Comparing Evolutionary Algorithms for Black-Box Optimization

IMT2022025 Varnit Mittal, IMT2022075 Aditya Priyadarshi, IMT2022076 Mohit Naik

November 2024

Introduction

Black-box optimization involves solving optimization problems where the internal structure or exact mathematical representation of the objective function is unknown or inaccessible. In such problems, the objective function is treated as a ”black box” that provides output values for given input parameters, but no gradient or derivative information is available. This makes black-box optimization particularly challenging, especially when the search space is high-dimensional, non-convex, or contains multiple local optima. Black-box optimization is ubiquitous in real-world scenarios, such as hyperparameter tuning in machine learning, engineering design optimization, and financial modeling.

A common constraint in black-box optimization problems is the limitation on the number of objective function evaluations. This constraint arises because each evaluation can be computationally expensive or time-consuming, particularly when the function represents a complex simulation or real-world process. As a result, optimization algorithms must achieve high-quality solutions within a finite budget of evaluations, making efficiency and adaptability critical performance criteria.

Evolutionary algorithms (EAs) are a class of metaheuristic optimization techniques inspired by natural evolution and biological processes, such as selection, mutation, and recombination. These algorithms are well-suited for black-box optimization because they rely solely on objective function evaluations, making no assumptions about its underlying structure. By maintaining and iteratively evolving a population of candidate solutions, EAs explore the search space effectively, striking a balance between exploration (diverse sampling of the search space) and exploitation (refinement of promising areas).

In this work, we implement and compare three widely-used evolutionary algorithms — Differential Evolution (DE), Covariance Matrix Adaptation Evolution Strategy (CMAES), and Particle Swarm Optimization (PSO) — along with their respective variants. By benchmarking these methods on a standard suite of black-box optimization functions, we aim to evaluate their performance and identify the strengths and limitations of each approach.

Problem Statement

Given a function f in n_dim dimensions along with upper and lower bounds for all of its dimensions (to constrain the search space), and a maximum allowed limit of function evaluations max_evals, the job is to find the optimizer x^* of f. The optimum value will then be $f(x^*)$. Thus, the problem statement is to find

$$x^* = \arg\min_x \ f(x)$$

under the given constraints of the search space and the number of evaluations.

Differential Evolution

Differential Evolution (DE) is a population-based metaheuristic optimization algorithm used to solve black-box optimization problems. It operates by iteratively improving a population of candidate solutions through mutation, crossover, and selection processes. DE is particularly suited for continuous optimization problems and performs well in high-dimensional and non-convex search spaces, as well as being robust and comparatively simple to implement.

The main components of the algorithm are -

1. Initialization: A population of candidate solutions is randomly initialized within the given bounds of the search space. Each candidate solution represents a point in the search space.
2. Mutation: New candidate solutions, known as mutant vectors, are generated by combining existing population members using scaled differences. This introduces diversity and helps the algorithm explore the search space.
3. Crossover: A crossover operation combines elements of the mutant vectors with the original population, creating trial vectors. This allows the algorithm to exploit promising areas of the search space.
4. Selection: For each candidate in the population, the trial vector is evaluated, and it replaces the original candidate if it has a better fitness value.
5. Termination: The algorithm iterates until a predefined termination condition, such as a maximum number of function evaluations, is met.

Below is the pseudocode for the classical Differential Evolution algorithm:

Algorithm 1 Classical Differential Evolution (CDE)

```
1: Initialize population $X = \{x_1, x_2, \dots, x_n\}$ randomly within bounds
2: Evaluate the fitness for each individual in $X$, $y_i = f(x_i)$
3: while termination condition (max_evals) not met do
4:     for each individual $x_i$ in $X$ do
5:         Mutation:
6:         Select four random, distinct individuals $x_{r1}, x_{r2}, x_{r3}, x_{r4}$
7:         Generate mutant vector $v_i = x_{r1} + f_\mu \cdot (x_{r2} - x_{r3})$
8:         Crossover:
9:         for each dimension $j$ do
10:            if rand$_j < cr$ or $j =$ rand_index then
11:                $u_{ij} = v_{ij}$
12:            else
13:                $u_{ij} = x_{ij}$
14:            end if
15:        end for
16:        Selection:
17:        if $f(u_i) < f(x_i)$ then
18:            $x_i = u_i$
19:        end if
20:    end for
21:    Update fitness for the new population $y_i = f(x_i)$
22: end while
23: return Best solution $x^*$ and its fitness $f(x^*)$
```

We also implemented an adaptive variant of the above algorithm, Adaptive Differential Evolution (ADE), which has a number of improvements -

1. **Dynamic Adaptation of Parameters:**
   - Unlike Classical Differential Evolution (CDE), ADE adapts key parameters, namely the mutation factor (f_μ) and crossover rate (cr), during the optimization process.
   - The mutation factor is sampled from a Cauchy distribution, while the crossover rate is sampled from a normal distribution.
   - Both parameters are updated adaptively based on the success of previous generations, which allows the algorithm to adjust its behavior to the problem landscape.

2. **Greedy Selection for Mutation:**
    - The algorithm selects mutation targets from the top $p\%$ of the population, favoring solutions that have shown better performance in previous iterations.
    - This greedy selection helps to focus the search on the most promising solutions, increasing exploitation of the search space.
3. **Use of Archive for Diversity:**
    - ADE introduces an archive to store inferior solutions, which helps enhance diversity within the population.
    - By combining the current population with archived solutions, the algorithm explores different regions of the search space, preventing premature convergence.
4. **Balance Between Exploration and Exploitation:**
    - The algorithm maintains a balance between exploration and exploitation by combining top individuals with archived solutions.
    - This approach ensures that the algorithm does not get stuck in local optima and can explore more diverse areas of the solution space.
5. **Adaptive Parameter Updates:**
    - The mutation factor (f_μ) and the crossover rate (cr) are updated using the success of previous generations.
    - The updates are controlled by a learning rate, c, which ensures that successful values from the current generation influence the parameters for the next generation.
6. **Improved Convergence and Robustness:**
    - By adapting its parameters and maintaining population diversity, ADE improves convergence speed and robustness in complex optimization problems.
    - The dynamic adjustment allows ADE to better navigate the problem landscape, improving the quality of the solutions found.

Algorithm 2 Adaptive Differential Evolution (ADE)

```
1: Initialize population $X = \{x_1, x_2, \ldots, x_n\}$ randomly within bounds
2: Evaluate fitness $y_i = f(x_i)$ for each individual in population $X$
3: Initialize archive $A = \emptyset$ to store inferior solutions
4: Set initial values: $n_\mu = 0.5$, median $= 0.5$, greediness parameter $p = 0.05$
5: while termination condition not met do
6:     for each individual $x_i$ in population $X$ do
7:         Mutation:
8:         Draw mutation factor $f_\mu$ from a Cauchy distribution with location $median$ and scale 0.1
9:         Select the best $p\%$ individuals and combine them with the archive $A$
10:        Select three random individuals $x_{r1}, x_{r2}, x_{r3}$ from the combined population
11:        Compute mutation vector $x_{\mu_i} = x_i + f_{\mu_i} \cdot (x_{best} - x_i) + f_{\mu_i} \cdot (x_{r1} - x_{r2})$
12:        Crossover:
13:        Draw crossover rate $CR$ from a normal distribution with mean $n_\mu$
14:        Perform crossover on individual $x_{\mu_i}$ with probability determined by $CR$ for each dimension
15:        Selection:
16:        Evaluate fitness $y_{\mu_i} = f(x_{\mu_i})$
17:        if $y_{\mu_i} < y_i$ then
18:            Update $x_i \leftarrow x_{\mu_i}$ and $y_i \leftarrow y_{\mu_i}$
19:            Add $x_i$ to archive $A$
20:        end if
21:        Update $n_\mu$ and $median$ adaptively based on successful mutation and crossover rates
22:    end for
23:    Trim archive size if necessary (limit to population size)
24: end while
25: Return the best solution $x^*$ and its fitness $f(x^*)$
```

Covariance Matrix Adaptation Evolution Strategy

CMAES (Covariance Matrix Adaptation Evolution Strategy) is an advanced evolutionary optimization algorithm designed to solve continuous optimization problems. It adapts the search distribution over time, improving its ability to explore and exploit the search space. The key idea of CMAES is to maintain and evolve a multivariate Gaussian distribution over the candidate solutions, adjusting its mean and covariance matrix to guide the search towards optimal solutions. The main components of the CMAES algorithm are:

1. **Initialization:** The population is initialized with candidate solutions sampled from a multivariate normal distribution centered around the initial mean. The covariance matrix is initially set to the identity matrix, and the step-size is initialized. The population is evaluated, and the best solution is recorded.
2. **Sample Generation:** New candidate solutions (mutants) are generated by sampling from a multivariate normal distribution centered around the current mean, scaled by the current step-size and covariance matrix. This allows the algorithm to explore the search space based on the distribution of the population.
3. **Selection:** The candidate solutions (mutants) are evaluated, and the best solutions are selected based on their fitness values. The selected solutions form a set of parents for the next generation. The algorithm uses a weighted average of these parents to update the mean solution.
4. **Adaptation:** The covariance matrix is adapted based on the selected parents, allowing the algorithm to shape the search distribution according to the search space's local structure. The step-size is also updated adaptively based on the success of the search.
5. **Termination:** The algorithm iterates through the above steps until a termination condition is met, such as reaching the maximum number of evaluations or achieving an acceptable fitness level.

Algorithm 3 Covariance Matrix Adaptation Evolution Strategy (CMAES)

```
 1: Input: Problem dimension $n$, population size $n$, initial mean $\mu$, initial step-size $\sigma$
 2: Compute number of parents $\mu_{parents} = \lfloor n/2 \rfloor$
 3: Compute weight vector $w_a[i] = \log\left(\frac{n+1}{2}\right) - \log(i+1)$ for $i = 0, \ldots, n-1$
 4: Compute effective number of parents $\mu_{eff} = \frac{\left(\sum_{i=1}^{\mu_{parents}} w_a[i]\right)^2}{\sum_{i=1}^{\mu_{parents}} w_a[i]^2}$
 5: Compute adaptive strategy parameters:
 6: $c_s = \frac{\mu_{eff}+2}{n+\mu_{eff}+5}$
 7: $d_\sigma = 1 + 2\max\left(0, \sqrt{\frac{\mu_{eff}-1}{n+1}} - 1\right) + c_s$
 8: $c_c = \frac{4+\mu_{eff}/n}{n+4+2\mu_{eff}/n}$
 9: $c_1 = \frac{2}{(n+1.3)^2+\mu_{eff}}$
10: $c_w = \min\left(1 - c_1, \frac{1/4+\mu_{eff}+1/\mu_{eff}-2}{(n+2)^2+\mu_{eff}/2}\right)$
11: Compute normalized weights:
12: $w[i] = \begin{cases} \frac{1}{\sum_{j \in \text{positive}} w_a[j]} w_a[i] & \text{if } w_a[i] \geq 0 \\ \frac{0.1}{\sum_{j \in \text{negative}} |w_a[j]|} w_a[i] & \text{otherwise} \end{cases}$
13: Normalization constants:
14: $p_{s,1} = 1 - c_s$
15: $p_{s,2} = \sqrt{c_s(2-c_s)\mu_{eff}}$
16: $p_{c,1} = 1 - c_c$
17: $p_{c,2} = \sqrt{c_c(2-c_c)\mu_{eff}}$
18: $\chi_{\text{eff}} = \sqrt{n}\left(1 - \frac{1}{4n} + \frac{1}{21n^2}\right)$
19: Initialize:
20: Population $X = \phi$
21: $p_s = 0$ (cumulative step-size vector)
22: $p_c = 0$ (evolution path)
23: Covariance matrix $C = I$
24: Compute eigendecomposition $C = e_{ve} \cdot \text{diag}(e_{va}) \cdot e_{ve}^T$
25: while termination condition not met do
26:     for $k = 1$ to $n$ do
27:         Sample $z \sim \mathcal{N}(0, I)$
28:         Compute search direction $d_k = e_{ve} \cdot \text{diag}(e_{va}^{1/2}) \cdot z$
29:         Generate candidate $x_k = \mu + \sigma \cdot d_k$
30:         Evaluate $f(x_k)$
31:         Update best solution if $f(x_k)$ improves
32:     end for
33:     Sort population by fitness
34:     Select top $\mu_{parents}$ solutions
35:     Update mean $\mu = \sum_{i=1}^{\mu_{parents}} w_i \cdot x_i$
36:     Update cumulative step-size adaptation
37:     $p_s = p_{s,1} \cdot p_s + p_{s,2} \cdot (C^{-1/2} \cdot (x_{\text{selected}} - \mu)/\sigma)$
38:     Adapt step-size
39:     $\sigma = \sigma \cdot \exp\left(\frac{c_s}{d_\sigma}\left(\frac{\|p_s\|}{\chi_{\text{eff}}} - 1\right)\right)$
40:     Compute rank-one update of covariance matrix
41:     Compute rank-$\mu$ update of covariance matrix
42:     Symmetrize and eigendecompose covariance matrix $C$
43: end while
44: Return best solution $x^*$
```

We also implemented a variant, FCMAES (Fast Covariance Matrix Adaptation Evolution Strategy), which introduces several key modifications to the standard CMAES algorithm to improve computational efficiency, particularly in high-dimensional optimization problems. The primary differences lie in its sampling strategy and adaptation mechanism. The pseudocode is given below -

Algorithm 4 Fast Covariance Matrix Adaptation Evolution Strategy (FCMAES)

```
1: Input: Problem dimension $n$, population size $n$, initial mean $\mu$, initial step-size $\sigma$
2: Compute weight vector $w_a[i] = \log\left(\frac{n+1}{2}\right) - \log(i+1)$
3: Compute normalized weights:
4: $w[i] = \begin{cases} \frac{1}{\sum_{j \in \text{positive}} w_a[j]} w_a[i] & \text{if } w_a[i] \geq 0 \\ \frac{0.1}{\sum_{j \in \text{negative}} |w_a[j]|} w_a[i] & \text{otherwise} \end{cases}$
5: Initialize algorithm-specific parameters:
6: $c = \frac{2}{n+5}$
7: $c_1 = \frac{1}{3\sqrt{n}+5}$
8: $c_s = 0.3$
9: $q^* = 0.27$
10: $d_s = 1.0$
11: $n_{steps} = n$
12: Precompute sampling constants:
13: $x_1 = 1 - c_1$
14: $x_2 = \sqrt{(1-c_1)c_1}$
15: $x_3 = \sqrt{c_1}$
16: Compute effective number of parents $\mu_{eff} = \frac{\left(\sum_{i=1}^{\mu_{parents}} w_a[i]\right)^2}{\sum_{i=1}^{\mu_{parents}} w_a[i]^2}$
17: Initialize:
18: Population $X =$
19: Evolution path $p = 0$
20: Historical paths $\hat{p} = 0$
21: Success tracking $s = 0$
22: while termination condition not met do
23:     if generation $< n$ then
24:         for $k = 1$ to $n$ do
25:             Sample $z \sim \mathcal{N}(0, I)$
26:             Generate candidate $x_k = \mu + \sigma \cdot z$
27:         end for
28:     else
29:         for $k = 1$ to $n$ do
30:             Sample $z_1, z_2, z_3 \sim \mathcal{N}(0, I)$
31:             Generate candidate $x_k = \mu + \sigma \cdot (x_1 \cdot z_1 + x_2 \cdot z_2 \cdot \hat{p}[k] + x_3 \cdot z_3 \cdot p)$
32:         end for
33:     end if
34:     Evaluate fitness of population
35:     Sort population by fitness
36:     Select top $\mu_{parents}$ solutions
37:     Update mean $\mu = \sum_{i=1}^{\mu_{parents}} w_i \cdot x_i$
38:     Update evolution path $p$
39:     Periodic update of historical paths $\hat{p}$
40:     Compute success metric $q$ based on solution ranks
41:     Adaptive step-size update:
42:     $s = (1 - c_s)s + c_s(q - q^*)$
43:     $\sigma = \sigma \cdot \exp(s/d_s)$
44: end while
45: Return best solution $x^*$
```

Unlike traditional CMAES, which uses a full covariance matrix update in each generation, FCMAES uses a more lightweight approach with simplified evolution paths and a modified sampling mechanism. Specifically, it introduces a hybrid sampling strategy where for the first m generations (where m is the problem dimension), individuals are sampled purely from a standard normal distribution, and subsequently, the sampling incorporates additional terms that adapt based on historical search directions. The step-size adaptation is also more dynamic, using a success-based rule that tracks the performance of solutions across generations. This approach can potentially reduce computational complexity from

O(n^3) in standard CMAES to approximately O(n^2), making it significantly faster for high-dimensional problems, with speedup factors ranging from 1.5x to 3x depending on the specific optimization landscape and problem dimensionality.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a nature-inspired optimization algorithm based on the social behavior of birds flocking or fish schooling. It is a population-based optimization method where individuals (called particles) explore the search space for optimal solutions. Each particle adjusts its position in the solution space according to its own experience and the experience of its neighbors.

Each particle in the swarm represents a potential solution to the optimization problem, and the algorithm evolves by updating the particles' positions and velocities over several iterations. The update is influenced by two factors:

1. The particle's own best-known position (its personal best).
2. The best-known position found by any particle in its neighborhood (or the entire swarm, depending on the topology used).

The main components of PSO are -

1. **Particle Representation:** Each particle represents a potential solution, characterized by a position x_i and a velocity v_i in the search space.
2. **Initialization:** The population of particles is randomly initialized within the problem's search space, and each particle is assigned an initial velocity.
3. **Velocity Update:** The velocity of each particle is updated using a combination of:
    - **Inertia term:** the influence of the particle's previous velocity.
    - **Cognitive term:** the attraction to the particle's personal best position.
    - **Social term:** the attraction to the global best (or neighborhood best).
4. **Position Update:** The position of each particle is updated based on the updated velocity. Position updates are often clipped to ensure particles remain within the boundaries of the search space.
5. **Fitness Evaluation:** The fitness of each particle is evaluated based on an objective function. If a particle's current position yields a better fitness value than its personal best, the personal best is updated.
6. **Personal Best Position:** Each particle stores its best position found so far, denoted by $p_x[i]$, and its corresponding fitness value $p_y[i]$.
7. **Global or Neighborhood Best Position:** The best position found by the entire swarm or the local neighborhood, which influences the movement of each particle.
8. **Convergence and Stagnation Handling:** The algorithm includes mechanisms to detect stagnation (no improvement after several generations) and ensures diversity in the population by resetting particles when necessary.
9. **Inertia and Acceleration Coefficients:**
    - **Inertia weight:** Controls the influence of the previous velocity on the current velocity, encouraging exploration or exploitation.
    - **Cognitive acceleration coefficient c_1:** Controls the influence of the particle's own best-known position.
    - **Social acceleration coefficient c_2:** Controls the influence of the best-known position of the swarm or neighborhood.

Algorithm 5 Particle Swarm Optimization (PSO)

```
1: Input: Problem dimension $n$, population size $n$, initial positions $x$, initial velocities $v$, max generations $max\_gens$, cognitive and social weights
2: Initialize particles' positions $x[i]$ and velocities $v[i]$ randomly
3: Initialize personal best positions $p_x[i]$ and personal best fitness values $p_y[i]$
4: Initialize global best position $x_g$ and fitness value $y_g$
5: while not converged or maximum generations reached do
6:     for each particle $i$ do
7:         Evaluate fitness $y[i] = f(x[i])$
8:         if $y[i] < p_y[i]$ then
9:             Update personal best: $p_x[i] = x[i]$, $p_y[i] = y[i]$
10:        end if
11:        if $y[i] < y_g$ then
12:            Update global best: $x_g = x[i]$, $y_g = y[i]$
13:        end if
14:    end for
15:    for each particle $i$ do
16:        Update velocity: $v[i] = \chi \cdot (w \cdot v[i] + c_1 \cdot r_1 \cdot (p_x[i] - x[i]) + c_2 \cdot r_2 \cdot (x_g - x[i]))$
17:        Update position: $x[i] = x[i] + v[i]$
18:        Apply boundary conditions (clipping) to $x[i]$
19:    end for
20:    Update inertia weight $w$, cognitive weight $c_1$, and social weight $c_2$
21: end while
22: Return: Best position $x_g$ and its corresponding fitness $y_g$
```

We also implement a variant, Iterative Particle Swarm Optimization or IPSO. IPSO is an enhancement to the standard PSO that introduces several important improvements, particularly in the areas of population growth, adaptation, and social learning.

1. **Population Growth**: Unlike standard PSO, where the swarm size remains fixed throughout the optimization process, IPSO dynamically adjusts the population. The algorithm increases the number of particles when certain conditions are met, such as achieving a high success rate over a defined number of generations. This helps the algorithm scale and adapt to complex problems.

2. **Adaptive Cognitive and Social Coefficients**: In IPSO, the cognitive and social acceleration coefficients $c1$ and $c2$ are not fixed. Instead, they evolve over time, decreasing and increasing respectively. This adaptation encourages a balance between exploration and exploitation as the algorithm progresses, fostering better convergence.

3. **Constriction and Velocity Updates**: IPSO uses a constriction coefficient χ to update particle velocities, which helps control the particles' movements more efficiently and prevents overshooting. This differs from the basic PSO velocity update formula, where the velocity is simply adjusted based on individual and group bests.

4. **Enhanced Social Learning**: The algorithm introduces vertical social learning, where particles that are further away from the global best position have their learning rate adjusted adaptively. This method of adapting the exploration of particles based on their distance from the best solution leads to more effective search behavior.

5. **Success Memory and Stagnation Handling**: IPSO tracks the success of past iterations with a memory of the best fitness values, enabling it to detect stagnation. If progress slows down, the algorithm may trigger a population expansion or other adaptation strategies to avoid getting stuck in local minima.

Algorithm 6 Iterative Particle Swarm Optimization (IPSO)

```
1: Input: Problem dimension $n$, swarm size $n\_individuals$, max generations $max\_gens$, constriction
   $\chi$, cognitive and social coefficients $c1$, $c2$
2: Initialize positions $x[i]$ using Latin Hypercube Sampling, velocities $v[i] = 0$
3: Evaluate fitness $y[i] = f(x[i])$ for each particle
4: Set initial personal best $p_x[i] = x[i]$, $p_y[i] = y[i]$
5: Set initial global best $x_g$, $y_g$
6: Initialize generation counter $n\_gens = 0$
7: while $n\_gens < max\_gens$ do
8:     for each particle $i$ do
9:         Evaluate fitness $y[i] = f(x[i])$
10:        if $y[i] < p_y[i]$ then
11:            Update personal best: $p_x[i] = x[i]$, $p_y[i] = y[i]$
12:        end if
13:        if $y[i] < y_g$ then
14:            Update global best: $x_g = x[i]$, $y_g = y[i]$
15:        end if
16:        Update velocity: $v[i] = \chi \cdot (v[i] + c1 \cdot r_1 \cdot (p_x[i] - x[i]) + c2 \cdot r_2 \cdot (p_x[best\_neighbor] - x[i]))$
17:        Update position: $x[i] = \text{clip}(x[i] + v[i], \text{lower\_bound}, \text{upper\_bound})$
18:        Evaluate fitness for updated position: $y[i] = f(x[i])$
19:    end for
20:    if population growth condition met then
21:        Generate new particle: $xx = \text{generate\_new\_particle}(p_x, p_y)$
22:        Add new particle to swarm: $x = \text{append}(x, xx)$, $y = \text{append}(y, yy)$
23:    end if
24:    Increment generation counter $n\_gens \leftarrow n\_gens + 1$
25: end while
26: Return: Global best $x_g$ and $y_g$
```

Accuracy and Runtime Analysis

For the comparisons, we have kept `n_individuals` = 100 and `max_evals` = 10000. We compared the algorithms on these following benchmark functions –

1. **Paraboloid (Sphere Function)**

$$f(x) = 10 * \sum_{i=1}^{n} x_i^2$$

2. **Rastrigin Function**

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

3. **Discus Function**

$$f(x) = 10^6 x_1^2 + \sum_{i=2}^{n} x_i^2$$

4. **Xin-She Yang Function**

$$f(x) = \left(\sum_{i=1}^{n} |x_i|\right) \cdot \exp\left(-\sum_{i=1}^{n} \sin(x_i^2)\right)$$

Around 10-15 more benchmarks are available in the source code.

1. Running Time Comparison of Various Algorithms

We did the following two visualizations:

1. Figure 1: Runtime comparison across algorithms (x-axis) for different benchmark functions, shown using colored lines for each benchmark.
2. Figure 2: Runtime comparison across benchmark functions (x-axis) for different algorithms, where lines represent algorithm-specific performance

From Figures 1 and 2, we conclude that

- **ADE** consistently exhibits the highest runtime.
- **FCMAES** and **SPSO** are the most efficient, achieving the lowest runtimes.
- **CDE** and **IPSO** show moderate performance, while **CMAES** is stable but slightly slower than FCMAES.
- **Paraboloid** and **Discus** benchmarks have the lowest runtimes.
- **Xin_She_Yang** imposes higher runtimes, particularly for IPSO and ADE.
- **Rastrigin** shows moderate but stable runtimes.

General Trends

- **FCMAES** and **SPSO** outperform other algorithms in terms of runtime efficiency.
- **ADE** consistently records the highest runtimes.
- Benchmark complexity (e.g., **Xin_She_Yang**) significantly affects runtime performance across algorithms.

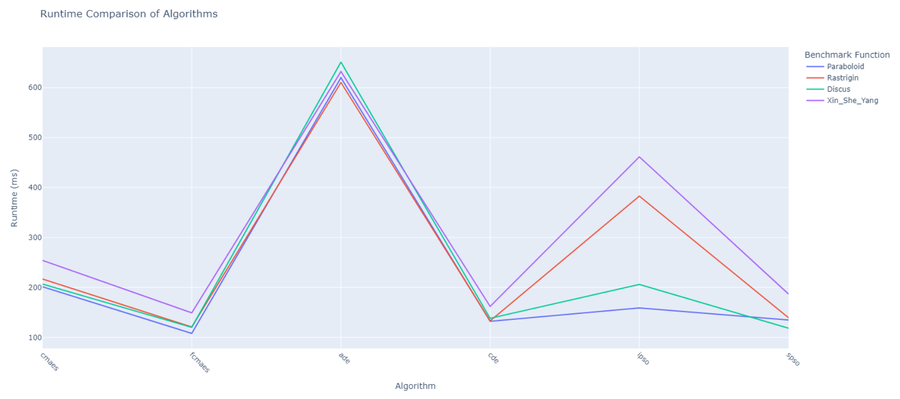

Figure 1: Running Time analysis of various algorithm (with axis as Benchmark functions)

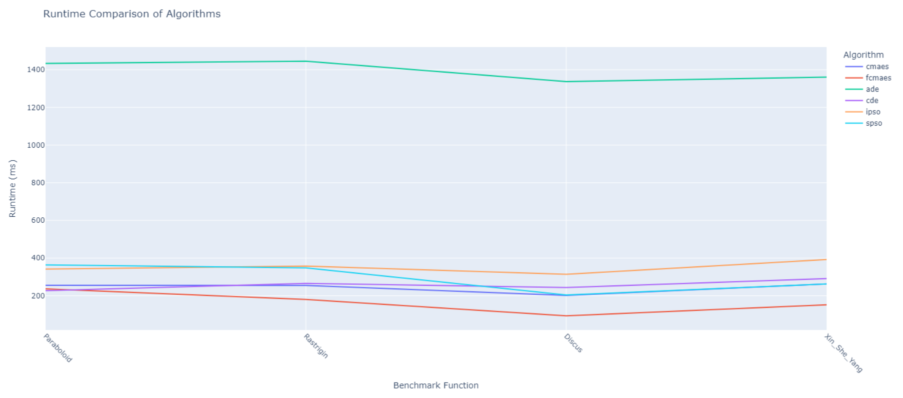

Figure 2: Running Time analysis of various algorithm (with axis as Algorithms)

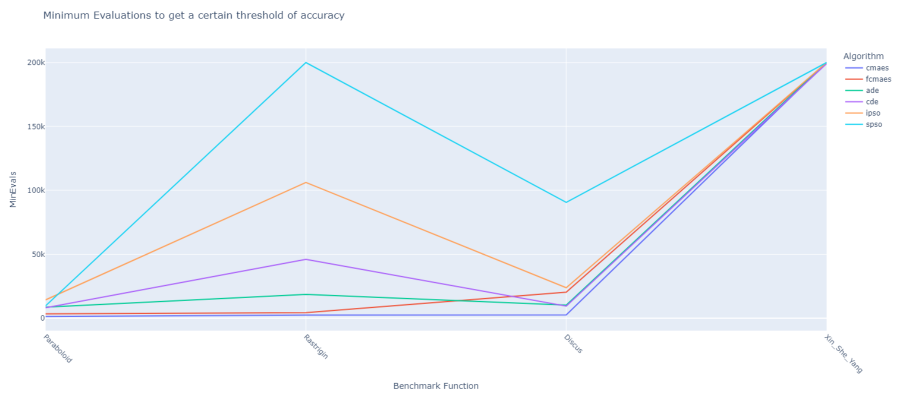

Figure 3: Min_evals to get threshold accuracy (with axis as Benchmark functions)

2. Minimum Evaluations Required to Achieve Threshold Accuracy

The graph (Figure 3) illustrates the minimum number of function evaluations needed by different algorithms to achieve a threshold accuracy of $|y_{\text{pred}} - y_{\text{true}}| \leq 10^{-3}$ for various benchmark functions. The minimum evaluations were determined using binary search with an upper limit $r = 2 \times 10^5$, as higher values proved computationally expensive. Key observations are as follows:

- **SPSO** consistently requires the highest number of evaluations, especially for the **Rastrigin** and **Xin_She_Yang** functions, where the evaluation count approaches the upper limit of 200,000.
- **IPSO** shows moderate performance but exhibits a sharp increase in evaluations for the **Xin_She_Yang** function.
- **CDE** maintains stable performance with relatively lower evaluations compared to SPSO and IPSO. However, it requires higher evaluations for the **Rastrigin** function.
- **FCMAES** and **CMAES** perform the best, requiring the lowest number of evaluations across most benchmark functions, highlighting their computational efficiency.
- **ADE** performs slightly less efficiently than FCMAES and CMAES but still requires fewer evaluations compared to SPSO and IPSO.
- For simpler functions such as **Paraboloid** and **Discus**, all algorithms require fewer evaluations. However, the **Xin_She_Yang** function imposes the highest computational burden on all algorithms.

In summary, **FCMAES** and **CMAES** are the most evaluation-efficient algorithms, while **SPSO** incurs the highest computational cost, particularly for complex benchmark functions.

Conclusion

We implemented three different kinds of evolutionary algorithms, **Differential Evolution**, **Covariance Matrix Adaptation Evolution Strategy**, and **Particle Swarm Optimization**, to find the optimum of an unknown function i.e. without gradient knowledge. We also implemented some variants of these, with added/improved features. We then benchmarked these algorithms on a suite of standard black box functions, which revealed some key insights into their performance in terms of both accuracy and runtime efficiency. Among the algorithms tested, FCMAES and CMAES demonstrated the best overall performance, consistently requiring fewer function evaluations and achieving high accuracy in the least amount of time. SPSO, while effective, tended to incur higher computational costs, especially for more complex functions. ADE, though slightly slower, still performed reasonably well, and CDE showed stable results across most benchmarks. Overall, all the algorithms provided a really robust and efficient way to optimize high-dimensional, non-convex, multimodal black-box functions with a finite budget of function evaluations.

Contributions

- IMT2022025 Varnit Mittal - Implemented CMAES and its variant.
- IMT2022075 Aditya Priyadarshi - Implemented PSO and its variant.
- IMT2022076 Mohit Naik - Implemented DE and its variant.

The rest of the code and documentation was done jointly.

References

[1] R. Storn and K. Price, “Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces,” 1997.

[2] J. Zhang and A. C. Sanderson, “Jade: Adaptive differential evolution with optional external archive,” 2009.

[3] N. Hansen, “The cma evolution strategy: A tutorial,” 2023.

[4] Z. Li, Q. Zhang, X. Lin, and H.-L. Zhen, “Fast covariance matrix adaptation for large-scale black-box optimization,” 2020.

[5] G. Venter and J. Sobieszczanski-Sobieski, “Particle swarm optimization,” 2003.

[6] M. A. Montes de Oca, T. Stutzle, K. Van den Enden, and M. Dorigo, “Incremental social learning in particle swarms,” 2011.

[7] C. Team, “The bbob (black box optimization benchmark) test suite,” 2024.