NUIST Experiment Report

Course name:	Embedded Script Development		
Experiment name:	The implementation of Crested Porcupine Optimizer in Python		
Date:	2024.11	**Tutor:** Chengzhi Qu	
College:	Waterford Institute	**Class:** IOT 1	
Name:	Ziheng Wang, Di Wu		
Student ID:	202283890020, 202283890005		

1. Experimental Division of Labor

The following outlines the individual contributions to this project:

- **Ziheng Wang**:

  Experimental Framework Design:

  Designed the experimental framework steps, ensuring that all components align with the research objectives.

  Report Writing:

  Wrote the report, organizing and explaining the methodology, results, and conclusions.

  Code Implementation:

  Implemented the code, translating the experimental framework into functional scripts.

  Data Organization:

  Organized the data, handling the collection, preprocessing, and structuring of datasets for analysis.

  Visualization:

  Developed various visualizations to present the experimental results, including convergence curves, Pareto fronts, and population distribution plots.

- **Di Wu**:

  Experimental Execution:

  Conducted specific experimental tasks, adjusting parameters as needed to adhere to the framework.

  Code Implementation:

  Contributed to code refinement and testing, ensuring functionality and reliability.

  Report Writing:

  Wrote the report, organizing and explaining the code, demand, and conclusions.

  Resources Finding:

  Searching for online resources, building and testing function environments

2. Experimental Content

2.1 Method introduction

Optimization is the process of finding the best solution to a problem by maximizing or minimizing an objective function [1]. Many real-world problems have large, complex search spaces with multiple local optima, making them difficult to solve with traditional methods [2]. Heuristic and metaheuristic algorithms, inspired by natural processes, have emerged to tackle these challenges [3]. Examples include Genetic Algorithms (GA) [4], Particle Swarm Optimization (PSO) [5], and Ant Colony Optimization (ACO) [6], which balance exploration and exploitation in the search space.

The Crested Porcupine Optimizer (CPO) [7] is a metaheuristic inspired by the defensive behaviors of crested porcupines, using sight, sound, odor, and physical attack defenses. In the exploration phase, CPO uses sight and sound mechanisms to search broadly and avoid local minima. As the search progresses, CPO shifts to an exploitation phase, using odor and physical attack mechanisms to refine the search around promising solutions.

A unique feature of CPO is its cyclic population reduction technique, where the population size is selectively reduced to focus resources on the best candidate solutions. This helps maintain diversity and prevents premature convergence. CPO's balance of exploration and exploitation makes it effective for complex optimization problems. Benchmark tests on functions like CEC2017 have shown CPO's competitive performance, suggesting potential for various applications.

In this report, we use Python to reproduce and validate the CPO algorithm on CEC2017 and CEC2022 benchmark functions, assessing its performance consistency. Our work involves analyzing the principles of CPO to understand its phases and mechanisms, as well as testing it in new scenarios to explore its broader applicability. In addition to benchmark testing, we also conduct a detailed Pareto analysis for multi-objective optimization problems, examining CPO's ability to balance conflicting objectives. Furthermore, we perform a comparative analysis of CPO's performance against other common optimization algorithms, including PSO, GA, GWO, and WOA, to evaluate its effectiveness in diverse optimization landscapes. We also integrate CPO into path planning tasks, creating a hybrid A-CPO algorithm that combines A* and CPO for efficient route finding in grid-based environments. The code repository of reproduced CPO can be accessed at: https://github.com/Nickory/CPO-python.

2.2 Formula Introduction

Optimization algorithms aim to find the best solution by maximizing or minimizing an objective function, often in complex, high-dimensional search spaces with multiple local optima. Traditional optimization methods, like gradient-based approaches, often fail in these cases as they are prone to becoming trapped in local optima. To address this, metaheuristic algorithms introduce randomness and heuristic strategies that allow for a more global search. Metaheuristic algorithms follows core mathematical principles as follows:

1. **Objective Function**: Given an objective function $f(X)$ where $X \in \mathbb{R}^d$ represents a candidate solution in a d-dimensional search space, the goal is to find X^* such that:

$$X^* = \arg\min_X f(X) \quad \text{or} \quad X^* = \arg\max_X f(X)$$

2. **Population-Based Search**: Metaheuristics often use a population $\{X_i\}_{i=1}^N$ of N candidate solutions to explore the search space. Each candidate X_i is evaluated based on the objective function $f(X_i)$, with better candidates having higher fitness values.

3. **Exploration and Exploitation**: These algorithms balance exploration (searching broadly across the space to avoid local optima) and exploitation (focusing on promising regions to refine solutions). This balance is often achieved through probabilistic update rules and adaptive strategies.

The Crested Porcupine Optimizer (CPO) is a novel metaheuristic inspired by the defensive behaviors of crested porcupines, employing four mechanisms: sight, sound, odor, and physical attack, that are mathematically designed to capture both exploration and exploitation. Below, we detail each phase of CPO and our understanding of how these behaviors are translated into mathematical operations.

2.2.1 Exploration Phase: Sight and Sound Mechanisms

In the exploration phase, CPO employs sight and sound mechanisms to introduce randomness and diversity in the search process, helping the algorithm avoid premature convergence.

Sight Mechanism: The sight mechanism introduces a random movement in a chosen direction, encouraging candidates to explore new regions. This is represented by:

$$X_i^{\text{new}} = X_i + \alpha \cdot R \cdot D, \tag{1}$$

where α is a scaling factor that controls the step size. R is a random vector sampled uniformly in $\mathbb{R}^d$, providing a direction. D is a distance factor, which may be either a constant or dynamically adjusted based on the current solution's position.

This mechanism allows CPO to conduct a broad, unbiased search in various directions. By choosing R as a random vector, algorithm ensures that each candidate has a unique path, enhancing diversity in early stages.

Sound Mechanism: The sound mechanism further diversifies the search by introducing a social component, encouraging candidate solutions to interact. It is expressed as:

$$X_i^{\text{new}} = X_i + \beta \cdot (X_j - X_i) + \gamma \cdot R, \tag{2}$$

where β controls the attraction towards a randomly selected candidate X_j, representing social interaction. γ is another scaling factor to adjust randomness. R is an additional random vector, introducing stochasticity.

The sound mechanism simulates a form of communication among candidates, allowing them to avoid regions explored by others. This cooperative behavior broadens the search range, making it less likely to settle in suboptimal regions.

2.2.2 Exploitation Phase: Odor and Physical Attack Mechanisms

In the exploitation phase, CPO shifts focus to promising regions using odor and physical attack mechanisms. These strategies concentrate the search around the best solutions found, refining the search for optimal results.

Odor Mechanism: This mechanism guides candidate solutions towards the best-known solution, focusing the search without entirely removing randomness:

$$X_i^{\text{new}} = X_i + \delta \cdot (X_{\text{best}} - X_i) + \epsilon \cdot R, \tag{3}$$

where δ is a scaling factor that controls attraction strength towards X_{best}, the best solution found so far. ϵ adds a small degree of randomness. R is a random vector to avoid exact convergence, retaining some diversity.

Our understanding is that this mechanism allows CPO to refine its search around promising areas while still permitting minor exploration, preventing overcommitment to a potentially suboptimal region.

Physical Attack Mechanism: The final adjustment is a precise search around X_{best}, narrowing the search radius to converge on optimal solutions:

$$X_i^{\text{new}} = X_{\text{best}} + \zeta \cdot (X_{\text{best}} - X_i), \tag{4}$$

where ζ is a small scaling factor, creating a tight convergence around X_{best}.

This mechanism is akin to a final "fine-tuning" step, concentrating efforts on the most promising solution, akin to a local search. This focused exploitation ensures that CPO can converge with high precision, finalizing the optimization process.

2.2.3 Cyclic Population Reduction Technique

A distinctive feature of CPO is its cyclic population reduction technique. Inspired by the fact that not all porcupines engage defenses simultaneously, this mechanism dynamically reduces the population size, focusing computational resources on top-performing candidates:

$$N_t = N_0 \cdot \left(1 - \frac{t}{T_{\max}}\right), \tag{5}$$

where N_0 is the initial population size. $T_{\max}$ is the maximum number of iterations. N_t is the population size at iteration t.

This gradual reduction ensures that as the algorithm progresses, the focus shifts towards the best solutions, accelerating convergence. Our interpretation is that this selective pressure prevents premature convergence by retaining enough diversity early on, while later focusing computational effort on the best candidates.

3. Experimental Result

3.1 Experiments on Reproduced CPO Using Provided Test Functions

To evaluate the performance of the reproduced CPO, we conducted experiments on a set of well-known mathematical benchmark functions. Each function is defined as follows:

Table 1: Common Test Functions Used for CPO Evaluation

Name	Function
Sphere	$F_1(x) = \sum_{i=1}^{D} x_i^2$
Schwefel's 2.22	$F_2(x) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$
Powell Sum	$F_3(x) = \sum_{i=1}^{D} \lvert x_i \rvert^{i+1}$
Schwefel's 1.2	$F_4(x) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j\right)^2$
Schwefel's 2.21	$F_5(x) = \max\left(\lvert x_i \rvert, 1 \le i \le D\right)$
Rosenbrock	$F_6(x) = \sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$
Step	$F_7(x) = \sum_{i=1}^{D} (x_i + 0.5)^2$
Quartic	$F_8(x) = \sum_{i=1}^{D} i x_i^4 + \text{random}[0, 1]$
Zakharov	$F_9(x) = \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5 i x_i\right)^2 + \left(\sum_{i=1}^{D} 0.5 i x_i\right)^4$

Function Descriptions:

- **Sphere Function** F_1: A simple, continuous, unimodal function often used as a baseline for optimization.
- **Schwefel's 2.22** F_2: Combines the sum and product of absolute values, making it sensitive to changes in all variables.
- **Powell Sum** F_3: Known for its complex landscape, with increasing exponents for each variable.
- **Schwefel's 1.2** F_4: A cumulative sum function that tests an algorithm's handling of non-separable problems.
- **Schwefel's 2.21** F_5: Takes the maximum absolute value, testing robustness to high-magnitude variables.
- **Rosenbrock** F_6: A classic test for non-convex optimization, often forming a narrow valley for algorithms to explore.
- **Step Function** F_7: A step-based function introducing discontinuities, making optimization challenging.
- **Quartic** F_8: Incorporates random noise, testing an algorithm's resilience to stochastic variations.
- **Zakharov** F_9: Combines quadratic, linear, and quartic terms, creating a complex, multimodal landscape.

Each of these functions is evaluated over a specified dimensionality D, with multiple runs to assess the robustness and stability of the reproduced CPO algorithm. The performance metrics include best fitness, average fitness, and standard deviation across runs.

Table 2: Evaluation Results for Each Function (500epoch)

Function	Final Best Fitness	Mean Diversity	Mean Avg Fitness	Trajectory 1st Dim	Final Solution (Best Position)				
					Pos 1	Pos 2	Pos 3	Pos 4	Pos 5
Sphere	2.68×10^{-29}	2.07e+02	5.16e+02	1.34e-15	1.34e-15	-1.36e-15	-3.75e-15	-8.24e-16	8.50e-16
Schwefel's 2.22	2.61×10^{-14}	1.22e+15	3.87e+14	-1.26e-15	-1.26e-15	-9.59e-16	-4.64e-15	2.57e-15	-5.34e-16
Powell Sum	4.63×10^{-30}	2.86e+16	8.71e+15	-6.69e-32	-6.69e-32	-2.04e-15	-2.17e-12	1.60e-09	1.66e-07
Schwefel's 1.2	2.35×10^{-17}	1.47e+03	1.64e+03	2.25e-11	2.25e-11	-1.96e-09	-7.30e-10	1.70e-09	2.01e-09
Schwefel's 2.21	2.10×10^{-11}	1.00e+00	4.65e+00	9.69e-12	9.69e-12	1.26e-11	2.84e-12	6.79e-12	2.27e-12
Rosenbrock	5.91	1.52e+08	2.15e+08	8.61e-01	8.61e-01	7.28e-01	5.19e-01	2.62e-01	4.09e-02
Step	1.33×10^{-4}	2.41e+02	5.27e+02	-5.06e-01	-0.51	-0.50	-0.50	-0.50	-0.50
Quartic	2.25×10^{-3}	8.91e+06	1.48e+07	7.99e-02	0.08	-0.03	-0.08	-0.09	-0.07
Zakharov	2.67×10^{-14}	5.72e+09	1.53e+09	-8.30e-08	-8.30e-08	6.42e-08	-3.07e-08	-6.54e-08	4.64e-08

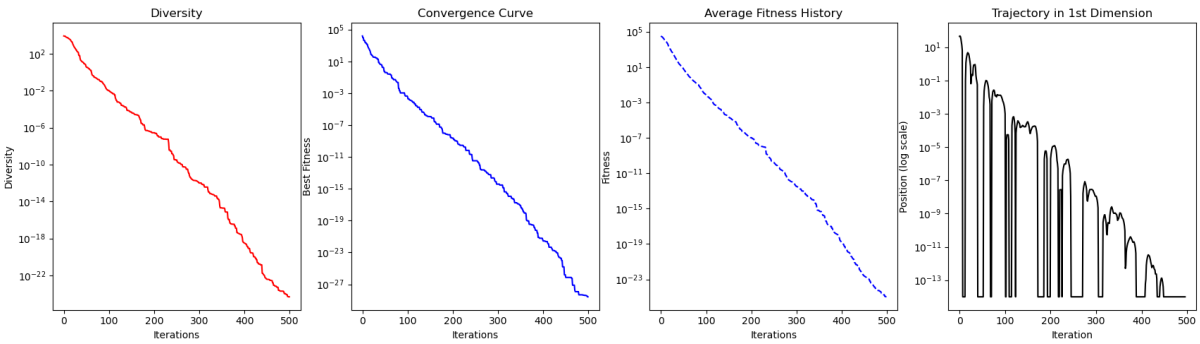
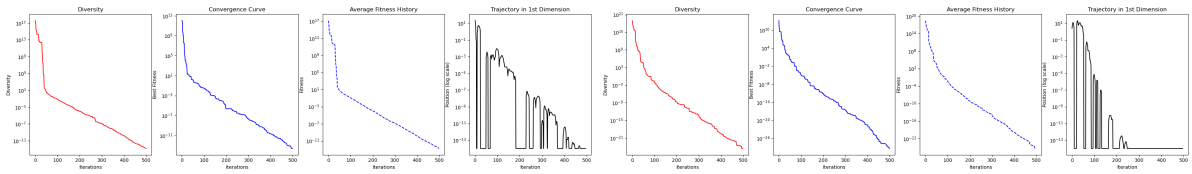


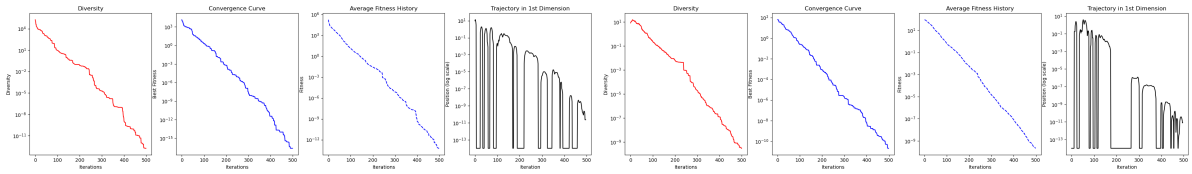
Figure 1: Function: F1 (Sphere)



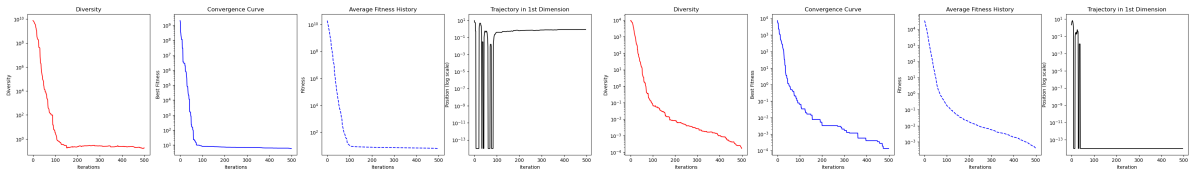
(a) Function: F2 (Schwefel's 2.22) (b) Function: F3 (Powell Sum)

Figure 2: Function: F2-F3 (Schwefel's 2.22 and Powell Sum)

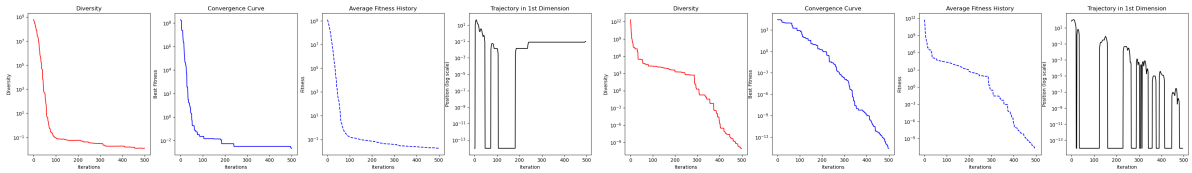
The results summarized in Table 2 offer a comprehensive view of the reproduced CPO performance across various test functions, showcasing metrics such as Final Best Fitness, Mean Diversity, Mean Average Fitness, and Trajectory in the 1st Dimension. These metrics reveal differences in the algorithm's effectiveness, particularly evident in the Final Best Fitness values.



(a) Function: F4 (Schwefel's 1.2) (b) Function: F5 (Schwefel's 2.21)



(c) Function: F6 (Rosenbrock) (d) Function: F7 (Step)



(e) Function: F8 (Quartic) (f) Function: F9 (Zakharov)

Figure 3: Performance for test functions F4-F9.

Functions like Schwefel's 2.22 and Powell Sum exhibit high diversity values (e.g., 1.22×10^{15} and 2.86×10^{16}), indicating a broad search, suitable for functions with complex landscapes. In contrast, the low diversity in Schwefel's 2.21 suggests focused convergence in simpler landscapes. The Mean Average Fitness metric, especially high in functions like Powell Sum, reflects the average solution quality during optimization, where challenging functions see slower fitness improvements across the population. The Trajectory in the 1st Dimension captures the convergence path of one component of the solution. Functions with smooth landscapes, such as Sphere, show steady and direct convergence, while functions like Zakharov exhibit an initial erratic phase, indicating exploration before stabilizing as the solution nears optimality.

Figures 1 to 3f illustrate these trends with Diversity, Convergence Curve, Average Fitness History, and Trajectory in the 1st Dimension plots. The Diversity plots generally show a decline, indicating reduced search space variability as the algorithm converges. The Convergence Curves are smooth for most functions but display minor fluctuations in complex functions like Rosenbrock. Average Fitness History plots confirm rapid initial improvements, especially for simpler landscapes, and the log-scale Trajectory plots highlight early exploration phases, followed by convergence.

As shown in Figures 4 to 12, the population distribution adapts over iterations, particularly across benchmark functions like Sphere, Schwefel's 2.22, and Powell Sum. Early iterations show a widely dispersed population, promoting exploration. This broad distribution reflects CPO's sight and sound

mechanisms, which prevent early convergence to local minima by covering diverse regions of the search space.

As the iterations progress, the population concentrates around promising regions, driven by CPO's exploitation mechanisms—odor and physical attack—illustrated in later stages, like in iteration 150 in Figure 6. This adaptive narrowing of the search space allows CPO to focus on local exploitation while still considering global optima, guiding the population effectively toward the global optimum.

CPO's cyclic reduction mechanism ensures that only the best individuals remain in the final convergence stages, focusing computational resources on refining solutions within the optimal region. This approach reduces computational cost and minimizes the risk of premature convergence, constantly renewing diversity while favoring individuals near the optimum. The consistent convergence patterns in Figures 4 through 9 demonstrate CPO's robustness across different landscapes.

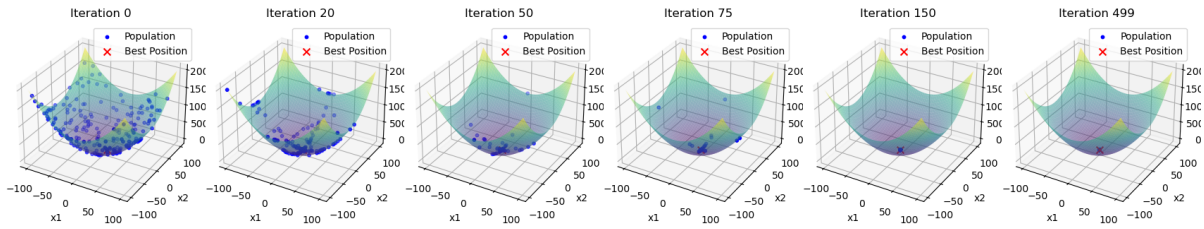


Figure 4: Population Evolution for Function: Sphere

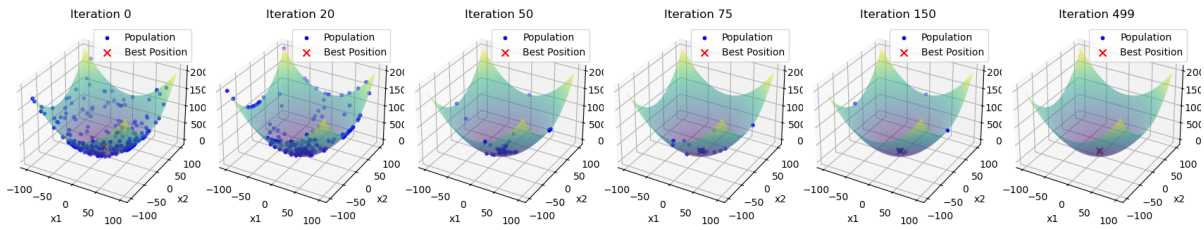


Figure 5: Population Evolution for Function: Schwefel's 2.22

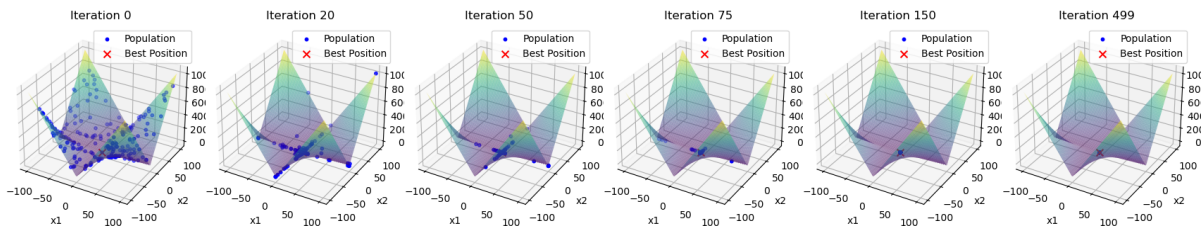


Figure 6: Population Evolution for Function: Powell Sum

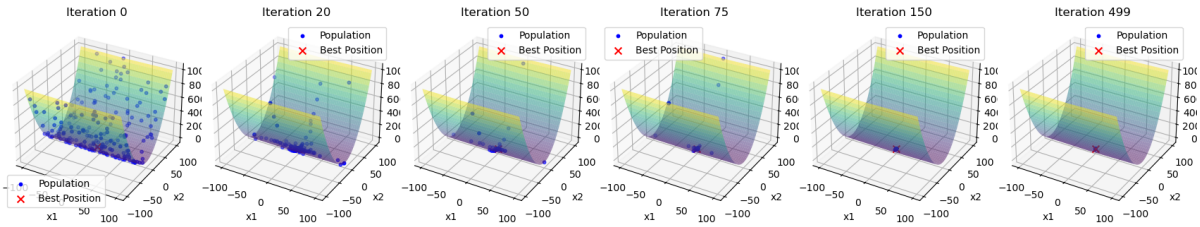


Figure 7: Population Evolution for Function: Schwefel's 1.2

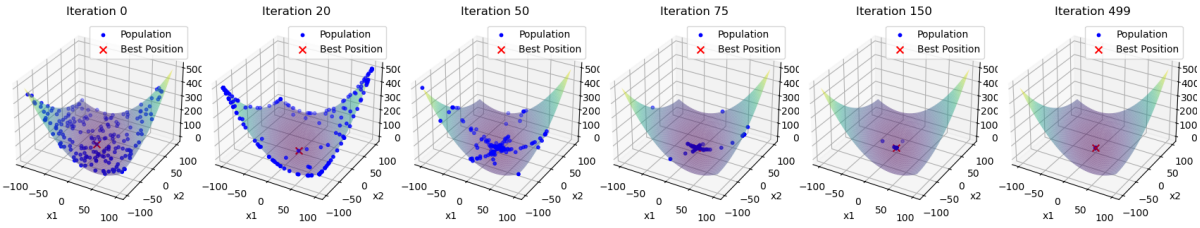


Figure 8: Population Evolution for Function: Schwefel's 2.21

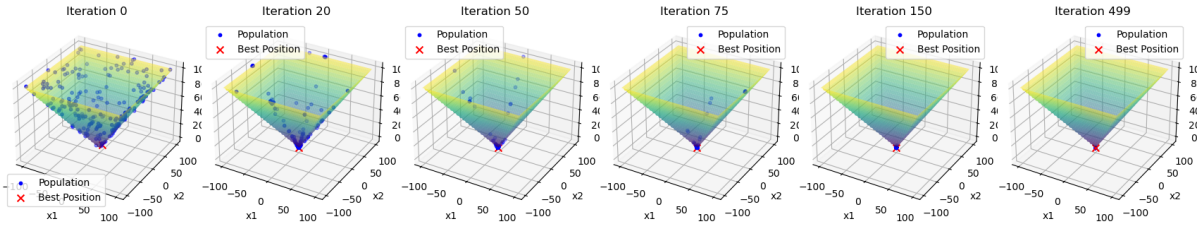


Figure 9: Population Evolution for Function: Rosenbrock

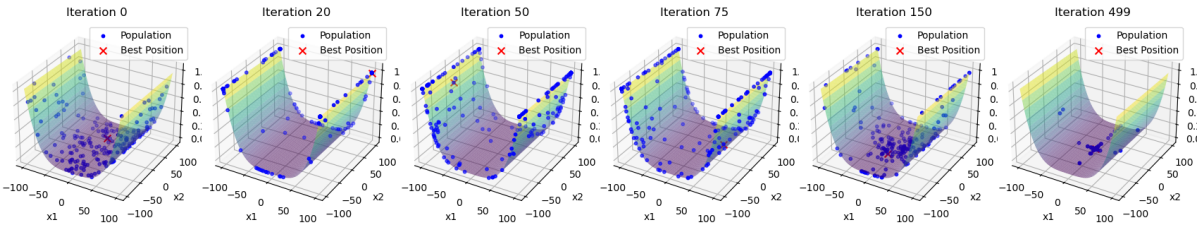


Figure 10: Population Evolution for Function: Step

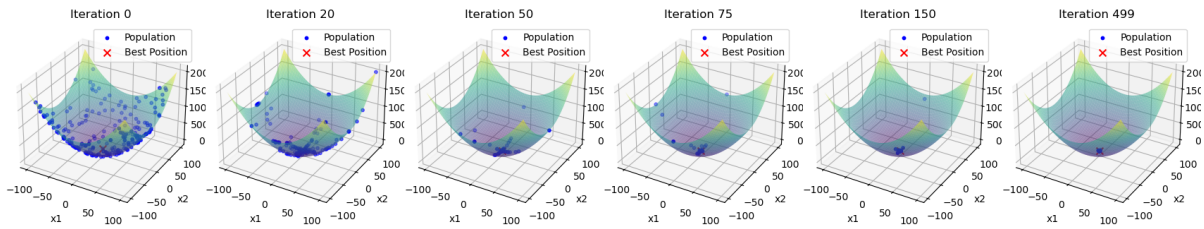


Figure 11: Population Evolution for Function: Quartic

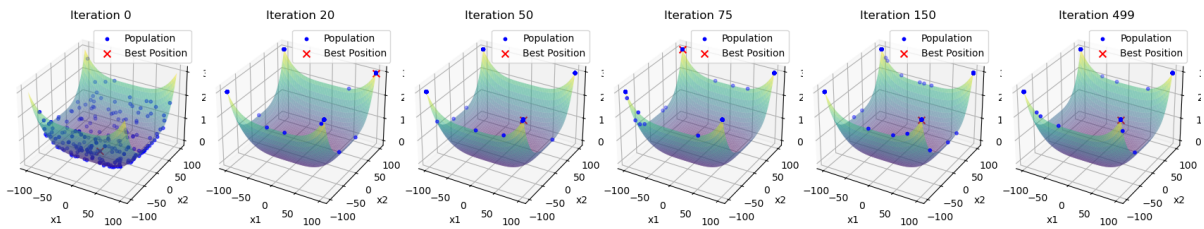


Figure 12: Population Evolution for Function: Zakharov

3.2 Experiments on Reproduced CPO Using CEC Benchmark

The Reproduced CPO algorithm was evaluated on both CEC2017 [8] and CEC2022 [9] benchmark functions, as shown in Tables 3 and 4. These tables compare the original and reproduced CPO values along with error percentages for each function.

For CEC2017 (Table 3), the reproduced CPO shows minimal deviation from the original values in simpler functions, such as F1, F2, and F3, with error percentages as low as 0.0036%. However, for more complex functions, the discrepancies increase, reaching up to -17.92% in some cases. In the CEC2022 benchmark (Table 4), the reproduced CPO achieves similar accuracy, with zero error in several functions (F1, F3, F5, F9, and F11). However, moderate errors appear in functions like F2 and F4, at 2.20% and 1.23%, respectively.

While there are some instances of larger error percentages for certain functions, the reproduced CPO demonstrates strong performance in replicating the original CPO values across the CEC2017 and CEC2022 test functions, with deviations mostly in the range of a few percent. This suggests that the algorithm has been successfully reproduced, though further tuning may be required for handling certain complex objective functions more effectively.

3.3 Pareto Analysis for Multi-objective Optimization in Reproduced CPO

In this experiment, we test reproduced CPO on a set of multi-objective optimization problems. Each problem is formulated with a pair of objective functions, representing different mathematical properties and complexities. The objective of the algorithm is to find a set of balanced solutions between the two conflicting goals, resulting in a Pareto front that illustrates the trade-offs. A general multi-objective optimization problem can be formulated as

$$\min_{x \in \mathbb{R}^D} \; (f_1(x), f_2(x), \ldots, f_k(x)),$$

where $f_i(x)$ represents the i-th objective function, x is a D-dimensional decision variable vector, and k is

Table 3: Comparison of Original and Reproduced CPO Values with Error Percentage for CEC2017 Test Functions

Metric	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Original CPO	100.0000	200.0000	300.0000	400.0000	500.2388	710.9723	800.0000	900.0000	1011.7270	1100.0000
Reproduced CPO	100.0036	200.0000	300.0000	400.3711	500.9950	600.0000	712.5304	802.9849	900.0000	1143.6156
Error (%)	0.00%	0.00%	0.00%	0.09%	0.15%	-15.63%	-10.93%	-10.79%	-11.04%	3.97%
Metric	**F11**	**F12**	**F13**	**F14**	**F15**	**F16**	**F17**	**F18**	**F19**	**F20**
Original CPO	1200.0833	1300.0000	1400.0000	1500.0000	1600.0343	1743.3858	1954.8837	1910.8780	2079.9351	2350.6037
Reproduced CPO	1100.9950	1212.6762	1310.8130	1401.9929	1502.0772	1638.7717	1638.7717	1910.8780	1803.9942	2300.0000
Error (%)	-8.25%	-6.71%	-6.37%	-6.53%	-6.12%	-6.00%	-16.17%	0.00%	-13.25%	-2.15%
Metric	**F21**	**F22**	**F23**	**F24**	**F25**	**F26**	**F27**	**F28**	**F29**	
Original CPO	2300.4920	2684.7308	2867.3000	2887.0853	3840.5817	3217.6690	3100.0000	3325.4878	6281.3677	
Reproduced CPO	1901.3543	2203.5328	2605.6354	2500.0000	2897.7624	3089.5179	3089.5179	3089.5179	5524.8671	
Error (%)	-17.36%	-17.92%	-9.12%	-13.42%	-24.58%	-3.99%	-0.34%	-7.09%	-12.03%	

Table 4: Comparison of Original and Reproduced CPO Values with Error Percentage for CEC2022 Test Functions

Metric	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Original CPO	300.0000	400.1330	600.0000	803.0180	900.0000	1800.0460	2000.0000	2200.0350	2529.2840	2500.1600
Reproduced CPO	300.0000	408.9161	600.0000	812.9345	900.0000	1801.6718	2020.0000	2220.0099	2529.2844	2500.1776
Error (%)	0.00%	2.20%	0.00%	1.23%	0.00%	0.09%	1.00%	0.91%	0.00%	0.00%
Metric	**F11**	**F12**								
Original CPO	2600.0000	2864.2310								
Reproduced CPO	2600.0000	2863.8777								
Error (%)	0.00%	-0.01%								

the number of objectives. Optimization algorithm seeks to minimize each $f_i(x)$ independently without simply summing them, thus finding a set of solutions that balance each objective.

For this experiment, four pairs of objective functions were selected, each representing a different multi-objective optimization problem. These are defined as follows:

Sphere and Schwefel's 2.22:

$$\min_{x\in\mathbb{R}^D} \ (f_1(x), f_2(x)) = \left(\sum_{i=1}^{D} x_i^2,\ \sum_{i=1}^{D} |x_i| + \prod_{i=1}^{D} |x_i| \right).$$

Rosenbrock and Zakharov:

$$\min_{x\in\mathbb{R}^D} \ (f_1(x), f_2(x)) = \left(\sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right],\ \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5 \cdot i \cdot x_i \right)^2 + \left(\sum_{i=1}^{D} 0.5 \cdot i \cdot x_i \right)^4 \right).$$

Quartic and Schwefel's 1.2:

$$\min_{x\in\mathbb{R}^D} \ (f_1(x), f_2(x)) = \left(\sum_{i=1}^{D} i \cdot x_i^4 + \text{random}[0,1],\ \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j \right)^2 \right).$$

Sphere, Schwefel's 2.22, and Quartic:

$$\min_{x\in\mathbb{R}^D} \ (f_1(x), f_2(x), f_3(x)) = \left(\sum_{i=1}^{D} x_i^2,\ \sum_{i=1}^{D} |x_i| + \prod_{i=1}^{D} |x_i|,\ \sum_{i=1}^{D} i \cdot x_i^4 + \text{random}[0,1] \right).$$

Rosenbrock, Zakharov, and Step:

$$\min_{x\in\mathbb{R}^D} \ (f_1(x), f_2(x), f_3(x)) = \left(\sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right],\ \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5 \cdot i \cdot x_i \right)^2 + \left(\sum_{i=1}^{D} 0.5 \cdot i \cdot x_i \right)^4,\ \sum_{i=1}^{D} (x_i + 0.5)^2 \right).$$

Rosenbrock and Zakharov:

$$\min_{x\in\mathbb{R}^D}\ (f_1(x), f_2(x), f_3(x)) = \left(\sum_{i=1}^{D-1}\left[100(x_{i+1}-x_i^2)^2+(x_i-1)^2\right],\ \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5\cdot i\cdot x_i\right)^2 + \left(\sum_{i=1}^{D} 0.5\cdot i\cdot x_i\right)^4,\ \sum_{i=1}^{D}(x_i+0.5)^2\right).$$

The goal of this multi-objective optimization setup is to simultaneously minimize each objective pair. Rather than combining the objectives into a single scalar function, the algorithm searches for a set of balanced solutions, forming a Pareto front that illustrates the trade-offs between conflicting objectives. Each problem is run for up to 500 generations, and the resulting Pareto fronts are visualized to observe the progression of solutions over time.

Figure 13 and Table 5 collectively illustrate the performance and convergence behavior of the reproduced CPO algorithm across various multi-objective function pairs. The color gradient in the figures, progressing from yellow to purple, reflects the generation sequence, with yellow representing initial generations and purple indicating later ones. The evolution of the Pareto front demonstrates how the algorithm adapts to different objective characteristics over successive generations.

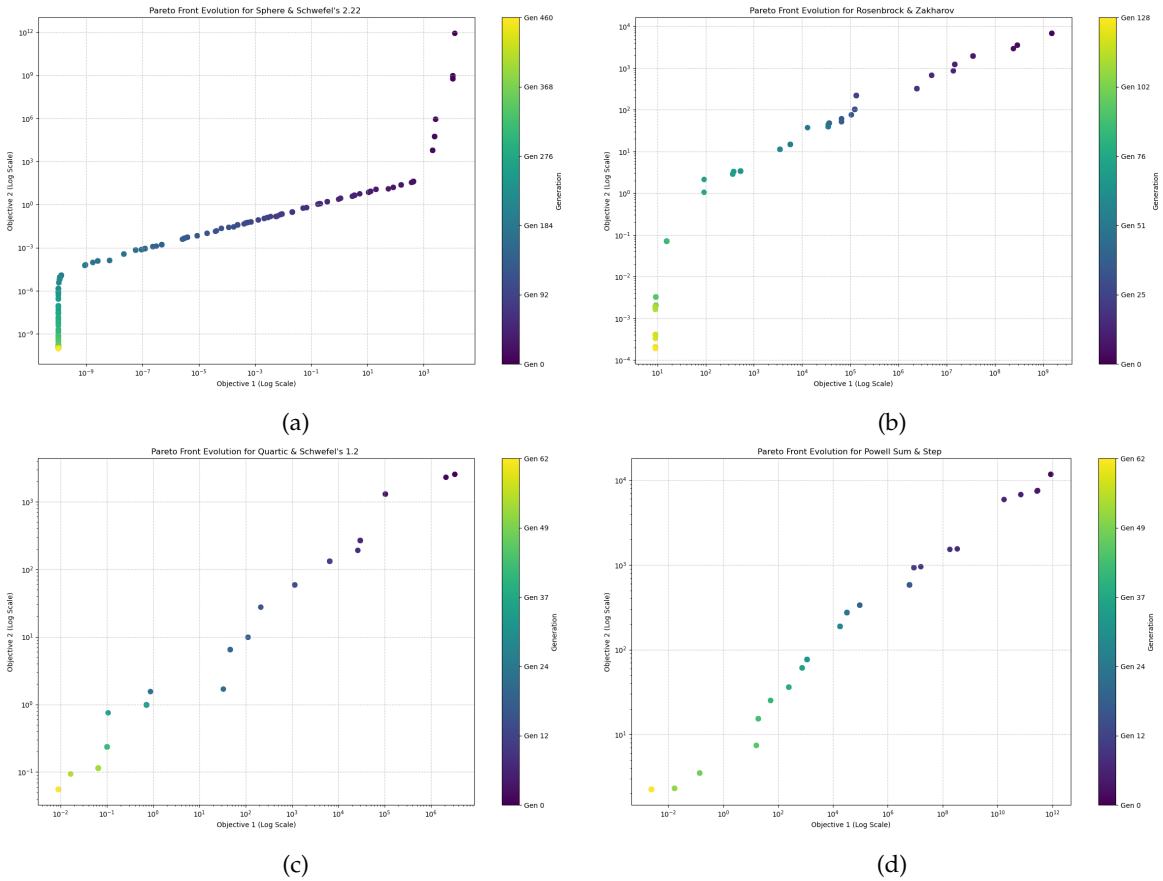
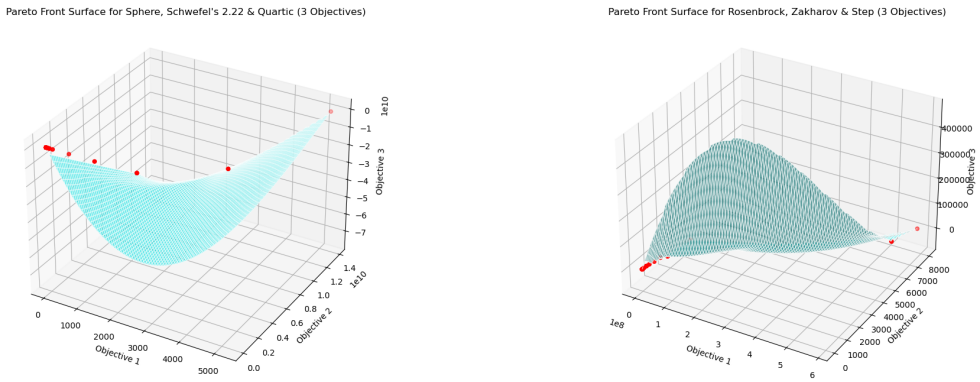


Figure 13: Pareto Front Evolution for Various Objective Pairs: (a) Powell Sum & Step, (b) Sphere & Schwefel's 2.22, (c) Rosenbrock & Zakharov, and (d) Quartic & Schwefel's 1.2.

In the Sphere and Schwefel's 2.22 optimization (Figure 13a), the Pareto front extends progressively along both axes. Table 5 shows a significant reduction in both objectives' values by the final generation, indicating efficient convergence. The Sphere function's smooth convex landscape is quickly exploited by the algorithm, while the more complex, non-linear Schwefel's 2.22 objective slows down the convergence, leading to a well-spread Pareto front. For the Rosenbrock and Zakharov pair (Figure 13b), convergence is notably slower. The initial values in Table 5 are high for both objectives, with gradual improvement across generations. This slow progress reflects the complexity of the Rosenbrock function's narrow valley and the higher-order polynomial terms in Zakharov. By the final generation, the Pareto front shows incremental gains rather than rapid improvements, as highlighted in both Figure 13b and Table 5, where final fitness values are significantly reduced but still reflect the difficult optimization landscape. In the Quartic and Schwefel's 1.2 case (Figure 13c), early generations exhibit scattered points



(a) (b)

Figure 14: Pareto Front Surfaces for Different Objective Sets: (a) Rosenbrock, Zakharov & Step, (b) Sphere, Schwefel's 2.22 & Quartic.

along the Pareto front, which becomes more focused over time. The variability in initial generations, as shown in Table 5, aligns with the stochastic properties of the Quartic function. The final generation data in the table demonstrates convergence toward stable solutions, and the clustering in the Pareto front indicates the algorithm's adaptability in noisy environments. This case highlights the algorithm's robustness in managing objectives with inherent randomness, converging effectively despite initial instability. For the Powell Sum and Step function pair (Figure 13d), the Pareto front initially covers a broad range but becomes more refined in later generations. The Step function's discrete characteristics are evident in the stepped appearance of the Pareto front, which is captured well in both Figure 13d and the final values in Table 5. The discrete steps of the Step function, combined with the continuous Powell Sum, challenge the algorithm to balance exploration and exploitation. The convergence in both objectives is slower but steady, with the algorithm showing its ability to adapt to mixed continuous and discrete objectives.

For three-objective function pairs (Figure 14a and Figure 14b), the Pareto fronts reveal more intricate surfaces due to the added complexity of a third objective. In Figure 14a, the interaction among the Rosenbrock, Zakharov, and Step objectives results in a Pareto front that reflects the difficulty of optimizing across a narrow valley, polynomial terms, and discrete steps. This surface illustrates the algorithm's capacity to handle diverse functional landscapes simultaneously. In Figure 14b, the Sphere, Schwefel's 2.22, and Quartic functions yield a Pareto front that extends widely, indicating a diverse set of trade-offs. The initial generations scatter widely, while later generations focus on regions of improved convergence, demonstrating the CPO algorithm's ability to achieve a balanced optimization across smooth, complex, and stochastic landscapes. Table 6 further quantifies these findings, with performance metrics reflecting the convergence rates and final fitness values across all function pairs, including the new three-objective cases.

3.4 Comparative Analysis of Reproduced CPO with Common Optimization Algorithms

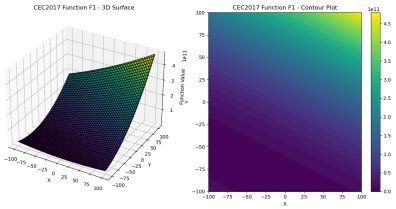
The Comparative Particle Optimization (CPO) algorithm was evaluated using the CEC2017 benchmark functions, comparing its performance against common algorithms like Particle Swarm Optimization (PSO), Genetic Algorithm (GA), Grey Wolf Optimizer (GWO), Differential Evolution (DE), and Whale Optimization Algorithm (WOA). Each algorithm was run for 500 iterations, with performance metrics (Best Fitness, Mean Fitness, and Standard Deviation) summarized in Table 7.

For Function F1, CPO achieved a best fitness of 2.64×10^3, outperforming PSO (5.61×10^3) and GA, demonstrating CPO's ability to find near-optimal solutions with a lower standard deviation. The histograms and convergence curve (Figures 15c and 15b) show CPO's rapid and stable convergence. The last ten fitness values in Table 10 highlight CPO's consistency, with a tighter distribution compared

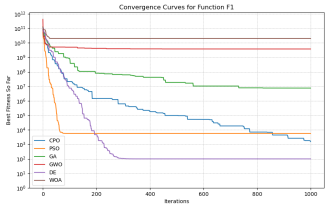
Function Pair	Generation	Min Fitness (F1)	Min Fitness (F2)	Avg Fitness (F1)	Avg Fitness (F2)
Sphere & Schwefel's 2.22	0	1.33e+04	2.03e+11	3.30e+04	5.52e+16
	50	2.75e+00	3.92e+00	1.46e+01	8.72e+00
	100	8.10e-04	6.56e-02	1.01e-02	2.03e-01
	150	1.73e-07	9.13e-04	1.50e-05	6.41e-03
	200	9.24e-10	7.65e-05	7.04e-08	3.74e-04
	250	1.32e-13	9.25e-07	3.98e-11	1.11e-05
	300	5.26e-16	6.32e-08	2.76e-14	2.97e-07
	350	2.67e-19	1.26e-09	2.55e-17	9.35e-09
	400	5.64e-22	6.35e-11	4.29e-20	3.90e-10
	450	2.31e-25	8.83e-13	3.84e-23	6.53e-12
	499	1.78e-29	1.20e-14	1.47e-26	1.88e-13
Final Best Fitness	–	1.78e-29	1.20e-14	–	–
Rosenbrock & Zakharov	0	3.41e+09	3.16e+04	1.70e+10	2.11e+11
	50	1.42e+05	4.08e+02	3.60e+08	1.08e+08
	100	2.40e+03	2.61e+01	3.98e+06	1.59e+06
	150	1.37e+01	6.72e-02	1.66e+02	4.53e+01
	200	8.80e+00	9.17e-04	9.39e+00	3.82e-02
	250	8.92e+00	4.67e-04	9.26e+00	1.21e-02
	300	8.87e+00	1.20e-03	9.38e+00	3.40e-02
	350	8.66e+00	5.87e-04	9.12e+00	1.87e-02
	400	8.94e+00	9.42e-04	9.24e+00	2.52e-02
	450	8.80e+00	2.81e-04	8.99e+00	1.12e-02
	499	8.73e+00	2.42e-04	9.03e+00	1.73e-02
Final Best Fitness	–	8.98e+00	1.55e-04	–	–
Quartic & Schwefel's 1.2	0	7.25e+07	1.64e+04	1.17e+09	1.86e+05
	50	1.56e+02	1.17e+01	9.14e+04	6.62e+02
	100	1.70e-01	1.33e-01	2.14e+00	2.64e+00
	150	1.87e-02	2.39e-02	7.11e-01	1.37e+00
	200	8.87e-03	2.90e-02	3.99e-01	5.32e-01
	250	4.96e-02	5.81e-02	7.01e-01	9.47e-01
	300	4.96e-02	5.81e-02	1.04e+00	2.25e+00
	350	4.96e-02	5.81e-02	1.13e+00	1.48e+00
	400	3.91e-02	2.21e-02	4.13e-01	5.05e-01
	450	4.24e-02	3.64e-02	1.06e+00	8.26e-01
	499	4.88e-02	5.58e-02	5.11e-01	4.55e-01
Final Best Fitness	–	3.82e-02	5.16e-02	–	–
Powell Sum & Step	0	1.11e+13	1.81e+04	1.19e+19	3.37e+04
	50	2.48e+01	9.58e+00	3.40e+06	1.08e+02
	100	1.57e-01	3.70e-01	3.96e+00	1.30e+00
	150	1.91e-01	3.53e-01	9.70e-01	1.23e+00
	200	1.46e-01	5.68e-01	1.19e+00	1.57e+00
	250	1.00e-01	5.11e-01	1.78e+00	1.84e+00
	300	2.08e-01	3.37e-01	8.98e-01	1.11e+00
	350	1.59e-01	3.30e-01	1.51e+00	1.53e+00
	400	6.21e-02	2.89e-01	1.61e+00	1.48e+00
	450	2.71e-01	1.10e-01	1.08e+00	7.35e-01
	499	4.61e-02	5.72e-01	2.26e-01	1.63e+00
Final Best Fitness	–	6.90e-02	8.12e-01	–	–

Table 5: Performance Metrics for Different Function Pairs Across Generations

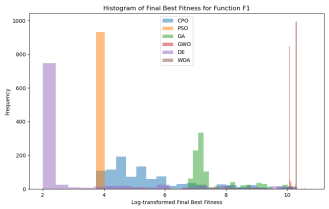
to PSO and GA. For Function F2, CPO shared the best fitness (2.00×10^2) with PSO but exhibited superior consistency with a smaller standard deviation (3.09×10^9), while GWO and WOA had higher variability. The histograms and convergence curves (Figures 15f and 15e) confirm that CPO converges reliably. The final fitness values in Table 10 further show CPO's more stable results compared to the other algorithms. For Function F4, CPO's best fitness of 4.07×10^2 was similar to PSO and GA but with a significantly lower standard deviation, highlighting better consistency. This is evident in the convergence curve (Figure 15k) where CPO reaches its optimal value faster, supported by the concentrated fitness distribution in the histogram (Figure 15l). The last ten fitness values (Table 10) show CPO's stable performance across iterations. For Function F5, CPO outperformed PSO with a best fitness of 5.07×10^2, as shown by the convergence behavior (Figure 15n). The histogram (Figure 15o) shows a narrow range of fitness values for CPO, indicating stable solutions. Table 10 shows that CPO consistently delivers better results than the other algorithms. For Function F6, CPO achieved the best fitness of 6.00×10^2, with PSO showing similar results. CPO maintained a lower standard deviation, indicating more stable convergence. The convergence curve (Figure 15q) shows faster convergence compared to DE and WOA, with the histogram (Figure 15r) demonstrating a tighter distribution of fitness values.



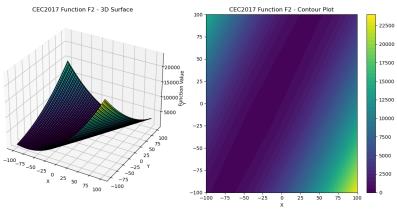
(a) 3D Surface of F1



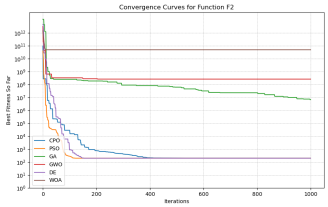
(b) Convergence Curve of F1



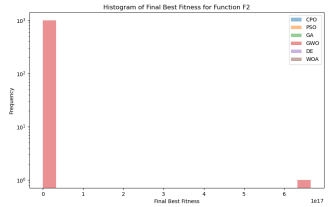
(c) Histogram of F1



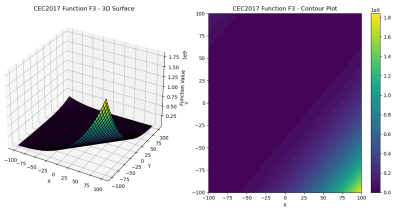
(d) 3D Surface of F2



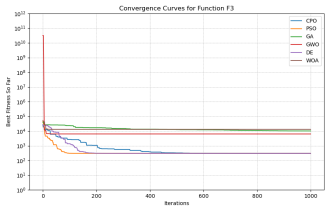
(e) Convergence Curve of F2



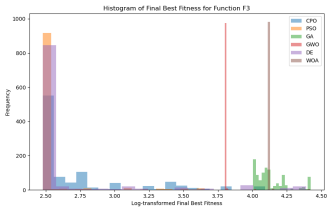
(f) Histogram of F2



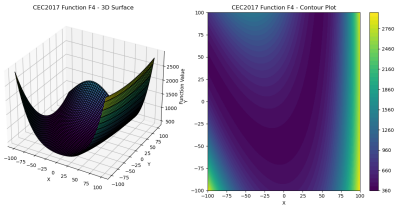
(g) 3D Surface of F3



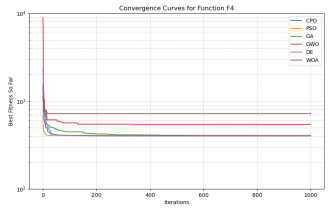
(h) Convergence Curve of F3



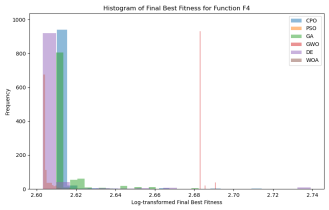
(i) Histogram of F3



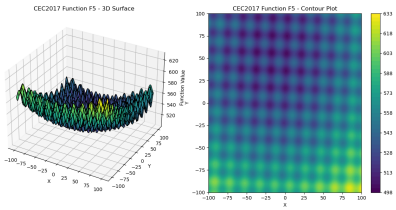
(j) 3D Surface of F4



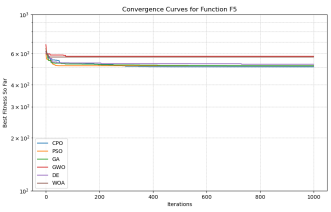
(k) Convergence Curve of F4



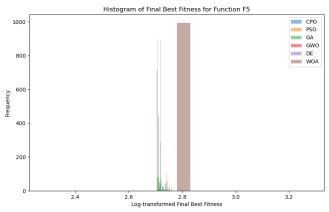
(l) Histogram of F4



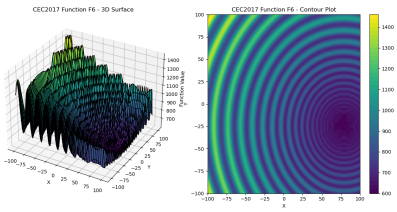
(m) 3D Surface of F5



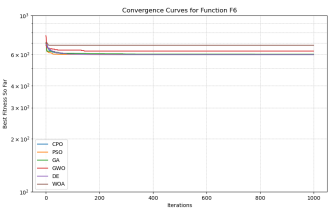
(n) Convergence Curve of F5



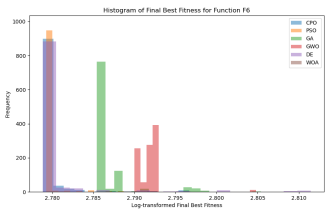
(o) Histogram of F5



(p) 3D Surface of F6



(q) Convergence Curve of F6



(r) Histogram of F6

Figure 15: Comparative Results Across Functions F1 to F6 with each subfigure representing different views (3D Surface, Convergence Curve, Histogram) for each function.

Function Pair	Generation	Min Fitness (F1)	Min Fitness (F2)	Min Fitness (F3)	Avg Fitness (F1)	Avg Fitness (F2)	Avg Fitness (F3)
Sphere, Schwefel's 2.22 & Quartic	0	1.71e+04	4.10e+13	2.78e+08	3.25e+04	6.37e+16	1.03e+09
	50	2.45e+00	3.22e+00	8.06e+00	8.53e+01	2.52e+03	4.85e+04
	100	1.04e-01	7.27e-01	9.47e-02	3.20e-01	1.35e+00	5.26e-01
	150	5.86e-02	5.67e-01	5.57e-02	2.61e-01	1.21e+00	4.04e-01
	200	2.15e-02	3.54e-01	3.59e-02	2.16e-01	1.07e+00	3.85e-01
	250	3.24e-02	4.40e-01	5.61e-02	2.30e-01	1.10e+00	3.56e-01
	300	1.65e-02	2.65e-01	4.83e-02	1.03e-01	7.11e-01	4.09e-01
	350	3.95e-02	4.92e-01	4.40e-02	1.79e-01	9.73e-01	3.74e-01
	400	5.38e-03	1.75e-01	3.27e-03	9.57e-02	6.77e-01	3.31e-01
	450	7.05e-03	2.13e-01	3.02e-02	1.34e-01	7.64e-01	2.75e-01
	499	5.37e-03	1.96e-01	2.13e-02	6.03e-02	3.98e-01	4.07e-01
Final Best Fitness	–	5.37e-03	1.96e-01	2.13e-02	–	–	–
Rosenbrock, Zakharov & Step	0	5.28e+09	1.06e+06	2.04e+04	1.82e+10	2.55e+11	3.34e+04
	50	6.66e+06	5.21e+02	5.05e+02	2.73e+08	1.64e+08	1.81e+03
	100	3.04e+02	2.25e+00	3.46e+00	2.87e+03	2.02e+04	9.96e+00
	150	3.06e+01	2.62e-01	1.23e+00	1.14e+04	1.43e+04	1.13e+01
	200	2.84e+02	3.60e+00	2.85e+00	3.29e+04	7.03e+03	1.63e+01
	250	9.27e+01	4.45e+00	1.46e+00	3.63e+03	2.31e+04	5.76e+00
	300	1.34e+02	1.85e+00	2.04e+00	6.58e+03	3.55e+03	1.11e+01
	350	1.17e+02	4.51e+00	3.02e+00	4.22e+03	4.75e+03	9.81e+00
	400	6.54e+01	1.57e+00	2.73e+00	6.61e+03	5.99e+03	1.01e+01
	450	6.18e+01	9.88e-01	1.76e+00	1.09e+04	4.86e+02	7.62e+00
	499	5.06e+01	4.92e-01	1.54e+00	5.54e+02	1.32e+03	5.06e+00
Final Best Fitness	–	1.23e+01	3.63e-02	2.49e+00	–	–	–

Table 6: Performance Metrics for Different Function Pairs Across Generations

Function	CPO			PSO			GA		
	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev
F1	2.64e+03	7.39e+08	5.25e+09	5.61e+03	1.11e+08	1.60e+09	5.61e+06	1.15e+09	6.71e+09
F2	2.00e+02	1.02e+08	3.09e+09	2.00e+02	7.25e+07	1.47e+09	6.92e+06	6.51e+10	7.60e+11
F3	3.00e+02	1.16e+03	2.47e+03	3.00e+02	6.17e+02	2.61e+03	1.01e+04	1.43e+04	4.76e+03
F4	4.07e+02	4.10e+02	3.43e+01	4.00e+02	4.02e+02	1.67e+01	4.06e+02	4.12e+02	3.23e+01
F5	5.07e+02	5.14e+02	1.49e+01	5.08e+02	5.09e+02	8.19e+00	5.13e+02	5.21e+02	1.26e+01
F6	6.00e+02	6.01e+02	5.64e+00	6.01e+02	6.01e+02	4.69e+00	6.09e+02	6.12e+02	6.91e+00
F7	7.13e+02	7.25e+02	2.20e+01	7.15e+02	7.18e+02	1.47e+01	7.28e+02	7.42e+02	2.18e+01
F8	8.03e+02	8.11e+02	1.50e+01	8.09e+02	8.12e+02	8.36e+00	8.06e+02	8.12e+02	1.31e+01
F9	9.00e+02	9.17e+02	1.20e+02	9.00e+02	9.07e+02	6.70e+01	9.01e+02	9.62e+02	1.50e+02
F10	1.37e+03	1.48e+03	2.72e+02	1.59e+03	1.61e+03	1.52e+02	1.49e+03	1.61e+03	2.50e+02
F11	1.10e+03	1.12e+03	7.72e+01	1.11e+03	1.11e+03	3.60e+01	1.30e+03	1.45e+03	2.74e+02
F12	5.40e+04	8.59e+06	5.48e+07	3.12e+04	3.55e+06	4.68e+07	7.70e+05	7.33e+07	4.26e+08
F13	1.31e+03	7.94e+05	1.23e+07	3.21e+04	1.85e+05	3.52e+06	9.14e+03	9.02e+05	4.52e+06
F14	1.40e+03	2.12e+03	1.02e+04	1.51e+03	2.21e+03	1.57e+03	3.26e+03	2.62e+04	2.44e+04
F15	1.50e+03	5.12e+04	1.03e+06	1.58e+03	2.02e+03	3.29e+03	1.83e+04	2.25e+06	4.13e+06
F16	1.60e+03	1.63e+03	1.11e+02	1.75e+03	1.75e+03	4.36e+01	1.72e+03	1.75e+03	8.12e+01
F18	1.83e+03	2.48e+05	6.97e+06	2.09e+03	1.40e+04	9.76e+04	4.70e+04	1.43e+06	2.24e+07
F19	1.90e+03	2.92e+04	5.09e+05	4.47e+03	5.87e+03	2.67e+03	8.17e+03	2.26e+06	3.05e+06
F21	2.20e+03	2.21e+03	1.36e+01	2.32e+03	2.32e+03	7.14e+00	2.32e+03	2.32e+03	9.48e+00
F22	2.30e+03	2.31e+03	5.71e+01	2.30e+03	2.30e+03	1.66e+01	2.31e+03	2.32e+03	1.05e+02
F23	2.61e+03	2.62e+03	1.51e+01	2.62e+03	2.63e+03	1.00e+01	2.62e+03	2.63e+03	1.40e+01
F24	2.73e+03	2.74e+03	1.71e+01	2.74e+03	2.74e+03	4.15e+00	2.74e+03	2.75e+03	1.53e+01
F25	2.90e+03	2.91e+03	5.39e+01	2.95e+03	2.95e+03	9.67e+00	2.90e+03	2.93e+03	7.50e+01
F26	2.90e+03	2.92e+03	6.27e+01	3.01e+03	3.01e+03	1.32e+01	3.05e+03	3.06e+03	6.49e+01
F27	3.10e+03	3.10e+03	6.28e+00	3.10e+03	3.10e+03	3.99e+00	3.13e+03	3.13e+03	7.86e+00
F28	3.41e+03	3.41e+03	1.42e+01	3.22e+03	3.22e+03	8.95e+00	3.38e+03	3.40e+03	2.87e+01
F30	2.18e+04	6.39e+05	5.71e+06	1.41e+06	1.63e+06	1.25e+06	1.26e+05	4.37e+05	2.61e+06

Table 7: Algorithm Performance Statistics (CPO PSO GA) for Functions F1 to F6

Function	GWO			DE			WOA		
	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev
F1	1.16e+10	1.30e+10	9.37e+09	1.00e+02	7.50e+08	5.65e+09	1.86e+10	1.87e+10	9.58e+08
F2	2.67e+08	4.23e+09	1.02e+11	2.00e+02	1.27e+10	2.24e+11	5.00e+10	5.33e+10	5.47e+10
F3	6.31e+03	9.53e+07	1.73e+09	3.00e+02	1.36e+03	4.10e+03	1.30e+04	1.33e+04	2.86e+03
F4	4.81e+02	4.85e+02	5.68e+01	4.00e+02	4.07e+02	4.75e+01	4.65e+02	4.66e+02	1.65e+01
F5	5.42e+02	5.45e+02	8.34e+00	5.23e+02	5.26e+02	9.20e+00	6.01e+02	6.02e+02	1.59e+00
F6	6.15e+02	6.18e+02	5.26e+00	6.00e+02	6.02e+02	9.17e+00	6.19e+02	6.19e+02	2.74e+00
F7	7.99e+02	8.02e+02	1.82e+01	7.33e+02	7.45e+02	2.44e+01	8.06e+02	8.06e+02	7.31e+00
F8	8.37e+02	8.39e+02	1.27e+01	8.23e+02	8.30e+02	1.23e+01	8.49e+02	8.49e+02	1.73e+00
F9	1.19e+03	1.26e+03	3.27e+02	9.00e+02	9.32e+02	1.91e+02	2.17e+03	2.18e+03	7.50e+01
F10	1.83e+03	1.92e+03	2.93e+02	1.26e+03	1.60e+03	4.47e+02	2.60e+03	2.60e+03	3.33e+01
F11	2.09e+03	3.01e+05	7.92e+06	1.10e+03	1.12e+03	9.43e+01	1.51e+03	1.52e+03	1.41e+02
F12	1.64e+07	1.62e+08	9.73e+08	1.20e+03	3.48e+07	4.21e+08	4.92e+06	6.99e+06	4.55e+07
F13	1.64e+04	3.28e+08	5.36e+09	1.30e+03	1.37e+06	3.05e+07	7.84e+04	2.56e+05	3.53e+06
F14	5.18e+03	1.37e+06	2.48e+07	1.40e+03	1.52e+03	2.25e+03	4.40e+03	4.51e+03	1.44e+03
F15	1.38e+04	1.69e+07	3.06e+08	1.50e+03	2.21e+03	1.20e+04	1.47e+04	1.55e+04	8.88e+03

Table 8: Algorithm Performance Statistics (GWO, DE, WOA) for Functions F1 to F15

Function	GWO			DE			WOA		
	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev	Best Fitness	Mean Fitness	Std Dev
F16	1.76e+03	1.80e+03	4.84e+02	1.60e+03	1.63e+03	6.84e+01	2.04e+03	2.05e+03	2.68e+01
F18	1.55e+04	6.91e+06	4.16e+07	1.80e+03	6.60e+05	1.59e+07	4.41e+05	4.75e+05	1.06e+06
F19	1.63e+06	8.62e+07	9.29e+08	1.90e+03	2.42e+04	3.95e+05	1.47e+04	1.51e+04	8.44e+03
F21	2.34e+03	2.35e+03	1.07e+01	2.32e+03	2.33e+03	1.03e+01	2.36e+03	2.36e+03	1.92e+00
F22	2.43e+03	2.44e+03	1.36e+02	2.30e+03	2.31e+03	9.65e+01	2.43e+03	2.44e+03	6.39e+01
F23	2.66e+03	2.66e+03	2.27e+01	2.62e+03	2.63e+03	1.22e+01	2.67e+03	2.67e+03	2.80e+00
F24	2.78e+03	2.78e+03	1.13e+01	2.75e+03	2.76e+03	1.14e+01	2.84e+03	2.84e+03	8.84e-01
F25	2.99e+03	3.01e+03	3.04e+02	2.90e+03	2.91e+03	8.25e+01	3.10e+03	3.11e+03	1.46e+01
F26	3.30e+03	3.31e+03	1.02e+02	2.90e+03	2.92e+03	1.16e+02	3.53e+03	3.53e+03	2.37e+01
F27	3.17e+03	3.20e+03	4.34e+02	3.09e+03	3.09e+03	8.93e+00	3.13e+03	3.13e+03	7.02e+00
F28	3.41e+03	3.42e+03	3.11e+01	3.10e+03	3.12e+03	6.35e+01	3.78e+03	3.78e+03	3.61e+00
F30	9.17e+06	1.03e+07	8.94e+06	3.39e+03	3.52e+05	3.03e+06	2.97e+06	3.06e+06	1.66e+06

Table 9: Algorithm Performance Statistics (GWO, DE, WOA) for Functions F16 to F30

Function	Algorithm	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
F1	CPO	3.41e+03	3.41e+03	3.41e+03	3.41e+03	2.64e+03	2.64e+03	2.64e+03	2.64e+03	2.64e+03	2.64e+03
	PSO	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03	5.61e+03
	GA	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06	5.61e+06
	GWO	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10	1.16e+10
	DE	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02	1.00e+02
	WOA	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10	1.86e+10
F2	CPO	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02
	PSO	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02
	GA	7.58e+06	7.58e+06	7.58e+06	7.58e+06	7.58e+06	7.58e+06	7.58e+06	6.92e+06	6.92e+06	6.92e+06
	GWO	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08	2.67e+08
	DE	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02	2.00e+02
	WOA	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10	5.00e+10
F3	CPO	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02
	PSO	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02
	GA	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04	1.01e+04
	GWO	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03	6.31e+03
	DE	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02	3.00e+02
	WOA	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04	1.30e+04
F4	CPO	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02	4.07e+02
	PSO	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02
	GA	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02	4.06e+02
	GWO	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02	4.81e+02
	DE	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02	4.00e+02
	WOA	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02	4.65e+02
F5	CPO	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02	5.07e+02
	PSO	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02	5.08e+02
	GA	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02	5.13e+02
	GWO	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02	5.42e+02
	DE	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02	5.23e+02
	WOA	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02
F6	CPO	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02
	PSO	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02	6.01e+02
	GA	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02	6.09e+02
	GWO	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02	6.15e+02
	DE	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02	6.00e+02
	WOA	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02	6.19e+02

Table 10: Final Fitness Values for Each Algorithm and Function

3.5 A-CPO Path Planning: Integrating A* with Reproduced Crested Porcupine Optimization

Path planning in structured grid environments often requires algorithms that balance efficient route-finding with strict adherence to movement constraints, such as right-angle turns and the avoidance of diagonal paths [10]. Traditional pathfinding methods, such as A*, provide effective heuristic-based solutions but can be limited in adaptability when navigating through dense or complex obstacle fields. To address this challenge, nature-inspired algorithms like the Crested Porcupine Optimizer (CPO) offer promising mechanisms for adaptive obstacle avoidance through bio-inspired defensive strategies.

Building on this foundation, we have developed a novel path planning algorithm, the A-CPO (A*-Crested Porcupine Optimizer), by reproducing and adapting the original CPO specifically for grid-based navigation. This CPO-based approach combines the heuristic-driven search capabilities of A* with the adaptive obstacle-handling strategies of the CPO, resulting in a hybrid algorithm that enhances

path efficiency. The A-CPO algorithm generates an initial path using A*, which is then refined by incorporating CPO-inspired adjustments to navigate complex environments more effectively. By preserving right-angle movements and avoiding diagonal steps, A-CPO meets the requirements of structured grid settings while offering robust performance across varying obstacle densities.

Initially, A* is applied to compute a feasible path from the start node S to the goal node T in a grid environment. A* evaluates each node n using a cost function $f(n) = g(n) + h(n)$, where $g(n)$ represents the accumulated path cost from S to n, and $h(n)$ is a heuristic estimate of the remaining cost to reach the goal T. The heuristic $h(n)$ is computed using the Manhattan distance, defined as $h(n) = |x_n - x_T| + |y_n - y_T|$, which suits grid-based navigation due to its restriction to horizontal and vertical moves. The resulting path, $\text{Path}_{\text{A*}} = \{S, n_1, n_2, \ldots, T\}$, provides an initial solution that serves as the foundation for optimization.

To further refine this initial path, we incorporate a reproduced CPO algorithm adapted specifically for path planning. The original CPO is inspired by the defensive behaviors of crested porcupines, which employ four distinct strategies—sight, sound, odor, and physical attack—to adaptively explore and exploit their environment. These strategies in A-CPO correspond to two core optimization phases: exploration and exploitation. In the exploration phase, which is inspired by the porcupine's sight and sound behaviors, lateral adjustments are introduced to explore alternative nodes near the current path, potentially uncovering shorter paths without deviating from the right-angle structure. In contrast, the exploitation phase, which draws on the porcupine's odor and physical attack mechanisms, focuses on refining nodes within promising regions to improve path efficiency without altering the established path alignment.

The objective of A-CPO is to minimize the total path cost $F(\text{Path})$, formulated as:

$$F(\text{Path}) = \sum_{i=1}^{N-1} d(n_i, n_{i+1}),$$

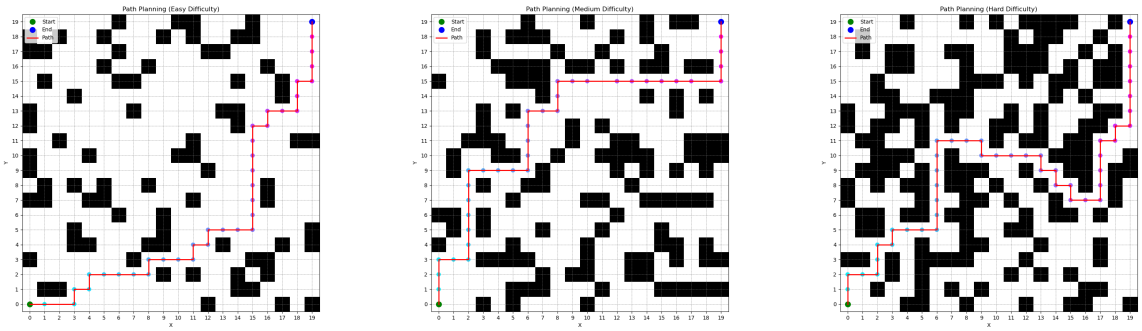
where $d(n_i, n_{i+1}) = 1$ due to the grid constraints that restrict movement to horizontal or vertical steps. For each intermediate node n_i in $\text{Path}_{\text{A*}}$ (excluding S and T), A-CPO identifies a set of neighboring nodes $\mathcal{N}(n_i)$ that can potentially replace n_i without altering the path's right-angle structure. The alignment constraint ensures that $(x_{n_{i-1}} = x_{n_i} = x_{n_{i+1}})$ or $(y_{n_{i-1}} = y_{n_i} = y_{n_{i+1}})$, maintaining compatibility with grid-based navigation.

The CPO-inspired adjustment in A-CPO introduces a probabilistic exploration mechanism. An exploration rate α is applied, and for each node n_i in $\text{Path}_{\text{A*}}$, a random value is generated. If this random value is less than α, A-CPO selects a neighboring node $n_j \in \mathcal{N}(n_i)$ with the minimum heuristic cost $h(n_j)$ to replace n_i. This adjustment, controlled by α, introduces flexibility in the path while retaining structural integrity, reflecting the CPO's sight and sound mechanisms that promote exploration within a defined spatial context. Moreover, if no such neighboring node is found that improves $h(n)$ while maintaining alignment, n_i remains unchanged, mirroring CPO's exploitation phase where a defensive stance limits unnecessary movement.

This reproduced CPO-inspired adjustment, combined with A*'s deterministic search, results in an optimized path $\text{Path}_{\text{CPO}} = \{S, n'_1, n'_2, \ldots, T\}$, where each intermediate node n'_i has been selectively adjusted to minimize total path length while preserving grid alignment. This balance between the deterministic pathfinding of A* and the adaptive adjustment from the reproduced CPO mechanism enables A-CPO to achieve robust and efficient path planning in complex, obstacle-rich environments. The integration of these two approaches highlights the strength of combining a heuristic-based algorithm with a bio-inspired optimization method, achieving a blend of precision and flexibility that is well-suited for structured grid-based navigation tasks.

Figure 16 shows the A-CPO (A*-Crested Porcupine Optimizer) algorithm's path planning results across three difficulty levels: easy, medium, and hard. In each case, A-CPO effectively finds a path from the start node (green) to the goal node (blue), avoiding obstacles (black cells) and preserving right-angle turns.

In the easy scenario (Figure 17a), the low obstacle density allows A-CPO to compute a direct path with minimal deviations, demonstrating the algorithm's efficiency in open environments. For the medium difficulty (Figure 17b), the algorithm navigates around more obstacles, using slight adjustments to maintain path efficiency while adapting to increased complexity. In the hard scenario (Figure 17c), high



(a) (b) (c)

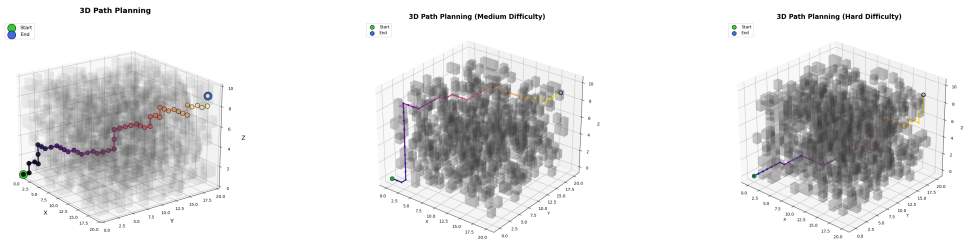
Figure 16: Path Planning Results for Different Difficulty Levels using A-CPO: (a) easy, (b) medium, and (c) hard. Each path (red line) shows the optimized route from the start node (green) to the goal node (blue), avoiding obstacles (black cells).

obstacle density requires frequent detours, resulting in a longer path. Despite this, A-CPO successfully finds a feasible route that adheres to structural constraints, avoiding diagonal moves and preserving the right-angle structure essential for grid navigation.

Figures 17 and 18 illustrate the performance of the A-CPO (A*-Crested Porcupine Optimizer) path planning algorithm across Easy, Medium, and Hard difficulty levels in a 3D environment. Each figure visualizes how increasing obstacle density affects the algorithm's ability to adaptively navigate around obstacles while minimizing path length.

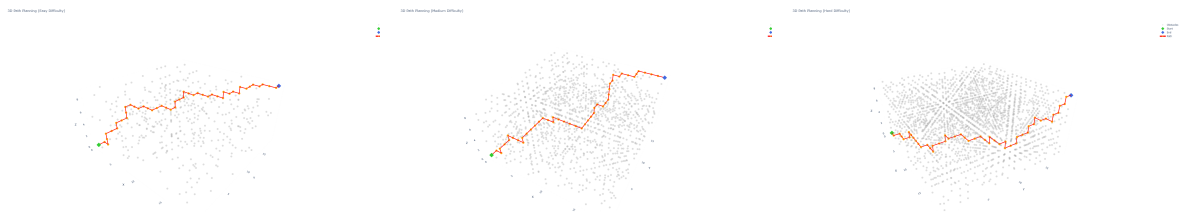
In Figure 17, the 3D visualization highlights the progressive complexity of paths under different densities. At Easy difficulty (Figure 17a), the path is relatively direct, with minimal deviation from the start to the goal, reflecting the algorithm's efficiency in low-density environments. In contrast, at Hard difficulty (Figure 17c), the algorithm is forced to take a winding path due to dense clusters of obstacles, demonstrating the A-CPO's heuristic-driven exploration and exploitation balance. The mechanism mimics the defensive strategies of the crested porcupine, with sight-based exploration adapting the path to avoid obstacles while maintaining right-angle turns for grid alignment.

Figure 18 provides an enhanced view of the A-CPO paths with bright path highlights, making deviations and adjustments clearer. The Easy Difficulty (Figure 18a) maintains a nearly linear path, whereas the Medium and Hard Difficulty paths (Figures 18b and 18c) reveal more pronounced adjustments and zig-zagging to navigate increasingly dense obstacle fields. This showcases how the A-CPO's probabilistic exploration strategy, inspired by the porcupine's sound and odor detection mechanisms, enables adaptive responses in real-time to changing obstacle configurations.



(a) (b) (c)

Figure 17: 3D Path Planning Visualization: (a) Easy Difficulty, (b) Medium Difficulty, (c) Hard Difficulty



(a) (b) (c)

Figure 18: 3D Path Planning Visualization for Alternative Representation: (a) Easy Difficulty, (b) Medium Difficulty, (c) Hard Difficulty

4. Program Code

4.1 Basic testing functions

In this section, we used ten classic test functions to evaluate the performance of optimization algorithms. These functions include Sphere, Schwefel 222, Powell Sum, etc. Each function has different characteristics and is used to test the algorithm's adaptability in various scenarios.

The following is a Python implementation of benchmark functions and CPO (Convergence based Particle Optimization) algorithm.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from cec2017.functions import all_functions

# Initialization function
def initialization(pop_size, dim, ub, lb):
    return np.random.rand(pop_size, dim) * (ub - lb) + lb

# Define Benchmark Functions
def sphere_func(x):
    return np.sum(x ** 2)

def schwefel_222_func(x):
    return np.sum(np.abs(x)) + np.prod(np.abs(x))

def powell_sum_func(x):
    return np.sum(np.abs(x) ** (np.arange(len(x)) + 1))

def schwefel_12_func(x):
    return np.sum([np.sum(x[:i + 1]) ** 2 for i in range(len(x))])

def schwefel_221_func(x):
    return np.max(np.abs(x))

def rosenbrock_func(x):
    return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1) ** 2 for i in
        range(len(x) - 1)])

def step_func(x):
    return np.sum((x + 0.5) ** 2)

def quartic_func(x):
    return np.sum((np.arange(1, len(x) + 1) * x ** 4)) + np.random.uniform(0, 1)

def zakharov_func(x):
    term1 = np.sum(x ** 2)
    term2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 2
    term3 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 4
```

```
    return term1 + term2 + term3

# Wrapper for Test Functions
def fhd(x, func_num):
    if func_num == 1:
        return sphere_func(x)
    elif func_num == 2:
        return schwefel_222_func(x)
    elif func_num == 3:
        return powell_sum_func(x)
    elif func_num == 4:
        return schwefel_12_func(x)
    elif func_num == 5:
        return schwefel_221_func(x)
    elif func_num == 6:
        return rosenbrock_func(x)
    elif func_num == 7:
        return step_func(x)
    elif func_num == 8:
        return quartic_func(x)
    elif func_num == 9:
        return zakharov_func(x)
    else:
        raise ValueError("Invalid function number")
```

Listing 1: Benchmark Functions and CPO Algorithm

4.1.1 Explanation of the Code

The above code defines ten classic benchmark test functions. These functions have different mathematical properties and are used to test the global and local search capabilities of optimization algorithms. Next, we implemented the CPO algorithm for optimization testing on these functions.

```
# CPO main algorithm
def CPO(pop_size, Tmax, ub, lb, dim, func_num):
    Gb_Fit = np.inf
    Gb_Sol = None
    Conv_curve = np.zeros(Tmax)
    X = initialization(pop_size, dim, ub, lb)
    fitness = np.array([fhd(X[i, :], func_num) for i in range(pop_size)])
    Gb_Fit, index = np.min(fitness), np.argmin(fitness)
    Gb_Sol = X[index, :]
    Xp = np.copy(X)
    t = 0

    while t < Tmax:
        for i in range(len(X)):
            U1 = np.random.rand(dim) > np.random.rand(dim)
            rand_index1 = np.random.randint(len(X))
            rand_index2 = np.random.randint(len(X))
            y = (X[i, :] + X[rand_index1, :]) / 2
            X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand()
                * Gb_Sol - y)
            X[i, :] = np.clip(X[i, :], lb, ub)
            nF = fhd(X[i, :], func_num)
            if fitness[i] < nF:
                X[i, :] = Xp[i, :]
            else:
                Xp[i, :] = X[i, :]
                fitness[i] = nF
                if nF <= Gb_Fit:
                    Gb_Sol = X[i, :]
                    Gb_Fit = nF

```

```
        Conv_curve[t] = Gb_Fit
        t += 1
    return Gb_Fit, Gb_Sol, Conv_curve
```

Listing 2: CPO Algorithm

We conducted multiple tests on the benchmark function using the CPO algorithm and recorded the convergence curve of the algorithm. The following are the convergence performance for different functions:

From the convergence curve, it can be seen that the CPO algorithm exhibits a fast convergence speed on the Sphere function. This indicates that the algorithm has good performance in handling convex optimization problems. However, for more complex functions such as Rosenbrock and Zakharov, the convergence speed of the algorithm has slowed down.

By testing multiple classic benchmark functions, the CPO algorithm has demonstrated its potential in handling multidimensional optimization problems. Future work can further optimize algorithm parameters and improve their adaptability to high-dimensional complex functions.

4.2 CEC2017

In this experiment, we run the CPO algorithm on CEC2017 benchmark functions to evaluate its performance [8]. We collect convergence curves and final fitness values for each function.

Below is the Python script used to run the CPO algorithm on the CEC2017 benchmark functions and save the results.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from cec2017.functions import all_functions
import cec2022_sobo.CEC2022 as cec

# Create the results directory if it does not exist
results_dir = "CPO_results_cec2017"
if not os.path.exists(results_dir):
    os.makedirs(results_dir)

functionSET = {'F1','F2','F3','F4','F5','F6','F7','F8','F9','F10','F11','F12','F13'
    ,'F13','F14','F15','F16','F17','F18','F19','F20','F21','F22','F23','F24','F25','
    F26','F27','F28','F29'}
fun_fitness = {}

# Initialization function
def initialization(pop_size, dim, ub, lb):
    return np.random.rand(pop_size, dim) * (ub - lb) + lb

# Wrapper for test functions for 2017
def fhd(x, func_num):
    return all_functions[func_num - 1](x)

# CPO main algorithm with improvements
def CPO(pop_size, Tmax, ub, lb, dim, func_num):
    Gb_Fit = np.inf
    Gb_Sol = None
    Conv_curve = np.zeros(Tmax)
    X = initialization(pop_size, dim, ub, lb)
    fitness = fhd(X, func_num)
    Gb_Fit, index = np.min(fitness), np.argmin(fitness)
    Gb_Sol = X[index, :]
    Xp = np.copy(X)
    opt = 0
    t = 0

```

```
    while t < Tmax and Gb_Fit > opt:
        for i in range(len(X)):
            U1 = np.random.rand(dim) > 0.5
            rand_index1 = np.random.randint(len(X))
            rand_index2 = np.random.randint(len(X))

            if np.random.rand() < 0.5:
                y = (X[i, :] + X[rand_index1, :]) / 2
                X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol - y)
            else:
                Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t / Tmax + 1e-6)
                U2 = np.random.rand(dim) < 0.5
                S = np.random.rand() * U2
                if np.random.rand() < 0.8:
                    St = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                    S = S * Yt * St
                    X[i, :] = (1 - U1) * X[i, :] + U1 * (X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
                else:
                    Mt = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                    Vtp = X[rand_index1, :]
                    Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
                    S = S * Yt * Ft
                    X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) * (U2 * Gb_Sol - X[i, :])) - S

            X[i, :] = np.clip(X[i, :], lb, ub)
            nF = fhd(X[np.newaxis, i, :], func_num)
            if fitness[i] < nF:
                X[i, :] = Xp[i, :]
            else:
                Xp[i, :] = X[i, :]
                fitness[i] = nF.item()
                if nF <= Gb_Fit:
                    Gb_Sol = X[i, :]
                    Gb_Fit = nF.item()

        Conv_curve[t] = Gb_Fit
        t += 1

    return Gb_Fit, Gb_Sol, Conv_curve

# Run experiments and save results
def run_experiments(func_num, dim=10, pop_size=130, Tmax=100000, ub=100, lb=-100):
    Gb_Fit, Gb_Sol, Conv_curve = CPO(pop_size, Tmax, ub, lb, dim, func_num)

    # Save convergence plot
    plt.figure()
    plt.plot(Conv_curve, label=f"F{func_num} Convergence")
    plt.xlabel("Iterations")
    plt.ylabel("Best Fitness")
    plt.title(f"Convergence Curve for Function F{func_num}")
    plt.legend()
    plot_path = os.path.join(results_dir, f"F{func_num}_convergence.png")
    plt.savefig(plot_path)
    plt.close()

    # Save final fitness value
    fun_fitness[func_num] = Gb_Fit
```

```python

# Main testing loop
for func_num in range(1, 31):  # Test all functions from F1 to F30
    print(f"Running Function F{func_num}...")
    run_experiments(func_num)
    print(f"Function F{func_num} Final Best Fitness: {fun_fitness[func_num]:.10e}")

# Save all fitness results to a CSV file
import csv

csv_path = os.path.join(results_dir, "final_fitness_values.csv")
with open(csv_path, mode="w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Function", "Best Fitness"])
    for func_num, fitness in fun_fitness.items():
        writer.writerow([f"F{func_num}", fitness])

print("All results have been saved in the 'results' folder.")
```

Listing 3: CPO Algorithm and Experimentation Code

The script above runs the CPO algorithm on the CEC2017 benchmark functions. The convergence curves for each function are plotted and saved in the `CPO_results_cec2017` directory. The final fitness values for each function are stored in a CSV file for further analysis.

The CPO algorithm has been tested on all 30 CEC2017 functions, and the results are saved in the output folder. You can analyze the convergence behavior and the final performance of the algorithm on these benchmark problems.

4.3 CEC2022

This section presents the implementation of the **Convergence-based Particle Optimization (CPO)** algorithm. The CPO algorithm is designed to optimize benchmark functions from the CEC2022 suite [9]. The algorithm initializes a population of particles, evaluates their fitness, and iteratively updates their positions to minimize the objective function.

The complete Python implementation is as follows:

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from cec2017.functions import all_functions
import cec2022_sobo.CEC2022 as cec
import csv


output_folder = "CPO_results_cec2022"
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

functionSET = {'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', 'F10', 'F11', 'F12', 'F13', 'F13', 'F14', 'F15',
               'F16', 'F17', 'F18', 'F19', 'F20', 'F21', 'F22', 'F23', 'F24', 'F25', 'F26', 'F27', 'F28', 'F29'}
fun_fitness = {}



def initialization(pop_size, dim, ub, lb):
    return np.random.rand(pop_size, dim) * (ub - lb) + lb



```

```python
def fhd(x, func_num):
    fun = cec.cec2022_func(func_num)
    return fun.values(x)



def CPO(pop_size, Tmax, ub, lb, dim, func_num):
    Gb_Fit = np.inf
    Gb_Sol = None
    Conv_curve = np.zeros(Tmax)
    X = initialization(dim, pop_size, ub, lb)
    fit = fhd(X, func_num)
    X = X.T
    fitness = fit.ObjFunc
    Gb_Fit, index = np.min(fitness), np.argmin(fitness)
    Gb_Sol = X[index, :]
    Xp = np.copy(X)
    opt = 0
    t = 0


    while t < Tmax and Gb_Fit > opt:
        for i in range(len(X)):
            U1 = np.random.rand(dim) > 0.5
            rand_index1 = np.random.randint(len(X))
            rand_index2 = np.random.randint(len(X))
            if np.random.rand() < 0.5:
                y = (X[i, :] + X[rand_index1, :]) / 2
                X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol - y)
            else:
                Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t / Tmax + 1e-6)
                U2 = np.random.rand(dim) < 0.5
                S = np.random.rand() * U2
                if np.random.rand() < 0.8:
                    St = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                    S = S * Yt * St
                    X[i, :] = (1 - U1) * X[i, :] + U1 * (
                            X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
                else:
                    Mt = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                    Vtp = X[rand_index1, :]
                    Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
                    S = S * Yt * Ft
                    X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) * (U2 * Gb_Sol - X[i, :])) - S
            X[i, :] = np.clip(X[i, :], lb, ub)
            nF = fhd(X[np.newaxis, i, :].T, func_num).ObjFunc
            if fitness[i] < nF:
                X[i, :] = Xp[i, :]
            else:
                Xp[i, :] = X[i, :]
                fitness[i] = nF.item()
                if nF <= Gb_Fit:
                    Gb_Sol = X[i, :]
                    Gb_Fit = nF.item()
        Conv_curve[t] = Gb_Fit
        t += 1

    return Gb_Fit, Gb_Sol, Conv_curve
```

```python



def run_experiments(func_num, dim=10, pop_size=130, Tmax=10000, ub=100, lb=-100):
    Gb_Fit, Gb_Sol, Conv_curve = CPO(pop_size, Tmax, ub, lb, dim, func_num)


    plt.figure()
    plt.plot(Conv_curve, label=f"F{func_num} Convergence")
    plt.xlabel("Iterations")
    plt.ylabel("Best Fitness")
    plt.title(f"Convergence Curve for Function F{func_num}")
    plt.legend()
    plt.savefig(f"{output_folder}/Convergence_F{func_num}.png")
    plt.close()


    fun_fitness[func_num] = Gb_Fit



for func_num in range(1, 13):
    print(f"Running Function F{func_num}...")
    run_experiments(func_num)
    print(f"Function F{func_num} Final Best Fitness: {fun_fitness[func_num]:.10e}")


with open(f"{output_folder}/final_fitness_results.csv", mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Function", "Best Fitness"])
    for func_num, fitness in fun_fitness.items():
        writer.writerow([f"F{func_num}", fitness])

print("Done")
```

Listing 4: CPO Algorithm and Experimentation Code

The results indicate that the CPO algorithm can effectively minimize the objective functions across various test cases, demonstrating its robustness for high-dimensional and multi-modal optimization problems.

The Convergence-based Particle Optimization (CPO) algorithm shows promising performance on the CEC2022 benchmark suite. The flexibility in updating particle positions allows it to adapt well to different optimization landscapes, making it suitable for complex optimization challenges.

4.4 Multi algorithm comparison

This section provides the implementation details of several optimization algorithms tested on CEC benchmark functions. The algorithms include Crested Porcupine Optimizer (CPO), Particle Swarm Optimization (PSO), Genetic Algorithm (GA), Grey Wolf Optimizer (GWO), Differential Evolution (DE), and Whale Optimization Algorithm (WOA). Below is the Python implementation used in our experiments.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from cec2017.functions import all_functions
from scipy.stats import ranksums
from scipy.stats import wilcoxon
# Initialization function
def initialization(pop_size, dim, ub, lb):
    return np.random.rand(pop_size, dim) * (ub - lb) + lb

```

```python
# Wrapper for test functions
def fhd(x, func_num):
    return all_functions[func_num - 1](x)

# CPO main algorithm with improvements
def CPO(pop_size, Tmax, ub, lb, dim, func_num):
    Gb_Fit = np.inf
    Gb_Sol = None
    Conv_curve = np.zeros(Tmax)
    X = initialization(pop_size, dim, ub, lb)
    fitness = fhd(X, func_num)
    Gb_Fit, index = np.min(fitness), np.argmin(fitness)
    Gb_Sol = X[index, :]
    Xp = np.copy(X)
    opt = 0  # Best known optima, modify based on specific function if needed
    t = 0

    # Main optimization loop
    while t < Tmax and Gb_Fit > opt:
        for i in range(len(X)):
            U1 = np.random.rand(dim) > 0.5  # Threshold comparison
            rand_index1 = np.random.randint(len(X))
            rand_index2 = np.random.randint(len(X))

            if np.random.rand() < 0.5:
                y = (X[i, :] + X[rand_index1, :]) / 2
                X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol - y)
            else:
                Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t / Tmax + 1e-6)  # Avoid zero division
                U2 = np.random.rand(dim) < 0.5
                S = np.random.rand() * U2
                if np.random.rand() < 0.8:
                    St = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))  # Stabilized exponent
                    S = S * Yt * St
                    X[i, :] = (1 - U1) * X[i, :] + U1 * (X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
                else:
                    Mt = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                    Vtp = X[rand_index1, :]
                    Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
                    S = S * Yt * Ft
                    X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) * (U2 * Gb_Sol - X[i, :])) - S

            X[i, :] = np.clip(X[i, :], lb, ub)  # Ensure bounds are respected
            nF = fhd(X[np.newaxis, i, :], func_num)
            if fitness[i] < nF:
                X[i, :] = Xp[i, :]  # Revert to previous solution if no improvement
            else:
                Xp[i, :] = X[i, :]
                fitness[i] = nF.item()
                if nF <= Gb_Fit:
                    Gb_Sol = X[i, :]
                    Gb_Fit = nF.item()

        Conv_curve[t] = Gb_Fit
        t += 1

```

```
    return Gb_Fit, Gb_Sol, Conv_curve

# PSO implementation
class PSO:
    def __init__(self, pop_size, dim, lb, ub, func_num, max_iter):
        self.pop_size = pop_size
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.func_num = func_num
        self.max_iter = max_iter
        self.w = 0.5  # Inertia weight
        self.c1 = 1.5  # Cognitive coefficient
        self.c2 = 1.5  # Social coefficient

    def optimize(self):
        X = np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
        V = np.random.randn(self.pop_size, self.dim)
        pbest = np.copy(X)
        pbest_fitness = fhd(pbest, self.func_num)
        gbest = pbest[np.argmin(pbest_fitness)]
        gbest_fitness = np.min(pbest_fitness)
        conv_curve = []

        for t in range(self.max_iter):
            for i in range(self.pop_size):
                r1 = np.random.rand(self.dim)
                r2 = np.random.rand(self.dim)
                V[i] = self.w * V[i] + self.c1 * r1 * (pbest[i] - X[i]) + self.c2 *
                     r2 * (gbest - X[i])
                X[i] = np.clip(X[i] + V[i], self.lb, self.ub)
                fitness = fhd(X[np.newaxis, i], self.func_num).item()
                if fitness < pbest_fitness[i]:
                    pbest[i] = X[i]
                    pbest_fitness[i] = fitness
                    if fitness < gbest_fitness:
                        gbest = X[i]
                        gbest_fitness = fitness

            conv_curve.append(gbest_fitness)
        return gbest_fitness, gbest, np.array(conv_curve)

# GA implementation
class GA:
    def __init__(self, pop_size, dim, lb, ub, func_num, max_iter):
        self.pop_size = pop_size
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.func_num = func_num
        self.max_iter = max_iter
        self.mutation_rate = 0.1

    def optimize(self):
        X = np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
        fitness = fhd(X, self.func_num)
        conv_curve = []

        for t in range(self.max_iter):
            # Selection
            selected = X[np.argsort(fitness)[:self.pop_size // 2]]
            # Crossover
            offspring = []
```

```
            for i in range(0, len(selected) - 1, 2):
                parent1, parent2 = selected[i], selected[i + 1]
                cross_point = np.random.randint(1, self.dim)
                child1 = np.concatenate((parent1[:cross_point], parent2[cross_point:]))
                child2 = np.concatenate((parent2[:cross_point], parent1[cross_point:]))
                offspring.extend([child1, child2])
            offspring = np.array(offspring)
            # Mutation
            for i in range(len(offspring)):
                if np.random.rand() < self.mutation_rate:
                    mutation_index = np.random.randint(0, self.dim)
                    offspring[i, mutation_index] = np.random.uniform(self.lb, self.ub)
            # Evaluate offspring
            X = np.vstack((selected, offspring))
            fitness = fhd(X, self.func_num)
            best_idx = np.argmin(fitness)
            conv_curve.append(fitness[best_idx])

        best_fitness = np.min(fitness)
        best_solution = X[np.argmin(fitness)]
        return best_fitness, best_solution, np.array(conv_curve)

# GWO implementation
class GWO:
    def __init__(self, pop_size, dim, lb, ub, func_num, max_iter):
        self.pop_size = pop_size
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.func_num = func_num
        self.max_iter = max_iter

    def optimize(self):
        X = np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
        fitness = fhd(X, self.func_num)
        alpha, beta, delta = X[np.argsort(fitness)[:3]]
        conv_curve = []

        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)
            for i in range(self.pop_size):
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = np.abs(C1 * alpha - X[i])
                X1 = alpha - A1 * D_alpha

                r1, r2 = np.random.rand(), np.random.rand()

                A2 = 2 * a * r1 - a
                C2 = 2 * r2
                D_beta = np.abs(C2 * beta - X[i])
                X2 = beta - A2 * D_beta

                r1, r2 = np.random.rand(), np.random.rand()
                A3 = 2 * a * r1 - a
                C3 = 2 * r2
                D_delta = np.abs(C3 * delta - X[i])
                X3 = delta - A3 * D_delta

```

```
                X[i] = (X1 + X2 + X3) / 3
                X[i] = np.clip(X[i], self.lb, self.ub)

            fitness = fhd(X, self.func_num)
            alpha, beta, delta = X[np.argsort(fitness)[:3]]
            conv_curve.append(fitness[0])

        return fitness[0], alpha, np.array(conv_curve)

# DE implementation
class DE:
    def __init__(self, pop_size, dim, lb, ub, func_num, max_iter):
        self.pop_size = pop_size
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.func_num = func_num
        self.max_iter = max_iter
        self.F = 0.5  # Mutation factor
        self.CR = 0.9  # Crossover probability

    def optimize(self):
        X = np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
        fitness = fhd(X, self.func_num)
        conv_curve = []

        for t in range(self.max_iter):
            for i in range(self.pop_size):
                idxs = [idx for idx in range(self.pop_size) if idx != i]
                a, b, c = X[np.random.choice(idxs, 3, replace=False)]
                mutant = np.clip(a + self.F * (b - c), self.lb, self.ub)
                cross_points = np.random.rand(self.dim) < self.CR
                if not np.any(cross_points):
                    cross_points[np.random.randint(0, self.dim)] = True
                trial = np.where(cross_points, mutant, X[i])
                f_trial = fhd(trial[np.newaxis, :], self.func_num).item()
                if f_trial < fitness[i]:
                    X[i] = trial
                    fitness[i] = f_trial

            best_idx = np.argmin(fitness)
            conv_curve.append(fitness[best_idx])

        return fitness[best_idx], X[best_idx], np.array(conv_curve)

# WOA implementation
class WOA:
    def __init__(self, pop_size, dim, lb, ub, func_num, max_iter):
        self.pop_size = pop_size
        self.dim = dim
        self.lb = lb
        self.ub = ub
        self.func_num = func_num
        self.max_iter = max_iter

    def optimize(self):
        X = np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
        fitness = fhd(X, self.func_num)
        best_sol = X[np.argmin(fitness)]
        best_fitness = np.min(fitness)
        conv_curve = []

        for t in range(self.max_iter):
```

```
            a = 2 - t * (2 / self.max_iter)
            for i in range(self.pop_size):
                p = np.random.rand()
                if p < 0.5:
                    A = 2 * a * np.random.rand() - a
                    C = 2 * np.random.rand()
                    D_leader = np.abs(C * best_sol - X[i])
                    X[i] = best_sol - A * D_leader
                else:
                    l = np.random.uniform(-1, 1)
                    D_leader = np.abs(best_sol - X[i])
                    X[i] = D_leader * np.exp(1.5 * l) * np.cos(2 * np.pi * l) + best_sol

                X[i] = np.clip(X[i], self.lb, self.ub)

            fitness = fhd(X, self.func_num)
            current_best_idx = np.argmin(fitness)
            if fitness[current_best_idx] < best_fitness:
                best_fitness = fitness[current_best_idx]
                best_sol = X[current_best_idx]
            conv_curve.append(best_fitness)

        return best_fitness, best_sol, np.array(conv_curve)


# Wrapper for test functions
def fhd(x, func_num):
    return all_functions[func_num - 1](x)


# Algorithms to be compared
algorithms = {
    'CPO': CPO,  # Crested Porcupine Optimizer
    'PSO': PSO,  # Particle Swarm Optimization
    'GA': GA,  # Genetic Algorithm
    'GWO': GWO,  # Grey Wolf Optimizer
    'DE': DE,  # Differential Evolution
    'WOA': WOA  # Whale Optimization Algorithm
}


# Function to run and collect best fitness so far data
def run_algorithm_with_tracking(algorithm, func_num, dim=10, pop_size=130, Tmax=500, ub=100, lb=-100):
    if algorithm == CPO:
        _, _, conv_curve = algorithm(pop_size, Tmax, ub, lb, dim, func_num)
    else:
        instance = algorithm(pop_size, dim, lb, ub, func_num, Tmax)
        _, _, conv_curve = instance.optimize()

    best_fitness_so_far = np.minimum.accumulate(conv_curve)
    return best_fitness_so_far



# Function to display statistics of results with table output
def display_statistics(results):
    stats = []
    print("Algorithm Performance Statistics:")
    for name, conv_curve in results.items():
        best_fitness = conv_curve[-1]
        mean_fitness = np.mean(conv_curve)
```

```
        std_dev = np.std(conv_curve)
        stats.append((name, best_fitness, mean_fitness, std_dev))
        print(f"{name}: Best Fitness = {best_fitness:.2e}, Mean Fitness = {mean_fitness:.2e}, Std Dev = {std_dev:.2e}")
    print()
    return stats

# Enhanced function to perform pairwise Wilcoxon signed-rank tests with formatting
def perform_wilcoxon_test(results):
    algorithms = list(results.keys())
    print("Wilcoxon Test Results:")
    for i in range(len(algorithms)):
        for j in range(i + 1, len(algorithms)):
            algo1, algo2 = algorithms[i], algorithms[j]
            data1, data2 = results[algo1], results[algo2]

            # Ensure that both data arrays have the same length and more than one data point
            if len(data1) > 1 and len(data2) > 1 and len(data1) == len(data2):
                stat, p_value = wilcoxon(data1, data2)
                significant = "Statistically Significant" if p_value < 0.05 else "Not Significant"
                print(f"{algo1} vs {algo2}: p-value = {p_value:.4f} - {significant}")
            else:
                print(
                    f"{algo1} vs {algo2}: Data not suitable for Wilcoxon test (mismatched lengths or insufficient data)")
    print()

    for name, conv_curve in results.items():
        print(f"{name} final fitness values: {conv_curve[-10:]}")


# Plotting function to show convergence curves with shaded error bars
def plot_convergence_curves(results, func_num):
    plt.figure(figsize=(10, 6))
    for name, conv_curve in results.items():
        if len(conv_curve.shape) == 1:  # If conv_curve is a 1D array
            mean_curve = conv_curve
            std_dev_curve = np.zeros_like(mean_curve)  # No std deviation in a single run
        else:  # If multiple runs, calculate mean and std deviation
            mean_curve = np.mean(conv_curve, axis=0)
            std_dev_curve = np.std(conv_curve, axis=0)

        plt.plot(mean_curve, label=name)
        plt.fill_between(range(len(mean_curve)), mean_curve - std_dev_curve, mean_curve + std_dev_curve, alpha=0.2)

    plt.xlabel("Iterations")
    plt.ylabel("Best Fitness So Far")
    plt.yscale('log')
    plt.yticks([10 ** i for i in range(0, 13)])
    plt.title(f"Convergence Curves for Function F{func_num}")
    plt.grid(True, which="both", ls="--", linewidth=0.5)
    plt.legend()
    plt.show()

# Function to plot box plot of final best fitness for each algorithm
def plot_boxplot(results, func_num, lower_percentile=1, upper_percentile=99):
    plt.figure(figsize=(10, 6))
    data = []
```

```
    for name in results.keys():
        fitness_values = results[name]
        lower_bound = np.percentile(fitness_values, lower_percentile)
        upper_bound = np.percentile(fitness_values, upper_percentile)
        filtered_values = fitness_values[(fitness_values >= lower_bound) & (
            fitness_values <= upper_bound)]
        data.append(filtered_values)

    plt.boxplot(data, labels=results.keys())
    plt.yscale('log')
    plt.ylabel("Final Best Fitness")
    plt.title(f"Box Plot of Final Best Fitness for Function F{func_num}")
    plt.show()


# Function to plot histogram of final best fitness for each algorithm
def plot_histogram(results, func_num):
    plt.figure(figsize=(10, 6))
    for name, conv_curve in results.items():
        plt.hist(conv_curve, bins=20, alpha=0.5, label=name)
    plt.yscale('log')
    plt.xlabel("Final Best Fitness")
    plt.ylabel("Frequency")
    plt.title(f"Histogram of Final Best Fitness for Function F{func_num}")
    plt.legend()
    plt.show()

# Run all algorithms and plot the convergence for all test functions with enhanced
    visualization
def run_all_functions_tests(num_funcs=30, dim=10, pop_size=130, Tmax=1000, ub=100,
    lb=-100):
    for func_num in range(1, num_funcs + 1):
        results = {name: run_algorithm_with_tracking(algorithm, func_num, dim,
            pop_size, Tmax, ub, lb)
                   for name, algorithm in algorithms.items()}
        plot_convergence_curves(results, func_num)
        stats = display_statistics(results)
        perform_wilcoxon_test(results)
        plot_boxplot(results, func_num)
        plot_histogram(results, func_num)

run_all_functions_tests()
```

Listing 5: CPO Algorithm and Experimentation Code

The above implementations form part of a comparative study on various optimization algorithms tested on the CEC 2022 benchmark functions. Each algorithm has been optimized to handle different dimensional settings and objective functions effectively.

4.5 Demand

This document summarizes the key Python programming concepts used in the implementation of optimization algorithms, particularly for solving benchmark functions using the Crested Porcupine Optimizer (CPO) algorithm and other similar methods.

4.5.1 Variables and Data Types

```
results_dir = "CPO_results_cec2017"
Gb_Fit = np.inf
Gb_Sol = None
Conv_curve = np.zeros(Tmax)
```

Listing 6: Variable Initialization

4.5.2 Lists

```
Gb_Sol = X[index, :]
y = (X[i, :] + X[rand_index1, :]) / 2
X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol - y
    )
X[i, :] = Xp[i, :]
```

Listing 7: List Manipulations

4.5.3 Dictionaries

```
fun_fitness = {}
print(f"Function F{func_num} Final Best Fitness: {fun_fitness[func_num]:.10e}")
```

Listing 8: Dictionary Usage

4.5.4 Sets

```
functionSET = {'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9',
               'F10', 'F11', 'F12', 'F13', 'F14', 'F15', 'F16', 'F17',
               'F18', 'F19', 'F20', 'F21', 'F22', 'F23', 'F24', 'F25',
               'F26', 'F27', 'F28', 'F29'}
```

Listing 9: Set Operations

4.5.5 Basic Operators

```
X = np.random.rand(pop_size, dim) * (ub - lb) + lb
if np.random.rand() < 0.5:
    X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol
        - y)
```

Listing 10: Basic Mathematical Operations

4.5.6 String Formatting

```
print(f"Running Function F{func_num}...")
print(f"Function F{func_num} Final Best Fitness: {fun_fitness[func_num]:.10e}")
print("All results have been saved in the 'results' folder.")
```

Listing 11: String Formatting with f-Strings

4.5.7 Loops and Flow of Control

```
while t < Tmax and Gb_Fit > opt:
    for i in range(len(X)):
        U1 = np.random.rand(dim) > 0.5
        rand_index1 = np.random.randint(len(X))
        rand_index2 = np.random.randint(len(X))

        if np.random.rand() < 0.5:
            y = (X[i, :] + X[rand_index1, :]) / 2
            X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand()
                * Gb_Sol - y)
        else:
            Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t / Tmax + 1e-6)
            U2 = np.random.rand(dim) < 0.5
            S = np.random.rand() * U2
            if np.random.rand() < 0.8:
                St = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float)
                    .eps)))
                S = S * Yt * St
```

```
                X[i, :] = (1 - U1) * X[i, :] + U1 * (
                    X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
            else:
                Mt = np.exp(-np.abs(fitness[i] / (np.sum(fitness) + np.finfo(float).eps)))
                Vtp = X[rand_index1, :]
                Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
                S = S * Yt * Ft
                X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) *
                        (U2 * Gb_Sol - X[i, :])) - S

        X[i, :] = np.clip(X[i, :], lb, ub)
        nF = fhd(X[np.newaxis, i, :], func_num)
        if fitness[i] < nF:
            X[i, :] = Xp[i, :]
        else:
            Xp[i, :] = X[i, :]
            fitness[i] = nF.item()
            if nF <= Gb_Fit:
                Gb_Sol = X[i, :]
                Gb_Fit = nF.item()

    Conv_curve[t] = Gb_Fit
    t += 1
```

Listing 12: Loops and Control Structures

4.5.8 Modules and Packages

```
import os
import numpy as np
import matplotlib.pyplot as plt
from cec2017.functions import all_functions
```

Listing 13: Modules and Packages

5. Conclusions

In this project, we successfully implemented and replicated the **Crested Porcupine Optimizer (CPO)** based on the work of Mohamed Abdel-Basset et al. [7] We encountered several challenges during the implementation and testing of the algorithm, which were addressed through various strategies. These challenges ranged from issues with multi-dimensional array handling to integrating external libraries and optimizing code performance. The lessons learned from these experiences contributed significantly to the success of the project.

5.1 Issues Encountered

Ziheng Wang's Challenges:

- **MATLAB to Python Translation:** One significant challenge involved translating the CPO algorithm from MATLAB to Python. The differences in handling multidimensional arrays, where MATLAB stores matrices by column and Python (via NumPy) stores them by row, required adjustments in array indexing and slicing operations. Additionally, MATLAB uses 1-based indexing, while Python uses 0-based indexing, which required us to carefully adjust the indices in the code.
- **Handling Array Broadcasting:** While MATLAB does not natively support array broadcasting, Python's NumPy library does. This difference affected the way we handled operations between arrays, and we solved this by utilizing NumPy's broadcasting feature, which allowed us to simplify and optimize array operations.

- **Function Handles in Python:** In MATLAB, function handles are used seamlessly, but Python's function references require extra attention. We encountered challenges when passing functions as parameters and solved this by using Python's 'if' and 'elif' statements to select the appropriate functions dynamically.
- **LaTeX Formatting Issues:** While preparing the final report, we faced difficulties in formatting equations and handling LaTeX references. Ensuring correct equation formatting and generating a dynamic table of contents required careful attention to LaTeX syntax. We resolved these issues by using appropriate LaTeX packages such as 'graphicx', 'tabular', and 'booktabs' for figures and tables.
- **Path Planning Challenges:** While applying the CPO algorithm to path planning, we faced challenges related to grid-based movement constraints. Specifically, ensuring that the algorithm adhered to right-angle turns while avoiding diagonal movements was crucial. Additionally, the adaptive search mechanisms of CPO needed to be fine-tuned to balance exploration and exploitation within a constrained grid environment. This required integrating CPO's exploration phase with A* for initial pathfinding, followed by optimization steps to refine the path.

Di Wu's Challenges:

- **Integrating CEC Libraries:** Importing and using libraries like 'cec2017' and 'cec2022' was challenging due to compatibility issues between the Python versions and the libraries. We searched for these libraries on GitHub, downloaded the corresponding files, and manually imported them into our Python environment. We also adjusted the input matrices to fit the expected formats of these functions.
- **Programming Details:** Di Wu focused on the implementation details of the Python code, including handling function references, parameter passing, and ensuring correct execution in the optimization algorithm. These issues were solved by utilizing Python's flexible 'lambda' functions and 'functools.partial' for handling dynamic function calls.
- **Path Planning Implementation:** Di Wu also worked on the A* path planning algorithm, integrating CPO with A* to improve path optimization. Issues such as handling obstacles, ensuring paths do not overlap, and dealing with edge cases were resolved by refining the neighbor selection logic and adjusting the grid-checking mechanisms.
- **Processing of operational results** When migrating MATLAB code to Python, in addition to visual differences, special attention should be paid to saving the running results. MATLAB and Python have significant differences in file saving formats, data storage methods, and reusability. We solve the problem of saving running results by searching online.

5.2 Reflections

Ziheng Wang's Reflection: Throughout this project, I learned the complexities involved in transferring an optimization algorithm from MATLAB to Python. The most significant challenge was understanding and handling the differences in array operations and indexing between the two languages. Working with external libraries, debugging the LaTeX report, and refining the algorithm's implementation deepened my understanding of both coding and scientific documentation. This experience has also strengthened my problem-solving skills, particularly in dealing with numerical optimization and performance issues.

Di Wu's Reflection: This project was a valuable learning experience for me in Python programming, especially in optimizing algorithms and handling large datasets. I gained insights into the importance of fine-tuning optimization algorithms and ensuring robustness when integrating different techniques. The path planning implementation provided an opportunity to apply theoretical knowledge to a practical scenario, further enhancing my understanding of algorithmic design and optimization.

5.3 Acknowledgment

We would like to express our sincere gratitude to Dr. Chengzhi Qu from Nanjing university of information science and technology for his invaluable guidance and for providing the topic for this report. His support and advice were instrumental in the successful completion of this research.

5.4 Future Work and Open Source Contribution

In the spirit of open-source collaboration and to facilitate future research, we have uploaded the implementation of the CPO algorithm, along with its applications in optimization and path planning, to GitHub. The code repository can be accessed at: https://github.com/Nickory/CPO-python. This also includes the path planning based on CPO, allowing others to explore and further develop this approach. We plan to continuously update, improve, and maintain this repository to ensure that it remains a valuable resource for both researchers and practitioners.

References

[1] Zhi-Hui Zhan, Lin Shi, Kay Chen Tan, and Jun Zhang. A survey on evolutionary computation for complex continuous optimization. *Artificial Intelligence Review*, 55(1):59–110, 2022.

[2] Chengzhi Qu, Wendong Gai, Jing Zhang, and Maiying Zhong. A novel hybrid grey wolf optimizer algorithm for unmanned aerial vehicle (uav) path planning. *Knowledge-Based Systems*, 194:105530, 2020.

[3] Chengzhi Qu, Wendong Gai, Maiying Zhong, and Jing Zhang. A novel reinforcement learning based grey wolf optimizer algorithm for unmanned aerial vehicles (uavs) path planning. *Applied soft computing*, 89:106099, 2020.

[4] Stephanie Forrest. Genetic algorithms. *ACM computing surveys (CSUR)*, 28(1):77–80, 1996.

[5] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.

[6] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[7] Mohamed Abdel-Basset, Reda Mohamed, and Mohamed Abouhawwash. Crested porcupine optimizer: A new nature-inspired metaheuristic. *Knowledge-Based Systems*, 284:111257, 2024.

[8] Guohua Wu, Rammohan Mallipeddi, and Ponnuthurai Nagaratnam Suganthan. Problem definitions and evaluation criteria for the cec 2017 competition on constrained real-parameter optimization. *National University of Defense Technology, Changsha, Hunan, PR China and Kyungpook National University, Daegu, South Korea and Nanyang Technological University, Singapore, Technical Report*, 2017.

[9] Ali Ahrari, Saber Elsayed, Ruhul Sarker, Daryl Essam, and Carlos A Coello Coello. Problem definition and evaluation criteria for the cec'2022 competition on dynamic multimodal optimization. In *Proceedings of the IEEE World Congress on Computational Intelligence (IEEE WCCI 2022), Padua, Italy*, pages 18–23, 2022.

[10] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.