INFORMATICA, 2025, Vol. 36, No. 3, 677–712

© 2025 *Vilnius University*
DOI: https://doi.org/10.15388/25-INFOR603

Performance Comparison of Single-Objective Evolutionary Algorithms Implemented in Different Frameworks

Miha RAVBER*, Marko ŠMID, Matej MORAVEC, Marjan MERNIK, Matej ČREPINŠEK

University of Maribor, Faculty of Electrical Engineering and Computer Science, Koroška cesta 46, 2000 Maribor, Slovenia
e-mail: miha.ravber@um.si, marko.smid2@um.si, matej.moravec@um.si, marjan.mernik@um.si, matej.crepinsek@um.si

Received: December 2024; accepted: September 2025

Abstract. Fair comparison with state-of-the-art evolutionary algorithms is crucial, but is obstructed by differences in problems, parameters, and stopping criteria across studies. Metaheuristic frameworks can help, but often lack clarity on algorithm versions, improvements, or deviations. Some also restrict parameter configuration. We analysed source codes and identified inconsistencies between implementations. Performance comparisons across frameworks, even with identical settings, revealed significant differences, sometimes even with the authors' own code. This questions the validity of comparisons using such frameworks. We provide guidelines to improve open-source metaheuristics, aiming to support more credible and reliable comparative studies.

Key words: metaheuristics, evolutionary algorithms, metaheuristic optimization framework, algorithm comparison, benchmarking.

1. Introduction

Accurate and precise experimentation is critical when comparing evolutionary algorithms, ensuring validity for both researchers and practitioners. Such comparisons are supported by established benchmarking guidelines (Bartz-Beielstein *et al.*, 2020; Eiben and Jelasity, 2002; LaTorre *et al.*, 2020; Črepinšek *et al.*, 2014; Ravber *et al.*, 2016, 2022b). Numerous open-source metaheuristic frameworks have been developed to support research efforts. These provide widely used algorithms and standard test problems, and are implemented in various programming languages to facilitate rapid development and comparison (Molina *et al.*, 2020; Silva *et al.*, 2018; Parejo *et al.*, 2012; Dzalbs, 2021; Osaba *et al.*, 2021). Consequently, such frameworks have gained significant popularity within the research community.

*Corresponding author.

Framework selection is often driven by the user's preferred programming language (Oztas and Erdem, 2021). However, beyond language, the correctness of algorithm implementations and other framework components is crucial. Variations in implementation can lead to significant differences in performance, even when using identical parameters.

This study addresses the research question: To what extent does the implementation of a metaheuristic affect its problem-solving performance? We investigate this by focusing exclusively on the quality of solutions across multiple frameworks. Computational performance and resource usage have already been examined in prior work (Merelo-Guervós *et al.*, 2016a; Villalobos *et al.*, 2018). Our analysis includes reviewing the algorithm source codes within each framework to identify deviations from the original implementations, and evaluate whether the observed performance aligns with the intentions of the original algorithm authors. The primary objectives of this work are:

1. To compare the performance and functionality of evolutionary algorithms implemented across different frameworks, while excluding other components of the frameworks.
2. To analyse the source code of each implementation to explain the observed performance differences.

We selected 13 diverse metaheuristic frameworks (DEAP, EARS, jMetal, MEALPY, MOEA, NiaPY, pagmo2, PlatEMO, YPEA, Nevergrad, metaheuristicOpt, EvoloPy, and pymoo) for comparison, chosen based on their widespread use, as indicated by citations and repository stars. We compared six single-objective algorithms, which were implemented widely across the selected frameworks: Artificial Bee Colony (ABC) (Karaboga and Basturk, 2007), Differential Evolution (DE) (Storn and Price, 1997), Genetic Algorithm (GA) (Holland, 1992), Grey Wolf Optimizer (GWO) (Mirjalili *et al.*, 2014), Particle Swarm Optimization (PSO) (Kennedy and Eberhart, 1995), and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen and Ostermeier, 2001). Notably, these algorithms are among the most widely recognized and utilized within the evolutionary computation community, as evidenced by the numerous citations of their original works (Ma *et al.*, 2023; Wang *et al.*, 2021; Dokeroglu *et al.*, 2019; Darwish, 2018; Tzanetos and Dounias, 2020; Molina *et al.*, 2020; Wang *et al.*, 2025). Additionally, many new metaheuristics are based on these algorithms (Velasco *et al.*, 2024). When possible, we included original implementations published by the algorithm authors; however, for GA and PSO, such official source code was not available.

To ensure fairness in comparison, we validated the implementation of all test problems carefully across all frameworks. A statistical analysis was conducted using the EARS framework, which supports reproducible and unbiased evaluation. We adopted the Chess Rating System for Evolutionary Algorithms (CRS4EAs) (Veček *et al.*, 2014), for performance comparison. Additionally, the Comparing Continuous Optimizers (COCO) platform (Hansen *et al.*, 2021) was used to plot runtime profiles, specifically, Empirical Cumulative Distribution Functions (ECDF), to analyse algorithm performance further. The main contributions of this work are:

- A comprehensive performance comparison of widely used metaheuristic algorithms across 13 frameworks, incorporating the original authors' implementations when available.
- A detailed analysis of source code discrepancies between the framework implementations and original algorithm descriptions, shedding light on how variations arise.
- A demonstration that even minor differences in implementation can lead to significant performance changes, highlighting the importance of correctness in algorithm comparisons.
- Practical recommendations for improving open-source metaheuristic implementations to support more consistent and trustworthy benchmarking studies.

The remainder of this paper is organized as follows. Section 2 provides a literature review on comparative studies of metaheuristic frameworks. Section 3 introduces the selected frameworks. Section 4 outlines the experimental design. The results and analysis are presented in Section 5. Section 6 concludes the paper and suggests directions for future work.

2. Related Work

Several studies have explored the characteristics of metaheuristic frameworks, focusing on aspects such as code understandability, supported algorithms, parallelization capabilities, platform compatibility, visualization support, tuning options, solution representation, statistical analysis, programming language, and execution speed (Dzalbs, 2021; Osaba *et al.*, 2021; Parejo *et al.*, 2012; Silva *et al.*, 2018; Ramírez *et al.*, 2023). While these works provide valuable insights into the features and design of metaheuristic frameworks, they do not directly address the performance implications of different algorithm implementations within these frameworks. To address this gap, our study compares implementations of metaheuristic algorithms across multiple frameworks, and the following papers are some of the most relevant to our work.

In Biedrzycki (2021), Biedrzycki compared official implementations of the CMA-ES algorithm, sourced from the author's homepage, revealing substantial performance differences between versions. These findings highlight that the choice of implementation can impact research outcomes significantly. To enhance research credibility, the study proposes guidelines to improve publication quality and reduce erroneous claims of algorithm superiority. Unlike Biedrzycki's focus on a single algorithm (CMA-ES) and its versions, our work examines discrepancies in the implementations of six algorithms, including CMA-ES, across 13 metaheuristic frameworks. By comparing CMA-ES implementations across diverse frameworks, we anticipate similar, or even greater performance variations, due to the broader scope of the framework-specific differences, providing a more comprehensive analysis of the implementation impacts.

In the paper Merelo-Guervós *et al.* (2016a), the authors compared the performance of various programming languages in implementing basic evolutionary algorithm operations. Typically, compiled languages like Java or C/C++ are favoured for such tasks, but this research explored the speed of both popular and less common languages. The results

show that compiled languages generally perform better, but languages like Go and Python can also be fast enough for most purposes. The choice of data structures and code layout impacts performance significantly. There is no one-size-fits-all language, but Java with BitSets or C# tends to perform well. The study suggests exploring hybrid architectures combining languages for optimal performance and prototyping benefits, fostering future research in this direction. The paper emphasizes primarily the execution speed of the algorithms, whereas our article focuses on examining the implementation differences that impact the algorithm's performance on benchmark problems.

In the paper Merelo-Guervós *et al.* (2016b), the authors evaluated the performance of various programming languages in implementing three common genetic algorithm operations: two genetic operators (mutation and crossover) and OneMax, a benchmark used in both practical and theoretical approaches to genetic algorithms. Their aim was to provide insights for practitioners in choosing languages for evolutionary algorithm implementations. Java was generally considered the fastest, but Clojure was able to surpass it in some functions. Functional languages showed remarkable performance, despite their limited popularity in the evolutionary algorithm community. Using a vector of bits instead of bit strings or lists generally yielded better performance, and immutable data structures, common in functional languages, are faster when available. The study suggested future research directions, including hybrid language architectures for improved performance and a multi-language framework. The focus of the paper lies on evaluating programming languages for genetic algorithm operations, shedding light on the language choice for evolutionary algorithm implementations and how it affects efficiency, whereas our work centres on comparing the performance of the same algorithms implemented in different frameworks.

In the paper Alba *et al.* (2007), the authors examined the impact of different Java data structure implementations on the performance of an evolutionary algorithm. While most evolutionary algorithm studies focus on abstract data representations, this research delved into how these representations were implemented in programming languages, specifically Java. The study compared six implementations of a steady-state genetic algorithm, divided into two groups, based on population representation (array or vector) and individual representation. The results have shown that the choice of data structure for population representation affects the implementation time significantly, while the gene type had minimal impact. The study emphasized the importance of considering data implementation in EA research, and suggests further exploration in this area. However, the study only considered the efficiency of evolutionary algorithms, which differs from our paper's primary focus on evaluating their impact on performance.

In the paper Merelo *et al.* (2011), the authors highlighted the significance of efficient implementation in evolutionary algorithms, emphasizing that improvements in implementation often have a more substantial impact on performance than changes to the algorithm itself. The study applied standard methodologies for performance enhancement to EAs, and identified implementation options that yielded the best results for specific problem configurations, especially when factors like population or chromosome size increased. The findings demonstrated that applying profilers to identify bottlenecks in EA implementations and then optimizing those code segments through informed programming, can

enhance running times significantly without sacrificing algorithmic performance. Additionally, various techniques, like multithreading (Kovačević *et al.*, 2022), and message passing, can improve EA performance further, expanding their applicability. The study highlighted the significance of efficient implementation in evolutionary algorithms, concentrating on optimizing the implementation for specific problem configurations, which differs from our paper's primary objective of benchmarking different implementations on various problems.

3. Metaheuristic Optimization Frameworks

This section presents the characteristics of the frameworks included in our comparison. We reviewed frameworks supporting continuous single-objective optimization problems. Due to the numerous available frameworks, we limited our study to 13 widely used frameworks with publicly available source codes. The selected frameworks and their characteristics, current as of July 2025, are detailed in Table 1. The table includes the latest release, repository stars, number of citations of the associated paper (as reported by Google Scholar), number of implemented algorithms (both single- and multi-objective), programming language, and the official publication. When evaluating the popularity of the frameworks, the citation count revealed that the three most well-received frameworks (DEAP, pymoo, and PlatEMO) were featured in the comparative analysis. Furthermore, DEAP, Nevergrad, and pymoo were ranked first, second, and third, respectively, in terms of popularity based on the repository stars.

The algorithms selected for comparison are shown in Table 2. Differential Evolution was implemented in all 13 frameworks, whereas the other algorithms varied in availability. Notably, CMA-ES implementations in several frameworks contained errors, such as infinite loops or crashes, likely due to the algorithm's complexity. Consequently, these flawed CMA-ES implementations were excluded from the comparison.

4. Experimental Methodology

In the experiments, the algorithms shared the same control parameters, as listed in Table 3. These control parameters were set based on the authors' recommendations, and were not subject to optimization. Importantly, the parameter settings were standardized across all the algorithms to ensure that they did not impact the comparison.

The DE strategy employed in our study was rand/1/bin. In contrast, GAs lack a standardized strategy but offer flexibility in choosing operators for selection, mutation, and crossover. Not all frameworks supported the same operators. For our comparison, we selected tournament selection, Polynomial Mutation (PM) (Deb and Deb, 2014), and Simulated Binary Crossover (SBX) (Deb *et al.*, 1995), as they were the most prevalent across the selected frameworks. We omitted five of the 11 frameworks that did not support all chosen operators, as their GA implementations differed significantly and did not represent

Table 1
Selected frameworks with their characteristics.

Framework	Latest release	Repository stars	Number of citations	Number of algorithms	Programming language	Publication	Source code
DEAP	1.4.1, Jul, 2023	6137	2830	6	Python	(Fortin *et al.*, 2012)	github.com/DEAP/deap
EARS	v4.0.0, Nov, 2024	22	163	56	Java	(Veček *et al.*, 2014)	github.com/UM-LPM/EARS
jMetal	jmetal-6.7, May, 2025	535	1460	33	Java	(Durillo and Nebro, 2011)	github.com/jMetal/jMetal
MEALPY	v3.0.2, May, 2025	1045	204	129	Python	(Van Thieu and Mirjalili, 2023)	github.com/thieu1995/mealpy
MOEA	Version 5.1, Jun, 2025	339	101	26	Java	(Hadka, 2014)*	github.com/MOEAFramework/MOEAFramework
NiaPY	2.5.2, Jan, 2025	267	98	37	Python	(Vrbančič *et al.*, 2018)	github.com/NiaOrg/NiaPy
pagmo2	pagmo 2.19.1, Aug, 2024	879	325	352	C++	(Biscani and Izzo, 2020)	github.com/esa/pagmo2
PlatEMO	PlatEMO v4.12, Apr, 2025	1872	2391	322	MATLAB	(Tian *et al.*, 2017)	github.com/BIMK/PlatEMO
EvoloPy	N/A	476	130	14	Python	(Faris *et al.*, 2016)	github.com/7ossam81/EvoloPy
metaheuristicOpt	2.0.0, Jun, 2019	4	17	20	R	(Riza *et al.*, 2018)	github.com/cran/metaheuristicOpt
Nevergrad	1.0.12, Apr, 2025	4094	336	542	Python	(Rapin and Teytaud, 2018)*	github.com/facebookresearch/nevergrad
pymoo	0.6.1.15, May, 2025	2581	2235	20	Python	(Blank and Deb, 2020)	github.com/anyoptimization/pymoo
YPEA	N/A	49	13	15	MATLAB	(Kalami Heris, 2019)*	github.com/smkalami/ypea

* No official reference found.

Table 2
Implemented optimization algorithms in each framework.

Framework \ Algorithm	ABC	DE	GA	GWO	PSO	CMA-ES
DEAP		✓	✓		✓	✓
EARS	✓	✓		✓	✓	✓*
jMetal		✓	✓		✓	✓
MEALPY	✓	✓	✓	✓	✓	✓*
MOEA		✓	✓			✓*
NiaPY	✓	✓	✓	✓	✓	
pagmo2	✓	✓	✓	✓	✓	✓
PlatEMO	✓	✓	✓	✓	✓	✓*
YPEA	✓	✓	✓		✓	✓*
Nevergrad		✓			✓	✓*
metaheuristicOpt	✓	✓	✓	✓	✓	
EvoloPy		✓	✓	✓	✓	
pymoo		✓	✓		✓	✓

* Algorithm was implemented but did not work.

Table 3
Parameter settings of all the algorithms.

ABC	DE	GA	GWO	PSO	CMA-ES
pop_size = 125 L = 100	pop_size = 50 CR = 0.9 F = 0.5	pop_size = 100 $p_c = 0.95$ $p_m = 0.025$ $\eta c = \eta m = 20$	pop_size = 30	pop_size = 30 $\omega = 0.7$ c1 = 2 c2 = 2	pop_size = 30 $\sigma = 0.5$ x_0 = random position

pop_size – population size, L – stagnation limit, CR – crossover probability, F – mutation factor, p_c – crossover probability, p_m – mutation factor, ηc – crossover distribution index, ηm – mutation distribution index, ω – inertia weight, $c1$ – cognitive coefficient, $c2$ – social coefficient, σ – initial Standard Deviation in each coordinate, x_0 – initial guess of the minimum solution.

the same method. These choices were made to try to ensure a consistent and standardized comparison of algorithms across the different frameworks.

During the experiments, we used a set of 12 continuous single-objective problems, detailed in Table 4, which are widely recognized benchmarks for evaluating optimization algorithm performance. To ensure sufficient challenge, we employed shifted versions of some problems, to address the centre bias in algorithms like GWO (Morales-Castañeda *et al.*, 2025; Camacho Villalón *et al.*, 2020; Niu *et al.*, 2019).

Appropriate stopping criteria lack specific guidelines and depend on problem complexity and desired solution quality. The studies (Sahin and Akay, 2016; Lee *et al.*, 2020; Ezugwu *et al.*, 2020), showed varied approaches, with evaluation counts ranging from 5,000 to 500,000. We applied a uniform stopping criterion of 15,000 evaluations across all problems. However, it is important to note that some of the authors' implementations, as well as the DEAP, MOEA, pagmo2, metaheuristicOpt and EvoloPy frameworks, do not support the use of the maximum number of evaluations inherently as the stopping

Table 4
Continuous single-objective optimization problems, including dimensions, bounds, and global optima.

Problem function	d	Range	$f_{\min}$
$f_1(x) = \sum_{i=1}^{d} x_i^2$	60	$[-100, 100]$	0
$f_2(x) = \sum_{i=1}^{d} i x_i^2$	60	$[-100, 100]$	0
$f_3(x) = \sum_{i=1}^{d} \left(\sum_{j=1}^{i} x_i\right)^2$	60	$[-100, 100]$	0
$f_4(x) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$	60	$[-5.12, 5.12]$	0
$f_5(x) = -20\exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\right) + 20 + \exp(1)$	60	$[-32, 32]$	0
$f_6(x) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	60	$[-600, 600]$	0
$f_7(x) = \sum_{i=1}^{d-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right]$	60	$[-30, 30]$	0
$f_8(x) = \left[\frac{1}{500} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2}(x_i - a_{i,j})^6}\right]^{-1}$	2	$[-65.536, 65.536]$	0.998
$f_9(x) = 4x_1^2 + x_1x_2 - 4x_2^2 - 2.1x_1^4 + 4x_2^4 + \frac{1}{3}x_1^6$	2	$[-5, 5]$	-1.031
$f_{10}(x) = 1\left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{(8\pi)}\right)\cos(x_1) + \frac{5}{\pi}$	2	$[-5, 0] \times [10, 15]$	0.397
$f_{11}(x) = \left[1 + (x_1 + x_2 + 1)^2\left(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2\right)\right] \times \left[30 + (2x_1 - 3x_2)^2\left(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2\right)\right]$	2	$[-2, 2]$	3
$f_{12}(x) = -\sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{3} A_{ij}(x_j - P_{ij})^2\right)$	3	$[0, 1]$	-3.862

d – denotes the number of dimensions of the problem.
Shifted functions (f_1 to f_6): $f(x) = f_{\text{base}}(x - o)$, where $o_i \sim U(0.8 \cdot lb_i, 0.8 \cdot ub_i)$.
f_1 – Shifted Sphere, f_2 – Shifted Sum of Squares, f_3 – Shifted Schwefel02, f_4 – Shifted Rastrigin, f_5 – Shifted Ackley, f_6 – Shifted Griewank, f_7 – Rosenbrock, f_8 – Shekel's Foxholes, f_9 – Six-Hump Camel Back, f_{10} – Branin, f_{11} – Goldstein-Price, f_{12} – Hartman.

criterion. Instead, they offer the option to set the maximum number of generations or iterations, which is not recommended, due to variations in the number of evaluations consumed per generation by different algorithms (Ravber *et al.*, 2022b; Mernik *et al.*, 2015). To address this and ensure a fair comparison, we integrated a custom evaluation counter into each framework, to monitor the exact number of evaluations performed, and detect any instances where the limit was exceeded and by how much.

Once all the results were collected, we conducted the statistical analysis using the EARS framework. EARS is a specialized framework developed specifically for benchmarking evolutionary algorithms (EARS, 2019). It offers a comprehensive suite of tools and utilities that facilitate the setup and execution of benchmarks, as well as the collection and analysis of results. Designed with the goal of ensuring fairness in comparisons between evolutionary algorithms, EARS provides researchers with a reliable and standardized platform for conducting rigorous evaluations. All algorithms in EARS are based on the authors' source codes, except CMA-ES, which uses the MOEA Framework implementation.

To analyse the runtime performance of each algorithm, we tracked their solution quality improvements throughout their runs across the 12 benchmark problems. We utilized cocoviz,[1] a minimalist visualization toolkit provided by the COCO platform (Hansen *et al.*, 2021), to generate Empirical Cumulative Distribution Functions. These ECDF plots illustrate the proportion of runs achieving specified target values within the given evaluation budgets, enabling detection of performance differences per problem, and highlighting framework-specific variations in algorithm efficiency.

EARS utilizes a novel method called CRS4EAs to facilitate the comparison of evolutionary algorithms. The chess rating system employed in CRS4EAs is Glicko-2 (Glickman, 2012). In a study by Veček *et al.* (2017), CRS4EAs was shown to be comparable to statistical tests like the Friedman test followed by the Nemenyi test. In another research paper by Eftimov and Korošec (2019), they established that the CRS4EAs approach was also comparable to the pDSC (Practical Deep Statistical Comparison) method. These studies provided evidence of the effectiveness and reliability of CRS4EAs in facilitating accurate and robust comparisons of evolutionary algorithms.

In the CRS4EAs approach, the algorithms are represented as chess players who participate in a tournament. In this tournament, the algorithms compete against each other in pairs, with each algorithm playing multiple games (N times) against all the other algorithms for a given problem. To ensure robust and reproducible results, we conducted 50 games (independent runs) per algorithm for each problem, a conservative choice exceeding the typical 15–30 runs used in metaheuristics research (Hansen *et al.*, 2010; LaTorre *et al.*, 2020; Ezugwu *et al.*, 2020). The CRS4EAs approach, which does not assume normality in the performance data, further enhances reliability, as it produces consistent rankings even with fewer runs (Veček *et al.*, 2017). All the games were conducted under identical conditions, with the same stopping criterion applied to each algorithm. The objective for each algorithm was to obtain the best possible solution for the given problem. Following the games, the solutions were compared to determine the game outcomes, which could be a win, a loss, or a draw for each algorithm. Specifically, for each problem instance, the outcome of a game between two algorithms was determined by comparing their fitness values (e.g. objective function values). A win was assigned to the algorithm with the better fitness value, a loss to the algorithm with the worse fitness value, and a draw was declared if the absolute difference between their fitness values was less than a predefined threshold (draw limit), $\epsilon = 10^{-8}$. This threshold, consistent with the precision used in CEC competitions and in COCO (Hansen *et al.*, 2021), ensures that negligible differences in fitness values are treated as draws, avoiding overemphasis on numerically insignificant variations due to floating-point precision or problem-specific tolerances. After the tournament, the results were utilized to calculate the Rating (R), Rating Deviation (RD), and Rating Interval (RI) for each algorithm. The final outcome was a ranking of algorithms based on their ratings, with higher ratings indicating better performance, reflecting the algorithm's relative strength. The rating deviation indicates the reliability of an algorithm's rating. Rating intervals, calculated as [R – 2RD, R + 2RD], provide a 95% confidence

[1] https://github.com/numbbo/coco-visualize

level that the true rating lies within this interval, based on the empirical 68–95–99.7 rule, assuming a normal distribution for rating deviations, as standard in the Glickman rating system (Glickman, 1999). The 95% confidence level ($\alpha = 0.05$) was chosen for its balance between statistical rigour and interpretability, as it is a common standard in significance testing, unlike the more conservative 99.7% level ($\alpha = 0.01$). Statistical significance was determined by evaluating whether the rating intervals of the compared algorithms overlapped. Non-overlapping intervals indicate significant performance differences between implementations. This approach ensures robust and statistically significant conclusions in comparing evolutionary algorithms using the CRS4EAs method.

5. Experimental Results

As mentioned previously, the rating intervals were computed using the EARS framework. Each algorithm participated in a tournament, resulting in six rating charts containing the rating intervals.

It is important to note that the charts depicting the rating intervals may contain a different number of algorithms, as not every framework included all of the selected algorithms. Additionally, the absence of the authors' versions of GA and PSO contributed further to variations in the chart compositions. The remaining algorithms, namely, ABC, DE, GWO, and CMA-ES, have their source code accessible on the authors' official websites, and they are available in different programming languages. However, in the comparison, we included only one version (one programming language) of each algorithm from the authors' implementations. We chose the canonical version of each algorithm without modifications, because the specific implementation choice can affect the results significantly (Biedrzycki, 2021). For ABC, we utilized the Matlab version, as it was the first version available on the authors' homepage and bears the authors' signature in the code. Similarly, for DE, we selected the Java version, since, from the comments in the code, it is evident that this version was written by one of the original authors. Regarding GWO, the authors' homepage offers links for GWO implementations in seven different languages. Nevertheless, we chose to use the Matlab version, as it was the one written by the original author. For CMA-ES, there were also implementations available in multiple languages on the author's homepage. We selected the Python implementation, as it is maintained actively by the authors (Hansen *et al.*, 2019).

Figure 1 illustrates the rating and rating intervals for ABC implementations in seven different frameworks, as well as the author's implementation in Matlab (ABC-Author-Matlab). The concept of rating intervals and how they were calculated is explained in the Experimental Methodology (Section 4). For clarity, the same methodology was applied to all the algorithms, and their rating intervals are presented in subsequent figures. The implementations are ranked based on their ratings, with the best performing implementation placed in the top right corner, and the worst performing in the bottom left corner. From the chart, we can determine that the implementation in the EARS framework (ABC-EARS) achieved the highest rating, indicating superior performance, while the implementation in the metaheuristicOpt framework (ABC-metaheuristicOpt) obtained the lowest rating.

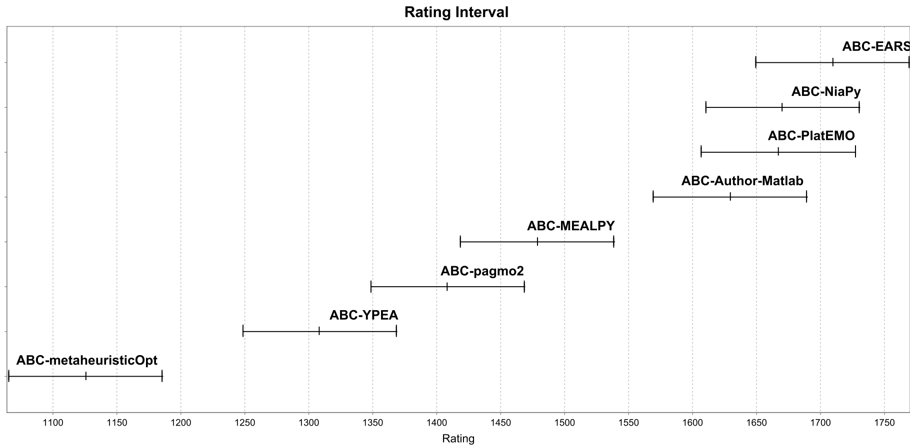

Fig. 1. Rating intervals of ABC implemented in different frameworks, including the author's implementation.

A comprehensive comparison of algorithms considers both their ratings and rating intervals. Non-overlapping rating intervals indicate statistically significant performance differences at the 95% confidence level. In Fig. 1, ABC-EARS, ABC-NiaPY, ABC-PlatEMO, and ABC-Author-Matlab significantly outperform ABC-MEALPY, ABC-pagmo2, ABC-YPEA, and ABC-metaheuristicOpt. ABC-MEALPY is significantly better than ABC-YPEA and ABC-metaheuristicOpt. Additionally, ABC-pagmo2 and ABC-YPEA are significantly better than ABC-metaheuristicOpt. However, partial overlap in rating intervals, such as between ABC-EARS and ABC-NiaPy, suggests smaller performance differences that may not always be statistically significant. For a detailed analysis of such cases, the ECDF plots provide further insights into performance variations.

It is important to recognize that, in stochastic algorithms, the ranking of similarly performing algorithms could vary if the experiment is rerun. The degree of overlap in the rating intervals affects the likelihood of this variability occurring. The more the intervals overlap, the smaller the observed differences in performance, and the higher the possibility of changes in the ranking upon rerunning the experiment. For example, when we examined the rating intervals of ABC-NiaPY and ABC-PlatEMO, we noticed that they had a large overlap of intervals, suggesting that their rankings could vary if the experiment were to be repeated. In contrast, when we considered the rating intervals of ABC-pagmo2 and ABC-YPEA, we observed much less overlap, indicating that their rankings are less likely to change.

For a more detailed analysis of the performance of different implementations, we plotted runtime profiles for each problem individually using cocoviz. The x-axis in the figures represents the budget, defined as the number of function evaluations, while the y-axis shows the fraction of target values reached. For each problem, the target value corresponded to its global optimum; therefore, reaching the target value indicates that the implementation found the global optimum.

The runtime profiles for all 12 problems for ABC are shown in Fig. 2. The 60-dimensional (60D) problems proved to be particularly challenging, as none of the imple-

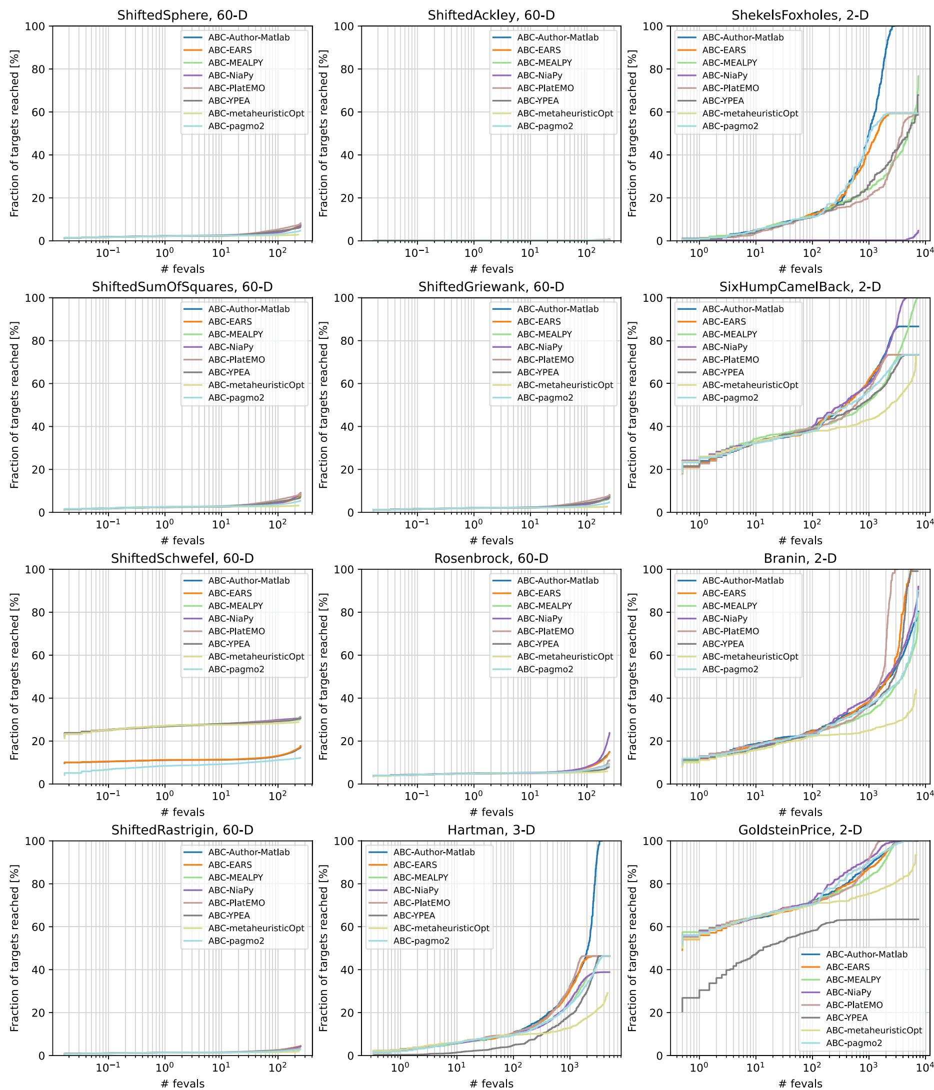

Fig. 2. Profiles per problem for the ABC implementations.

mentations reached the global optimum. Among them, the Shifted Ackley problem was the most difficult. In contrast, the 2-dimensional (2D) Goldstein–Price problem was the least challenging: six out of the eight implementations found the global optimum, with PlatEMO converging the fastest. Interestingly, for the Shekel's Foxholes and Hartman problems, only the authors' implementation reached the global optimum. Among the 60D problems, the best performance was observed on the Shifted Schwefel problem, although the EARS, pagmo2, and the authors' implementations performed slightly worse than the others.

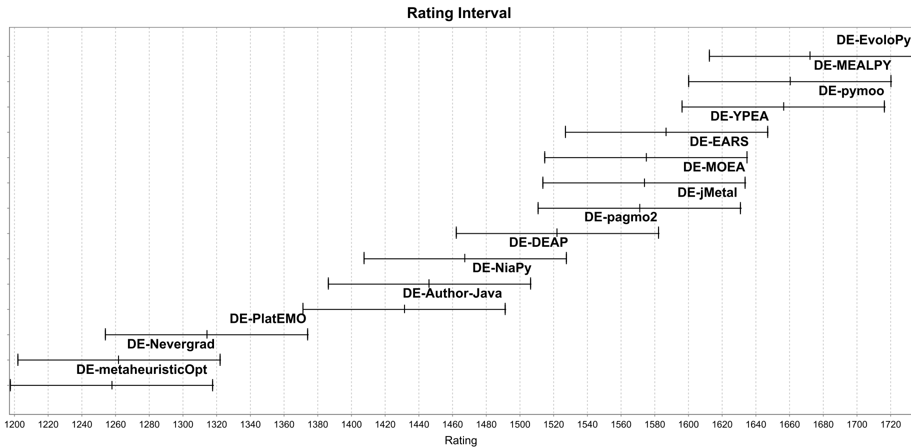

Fig. 3. Rating intervals of DE implemented in different frameworks, including the authors' implementation.

Figure 3 illustrates the rating intervals for DE across all 13 frameworks, including the authors' Java implementation (DE-Author-Java). The rating interval of the author's implementation overlaps with only four other implementations: PlatEMO, NiaPy, DEAP, and pagmo2. Among these, NiaPy showed the greatest degree of overlap, suggesting minimal deviation from the authors' code. In contrast, although PlatEMO and the authors' implementation were not significantly different in performance, their intervals barely overlapped, indicating greater deviation. The rating intervals of nine implementations did not overlap with the author's implementation at all, indicating substantial differences. The top three ranked implementations stood out the most, outperforming six other implementations as well as the authors' version.

The runtime profiles for DE are presented in Fig. 4. Compared to the ABC implementations, the DE implementations performed better overall. However, no implementation was able to find the global optimum for any of the 60D problems. Among the 2D problems, only YPEA failed to find the global optimum for Goldstein–Price, and only Nevergrad and metaheuristicOpt failed to do so for Branin. For the Six-Hump Camel problem, six out of the 14 implementations reached the optimum. For Shekel's Foxholes, only three implementations reached the optimum. NiaPy came close, but had very slow convergence. Interestingly, for the 3D Hartman problem, none of the implementations found the optimum. Among the 60D problems, the largest performance differences were observed on the Shifted Schwefel problem. The implementations can be divided roughly into two groups: those with faster convergence, and those with slower convergence. The only outlier was pagmo2, which did not belong to either group, and had the worst performance.

In Fig. 5, we can observe the rating intervals for the authors' implementation of GWO, along with implementations in seven different frameworks. None of the implementations have a high degree of overlap with the authors' implementation, there is, however, at least some overlap with three implementations (metaheuristicOpt, NiaPy, and pagmo2). The implementation in EvoloPy outperformed all of the other implementations significantly,

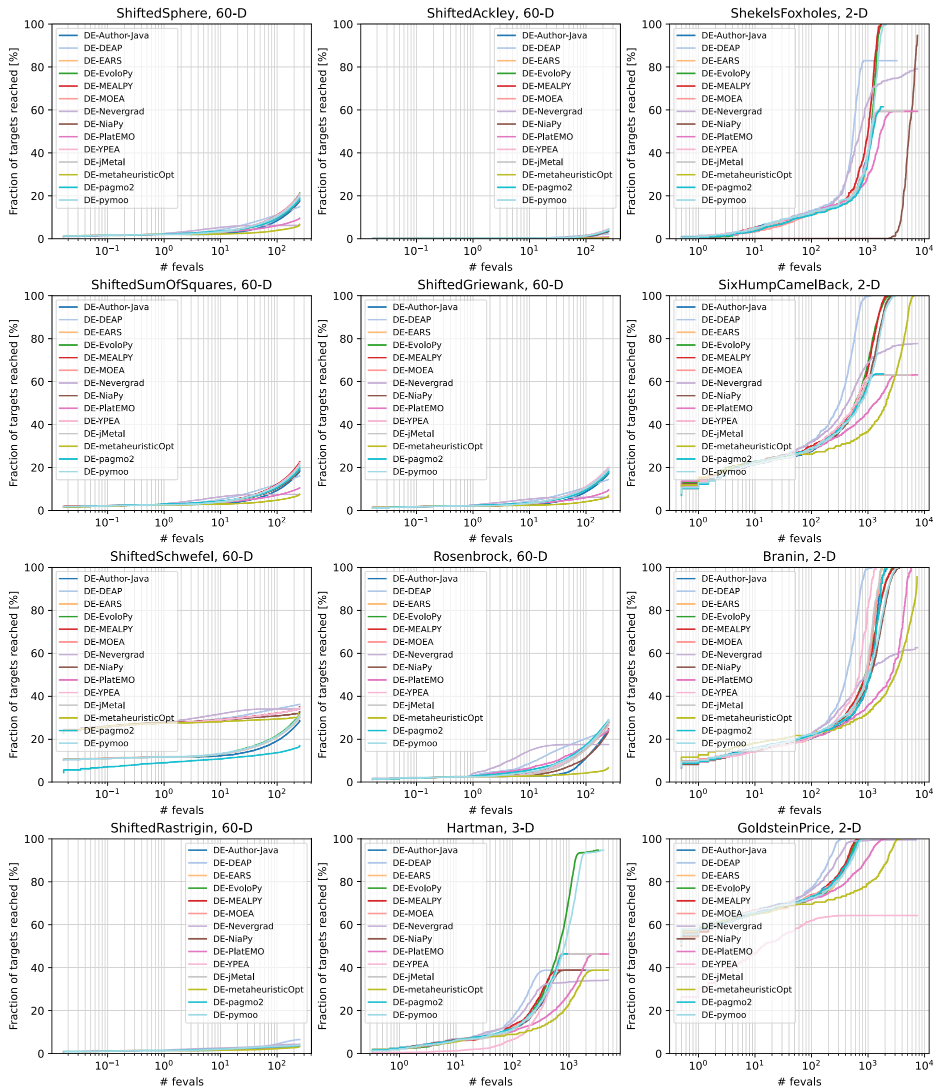

Fig. 4. Profiles per problem for the DE implementations.

indicating the biggest deviation in its implementation. The worst-performing implementation was by MEALPY. It was outperformed by the authors' implementation, as well as four other implementations.

The runtime profiles for GWO are presented in Fig. 6. Among the 60D problems, all the implementations performed best on Rosenbrock, the only non-shifted problem in this group. This supports the claim that GWO has a centre bias (Camacho Villalón *et al.*, 2020; Niu *et al.*, 2019). Interestingly, all the implementations converged to a very similar local optimum at a comparable rate, except for MEALPY, which lagged behind slightly. Similar

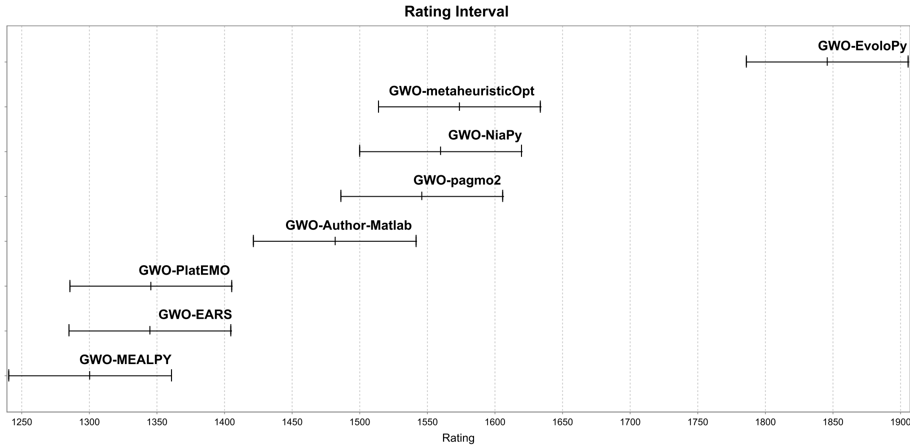

Fig. 5. Rating intervals of GWO implemented in different frameworks, including the authors' implementation.

to DE and ABC, the GWO implementations can be divided into two groups on the Shifted Schwefel problem. The implementations did not perform much better on the 3D Hartman problem. Only PlatEMO reached the global optimum on the Branin problem. The largest differences between implementations were observed on Shekel's Foxholes.

In Fig. 7, we present the rating intervals for GA implementations across six frameworks. Despite the absence of an official reference implementation, the results reveal notable variability, even with consistent genetic operators. The pagmo2 implementation performed best, significantly outperforming pymoo, PlatEMO, and DEAP. Conversely, DEAP performed worst, significantly outperformed by all others. The implementations form three distinct groups with non-overlapping rating intervals between groups: pagmo2, jMetal, and MOEA in the first group; pymoo and PlatEMO in the second; and DEAP alone in the third. These results highlight significant performance differences among implementations, even with identical operators, underscoring the impact of implementation-specific details.

The runtime profiles for GA are presented in Fig. 8. Similar patterns were observed across most 60D problems, except for Rosenbrock and Shifted Schwefel, where the behaviour differed. The implementations performed slightly better on the 3D Hartman problem, but the improvement was minimal. They performed significantly better on the 2D problems, although none reached the global optimum. The largest differences among implementations were observed on Shekel's Foxholes, where pymoo emerged as the clear winner, approaching the global optimum.

In Fig. 9, we examine the rating intervals for PSO implemented in twelve different frameworks. Similar to GA, an official author's source code was unavailable for reference, leading to notable differences in the performance of these implementations. The resulting graph resembles a staircase, with mostly overlapping rating intervals and only a few instances of clear separation. Among the implementations, PSO-Nevergrad performed the least effectively, being outperformed by every implementation, while PSO-EvoloPy

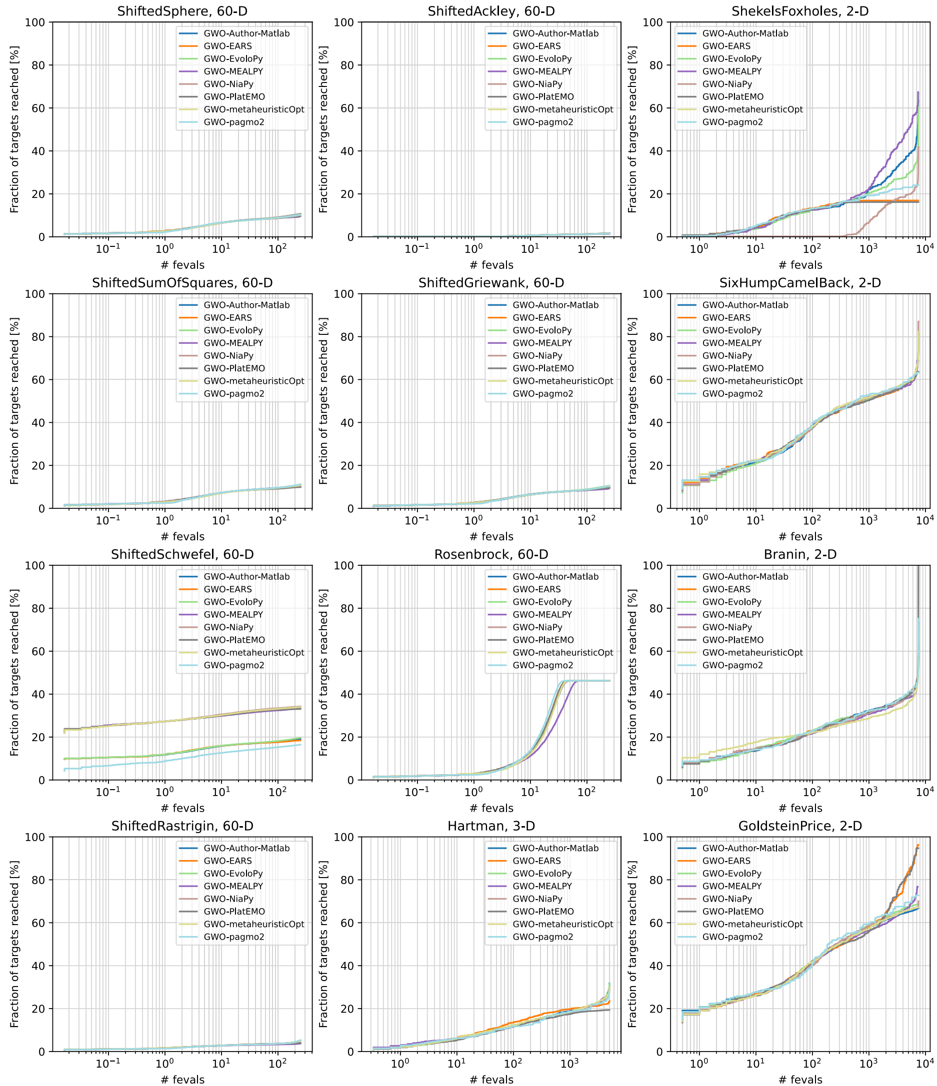

Fig. 6. Profiles per problem for the GWO implementations.

stood out as the top performer, outperforming all the others except for pagmo2. This performance gap raises questions about the potential significant differences between PSO-EvoloPy's and PSO-Nevergrad's implementations. Interestingly, there are two groups of implementations with similar performance. The first group includes PSO-jMetal, PSO-EARS, and PSO-DEAP, while the second group comprises PSO-metaheuristicOpt and PSO-YPEA. These variations in performance emphasize the need for an official source code as a reference point. In summary, the differences in implementation performance underscore the importance of providing a detailed implementation overview for reliable comparisons and evaluations of PSO algorithms in the absence of an official source code.

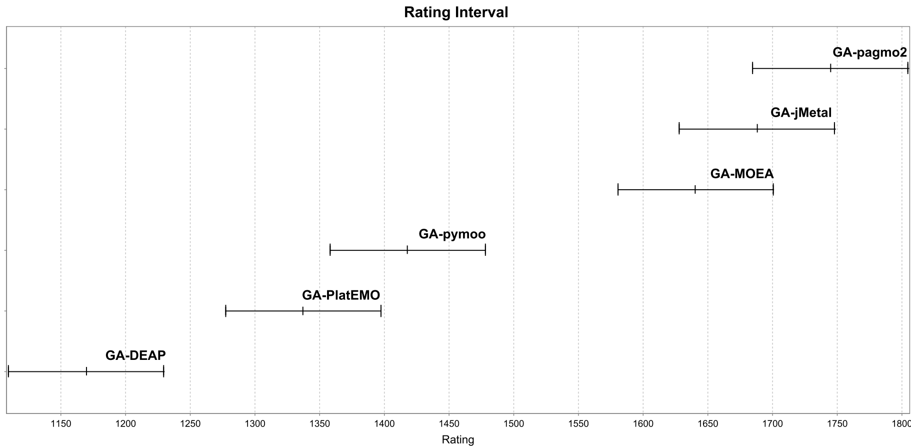

Fig. 7. Rating intervals of GA implemented in different frameworks.

The runtime profiles for PSO are presented in Fig. 10. On the 60D problems, pagmo2 outperformed the others consistently, except on the Shifted Schwefel problem, where DEAP had the best performance. In most of the 60D cases, the implementations in pagmo2, EvoloPy, DEAP, and pymoo stood out. On the 3D Hartman problem, pymoo and EvoloPy were the clear winners, coming close to the global optimum. On the 2D problems, only some implementations reached the global optimum, though many came very close. Based on the proximity to the optimum, Goldstein–Price appeared to be the least challenging problem. Interestingly, similar to GA, the poorest performance on the Goldstein–Price problem was exhibited by the implementation in YPEA.

Figure 11 displays the rating intervals for the authors' implementation of CMA-ES (CMA-ES-Author-Python), along with implementations in four different frameworks. The implementation in pagmo2 performed almost identically to the author's, as indicated by their overlapping rating intervals. DEAP performed slightly worse, but its interval still overlaps a lot with the authors' code. The implementation in pymoo performed the best, outperforming all of the other implementations significantly except the authors'. On the other hand, the implementation in jMetal performed the worst, being significantly outperformed by every other implementation, indicating a greater divergence in its implementation from the rest.

The runtime profiles for CMA-ES are presented in Fig. 12. Among all the algorithms, CMA-ES showed the best performance on the 60D problems, with the jMetal implementation even finding the global optimum on the Shifted Sphere problem. The implementation in pymoo also stood out on the most challenging problem for all implementations (Shifted Ackley), where it came closest to the global optimum. On many 60D problems, the implementations had very similar performance. On the 3D Hartman problem, the pymoo implementation found the global optimum, while the author's implementation came very close. Once again, the highest number of global optima was reached on the 2D problems. The jMetal implementation performed the worst overall, lagging far behind the others.

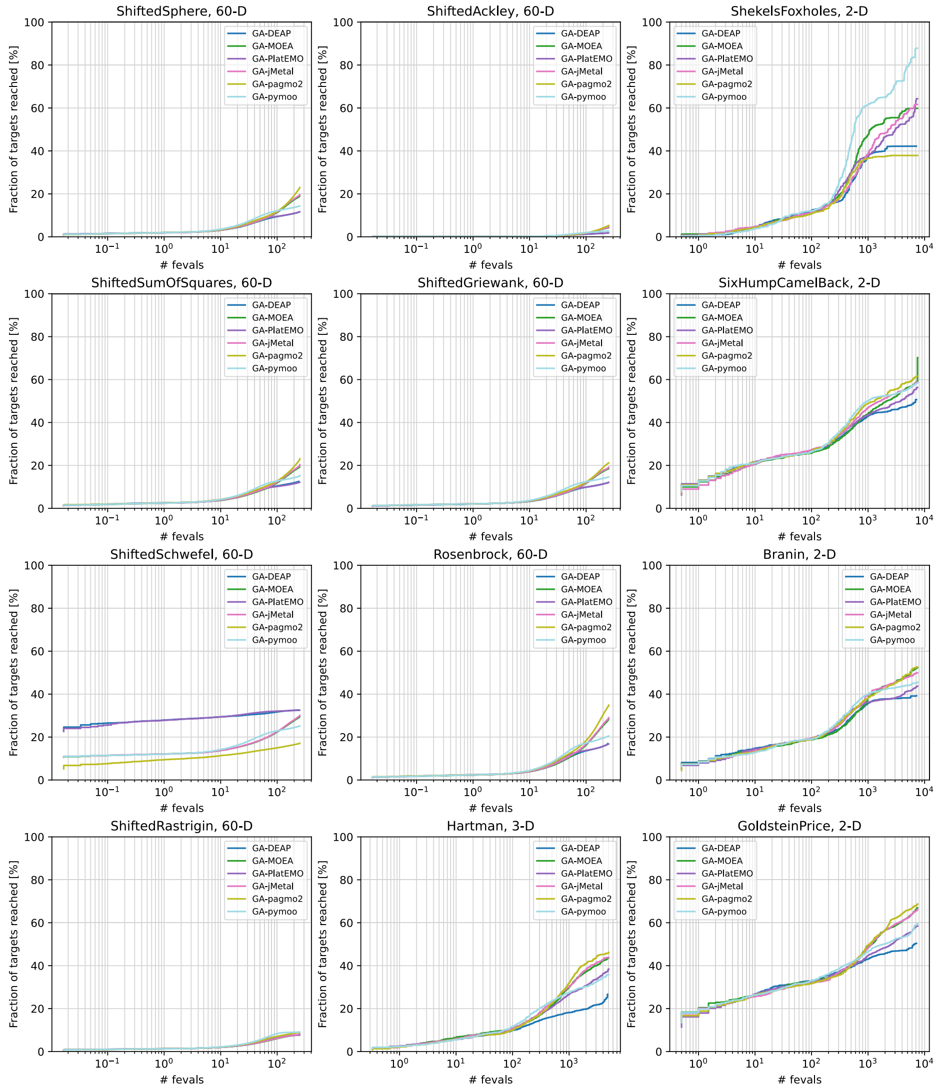

Fig. 8. Profiles per problem for the GA implementations.

The most surprising results were observed on the Shekel's Foxholes problem, where all the implementations performed very poorly.

5.1. *Source Code Analysis*

In this section, we analyse the source code of each algorithm implementation across the evaluated frameworks to identify the causes of observed performance differences. Since all algorithms were tested under the same experimental conditions, as described in Section 4, any variations in results can be attributed to differences in implementation details

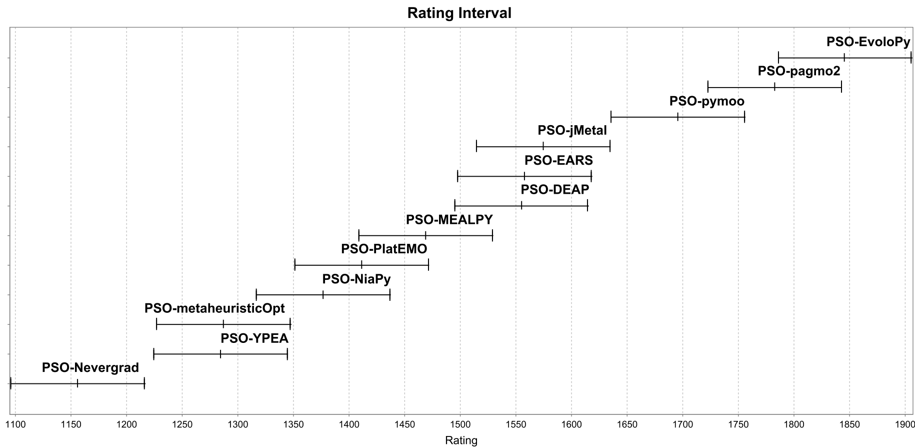

Fig. 9. Rating intervals of PSO implemented in different frameworks.

and programming language choices. A key factor influencing these differences is the Pseudorandom Number Generator (PRNG) and the seed used. Although metaheuristics are generally not highly sensitive to the choice of PRNG, transparency regarding the PRNG employed is essential for reproducibility and fair comparison (Ding and Tan, 2014; Zelinka *et al.*, 2013; Funakoshi *et al.*, 2016; Ma and Vandenbosch, 2012). Most implementations adopt the Mersenne Twister as their PRNG. However, notable exceptions exist: the implementations in jMetal and the author's implementation of DE utilized the JavaRandom class, which employs a Linear Congruential Generator (LCNG) as its PRNG. In contrast, the implementations in pymoo, EvoloPy, MealPy, NiaPy, Nevergrad, and the author's implementation of CMA-ES rely on the random module from the NumPy library, which defaults to the Permuted Congruential Generator (PCG64). Regarding initialization, most implementations use the current time, a true random generator, or entropy as the initial seed. Exceptions include the Matlab-based implementations in PlatEMO, YPEA, and the author's implementations of ABC and GWO, which default to a seed of 0. To ensure a fair comparison, we standardized the seed for all Matlab implementations to be based on the current time.

The results revealed significant differences in algorithm performance across the frameworks, indicating that almost all the algorithms varied at least partially in their implementation. This raises a crucial question: how should these results be interpreted? Typically, comparisons aim to identify which algorithms perform best for specific real-world problems. In our study, however, the focus was on consistency between the implementations. Ideally, the same algorithm should produce identical results across all the frameworks, ensuring that the choice of framework does not influence the experimental outcomes. Furthermore, if an implementation is based on the original authors' source code, it should perform identically to the original. The more the rating intervals and performance profiles overlap, the more similar their performance and implementation are. Only then can we be confident in their correctness and ensure fair comparisons. Even if an algorithm performs better than the authors' original version, deviations in implementation undermine

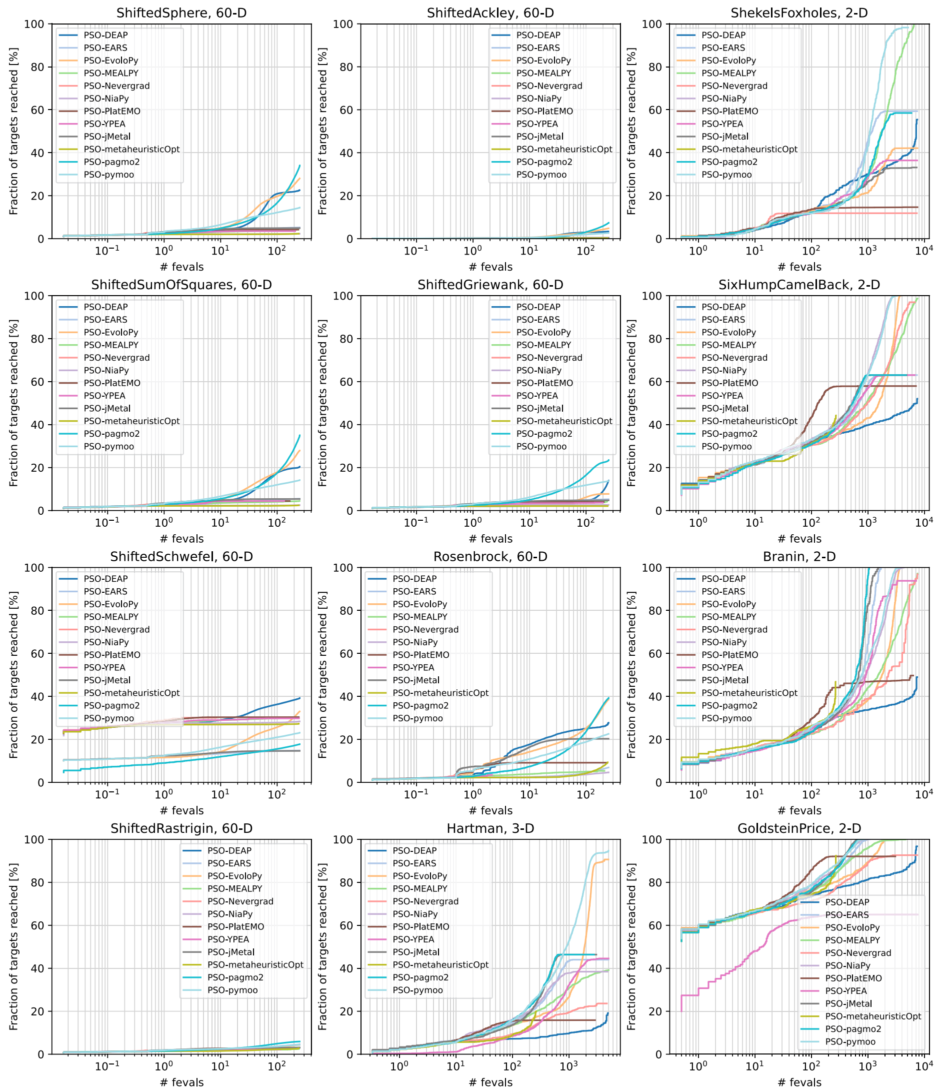

Fig. 10. Profiles per problem for PSO implementations.

the fairness of the comparisons. When the frameworks include implementations that deviate from the authors' intent, any experiments conducted using such frameworks become questionable in their validity and fairness.

To determine the reasons behind the differences observed between the frameworks, we conducted a thorough analysis of the source code for all the implementations. The authors' implementations of the algorithms served as our primary reference point, and key equations were provided for each algorithm. For PSO and GA, where official source codes from the authors were unavailable, we relied on the pseudocodes presented in Kennedy and Eberhart (1995), Holland (1992).

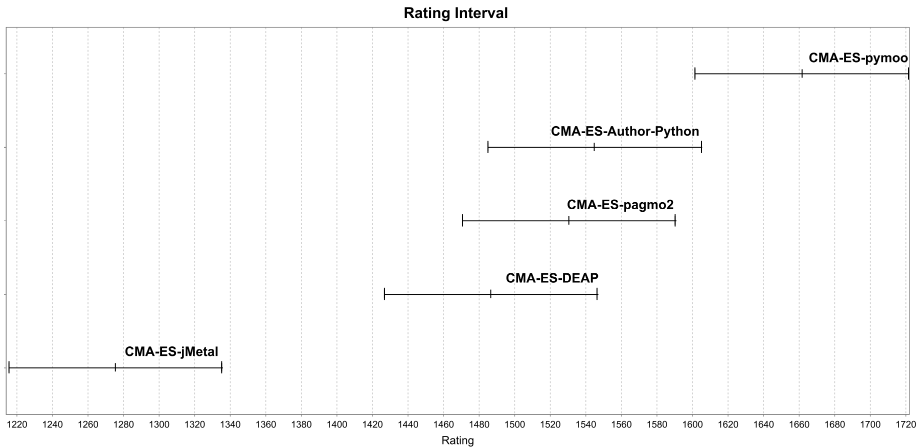

Fig. 11. Rating intervals of CMA-ES implemented in different frameworks.

It is important to acknowledge that variations in performance may arise even when algorithm implementations are functionally identical. These differences can stem from language-specific factors such as numerical libraries, default precision, rounding behaviour, and random number generation. For example, in our earlier study (Ravber *et al.*, 2022a), we observed that evolutionary algorithms implemented in different programming languages, using the same random number generator and seed, began to diverge after several iterations. This indicates that subtle differences in implementation details can significantly affect algorithm behaviour. Furthermore, Goldberg (1991) emphasizes that compliance with the IEEE 754 floating-point standard does not guarantee identical results across platforms or compilers, due to differences in how operations are performed and rounded. Therefore, when evaluating and comparing algorithms, the characteristics of the programming language and its execution environment should be considered to ensure fair and reliable assessments.

In the subsequent sections we present a comprehensive analysis of the source codes for each algorithm, to identify any deviations from the original authors' version, or other implementations that may contribute to the observed differences in performance. We provide a detailed description of the implementations of all the selected algorithms, thereby shedding light on the specific features introduced by each framework. By delving into the details of the implementations and identifying variations across the frameworks, this analysis aims to provide a better understanding of the factors influencing the performance differences between the algorithms. It allows us to identify the impact of each framework's specific implementation choices, and aids in making informed comparisons and evaluations of the algorithms in a fair and comprehensive manner.

5.1.1. *ABC Source Code Analysis*

The Artificial Bee Colony algorithm, introduced by Karaboga and Basturk (2007), is a swarm-based metaheuristic inspired by the foraging behaviour of bees. It is used widely for

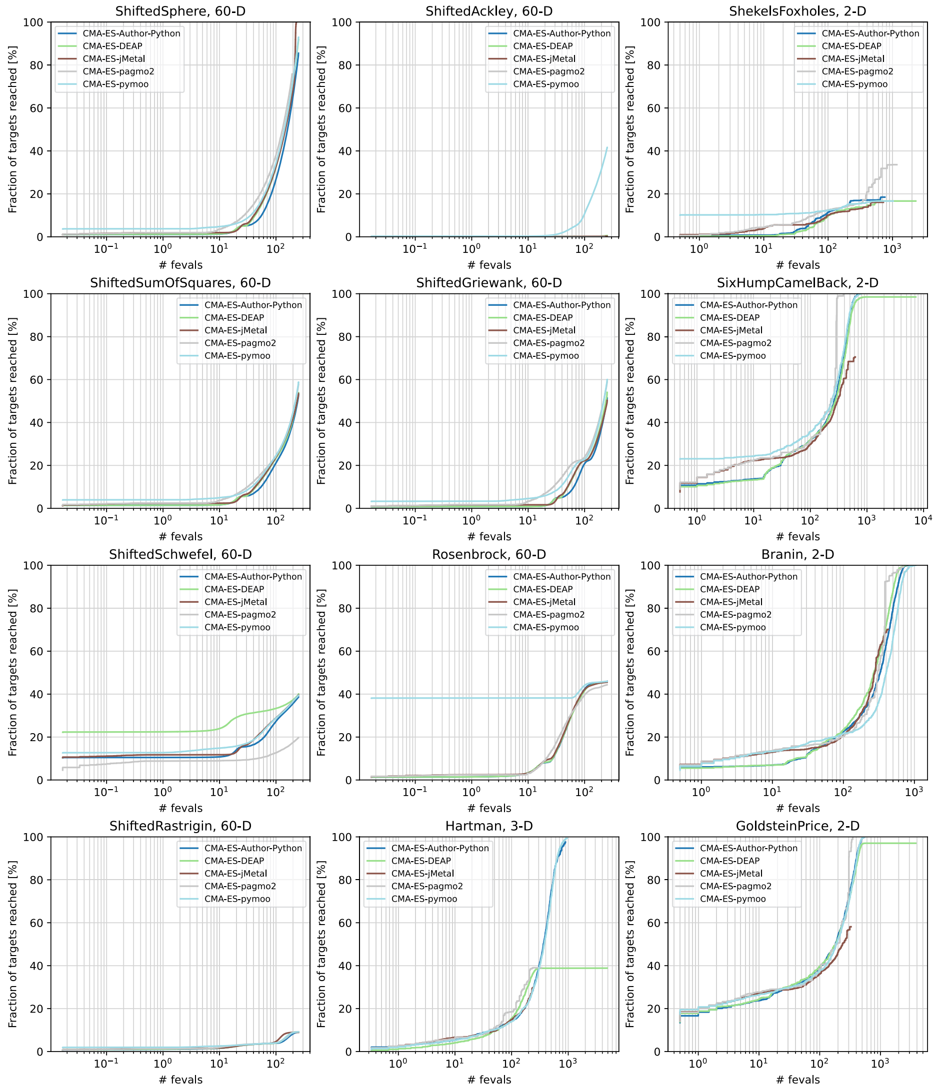

Fig. 12. Profiles per problem for the CMA-ES implementations.

optimization due to its simplicity and effectiveness. The authors' homepage hosts multiple versions of ABC implemented in various programming languages, along with an enhanced variant based on the paper (Mernik *et al.*, 2015). To analyse implementation differences across frameworks, we focused on the core components of the original ABC algorithm, as defined by the authors' source code. These components include the fitness calculation (equation (1)), solution update mechanism (equation (2)), and probability computation for onlooker bees (equation (3)). Below, we present the key equations of the original ABC

algorithm, which served as a reference for our analysis:

$$f_{abc}(x_i) = \begin{cases} \frac{1}{1+f(x_i)}, & \text{if } f(x_i) \geqslant 0, \\ 1 + |f(x_i)|, & \text{if } f(x_i) < 0, \end{cases} \tag{1}$$

$$v_{ij} = x_{ij} + \text{rand}(-1, 1) \cdot (x_{ij} - x_{kj}), \tag{2}$$

$$p_i = 0.9 \cdot \frac{f(x_i)}{\max(F)} + 0.1, \quad F = \{f(x_1), f(x_2), \dots, f(x_{SN})\}, \tag{3}$$

where x_i is a food source (solution), $f(x_i)$ is its objective function value, v_i is a candidate solution, j is a randomly selected dimension, x_k is a randomly chosen neighbour solution ($k \neq i$), and SN is the number of food sources (i.e. the population size).

Our analysis compares ABC implementations across seven frameworks (EARS, NiaPy, pagmo2, MEALPY, PlatEMO, YPEA, and metaheuristicOpt) against the original authors' source code. Notably, the source code differs from the description provided in the original paper. For example, the paper defines the probability as $p_i = \frac{f(x_i)}{\sum_{n=1}^{SN} f(x_n)}$, which risks division by zero if all solutions converge to a fitness of zero. In contrast, equation (3) in the source code introduces a normalization term, adding a constant to avoid this issue. Furthermore, the specialized fitness calculation in equation (1) is unique to the source code, and is not mentioned in the original paper. These discrepancies underscore the importance of using the authors' source code as the baseline for comparisons.

Among the frameworks, only EARS adopts the ABC fitness calculation from equation (1). The implementations in pagmo2 and YPEA do not halve the population size to determine the number of food sources. PlatEMO and YPEA use alternative probability formulas: while the original equation (equation (3)) is linear, their versions are exponential and normalized. In metaheuristicOpt, the formula differs slightly as well, replacing $max(F)$ with $sum(F)$. Additionally, metaheuristicOpt employs greedy selection, where a new solution replaces its parent only if it is better. MEALPY and PlatEMO modify all the dimensions of a solution during the employed and onlooker bee phases, in contrast to the random dimension selection described in equation (2). Interestingly, pagmo2 reverses the order of the onlooker and scout bee phases, altering the algorithm's flow compared to the original design. MEALPY, metaheuristicOpt, and YPEA diverge further, by replacing all the scout bees exceeding the trial limit, whereas the original algorithm replaced only one, as implied by the scout phase logic. Some frameworks also differ in boundary handling. The authors' implementation uses clipping, as does YPEA, although the latter restricts all bounds to [0, 1] regardless of the problem. In metaheuristicOpt, solutions are repaired at the end of the main loop, which may result in the evaluation of infeasible solutions. In contrast, PlatEMO does not apply any boundary control. It is worth noting that we had to modify the source code of YPEA to change the parameter limit. Moreover, metaheuristicOpt only supports a maximum number of iterations as the stopping criterion, which is particularly problematic for ABC, as it does not account for the scout bee phase (Mernik *et al.*, 2015). The framework also includes numerous redundant calls to the fitness function, which can easily go unnoticed if performance is evaluated based solely on the number of iterations. Since our stopping criterion was defined in terms of a maximum

number of function evaluations, we had to account for these additional, unintended evaluations when setting the maximum number of iterations. As a result, the implementation executed significantly fewer iterations, which was also reflected in its overall performance.

These implementation differences impact the solution quality significantly. Our findings emphasize the importance of documenting implementation choices carefully, as seemingly minor variations can lead to substantial performance differences. This analysis also highlights broader concerns applicable to other algorithms, such as the treatment of fitness calculations, stopping criteria, and boundary conditions, which are critical for ensuring consistent and reliable comparisons across the frameworks.

5.1.2. *DE Source Code Analysis*

Differential Evolution, introduced by Storn and Price (1997), is a population-based metaheuristic used widely for continuous optimization, due to its robustness and simplicity. This analysis focuses on the classical DE strategy (rand/1/bin), comparing its implementation across all 13 frameworks against the original authors' source code. The core components of DE include mutation (equation (4)), crossover (equation (5)), and selection (equation (6)), which we summarize in the following equations:

$$y_j = x_{a,j} + F \cdot (x_{b,j} - x_{c,j}), \quad \text{for } j \text{ where } \mathrm{rand}(0,1) < CR \text{ or } j = R, \tag{4}$$

$$y_j = \begin{cases} x_{a,j} + F \cdot (x_{b,j} - x_{c,j}), & \text{if } \mathrm{rand}(0,1) < CR \text{ or } j = R, \\ x_{i,j}, & \text{otherwise,} \end{cases} \tag{5}$$

$$x_i = \begin{cases} y, & \text{if } f(y) \leqslant f(x_i), \\ x_i, & \text{otherwise,} \end{cases} \tag{6}$$

where x_i is the current solution, y is the trial solution, x_a, x_b, x_c are distinct randomly selected solutions ($a \neq b \neq c \neq i$), $F \in [0, 2]$ is the scaling factor, $CR \in [0, 1]$ is the crossover rate, and R is a randomly chosen dimension index.

The original author's source code for DE (rand/1/bin) does not include a boundary control method, allowing solutions to exceed the problem bounds. This omission can influence performance, as boundary handling affects the solution quality significantly (Kadavy *et al.*, 2022). Our analysis shows that most frameworks diverge from the original by incorporating boundary control, often in different ways.

Out of the 13 frameworks, all but one implemented boundary control, most commonly using clipping to keep the solutions within the problem bounds. In contrast, DEAP omits boundary control, aligning most closely with the author's implementation in this regard. In YPEA, clipping is always performed in the fixed range [0, 1], regardless of the actual problem bounds. Notably, pagmo2 and pymoo use random reinitialization for out-of-bounds solutions instead of clipping. This approach encourages exploration, but diverges from standard practices. For EARS and pagmo2, the primary difference from the original DE implementation is the addition of boundary control. In contrast, DEAP's adherence to the original design contributes to its performance similarity with the authors' code. Other frameworks introduce further deviations. Nevergrad is the only framework that does not

support the rand/1/bin strategy. Instead, it uses a variant similar to current-to-best/1/bin, which alters the algorithm's behaviour significantly. In pymoo, an additional polynomial mutation is applied with a 0.1 probability. In EvoloPy, recombination is not guaranteed to occur for a randomly chosen dimension, which deviates from the standard implementation described in equation (5). jMetal sorts the population after selection, which is not part of the original DE procedure and may affect the selection dynamics. NiaPy, metaheuristicOpt, EvoloPy, and pymoo modify the selection process (equation (6)) by replacing the parent solution x_i only if the trial solution y is strictly better ($f(y) < f(x_i)$), rather than equal to, or better ($f(y) \leqslant f(x_i)$). This stricter replacement condition may reduce diversity and slow convergence. DEAP and PlatEMO deviate in how they select agents for mutation (equation (4)). Instead of selecting distinct agents x_a, x_b, x_c randomly, both frameworks apply tournament selection without ensuring distinctness. This may result in biased mutation vectors. Additionally, PlatEMO sets the base vector x_a to the current solution x_i, modifying the strategy from rand/1/bin to a variant closer to current/1/bin, which can reduce the exploration capability. These implementation differences can affect the solution quality significantly. The most impactful deviations are Nevergrad's use of a different strategy and the redundant fitness evaluations found in metaheuristicOpt. These findings highlight the importance of consistent implementation, particularly in boundary handling and operator definitions, which are central to many evolutionary algorithms. Documenting such variations is essential for reliable benchmarking, and for understanding performance differences across metaheuristic frameworks.

5.1.3. *GWO Source Code Analysis*

The Grey Wolf Optimizer, introduced by Mirjalili *et al.* (2014), is a swarm-based metaheuristic inspired by the social hierarchy and hunting behaviour of grey wolves. It is used widely for optimization, due to its balance of exploration and exploitation. This analysis compares GWO implementations in seven frameworks (EvoloPy, metaheuristicOpt, NiaPy, pagmo2, PlatEMO, EARS, and MEALPY) against the original authors' Matlab source code. The core components of GWO include the calculation of distance vectors (equation (7)), position updates (equations (8) and (9)), and coefficient adjustments (equation (10)), which we summarize in the following equations:

$$\vec{D}_\omega = \left|\vec{C} \cdot \vec{x}_\omega(t) - \vec{x}(t)\right|, \quad \omega \in \{\alpha, \beta, \delta\}, \tag{7}$$

$$\vec{x}_1 = \vec{x}_\alpha(t) - \vec{A}_1 \cdot \vec{D}_\alpha, \quad \vec{x}_2 = \vec{x}_\beta(t) - \vec{A}_2 \cdot \vec{D}_\beta, \quad \vec{x}_3 = \vec{x}_\delta(t) - \vec{A}_3 \cdot \vec{D}_\delta, \tag{8}$$

$$\vec{x}(t+1) = \frac{\vec{x}_1 + \vec{x}_2 + \vec{x}_3}{3}, \tag{9}$$

$$\vec{A} = 2 \cdot \vec{a} \cdot \text{rand}(0,1) - a, \quad \vec{C} = 2 \cdot \text{rand}(0,1), \quad a = 2 - t \cdot \frac{2}{T}, \tag{10}$$

where $\vec{x}(t)$ is the position of a search agent at iteration t, $\vec{x}_\alpha$, $\vec{x}_\beta$, $\vec{x}_\delta$ are the positions of the top three solutions (alpha, beta, delta wolves), $\vec{D}_\omega$ is the distance vector, $\vec{A}$ and $\vec{C}$ are coefficient vectors, $\vec{a}$ decreases linearly from 2 to 0 over iterations, and T is the maximum number of iterations.

Among the seven framework implementations, EvoloPy stands out the most, significantly outperforming every other implementation except for pagmo2. Analysis of its source code revealed only one difference from the authors' implementation, yet it appears to have had a substantial impact. The difference lies in how EvoloPy updates the alpha, beta, and delta wolves. The updates are cascading: when the alpha is updated, the previous alpha becomes beta, and the previous beta becomes delta. Similarly, updating beta shifts the previous beta to delta. This results in different beta and delta positions compared to other implementations, which, in turn, affects the subsequent position updates. The implementation in MEALPY uses a greedy selection for new solutions, which, surprisingly, leads to worse performance. In metaheuristicOpt, there are two notable differences. A minor one is that the iterations start from 1 instead of 0. The more significant difference is that the alpha, beta, and delta wolves are updated for each wolf in the population, rather than just once per generation. In PlatEMO, there is a small deviation in how a (equation (10)) is calculated. PlatEMO uses the number of evaluations instead of iterations. No differences were observed in the implementations in NiaPy, pagmo2, or EARS, suggesting that their performance differences can be attributed solely to the programming language used. Language-specific factors, such as numerical precision and floating-point behaviour, can influence the outcomes significantly. These effects were particularly evident in frameworks like EARS and pagmo2, where the implementation was consistent, but the performance lagged behind the authors' implementation. Therefore, the programming language and its associated runtime environment can play a significant role in the performance of metaheuristic algorithms.

5.1.4. *GA Source Code Analysis*

The Genetic Algorithm, introduced by Holland (1992), is a versatile metaheuristic inspired by natural selection, used widely for optimization, due to its adaptability to diverse problem types. GA supports numerous variations in selection, crossover, and mutation operators, but our study adopts a real-valued representation with tournament selection (equation (11)), SBX (equation (12)), and PM (equation (13)) as the baseline. The absence of an official source code from the original author complicates direct comparisons, leading to significant performance variations driven by differences in operator implementations. The core components of the baseline GA are summarized in the following equations:

$$x_{best} = \arg\min\big\{f(x_i) \mid x_i \in T\big\}, T = \{x_{i_1}, \ldots, x_{i_k}\} \sim \text{Uniform}(\mathcal{P}), |T| = k, \quad (11)$$

$$\begin{aligned} c_{1,j} &= 0.5\big[(1+\beta_j)a_j + (1-\beta_j)b_j\big], \\ c_{2,j} &= 0.5\big[(1-\beta_j)a_j + (1+\beta_j)b_j\big], \\ \beta_j &= \begin{cases} (2u_j)^{\frac{1}{\eta+1}}, & \text{if } u_j \leqslant 0.5, \\ \left(\frac{1}{2(1-u_j)}\right)^{\frac{1}{\eta+1}}, & \text{otherwise,} \end{cases} \\ &\text{with probability } p_c, \end{aligned} \quad (12)$$

$$c'_j = c_j + \delta_j(ub - lb),$$
$$\delta_j = \begin{cases} (2r_j)^{\frac{1}{\eta_m+1}} - 1, & \text{if } r_j < 0.5, \\ 1 - [2(1-r_j)]^{\frac{1}{\eta_m+1}}, & \text{otherwise,} \end{cases}$$
$$\text{with probability } p_m, \tag{13}$$

where x_i is an individual, T is the tournament of size k, sampled uniformly from the population $\mathcal{P}$, x_{best} is the best individual in the tournament, a, b are parent solutions, c_1, c_2 are offspring, $u_j, r_j \sim \text{Uniform}(0, 1)$, η is the SBX distribution index, η_m is the PM distribution index, and lb, ub are the lower and upper bounds, respectively.

Our analysis examines GA implementations across six frameworks, using the above equations as the baseline due to the absence of an official reference implementation. Five frameworks lacking support for the selected operators (tournament selection, PM, and SBX) were omitted, as they used different operators. For example, tournament selection is unavailable in EvoloPy, metaheuristicOpt and YPEA, which, instead, use roulette wheel selection. The PM and SBX operators are not available in MEALPY, NiaPy, EvoloPy, and YPEA. Despite using similar operators, the selected frameworks exhibited significant performance variations due to implementation-specific details. For instance, DEAP lacks boundary control, allowing candidate solutions to exceed defined bounds, which can produce infeasible results and reduce solution quality (Kadavy *et al.*, 2022). PlatEMO scales the mutation probability in equation (13) by dividing it by the number of problem dimensions, potentially limiting exploration in high-dimensional problems. In contrast, MOEA, jMetal, and pagmo2 implement elitism, preserving the best individuals between generations. MOEA retains one top solution, while jMetal and pagmo2 keep two, subtly affecting population dynamics and convergence behaviour.

These findings underscore that implementation details, such as boundary control, mutation scaling, and elitism strategies, significantly influence performance, even with consistent operators. The lack of an original GA source code amplifies these variations, highlighting challenges in benchmarking GAs. Consistent operator implementation and boundary handling are critical for reliable comparisons across frameworks.

5.1.5. *PSO Source Code Analysis*

The Particle Swarm Optimization algorithm, introduced by Kennedy and Eberhart (1995), is a swarm-based metaheuristic inspired by the social behaviour of bird flocks, used widely for optimization due to its simplicity and effectiveness. While the original PSO defined a basic velocity update, modern implementations often adopt the modified version by Shi and Eberhart (1998), which introduced an inertia weight w to balance the global and local search. The core components of the modified PSO, used as our baseline, are velocity update (equation (14)) and position update (equation (15)), summarized in the following equations:

$$v_i = w \cdot v_i + c_1 \cdot rand(0,1) \cdot (p_i - x_i) + c_2 \cdot rand(0,1) \cdot (g - x_i), \tag{14}$$
$$x_i(t+1) = x_i(t) + v_i(t+1), \tag{15}$$

where $x_i(t)$ is the position of particle i at iteration t, $v_i(t)$ is its velocity, p_i is the particle's best-known position, g is the global best position, w is the inertia weight, c_1, c_2 are acceleration coefficients, and $rand(0, 1)$ is a uniform random number.

The incorporation of the inertia weight w assumes the crucial role of balancing the global and local search aspects. It can adopt the form of a positive constant, or a positive linear or nonlinear function of time. In the literature, one can discover numerous techniques for calculating the inertia weight w (Shami *et al.*, 2022). Of the 12 frameworks implementing PSO, only DEAP adheres to the original velocity update version. Conversely, the other 11 implementations adopt the enhanced version, with a uniform weight value of 0.7, as detailed in Table 3. It is worth mentioning that MEALPY, YPEA and EvoloPy diverge slightly, by employing a linear time-dependent function for the weight. In the original paper by the author, initial guidelines were absent for setting velocities, applying boundary control methods, and specifying maximum velocity (V_{max}). These aspects were introduced in later papers. Subsequent source code analysis across frameworks revealed notable discrepancies, primarily concerning velocity initialization, boundary control for positions and velocities, and velocity update methods. Let's delve into the practices adopted by each framework:

1. Initial Velocities Generation
   - **NiaPy**, **YPEA** and **EvoloPy**: Set initial velocities to 0.
   - **DEAP**, **Nevergrad** and **metaheuristcOpt**: Generate random velocities between V_{min} and V_{max}.
   - **MEALPY**, **EARS**, **pagmo2** and **jMetal**: Utilize equations that incorporate variable bounds to generate the initial velocities.
   - **PlatEMO**: Initializes the initial velocities to match the particle positions.
   - **pymoo**: The initial velocity can be set to zero or initialized randomly (default). First, the maximum velocity vector is computed $V_{max} = max_velocity_rate \cdot (ub - lb)$, where $max_velocity_rate = 0.2$. Then, each dimension's velocity is sampled as $V = rand(0, 1) \cdot V_{max}$.

2. Boundary Control for Positions and Velocities
   - **DEAP**, **YPEA** and **PlatEMO**: Lack position boundary control methods.
   - **jMetal**, **NiaPy**, **EARS**, **metaheuristcOpt**, **EvoloPy**, **Nevergrad** and **pagmo2**: Apply a common clipping method for position boundary control. In Nevergrad, however, clipping is always performed in the range [0, 1].
   - **MEALPY**: Generates a new random solution if the current solution is infeasible.
   - **pymoo**: If a solution is infeasible, the equations (14) and (15) are reapplied iteratively (up to 20 times). If the position is still infeasible, it is initialized randomly.
   - **MEALPY**, **PlatEMO**, **EARS**, **Nevergrad** and **jMetal**: Do not implement velocity boundary control methods.
   - **jMetal** and **pagmo2**: Reset velocity to 0 if a solution is infeasible.
   - **pagmo2**, **metaheuristcOpt**, **pymoo**, **EvoloPy** and **DEAP**: Apply velocity clipping using V_{max} values.

- **NiaPy**: Employs boundary reflection for velocities with V_{max} set at 1.5.
- **YPEA**: Employs boundary reflection for velocities, but applies it when positions go out of bounds, not when velocities do.

3. Velocity Update Methods

- **DEAP**: Utilizes the same velocity update method as the original paper.
- **MEALPY**, **YPEA** and **EvoloPy**: Adopt the equation (14), with w being a linear function over time.
- **EARS**: Utilizes equation (14) without incorporating the current velocity v_i, and it employs a constant value of 0.7 for the parameter w.
- **jMetal**, **metaheuristcOpt**, **pymoo**, **Nevergrad** and **NiaPy**: Recalculate velocity entirely, without using the current velocity v_i.
- **PlatEMO**: Instead of using the current velocity, employs particle positions in the velocity update equation.
- **jMetal** and **PlatEMO**: Do not include acceleration constants $c1$ and $c2$ in the calculations.
- **jMetal**: Utilizes a modified range $[0, c]$ for random values, with $c = 0.5 + \log(2)$.
- **jMetal**: Implements a different velocity update equation when a particle's personal best solution equals the global best.
- **pagmo2**: Provides six distinct variants for velocity updates, with the default variant set to 5, which incorporates constriction coefficients as described in Clerc and Kennedy (2002).

It is worth noting that pagmo2 incorporates a swarm topology, which likely contributed to its strong performance, ranking second among all implementations. These observations highlight that greater discrepancies can emerge in implementations when no official source code is available. Frameworks diverge in their approaches to initializing velocities, applying boundary control methods, and updating velocities, indicating the significance of clear guidelines and established practices in achieving consistent results across different implementations.

5.1.6. *CMA-ES Source Code Analysis*

The Covariance Matrix Adaptation Evolution Strategy, introduced by Hansen and Ostermeier (2001), is a powerful metaheuristic for continuous optimization, renowned for its ability to adapt the search distribution to the problem landscapes. Its complex algorithmic structure, involving covariance matrix updates and step-size adaptation, distinguishes it from simpler algorithms like PSO or GA. The core components of CMA-ES are sampling (equation (16)), mean update (equation (17)), covariance matrix update (equation (18)), and step size update (equation (19)), summarized in the following equations, which served as our baseline for comparison:

$$x_i = m(t) + \sigma(t) \cdot \mathcal{N}\big(0, C(t)\big), \quad i = 1, \ldots, \lambda, \tag{16}$$

$$m(t+1) = m(t) + c_m \cdot \sum_{i=1}^{\mu} w_i \big(x_{i:\lambda} - m(t)\big), \tag{17}$$

$$C(t+1) = (1-c_c) \cdot C(t) + c_c \cdot \left(\sum_{i=1}^{\mu} w_i \cdot \frac{x_{i:\lambda} - m(t)}{\sigma(t)} \cdot \frac{x_{i:\lambda} - m(t)}{\sigma(t)}^{\top} + c_c^{\delta} \cdot C(t) \right), \tag{18}$$

$$\sigma(t+1) = \sigma(t) \cdot \exp\left(\frac{c_\sigma}{d_\sigma} \cdot \left(\frac{\|\mathcal{N}(0,I)\|_E}{\mathbb{E}[\|\mathcal{N}(0,I)\|]} - 1 \right) \right), \tag{19}$$

where $m(t)$ is the mean of the search distribution at iteration t, $\sigma(t)$ is the step-size, $C(t)$ is the covariance matrix, $x_{i:\lambda}$ are the top μ solutions from λ offspring, w_i are recombination weights, c_m, c_c, c_σ, d_σ, c_c^{δ} are learning rates, and $\mathcal{N}(0, C(t))$ is a multivariate normal distribution.

Our analysis compares CMA-ES implementations in jMetal, DEAP, pagmo2, and pymoo against the authors' implementation. Due to the complexity of CMA-ES, six frameworks (EARS, MEALPY, MOEA, PlatEMO, YPEA, and Nevergard) produced non-functional implementations that yielded unreliable or no results. This underscores a key challenge in deploying complex algorithms without robust and verifiable reference implementations.

It is important to note that, although the author's implementation supports a stopping criterion based on the maximum number of evaluations, it exceeded the specified limit consistently by 51 evaluations. While this number is small, it should not be overlooked. In all implementations, the learning rate c_m (equation (17)) was set to 1 implicitly. Interestingly, the pymoo framework uses the version of CMA-ES published on PyPI by the author Hansen *et al.* (2019), yet it performs slightly better. Since the code and parameters are identical, the differences can be attributed entirely to the framework's handling. When bypassing pymoo's wrapper class and calling the authors' code directly, the results matched those of the authors' implementation. In pymoo, the original algorithm is executed within a loop, with the stopping criterion managed externally by the framework, which ensures termination exactly at the specified number of evaluations. The implementations in pagmo2 and DEAP show more similarity to the authors' version, though some deviations still exist. In pagmo2, a cap is added to the exponent in equation (19), limiting it to a maximum value of 0.6. DEAP modifies the calculation of the cumulation factor c_c (equation (18)), diverging from the original formulation. The jMetal implementation deviates more substantially from the authors' code, which was reflected in its lower performance. jMetal employs boundary clipping, while the authors' code uses a bound transform approach. This transform applies a smooth, piecewise linear and quadratic function to map solutions back into the feasible space, preserving continuity and improving search behaviour near boundaries. Furthermore, the jMetal implementation does not include the coordinate-wise Standard Deviation capping mechanism found in the authors' version, where Standard Deviations are limited to $2/3$ of the search space per dimension using the $maxstd$ parameter. Additionally, jMetal computes the learning rate c_μ for the $rank\mu$ up-

date in the covariance matrix adaptation (equation (18)) using a larger denominator. This results typically in a smaller c_μ, slowing the adaptation of the covariance matrix C.

6. Conclusions

In conclusion, when proposing a new evolutionary algorithm, it is important to compare it thoroughly with existing state-of-the-art algorithms. Replication experiments are necessary to address the limitations of directly comparing results from different studies. Metaheuristic frameworks make these experiments easier by providing pre-implemented algorithms. Our study reviewed the source codes and found differences between the implementations and the original authors' versions. Comparing performance across various frameworks, both with and without the authors' source code, showed significant differences with the same settings. While it is expected that algorithms implemented in different frameworks will exhibit varying performances, the extent of these differences was unexpectedly significant. These findings raise concerns about the reliability of previous studies that used these frameworks for comparisons. Although we compared only a handful of metaheuristics, we can claim confidently that the same findings apply to any other metaheuristic.

To improve the fairness and legitimacy of future experiments, researchers should validate the code they use. This would not only ensure accuracy and consistency, but also benefit the research community by promoting the use of correct implementations and enhancing the overall quality of scientific studies. When authors don't use frameworks and either write the code themselves or get it from unofficial sources, even bigger problems can arise. Our study found significant differences across six evolutionary algorithms in different frameworks, caused solely by the framework implementations. These differences ranged from small to large, showing how even tiny changes can affect algorithm performance, and, to our surprise were not attributed only to the programming language used. We highlight the importance of making sure framework implementations match the original authors' versions as closely as possible. If this is not possible, it is important to disclose any differences clearly and explain why they exist. Metaheuristic frameworks should state clearly which version of the algorithm they implement, and provide proper references to the original or modified formulations. Researchers must provide detailed information on algorithm parameters, operators, and strategies, including any deviations from the original version, and cite the exact source of the implementation if it was obtained from a specific framework, website, or repository. To enhance transparency and usability, frameworks should offer users the option to configure all the available control parameters, ensuring awareness of these settings, as not all users possess the expertise to modify source code, which also may not always be accessible. Furthermore, every framework should support setting the stopping criterion as the maximum number of function evaluations, as iteration-based criteria can be algorithm-dependent, and may fail to account for additional or unintended evaluations, such as those occurring in the ABC algorithm (Ravber *et al.*, 2022b). Using function evaluations as the stopping criterion also facilitates the detection

of such unintended evaluations, thereby ensuring consistent and comparable performance assessments across diverse algorithms.

Despite these limitations, metaheuristic frameworks remain an essential tool for developing and evaluating new optimization algorithms. They reduce implementation effort by offering pre-implemented and debugged metaheuristics, support fair comparisons, and simplify the integration of new methods. Frameworks facilitate reuse, hybridization, and monitoring, often providing tools for visualization, parameter tuning, and even parallel or distributed computing. Their structured design allows researchers to focus on solving the actual optimization problem rather than dealing with implementation details. Moreover, using an existing framework encourages reproducibility and transparency, particularly when the newly proposed method is integrated directly into the same framework. This practice strengthens the contribution to the broader research community. While no framework is flawless, their use should be encouraged, provided researchers remain mindful of their limitations and clearly document any deviations (Molina *et al.*, 2020; Silva *et al.*, 2018; Parejo *et al.*, 2012; Dzalbs, 2021; Osaba *et al.*, 2021).

Promoting Open Science practices, such as sharing source code publicly, ensures that evaluations are robust and transparent. This openness helps the scientific community draw valid conclusions from comparative studies, and fosters a collaborative environment where knowledge and resources are freely accessible. By embracing Open Science, we can enhance the reliability, reproducibility, and overall quality of research in the field of evolutionary computation.

Acknowledgements

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0041 and P2-0114).

References

Alba, E., Ferretti, E., Molina, J.M. (2007). The influence of data implementation in the performance of evolutionary algorithms. In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 764–771.

Bartz-Beielstein, T., Doerr, C., van den Berg, D., Bossek, J., Chandrasekaran, S., Eftimov, T., Fischbach, A., Kerschke, P., La Cava, W., Lopez-Ibanez, M., Malan, K.M., Moore, J.H., Naujoks, B., Orzechowski, P., Volz, V., Wagner, M., Weise, T. (2020). *Benchmarking in optimization: best practice and open issues*. arXiv preprint arXiv:2007.03488.

Biedrzycki, R. (2021). Comparison with state-of-the-art: traps and pitfalls. In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 863–870.

Biscani, F., Izzo, D. (2020). A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53), 2338. https://doi.org/10.21105/joss.02338.

Blank, J., Deb, K. (2020). Pymoo: multi-objective optimization in python. *Ieee Access*, 8, 89497–89509.

Camacho Villalón, C.L., Stützle, T., Dorigo, M. (2020). Grey wolf, firefly and bat algorithms: three widespread algorithms that do not contain any novelty. In: *International Conference on Swarm Intelligence*. Springer, pp. 121–133.

Clerc, M., Kennedy, J. (2002). The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1), 58–73.

Črepinšek, M., Liu, S.-H., Mernik, M. (2014). Replication and comparison of computational experiments in applied evolutionary computing: common pitfalls and guidelines to avoid them. *Applied Soft Computing*, 19, 161–170.
Darwish, A. (2018). Bio-inspired computing: algorithms review, deep analysis, and the scope of applications. *Future Computing and Informatics Journal*, 3(2), 231–246.
Deb, K., Deb, D. (2014). Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1), 1–28.
Deb, K., Agrawal, R.B., (1995). Simulated binary crossover for continuous search space. *Complex Systems*, 9(2), 115–148.
Ding, K., Tan, Y. (2014). Comparison of random number generators in particle swarm optimization algorithm. In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 2664–2671.
Dokeroglu, T., Sevinc, E., Kucukyilmaz, T., Cosar, A. (2019). A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering*, 137, 106040.
Durillo, J.J., Nebro, A.J. (2011). jMetal: a Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10), 760–771.
Dzalbs, I. (2021). *OptPlatform: metaheuristic optimisation framework for solving complex real-world problems*. PhD thesis, Brunel University London.
EARS (2019). EARS – Evolutionary Algorithms Rating System (Github), available at https://github.com/UM-LPM/EARS.
Eftimov, T., Korošec, P. (2019). Identifying practical significance through statistical comparison of metaheuristic stochastic optimization algorithms. *Applied Soft Computing*, 85, 105862.
Eiben, A.E., Jelasity, M. (2002). A critical note on experimental research methodology in EC. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, Vol. 1. IEEE, pp. 582–587.
Ezugwu, A.E., Adeleke, O.J., Akinyelu, A.A., Viriri, S. (2020). A conceptual comparison of several metaheuristic algorithms on continuous optimisation problems. *Neural Computing and Applications*, 32, 6207–6251. https://doi.org/10.1007/s00521-019-04132-w.
Faris, H., Aljarah, I., Mirjalili, S., Castillo, P.A., Guervós, J.J.M. (2016). EvoloPy: an open-source nature-inspired optimization framework in python. *IJCCI (ECTA)*, 1, 171–177.
Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., Gagné, C. (2012). DEAP: evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13, 2171–2175.
Funakoshi, T., Nojima, Y., Ishibuchi, H. (2016). Effects of different implementations of a real random number generator on the search behavior of multiobjective evolutionary algorithms. In: *2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS)*. IEEE, pp. 172–177.
Glickman, M.E. (1999). Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 48(3), 377–394.
Glickman, M.E. (2012). *Example of the Glicko-2 system*. Boston University.
Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing surveys (CSUR)*, 23(1), 5–48.
Hadka, D. (2014). *MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization*. Available at http://moeaframework.org.
Hansen, N., Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2), 159–195.
Hansen, N., Akimoto, Y., Baudis, P. (2019). CMA-ES/pycma on Github. Zenodo. https://doi.org/10.5281/zenodo.2559634.
Hansen, N., Auger, A., Finck, S., Ros, R. (2010). *Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup*. Research Report RR-7215, INRIA. https://hal.inria.fr/inria-00462481.
Hansen, N., Auger, A., Ros, R., Mersmann, O., Tušar, T., Brockhoff, D. (2021). COCO: a platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1), 114–144.
Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press.
Kadavy, T., Viktorin, A., Kazikova, A., Pluhacek, M., Senkerik, R. (2022). Impact of boundary control methods on bound-constrained optimization benchmarking. *IEEE Transactions on Evolutionary Computation*, 26(6), 1271–1280.
Kalami Heris, S. (2019). Yarpiz Evolutionary Algorithms Toolbox (YPEA).

Karaboga, D., Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. *Journal of global optimization*, 39(3), 459–471.

Kennedy, J., Eberhart, R. (1995). Particle swarm optimization. In: *Proceedings of ICNN'95 – International Conference on Neural Networks*, Vol. 4, pp. 1942–1948.

Kovačević, Ž., Ravber, M., Liu, S.-H., Črepinšek, M. (2022). Automatic compiler/interpreter generation from programs for domain-specific languages: code bloat problem and performance improvement. *Journal of Computer Languages*, 70, 101105.

LaTorre, A., Molina, D., Osaba, E., Del Ser, J., Herrera, F. (2020). *Fairness in bio-inspired optimization research: a prescription of methodological guidelines for comparing meta-heuristics*. arXiv preprint arXiv:2004.09969.

Lee, H.M., Jung, D., Sadollah, A., Kim, J.H. (2020). Performance comparison of metaheuristic algorithms using a modified Gaussian fitness landscape generator. *Soft Computing*, 24, 7383–7393. https://doi.org/10.1007/s00500-019-04363-y. https://link.springer.com/article/10.1007/s00500-019-04363-y.

Ma, Z., Vandenbosch, G.A. (2012). Impact of random number generators on the performance of particle swarm optimization in antenna design. In: *2012 6th European conference on antennas and propagation (EUCAP)*. IEEE, pp. 925–929.

Ma, Z., Wu, G., Suganthan, P.N., Song, A., Luo, Q. (2023). Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms. *Swarm and Evolutionary Computation*, 77, 101248.

Merelo, J., Romero, G., Arenas, M.G., Castillo, P.A., Mora, A.M., Laredo, J.L.J. (2011). Implementation matters: programming best practices for evolutionary algorithms. In: *International Work-Conference on Artificial Neural Networks*. Springer, pp. 333–340.

Merelo-Guervós, J.-J., Blancas-Álvarez, I., Castillo, P.A., Romero, G., Rivas, V.M., García-Valdez, M., Hernández-Águila, A., Romáin, M. (2016a). A comparison of implementations of basic evolutionary algorithm operations in different languages. In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 1602–1609.

Merelo-Guervós, J.J., Blancas-Alvarez, I., Castillo, P.A., Romero, G., García-Sánchez, P., Rivas, V.M., García-Valdez, M., Hernández-Águila, A., Román, M. (2016b). Ranking the performance of compiled and interpreted languages in genetic algorithms. In: *Proceedings of the International Conference on Evolutionary Computation Theory and Applications, Porto, Portugal*, Vol. 11, pp. 164–170.

Mernik, M., Liu, S.-H., Karaboga, D., Črepinšek, M. (2015). On clarifying misconceptions when comparing variants of the artificial bee colony algorithm by offering a new implementation. *Information Sciences*, 291, 115–127.

Mirjalili, S., Mirjalili, S.M., Lewis, A. (2014). Grey wolf optimizer. *Advances in Engineering Software*, 69, 46–61.

Molina, D., Poyatos, J., Del Ser, J., García, S., Hussain, A., Herrera, F. (2020). Comprehensive taxonomies of nature-and bio-inspired optimization: inspiration versus algorithmic behavior, critical analysis recommendations. *Cognitive Computation*, 12(5), 897–939.

Morales-Castañeda, B., Pérez-Cisneros, M., Cuevas, E., Zaldívar, D., Toski, M., Rodríguez, A. (2025). Analyzing metaheuristic algorithms performance and the causes of the zero-bias problem: a different perspective in benchmarks. *Evolutionary Intelligence*, 18(2), 1–19.

Niu, P., Niu, S., liu, N., Chang, L., (2019). The defect of the Grey Wolf optimization algorithm and its verification method. *Knowledge-Based Systems*, 171, 37–43.

Osaba, E., Villar-Rodriguez, E., Del Ser, J., Nebro, A.J., Molina, D., LaTorre, A., Suganthan, P.N., Coello, C.A.C., Herrera, F. (2021). A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems. *Swarm and Evolutionary Computation*, 64, 100888.

Oztas, G.Z., Erdem, S. (2021). Framework selection for developing optimization algorithms: assessing preferences by conjoint analysis and best–worst method. *Soft Computing*, 25(5), 3831–3848.

Parejo, J.A., Ruiz-Cortés, A., Lozano, S., Fernandez, P. (2012). Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16, 527–561.

Ramírez, A., Barbudo, R., Romero, J.R. (2023). An experimental comparison of metaheuristic frameworks for multi-objective optimization. *Expert Systems*, 40(4), 12672.

Rapin, J., Teytaud, O. (2018). *Nevergrad – A gradient-free optimization platform*. GitHub.

Ravber, M., Mernik, M., Črepinšek, M. (2016). The impact of quality indicators on the rating of multi-objective evolutionary algorithms. In: *Conference on Bioinspired Optimization Methods and their Applications (BIOMA 2016)*, pp. 119–130.

Ravber, M., Moravec, M., Mernik, M. (2022a). Primerjava evolucijskih algoritmov implementiranih v različnih programskih jezikih. *Electrotechnical Review/Elektrotehniski Vestnik*, 89, (1–2), 46–52.

Ravber, M., Liu, S.-H., Mernik, M., Črepinšek, M. (2022b). Maximum number of generations as a stopping criterion considered harmful. *Applied Soft Computing*, 128, 109478.

Riza, L.S., Iip, Nugroho, E.P., Munir (2018). MetaheuristicOpt: an R Package for optimisation based on metaheuristics algorithms. *Pertanika Journal of Science and Technology*, 26(3), 1401–1411.

Sahin, O., Akay, B. (2016). Comparisons of metaheuristic algorithms and fitness functions on software test data generation. *Applied Soft Computing*, 49, 1202–1214. https://doi.org/10.1016/j.asoc.2016.09.045. https://www.sciencedirect.com/science/article/pii/S156849461630504X.

Shami, T.M., El-Saleh, A.A., Alswaitti, M., Al-Tashi, Q., Summakieh, M.A., Mirjalili, S. (2022). Particle swarm optimization: a comprehensive survey. *IEEE Access*, 10, 10031–10061.

Shi, Y., Eberhart, R. (1998). A modified particle swarm optimizer. In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360)*. IEEE, pp. 69–73.

Silva, M.A.L., de Souza, S.R., Souza, M.J.F., de Franca Filho, M.F. (2018). Hybrid metaheuristics and multi-agent systems for solving optimization problems: a review of frameworks and a comparative analysis. *Applied Soft Computing*, 71, 433–459.

Storn, R., Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11, 341–359.

Tian, Y., Cheng, R., Zhang, X., Jin, Y. (2017). PlatEMO: a MATLAB platform for evolutionary multi-objective optimization. *IEEE Computational Intelligence Magazine*, 12(4), 73–87.

Tzanetos, A., Dounias, G. (2020). A comprehensive survey on the applications of swarm intelligence and bio-inspired evolutionary strategies. In: *Machine Learning Paradigms: Advances in Deep Learning-Based Technological Applications*, pp. 337–378.

Van Thieu, N., Mirjalili, S. (2023). MEALPY: an open-source library for latest meta-heuristic algorithms in Python. *Journal of Systems Architecture*, 139, 102871. https://doi.org/10.1016/j.sysarc.2023.102871.

Veček, N., Črepinšek, M., Mernik, M. (2017). On the influence of the number of algorithms, problems, and independent runs in the comparison of evolutionary algorithms. *Applied Soft Computing*, 54, 23–45.

Veček, N., Mernik, M., Črepinšek, M. (2014). A chess rating system for evolutionary algorithms: a new method for the comparison and ranking of evolutionary algorithms. *Information Sciences*, 277, 656–679.

Velasco, L., Guerrero, H., Hospitaler, A. (2024). A literature review and critical analysis of metaheuristics recently developed. *Archives of Computational Methods in Engineering*, 31(1), 125–146.

Villalobos, I., Ferrer, J., Alba, E. (2018). Measuring the quality of machine learning and optimization frameworks. In: *Advances in Artificial Intelligence: 18th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2018, Granada, Spain, October 23–26, 2018, Proceedings 18*. Springer, pp. 128–139.

Vrbančič, G., Brezočnik, L., Mlakar, U., Fister, D., Fister Jr., I. (2018). NiaPy: Python microframework for building nature-inspired algorithms. *Journal of Open Source Software*, 3(23), 613. https://doi.org/10.21105/joss.00613.

Wang, C.-H., Hu, K., Wu, X., Ou, Y. (2025). Rethinking metaheuristics: unveiling the Myth of "Novelty" in metaheuristic algorithms. *Mathematics*, 13(13), 2158.

Wang, Z., Qin, C., Wan, B., Song, W.W. (2021). A comparative study of common nature-inspired algorithms for continuous function optimization. *Entropy*, 23(7), 874.

Zelinka, I., Chadli, M., Davendra, D., Senkerik, R., Pluhacek, M., Lampinen, J. (2013). Do evolutionary algorithms indeed require random numbers? Extended study. In: *Nostradamus 2013: Prediction, Modeling and Analysis of Complex Systems*. Springer, pp. 61–75.

M. Ravber received his BSc, MSc, and PhD in computer science from the University of Maribor, Maribor, Slovenia, in 2012, 2015, and 2018, respectively. He is currently an assistant professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia. He has worked in the Programming Methodologies Laboratory since 2015. His research interests include evolutionary computation, single- and multi-objective optimization, soft computing, and genetic programming.

M. Šmid received his BSc and MSc degrees in computer science from the University of Maribor, Maribor, Slovenia, in 2020 and 2022, respectively. He is currently pursuing a PhD in computer science and works as a technical assistant at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Since 2022, he has been a member of the Programming Methodologies Laboratory. His research interests include evolutionary computation, genetic algorithms, genetic programming, multi-agent systems, and self-organizing systems.

M. Moravec received his BSc and MSc degrees in computer science from the University of Maribor, Maribor, Slovenia, in 2017 and 2019, respectively. He is currently pursuing a PhD in computer science and works as a teaching assistant at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Since 2019, he has been a member of the Programming Methodologies Laboratory. His research interests include evolutionary computation and single- and multi-objective dynamic optimization.

M. Mernik received the MSc and PhD degrees in computer science from the University of Maribor in 1994 and 1998, respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He was a visiting professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, domain-specific (modelling) languages, grammar and semantic inference, and evolutionary computations. He is the editor-in-chief of the *Journal of Computer Languages*, as well as associate editors of the *Applied Soft Computing Journal*, and *Swarm and Evolutionary Computation Journal*. He has been named a Highly Cited Researcher for years 2017 and 2018. More information about his work is available at https://lpm.feri.um.si/en/members/mernik/.

M. Črepinšek earned his BSc (1999) and PhD (2007) in computer science from the University of Maribor, Slovenia. He currently serves as an associate professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor. His research interests span game development, mobile development, grammar inference, evolutionary computation, single- and multi-objective optimization, as well as computer science education.