Computational Intelligence (CI-MAI)

Optimizing Difficult Functions with Evolution Strategies

Toygar Yurt
Zoë Finelli
Onat Bitirgen

Universitat Politècnica de Catalunya
Master in Artificial Intelligence November 10, 2025

Contents

1 Introduction & Problem Setup

Traditional optimization methods, particularly those based on derivatives like quasi-Newton methods (e.g., L-BFGS-B), are highly efficient for solving problems that are smooth, continuous, and unimodal. However, they are susceptible to failure when faced with "difficult" optimization landscapes: those that are highly multimodal, non-differentiable, or deceptive [3] [1] . As noted in our lectures, these methods perform local search and are thus easily trapped in the nearest local optimum.

Evolutionary Algorithms (EAs) provide a robust, population-based, derivative-free alternative for global search in these complex spaces. This project focuses on Evolution Strategies (ES), a specific branch of EA. Unlike Genetic Algorithms (GAs) which traditionally operate on discrete, binary representations, ES is designed natively for continuous optimization problems, operating directly on real-valued vectors [1] [4] .

The objective of this project is to implement and analyze the performance of a self-adaptive Evolution Strategy on two benchmark functions. We compare the performance of two primary ES selection strategies—$(\mu + \lambda)$ and (μ, λ)—against each other and against the standard derivative-based L-BFGS-B algorithm [7] .

To test both **exploitation** (local tuning) and **exploration** (global search) capabilities, we selected two benchmark functions from the Surjanovic & Bingham library [10] [5] at **10 and 20 dimensions**. A key part of the analysis, as suggested in the project brief [3] , will be to study the effect of problem dimensionality (n) on algorithm performance and robustness.

- **Sphere Function:** $f(x) = \sum_{i=1}^{n} x_i^2$
  This is a simple, convex, and unimodal function. Its global optimum is $f(0, ..., 0) = 0$. It is used to validate our algorithm's basic ability to perform local optimization [10] .
- **Rastrigin Function:** $f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i))$
  This is a highly "difficult" multimodal function with a regular, grid-like structure of many local optima [10] . The global optimum is also $f(0, ..., 0) = 0$. It is designed specifically to trap algorithms that rely on local or gradient-based search [3] .

2 Methodology & Implementation (Our Work)

Our implementation was developed in Python 3 [12] , using ‘numpy‘ for vector operations, ‘scipy‘ for the baseline algorithm, and ‘matplotlib‘ for visualization [13] . The core logic is encapsulated in several modules within the ‘src/‘ directory [14] [12] .

2.1 ES Algorithm (evolution_strategy.py)

We implemented two distinct replacement strategies as defined in the course lectures [1] :

- $(\mu + \lambda)$**-ES ("plus"):** This is an **elitist** strategy where the μ best individuals are selected from a combined pool of the μ current parents and the λ new offspring. This strategy guarantees that the best-found solution is never lost [1] [12] [8] .

- (μ, λ)**-ES ("comma"):** This is a **non-elitist** strategy where the μ best individuals are selected *only* from the λ offspring (requiring $\lambda \geq \mu$). Parents are discarded, allowing the algorithm to "forget" local optima and potentially perform a more robust global search [1] [12] [8] .

2.2 Representation

Individuals are represented as a real-valued vector $\vec{x} \in \mathbb{R}^n$ for the object parameters (the solution) and one additional real-valued σ for the mutation strength (the strategy parameter) [8] [12] . This coupling of the solution and its mutation parameter into a single individual is a hallmark of ES and the foundation for self-adaptation [4] .

2.3 Mutation (Self-Adaptation)

We implemented a self-adaptive mutation strategy based on **"Mutation (I)"** from the lecture slides (one σ per individual) [2] . This allows the algorithm to learn the optimal mutation strength during the run.

1. **Strategy Parameter (σ) Mutation:** The mutation strength is updated first using a log-normal rule [8] [2] :
$$\sigma' = \sigma \cdot \exp(\tau \cdot N(0, 1))$$
where $N(0, 1)$ is a standard normal random variable and τ is the learning rate. Following standard recommendations for this mutation rule, we set $\tau = 1/\sqrt{n}$, where n is the problem dimension [9] .
2. **Object Parameter ($\vec{x}$) Mutation:** The object parameters are then mutated using the *new* σ' [8] :
$$\vec{x}' = \vec{x} + \sigma' \cdot \vec{N}(0, I)$$
where $\vec{N}(0, I)$ is a vector of n independent standard normal random variables.

2.4 Recombination & Selection

- **Recombination:** No recombination was used in this experiment [12] .
- **Selection:** Survivor selection is deterministic and rank-based. In both strategies, the μ-best individuals are selected based on having the lowest fitness (lowest function value) [8] [4] .

2.5 Justification of Design Choices

- **Why Evolution Strategies?** Our problem is continuous, real-valued optimization [3] . ES is the ideal EA for this domain, as its primary representation and operators (like Gaussian mutation) are designed for $\mathbb{R}^n$ [1] [4] .
- **Why Self-Adaptation?** The choice of mutation strength (σ) is one of the most critical parameters in an ES. A fixed σ would either converge too slowly (if too small) or fail to fine-tune (if too large). Self-adaptation (learning σ during the run) is a form of parameter control [5] that allows the algorithm to automatically adjust its search granularity.

- **Why Mutation (I) and not (II)?** We chose "Mutation (I)" (one σ) as a balance of simplicity and power, as outlined in the "Intro to ESs" slides [2] . It allows the algorithm to learn the *global* step size, which is a significant improvement over a fixed σ. We left the more complex n-sigma Mutation (II) (diagonal adaptation) for future work [2] [8] .
- **Why no Recombination?** We deliberately omitted recombination to isolate and study mutation as the sole search operator. This gives us a clearer picture of the power of self-adaptation and the difference between the 'plus' and 'comma' survival strategies [12] .

3 Experimental Design

To ensure a fair and statistically valid comparison, we used a dedicated 'ExperimentRunner' class [9] . As required by the project brief [3] , every configuration was run for 30 independent trials.

Our baseline comparison method is the L-BFGS-B algorithm, a quasi-Newton (derivative-based) method provided by 'scipy.optimize.minimize' [7] .

The parameters for all 8 experimental configurations are detailed in Table 1. Note that the learning rate τ for the ES is set to $1/\sqrt{n}$ and thus adapts based on the problem dimension [9] .

Table 1: Experimental Parameters for all 30-run trials [7] [9]

Function	Dim (n)	Strategy	μ	λ	τ (Learning Rate)	Initial σ	Target Fitness	Max Func. Evals
Sphere	10	L-BFGS-B	N/A	N/A	N/A	N/A	1×10^{-6}	100,000
Sphere	10	$(\mu + \lambda)$ 'plus'	15	100	$1/\sqrt{10} \approx 0.316$	0.5	1×10^{-6}	70,020
Sphere	10	(μ, λ) 'comma'	15	100	$1/\sqrt{10} \approx 0.316$	0.5	1×10^{-6}	70,020
Sphere	20	L-BFGS-B	N/A	N/A	N/A	N/A	1×10^{-6}	100,000
Sphere	20	(μ, λ) 'comma'	15	100	$1/\sqrt{20} \approx 0.224$	0.5	1×10^{-6}	70,020
Rastrigin	10	L-BFGS-B	N/A	N/A	N/A	N/A	1×10^{-6}	100,000
Rastrigin	10	$(\mu + \lambda)$ 'plus'	20	140	$1/\sqrt{10} \approx 0.316$	0.5	1×10^{-6}	70,020
Rastrigin	10	(μ, λ) 'comma'	20	140	$1/\sqrt{10} \approx 0.316$	0.5	1×10^{-6}	70,020
Rastrigin	20	L-BFGS-B	N/A	N/A	N/A	N/A	1×10^{-6}	100,000
Rastrigin	20	(μ, λ) 'comma'	30	200	$1/\sqrt{20} \approx 0.224$	0.5	1×10^{-6}	100,020

Performance Metrics: We collected the following metrics as required by the brief [3] :

1. **Best Fitness:** The final fitness value of the best individual.
2. **Function Evaluations:** The total number of calls to the objective function.
3. **Execution Time:** Total wall-clock time for the run.
4. **Success Rate:** The percentage of the 30 runs that reached the `target_fitness` of 1×10^{-6}.

4 Results & Discussion

All results are the product of the code in the 'src/' directory, executed by 'main.py' [7] . The full aggregated results are presented in Table 2, generated from our 'summary_statistics.csv' file [15] .

Table 2: Summary Statistics (Mean/Std) over 30 Independent Runs [15]

			Best Fitness			Function Evals		Time (s)		Converged	
Function	**Dim**	**Strategy**	**Mean**	**Std**	**Min**	**Mean**	**Std**	**Mean**	**Std**	**Sum**	**Mean**
rastrigin	10	L-BFGS-B	70.045	22.793	38.803	207.5	50.51	0.002	0.001	0	0.0
	10	comma	33.099	11.024	11.939	70020.0	0.00	0.599	0.015	0	0.0
	10	plus	35.454	10.469	19.899	70020.0	0.00	0.597	0.014	0	0.0
	20	L-BFGS-B	132.428	31.782	93.526	402.5	77.81	0.004	0.001	0	0.0
	20	comma	88.020	19.497	49.748	100030.0	0.00	0.860	0.022	0	0.0
sphere	10	L-BFGS-B	0.000	0.000	0.000	39.2	5.54	0.001	0.001	30	1.0
	10	comma	0.000	0.000	0.000	6465.0	317.04	0.044	0.002	30	1.0
	10	plus	0.000	0.000	0.000	6281.7	297.50	0.043	0.002	30	1.0
	20	L-BFGS-B	0.000	0.000	0.000	74.2	10.66	0.001	0.000	30	1.0
	20	comma	0.000	0.000	0.000	13388.3	487.73	0.094	0.005	30	1.0

4.1 Analysis 1: Sphere Function (Unimodal)

On the simple, unimodal Sphere function, all methods (10D and 20D) performed perfectly, achieving a **100% success rate** (30/30 runs) in finding the global optimum (Table 2).

The key difference was efficiency. The L-BFGS-B baseline was vastly more efficient, requiring only **≈39 evaluations** on average for 10D and **74.2 evaluations** for 20D. Our ES strategies required **≈6,300-6,500 evaluations** for 10D (Table 2).

Effect of Dimension ($n = 20$): As shown in Table 2, the (μ, λ)-ES also achieved a 100% success rate on the 20D Sphere function. The primary effect of the increased dimension was a predictable increase in the number of function evaluations required, with the ES needing **13388.3** evaluations on average, compared to ≈6,465 for the 10D version. This aligns with the convergence plot (Figure 2). This test successfully validates our ES's ability to perform exploitation.

4.2 Analysis 2: Rastrigin Function (Difficult & Multimodal)

The Rastrigin function yielded the most important findings.

ES vs. Baseline (n=10): As predicted, the derivative-based **L-BFGS-B failed completely** in 10D. It was consistently trapped in a poor local optimum (avg. fitness 70.045) with a 0% success rate (Table 2). Our **Evolution Strategies also achieved a 0% success rate** of reaching the *exact* target (1×10^{-6}), confirming this is a "difficult" problem.

Robustness (n=10): Despite not reaching the target, the 10D ES methods were *significantly* more robust at global exploration. They consistently found much deeper valleys, with average fitness values of 33.10 ('comma') and 35.45 ('plus') (Table 2). This is a massive improvement over the baseline's 70.045. The box plot in Figure 1 clearly illustrates this.

Effect of Dimension ($n = 20$): The 20-dimensional Rastrigin problem proved significantly more difficult. The L-BFGS-B baseline again failed, finding an average fitness of **132.428**. The (μ, λ)-ES also failed to find the global optimum, and its average best fitness of **88.020** was considerably worse than the ≈33.10 achieved in 10D. This highlights the "curse of dimensionality" and the challenge this function poses as the search space expands.

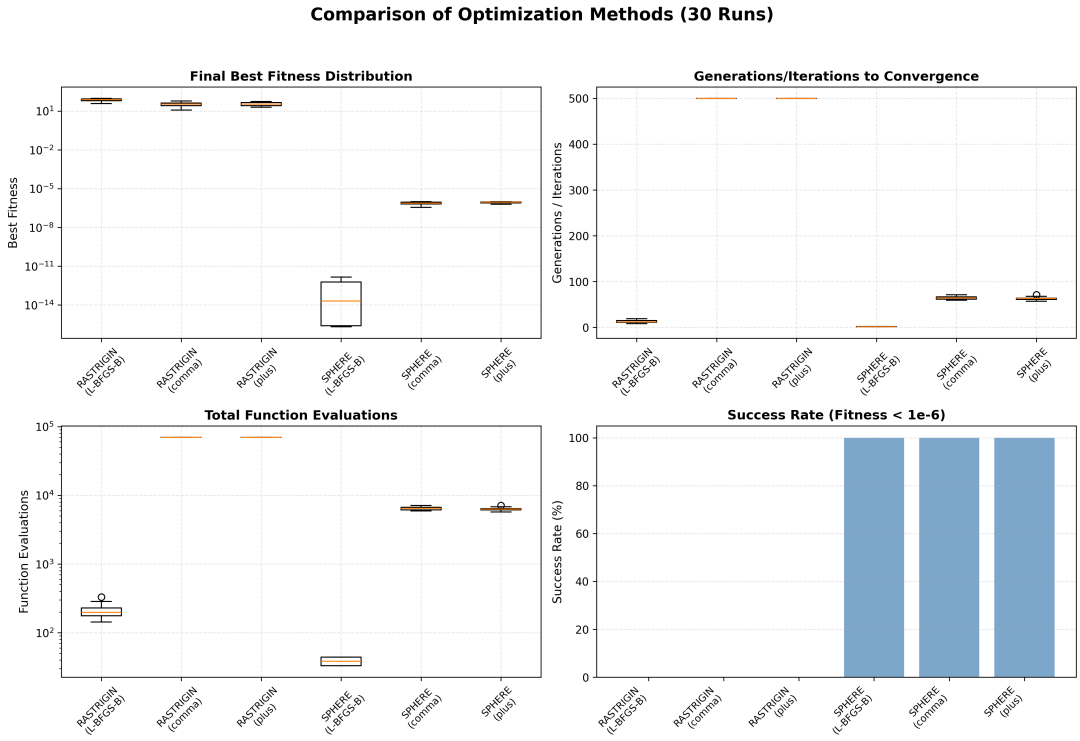

Figure 1: Comparison of Final Best Fitness (Box Plot) [16] . Note the logarithmic y-axis. The 20D results will now be included, showing the increased difficulty for Rastrigin. (Plot generated by ‘visualization.py‘ [11]).

4.3 Analysis 3: Strategy (Plus vs. Comma) in 10D

The summary table (Table 2) reveals the difference between our two ES strategies on the 10D Rastrigin function. The non-elitist (μ, λ)-’comma’ strategy found a *better* (lower) average local optimum (fitness 33.10) than the elitist $(\mu + \lambda)$-’plus’ strategy (fitness 35.45).

We hypothesize this is due to the ’comma’ strategy’s strength in **exploration**. By discarding parents at each generation [1] [8] , it is forced to move out of the current basin of attraction and can explore new regions. The ’plus’ strategy, by being elitist [1] [8] , likely held onto a "good" solution (a strong local optimum) and converged there prematurely.

4.4 Analysis 4: Convergence and Dimensionality

Figures 2 and 3 show the convergence curves comparing the 10D and 20D results.

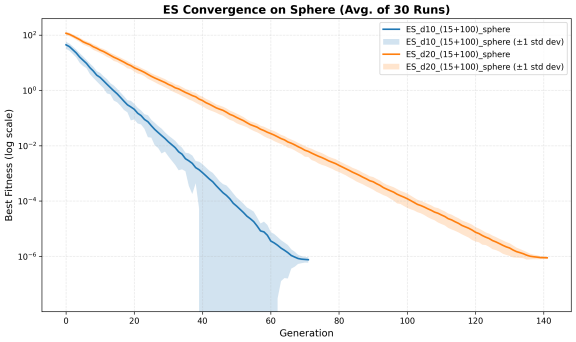

Figure 2: ES Convergence on Sphere (10D vs. 20D) [17] . (Plot generated by 'visualization.py' [11]).

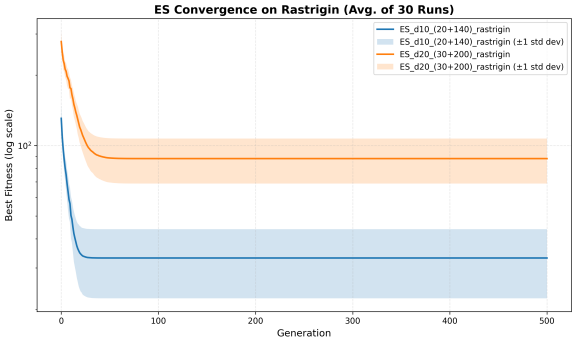

Figure 3: ES Convergence on Rastrigin (10D vs. 20D) [18] . (Plot generated by 'visualization.py' [11]).

On the **Sphere function** (Figure 2), the effect of dimension is clear: both 10D algorithms converge rapidly, while the 20D algorithm also converges reliably but requires approximately twice as many generations to reach the same fitness threshold. This is expected, as the problem is still convex but the search space is larger.

On the **Rastrigin function** (Figure 3), the impact of dimensionality is far more severe. The 10D runs quickly find a deep local optimum (fitness ≈33-35) and plateau. The 20D run, however, converges significantly slower and plateaus at a much higher (worse) fitness value of **88.020**. This demonstrates that as the search space grows, the ES (with these parameters) struggles to effectively explore and becomes trapped in poorer local optima.

5 Future Work & Modifications

While our experiment successfully demonstrated the core concepts, the 0% success rate on Rastrigin shows there is room for improvement. Based on our findings, future work could include:

- **Implement Recombination:** We deliberately omitted recombination. Adding intermediate recombination for both the object variables ($\vec{x}$) and sigmas (σ) could significantly speed up convergence [4] .
- **Implement Mutation (II):** We used one σ for all dimensions. Implementing the "diagonal" method (Mutation II) with n self-adaptive σ values (one per dimension) would allow the algorithm to learn different step sizes for each variable, which would be highly beneficial for problems like Rastrigin [2] [5] .
- **Tune Selection Pressure:** Our (μ, λ) ratios were just one choice. A deeper study on the selection pressure (λ/μ) could further optimize the balance between exploration and exploitation, especially for higher-dimensional problems.

6 Conclusion

This project successfully implemented and evaluated two core Evolution Strategies, $(\mu + \lambda)$-ES and (μ, λ)-ES, with self-adaptive mutation [8] [2] .

Our key findings are:

1. **ES is a valid optimizer:** Our ES implementation correctly solved the unimodal Sphere function 100% of the time in both 10D and 20D, though it was less efficient than the L-BFGS-B baseline (Table 2).
2. **ES is robust for difficult problems:** On the highly multimodal Rastrigin function, the derivative-based L-BFGS-B failed completely in all dimensions. Our ES strategies, while not finding the global optimum, proved significantly more robust by finding much deeper valleys in the search space (Table 2).
3. **Strategy matters for exploration:** In 10D, the non-elitist (μ, λ)-'comma' strategy outperformed the elitist $(\mu + \lambda)$-'plus' strategy on the "difficult" Rastrigin function (Table 2), highlighting its superior ability to escape local optima by "forgetting" its parents [1] .
4. **Dimensionality increases difficulty:** The problem difficulty, particularly for the multimodal Rastrigin function, increased significantly with dimension. While the 10D ES found deep local optima (fitness $\approx$33), the 20D ES was trapped in much worse solutions (fitness $\approx$ **88.020**), demonstrating the challenge of exploration in a larger search space (Table 2).

This confirms the core principles of Evolutionary Computation: while not always the fastest, population-based, derivative-free search is a powerful and robust tool for navigating complex problem landscapes where traditional methods fail [1] .

7 References

References

7.1 Academic & Course References

[1] Belanche, L. (2025). "2. Intro to Evolutionary Computation" [Lecture Slides, 2.Intro_to_EC_compressed.pdf]. Master in Artificial Intelligence, UPC.

[2] Belanche, L. (2025). "4. Intro to Evolution Strategies" [Lecture Slides, 4._Intro_to_ESs.pdf]. Master in Artificial Intelligence, UPC.

[3] Belanche, L. (2025). "Evolutionary Computation practical work" [Project Brief, practical-exercises.html]. Master in Artificial Intelligence, UPC.

[4] Bäck, T., & Schwefel, H. P. (1993). *An overview of evolutionary algorithms for parameter optimization*. Evolutionary computation, 1(1), 1-23.

[5] Hinterding, R., Michalewicz, Z., & Eiben, A. E. (1999). *Parameter control in evolutionary algorithms*.

[6] Surjanovic, S. & Bingham, D. (2013). "Virtual Library of Simulation Experiments". `https://www.sfu.ca/~ssurjano/optimization.html`

7.2 Project Artifacts

[7] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘main.py‘ [Source Code]. CI-MAI Project.

[8] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘evolution_strategy.py‘ [Source Code]. CI-MAI Project.

[9] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘experiment_runner.py‘ [Source Code]. CI-MAI Project.

[10] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘test_functions.py‘ [Source Code]. CI-MAI Project.

[11] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘visualization.py‘ [Source Code]. CI-MAI Project.

[12] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘README.txt‘ [Project Documentation]. CI-MAI Project.

[13] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘requirements.txt‘ [Project Documentation]. CI-MAI Project.

[14] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘__init__.py‘ [Source Code]. CI-MAI Project.

[15] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘summary_statistics.csv‘ [Result Data]. CI-MAI Project.

[16] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘comparison_boxplots.png‘ [Result Figure]. (Note: filename updated from .jpg to .png as in code).

[17] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘convergence_sphere.png‘ [Result Figure]. CI-MAI Project.

[18] Yurt, T., Finelli, Z., & Bitirgen, O. (2025). ‘convergence_rastrigin.png‘ [Result Figure]. CI-MAI Project.