Laboratory 3 – Genetic Algorithms

Variant 2 – Rastrigin Function Optimization using Rank Selection

Group 27

By Tian Duque Rey and Eduardo Sánchez Belchí

Task Description

In this lab assignment, the main goal is to find the global minimum of the 2D Rastrigin function, a common benchmark in evolutionary optimization. To do this, we implemented a Genetic Algorithm (GA) in Python, which evolves a population of candidate solutions over generations using principles inspired by natural selection. We used Rastrigin function using Rank Selection to choose parents and represented individuals with real-valued variables "X" and "Y" within the range [-5, 5]. The Rastrigin function is challenging due to its wide search space and many local minima, making it suitable for testing convergence.

Algorithm Implementation Details

The genetic algorithm was implemented using Python and NumPy. Each individual in the population is represented as a pair of real values (x, y) within the domain [-5, 5]. The fitness of each solution is evaluated using the 2D Rastrigin function:

The algorithm begins by initializing a random population. In each generation:

- Individuals are **ranked by fitness**, and parents are selected using **rank selection**, assigning higher probabilities to better-ranked solutions.
- A proportion of the population undergoes **crossover**, combining two parents using a **random interpolation** defined by a random value. The offspring are calculated as:

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

- The resulting offspring are then **mutated** by adding Gaussian noise with a standard deviation defined by the **mutation strength**. Each coordinate is perturbed as:

$$x' = x + \mathcal{N}(0, \sigma), \quad y' = y + \mathcal{N}(0, \sigma)$$

- A portion of the old population is **preserved (elitism)** and combined with the new offspring to form the next generation.

This process is repeated over a number of generations to promote convergence towards optimal values.

To tune the algorithm, we performed a grid search across a variety of parameter combinations (*test_1.py*):

- **Population Size**: [50, 100, 200]
- **Mutation Rate**: [0.01, 0.05, 0.1, 0.2]
- **Mutation Strength**: [0.05, 0.1, 0.3, 0.5, 0.8]
- **Crossover Rate**: [0.3, 0.5, 0.7, 0.9]
- **Generations**: [100, 200]

Each configuration was evaluated and compared by tracking the best fitness obtained at the final generation. The best set of parameters was then used in the following experiments. This systematic approach allowed us to identify the parameter set that achieved the lowest fitness values and demonstrated consistent convergence behavior.

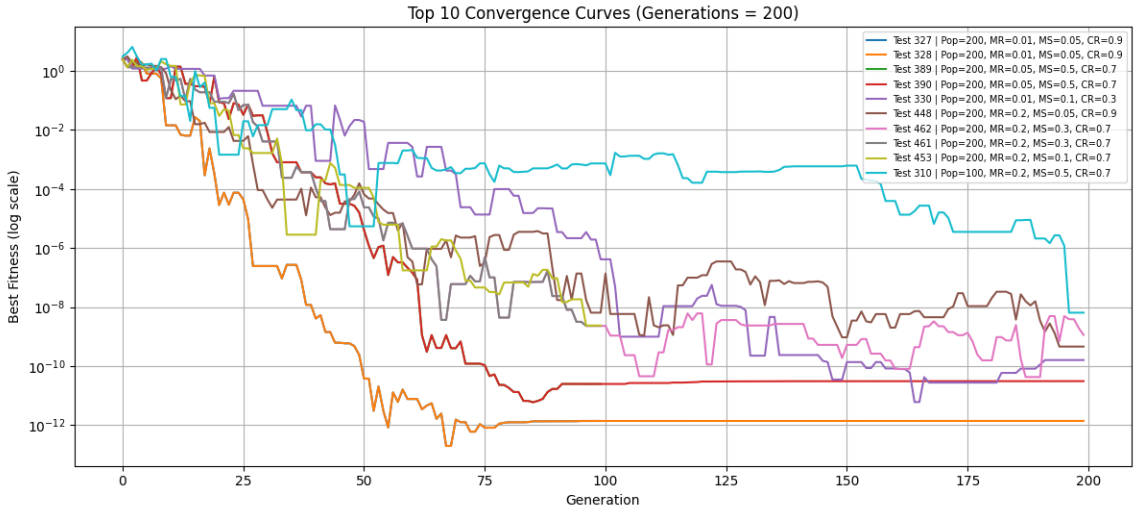
The next generation is formed by combining a subset of surviving individuals with the newly generated offspring, maintaining a constant population size.

Discussion

Experiment 1 – Hyperparameter Search

The results, visualized in the convergence plot "*fitness_plot_exp1_top10.png"*, showed a clear pattern: configurations with larger populations (especially 200), lower mutation rates (0.01–0.05), and moderate mutation strengths (0.05–0.3) consistently achieved faster and more stable convergence. These parameter choices effectively balanced exploration and exploitation, preventing premature convergence and improving the algorithm's ability to escape local minima.

Moreover, running the algorithm for more generations (200) significantly improved precision, with several runs achieving fitness values extremely close to the known global minimum of the **Rastrigin** function. The best-performing configuration—population size of 200, mutation rate of 0.01, mutation strength of 0.05, crossover rate of 0.9, and 200 generations—produced a final fitness as low as 1.8×10^{-13}. This parameter set was therefore selected as the baseline for all subsequent experiments.



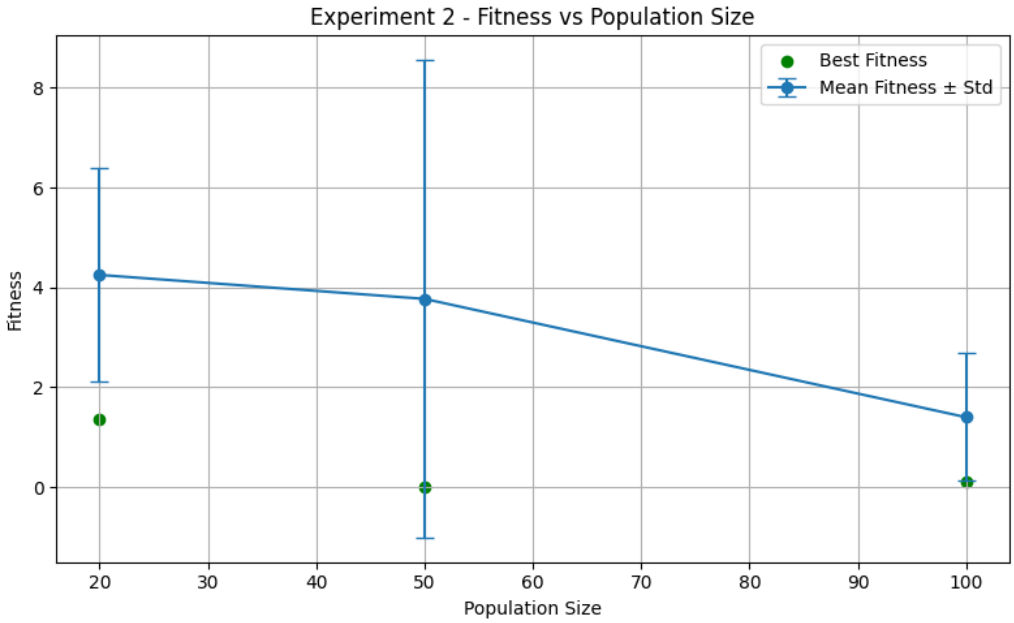
***Pic**. fitness_plot_exp1_top10.png*

Experiment 2 – Randomness and Population Size

The results, available in "*experiment2_randomness_summary.csv*", demonstrated high consistency across runs. The best fitness values remained close to zero, with a low standard deviation, indicating that the algorithm is highly robust to stochastic variations in initialization and genetic operations.

In the second part of the experiment, we explored the effect of reducing population size. As summarized in "*experiment2_population_summary.csv*", we observed a marked degradation in performance as the population decreased. With a size of 100, the algorithm maintained excellent convergence, achieving a best fitness of 0.10 and a mean of 1.40. However, with only 20 individuals, the best fitness dropped to 1.35, and the variance increased significantly.

The first figure, "*Fitness vs Population Size*", clearly demonstrates the influence of population size on the algorithm's performance. The line plot shows the average fitness values across multiple runs, while the error bars represent the standard deviation, and green points indicate the best fitness obtained for each size. As population size increases, both the mean fitness improves and the variance decreases, indicating more consistent and accurate convergence. For example, the population of size 100 consistently achieved fitness values near zero with very low variability, while the smallest population (20) exhibited higher average fitness and a larger standard deviation. This supports the conclusion that maintaining population diversity is crucial for robust and stable optimization.

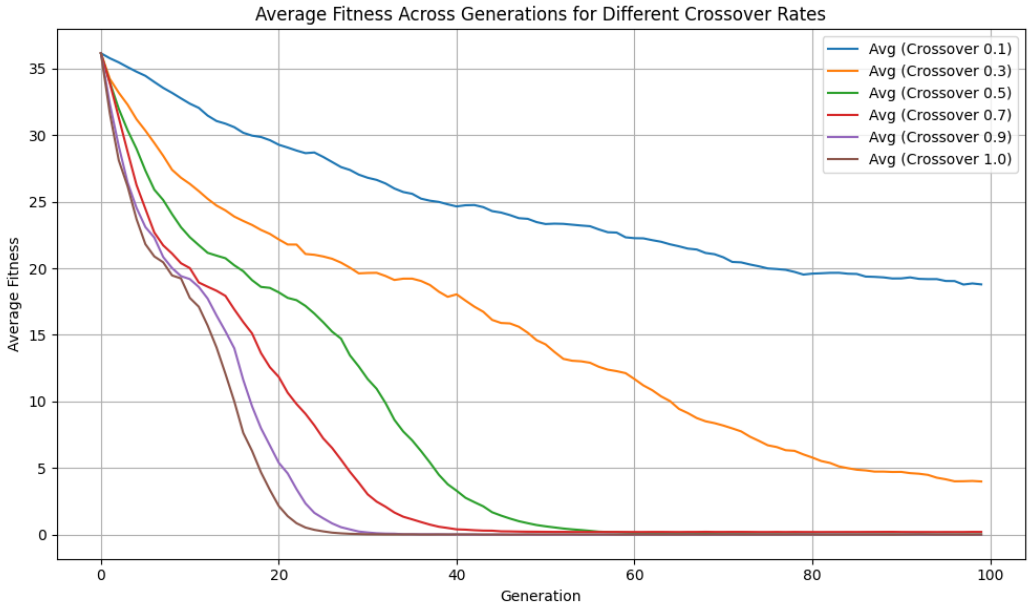


***Pic**. Fitness vs Population Size*

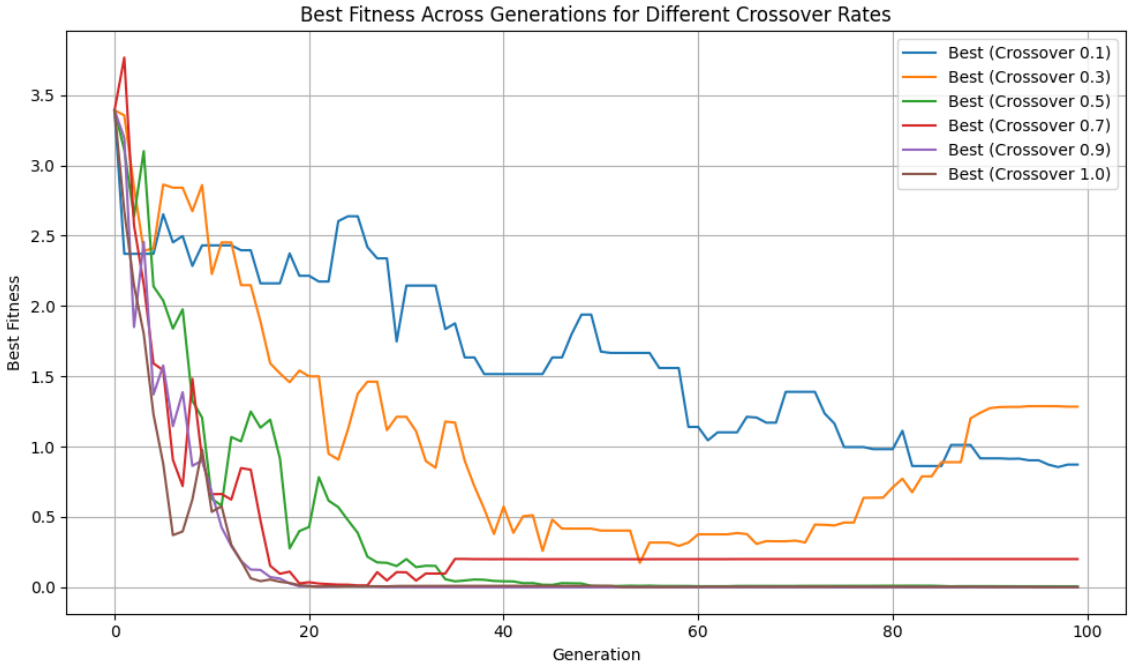
The second figure "*experiment2_randomness_barplot*" from file "*data_test*" confirms that the algorithm performs consistently across different random seeds, indicating low sensitivity to randomness. Overall, population size has a stronger impact on convergence than initialization variability.

Experiment 3 – Crossover Rate Impact

The **first plot**, *Average Fitness Across Generations*, shows a clear trend: higher crossover rates (≥ 0.7) result in significantly faster and smoother convergence. In contrast, lower rates (e.g., 0.1 and 0.3) struggle to reduce average fitness, even after 100 generations. This suggests that increasing the proportion of individuals undergoing crossover enhances the algorithm's ability to generate diverse and high-quality offspring.



The **second plot**, *Best Fitness Across Generations*, reinforces this finding. Crossover rates of 0.7 and above consistently reach fitness values near zero within 30–40 generations. On the other hand, low rates (particularly 0.1) converge slowly and remain far from the global minimum, indicating insufficient recombination.

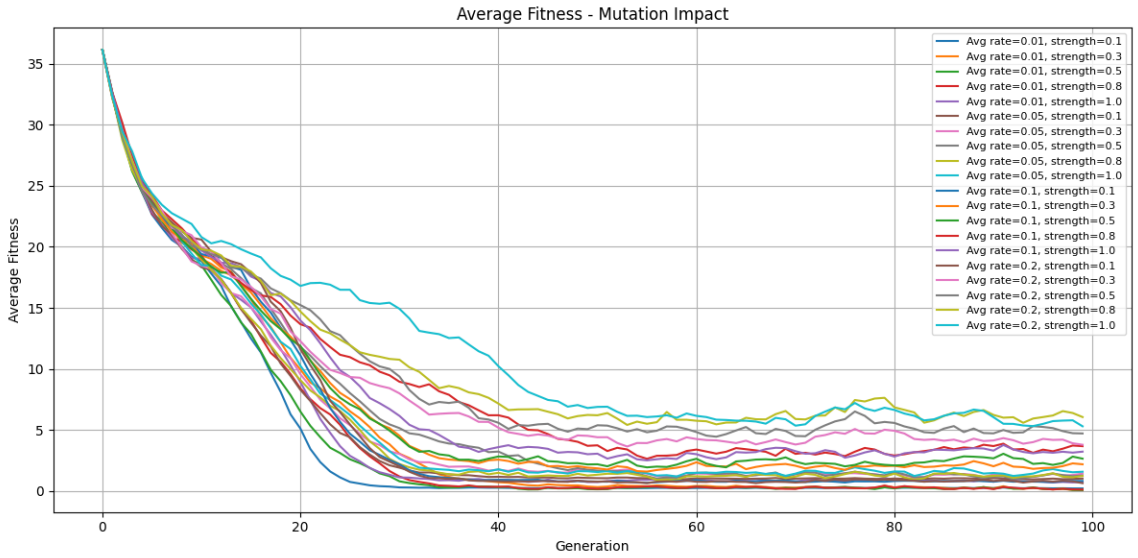


Pic. *Best Fitness Across Generations*

The numerical data in experiment3_summary.csv confirms these trends. The **average best fitness** for crossover rates 0.7–1.0 converges to values below 1e-2, while lower rates stagnate above 1.0. These results underline the crucial role of crossover in evolutionary performance: **a high crossover rate accelerates convergence by enabling the recombination of good traits** from different parents, while low rates hinder exploration and adaptation.

Experiment 4 - Mutation

The **first plot**, *Average Fitness – Mutation Impact*, shows that combinations with **low to moderate mutation rates (0.01–0.1)** and **mutation strengths between 0.1 and 0.5** tend to produce smoother and more effective convergence. In contrast, higher mutation strengths (especially 0.8–1.0) result in noisier behavior and slower convergence, often preventing the algorithm from refining high-quality solutions.



Pic. *Average Fitness – Mutation Impact*

The **second plot**, *"experiment4_best_fitness"*, confirms this trend. The best-performing configurations reached near-zero fitness before generation 40 when using mutation rates around 0.05–0.1 and moderate mutation strengths (0.1–0.3). On the other hand, high-strength configurations showed stagnation, failing to improve beyond fitness values of 0.5 or higher. This highlights the importance of **not overwhelming the population with excessive randomness**, which can disrupt progress rather than help it.

The raw data from experiment4_summary.csv supports these findings. Configurations like mutation rate 0.1 with strength 0.3 or 0.5 consistently show the **lowest average and best fitness values across generations**, indicating a good balance between exploration and exploitation.

Conclusion

Through this project, we successfully applied a genetic algorithm to optimize the 2D Rastrigin function, evaluating how different hyperparameters affect performance.

Our experiments showed that the best results were achieved with **large populations**, **low mutation rates**, **moderate mutation strengths**, and **high crossover rates**. These settings enabled fast and stable convergence toward the global minimum.

We also found that the algorithm is **robust to randomness** across different seeds, but **sensitive to population size**—smaller populations reduced diversity and impaired performance. Additionally, **higher crossover rates** improved recombination and convergence speed, while **moderate mutation settings** helped maintain balance between exploration and exploitation.