Stopping Criteria, Initialization, and Implementations of BFGS and their Effect on the BBOB Test Suite

Aurore Blelly, Matheus Felipe-Gomes, Anne Auger, Dimo Brockhoff

Stopping Criteria, Initialization, and Implementations of BFGS and their Effect on the BBOB Test Suite

Aurore Blelly[1], Matheus Felipe-Gomes[1], Anne Auger[2], Dimo Brockhoff[2,*]

[1] École Polytechnique, Palaiseau, France
[2] Inria, RandOpt team and CMAP, École Polytechnique, Palaiseau, France
firstname.lastname@inria.fr
[*] corresponding author

ABSTRACT

Benchmarking algorithms is a crucial task to understand them and to make recommendations for which algorithms to use in practice. However, one has to keep in mind that we typically compare only algorithm implementations and that care must be taken when making general statements about an algorithm while implementation details and parameter settings might have a strong impact on the performance. In this paper, we investigate those impacts of initialization, internal parameter setting, and algorithm implementation over different languages for the well-known BFGS algorithm. We must conclude that even in the default setting, the BFGS algorithms in Python's `scipy` library and in Matlab's `fminunc` differ widely—with the latter even changing significantly over time.

CCS CONCEPTS

• **Computing methodologies → Continuous space search**;

KEYWORDS

Benchmarking, Black-box optimization, BFGS, Influence of parameters and initialization, Implementation impact

ACM Reference Format:
Aurore Blelly[1], Matheus Felipe-Gomes[1], Anne Auger[2], Dimo Brockhoff[2,*]. 2018. Stopping Criteria, Initialization, and Implementations of BFGS and their Effect on the BBOB Test Suite. In *Proceedings of the Genetic and Evolutionary Computation Conference 2018 (GECCO '18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3205651.3208303

1 INTRODUCTION

Benchmarking algorithms is an important task in optimization. With the right numerical experiments, we can understand algorithms and their general properties and make recommendations about which algorithms to use in certain conditions in real-world situations. However, we must keep in mind that numerical experiments are comparing algorithm *implementations* and specific parameter settings, not a theoretical construct. Hence, it is important to understand the impacts of the algorithms' internal parameters on the algorithm performance. Moreover, it is interesting to compare various algorithm implementations, potentially also over different programming languages.

GECCO '18, July 15–19, 2018, Kyoto, Japan

To assist in benchmarking experiments, the Comparing Continuous Optimizers platform (COCO, [5]) has been developed. But even when almost automated benchmarking platforms such as COCO are used, care must be taken that details of the benchmarking experiments might have important consequences in comparisons. Two aspects come into mind immediately for experiments with test suites from the COCO platform: (i) the choice of the initialization in a generally unbounded search space and (ii) the choice of the function instances. The latter are a way to introduce slight modifications of a function [3] where the introduced differences in difficulty are assumed to be smaller among instances of the same function than between instances of different functions. Changing the instances over time allows to avoid overfitting of algorithms to the experimental setting, but is the mentioned assumption on the similarity of instances actually valid?

In this paper, we address the above questions about the influence of algorithmic and experimental setups on the performance of an algorithm for the representative example of the state-of-the-art BFGS method [10]. The BFGS algorithm has been already compared on the Comparing Continuous Optimizers platform (COCO, [5]) and the data of this experiment, run in MATLAB, has been available with the COCO platform since 2009 [9]. The algorithm is not only the default optimizer in MATLAB's `fminunc` but also the standard algorithm in the `scipy` module of Python (and available in almost any programming language to date).

Our main focus of this paper is thus to compare the Python and MATLAB implementations of BFGS and in particular to see whether the performance of BFGS stayed stable between the MATLAB version of 2009 and the 2017 version. Moreover, we will investigate how basic parameter choices in BFGS can affect its performance on the noiseless `bbob` test suite of COCO [6] and also investigate how the setting of the benchmark suite itself (in form of initialization and instance choice) affects the performance visualization.

In summary, this paper studies the impact of algorithm implementations, algorithm parameter settings and benchmark setup on the conclusions drawn from numerical benchmarking experiments and in particular reminds us that we have to be careful with respect to general statements about the **reproducibility** of benchmarking experiments. Out of the scope of this paper, though, is to understand *where* these differences actually come from. The reason to leave out this investigation is the fact that the BFGS implementations (in particular the ones of MATLAB) are black-boxes themselves for which we can only tune a few parameters but literally do not have access to the source code. We can therefore not really hope to gain a deep understanding about algorithmic differences ourselves.

2 THE BFGS ALGORITHM

Quasi-Newton methods address the problem of unconstrained black-box optimization by the determination of the stationary point of a function using a second-order approximation. The idea is to build an approximation of the Hessian matrix, instead of exactly computing it as in the Newton method and then following, in each step of the algorithm, the estimated Newton direction by a line search.

In order to tackle black-box problems, also the gradient is estimated here in the derivative-free mode by finite differences. In standard implementations, in addition to the current iterate, n solutions (where n denotes the problem dimension) are evaluated along the coordinate axes with a constant Euclidean distance between them and the current iterate. When using symmetric differences, even $2n$ function evaluations are needed in each iteration. With the information of the estimated gradient, the Hessian is approximated, and a line search along the estimated Newton direction is performed to compute the next iterate.

The most used quasi-Newton method is the approach, independently proposed by Broyden, Fletcher, Goldfarb, and Shanno (BFGS, see for example [10])—and also the focus of our study. It is implemented in the basic optimization packages (and often the default) in many programming languages such as MATLAB, Python, or R and thus one of the first choices when solving non-linear (black-box) optimization problems in practice.

We will focus here on three implementations: (i) the MATLAB 2009 version benchmarked by Ros [9], (ii) the MATLAB R2017a version via function `fminunc`, and (iii) the current version of Python's `scipy.optimize` module (version 1.0.1, function `fmin_bfgs`).

3 PERFORMANCE COMPARISONS AND GENERAL SETUP

Results from experiments according to [7] and [3] on the benchmark functions given in [2, 6] will be presented later in Figures 3, 4, 5, and 6. All experiments were performed with COCO [5], version 2.0, the plots were produced with version 2.2.1. [1]

The **average runtime (aRT)**, used in the figures and tables, depends on a given target function value, $f_t = f_{opt} + \Delta f$, 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 [4, 8].

4 SCIENTIFIC QUESTIONS AND EXPERIMENTAL PROCEDURE

We ran several BFGS variants in both MATLAB and Python in order to answer the following scientific questions:

- What are the effects of general benchmarking parameters such as the domain of the initial search point or the concrete bbob problem instances in COCO?
- What is the difference between the MATLAB and Python implementations of BFGS and did the implementations improve over time when compared on the bbob suite of COCO?
- What are the effects of changing/tuning the few internal algorithm parameters that the BFGS implementations provide to the outside?

Algorithm Tuning . We start with the last question in order to keep the investigation of the first two as simple as possible, using the best parameter setting found when answering the last question.

To this end, we tuned the algorithms over two specific functions: f10 (the rotated ellipsoid function) and f11 (the discus function), for dimensions 5 and 20. The choice was made on these functions because the algorithms presented a strange behavior in preliminary experiments on particularly these two functions when the default parameters were used: both the 2017 MATLAB and the Python version slowed down in performance significantly for no reason as if the algorithm decided to restart by itself.

For python's `fmin_bfgs`, we varied the parameter *epsilon* which characterizes the precision to which the derivative of the problem function will be approximated; for MATLAB's `fminunc` we changed the parameter *FiniteDifferenceStepSize* which is a scalar or vector step size factor used to create the finite differences in the gradient approximation. We only tuned it as a scalar, since we do not have a preferential direction. For both parameters, the default value is 10^{-8} and we decided to test values in $\{10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}\}$, given the experimentally found performance with the peak performance in the middle of this interval.

Figures 1 and 2 show the results in which the MATLAB versions of BFGS are denoted by M-1e-XX and the Python versions as P-1e-XX indicating the chosen parameter with their names. What we can see for both programming languages and in all shown settings, the default version falls behind the variants with smaller parameter values for *epsilon* and *FiniteDifferenceStepSize* respectively. Decreasing the parameters below the value of 10^{-10} almost always results in worse behavior—indicating that on the ellipsoid and discus functions, the value of 10^{-10} shall be chosen over the default here. All subsequent experiments described will use this setup. We can note already here that the effect of the two parameters is not exactly the same in both programming languages with the Python version being more sensitive but resulting in better performance than the MATLAB version.

Note further that changing the setting of the algorithms by augmenting the precision of the gradient estimation increased the CPU timing for the experiments by a factor of about 10.

Now that we have taken care of tuning the parameter related to the gradient estimation[2], we can continue with the more generic scientific questions.

Main Experiments and Algorithms Compared. We continue to answer the other two scientific questions above, by running Python and MATLAB versions of BFGS on the bbob test suite of COCO. We denote the default variants, running with the above tuned parameters on the BBOB-2009 instances (three times each instance from 1 to 5), and initialized uniformly at random within $[-5, 5]^n$ as P-2009 for Python's `scipy` version, as M-2009 for the MATLAB 2009 version of Ros (downloaded from the COCO data archive,

[1] Actually, the source code of the development branch of COCO was used to postprocess the algorithm data for this paper, resulting in the slightly increased version number of 2.2.9 in the plots.

[2] which is one of only few externally accessible parameters of the BFGS Python implementation and according to preliminary experiments the most sensitive one besides the initialization as we will see below

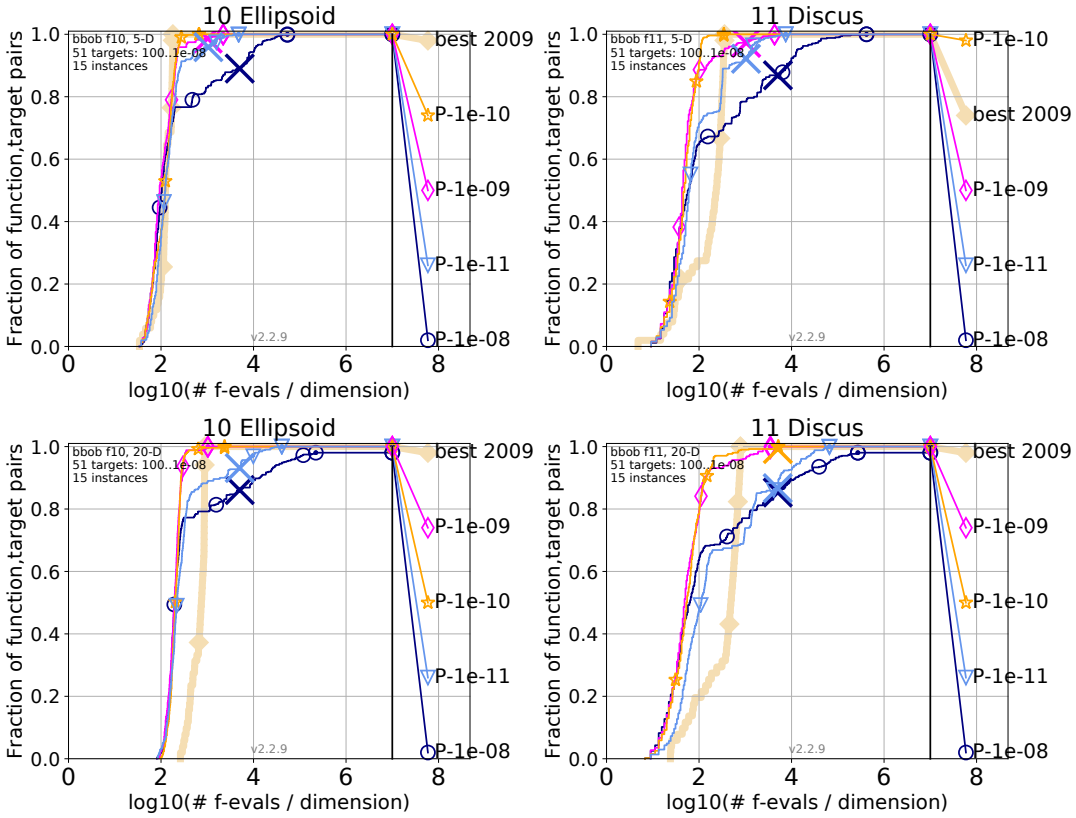

Figure 1: Expected running time by dimension for dimension 5 (upper figures) and 20 (lower figures) for different values of the step *epsilon* used to approximate the function derivative in Python's BFGS (values 10^{-8} (default), 10^{-9}, 10^{-10}, and 10^{-11}).

and therefore not tuned by us, [9]), and as M-2017 for the R2017a MATLAB version.

To see the effects of initialization and the choice of the actual function instances in COCO, we distinguish three additional variants of P-2009:

- *P-range*: initialization uniformly at random in $[-4, 4]^n$ instead of in $[-5, 5]^n$
- *P-StPt*: the starting point is fixed at 0^D (the middle of the bounding box) instead of chosen randomly.
- *P-instances*: the instances used are the one from 2017 (i.e. 1–5 and 61–70)

All other parameters are set as in P-2009 which corresponds to the setting of Ros [9]. In addition, we add, denoted as P-ScipyB here, the data set from Baudiš [1] of the same BFGS algorithm in `scipy` but which was run with a slight modification: each time the algorithm restarts, the restart is done at a point chosen randomly in the neighborhood to where the algorithm was stopped, instead of being independently chosen at random in $[-5, 5]^n$. This strategy is known under the name "Basin hopping" and is available in the `scipy` module as well.

5 CPU TIMING

In order to evaluate the CPU timing of the algorithms, we have run the BFGS algorithm on the bbob test suite [6] with restarts for a maximum budget equal to $2 \cdot 10^5 n$ function evaluations according to [7]. The Python and MATLAB codes were run on an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz with 64 processor cores (non-exclusively). The time per function evaluation for dimensions 2, 3, 5, 10, 20, and 40 can be seen in the following table (all values are given in $*10^{-4}$ seconds):

Algorithm	2-D	3-D	5-D	10-D	20-D	40-D
P-All	0.99	1.4	1.0	1.0	1.4	2.0
P-instances	1.2	0.97	0.80	1.1	1.3	2.0
P-range	0.88	0.96	0.91	0.94	1.5	2.2
P-StPt	1.3	1.0	0.88	0.96	1.4	2.1
P-Final	1.1	1.0	0.97	1.1	1.3	2.2
M-2017	24	14	5.6	1.8	1.0	1.0

MATLAB's BFGS is slower than Python's BFGS for small dimensions and, most likely due to its inherent parallelization, becomes quicker per function evaluation than the Python implementation for dimensions larger than 10.

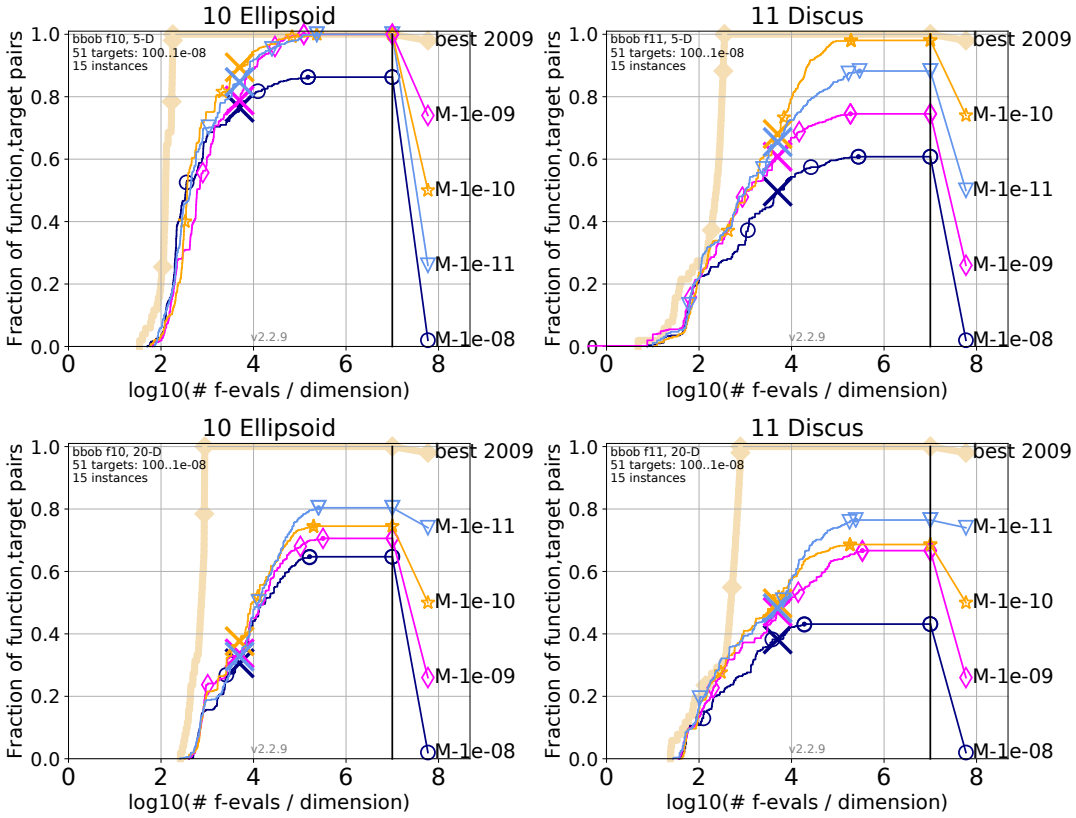

Figure 2: Expected running time by dimension for dimension 5 (upper figures) and 20 (lower figures) for different values of the parameter *FiniteDifferenceStepSize* used as a step size factor for finite differences in Matlab's BFGS (values 10^{-8} (default), 10^{-9}, 10^{-10}, and 10^{-11}).

6 RESULTS

The aim of this section is to answer the remaining two scientific questions, mentioned above: the impact of the algorithmic and experimental setups on the performance as well as the question of how much the actual implementations of the BFGS algorithm differ in different languages and over time.

6.1 Influence of the Experimental Set-up

When looking at the summary graphs, aggregating over all 24 bbob functions in Figs. 3 and 4, we do not see a large effect of the experimental setup—neither a changing starting condition nor the change of the instances have a large effect. However, this changes if we look at single functions.

First of all, the starting point has the biggest impact on the Griewank-Rosenbrock F8F2 function (f_{19}) where choosing the origin as the first search point improves significantly over a random choice—solving immediately about 30% of the targets with the first evaluation. To a lesser extend, the positive impact of evaluating the origin first can also be seen for the rotated Rosenbrock function (f_9) and in low dimension on the Gallagher 101 peaks function (f_{21}). This effect, however, is quite specific to the choice of the test functions in COCO and rather a defect in the test suite than a desirable algorithm feature.

Changing the initial sampling from within $[-5, 5]^n$ to $[-4, 4]^n$ on the contrary has no clear effect—neither positive nor negative.

Changing the instances (here for the comparison between the instances of BBOB-2009 and BBOB-2017) can also have an impact on the results, though overall small. Examples of differences, without a trend towards easier or harder years of the BBOB workhop instances can be seen on f_{10}, f_{11}, and f_{14} for the larger budgets/harder targets and for f_{21} for the easier targets and thus smaller budgets. The observed differences, however, are rather small, meaning that the instances of a given function represent globally the same difficulties over the years.

6.2 Comparison between Python's and MATLAB's BFGS

Arguably the most surprising differences can be observed between the MATLAB and the Python implementations of BFGS. Compared to the small effects of internal algorithm parameters and experimental setup, it plays a significant role which algorithm implementation we choose. The implementations in MATLAB and Python show entirely different ECDF characteristics where deviations become larger in higher dimensions.

The most significant differences can be observed on the ill-conditioned function group where the python versions are all dominating the MATLAB versions with the exception of the `scipy` implementation using the "Basin Hopping" strategy from [1], see below for details on this algorithm.

When looking at single functions, the 2009 Python and MATLAB versions are still almost similar on 15 of the 24 bbob functions (in 5-D) and on 16 functions in 20-D. The largest deviations can be observed on the ellipsoid and discus functions and for high precisions (i.e. difficult targets) on the bent cigar, sharp ridge and different powers functions. The latter might come from different handling of numerical imprecisions. Differences between the Python and MATLAB implementations becomes larger in higher dimension.

Comparison between Matlab 2009 and Matlab 2017. Benchmark results comparing algorithms in the same software package but from different versions are very rare and one would typically expect that the performance of an algorithm increases over time rather than the opposite. For the default MATLAB implementation of BFGS, however, the opposite is true: the R2017a version is clearly worse than the data set from 2009 provided by Ros.

M-2017 performs worse than M-2009 on many problems, especially on the easier ones. Both algorithms typically perform almost identical in the beginning of the search and at some point, the 2017 version degrades compared to the 2009 version. This is well seen on the separable and non-separable ellipsoid, the attractive sector, the bent cigar, and the Gallagher functions and, in higher dimension, also on the Rosenbrock functions. The opposite is the case for the discus, sharp ridge, and sum of different powers functions where the 2009 version becomes worse at some point.

Unfortunately, no installation of a 2009 MATLAB version is available in order to investigate in depth where these performance differences come from. It is not unlikely that the experiments of Ros from BBOB-2009 included a different parameter setting that was not mentioned in the original publication.

In order to double-check that it is not a bad parameter setting for BFGS' epsilon parameter that causes the observed differences, one can look at the impact of the change in epsilon from the above mentioned parameter tuning experiment and observes that the changes in performance caused by a different epsilon value are far smaller than the differences observed between the 2009 and the 2017 MATLAB versions (and also smaller than the sensitivity of Python's BFGS on its internal parameters).

Effects of Basin Hopping: Weak global structure vs adequate global structure. Last, we look in detail on the Basin Hopping strategy which differentiates Baudiš' BFGS version from the ones benchmarked here. Baudiš' BFGS algorithm (P-ScipyB) performs clearly different from the other BFGS variants, even if it is the same algorithm to begin with.

The performance of P-ScipyB is as expected. On functions with clear global structure, that is for which the function values of neighboring local optima are highly correlated, the Basin hopping strategy works well: if a small perturbation at a restart allows to "hop" to a neighboring basin of attraction with a corresponding local optima with better function value, the performance is increased. An example of such a well-structured function is the Rastrigin function. If there is no or only a weak global structure or in other words, if the local optima are randomly placed like for the Gallagher functions, "basin hopping" is ineffective and it is better to perform entirely independent restarts.

7 CONCLUSION

Application engineers often rely on benchmarking results when they have to choose an optimization algorithm to solve a given real-world (black-box) optimization problem. If not much knowledge about the chosen algorithm is available, practitioners fall back on default settings and thus often assume that the default settings do not change the performance over different implementations and/or that new software updates rather improve the algorithm behavior than to degrade it.

In this paper, we have investigated the Python and MATLAB implementations of BFGS, one of the most common (non-linear) optimization algorithms in a black-box setting (estimating the gradients by finite differences). Extensive numerical experiments on the well-known bbob function suite from the COCO platform reveal some interesting and at least unexpected behavior: (i) differences between the Python and MATLAB implementations are larger than the effects of certain internal parameter changes, (ii) the performance of Python's BFGS implementation from the `scipy` module gives consistently better results than the MATLAB version, and (iii) the comparison between the 2009 MATLAB version as benchmarked at BBOB-2009 by Ros shows better performance than the latest R2017a version of the same algorithm.

Moreover, we have also clarified with our experiments what are the performance impacts when changing the initialization of the algorithm and the set of instances coming from the COCO platform. Compared to the differences among the algorithm implementations, these differences are rather small on the bbob test suite, indicating that it will become most important in the future to educate the optimization community to pay more attention when generalizing statements on algorithm performance in particular when only a single algorithm implementation is tested.

REFERENCES

[1] Petr Baudiš. 2014. COCOpf: An algorithm portfolio framework. *arXiv preprint arXiv:1405.3487* (2014).

[2] 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.

[3] N. Hansen, A Auger, D. Brockhoff, D. Tušar, and T. Tušar. 2016. COCO: Performance Assessment. *ArXiv e-prints* arXiv:1605.03560 (2016).

[4] N. Hansen, A. Auger, S. Finck, and R. Ros. 2012. *Real-Parameter Black-Box Optimization Benchmarking 2012: Experimental Setup.* Technical Report. INRIA. http://coco.gforge.inria.fr/bbob2012-downloads

[5] N. Hansen, A. Auger, O. Mersmann, T. Tušar, and D. Brockhoff. 2016. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. *ArXiv e-prints* arXiv:1603.08785 (2016).

[6] N. Hansen, S. Finck, R. Ros, and A. Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions.* Technical Report RR-6829. INRIA. http://coco.lri.fr/downloads/download15.03/bbobdocfunctions.pdf Updated February 2010.

[7] N. Hansen, T. Tušar, O. Mersmann, A. Auger, and D. Brockhoff. 2016. COCO: The Experimental Procedure. *ArXiv e-prints* arXiv:1603.08776 (2016).

[8] 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

[9] Raymond Ros. 2009. Benchmarking the BFGS algorithm on the BBOB-2009 function testbed. In *GECCO (Companion)*, Franz Rothlauf (Ed.). ACM, 2409–2414.

[10] Stephen Wright and Jorge Nocedal. 1999. Numerical optimization. *Springer Science* 35, 67-68 (1999), 7.

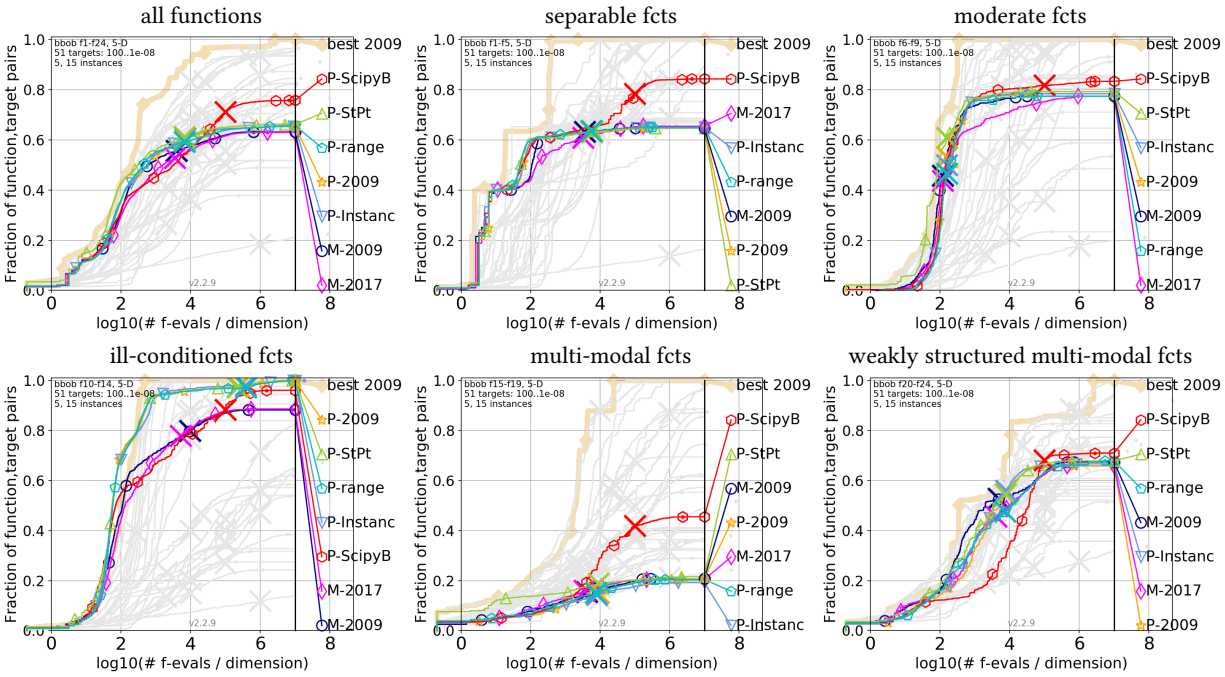

Figure 3: Bootstrapped empirical cumulative distribution of the number of function evaluations divided by dimension for 51 targets with target precision in $10^{[-8..2]}$ for all functions and subgroups in 5-D. As reference algorithm, the best algorithm from BBOB 2009 is shown as light thick line with diamond markers. In the background, all BBOB 2009 algorithms in gray.

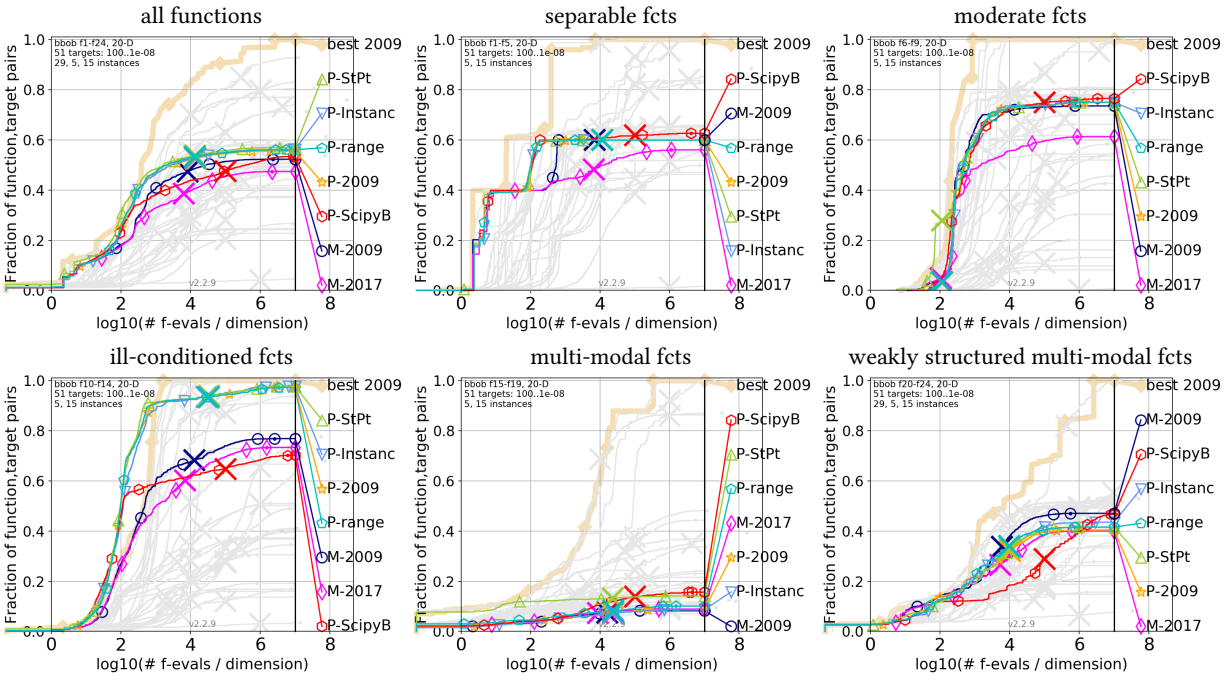

Figure 4: Bootstrapped empirical cumulative distribution of the number of function evaluations divided by dimension for 51 targets with target precision in $10^{[-8..2]}$ for all functions and subgroups in 20-D. As reference algorithm, the best algorithm from BBOB 2009 is shown as light thick line with diamond markers. In the background, all BBOB 2009 algorithms in gray.

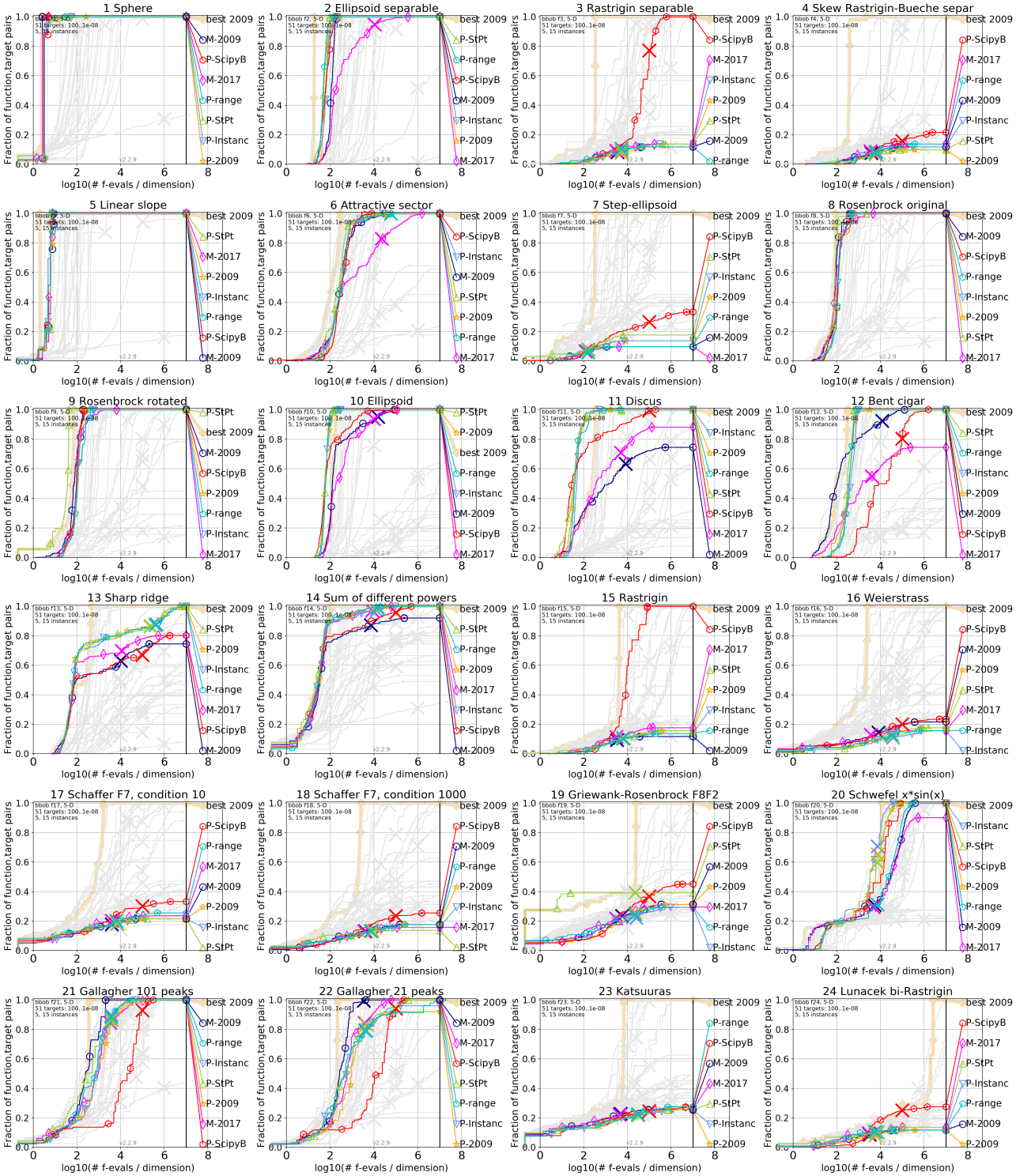

Figure 5: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of objective function evaluations, divided by dimension (FEvals/DIM) for the 51 targets $10^{[-8..2]}$ in dimension 5.

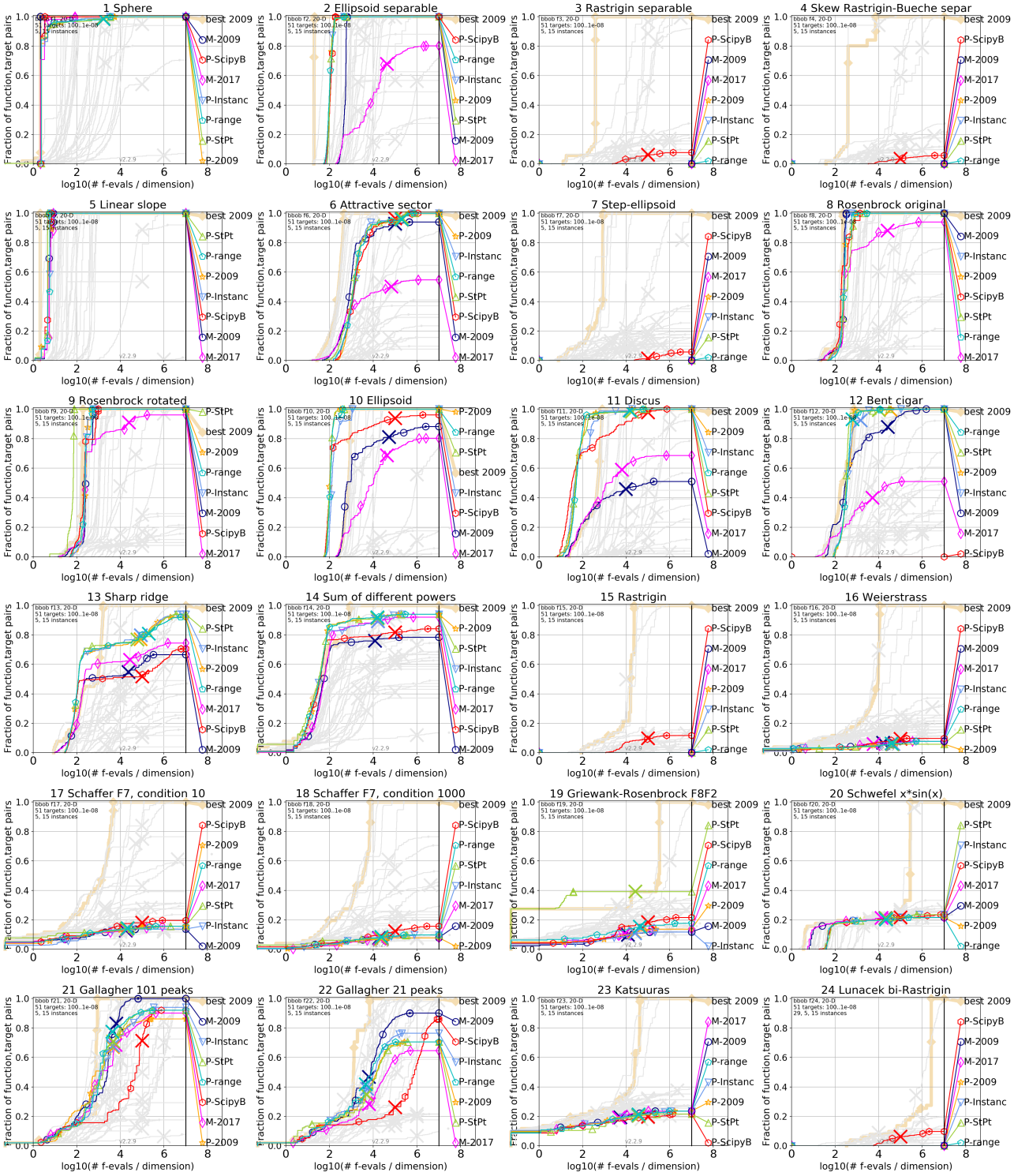

Figure 6: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of objective function evaluations, divided by dimension (FEvals/DIM) for the 51 targets $10^{[-8..2]}$ in dimension 20.