LABORATORY 3
GROUP 5

By Uhma Pawel and Maurizio Andrés Carrasquero

INTRODUCTION

In this laboratory, we implemented a genetic algorithm to optimize the **Styblinski–Tang function**, a well-known test function in continuous optimization. It is characterized by a **non-convex landscape** and **multiple local minima**, making it a suitable benchmark for evaluating the performance of global optimization techniques.

The goal was to find the global minimum of the Styblinski–Tang function in two dimensions:

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

The known global minimum of the function is at (x,y)≈(−2.903,−2.903)(x, y), where the function value is approximately f(x,y)≈−78.332

We applied a genetic algorithm using **Rank Selection** to choose parents, **Gaussian mutation**, and **random interpolation-based crossover**. Importantly, mutation and crossover were applied only to a portion of the population, while the rest was preserved. The population was initialized with values x,y∈[−5,5]x, y \in [-5, 5]x,y∈[−5,5].

IMPLEMENTATION

The genetic algorithm was implemented in Python, using a modular structure in genetic_algorithm.py. The algorithm follows the classic structure: initialize → evaluate → select → crossover → mutate → replace.

Below is a breakdown of the key components with relevant code snippets and explanations.

1. **Population Initialization:** The population is initialized randomly within the specified range (in this case, [−5,5][-5, 5][−5,5] for both x and y):

```
population = [
    (random.uniform(*x_range), random.uniform(*y_range))
    for x in range(self.population_size)
]
```

This generates a list of individuals (2D vectors) with uniformly distributed random values.

2. **Evaluate the Population:** Each individual's fitness is determined by the function value (lower is better).

```
def evaluate_population(self, population):
    fitness = []
    for x in population:
        fitness.append(styblinski_tang_2d(*x))
    return fitness
```

3. **Selection:** We use fitness-proportional selection, where better individuals (lower values) get higher selection weights.

```
def selection(self, population, fitness_values):
    # sort indices of the population by fitness (lowest fitness is best)
    sorted_indices = np.argsort(fitness_values)
    max_weight = len(population)
    #creating array of length max weights
    weights = [0] * max_weight
    for rank, idx in enumerate(sorted_indices):
        weights[idx] = max_weight - rank
    # normalize
    weights = np.array(weights)
    probabilities = weights / np.sum(weights)
    # how many are selected for reproduction
    num_selected = int(self.population_size * self.crossover_rate)
    # randomly select
    selected_indices = np.random.choice(max_weight, size=num_selected, replace=True, p=probabilities)
    return [population[i] for i in selected_indices]
```

4. **Crossover:** Pair parents to create children by blending their coordinates using a random mixing factor α.

```
def crossover(self, parents):
    np.random.shuffle(parents)
    children = []
    num_parents = len(parents)
    # if there are odd parents, duplicate the random parent
    if num_parents % 2:
        random_index = np.random.randint(num_parents)
        random_parent = parents[random_index]
        parents = np.concatenate([parents, [random_parent]])
        num_parents += 1
    for i in range(0, num_parents, 2):
        p1 = np.array(parents[i])
        p2 =  np.array(parents[i + 1])
        alpha = np.random.rand()
        # child = parent1 * random(x) + parent2 * (1-random(x))
        child = alpha * p1 + (1 - alpha) * p2
        children.append(child)
    return np.array(children)
```

5. **Mutation:** With a certain chance, we randomly tweak each coordinate with Gaussian noise.

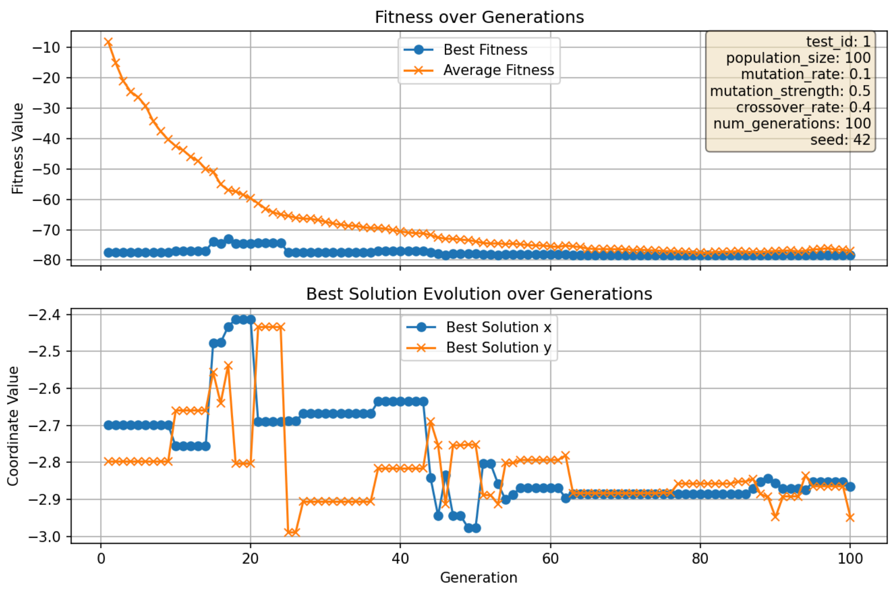
```
def mutate(self, individuals):
    for i in range(len(individuals)):
        #for each x and y
        for j in range(2):
            # decide by random if a child is mutated
            if np.random.rand() < self.mutation_rate:
                # a random gaussian mutation to a child
                individuals[i, j] += np.random.normal(0, self.mutation_strength)
                # don't let the values out of range
                x_range, y_range = init_ranges[styblinski_tang_2d]
                if j == 0:
                    individuals[i, j] = np.clip(individuals[i, j], x_range[0], x_range[1])
                elif j == 1:
                    individuals[i, j] = np.clip(individuals[i, j], y_range[0], y_range[1])
    return individuals
```

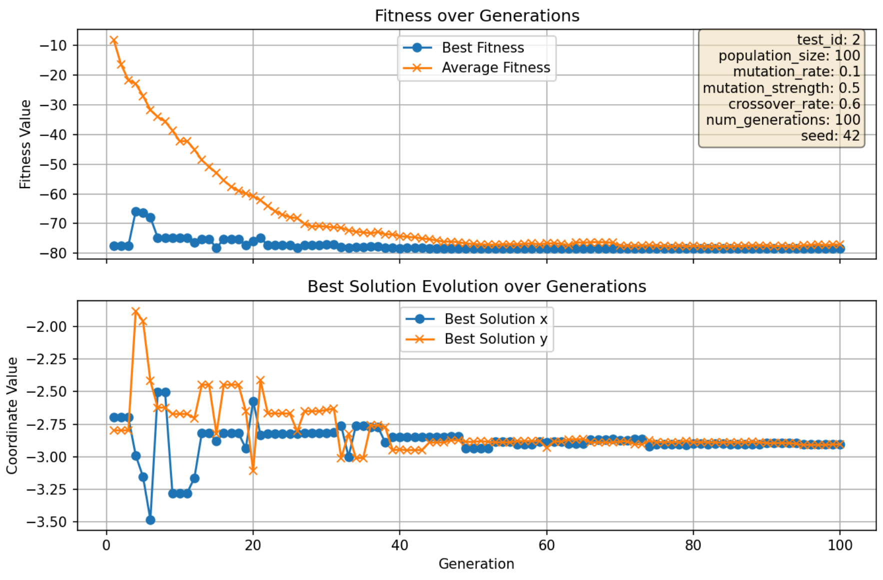
6. **Evolution loop:** Each generation, we: Evaluate current population, Select parents, Generate and mutate children, Replace part of the population with children

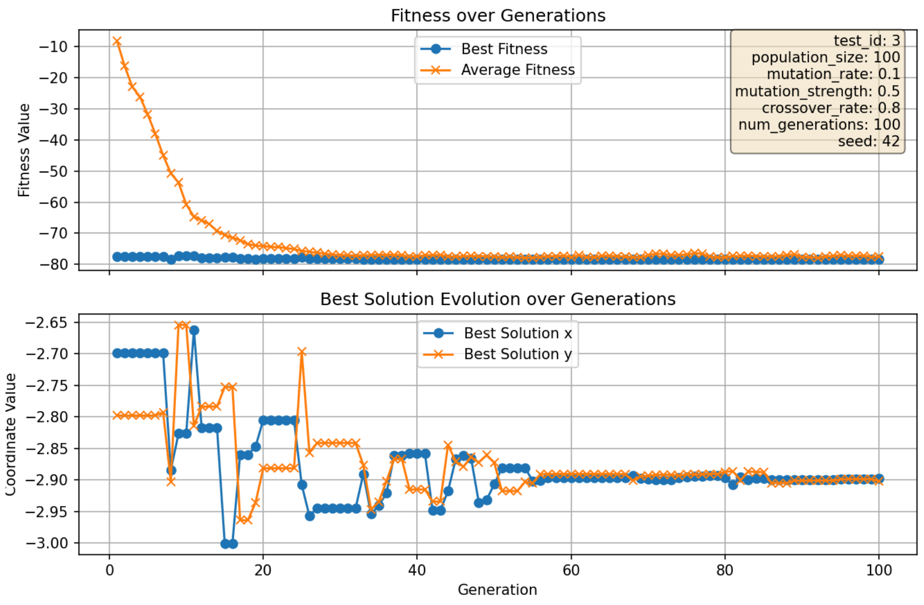
```
for generation in range(self.num_generations):
    fitness_values = self.evaluate_population(population)

    best_idx = np.argmin(fitness_values)
    best_solutions.append(population[best_idx])
    best_fitness_values.append(fitness_values[best_idx])
    average_fitness_values.append(np.average(fitness_values))

    parents_for_reproduction = self.selection(population, fitness_values)
    children = self.crossover(parents_for_reproduction)
    children = self.mutate(children)

    # MODIFY THIS
    # make is so that each children is replacing a random individual in a population
    indices = np.random.choice(range(len(population)), size=len(children), replace=False)
    for i, child in zip(indices, children):
        population[i] = child
```

Results

Crossover Rate tests:







Effect of Crossover Rate

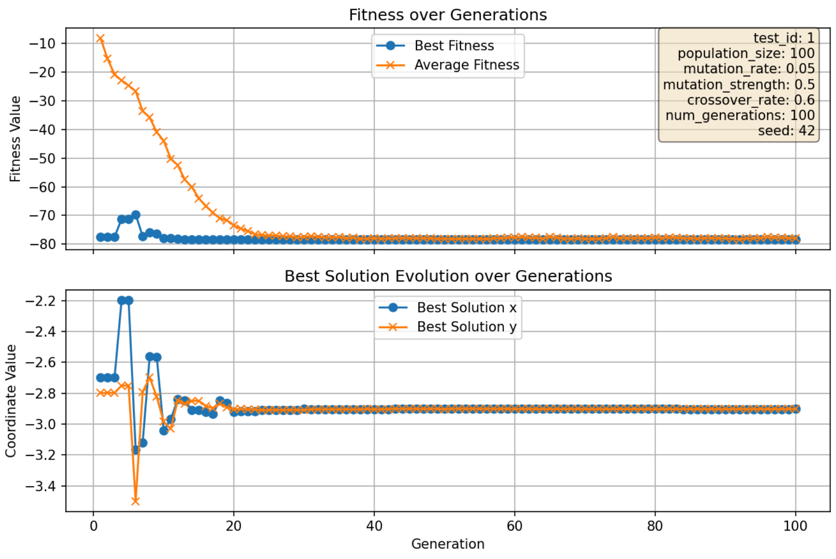
We tested three scenarios:

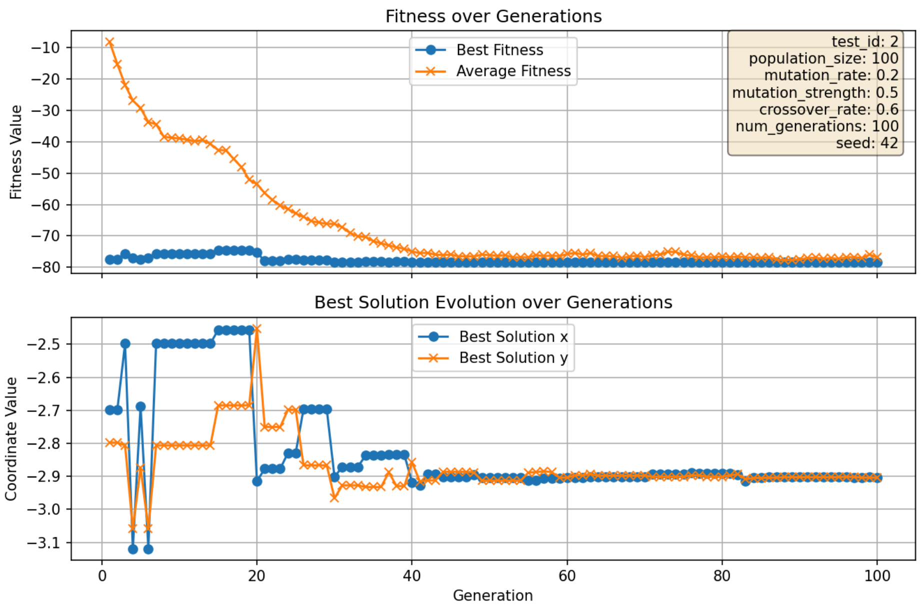
- Low crossover rate (0.4)
- Medium crossover rate (0.6)
- High crossover rate (0.8)

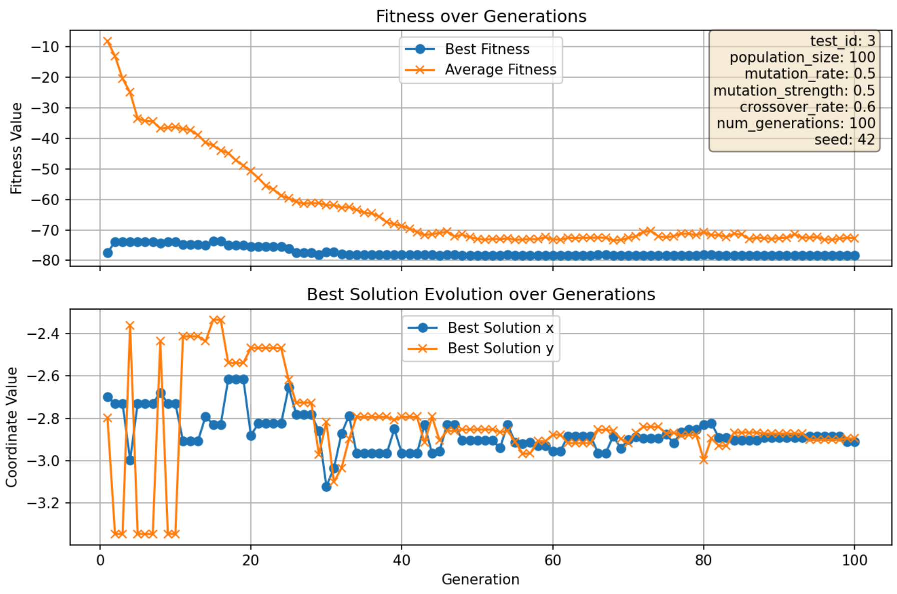
Summary:
Increasing the crossover rate generally improved the convergence speed, as more individuals participated in generating new offspring.

- At low crossover rates, the algorithm preserved too much of the current population, slowing down progress.
- At medium rates, there was a healthy balance between preservation and exploration.
- At high rates, the algorithm explored the search space aggressively, often reaching near-optimal solutions faster, but also increasing the risk of converging too early

Mutation Rate tests:







Effect of Mutation Rate

We tested:

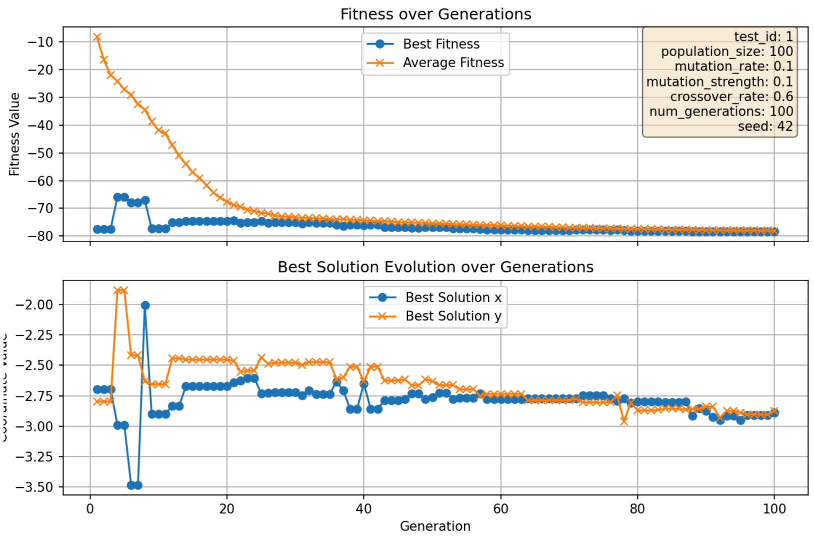
- Low mutation rate (0.05)
- Medium mutation rate (0.2)
- High mutation rate (0.5)

Summary:

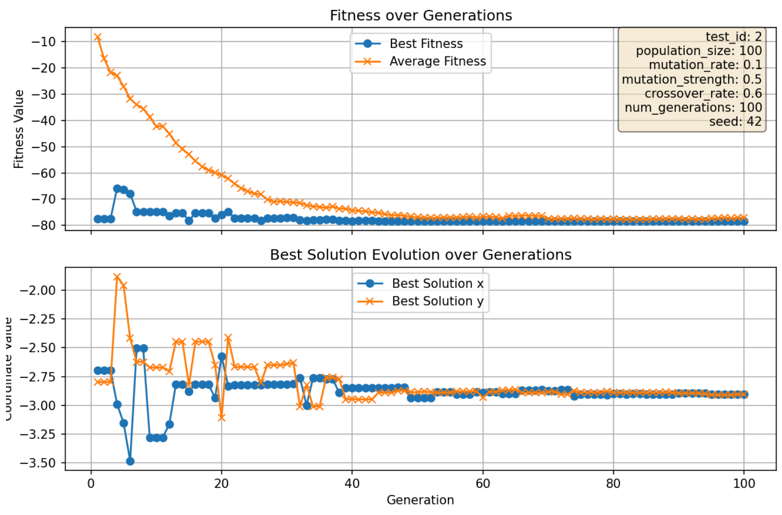
Mutation rate directly influenced the diversity of the population.

- Low mutation rates led to fast convergence, but may be stuck in local minimum
- Medium rates offered a good balance, allowing the algorithm to escape local optima
- High mutation leads to overshooting, and not being able to find the exact minimum

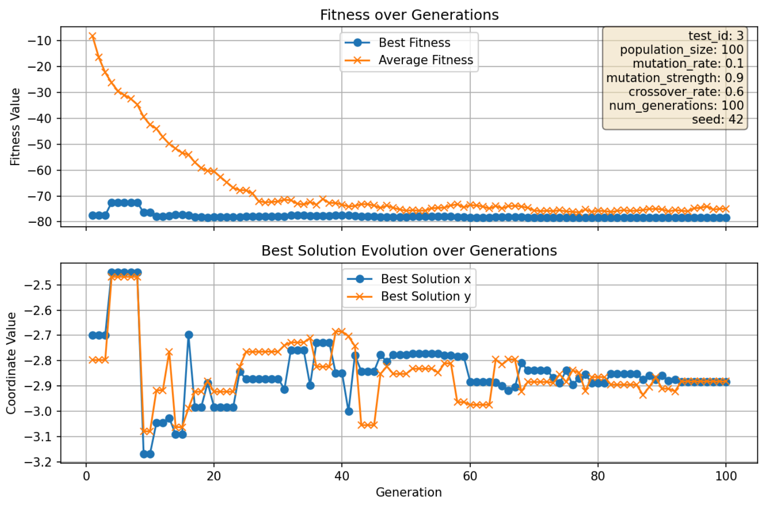
Mutation Strength tests:



test_id: 1
population_size: 100
mutation_rate: 0.1
mutation_strength: 0.1
crossover_rate: 0.6
num_generations: 100
seed: 42



test_id: 2
population_size: 100
mutation_rate: 0.1
mutation_strength: 0.5
crossover_rate: 0.6
num_generations: 100
seed: 42



test_id: 3
population_size: 100
mutation_rate: 0.1
mutation_strength: 0.9
crossover_rate: 0.6
num_generations: 100
seed: 42

Effect of Mutation Strength

We tested:

- Low mutation strength (0.1)
- Medium mutation strength (0.5)
- High mutation strength (0.9)

Summary:
Mutation strength controls how big the changes are when a gene is mutated.

- With low strength, changes are slow and it is ineffective to escape local minimum
- Medium strength helped the algorithm balance local search and broader exploration.
- High strength led to larger jumps which can escape local minima but also overshoot

Conclusion

Our experiments confirmed that genetic algorithms can effectively handle the challenges of optimizing non-convex functions with many local minima. However, the algorithm's performance is **highly sensitive to parameter tuning**, particularly the balance between **exploration** (via mutation) and **exploitation** (via selection and crossover).

Observations

- Medium crossover rate gives the best results. It balances keeping good solutions and trying new ones.
- Low crossover rate makes the algorithm slow. High crossover rate can lose good solutions too fast.
- Medium mutation rate helps escape local optima while still improving the solution.
- Low mutation rate can get stuck in bad areas. High mutation rate makes the results too random.
- Medium mutation strength makes helpful changes without jumping too far.
- Low mutation strength changes too little. High mutation strength changes too much and can make the search unstable.