Group 9 :

Kenan Mukendi Kayembe

Furkan Salman

Lab 3: Generational genetic algorithm

Definition of the Booth function to be optimized

```python
# Define the booth function to be optimized
Furkan Salman
def booth_function(x, y):
    return (x + 2 * y - 7) ** 2 + (2 * x + y - 5) ** 2


# Define the function to initialize the population
Furkan Salman
def populate(range_min, range_max, pop_size):
    # randomly select (x,y) pairs from a uniformly distributed range of (range_min, range_max)
    population = np.random.uniform(range_min, range_max, size=(pop_size, 2))
    return population


# Define a function to evaluate the fitness of each individual in the population
Furkan Salman
def evaluate_fitness(population):
    fitness_values = []
    for individual in population:
        # We use negative of booth_function for every individual to calculate their fitness
        # Because we are trying to find the individual that minimizes the function best
        fitness_values.append(-booth_function(*individual))
    return np.array(fitness_values)
```

The genetic algorithm begins by defining the Booth function to be optimized, which takes two input arguments, x and y. The function returns the value of the Booth function for the given x and y.

Next, we define a function called "populate" to initialize the population for the genetic algorithm. The population is a collection of individuals, each represented by a pair of x and y values. The "populate" function randomly selects x and y values from a uniformly distributed range between range_min and range_max. The size of the population is specified by pop_size.

We also define a function called "evaluate_fitness" to evaluate the fitness of each individual in the population. The fitness of an individual is a measure of how well it performs with respect to the optimization problem. In this case, the fitness of an individual is calculated as the negative value of the Booth function for that individual. The negative value is used because we are trying to minimize the Booth function, and genetic algorithms typically work by maximizing the fitness function.

The "evaluate_fitness" function returns an array of fitness values, one for each individual in the population. These fitness values are then used by the genetic algorithm to select the best individuals for reproduction and mutation in the next generation.

Genetic Algorithm Operations: Selection, Mutation, and Crossover

```python
# Define a function to select the parents for the next generation
# Fitness Proportional Selection (Roulette Wheel) method is used
Furkan Salman
def select_parents(population, fitness_values, pop_size):
    fitness_values = np.exp(fitness_values - np.max(fitness_values))
    fitness_values = fitness_values / np.sum(fitness_values)  # deriving probability to be selected from fitness values
    selected_indices = np.random.choice(range(pop_size), size=pop_size, replace=True, p=fitness_values)
    return population[selected_indices]


# Define a function to apply the mutation operator to an individual
Furkan Salman
def mutate(individual, mutation_prob, mutation_strength):
    mutated_individual = individual.copy()
    for i in range(len(individual)):
        if np.random.rand() < mutation_prob:
            mutated_individual[i] += np.random.normal(0, mutation_strength)
    return mutated_individual


# Define a function to apply the crossover operator to a pair of parents
# alpha -> weight of parent affecting its children
Furkan Salman
def crossover(parent1, parent2):
    child = np.zeros(2)  # initialize children
    alpha = np.random.rand()  # get a random weight
    child[0] = alpha * parent1[0] + (1 - alpha) * parent2[0]
    child[1] = alpha * parent1[1] + (1 - alpha) * parent2[1]
    return child
```

We implemented three functions. The first one is called "select_parents"; this function selects parents for the next generation using the Roulette Wheel method. It assigns probabilities to each individual in the population based on their fitness values and then uses these probabilities to randomly select individuals for the reproduction of the next generation.

The second one is "mutate". This function applies random changes to an individual's genes to promote diversity in the population. It takes an individual, a mutation probability, and a mutation strength as input and applies mutations to each gene of the individual with a certain probability.

The last one is "crossover". This function combines the genes of two parents to create new individuals. It randomly selects a weight between 0 and 1, then uses this weight to create a new individual by taking a weighted average of each parent's genes.

Main function

```
def main():
    # Get the range of values from user from which the population will be initialized
    range_min = float(input("Enter the minimum value for the range (default: -10) : "))
    range_max = float(input("Enter the maximum value for the range (default: 10) : "))

    # Get the population size from user
    pop_size = int(input("Enter the population size: "))

    # Get the number of generations from user
    while True:
        num_generations = int(input("Enter the number of generations (must be a positive integer): "))
        if num_generations > 0:
            break
        print("Invalid input. Please enter a positive integer.")

    # Get the mutation probability from user and limit it to be between 0 and 1
    while True:
        mutation_prob = float(input("Enter the mutation probability (between 0 and 1): "))
        if 0 <= mutation_prob <= 1:
            break
        print("Invalid input. Please enter a value between 0 and 1.")

    # Get the mutation strength from user
    mutation_strength = float(input("Enter the mutation strength: "))

    # Create a list to store the best fitness value of each generation
    best_fitness_values = []

    # Get the population
    population = populate(range_min, range_max, pop_size)
```

For the main function, we start by asking the user to enter the range of values, population size, number of generations, mutation probability, and mutation strength for the optimization process. Then, a list is created to store the best fitness value of each generation. The optimization process begins with a loop that runs for the specified number of generations. The fitness of each individual in the population is evaluated using the "evaluate_fitness" function and the best fitness value is stored.

```
    start_time = time.time()

    for i in range(num_generations):
        # Evaluate the fitness of the population
        fitness_values = evaluate_fitness(population)
        best_fitness_values.append(np.max(fitness_values))

        # Select the parents for the next generation
        parents = select_parents(population, fitness_values, pop_size)
```

```
        # Create the next generation
        next_generation = np.zeros((pop_size, 2))
        for j in range(pop_size):
            # Get parents from those selected before
            parent1 = parents[np.random.randint(pop_size)]
            parent2 = parents[np.random.randint(pop_size)]
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_prob, mutation_strength)
            next_generation[j] = child

        # Replace the current generation with the next generation
        population = next_generation

    # End of the optimization process
    end_time = time.time()
```

Here, we select parents for the next generation using the select_parents function, which selects the parents based on their fitness values.

Then, we create the next generation using the parents selected in the previous step. This is done by performing crossover and mutation operations on the parents to create new offspring.

Finally, we replace the current population with the new generation of offspring and restart the same operations for the specified number of generations.

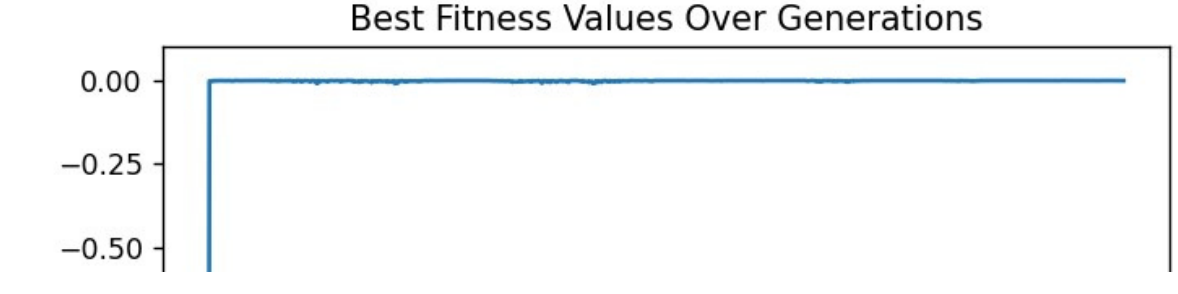
In the end, after the optimization process is complete, the best solution found and optimization time are printed. A plot is created to show the best fitness value of each generation.

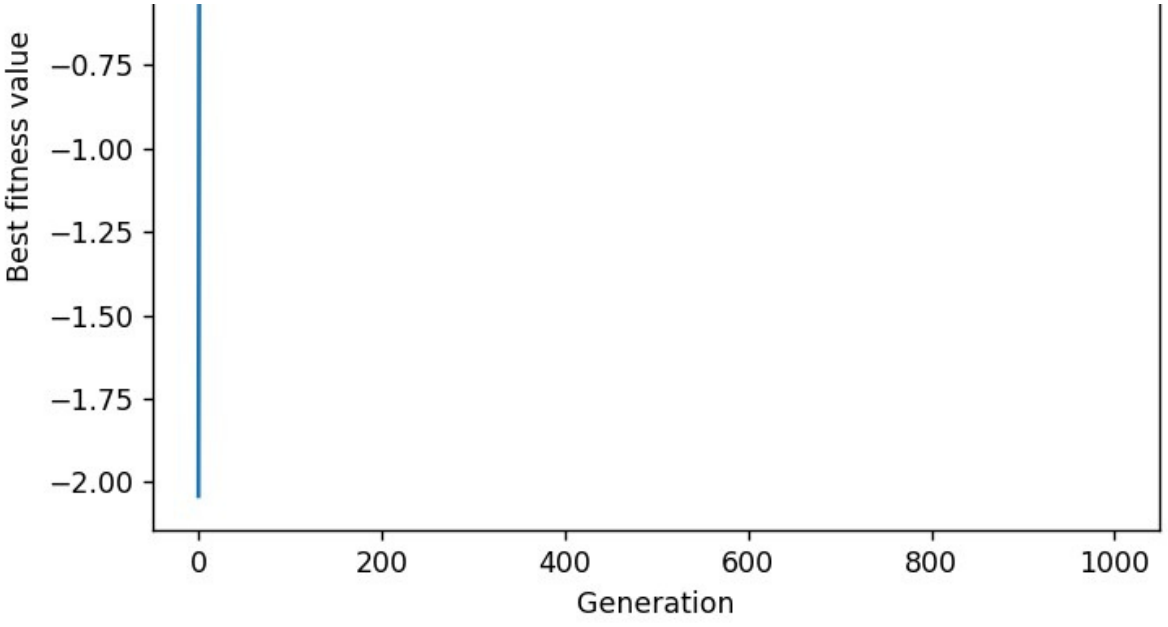
How Parameter Selection Affects Optimization:

Range:

Example of a narrow range with range = (-10,10) :

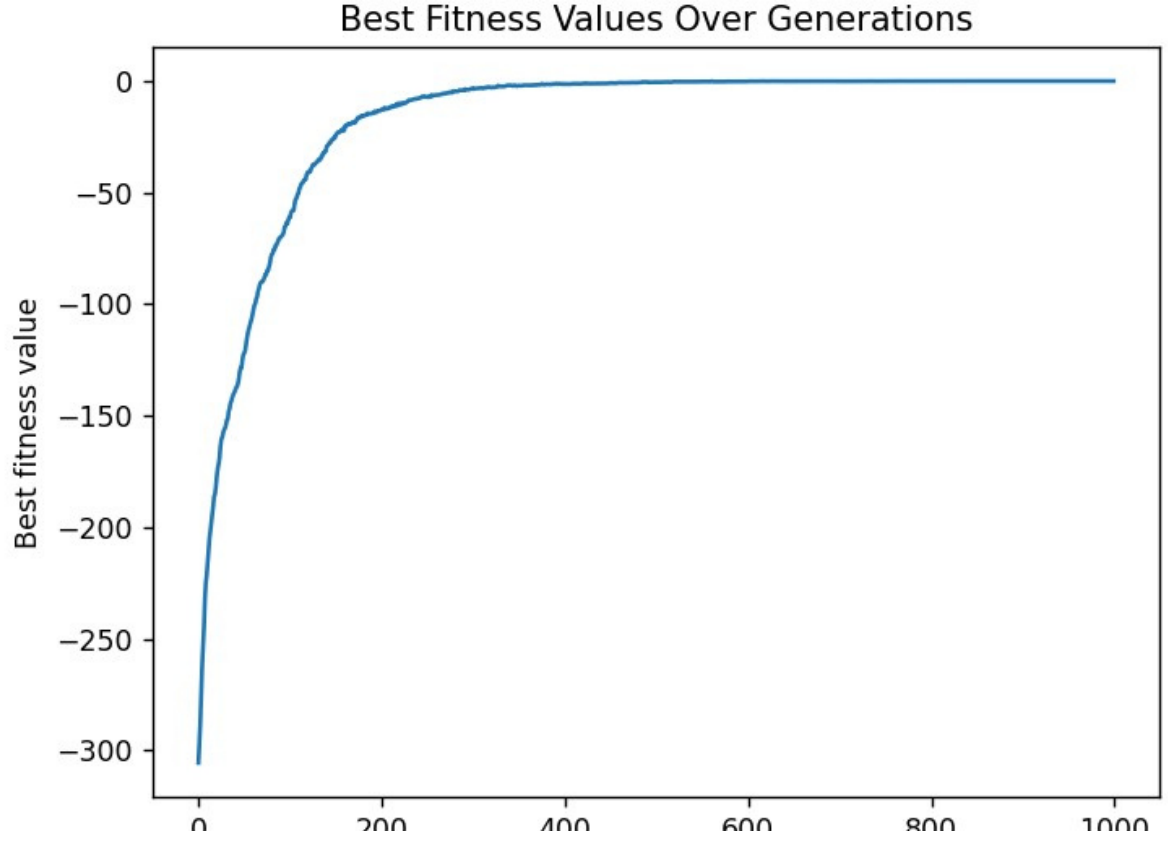
```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 100
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 1.0171353222633557, y = 2.99352606565929, f(x,y) = 0.00042360518159584846
Optimization time: 1.50 seconds
```





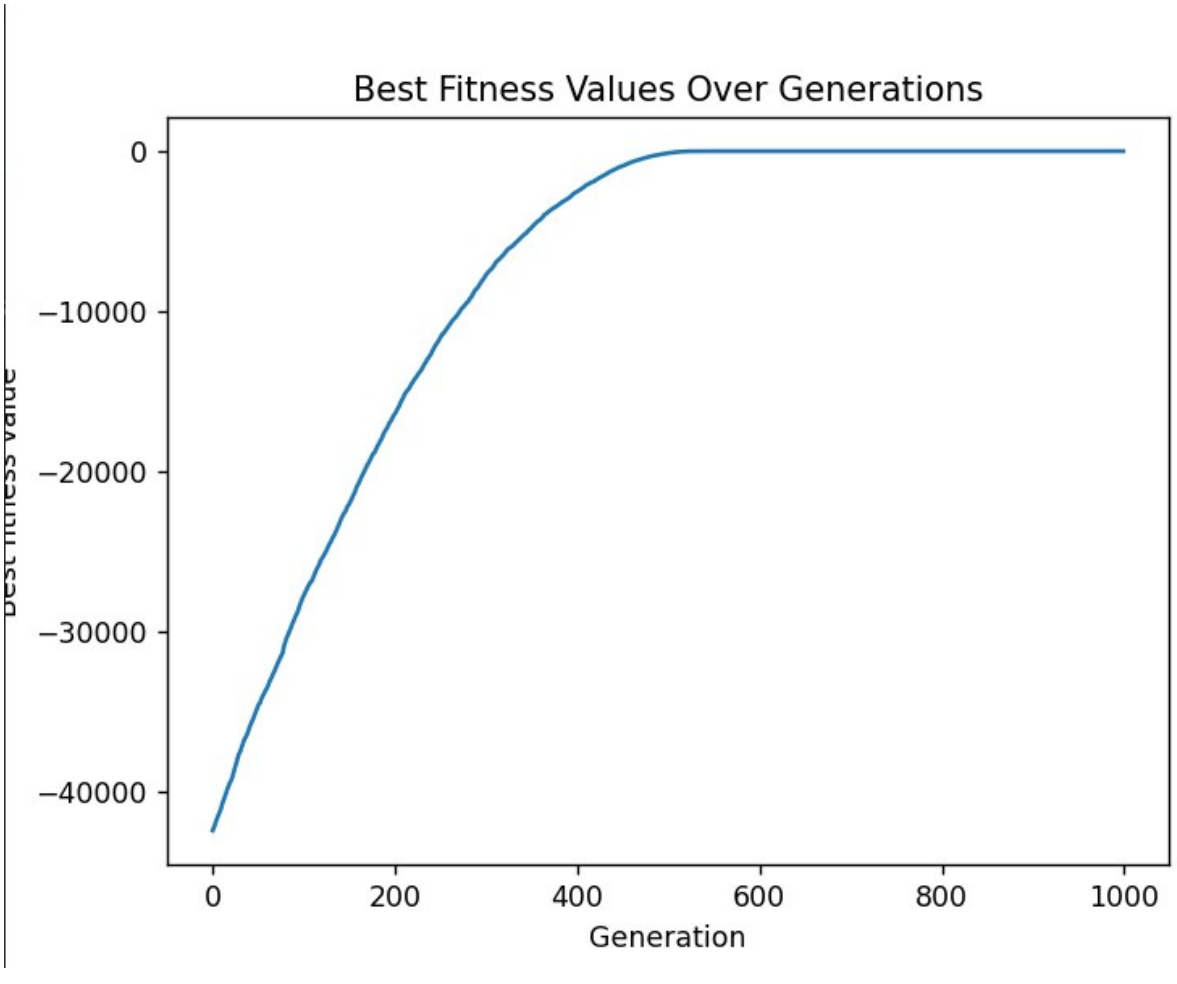
Example of a broader range with range = (-100,100) :

```
Enter the minimum value for the range (default: -10) : -100
Enter the maximum value for the range (default: 10) : 100
Enter the population size: 100
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 1.0248593640293675, y = 3.1581964975165726, f(x,y) = 7.924998285915181e-06
Optimization time: 1.42 seconds
```



Example of a range that doesn't include the solution (50,60) :

```
Enter the minimum value for the range (default: -10) : 50
Enter the maximum value for the range (default: 10) : 60
Enter the population size: 100
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 1.0341371625885096, y = 2.8769472454894767, f(x,y) = 0.00250039491738683
Optimization time: 1.61 seconds
```



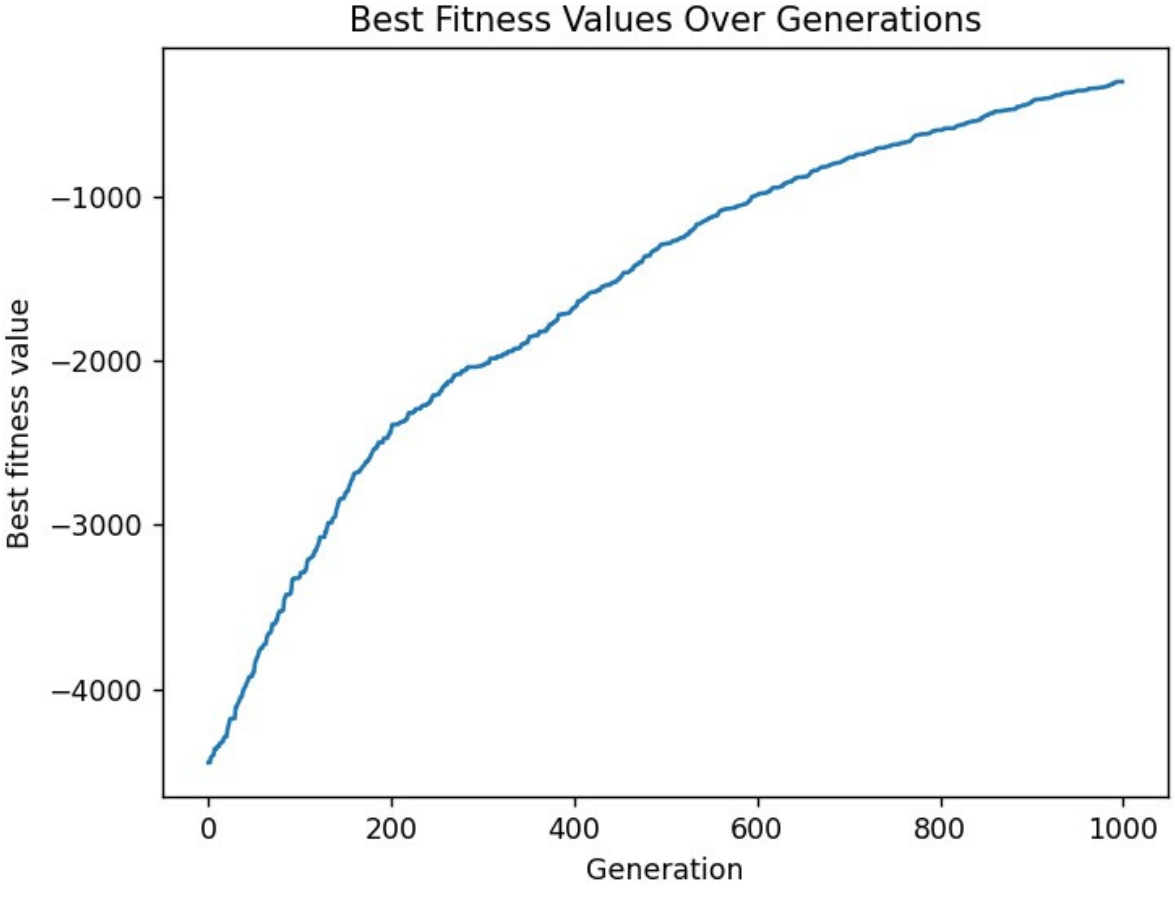
When the range is broad, it takes more time to converge compared to a narrower range. However, if our solution is not included in the range we selected, then a broader range helps us explore more space, which may lead us to the solution. As we can see above, a narrower range that includes or is closer to the solution is the best option.

Population Size:

Example of a small population size = 10:

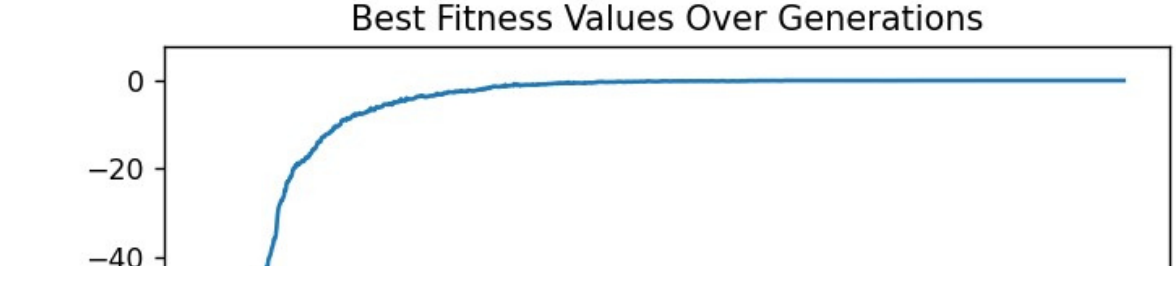
```
Enter the minimum value for the range (default: -10) : -100
Enter the maximum value for the range (default: 10) : 100
Enter the population size: 10
```

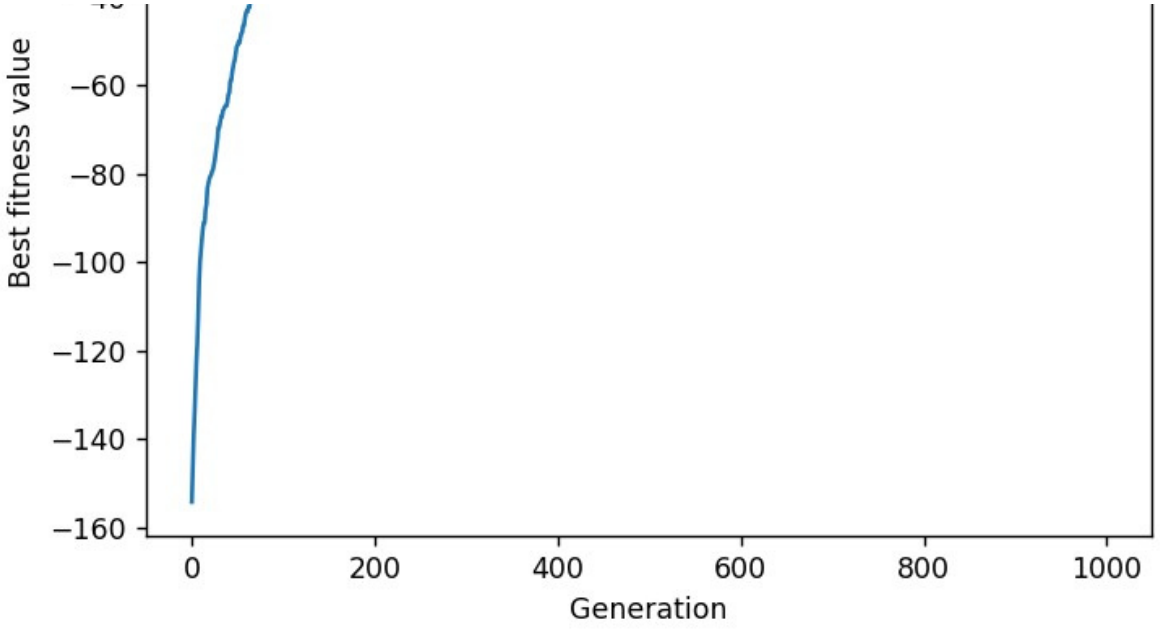
```
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 12.706605131603796, y = -9.760752524222168, f(x,y) = 304.2901845096794
Optimization time: 0.23 seconds
```



Example of a bigger population size = 100:

```
Enter the minimum value for the range (default: -10) : -100
Enter the maximum value for the range (default: 10) : 100
Enter the population size: 100
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 1.0382269834527937, y = 2.99684547562424, f(x,y) = 0.0004976268781364542
Optimization time: 1.65 seconds
```



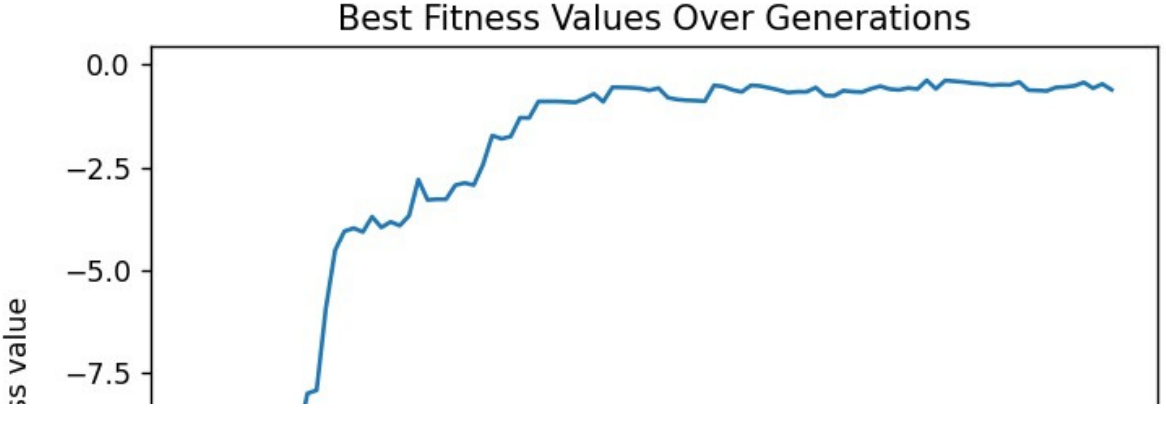


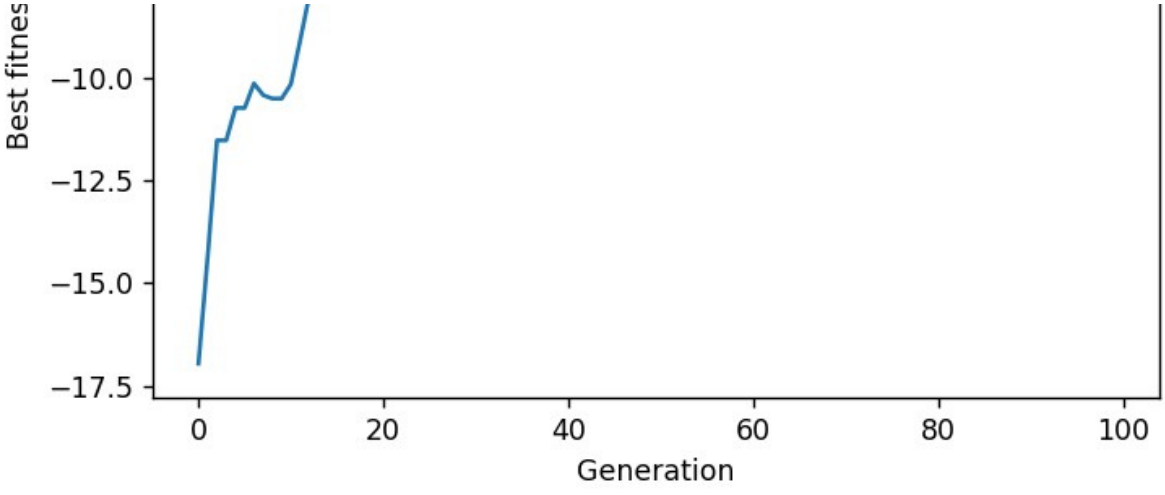
The examples above demonstrate that a population size of 100 almost converged in 200 iterations, whereas a population size of 50 did not converge to the solution even after 1000 iterations. Therefore, having a larger population helps the algorithm converge towards the solution by enabling each individual to explore and discover its environment, thereby producing better individuals. However, this advantage comes at the cost of increased optimization time. For instance, for a population of 10 individuals, 1000 iterations took 0.23 seconds, whereas for a population of 100 individuals, it took 1.65 seconds. This could be a significant issue when dealing with larger scale problems.

Number of Generations:

Example of a small number of generations = 100:

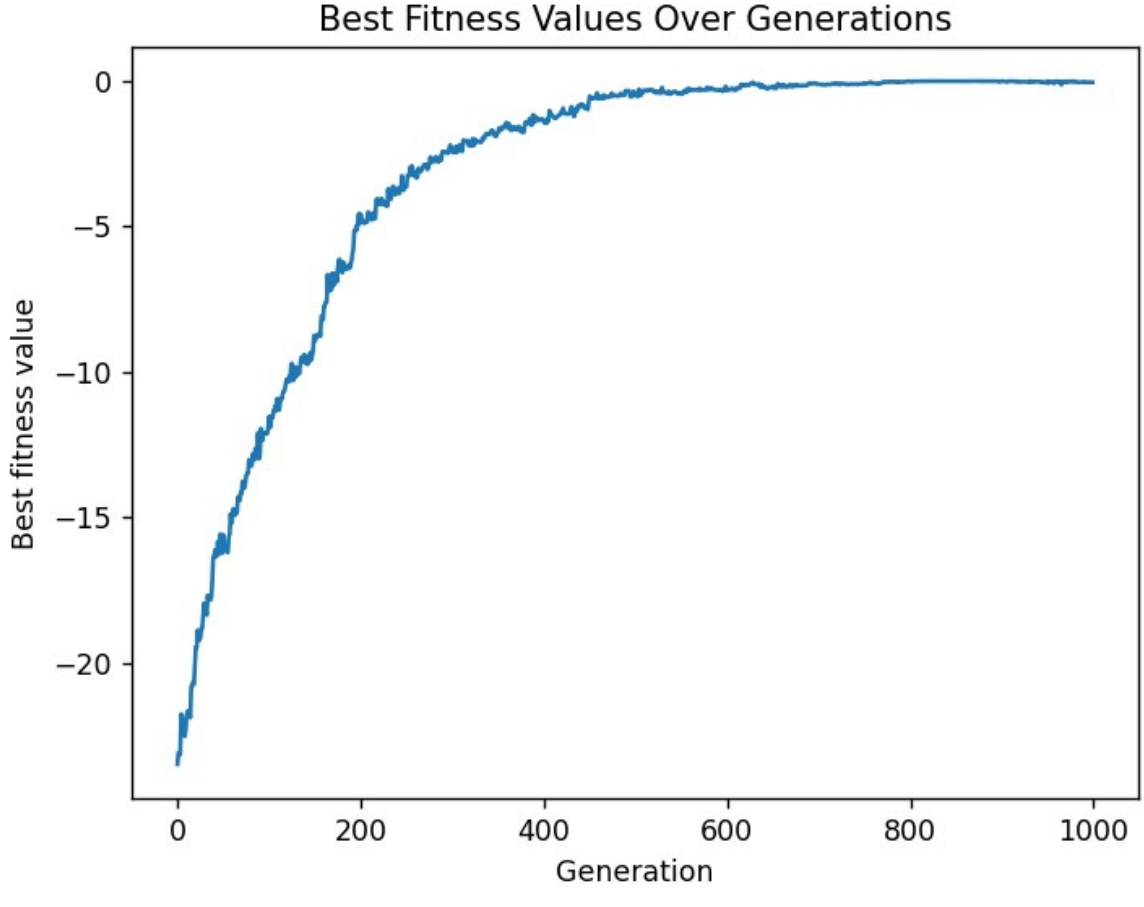
```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 20
Enter the number of generations (must be a positive integer): 100
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 0.39205188420431636, y = 3.6252062154839897, f(x,y) = 0.6205755051479469
Optimization time: 0.04 seconds
```





Example of bigger number of generations = 1000:

```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 20
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 1.1450996402368359, y = 2.8044089701575823, f(x,y) = 0.04218763277804337
Optimization time: 0.38 seconds
```

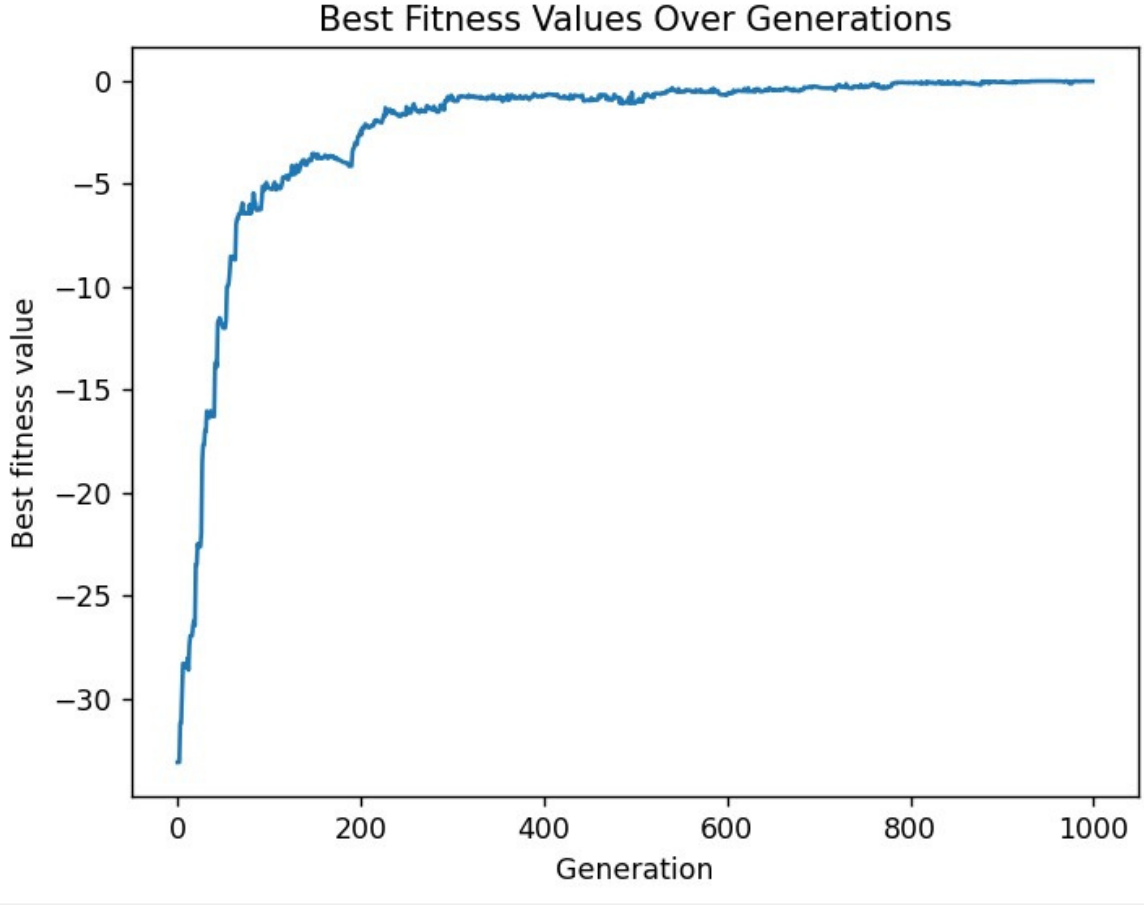


As you can see above we should also select a good number of generations to be created so the algorithm will be able to converge. In the first example even though it starts to converge since the number of generations is limited, it couldn't converge to the solution perfectly whereas in the second example where it has enough generations it was able to converge better to the solution but here also we have a trade-off between time and convergence because with more generations it will take more time to converge to the solution. To prevent this time-consuming process we can set a threshold to check if our algorithm has converged and break the loop.

Mutation Probability:

Example to low mutation probability p = 0.1:

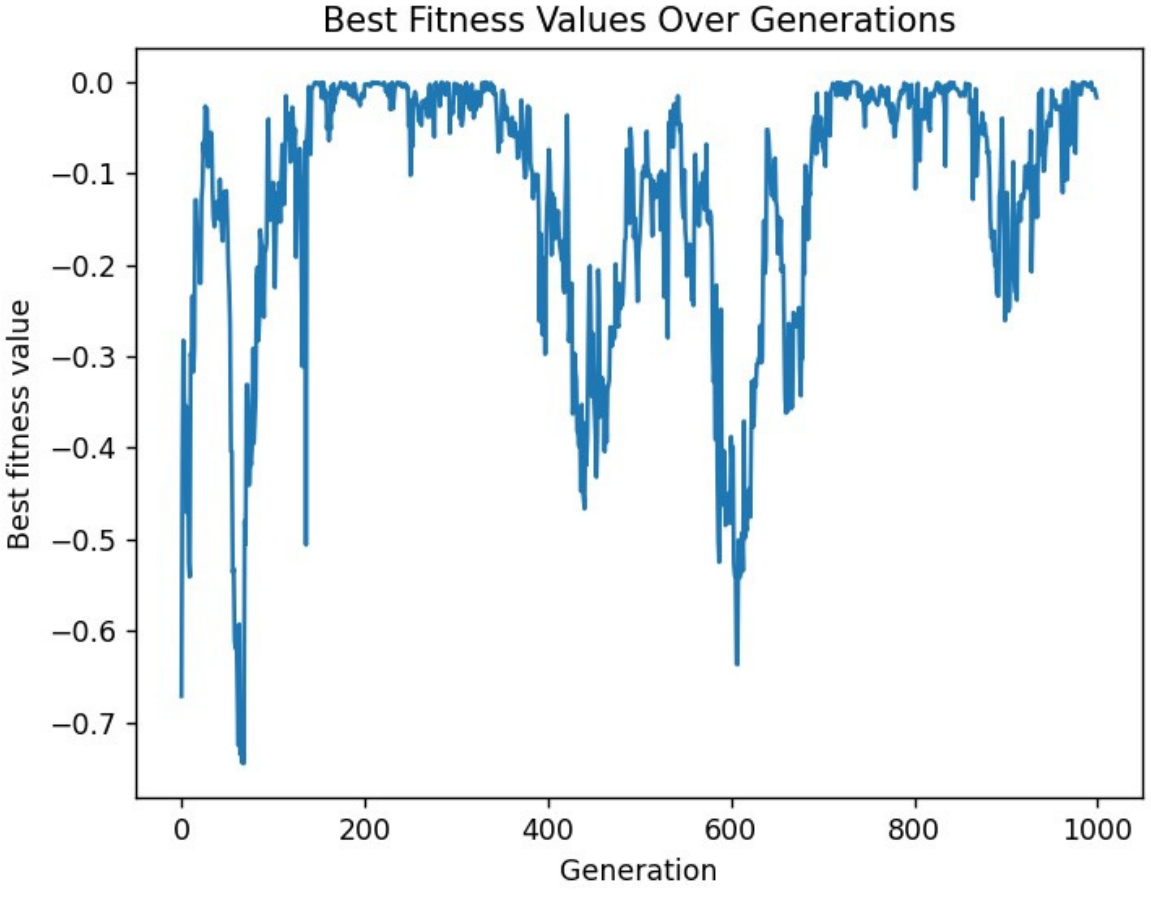
```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 10
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
Best solution found: x = 0.8978797744004183, y = 3.0589222369356817, f(x,y) = 0.008474811304134243
Optimization time: 0.24 seconds
```



Example to higher mutation probability p= 0.3:

```
Enter the minimum value for the range (default: -10) : -10
```

```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 10
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.3
Enter the mutation strength: 0.1
Best solution found: x = 0.70567228902357, y = 3.0865219942210556, f(x,y) = 0.016698061494222812
Optimization time: 0.23 seconds
```



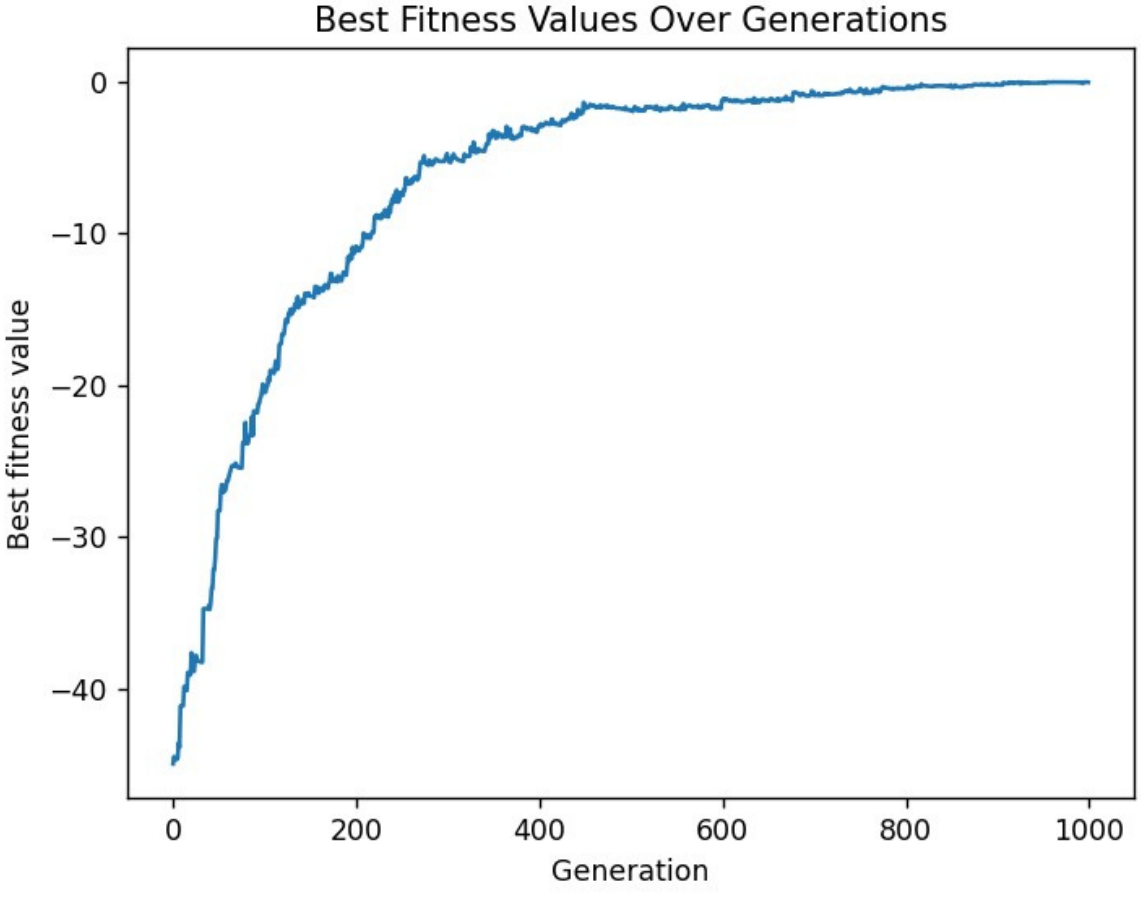
If the mutation probability is too low, the algorithm may converge prematurely, as it will be unable to explore alternative solutions. On the other hand, if the mutation probability is too high, the algorithm may become too random, and it will take longer to converge towards the optimal solution. Therefore, it is essential to choose an appropriate mutation probability that balances exploration and exploitation to ensure the algorithm converges to the optimal solution in a reasonable amount of time.

Mutation Strength

Example to low mutation strength = 0.1:

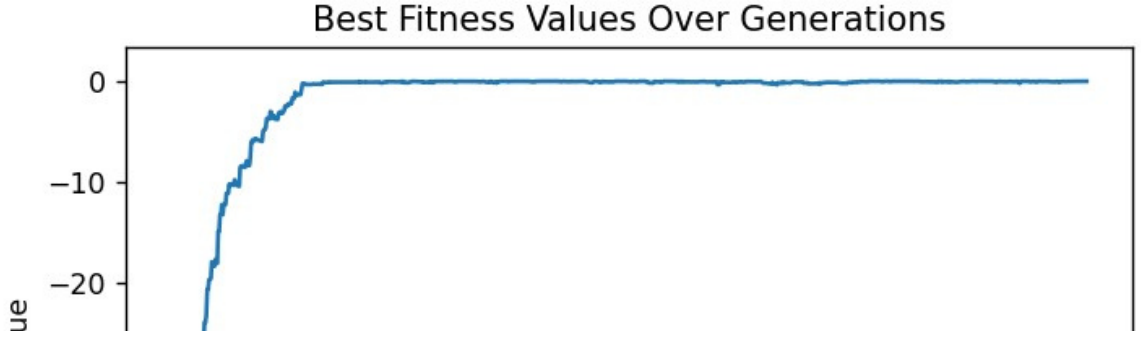
```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 10
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.1
```

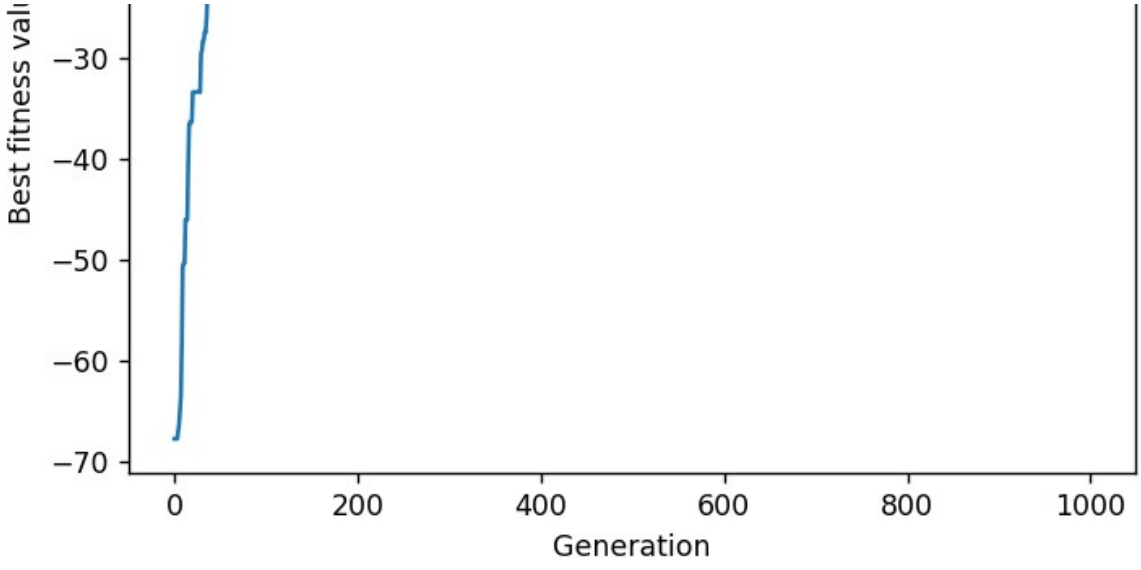
```
Best solution found: x = 0.895187899189543, y = 3.05052141283168, f(x,y) = 0.01969428726389692
Optimization time: 0.25 seconds
```



Example to higher mutation strength = 0.3:

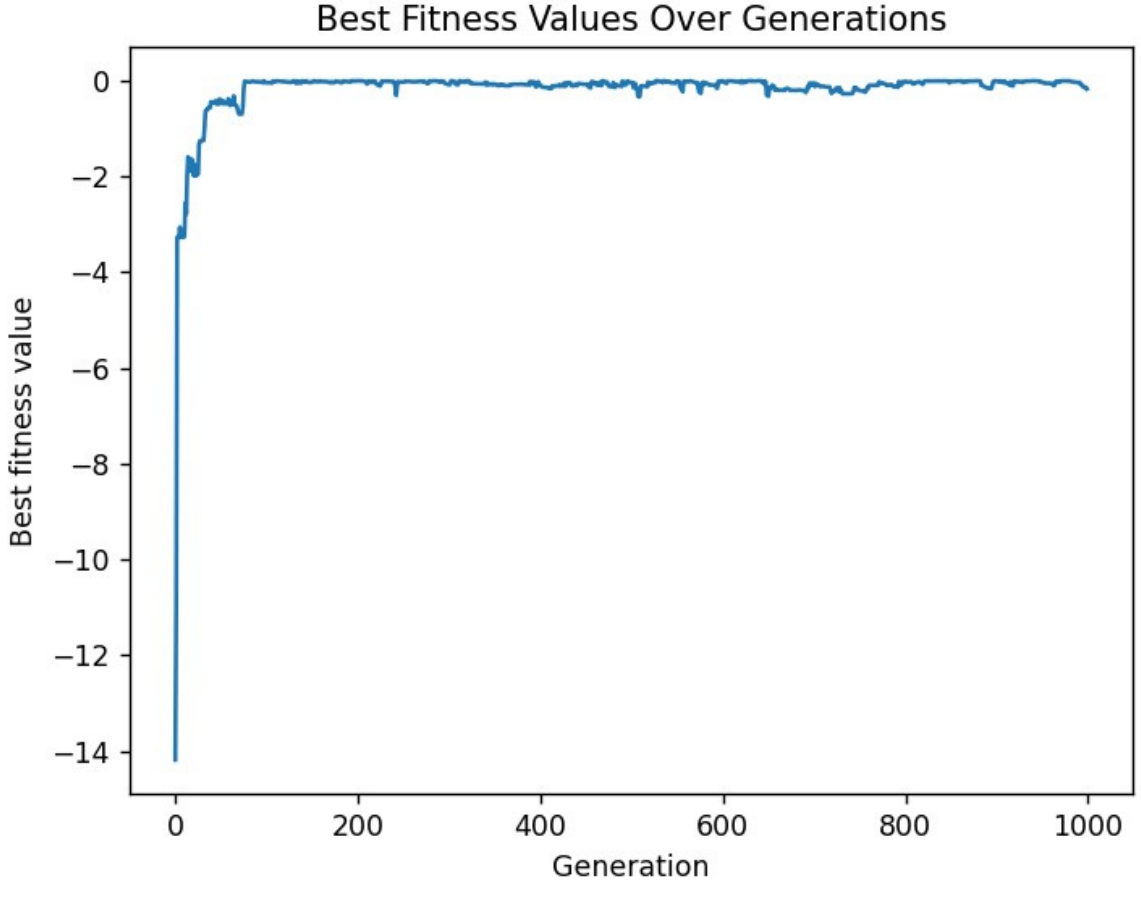
```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 10
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.3
Best solution found: x = 0.9439959395410016, y = 3.0342902308430824, f(x,y) = 0.0017432960628213116
Optimization time: 0.23 seconds
```





Example to unnecessarily high mutation strength = 0.7:

```
Enter the minimum value for the range (default: -10) : -10
Enter the maximum value for the range (default: 10) : 10
Enter the population size: 10
Enter the number of generations (must be a positive integer): 1000
Enter the mutation probability (between 0 and 1): 0.1
Enter the mutation strength: 0.7
Best solution found: x = 0.9729045005761401, y = 3.2366531131700746, f(x,y) = 0.16840559850714262
Optimization time: 0.24 seconds
```



Increasing mutation strength enables the algorithm to explore a wider range of solutions, which can result in faster convergence, as shown in the plots. However, this comes at the cost of always missing the best solution because the mutation is too powerful. This means that even if we explore the correct direction, the high mutation strength can lead to a solution that is more distant from the optimal solution. As a result, in the third example, even though it converges very quickly, it struggles to converge perfectly to the optimum, resulting in an outcome of 0.168, whereas the second example had 0.0017 and the first example had 0.019. Therefore, we should select a balanced mutation strength in order to optimize both speed and the quality of the solution.