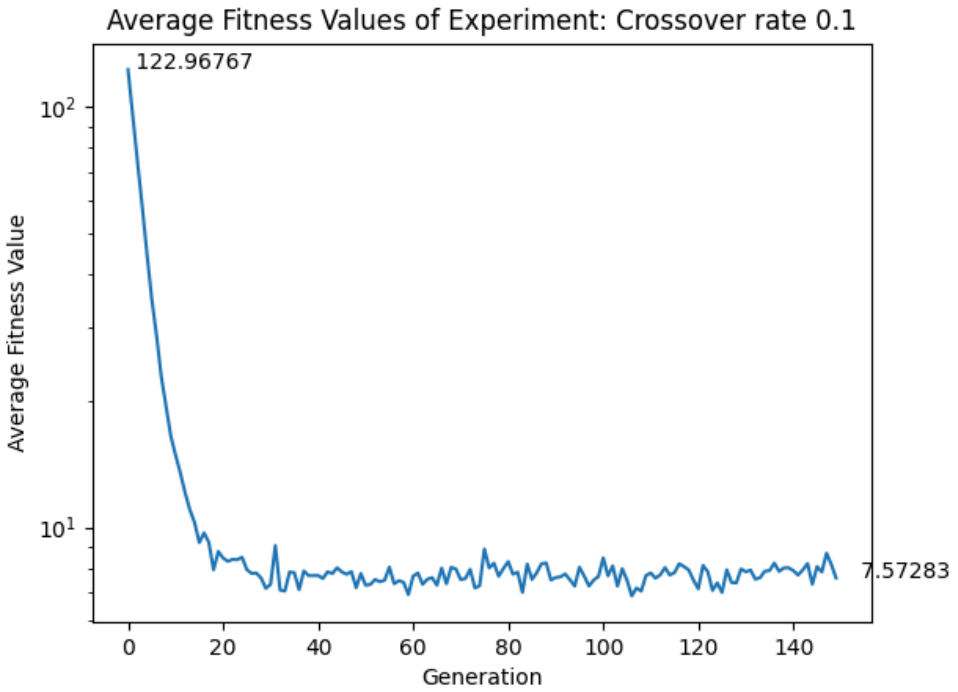
The table below shows the results of the experiments done by changing only the crossover rate:

| Seed | Experiment No. | Crossover rate | Average Fitness Value | Best Fitness Value | Best Solution |
|---|---|---|---|---|---|
| 456789 | 1 | 0,1 | 7,49586 | 0,013509 | (-11.3509, 1.28843) |
| | 2 | 0,25 | 7,54603 | 0,015573 | (-8.4427, 0.71279) |
| | 3 | 0,5 | 7,37460 | 0,001127 | (-9.8873, 0.97759) |
| | 4 | 0,75 | 7,82460 | 0,500597 | (-9.9658, 0.99319) |
| | 5 | 0,9 | 7,66096 | 0,284662 | (-9.7793, 0.95635) |
| 963852 | 6 | 0,1 | 7,57183 | 0,008270 | (-10.8270 , 1.17224) |
| | 7 | 0,25 | 7,24619 | 0,014115 | (-8.5885, 0.73761) |
| | 8 | 0,5 | 7,12209 | 0,000210 | (-9.9790, 0.99580) |
| | 9 | 0,75 | 8,15570 | 0,004202 | (-9.58130, 0.91801) |
| | 10 | 0,9 | 7,84191 | 0,301938 | (-9.9355, 0.98716) |

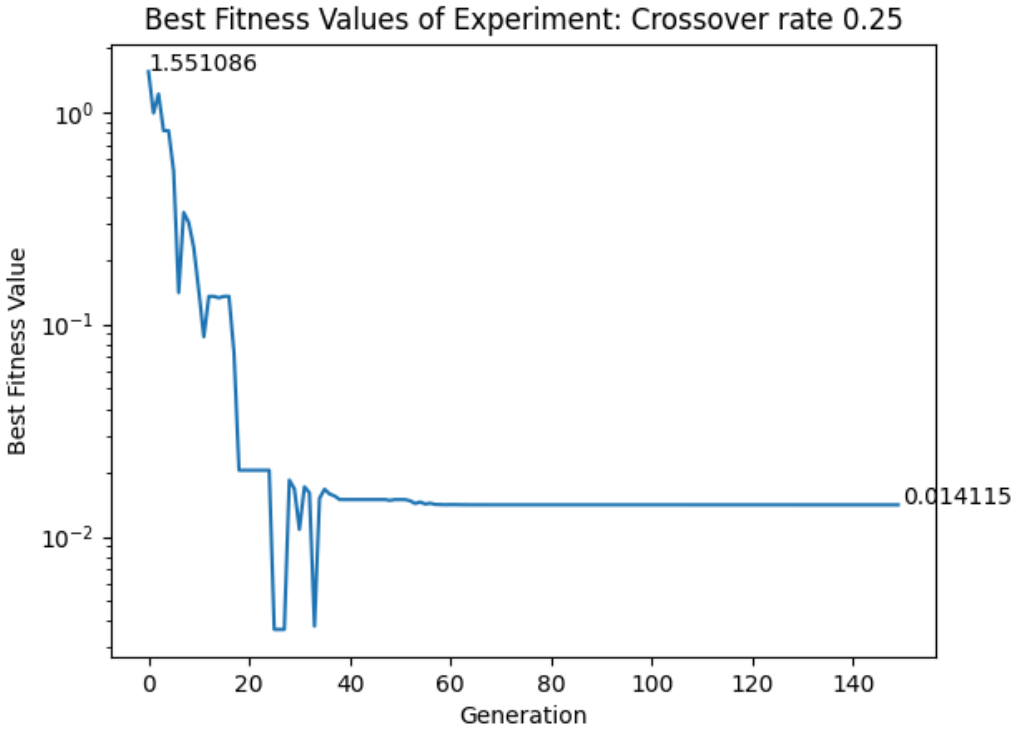
From these results, it can be seen that the closer the crossover rate is to the value of 0.5, the better it performs in finding the optimal value. As the values of the crossover rate start increasing above or below that rate, the performance and the values obtained tend to be more arbitrary, and the behaviour of the algorithm seems more erratic.

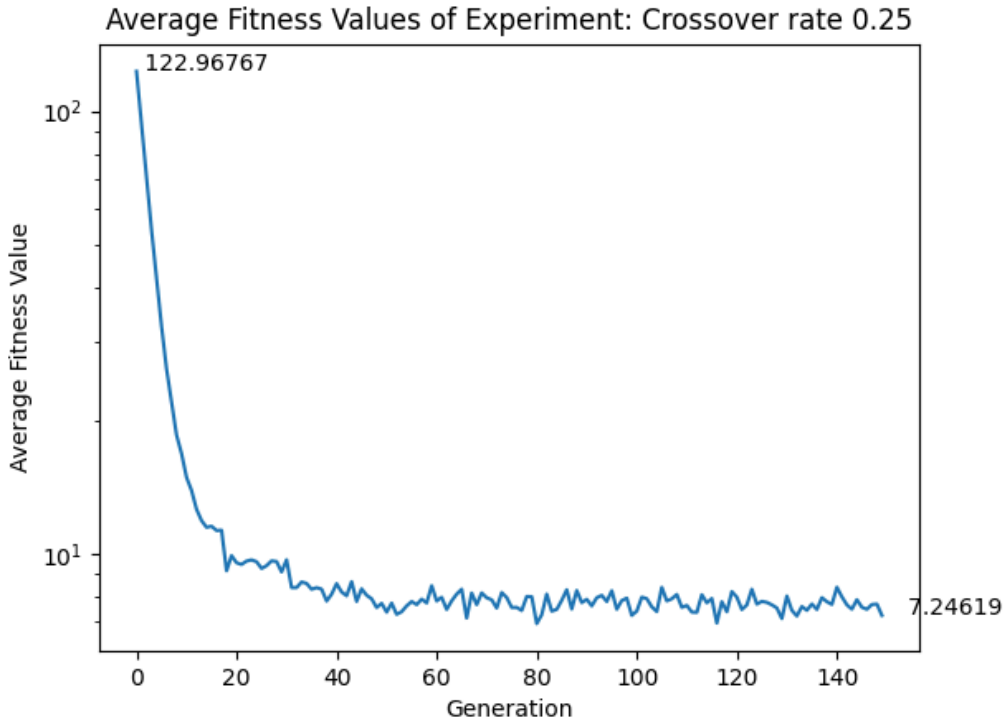
The following graph depicts the results of running the genetic algorithm with a crossover rate of 0.1:



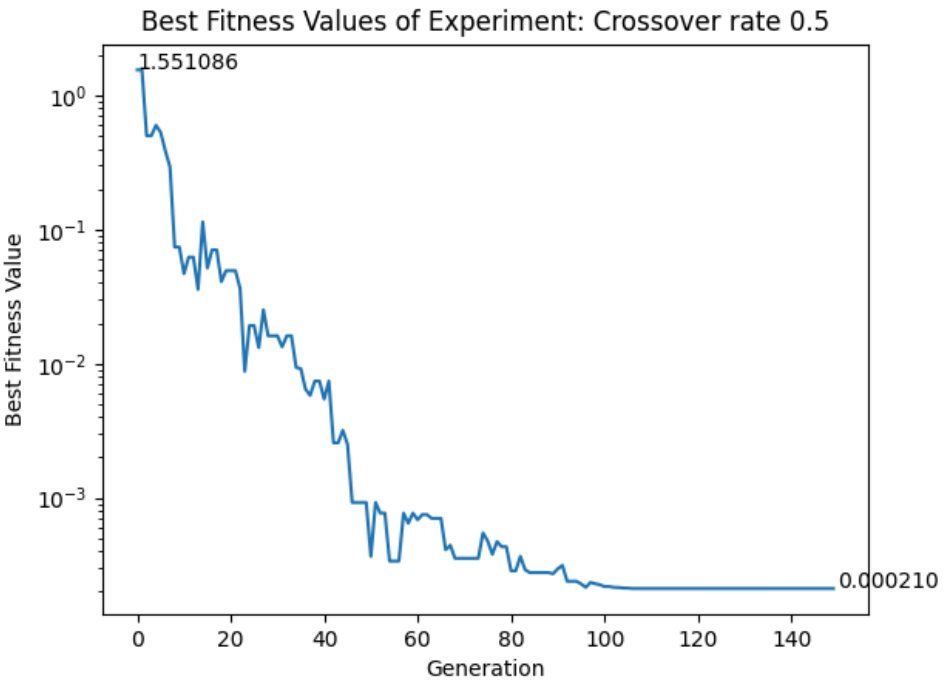



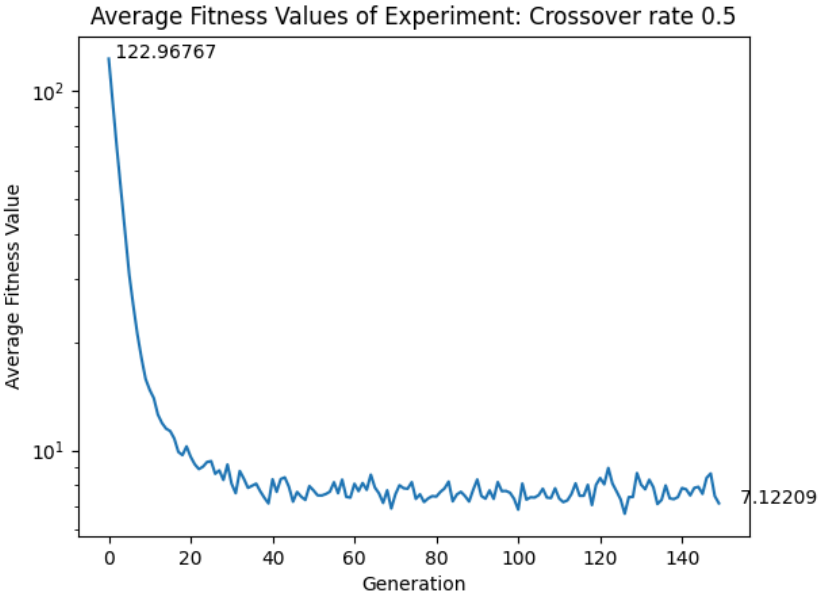
The following graphs depict the results obtained from the experiments run with a crossover rate of 0.25



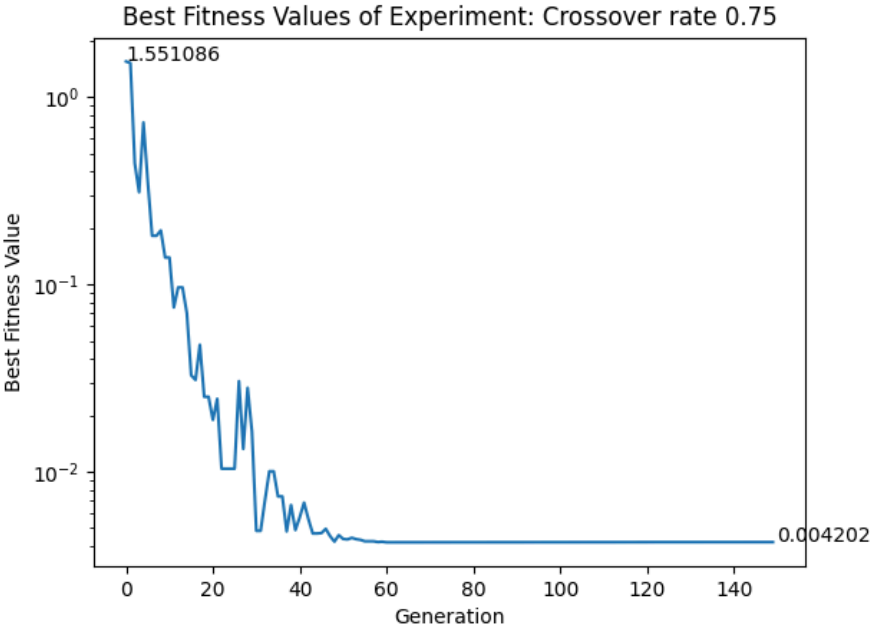



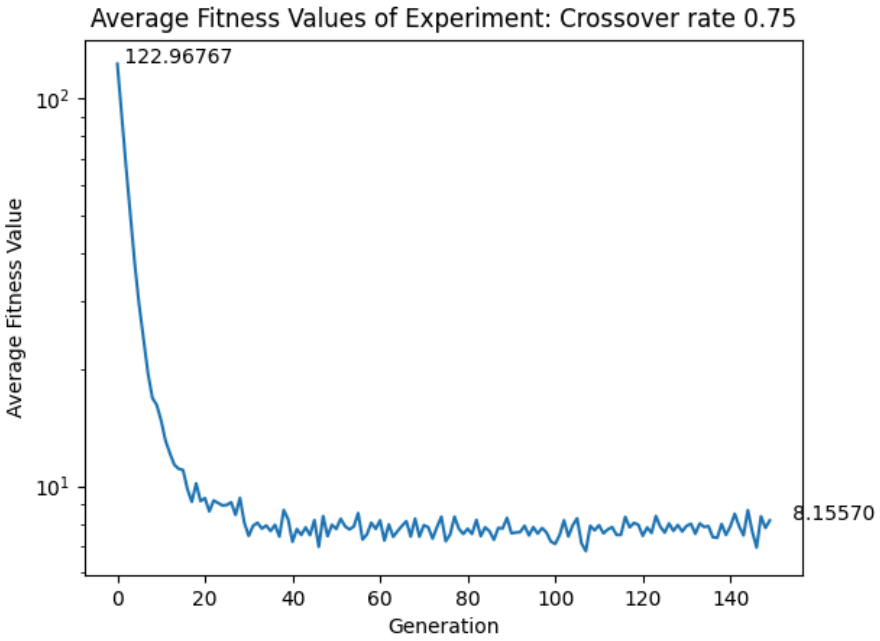
The following graphs depict the results obtained with the experiments with a crossover rate of 0.5:



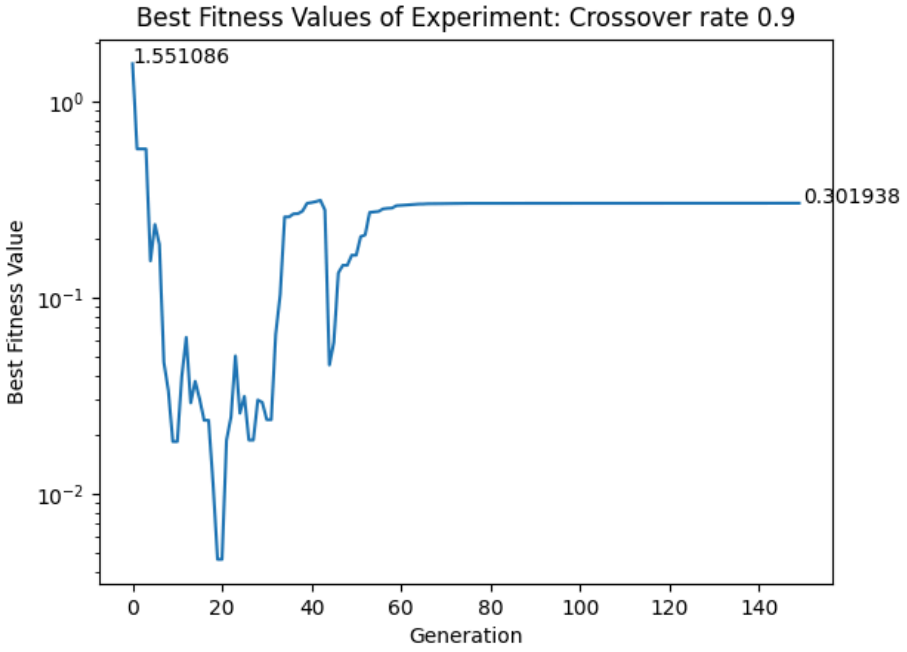



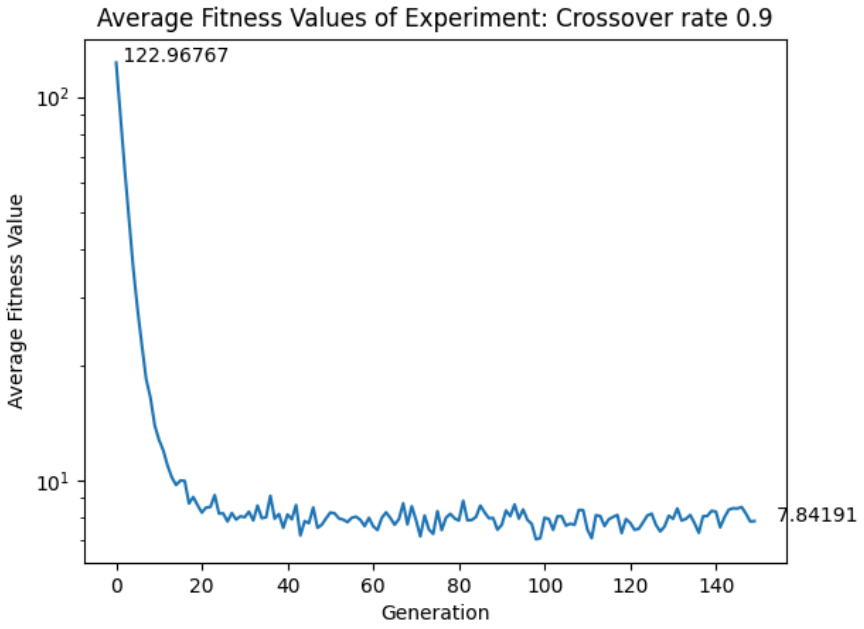
The following graphs show the results obtained from the experiments with a crossover rate of 0.75:



The following graphs show the results of the experiments run with the crossover rate of 0.9:




As the crossover rate approached 0.5 (from both 0.1 and 0.9), the algorithm took longer to find the best fitness value, however after converging, balanced crossover resulted in higher accuracy of the final result. Those effects stem from the fact that if an offspring has an even ratio of both parents genes, it will result in a much more balanced outcome, even if counting the mutation. However if children are more like a singular parent, then on average, children are converging to a lesser degree and the overall population is more varied value-wise.

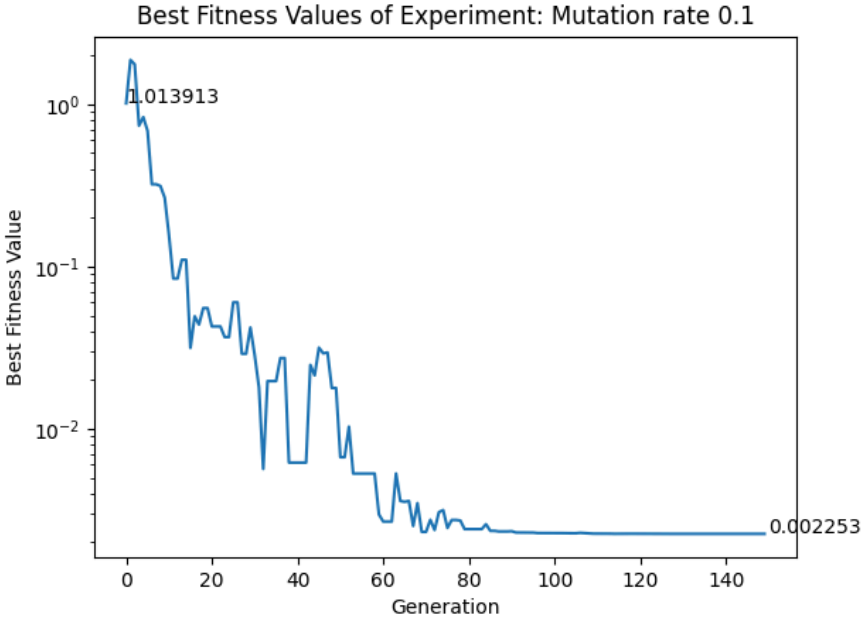
### 4. Mutation and the convergence

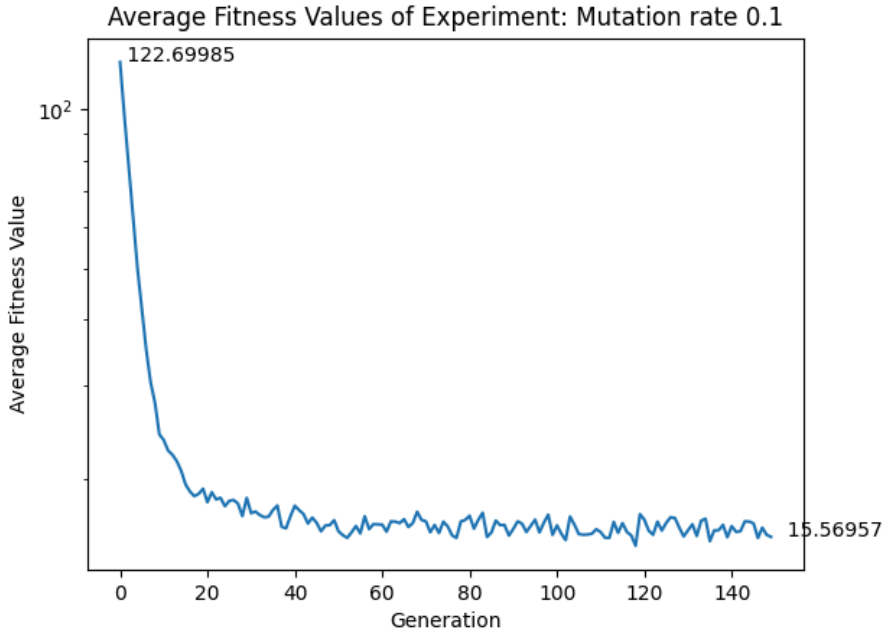
The following table shows the results of the experiments done by changing the mutation rate of the genetic algorithm:

| Seed | Experiment No. | Mutation rate | Average Fitness Value | Best Fitness Value | Best Solution |
|---|---|---|---|---|---|
| 963852 | 1 | 0,1 | 16,16919 | 0,003495 | (-9.6505, 0.93133) |
| | 2 | 0,05 | 7,12209 | 0,000210 | (-9.9790, 0.99580) |
| | 3 | 0,001 | 0,10862 | 0,003013 | (-9.6987, 0.94064) |
| 789123 | 4 | 0,1 | 15,56957 | 0,002253 | (-9.77474, 0.95546) |
| | 5 | 0,05 | 7,63504 | 0,001429 | (-10.1429, 1.02879) |
| | 6 | 0,001 | 0,10974 | 0,000698 | (-9.9302, 0.98608) |

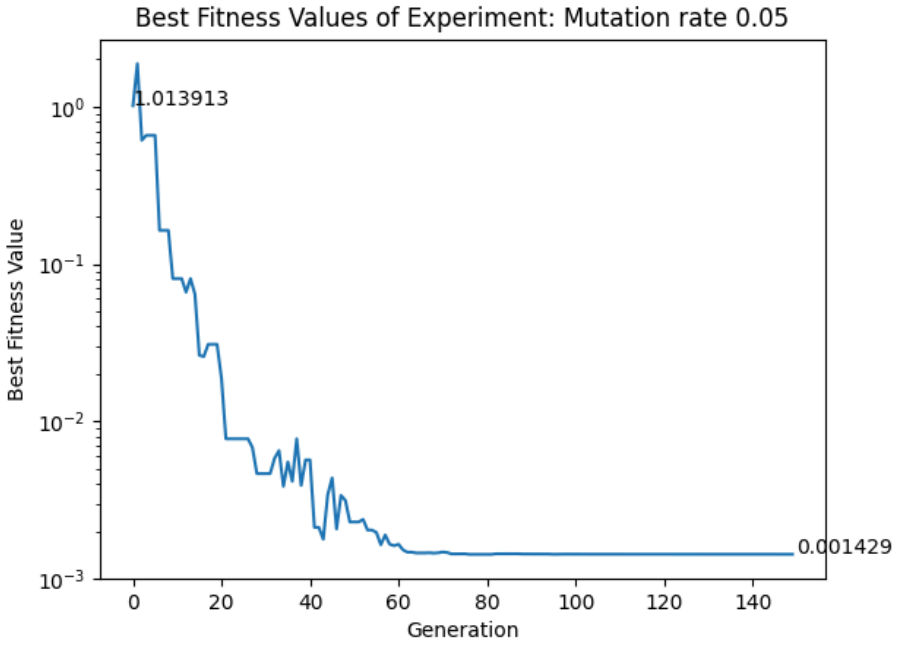
From these results can be seen that depending on how large the mutation rate is, the average fitness values increase or decrease with it. Additionally, in terms of accuracy, the genetic algorithm seems to perform better with low values of the mutation rate, 0.005 being a good candidate for it, since it keeps a significant mutation without going overboard with it.

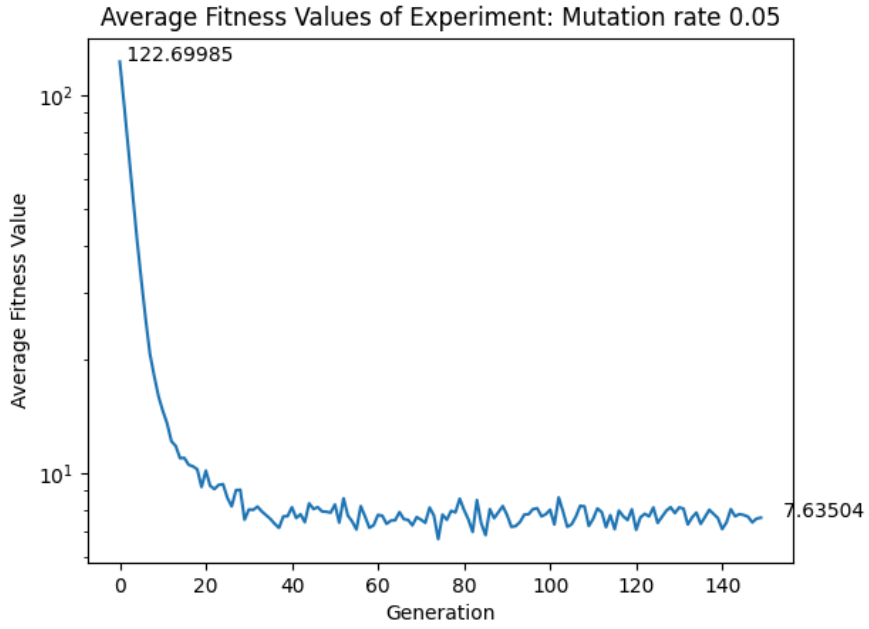
The following graphs depict the results obtained by these experiments depending on the change of the mutation rate (e.g. 0.1).



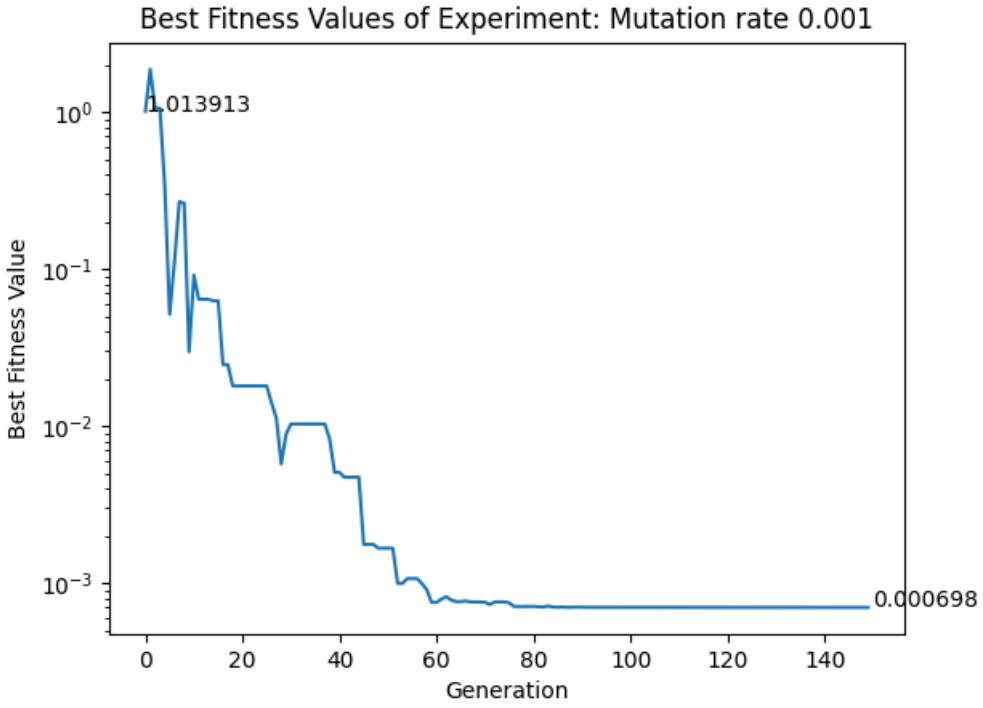



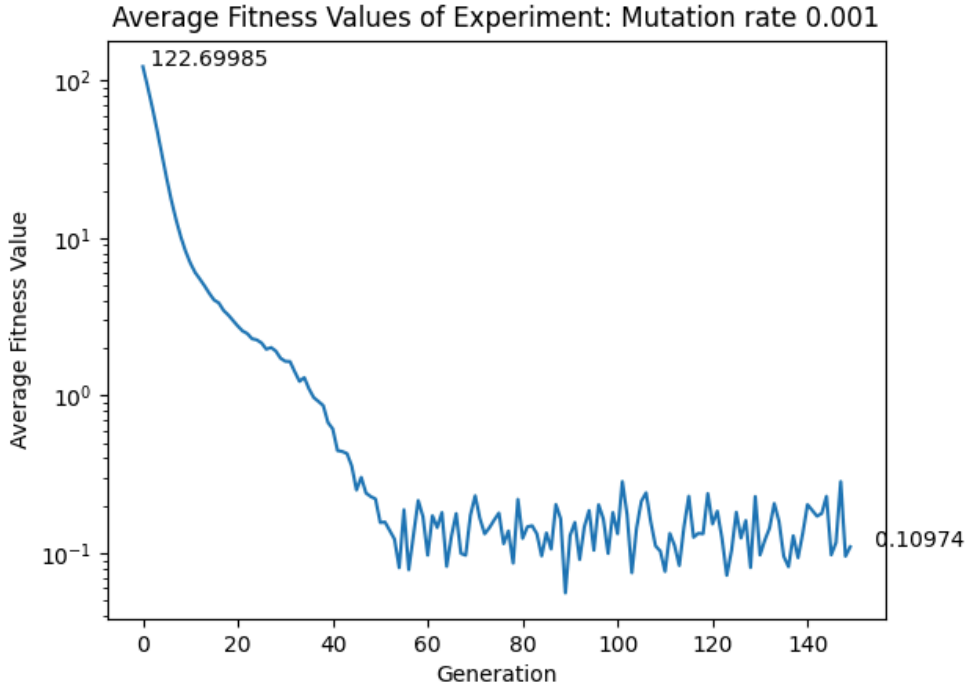
The following graphs depict the results obtained by these experiments depending on the change of the mutation rate (e.g. 0.05).



The following graphs depict the results obtained by these experiments depending on the change of the mutation rate (e.g. 0.001).




From the graphs shown before, it can be concluded that for greater values of the mutation rate, the more jumps the genetic algorithm does before converging to a fitness value, similarly, the lower the mutation rate, the algorithm converges to a value by “stepping down”, meaning, its behaviour resembles more a staircase down to a fitness value.

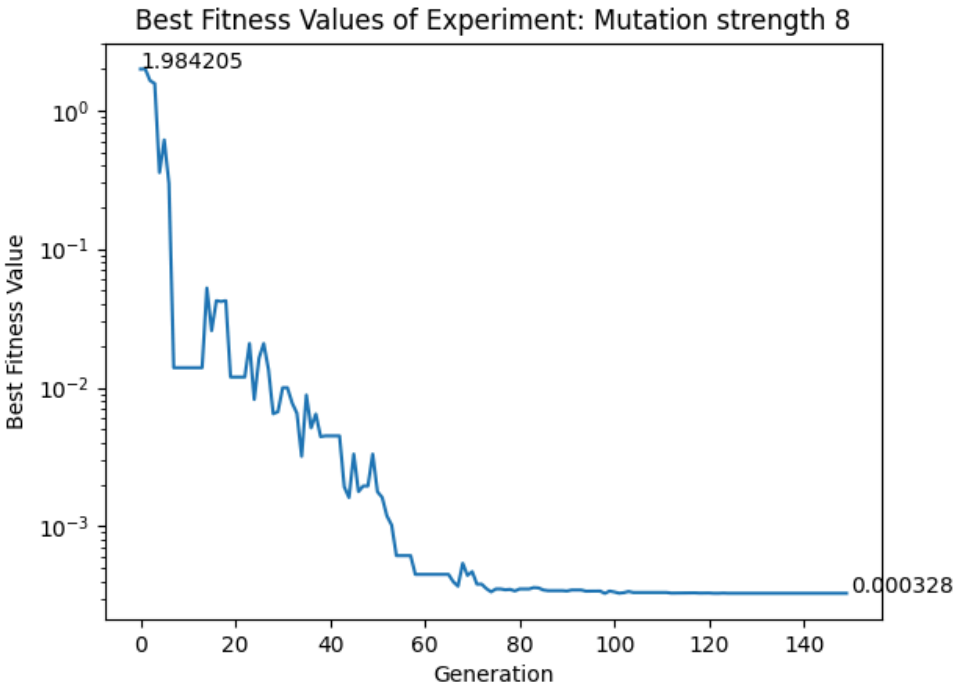
The following table shows the experiments' results by changing the mutation strength value.

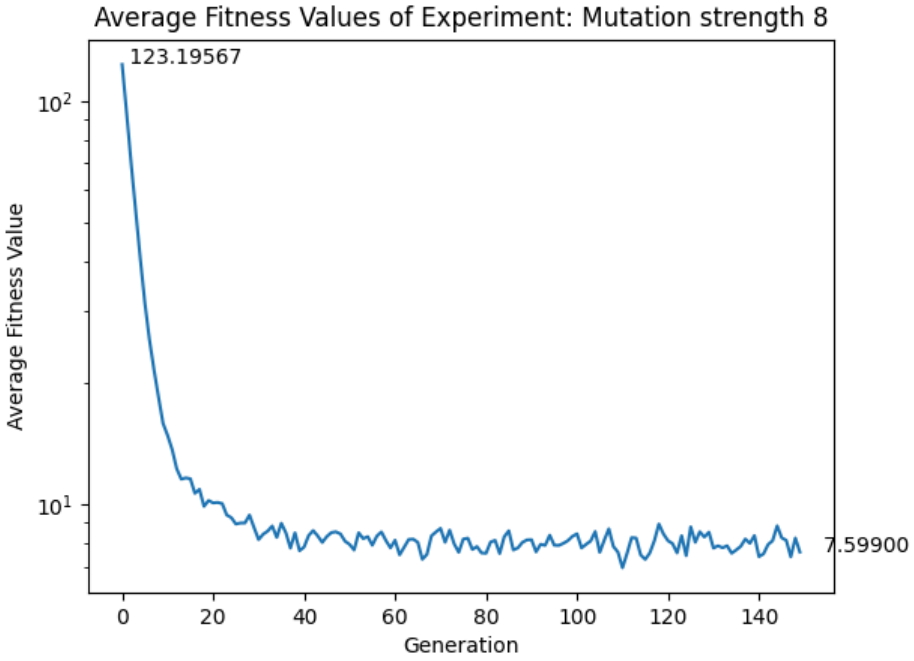
| Seed | Experiment No. | Mutation strength | Average Fitness Value | Best Fitness Value | Best Solution |
|---|---|---|---|---|---|
| 789123 | 1 | 8 | 7,85353 | 0,001760 | (-10.1760, 1.0355) |
| | 2 | 2 | 6,05111 | 0,001088 | (-10.1088, 1.02187) |
| | 3 | 0,5 | 3,19689 | 0,006595 | (-10.6595, 1.13626) |
| 456789 | 4 | 8 | 7,59900 | 0,000328 | (-9.9672, 0.99344) |
| | 5 | 2 | 5,62975 | 0,003498 | (-9.6502, 0.93126) |
| | 6 | 0,5 | 2,96832 | 0,004257 | (-9.5743, 0.91666) |

From these results, it can be seen that for achieving a more optimal fitness value, the strength of the mutation is better to be kept at a value higher than 1, preferably one that can allow the offspring to jump from one place to other still inside the interest zone (e.g., the area in which the Bukin function is defined). The lower values of this strength only seem to aid in the exploitation of a current minima.

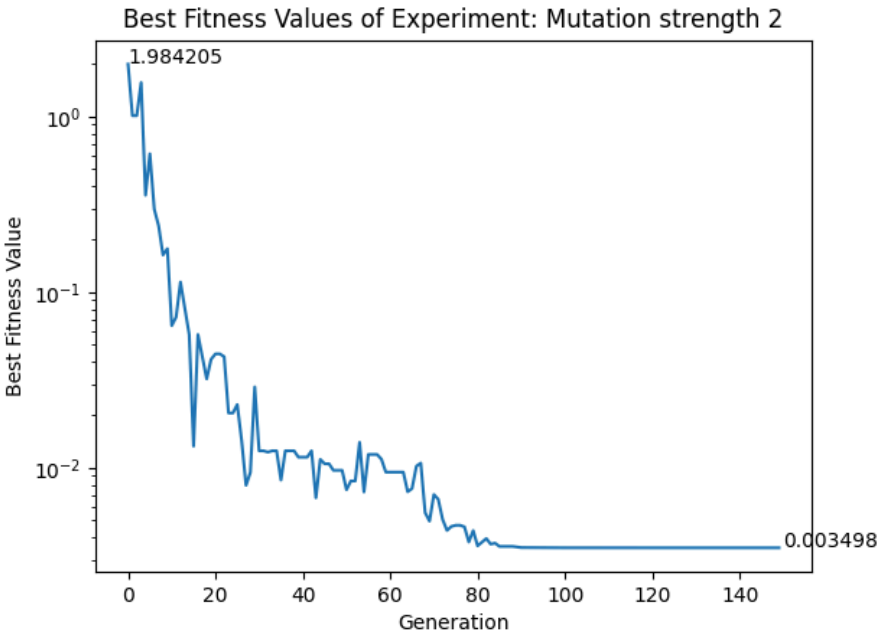
The following graphs depict the impact of the change in mutation strength in the convergence of the genetic algorithm.

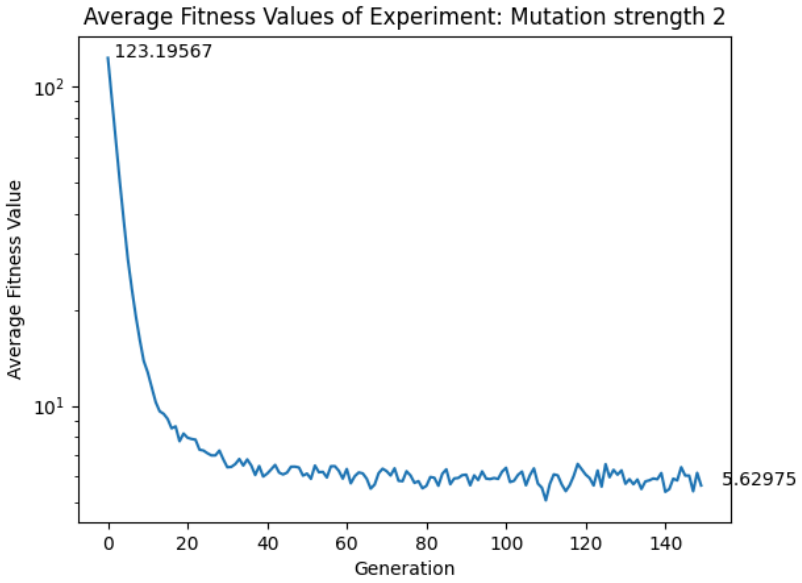
The first one shows the experiment done with a mutation strength of 8.



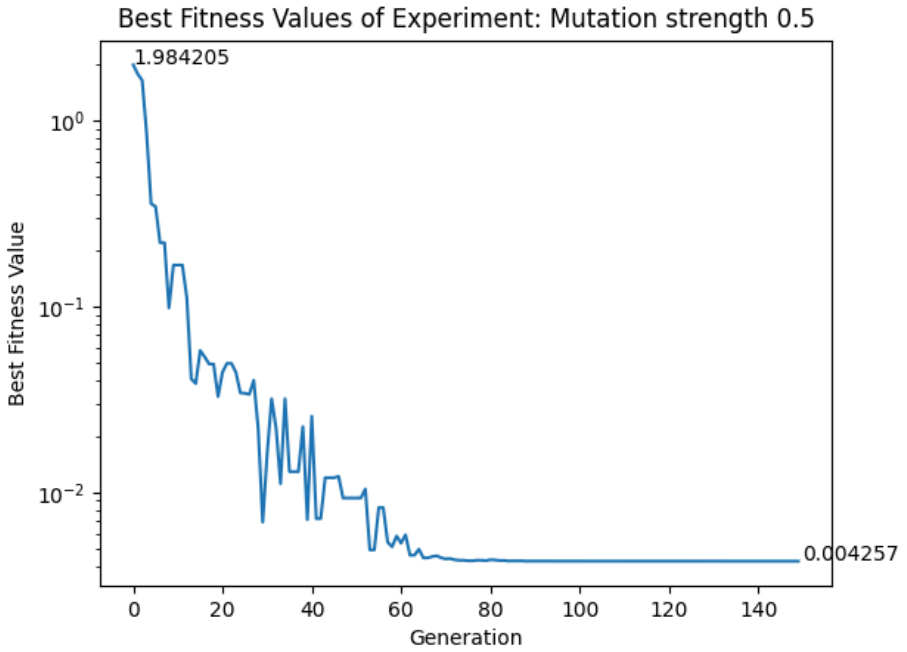



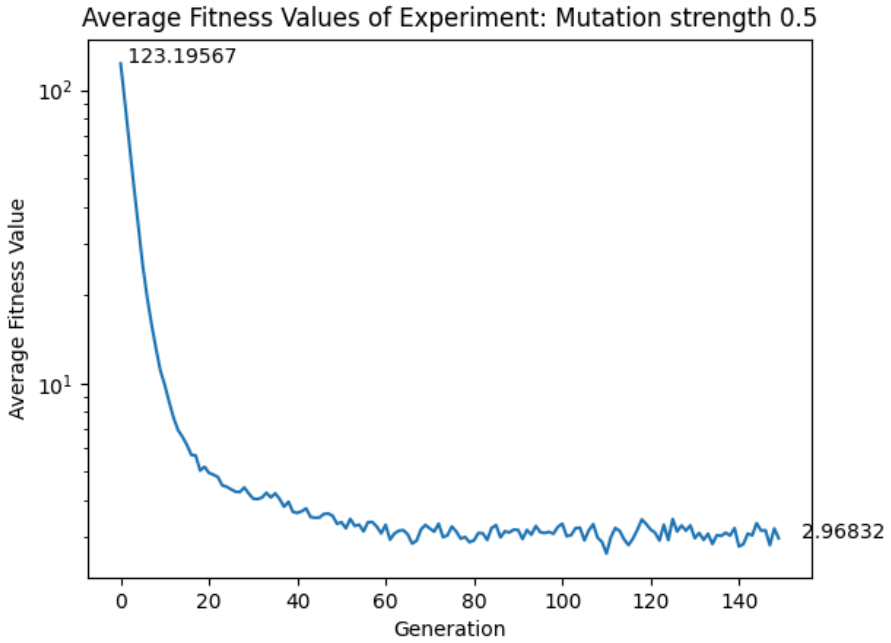
The following graphs depict the experiment done with mutation strength of 2:



The following graphs depict the results of the experiments done with a mutation strength of 0.5:




As it can be seen on the graphs, the behaviour of the genetic algorithm to the strength of the mutation is that for lower values of it, the average fitness value tends to be lower. In addition, the steps taken approaching the best fitness value tend to be more radical in higher values of the mutation strength.