



HAL
open science

Benchmarking of Two Implementations of CMA-ES with Diagonal Decoding on the bbob Test Suite

Mohamed Gharafi

► **To cite this version:**

Mohamed Gharafi. Benchmarking of Two Implementations of CMA-ES with Diagonal Decoding on the bbob Test Suite. GECCO 2022 Companion - The Genetic and Evolutionary Computation Conference, Jul 2022, Boston, United States. pp.1700-1707, 10.1145/3520304.3534011 . hal-03671431

HAL Id: hal-03671431

<https://inria.hal.science/hal-03671431>

Submitted on 4 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Benchmarking of Two Implementations of CMA-ES with Diagonal Decoding on the bbob Test Suite

Mohamed Gharafi

Inria, CMAP, Ecole Polytechnique, IP Paris mohamed.gharafi@inria.fr

ABSTRACT

In this paper, we present a comparative benchmark of two implementations of CMA-ES, both with and without diagonal decoding. The benchmarked variants of CMA-ES with diagonal decoding adaptively split the update of the covariance matrix into an update with the original CMA-ES method and an update with the separable-CMA-ES method. Thus, the diagonal decoding should allow for improved performance on separable functions with minimal loss on nonseparable ones. To gain insight into how diagonal decoding impacts CMA-ES runs, an assessment of the performance gain or loss due to the use of diagonal decoding relative to the original CMA-ES, was performed on bbob problems using the COCO platform. We were also interested in variances that might emerge from the difference in the implementations. The data presented in this paper shows improved performance of the CMA-ES on separable functions when using diagonal decoding, without any apparent loss on nonseparable ones. In addition, a few performance variances were spotted in the weakly structured functions, which appeared uncorrelated with the use of diagonal decoding. However, they can be traced back to implementation differences, such as the stopping conditions that may result in different runs, as the data suggests.

CCS CONCEPTS

•Computing methodologies → Continuous space search;

KEYWORDS

Benchmarking, Black-box optimization

ACM Reference format:

Mohamed Gharafi. 2022. Benchmarking of Two Implementations of CMA-ES with Diagonal Decoding on the bbob Test Suite. In *Proceedings of Genetic and Evolutionary Computation Conference Companion, Boston, MA, USA, July 9–13, 2022 (GECCO '22 Companion)*, 8 pages. DOI: 10.1145/3520304.3534011

1 INTRODUCTION

The covariance matrix adaptation evolution strategy CMA-ES [9], is recognized as a state-of-the-art algorithm for continuous black-box optimization. It is a stochastic comparison-based algorithm that maintains a multivariate normal distribution for sampling new candidate solutions. It does so by updating the sampling distribution parameters such as (but not exclusive to) the covariance matrix and the mean vector.

The benchmarked implementations introduce an update of the covariance matrix, with both the original CMA-ES and separable CMA-ES at the same time. The initial assumption is that those

variants should allow for an improvement of the algorithm's performance on separable functions without deteriorating it on non-separable ones.

Using the COCO platform [8], we are interested in assessing the effect of the diagonal decoding acceleration added to the CMA-ES algorithm, on a diverse portfolio of problems in the bbob suite. Moreover, we want to underline any variance in performance that may originate from differences in the implementations.

In this paper we compare 4 variants of the CMA-ES algorithm: the first two are from a new version of `pycma` module [6] that allows for diagonal decoding, referred to as `pycma1` and `pycma0`, respectively with and without diagonal decoding. The second two are from the implementation by Akimoto and Hansen [2] [1] of CMA-ES also with and without diagonal decoding, referred to as `ddcma1` and `ddcma0` respectively.

2 ALGORITHM PRESENTATION

The original CMA-ES algorithm maintains a multivariate normal distribution $\mathcal{N}(m, \sigma^2 DCD)$ for sampling candidate solutions by repeating the steps shown below until at least one of the stopping criteria is met:

- 1) Sample λ candidate solutions $\{s_1, \dots, s_\lambda\} \sim \mathcal{N}(m, \sigma^2 DCD)$.
- 2) Evaluate the sampled candidate solutions on the objective function and sort them $\{f(s_{1:\lambda}) \leq \dots \leq f(s_{\lambda:\lambda})\}$
- 3) Update the mean vector m , the covariance matrix C and the step size σ [9]. However, the diagonal decoding matrix D is not updated instead $D = I$.

Whereas separable-CMA-ES updates D instead of C , which in this case set to I .

The dd-CMA-ES on the other hand, which stands for CMA-ES with diagonal decoding, is a variant of CMA-ES that allows for a speed up of the covariance matrix adaptation by splitting the update into D update and C update with different methods. The C matrix is updated by the original CMA-ES method, and the D matrix is updated by the separable CMA-ES method with a larger learning rate. However, this rate is adapted using a dampening mechanism in order to avoid disturbing the update of C , by keeping the Kullback-Leibler divergence of consecutive sampling distributions as small as possible.

3 IMPLEMENTATION AND EXPERIMENTAL PROCEDURE

The `pycma` and `ddcma` implementations benchmarked in this paper are both written in Python, though they are different codes (*e.g.* different stopping conditions). The diagonal decoding is activated by setting the respective options of each algorithm:

- `CMA_diagonal_decoding` ← 1 for `pycma1`
- `flg_variance.update` ← True for `ddcma1`

Algorithm	2-D	3-D	5-D	10-D	20-D	40-D
pycma0	3.4	2.9	2.6	3.3	3.2	3.4
pycma1	4.0	3.6	3.3	4.2	4.3	5.0
ddcma0	2.3	2.0	1.8	2.2	2.2	2.5
ddcma1	2.7	2.4	2.1	2.5	2.5	2.7

Table 1: CPU time in 1×10^{-4} seconds per function evaluation for varying dimensions

The code used to run benchmark experiments can be found in the repository [4]. The details of the CMA-ES-2019 algorithm used as baseline can be found in [5]. For all algorithms, the bbob problem is evaluated for the origin of the search space at the beginning of each run to avoid unfair advantage given to algorithms that exploit this strategy. The initial solution is given by the bbob function *problem.initial_proposal*, the initial sigma value is set to 2.0 and the population size is the default value for each dimension [9]. The optimization runs are done with IPOPOP restarts (*i.e.* the population size is doubled after each restart), with a maximum number of restarts of 9. The restart’s initial proposed solution is given by calling the function *problem.initial_proposal*, the rest of the parameters are kept at the implementation’s default values. The experiments’ initial conditions presented are the same ones used to generate the performance data of CMA-ES-2019 except the budget, which was set to $5 \times 10^5 \times n$ function evaluations, n being the dimensionality of the problem.

4 CPU TIMING

In order to evaluate the duration of each algorithm, we have reported here the timing results when running the experiments with a budget of 1×10^2 times the dimension with a single batch. Both codes were run on a linux machine with 64 cores, Intel®Xeon®E7 to E3 v4 processors. We are interested in CPU time per function evaluation for varying dimensions. The results can be found in Table 1. The numbers in the table should be taken with a grain of salt, as the experiments are handled by a job manager on a shared machine, which makes it difficult to track the conditions and the specs of the processor on which the experiment was run. Nevertheless, the results consistently show that ddcma is faster than pycma with a factor less than 2. This can be attributed to the fact that the latter is a larger module designed for more diverse use of CMA-ES compared to ddcma, which is a lightweight code designed around diagonal decoding.

5 RESULTS

Results from experiments according to [10] and [7] on the benchmark functions given in [3] are presented in Figures 1, 2, 3, 4 and 5 and Tables 2.

The experiments were performed with COCO [8], version 2.6, the plots were produced with version 2.6.

The **expected runtime (ERT)**, used in the tables, depends on a given target precision, $I_{\text{target}} = f_{\text{opt}} + \Delta f$, and is computed over all relevant trials as the number of function evaluations executed during each trial while the best function value did not reach f_t , summed over all trials and divided by the number of trials that

actually reached f_t [10, 11]. **Statistical significance** is tested with the rank-sum test for a given target Δf_t using, for each trial, either the number of needed function evaluations to reach f_t (inverted and multiplied by -1), or, if the target was not reached, the best Δf -value achieved, measured only up to the smallest number of overall function evaluations for any unsuccessful trial under consideration.

The impact of using diagonal decoding. According to the figures 2 and 3, we can clearly confirm that on **the separable functions group**, at lower budgets the algorithms with the diagonal decoding are better than the others. We can also observe that this effect is more expressed with higher dimensions. By looking further into single functions ECDFs we can see clearly in figure 5 that most of the performance gain comes from **f2**. On the other hand, besides **the weakly structured functions group**, the ECDFs shows no significant difference between the algorithms performance on the other groups.

The impact of the implementations differences. In the case of functions (**f20 - f24**) aka the **weakly structured functions**, we see a variance in the ECDFs that is uncorrelated with the usage of the diagonal decoding. If we consider for example the **Lunacek bi-Rastrigin** function **f24** in dimension 5 in figure 4, we see that both ddcma algorithms (*i.e.* with and without diagonal decoding) outperform the rest of the algorithms including best 2009. We can also see that ddcma0 is better than ddcma1, however the same thing cannot be said about cma algorithms, where cma0 is worse than the rest of the benchmarked algorithms.

The ranking of the algorithms changes again for **f24** in dimension 20, however, it is clear that the ddcma on the **the weakly structured functions group** outperforms the cma on the majority of the dimensions tested. The additional runs of cma with modified stopping conditions (not reported here) suggest that this difference is a result of the difference in restarts run length (*i.e.* number of iterations per restart), caused by the stopping conditions difference between the 2 implementations.

6 CONCLUSION / DISCUSSION

Two implementations of the CMA-ES algorithm with diagonal decoding were benchmarked in this paper, using bbob test suite on COCO platform. The goal was to assess both any gain or loss of performance over the suite’s problems, and highlight any variance that might come from the different codes of CMA-ES besides the usages of diagonal decoding. The data confirms the expected outcome of applying the diagonal decoding acceleration, which is a gain in performance on separable functions with almost no loss on other function groups. Although for weakly structured functions a variance in the ECDFs was observed, and *a priori* unrelated to the usage of diagonal decoding, but more influenced by the stopping conditions used to trigger restarts. Further investigation of single runs is needed to determine how these stopping conditions and the diagonal decoding interacts.

REFERENCES

- [1] Youhei Akimoto. version of 2020/06/10. ddcma.py. <https://gist.github.com/youheiakimoto/1180b67b5a0b1265c204cba991fa8518>. (version of 2020/06/10).
- [2] Y. Akimoto and N. Hansen. 2020. Diagonal Acceleration for Covariance Matrix Adaptation Evolution Strategies. *Evolutionary Computation* 28, 3 (09 2020), 405–435. DOI: <http://dx.doi.org/>

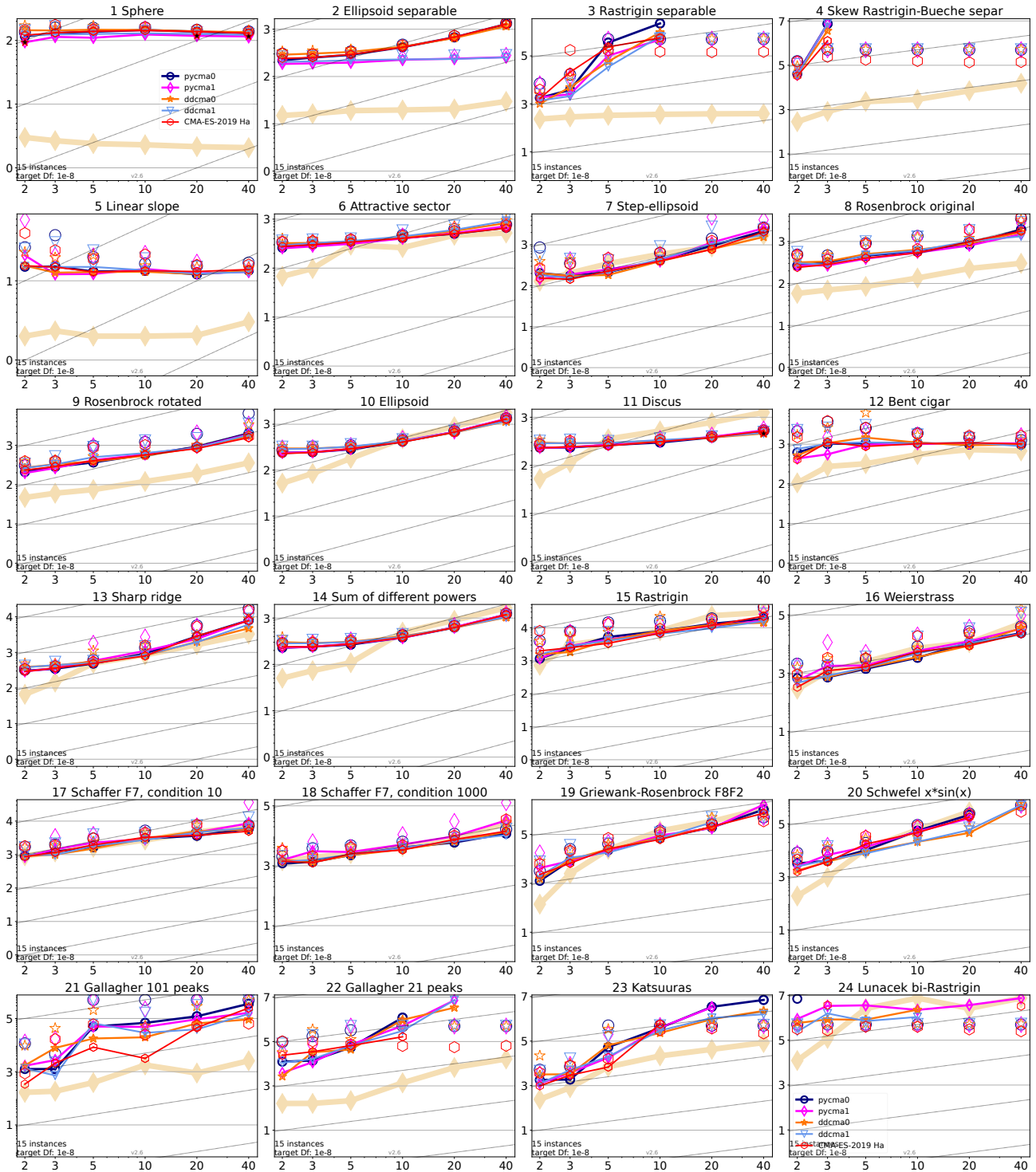


Figure 1: Expected running time (ERT in number of f -evaluations as \log_{10} value), divided by dimension for target function value 10^{-8} versus dimension. Slanted grid lines indicate quadratic scaling with the dimension. Different symbols correspond to different algorithms given in the legend of f_1 and f_{24} . Light symbols give the maximum number of evaluations from the longest trial divided by dimension. Black stars indicate a statistically better result compared to all other algorithms with $p < 0.01$ and Bonferroni correction number of dimensions (six). Legend: \circ : CMA-ES-2019 Hansen, \diamond : ddcma0, \star : ddcma1, ∇ : pycma0, \circ : pycma1.

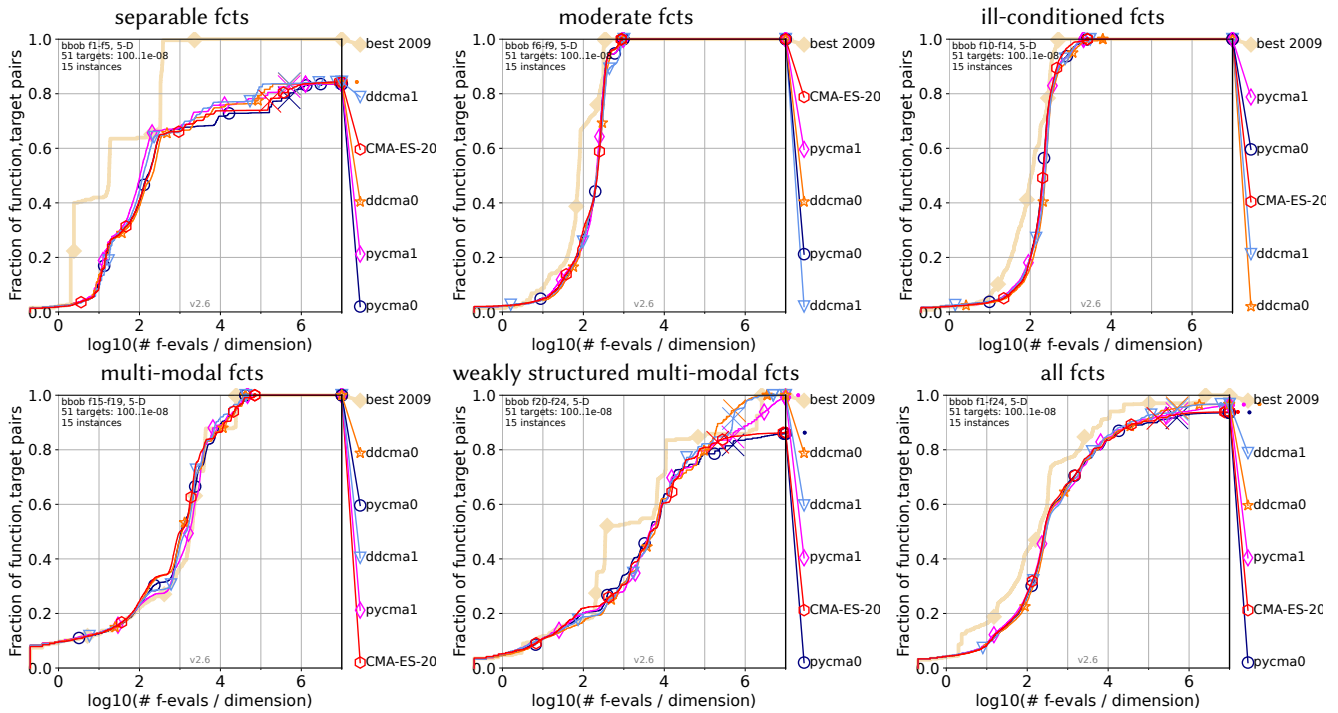


Figure 2: Bootstrapped empirical cumulative distribution of the number of f -evaluations divided by dimension (FEvals/DIM) for 51 targets with target precision in $10^{[-8..2]}$ for all functions and subgroups in 5-D. As reference algorithm, the best algorithm from BBOB 2009 is shown as light thick line with diamond markers.

org/10.1162/evco.a.00260 arXiv:https://direct.mit.edu/evco/article-pdf/28/3/405/1858973/evco.a.00260.pdf

- [3] S. Finck, N. Hansen, R. Ros, and A. Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Presentation of the Noiseless Functions*. Technical Report 2009/20. Research Center PPE. <http://coco.lri.fr/downloads/download15.03/bbobdocfunctions.pdf> Updated February 2010.
- [4] Mohamed Gharafi. 2022. <https://github.com/Mohamed-gharafi/bbob-dd-cma.git> (2022). commit: 240d3da.
- [5] Nikolaus Hansen. 2019. A Global Surrogate Assisted CMA-ES. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 664–672. DOI: <http://dx.doi.org/10.1145/3321707.3321842>
- [6] Nikolaus Hansen. 2022. pycma. <https://github.com/CMA-ES/pycma/tree/development>. (2022). commit: 3bec053.
- [7] N. Hansen, A. Auger, D. Brockhoff, D. Tušar, and T. Tušar. 2016. COCO: Performance Assessment. *ArXiv e-prints* arXiv:1605.03560 (2016).
- [8] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff. 2021. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. *Optimization Methods and Software* 36 (2021), 114–144. Issue 1. DOI: <http://dx.doi.org/https://doi.org/10.1080/10556788.2020.1808977>
- [9] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation* 9, 2 (2001), 159–195.
- [10] N. Hansen, T. Tušar, O. Mersmann, A. Auger, and D. Brockhoff. 2016. COCO: The Experimental Procedure. *ArXiv e-prints* arXiv:1603.08776 (2016).
- [11] Kenneth Price. 1997. Differential evolution vs. the functions of the second ICEO. In *Proceedings of the IEEE International Congress on Evolutionary Computation*. IEEE, Piscataway, NJ, USA, 153–157. DOI: <http://dx.doi.org/10.1109/ICEC.1997.592287>

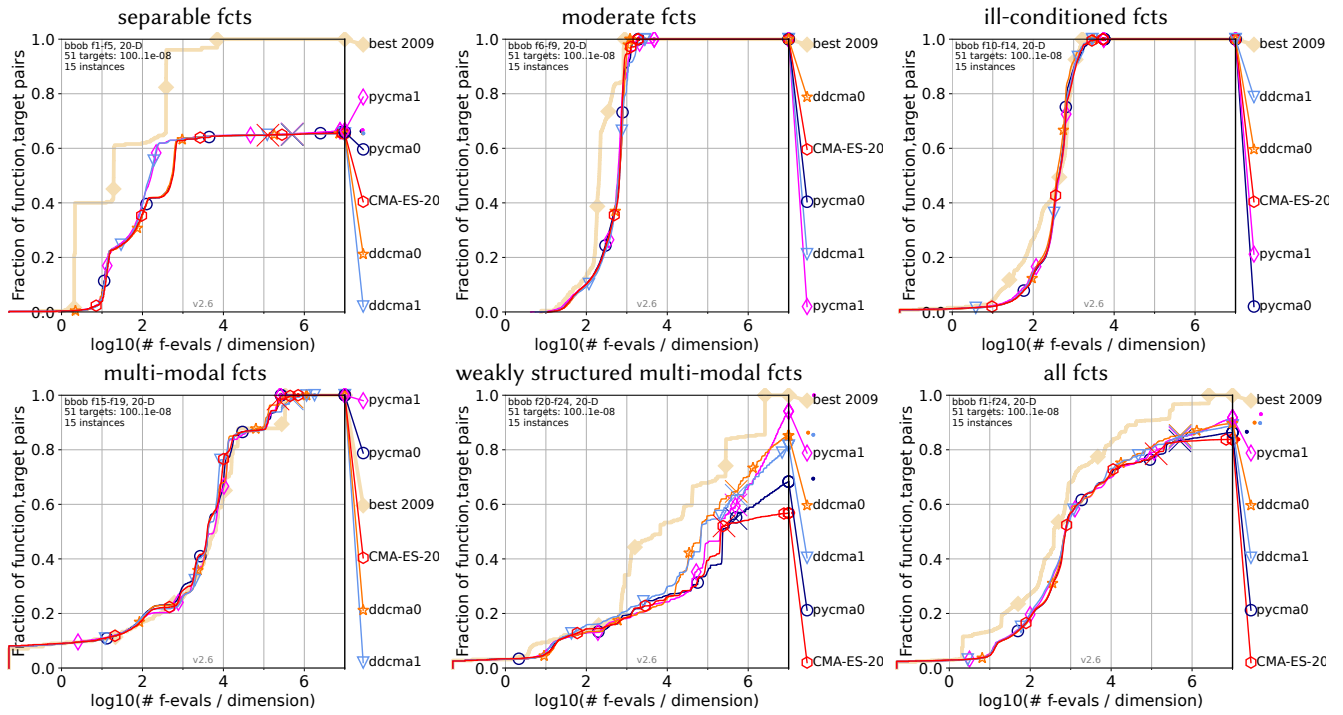


Figure 3: Bootstrapped empirical cumulative distribution of the number of f -evaluations divided by dimension (FEvals/DIM) for 51 targets with target precision in $10^{[-8..2]}$ for all functions and subgroups in 20-D. As reference algorithm, the best algorithm from BBOB 2009 is shown as light thick line with diamond markers.

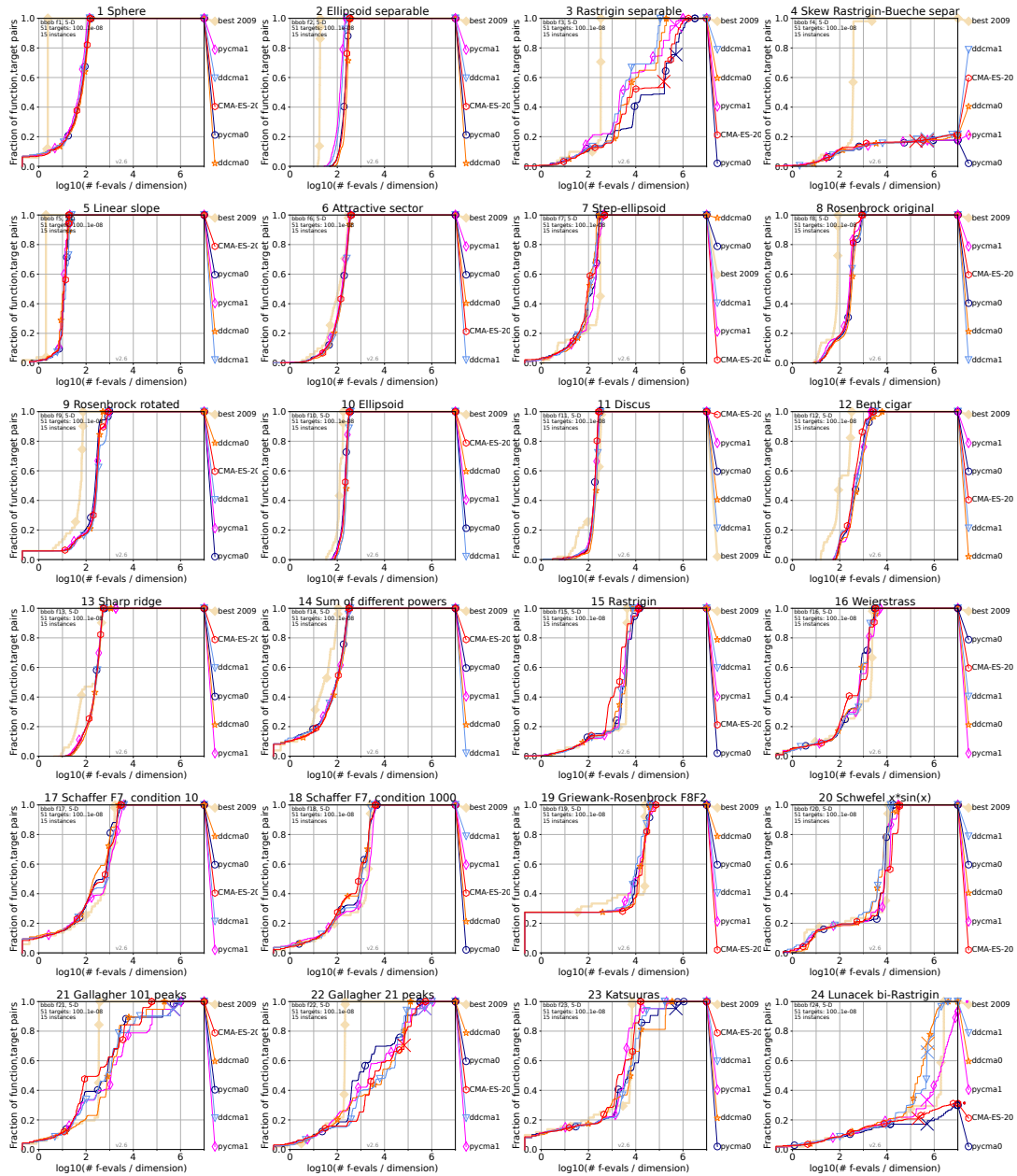


Figure 4: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of f -evaluations, divided by dimension (FEvals/DIM) for the 51 targets $10^{[-8..2]}$ in dimension 5.

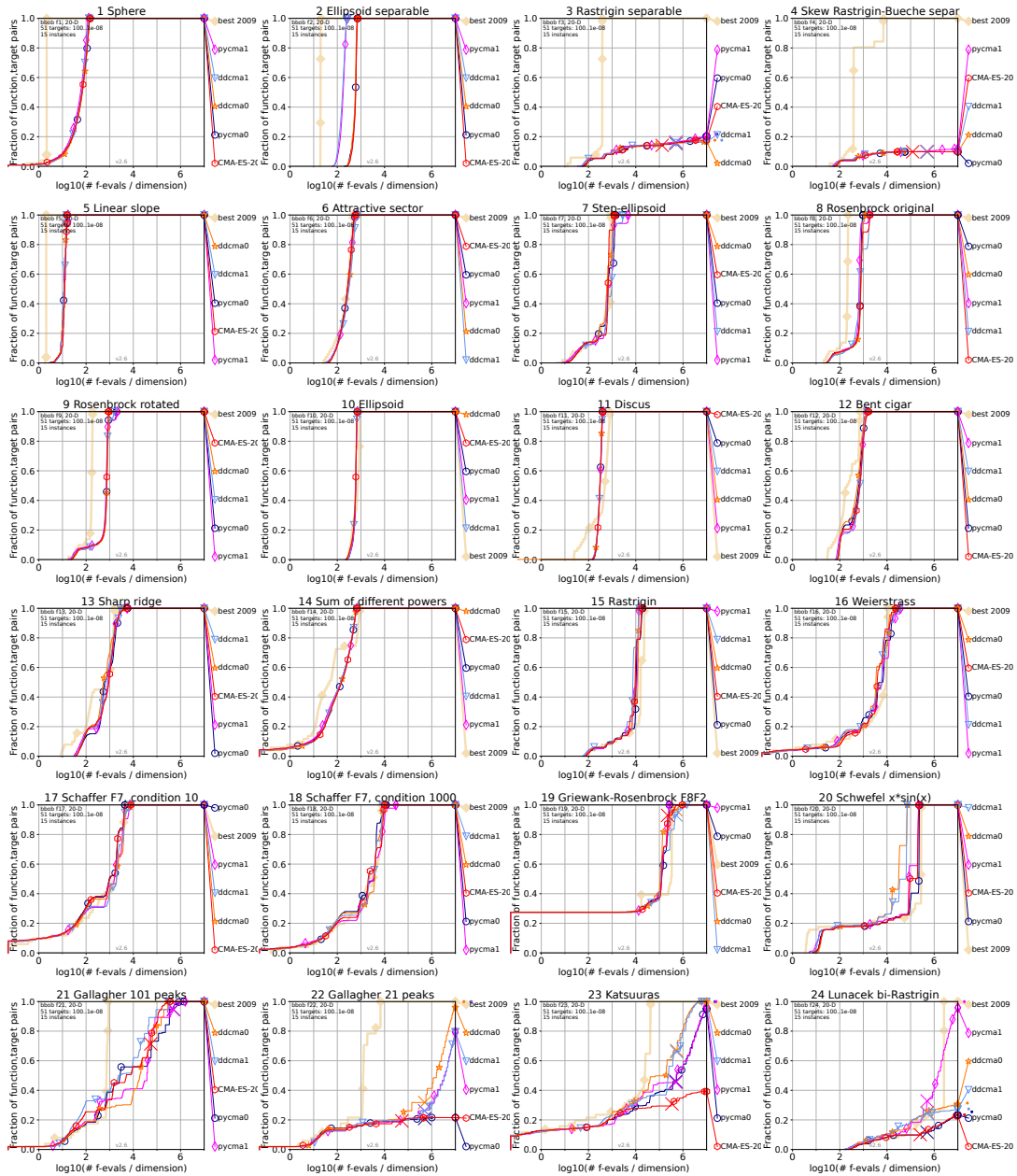


Figure 5: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of f -evaluations, divided by dimension (FEvals/DIM) for the 51 targets $10^{[-8..2]}$ in dimension 20.