Laboratory 3

Variant 3, Group 13

By Saumya Shah and Anas Tagui

Introduction

In this lab, we explore the application of a genetic algorithm to optimise a complex two-dimensional function: the **Bukin Function.**

The Bukin function is well-known for its steep valleys and challenging surface, making it an ideal candidate to test the convergence behavior of evolutionary algorithms.

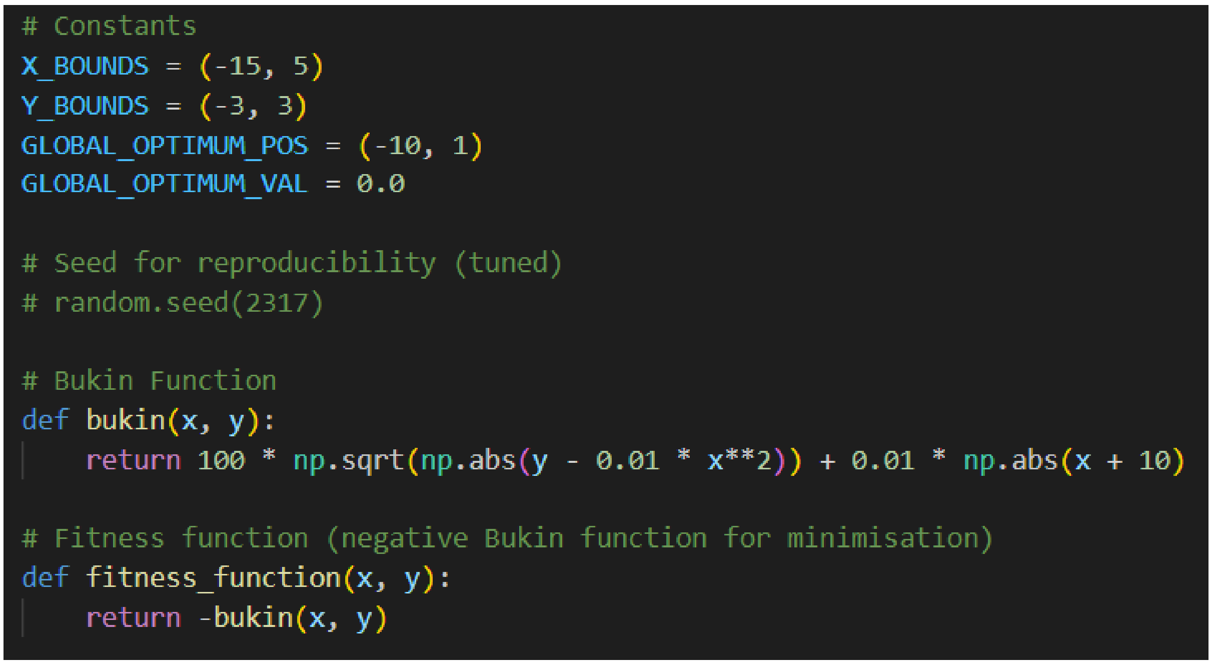
We implemented a genetic algorithm using **tournament selection**, **Gaussian mutation**, and **random interpolated crossover** to minimise the Bukin function over the defined domain $x \in [-15,5]$, $y \in [-3,3]$. The fitness function is defined as the negative of the Bukin value, allowing the algorithm to maximise fitness by minimising the original function.

The aim of the lab is to:

1. Design and implement a functional genetic algorithm.
2. Tune hyperparameters for optimal performance.
3. Analyse the effects of randomness, crossover rate, and mutation parameters on convergence and solution quality.
4. Assess how close the algorithm can get to the **global minimum** of the Bukin function, known to lie at $(-10,1)$.

Implementation

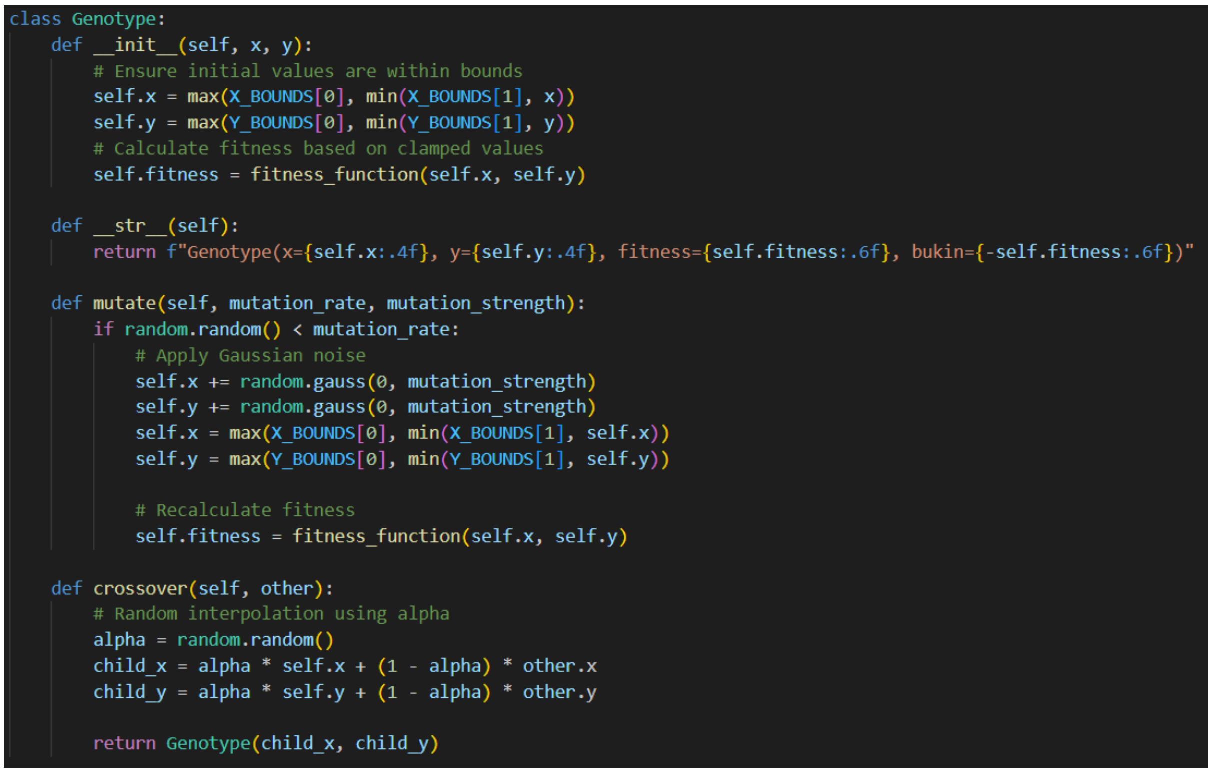
Constants and Functions



```python
# Constants
X_BOUNDS = (-15, 5)
Y_BOUNDS = (-3, 3)
GLOBAL_OPTIMUM_POS = (-10, 1)
GLOBAL_OPTIMUM_VAL = 0.0

# Seed for reproducibility (tuned)
# random.seed(2317)

# Bukin Function
def bukin(x, y):
    return 100 * np.sqrt(np.abs(y - 0.01 * x**2)) + 0.01 * np.abs(x + 10)

# Fitness function (negative Bukin function for minimisation)
def fitness_function(x, y):
    return -bukin(x, y)
```

Defines some global constants, such as the x and y bounds, and the fitness function.
The fitness function is defined as the negative of the Bukin function so that a higher fitness corresponds to our objective of minimising the Bukin function.

Genotype Class:



```python
class Genotype:
    def __init__(self, x, y):
        # Ensure initial values are within bounds
        self.x = max(X_BOUNDS[0], min(X_BOUNDS[1], x))
        self.y = max(Y_BOUNDS[0], min(Y_BOUNDS[1], y))
        # Calculate fitness based on clamped values
        self.fitness = fitness_function(self.x, self.y)

    def __str__(self):
        return f"Genotype(x={self.x:.4f}, y={self.y:.4f}, fitness={self.fitness:.6f}, bukin={-self.fitness:.6f})"

    def mutate(self, mutation_rate, mutation_strength):
        if random.random() < mutation_rate:
            # Apply Gaussian noise
            self.x += random.gauss(0, mutation_strength)
            self.y += random.gauss(0, mutation_strength)
            self.x = max(X_BOUNDS[0], min(X_BOUNDS[1], self.x))
            self.y = max(Y_BOUNDS[0], min(Y_BOUNDS[1], self.y))

            # Recalculate fitness
            self.fitness = fitness_function(self.x, self.y)

    def crossover(self, other):
        # Random interpolation using alpha
        alpha = random.random()
        child_x = alpha * self.x + (1 - alpha) * other.x
        child_y = alpha * self.y + (1 - alpha) * other.y

        return Genotype(child_x, child_y)
```

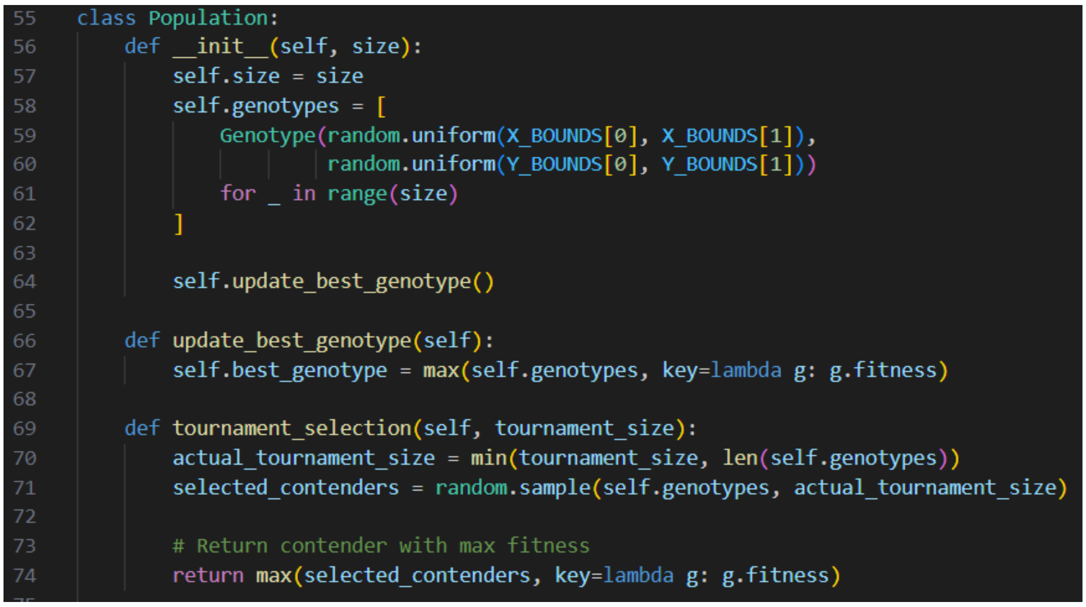
Represents a genotype in the population with x and y values.

The constructor function takes in x and y values, clips them to the given range, and calculates and stores its fitness.

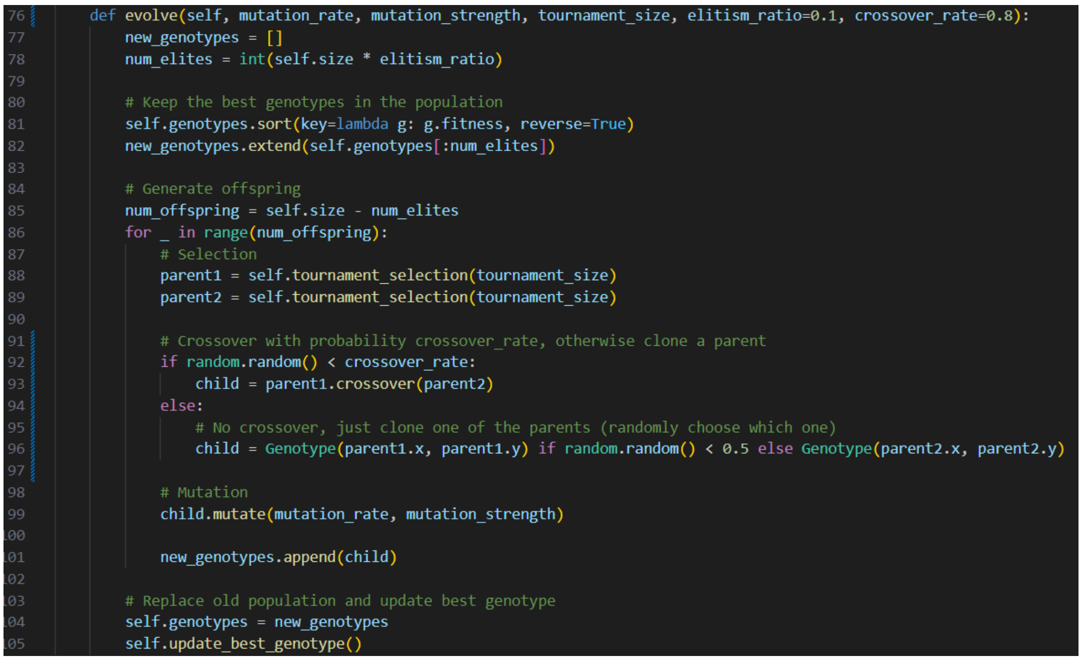
The `mutate` function takes in a mutation rate and strength, and adds Gaussian noise using the `mutation_strength` as the standard deviation for the Gaussian. Then, the values are clipped to the given range, and the fitness is recalculated.

The `crossover` function takes in another genotype, and uses random interpolation with the random alpha parameter. It returns a new child genotype.

Population Class:



```python
class Population:
    def __init__(self, size):
        self.size = size
        self.genotypes = [
            Genotype(random.uniform(X_BOUNDS[0], X_BOUNDS[1]),
                     random.uniform(Y_BOUNDS[0], Y_BOUNDS[1]))
            for _ in range(size)
        ]

        self.update_best_genotype()

    def update_best_genotype(self):
        self.best_genotype = max(self.genotypes, key=lambda g: g.fitness)

    def tournament_selection(self, tournament_size):
        actual_tournament_size = min(tournament_size, len(self.genotypes))
        selected_contenders = random.sample(self.genotypes, actual_tournament_size)

        # Return contender with max fitness
        return max(selected_contenders, key=lambda g: g.fitness)
```



```python
    def evolve(self, mutation_rate, mutation_strength, tournament_size, elitism_ratio=0.1, crossover_rate=0.8):
        new_genotypes = []
        num_elites = int(self.size * elitism_ratio)

        # Keep the best genotypes in the population
        self.genotypes.sort(key=lambda g: g.fitness, reverse=True)
        new_genotypes.extend(self.genotypes[:num_elites])

        # Generate offspring
        num_offspring = self.size - num_elites
        for _ in range(num_offspring):
            # Selection
            parent1 = self.tournament_selection(tournament_size)
            parent2 = self.tournament_selection(tournament_size)

            # Crossover with probability crossover_rate, otherwise clone a parent
            if random.random() < crossover_rate:
                child = parent1.crossover(parent2)
            else:
                # No crossover, just clone one of the parents (randomly choose which one)
                child = Genotype(parent1.x, parent1.y) if random.random() < 0.5 else Genotype(parent2.x, parent2.y)

            # Mutation
            child.mutate(mutation_rate, mutation_strength)

            new_genotypes.append(child)

        # Replace old population and update best genotype
        self.genotypes = new_genotypes
        self.update_best_genotype()
```

Represents the population for the genetic algorithm.

The constructor takes in a population size, and creates a list of genotypes of that size with x and y values randomly distributed over the given range. Then, it stores the best genotype by fitness using the `update_best_genotype()` function.

The `tournament_selection` function takes in a tournament size, creates a random sample of the given size from the population, and returns the contender with the maximum fitness.

The `evolve` function takes in all of the genetic algorithm parameters, and creates a list to store the new genotypes. Then, it adds the elites (genotypes with best fitness value) to the `new_genotypes` list according to the elitism ratio. Then, for each offspring to be created, it chooses two parents from the current population using `tournament_selection`, performs crossover randomly, mutates the child randomly, and adds it to the `new_genotypes` list. At the end, it updates the current genotypes with the new genotypes, and updates the best genotype.

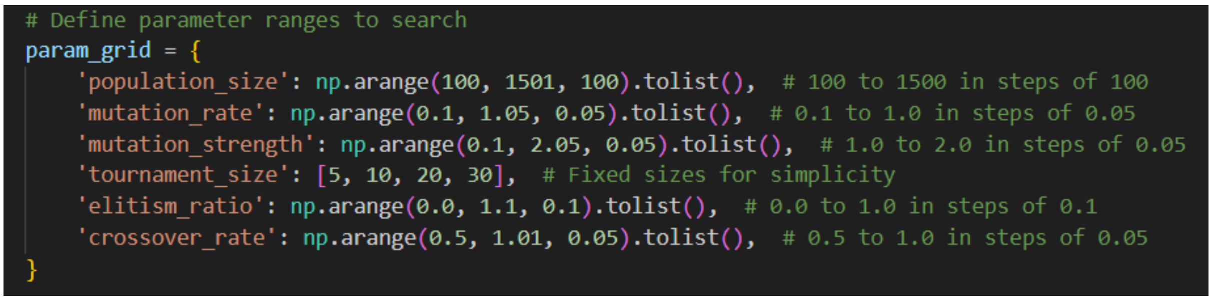
Main Function:

The main function runs `evolve` on the population for the given number of generations, and stores, prints, and plots values such as best Bukin value.

Discussion

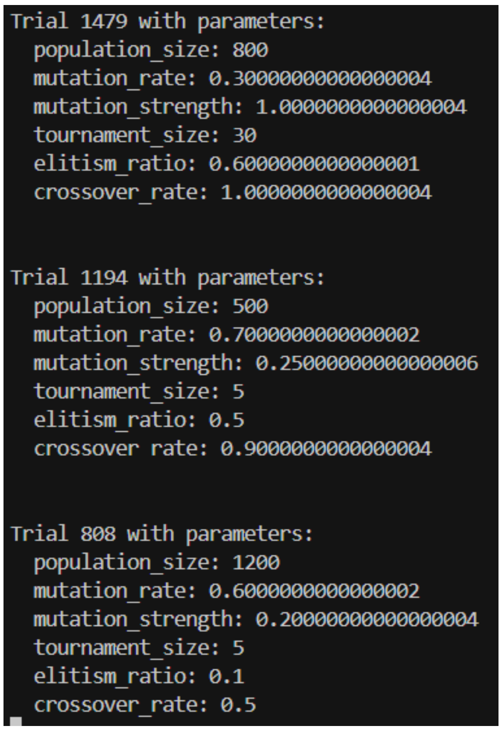
Finding Genetic Algorithm Parameters

To find suitable parameters for the genetic algorithm, random search was employed. First, the random seed was fixed and a suitable range of parameters was identified.



```python
# Define parameter ranges to search
param_grid = {
    'population_size': np.arange(100, 1501, 100).tolist(),  # 100 to 1500 in steps of 100
    'mutation_rate': np.arange(0.1, 1.05, 0.05).tolist(),  # 0.1 to 1.0 in steps of 0.05
    'mutation_strength': np.arange(0.1, 2.05, 0.05).tolist(),  # 1.0 to 2.0 in steps of 0.05
    'tournament_size': [5, 10, 20, 30],  # Fixed sizes for simplicity
    'elitism_ratio': np.arange(0.0, 1.1, 0.1).tolist(),  # 0.0 to 1.0 in steps of 0.1
    'crossover_rate': np.arange(0.5, 1.01, 0.05).tolist(),  # 0.5 to 1.0 in steps of 0.05
}
```

We noticed that the parameter space is too large for grid search (it has more than 500,000 possible combinations). Therefore, we opted to use random search. The number of trials run for this random search is too large to enumerate all tested parameters in a table (3000 possible combinations of parameters were tested in parallel using multiprocessing), so the table has been excluded. As an example, here is a screenshot from the terminal during the random search:

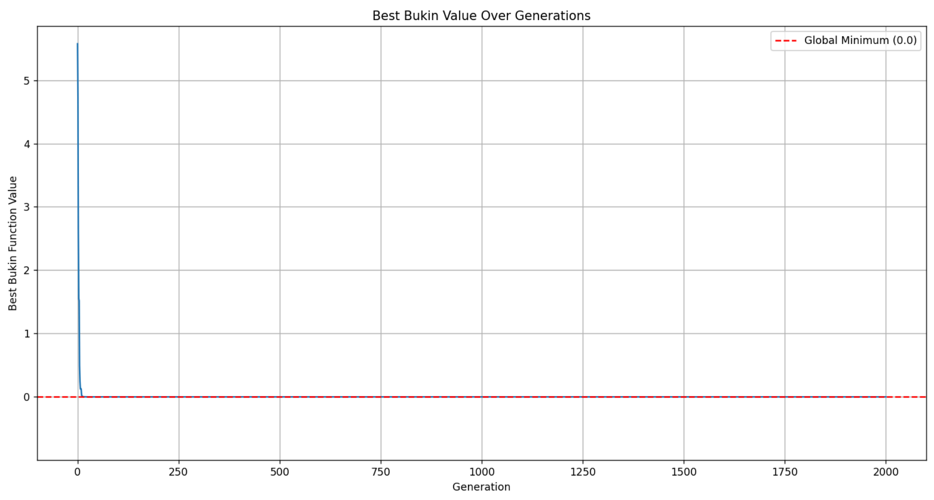


```
Trial 1479 with parameters:
  population_size: 800
  mutation_rate: 0.30000000000000004
  mutation_strength: 1.0000000000000004
  tournament_size: 30
  elitism_ratio: 0.6000000000000001
  crossover_rate: 1.0000000000000004


Trial 1194 with parameters:
  population_size: 500
  mutation_rate: 0.7000000000000002
  mutation_strength: 0.25000000000000006
  tournament_size: 5
  elitism_ratio: 0.5
  crossover rate: 0.9000000000000004


Trial 808 with parameters:
  population_size: 1200
  mutation_rate: 0.6000000000000002
  mutation_strength: 0.20000000000000004
  tournament_size: 5
  elitism_ratio: 0.1
  crossover_rate: 0.5
```

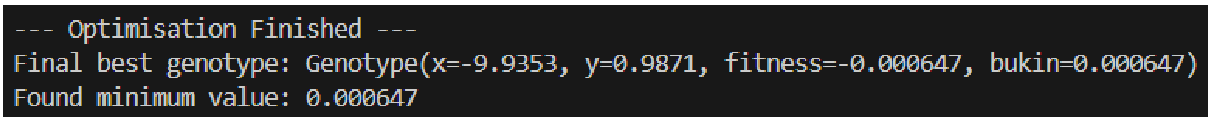
Then, the best combination was identified by comparing the fitness of the best genotype of the trial, and these parameters were tuned manually till a solution with maximum fitness was found. Then, a genetic algorithm with these hyperparameters was allowed to evolve for 2000 generations.

The tuned parameters are as follows:

```
generations = 2000
population_size = 500
mutation_rate = 0.5
mutation_strength = 1.00251
tournament_size = 10
elitism_ratio = 0.2
crossover_ratio = 1.0
```

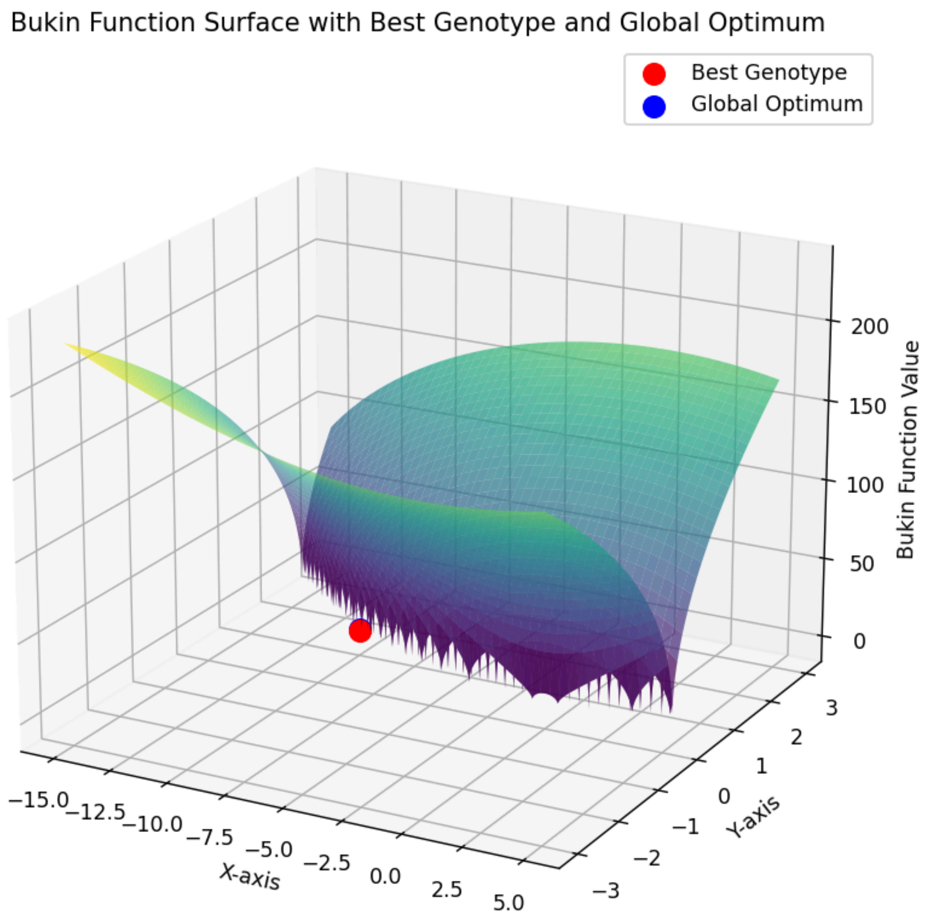


The above combination gives us `Bukin(-9.9353, 0.9871) = 0.000647.`



```
--- Optimisation Finished ---
Final best genotype: Genotype(x=-9.9353, y=0.9871, fitness=-0.000647, bukin=0.000647)
Found minimum value: 0.000647
```

We notice from the above graph that the best fitness value of the sample converges to a value very close to 0 in relatively few generations. From the graph below, we also observe that it is extremely close to the known global minimum at `(-10, 1)`.



Randomness in Genetic Algorithm

Seed Value	**Best Bukin Value**
42	0.000647
420	0.500801
2137	0.252882
314159	0.629669
271828	0.568960

Average: 0.390592

Standard Deviation: 0.233298

Different Population Sizes

% of Population	**Best Bukin Value**
100	0.000647
50	0.589342
25	0.870622
10	1.282621

We see that the best value obtained gets worse as we decrease the population size.

Crossover Impact

Bukin values below are averaged across the seeds (`1`, `20`, `300`)

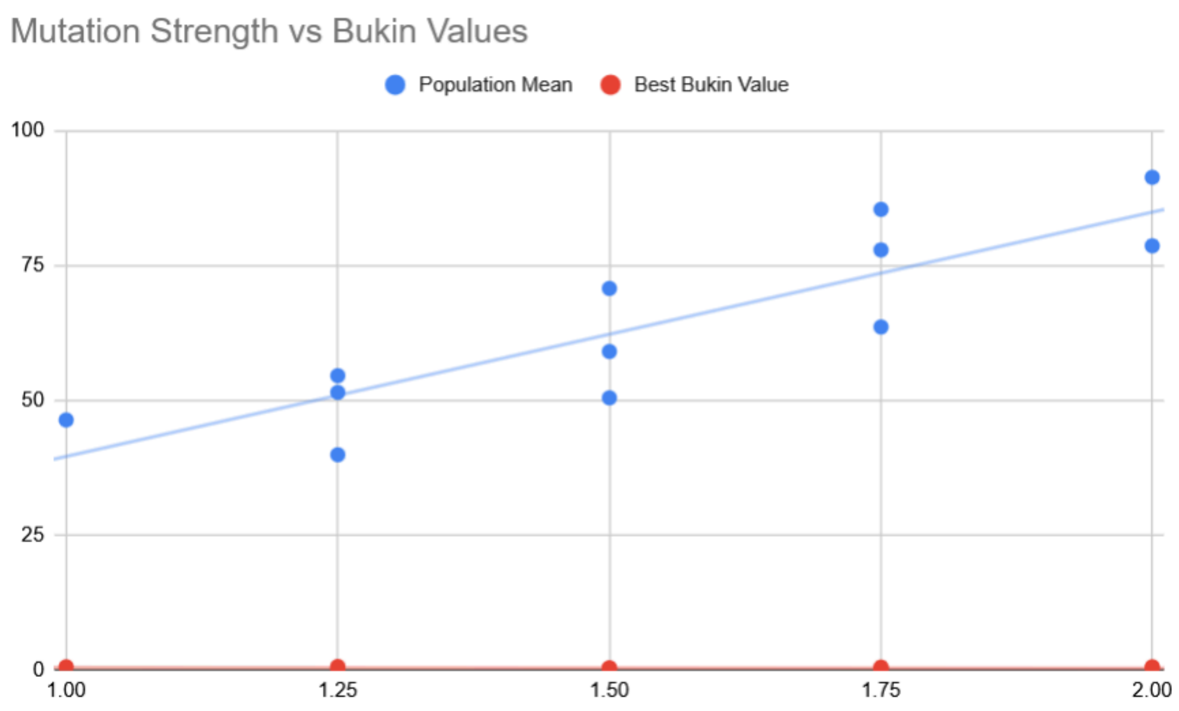
Crossover Rate	**Bukin Value**
1	0.312483
0.75	0.568960
0.5	0.572837
0.25	0.723314
0.1	1.081103

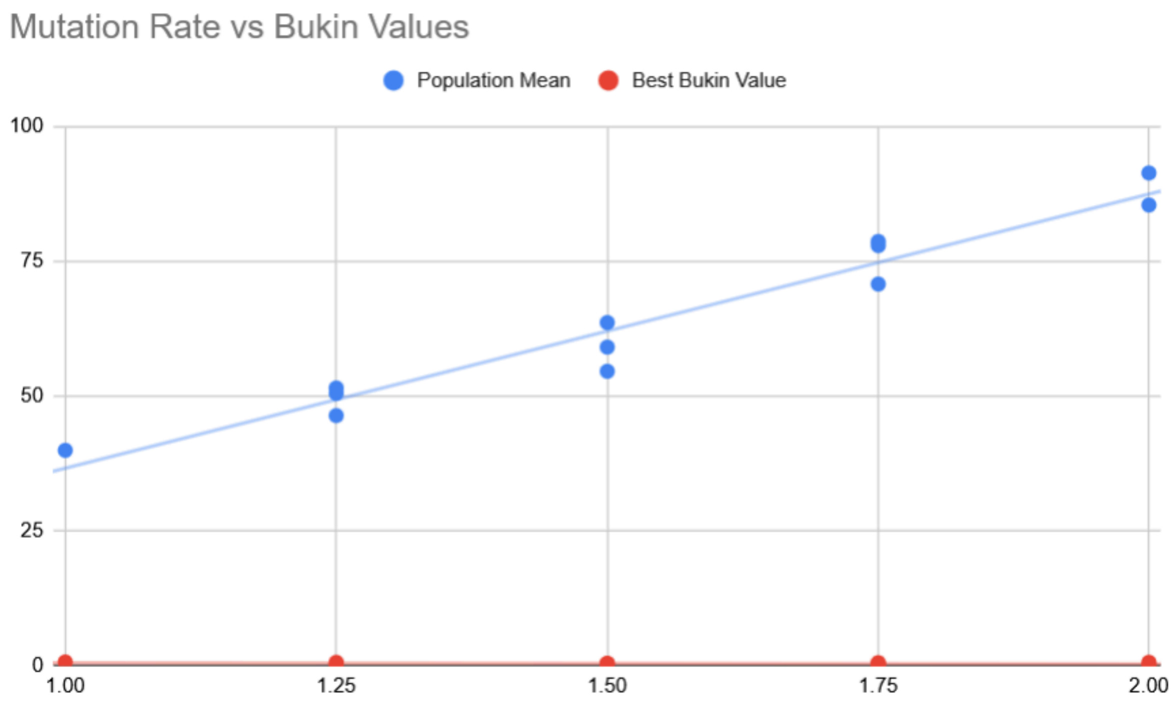
We see that the average best value obtained gets worse as we decrease the crossover rate.

Mutation & Convergence

Bukin values below are averaged across the seeds (`1`, `20`, `300`)
Mutation strength and mutation rate are represented as multipliers of the initial rates.

Mutation Strength	**Mutation Rate**	**Population Mean**	**Best Bukin Value**
1.25	1	39.894059	0.703810
1	1.25	46.357971	0.595319
1.25	1.25	51.483338	0.240602
1.25	1.5	54.597735	0.390527
1.5	1.25	50.459071	0.308425
1.5	1.5	59.049728	0.416091
1.5	1.75	70.743453	0.230561
1.75	1.5	63.644159	0.283488
1.75	1.75	77.918655	0.532125
1.75	2.0	85.434141	0.133825
2.0	1.75	78.632542	0.328477
2.0	2.0	91.377589	0.589701





We notice from the two graphs above that as the mutation strength or rate are increased, the population Bukin value mean increases significantly. This suggests that higher mutation parameters introduce greater variability within the population, which may lead to some genotypes being mutated to suboptimal values.

We also notice that the best Bukin values do not change significantly. This may be due to the elitism retaining the best genotypes so that they don't change despite the greater mutation in the rest of the population.

Conclusion

Through a series of experiments and parameter tuning, we successfully demonstrated the effectiveness of genetic algorithms in minimising the Bukin function. The best solution found was remarkably close to the global minimum, with a function value of approximately **0.000647** at coordinates close to (−10,1).

We observed that:

- **Larger populations** and **higher crossover rates** improve convergence towards the global minimum.
- **Randomness in initial population** and **mutation** introduces necessary diversity, but excessive mutation may hinder convergence by disrupting well-adapted genotypes.
- **Elitism** helps preserve the best individuals, preventing loss of optimal traits during evolution.

Overall, the genetic algorithm proved to be a powerful technique for solving non-linear optimisation problems, and our implementation successfully captured key evolutionary principles while delivering accurate results.