Comparison with State-of-the-Art: Traps and Pitfalls

Rafał Biedrzycki
Warsaw University of Technology
Institute of Computer Science
Warsaw, Poland
rbiedrzy@elka.pw.edu.pl

***Abstract*—When a new metaheuristic is proposed, its results are compared with the results of the state-of-the-art methods. The results of that comparison are the outcome of algorithms' implementations, but the origin, names, and versions of the implementations are usually not revealed. Instead, only papers that introduced state-of-the-art are cited. That approach is generally wrong because algorithms are usually described in articles in a way that explains the idea that is hidden behind them but omits the technical details. Therefore, developers have to fill in these details, and they might do so in different ways. The paper shows that even implementations made by one author who is the creator of an algorithm give results which differ considerably from one another. Therefore, for the comparison purpose, the best possible implementation should be identified and used. To illustrate how details that are hidden in the code of implementations influence the quality of the results, sources of quality differences are tracked down for selected implementations. It was found that sources of the differences are hidden in auxiliary code and also stem from implementing a different version of the algorithm which undergoes development. These findings imply best practice recommendations for researchers, implementation developers, and authors of the algorithms.**

***Index Terms*—experiments replication, benchmarking, algorithm-implementation gap, CMA-ES**

I. Introduction

"If I have seen further it is by standing on the shoulders of Giants" – this famous quotation from Sir Isaac Newton implies how important is the validity of previous findings. All of science is based on past theories and experiments. If the past experiments are defective the future findings can be wrong.

In the case of heuristic search, it was noticed [1] that not all contributions are evaluated in a scientifically correct way and reported objectively. The authors proposed a set of guidelines that can help to carry out reproducible experiments and write good reports. They also stress that specifying parameter settings and explaining how they were chosen is essential. The stopping rule must also be documented and justified. One of the pieces of advice is that authors of new algorithms should also describe their implementations (the code).

The current experimental research methodology was also criticized in [2]. The paper pointed out that many authors do not specify the scope of claims regarding their algorithm superiority. It was also stressed that authors of a new algorithm should use performance measures that are fitted to the task being solved, i.e. for some real-world problems the success ratio is a better measure than average quality. The authors also noticed that the level of details specified in a typical paper does not allow to reimplement the same algorithm. To solve this problem it was advocated to create a standardized evolutionary algorithm library that will be freely available for experimenters and will be extended by proposing new operators or algorithms.

In [3] common pitfalls in replication and comparison of computational experiments are discussed. The paper provides guidelines and a checklist that should be filled in whenever algorithms are experimentally compared. The guidelines stress the need for experiments reproducibility. One of the guidelines states that inventors of new algorithms should make their source code publicly available because often algorithm description is too abstract and details have been omitted. Therefore, the experiments cannot be replicated.

All aforementioned guidelines postulate the availability of the implementations but they say nothing about what to do when several implementations are available. Usually, in popular languages, there are many implementations of important or popular methods, e.g., in case of R language [4] there are 5 working implementations of Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [5] and two implementations of Classical Differential Evolution (CDE) [6], [7]. Those implementations were examined in [8]. It was observed that, under the same experimental conditions, there are differences in the results of different implementations of the same algorithm. In case of CMA-ES implementations the difference in success rate measured on functions from *globalOptTest* package [9] is up to 12%. It was also observed that there is no pair of implementations that give similar results. The sources of differences were attributed to the usage of different auxiliary functions, e.g. a method of bound constraints handling, and implementing different versions of the algorithm that is under development. The differences were also observed in the outcome of implementations of CDE, which is an extremely simple algorithm to implement. The source of that differences cannot be tracked in packages' documentation. The analysis of the source code revealed that one implementation of CDE used exponential instead of a binomial crossover. The second difference had its origin in a different approach to handling bound constraints.

In [10] the result of five implementations of CMA-ES were compared on 24 noiseless functions [11], formerly used in 2009 in GECCO Workshop on Real-Parameter Black-Box Optimization Benchmarking (BBOB). The implementations were taken from a trusted source, i.e., from one of its authors' homepage [12]. The implementations cover languages that are popular in the evolutionary optimization community, like Matlab and Python, and other popular languages, i.e., Java and C. The version in C language is important because it is used as an engine in many ports of CMA-ES to other programming languages, e.g. R. The simplified implementation in Python, which was created to facilitate reading the code and understanding it, was also used in the experiments because it is frequently directly translated into other languages, e.g. R [13], and used in practice. The results show that the outcome of examined implementations of CMA-ES differs considerably from one another. The best is Python implementation configured to use quadratic penalty for bound constraint handling instead of default method. The worst is C implementation followed by the Matlab version. Therefore, it is important which implementation is used in the experiments.

The technical report [14] contains a compilation of ideas and recommendations related to benchmarking in optimization. As benchmarking is an evolving field of research the report undergoes periodic updates. The paper raises many important issues, but only those related to this work will be referenced here. The report notices the need to make a distinction between algorithm (like CMA-ES) and algorithm instance (like pycma-es with population size $\mu = 8$). The paper also noticed that researchers should compare their work to the best algorithms, i.e. most current versions and implementations. However, in practice, the selection of the algorithms to be included in a benchmarking depends on the availability of off-the-shelf or easy-to-adjust implementations. Among other problems, two major issues in the current benchmarking landscape have been pointed out: a lack of detail in the description of the algorithms and lack of availability of the implementations of the algorithms.

This paper aims to show that guidelines defined in [1]–[3] have to be extended. The CMA-ES serves here as a model of important optimization method that fulfills the aforementioned guidelines, i.e., it was implemented by the author and made publicly available. When considering evolutionary algorithms, the CMA-ES is in an exceptionally good situation — one of its authors, Hansen, provided [12] high-quality implementations, covering four programming languages.

To illustrate better that performance differences are not negligible and that they do not stem from bugs in source code this paper extends the poster paper [10] by showing the empirical distribution function of runtimes (ECDF) for selected BBOB functions and by tracking down the source of performance differences in the code of implementations.

The paper is composed as follows. In section II the CMA-ES algorithm is briefly described. Section III describes the setup of experiments. In section IV the experimental results are presented and discussed. In section V best practice recommendations are formulated. Section VI concludes the paper.

II. The CMA-ES at a glance

The CMA-ES algorithm is a powerful and popular evolution strategy. It starts work from a given point $\mathbf{m}$. At each iteration, λ points are generated according to the formula: $\mathbf{x_i} = N_i(\mathbf{m}, \sigma^2 \mathrm{C})$, where $i \in \{1, \ldots, \lambda\}$, $\sigma \in \mathbb{R}_+$ is called step-size, and $\mathrm{C} \in \mathbb{R}^{n \times n}$ is the covariance matrix that determines the shape of the distribution ellipsoid. The newly generated points are evaluated by the objective function, and then the values of $\mathbf{m}$, σ, and C are updated to maximize the chance of future successful steps.

In the CMA-ES, unlike in classical evolutionary algorithms, setting up algorithm parameters is considered as a part of algorithm operation. The algorithm uses heuristics to set up its parameters, e.g., λ, or provides reasonable default values and adaptation mechanism, e.g., σ.

Interested readers are referred to [5], [15] or excellent tutorial [16].

III. Experimental setup

The experimental setup is identical to that used in [10]. The experiments were performed using version 2.2.1.10 of the COCO framework [11], using 24 noiseless functions formerly used in 2009 in GECCO Workshop on Real-Parameter Black-Box Optimization Benchmarking (BBOB). Apart from defining objective functions, COCO performs the experiments and creates plots and tables. In the experiments COCO takes a fixed target perspective, i.e. it analyses when the set of target objective function values is achieved. It uses a concept of average runtime (aRT), which is computed over all independent runs as the average number of target function evaluations used to approach global optimum with a given error margin. The statistical significance of results is tested by COCO with the Wilcoxon rank-sum test with Bonferroni correction by the number of functions (24). More details about the assessment procedure in COCO are available in [17].

The CMA-ES implementations were downloaded from Hansen's homepage [12]. The examined implementations cover languages that are popular in the evolutionary optimization community, like Matlab and Python, and other popular languages, i.e. Java and C. The C implementation is used as an engine in many ports of CMA-ES to other programming languages, e.g. R.

Hansen also provides implementations of simplified versions of CMA-ES, which were created to facilitate reading the code and understanding it. These versions are frequently directly translated into other languages, e.g. [13], and used in practice. Therefore, the simplified Python implementation was also included in the experiments. The exact versions of used packages, as were extracted from the source code, are as follows: Java – "0.99.40", C – "3.20.00.beta", Python purecma – "3.0.0", Python – "2.6.0, revision 4423", Matlab – "3.33.integer".

The Python implementation includes two variants of bound constraint handling: the coordinate transformation version [16] and weighted quadratic penalty [18]. Both versions were included in the experiments because C implementation of CMA-ES uses the transformation and the Matlab version uses the penalty approach. The Java version uses resampling [19] and the simplified CMA-ES version comes without constraint handling. To be able to use it the implementation was enriched by a simple additive quadratic penalty [20]. That kind of penalty is frequently used in other CMA-ES implementations, e.g. [21].

COCO is typically used in unconstrained searches, however it also defines functions that return the bounds of the area of interest. These functions are used in order to generate an initial solution and their contribution set bounds for constrained search, which better reflects a real-world application of the optimization algorithm, where bounds usually stem from physics.

For researchers from the benchmarking field, it is obvious that employing different variants of bound constraint handling, results in different instances of an algorithm. Therefore, differences in the results are expected if the search process hits the bounds. For a typical researcher wishing to use the CMA-ES for the comparison or the application, is usually not obvious. Usually, researchers are satisfied when they download and run the code. They feel no need to investigate or to reveal any information about used implementation or bound constraint handling method (BCHM), e.g. in CEC 2018 conference three papers used the CMA-ES but none of them revealed which implementation was used. Including implementations with different BCHMs in this study is in line with the typical approach of most researchers for whom any official implementation is equally satisfying. It also allows for additional verification if BCHMs are important, even in conditions that are not so demanding – i.e., when the optimum is expected to be far from the bounds. After initial comparison of all six versions of the CMA-ES, versions with the same BCHMs will be compared using the statistical test provided by COCO.

For all of the implementations under comparison, the initial step size (σ) was set to $0.3(u-l)$, where u was the upper and l was the lower bound of the parameters' value. This is a default setting in most of the official implementations. The heuristic of setting population sizes μ and λ was identical in all considered implementations so it was left untouched.

According to the recommendation of COCO, if an algorithm stops before exploiting its given budget it is restarted. The first starting point is defined by COCO, the starting points for restarts are generated by COCO's function: *initial_solution_proposal*.

The budget was set to $5*10^7$ objective function evaluations. All methods are stopped when the budget is exhausted. Most implementations directly support stopping criteria based on the budget. The only exception is the C version, for which the maximal number of iterations was set accordingly. All the other methods except simplified Python also implement iterations-based stopping criteria. In Java it is set by default to a large integer, so the stopping criterion based on it is disabled, but in the regular Python implementation, it is set by default to $100+150\cdot(N+3)^2/popSize^{0.5}$ and in Matlab other heuristic is used. Therefore, to perform reliable experiments the maximal number of iterations in Matlab and Python versions was set to a very large integer so it will not interrupt optimization too early.

IV. Results and discussion

Since the aim of this study is not to analyze the behavior of a new method but to show that there are differences between the outcome of different implementations of CMA-ES, only the results of experiments for selected functions for 5 dimensions are reported. The table with the results for all functions and the results of statistical tests are provided in [10].

The plots of the empirical distribution function of runtimes (ECDF) for selected functions are presented in Figure 1. That form of presentation of the results allows for visual assessment of the performance of the implementations as a function of the spent budget.

It can be observed, that there are differences between the outcome of all official implementations. For function 5 the differences are clearly visible when algorithms used about 500 evaluations of the objective function. The best is Python with the weighted quadratic penalty, the worst is Java. For function 16 differences are most clearly visible when algorithms used about 50000 evaluations. Surprisingly, the best was the simplest implementation (Python simple), it even beat the best virtual method from BBOB-2009 when the budget reached about 5000 evaluations. The worst was C, it was also the worst for functions 18 and 23. Even though Java was the worst for 5 and 19 it is the best for 23, it event bet the best virtual method when the budget reached about 40000.

The average runtime (aRT) together with the results of statistical tests for all functions in 5-D at target error level 10^{-5} are shown in Table I. The error level was not set to the smallest value available in COCO to mitigate the influence of floating-point errors, which are different in different languages. Implementations with the same bound constraint handling mechanism were compared in pairs using the statistical test provided by COCO. The star means that differences in the results of algorithms listed in the title of the column are statistically significant, with $p = 0.05$ or $p = 10^{-k}$, when k follows the star.

The results from Table I confirm the observation from Fig. 1 that there are differences between the outcome of all official implementations. Differences between the best and the worst methods usually are large, e.g. for function 5: 33 (Py. sq. pen.) vs 351 (Java); for function 16: 2.5 (Py. sim.) vs 24 (C). Comparison of Python and Python configured to used quadratic penalty (Py. sq. pen) shows that the bound constraint handling method (BCHM) had a noticeable influence on the results. To exclude this factor, implementations with the same BCHMs were compared in pairs using COCO's statistical test. For methods with BCHMs based on penalty (Matlab and appropriately configured Python), statistically significant differences were detected for 7 functions, for 4 of them, the p-value was at the level 10^{-4}. When comparing methods based on transformations of the parameters (Python and C) statistically significant differences were also found for 7 functions.

Observed differences put into question the validity of these published works which used poor implementations. These works are hard to find because nowadays nearly nobody

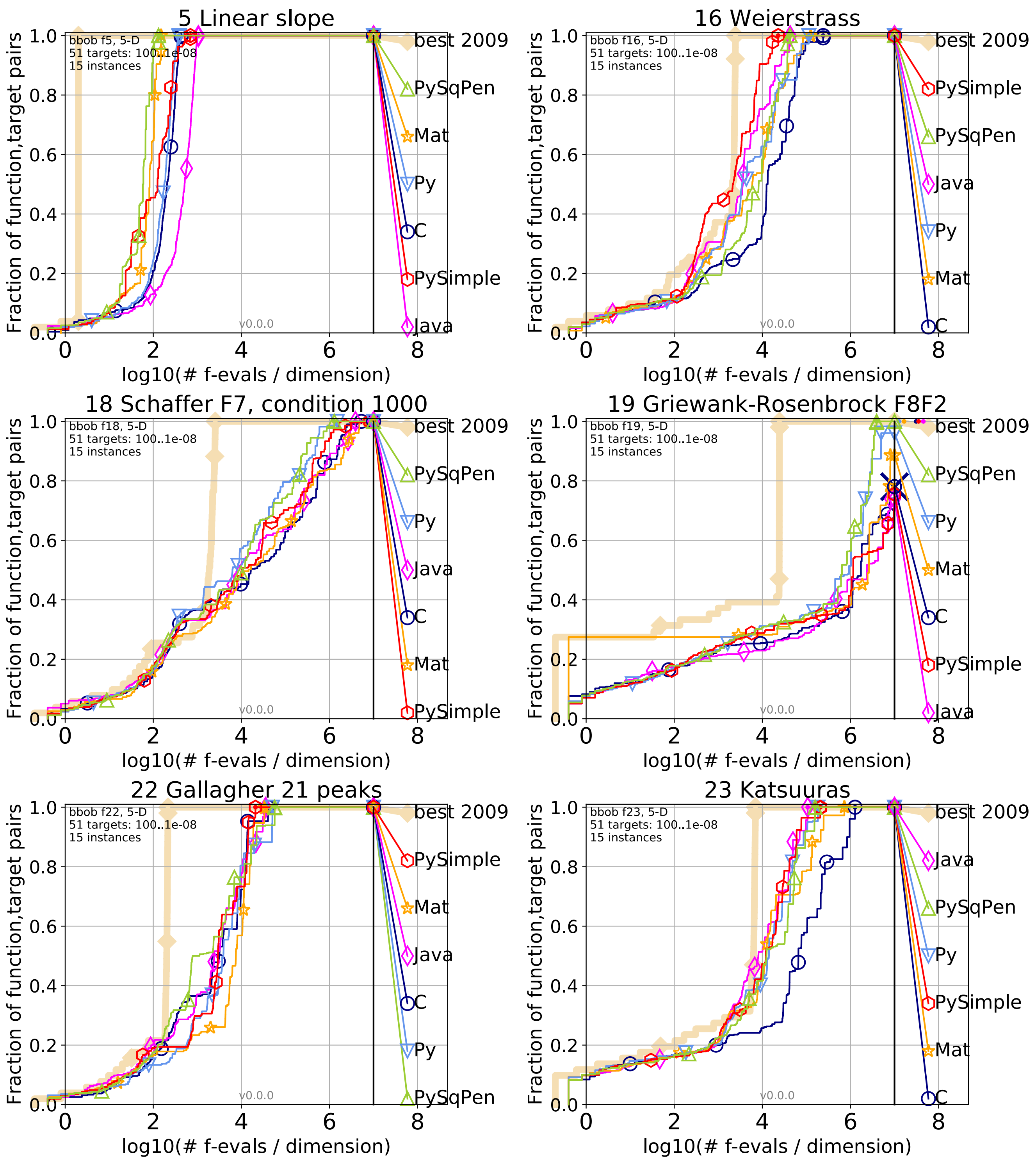

Fig. 1. The empirical distribution function of runtimes (ECDF) of CMA-ES implementations under comparison measured on selected noiseless functions from BBOB 2009, where *Py* is CMA-ES Python implementation, *PySqPen* is the same implementation, but with weighted quadratic penalty used as bound constraint handling, *Mat* stands for Matlab implementation, *PySimple* is the simplified CMA-ES implementation in Python, and *best 2009* is the best virtual optimization method from BBOB 2009.

TABLE I
The average runtime (aRT) divided by the best aRT measured during BBOB-2009 at the target error level 10^{-5} in 5-D as returned by COCO. The best results for each function were typeset in bold. As in COCO, the half difference between 10 and 90%-tile of bootstrapped run lengths was put in braces, as dispersion measure. The last row shows for how many functions the results of each implementation were the best. The star means that differences in the results of algorithms listed in the title of the column are statistically significant, with $p = 0.05$ or $p = 10^{-k}$, when k follows the star.

Fun.	C	Java	Matlab	Python	Py. sim.	Py. sq. pen.	Py. sq. pen. vs Matlab	Python vs C
f1	88 (8)	**78** (13)	84 (6)	86 (10)	89 (7)	81 (8)		
f2	46 (5)	44 (3)	41 (2)	**27** (3)	43 (6)	28 (4)	★4	★4
f3	379 (249)	**283** (246)	341 (395)	601 (254)	472 (702)	328 (315)		
f4	7045 (1e4)	6783 (7417)	9937 (2e4)	6026 (6540)	5732 (2212)	**3908** (2539)		
f5	140 (24)	351 (42)	47 (15)	115 (13)	97 (73)	**33** (9)		★
f6	2.6 (0.3)	2.4 (0.5)	2.4 (0.2)	2.5 (0.2)	**2.3** (0.3)	2.4 (0.1)		
f7	13 (12)	7.2 (10)	9.4 (13)	**2.8** (2)	8.1 (4)	2.9 (5)		
f8	12 (5)	10 (2)	12 (4)	12 (15)	10 (2)	**9.1** (1)	★	
f9	12 (2)	12 (4)	12 (6)	14 (9)	13 (4)	**11** (4)		
f10	8.4 (4)	5.2 (0.4)	4.8 (0.7)	4.3 (2)	4.7 (0.3)	**3.1** (0.2)	★4	★
f11	3.2 (1)	2.9 (0.2)	2.7 (0.2)	2.2 (1)	2.8 (0.3)	**1.5** (0.2)	★4	★
f12	6.8 (5)	5.7 (4)	4.2 (3)	6.4 (5)	6.2 (2)	**3.4** (2)		
f13	3.4 (0.8)	3.6 (0.9)	3.3 (0.6)	**2.1** (0.7)	3.1 (0.4)	2.3 (1)	★2	★3
f14	11 (0.8)	12 (2)	11 (2)	7 (0.8)	11 (1)	**6.4** (0.8)	★4	★3
f15	**20** (29)	**20** (22)	44 (24)	23 (36)	25 (26)	35 (48)		
f16	24 (31)	5.7 (8)	11 (11)	12 (20)	**2.5** (3)	8.2 (7)		
f17	20 (13)	**7.2** (5)	13 (6)	12 (12)	16 (11)	12 (13)		
f18	140 (104)	133 (313)	245 (325)	53 (43)	109 (108)	**40** (66)	★	
f19	358 (448)	389 (306)	290 (144)	117 (34)	382 (272)	**86** (67)		
f20	106 (252)	61 (59)	102 (120)	49 (31)	49 (60)	**35** (13)		
f21	15 (20)	**7.1** (5)	12 (12)	14 (4)	13 (15)	18 (18)		
f22	39 (31)	47 (78)	56 (47)	59 (33)	**35** (28)	43 (136)		
f23	50 (91)	**5** (3)	17 (36)	6.6 (8)	6 (4)	7.7 (12)		★
f24	∞	∞	**60** (60)	62 (44)	∞	∞		
Σ best	1	6	1	3	3	11		

reports the name and the version of the used implementation, e.g. in CEC 2018 conference three papers used the CMA-ES but none of them revealed which implementation was used.

A. Tracing the differences

In the previous section, it was demonstrated that the results of different CMA-ES implementations may substantially differ. To investigate the root cause of those differences the code of two implementations of CMA-ES that use the same bound constraint handling method, i.e. Matlab and Python with the weighted penalty, were carefully compared.

The first observed difference is hidden in CMA-ES stopping criteria, which is especially important for experiments with large budgets and restarts like in COCO. The C implementation checks 8 stopping conditions, but the Python extends this set by additional 3 conditions. Besides that, the default parameters of the conditions are different, e.g. *stopTolFun* and *stopTolFunHist* are 10 times smaller in C; *stopTolX* is 0 in C but 10^{-11} in Python. To verify how important are those differences, the set of stopping conditions in Python implementation was reduced to those available in C. The parameters of the stopping conditions were also taken from C implementation. In this setup, Python implementation calculated results only for functions from 1 to 18, then the program was interrupted due to numerical instabilities. The available results show no statistically significant differences between the default and modified Python setup. Therefore, there are some other, more important differences between C and Python implementations.

After examination of CMA-ES internal parameters, it was noticed that *CMAActive* flag is switched on in Python implementation but it is not available in other implementations. This flag enables concept of *ActiveCMA* [22]. After switching it off and repeating the experiments, the results were compared with the results of C implementation. This time, the statistically significant differences were still detected for functions 2 and 14. Therefore, the concept of *ActiveCMA* improves CMA-ES and is the strong source of differences in performance, but even after switching it off differences exist. We have to bear in mind that optimization algorithms are used by practitioners from other fields and by authors of new methods as a reference. They know that CMA-ES exists but they are not aware of all aspects of its development or implementation. To use the implementation in Python they have to create class *CMAEvolutionStrategy* which requires only two parameters: staring point and initial step size.

Yet another difference is hidden in the heuristics used to detect and escape from flat areas of the fitness function landscape. In C implementation when the objective function value of the best solution is equal to the median of objective function values the step size σ is increased. There is no such simple rule in Python.

When comparing C and Matlab versions it can be noticed that there is a difference in the process of initialization of internal recombination weights. The weights are used to set up $mueff$ parameter, which in turn is used to set up cs, $mucov$, $damps$, etc. The weights are also used in the computation of a new value of the distribution mean and covariance matrix adaptation. In the C version, a vector of weights is initialized with a formula with the $\log(\mu + 1)$ term, but in Matlab version the $\log(\max(\mu, \lambda/2) + 0.5)$ term is used. There are also differences in the initialization of learning rates ($ccov1, ccovmu$).

Many of the aforementioned differences are not in the CMA-ES core algorithm, i.e., usually, when pseudocode is published, the authors do not describe all sanity checks on algorithm parameters that should be implemented in practice. Internal parameter initialization is often unspecified as not being "scientific" enough to publish.

Concerning revealing technical details, the situation of the CMA-ES algorithm is above-standard, because at least two published papers included Matlab implementation of the method. Most other methods do not come with implementations. For them, the situation is much worse because articles describe only the main ideas of the algorithm, and based on that other researchers create implementations and use them for comparison.

B. Article-implementation relation

In the light of the differences that were found in the source code a question arises — which implementation is closer to the "true" CMA-ES algorithm, i.e., which most closely matches the canonical CMA-ES algorithm as known from the literature? To answer that, yet another question should be first answered: which article describes CMA-ES?

In most cases, articles that mention CMA-ES cite [5]. It appears to be correct because in the introduction section of that article the authors wrote "In Section 5, we formulate the CMA-ES algorithm". The paper also contains CMA-ES Matlab code.

The other article that is often cited when referring to CMA-ES is 5 years older [23]. In Hansen's commented bibliography [12] we read about the article: "The first CMA paper, where the covariance matrix adaptation is introduced into the $(1, \lambda)$-ES ($\mu = 1$)", so CMA was used in ES – but does that necessarily mean CMA-ES?

There are many more articles on CMA-ES, so it is not clear which article should be cited as a reference for CMA-ES. To solve this problem it was analyzed how one of the authors of CMA-ES refers to it. It was found that several articles are cited as CMA-ES, e.g., in an application paper [18] a sequence of four articles is cited [5], [23]–[25] when referring to CMA-ES. Therefore, still it is not clear which article describes the canonical CMA-ES version. It seems that nowadays the name "CMA-ES" really means "the family of algorithms".

The article-implementation relationship should be explained in the documentation of the implementations. Unfortunately, there are no references in the C code. In Python, there are two references connected with additional improvements of the method, but there is no reference to the article with a core of the algorithm. In Matlab and Java, there are five references, four of them are common for both implementations, i.e. [5], [24]–[26], and additionally Matlab version refers to [18] and Java refers to [27].

V. Best practice recommendations

Based on the aforementioned findings and the author's experience as a reviewer, author, and reader, this section further extends the guidelines from [1]–[3]:

1) Publishers should require the availability of the source code for all new optimization methods.
2) The code used for running experiments should be available, at least for the reviewers.
3) Authors should use the most canonical implementation of the state-of-the-art and reveal its origin, name, and version.
4) Authors should reveal how all parameters were set up, not only in the proposed method but also in methods used for comparison.
5) Authors of implementations should define article-implementation relation.
6) Authors of articles and authors of implementations should identify the method used for constraint handling.

Availability of the source code for all new methods is required for the reproducibility of the experiments. Without that neither reviewers nor peers can verify the results of the experiments and confirm or reject claims formulated in the article. Availability of the code will also have a positive influence on the number of citations of the article, which is important for both, the author and the publisher. It is more probable for the method to be used in a comparison or an application when the code is easily available. The requirement of the code availability was postulated by previous research [1]–[3] but it is still not implemented. Nowadays the metaheuristics community is flooded by propositions of new algorithms or improvements of the components of existing algorithms. At the time of that abundance, publishers should not be afraid that will be no authors willing to publish because of the requirement of source code publication. It is also no problem from a technical point of view, some optimization competitions, e.g. CEC'2017, have already a place for the codes [28]. Many journals, e.g. Swarm and Evolutionary Computation, provide a place for supplementary materials and research data which can include the code. In the worst case, small conferences without proper infrastructure can use or ask authors to use one of the existing code repositories, e.g. GitHub.

The code used for running experiments should be available, at least for the reviewers. This will allow for verification of the experimental setup. When most scientists are forced by their employers to publish it is easy to imagine that few of them will pick up runs with carefully chosen seeds or will narrow range of parameters during population initialization. Peer review cannot be done honestly without access to the implementation and the code used to run experiments.

The authors should use the most canonical implementation of the state-of-the-art and reveal its origin, name, and version. Proving advantage of a new method over a week implementation of state-of-the-art means nothing. For the verification during the review and for the reproducibility, the origin of the implementation, its name, and version should be provided. If the implementation does not support version numbering the date of the download should be used instead. If the implementation of state-of-the-art was made by the author of a new method its code should also be published. What is more, such self-made implementations should be verified by comparing their results to the results published by an author of the original algorithm.

The authors should reveal how all parameters were set up. Some internal parameters which are auxiliary, e.g. thresholds connected with parameter sanity check and stagnation detection are not published because of the assumption that they are not interesting to the readers, they are not ”scientific” enough. Sometimes such parameters are not revealed because it is hard to justify their adopted values. That is not acceptable because these ”not interesting” parameters are very important. A modified IPOP-CMA-ES [29] may serve as an example here. The modified code gave a much better result than the original one. These improvements may be equally attributed to two changes. One of them consisted of setting up auxiliary parameters, like thresholds used for stagnation detection and sanity checks.

The next frequently occurring problem with parameters is related to their tuning only for the proposed method but not for the state-of-the-art. This is not fair if the implementation under comparison were set up for another benchmark by its authors.

Authors of implementations should define article-implementation relation. Each implementation should refer to the article which described implemented method. If implementation was based on other implementation it should also be revealed. All differences from the published description should be pointed out. If implementation required something not specified in the article it should be documented as well.

Authors of articles and authors of implementations should identify the method used for constraint handling. It was shown in this study and in previous research [8], [10], [19], [30], that methods used for handling box constraints are the source of significant differences in the performance of the algorithms. Usually, methods used for box constraint handling are not identified because most authors focus only on the description of the main ideas of a new optimization method.

Even though a flood of new metaheuristics that claim to be the best is observed nowadays, the set of algorithms that are perceived by the community as state-of-the-art is nearly constant. How so many ”the best” methods may be forgotten or ignored? Even if only some of these methods were really the best we should observe high variability in the state-of-the-art and fast, real progress in the field of metaheuristics. Strict adherence to the guidelines defined here and in previous works should help to further improve the quality of publications and to reduce the number of articles erroneously claiming that their algorithms are the best. These articles act like a noise that hinders readers to find truly valuable ideas.

VI. Conclusions

The choice of a particular implementation of even a popular and standard algorithm may have a substantial impact on the results obtained in research studies or applications. The computational performance and, more importantly, the outcome quality differs substantially across different implementations, even those made by the authors of the algorithm. These differences were demonstrated using the CMA-ES optimization algorithm and its five official implementations. The sources of discrepancies are frequently hidden in the auxiliary code, e.g., in constraints handling code or in stop conditions, especially those designed for detecting numerical instabilities. The difference in the outcome of implementations also stems from implementing different versions of the algorithm that undergoes development.

The existence of the differences has important consequences. Many articles do not provide details about implementations used in experiments (source and version), which puts in question the utility of their findings.

The findings of this contribution imply best practice recommendations for researches – to select the most canonical implementation and reveal its name and version when publishing results, for implementation developers – to make a clear connection between their code and the original article describing the algorithm, and for algorithm's authors – to introduce proper naming conventions for their methods.

References

[1] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart, “Designing and reporting on computational experiments with heuristic methods,” *Journal of Heuristics*, vol. 1, no. 1, pp. 9–32, Sep 1995.

[2] A. E. Eiben and M. Jelasity, “A critical note on experimental research methodology in EC,” in *2002 IEEE Congress on Evolutionary Computation*. IEEE, 2002, pp. 582–587.

[3] M. Črepinšek, S.-H. Liu, and M. Mernik, “Replication and comparison of computational experiments in applied evolutionary computing: Common pitfalls and guidelines to avoid them,” *Applied Soft Computing*, vol. 19, pp. 161 – 170, 2014.

[4] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, Feb. 2021. [Online]. Available: https://www.R-project.org/

[5] N. Hansen and A. Ostermeier, “Completely derandomized self-adaptation in evolution strategies,” *Evolutionary Computation*, vol. 9, no. 2, pp. 159–195, 2001.

[6] R. Storn and K. Price, “Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces,” TR-95-012, ICSI, Tech. Rep., 1995.

[7] ——, “Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces,” *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec 1997.

[8] R. Biedrzycki, "Differences that make a difference: comparing implementations of selected optimization algorithms in R language," in *Proc.SPIE*, vol. 10808, 2018, pp. 10 808 – 10 808 – 12.

[9] K. Mullen, *globalOptTests: Objective functions for benchmarking the performance of global optimization algorithms*, 2014, R package version 1.1.

[10] R. Biedrzycki, "On equivalence of algorithm's implementations: The CMA-ES algorithm and its five implementations," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '19. New York, NY, USA: ACM, 2019, pp. 247–248.

[11] *COCO (COmparing Continuous Optimisers)*, Feb. 2018. [Online]. Available: http://coco.gforge.inria.fr/

[12] N. Hansen. (2018, Oct.) The CMA evolution strategy. [Online]. Available: http://cma.gforge.inria.fr/cmaesintro.html

[13] H. W. Borchers, *adagio: Discrete and Global Optimization Routines*, 2016, r package version 0.6.5.

[14] T. Bartz-Beielstein, C. Doerr, D. van den Berg, J. Bossek, S. Chandrasekaran, T. Eftimov, A. Fischbach, P. Kerschke, W. L. Cava, M. Lopez-Ibanez, K. M. Malan, J. H. Moore, B. Naujoks, P. Orzechowski, V. Volz, M. Wagner, and T. Weise, "Benchmarking in optimization: Best practice and open issues," arXiv:2007.03488[cs.NE], Tech. Rep., 2020.

[15] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, J. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, Eds. Springer, 2006, pp. 75–102.

[16] ——, "The CMA evolution strategy: A tutorial," *CoRR*, vol. abs/1604.00772, 2016.

[17] N. Hansen, A. Auger, D. Brockhoff, D. Tusar, and T. Tusar, "COCO: performance assessment," *CoRR*, vol. abs/1605.03560, pp. 1–16, 2016.

[18] N. Hansen, A. S. P. Niederberger, L. Guzzella, and P. Koumoutsakos, "A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 1, pp. 180–197, Feb 2009.

[19] R. Biedrzycki, "Handling bound constraints in CMA-ES: An experimental study," *Swarm and Evolutionary Computation*, vol. 52, no. 100627, 2020.

[20] Z. Michalewicz and M. Schoenauer, "Evolutionary algorithms for constrained parameter optimization problems," *Evol. Comput.*, vol. 4, no. 1, pp. 1–32, Mar. 1996.

[21] J. Bossek, *cmaesr: Covariance Matrix Adaptation Evolution Strategy*, 2016, R package version 1.0.3.

[22] G. A. Jastrebski and D. V. Arnold, "Improving evolution strategies through active covariance matrix adaptation," in *2006 IEEE International Conference on Evolutionary Computation*, July 2006, pp. 2814–2821.

[23] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE, May 1996, pp. 312–317.

[24] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *Evolutionary Computation*, vol. 11, no. 1, pp. 1–18, March 2003.

[25] N. Hansen and S. Kern, *Evaluating the CMA Evolution Strategy on Multimodal Test Functions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 282–291.

[26] R. Ros and N. Hansen, "A simple modification in CMA-ES achieving linear time and space complexity," in *Parallel Problem Solving from Nature – PPSN X*, G. Rudolph, T. Jansen, N. Beume, S. Lucas, and C. Poloni, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 296–305.

[27] A. Auger and N. Hansen, "A restart CMA evolution strategy with increasing population size," in *2005 IEEE Congress on Evolutionary Computation*, vol. 2. IEEE, Sept 2005, pp. 1769–1776.

[28] *P.N. Suganthan CEC2017 repository*, Feb. 2021. [Online]. Available: https://github.com/P-N-Suganthan/CEC2017-BoundContrained

[29] R. Biedrzycki, "A version of IPOP-CMA-ES algorithm with midpoint for CEC 2017 single objective bound constrained problems," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, June 2017, pp. 1489–1494.

[30] R. Biedrzycki, J. Arabas, and D. Jagodziński, "Bound constraints handling in differential evolution: An experimental study," *Swarm and Evolutionary Computation*, vol. 50, no. 100453, 2019.