COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting

Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, Dimo Brockhoff

COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting

Nikolaus Hansen[a,b], Anne Auger[a,b], Raymond Ros[c], Olaf Mersmann[d], Tea Tušar[e], and Dimo Brockhoff[a,b]

[a]Inria, Randopt team, Palaiseau, France; [b]CMAP, CNRS, Ecole Polytechnique, Institut Polytechnique de Paris, Palaiseau, France; [c]Université Paris-Sud, LRI, UMR 8623, Inria Saclay, France; [d]TU Dortmund University, Chair of Computational Statistics, Germany; [e]Jožef Stefan Institute, Ljubljana, Slovenia

ARTICLE HISTORY
First version: March 2016, submitted: June 2019, final: August 2020;
to appear in *Optimization Methods and Software*
DOI https://doi.org/10.1080/10556788.2020.1808977

ABSTRACT
We introduce COCO, an open source platform for Comparing Continuous Optimizers in a black-box setting. COCO aims at automatizing the tedious and repetitive task of benchmarking numerical optimization algorithms to the greatest possible extent. The platform and the underlying methodology allow to benchmark in the same framework deterministic and stochastic solvers for both single and multiobjective optimization. We present the rationals behind the (decade-long) development of the platform as a general proposition for guidelines towards better benchmarking. We detail underlying fundamental concepts of COCO such as the definition of a problem as a function instance, the underlying idea of instances, the use of target values, and runtime defined by the number of function calls as the central performance measure. Finally, we give a quick overview of the basic code structure and the currently available test suites.

KEYWORDS
Numerical optimization; black-box optimization; derivative-free optimization; benchmarking; performance assessment; test functions; runtime distributions; software

1. Introduction

We consider the continuous black-box optimization or search problem to minimize

$$f : X \subset \mathbb{R}^n \to \mathbb{R}^m \qquad n, m \geq 1 \ , \tag{1}$$

where the search domain X is typically a bounded hypercube or the entire continuous space.[1] More specifically, we aim to find, as quickly as possible, one or several solutions $\boldsymbol{x}$ in the search space X with *small* value(s) of $f(\boldsymbol{x}) \in \mathbb{R}^m$.

[1]We later also consider integer variables. They are embedded in the continuous space and labeled for the solver as integers, see also Section 4.

A continuous optimization algorithm, denoted as *solver*, addresses the above problem. In this paper we only consider zero-order black-box optimization [19, 57, 58]: while the search domain $X \subset \mathbb{R}^n$ and its boundaries are accessible, no other prior knowledge about f is available to the solver.[2] That is, f is considered as a black-box, also known as an oracle, and the *only* way the solver can acquire information on f is by querying the value $f(\boldsymbol{x})$ of a solution $\boldsymbol{x} \in X$. Zero-order black-box optimization is thus a derivative-free optimization setting.[3] We generally consider "time" to be the number of calls to the function f and will define "runtime" correspondingly.

When $m > 1$ in Equation (1), we are in the setting of multiobjective optimization. Here, we only consider the case where $m \in \{1, 2\}$, whereas the presented framework is in general designed to be extendable to other settings (see also Sections 4 and 7).

From these prerequisites, benchmarking solvers seems to be a rather simple and straightforward task. We run a solver on a collection of problems and display the results. However, under closer inspection, benchmarking turns out to be surprisingly tedious. A set of objective functions has to be selected, problem instances should be derived, an experimental design has to be established, a set of performance measures has to be chosen, data have to be recorded, and results have to be exposed and interpreted in a comprehensive and comprehensible way. Each of these steps asks for a great number of subtle decisions and is yet crucial for the validity of the outcome (the chain is only as strong as its weakest link). In particular, we require here to get results that can be meaningfully interpreted beyond the standard conclusion that on some problem some solver is better than another.[4]

The difficulty of proper experimental analysis and benchmarking of solvers has already been recognized in previous work, see in particular [9, 10, 51], which also give guidelines for better experimental work. In our work, we offer a conceptual guideline for benchmarking continuous optimization algorithms addressing these challenges that is implemented within the COCO framework.[5]

The COCO framework provides the following practical means for an automatized black-box optimization benchmarking procedure (see also Figure 1):

- an interface to several languages in which the benchmarked solver can be written, currently C/C++, Java, Matlab/Octave and Python,
- several suites of test problems, currently all written in C, where each problem can assume an arbitrary number of pseudo-randomized instances,
- data logging facilities,
- data post-processing written in Python that produces various plots and tables,

[2] In the multiobjective case also the upper values of interest in f-domain are provided, see also Appendix C.

[3] In [1], blackbox optimization (BBO) is defined as "*the study of design and analysis of algorithms that assume the objective and/or constraint functions are given by blackboxes*", and a blackbox is defined as "*. . . any process that when provided an input, returns an output, but the inner workings of the process are not analytically available. The most common form of blackbox is computer simulation, but other forms exist, such as laboratory experiments for example*". We prefer to define the black-box setup solely from the *interfacing* between problem and solver and the exchange of information, which is to a large degree independent of the underlying problem (e.g. its analytical nature or the availability of gradients). In our case, the inner workings of the black-boxes *are* analytically available to us, but the solver is not allowed to access or use them.

[4] A common major flaw, unless data or performance profiles are used, is to have no indication of *how much* better a solver is. That is, benchmarking results often provide no indication of *relevance*, for example, when the main output consists of hundreds of tabulated numbers only interpretable on an ordinal (ranking) scale. This problem is connected to the common shortcoming of not clearly distinguishing *statistical significance* and *relevance*. Statistical significance is only a secondary and by no means sufficient condition for relevance.

[5] COCO has been continuously developed since 2008. For implementation details, confer to the code basis on GitHub and the C API documentation.

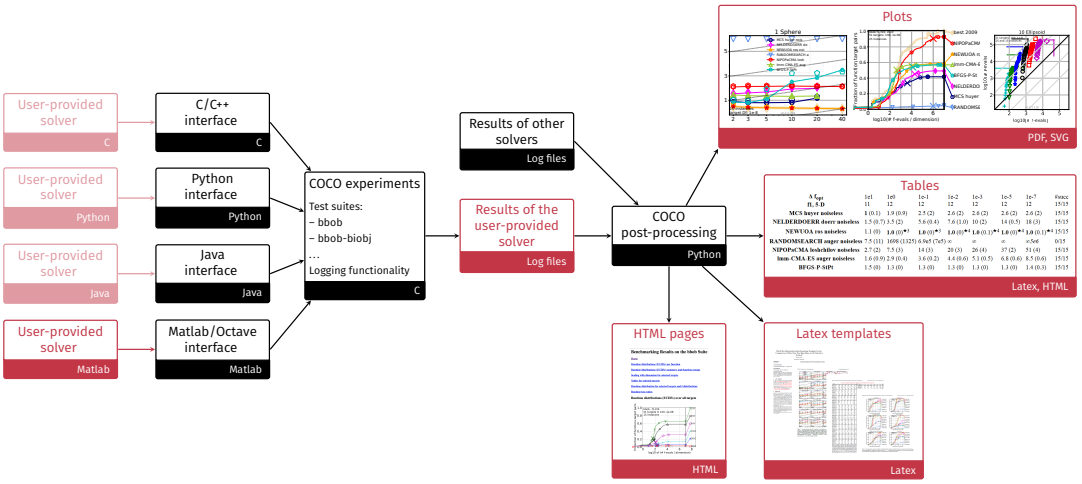


Figure 1. Overview of the COCO platform. COCO provides all black parts while users only have to connect their solver to the COCO interface in the language of interest, here for instance Matlab, and to decide on the test suite the solver should run on. The other red components show the output of the experiments (number of function evaluations to reach certain target precisions) and their post-processing and are automatically generated. For the *results of other solvers* part, COCO provides data from more than 200 previously-run benchmark experiments with a large variety of solvers collected over the last ten years from dozens of researchers.

- empirical results of other solvers that can be used for comparison,[6]
- HTML pages assembling these plots and tables to ease their inspection,
- LaTeX templates that include some selected results.

The underlying philosophy is to provide everything that experimenters need to set up and implement when they want to benchmark a given solver implementation *properly*. A desired side effect of reusing the same framework is that data collected over years or even decades can be effortlessly compared.[7] So far, the framework has been successfully used to benchmark over 200 different solvers by dozens of researchers. The data from all these experiments are openly accessible and can be seamlessly used directly within the COCO post-processing as will be showcased later. These data come from solvers of different nature: deterministic (pattern-search-based, trust-region derivative-free optimization, quasi-Newton, ...) and stochastic (evolution strategies, Bayesian optimization, differential evolution, ...) addressing single- and multiobjective problems.

The purpose of this paper is to present the COCO platform and give an overview of the main ideas it is based upon. The remainder of this section discusses related work and terminology that will be used later on. Section 2 discusses the motivations and objectives behind COCO. Section 3 presents the central theses describing our approach to benchmarking methodology. Section 4 gives a summary of already available test suites. Section 5 presents practical examples how to use the COCO software. Sections 6 and 7 show some usage statistics and extensions under development. Section 8 concludes the paper with a summary and discussion. The appendices provide details on how we chose test functions, the difference to competitive testing, and how we measure performance on biobjective functions.

[6]COCO provides a software environment for black-box optimization benchmarking but neither a server to run experiments nor the solvers themselves.

[7]See http://coco.gforge.inria.fr/data-archive and `cocopp.archives` in the `cocopp` Python module to access all collected data, in particular those submitted to the Black-Box Optimization Benchmarking (BBOB) workshop series at the GECCO conference.

1.1. Related Work

Benchmarking solvers is an important task in optimization, independent of which type of problem is tackled. Experimental studies reach back as far as the 1950s [45] and we refer to [10] for a recent summary of the history of benchmarking. A few benchmarking *platforms* have been proposed over the years alongside COCO that also facilitate the task of analysing data from benchmarking experiments.

Paver, the Performance Analysis and Visualization for Efficient Reproducibility software [20] is offered in the context of the GAMS World initiative as a collection of Python scripts and available on GitHub. Paver allows to read in CSV-formatted files of numerical benchmarking experiments from an arbitrary number of solvers and offers "counting solver runs with certain properties, computing mean values and quantiles of solver run attributes, and performance profiles" [20], in particular in an HTML-based format.

A similar analysis is supplied by the platform. Developed as a generic visual tool for benchmarking on arbitrary (combinatorial, numerical, etc.) problems, it provides data profiles and median convergence plots for a set of benchmarking experiments from an arbitrary number of solvers.[8] The software is written in Java and is available in a dockerized version. In contrast to Paver, where the user invokes the analysis from the command line, the user interacts with the optimizationBenchmarking.org framework via the browser.

The latest software for analysing solver data is the IOHProfiler [23] which is inspired by the COCO platform. Its post-processing part, in particular, allows to visualize solver performance (given in COCO format) interactively in the browser and offers both fixed-target and fixed-budget views.

In the context of benchmarking multiobjective solvers, a few additional software packages should be mentioned. PISA presented one of the first attempts to integrate the collection of data from benchmarking experiments and their analysis. It provides both multiobjective test problems (discrete and continuous; toy as well as real-world problems) and an extensive set of scripts to assess the solvers' performance in terms of quality indicator measures such as the hypervolume indicator and various epsilon indicators [13]. A special feature of PISA is the split of the solvers into a selector and a variator part where the latter contains the variation operators for a specific (set of) optimization problems. A disadvantage of PISA is that it has not been actively maintained for a long time.

Newer platforms that attempt to benchmark multiobjective solvers are the MOEA Framework (in Java), its Python counterpart Platypus, jMetal (originally in Java, now also in other languages) [25], PLATEMO (in Matlab) [67], and ecr (in R) [14]. All these platforms focus on providing a large amount of solvers and a comprehensive set of test functions and are open source. Performance assessment is typically restricted to statistical tests and tabular visualizations of achieved quality indicator values at certain budgets although certain proprietary products such as ModeFRONTIER provide additional visual output.

In all these platforms, except for the commercial ModeFRONTIER, the entire benchmarking experiment is only semi-automated in that the user still has to decide on the concrete benchmarking experiments and all their intricacies. At most the data collection and visualization are automated. COCO, on the contrary, provides concrete benchmarking suites and a predefined setup for the experiments in order to

[8] The optimizationBenchmarking.org framework allows to read in data from the COCO platform and one of the introductory examples displays the COCO data from the BBOB-2013 workshop.

facilitate the comparison of a large number of solvers. Providing the corresponding solver data sets from COCO experiments online[9] and also directly through the COCO post-processing interface (see Section 5.2) is another unique aspect that sets COCO apart.

Besides entire benchmarking platforms, many collections of test functions have been proposed. Examples are the problems collected by Moré, Garbow, and Hillstrom [54], Hock and Schittkowski [44], Schittkowski [64], Mittelmann, Neumaier, the CUTEr/st suite or the randomly generated, but structured problems from the GKLS generator [30]. The GAMSWorld webpage lists further common problem suites. Other sets of test functions to mention are those used for the competitions at the CEC conferences since 2005. They evolved quite a bit over the years and nowadays use the same technique of problem transformations as introduced with the `bbob` functions of the COCO platform.

Several test suites with multiobjective problems have also been proposed, of which several possess questionable properties, see [16] for a discussion.

The majority of the mentioned test suites contain a large proportion of simple-to-solve and non-scalable problems without instance variations. Aggregating the performance over all functions of such a test suite and interpreting the results must therefore be done with care—because the distribution of function difficulties in a benchmarking experiment clearly determines which solvers will excel. Alternatively, COCO aims to implement suites with balanced difficulties observed in practice and with a bias towards difficult-to-solve functions.

1.2. Terminology

We specify a few terms which are used later.

function We talk about an objective *Function* as a parametrized mapping $\mathbb{R}^n \to \mathbb{R}^m$ with scalable input space, $n \geq 1$, and usually $m \in \{1, 2\}$ (single- and bi-objective). Functions are parametrized such that different *instances* of the "same" function are available, e.g., translated or shifted versions.

quality indicator measure From the function values of a set of solutions we compute a scalar *Quality indicator* value which is used in the performance assessment. By convention and abuse of naming, the "quality" indicator is minimized. The indicator assigns to any solution *set* a real value. Our simplest quality indicator measure is the minimum of all so-far-observed f-values. In the multiobjective case, a typical example is the hypervolume of all so-far-observed solutions (where the position of the reference point is a further design decision to be made).

problem We talk about a *Problem*, as a specific *function instance* on which a solver is run. A problem can be evaluated for a solution $\boldsymbol{x} \in \mathbb{R}^n$ and returns $f(\boldsymbol{x})$. In the context of performance assessment, a target value is added to define a problem. A problem is considered as solved when the quality indicator value reaches this target, which is always given as a precision value, i.e., as a deviation from a (supposedly) optimal value.

runtime We define *Runtime* or *run-length* [47] as the *number of function evaluations* or f-evaluations conducted on a given problem until a prescribed target value is hit. Runtime is our central performance measure.

suite A test or benchmark *Suite* is a collection of problems in different dimensions. It typically consists of 1000 to 5000 problems (number of dimensions $\times$ number of functions $\times$ number of instances), where the number of objectives m is fixed.

[9] See http://coco.gforge.inria.fr/doku.php?id=algorithms

2. Why COCO?

Our aim in providing the COCO platform is threefold:

- diminish the time burden, the pitfalls, the bugs and the omissions of the repetitive coding task of setting up and running a benchmarking experiment,
- provide a *conceptual guideline for better benchmarking*, and
- provide a growing archive of comparative benchmarking data to the scientific community.

Our setup has a distinct boundary between the implementation of benchmark functions on the one hand and the experimental design, data collection and presentation on the other hand. Our benchmarking guideline has the following defining features.

(1) We benchmark solvers on a set (a suite) of benchmark functions. Benchmark functions are
   (a) used as black boxes for the solver, however, they are explicitly known to the scientific community,
   (b) designed to be comprehensible, to allow a meaningful interpretation of performance results,
   (c) difficult to "defeat" or exploit, that is, they should not have artificial regularities and artificial symmetries that can easily be exploited (intentionally or unintentionally),[10]
   (d) scalable with the input dimension [73],
   (e) instantiated from an arbitrary number of pseudo-randomized versions, i.e, instances,
   (f) models for "real-world" problems; the currently available test suites (see Section 4) model in particular well-known problem *difficulties* like ill-conditioning, multimodality and ruggedness.

   The input parameters to the solver must not depend on the specific function within the benchmark suite. They can, however, depend on the black-box *interface* (i.e., the signature of the function), namely on dimension and search domain of interest (e.g., to set variable bounds).

(2) There is no predefined budget (number of f-evaluations) for running an experiment, the experimental procedure is *budget-free* [41]. Specifically, all results are comparable irrespectively of the chosen budget and up to the smallest budget in the compared results. Hence, the larger the budget, the more data are generated to compare against. The smaller the budget, the less meaningful conclusions are possible (which becomes most evident when the budget approaches zero). This also implies an anytime assessment approach: the performance is not (only) measured after some given runtime or fixed budget or after reaching some given target but over the entire run of the solver.

(3) *Runtime*, measured in number of f-evaluations [34], is the only used performance measure. It is further aggregated and displayed in several ways. The advantages of runtime are that it
   - is independent of the computational platform, language, compiler, coding style, and other specific experimental conditions[11],

[10] For example, the global optimum is not in all-zeros, optima are not placed on a regular grid, most functions are not separable [73]. The objective to remain comprehensible makes it more challenging to design non-regular functions. Which regularities are common place in real-world optimization problems remains an open question.

[11] Runtimes measured in f-evaluations are widely comparable and designed to stay. The experimental procedure includes, however, also a timing experiment which records the internal computational effort of the solver in

- is independent, as a measurement, of the specific function on which it has been obtained, that is, *"taking 42 evaluations"* has the same meaning on any function, while *"reaching a function value of 42"* has not,
- is relevant, meaningful and easily interpretable without expert domain knowledge,
- is quantitative on the ratio scale[12] [66],
- assumes a wide range of values, and
- aggregates over a collection of values in a meaningful way.

A *missing* runtime value is considered as a possible outcome (see Section 3.2).

(4) The display of results is done with the distinct effort to be as comprehensible, intuitive and informative as possible. In our estimation, details can matter a lot.

We believe, however, that in the *process* of solver *design*, a benchmarking framework like COCO has its limitations. During the design phase, usually

- fewer benchmark functions should be used,
- the functions and measuring tools should be tailored to the given solver and the design question, and
- the overall procedure should be more informal and interactive with rapid iterations.

A benchmarking framework then serves to conduct the formalized validation experiment of the design *outcome* and can be used for regression testing. Johnson [51] and Hooker [46] provide excellent discussions of how to do experimentation beyond the specific benchmarking scenario.

3. Benchmarking methodology

This section details the benchmarking methodology used in COCO which, in particular, allows for a budget-free experimental design. We elaborate on functions, instances, problems, runtime, target values, restarts, simulated restarts, performance aggregation, and a budget-dependent benchmarking setup.

3.1. Functions, Instances, and Problems

In the COCO framework we consider **functions**, f_i, for each suite distinguished by their identifier $i = 1, 2, \ldots$. Functions are further *parametrized* by the (input) dimension, n, and the instance number, j. We can think of j as an index to a continuous parameter vector setting. It parametrizes, among other things, search space translations and rotations. In practice, the integer j identifies a single instantiation of these parameters. For a given m, we then have

$$f_i^j \equiv f[n, i, j] : \mathbb{R}^n \to \mathbb{R}^m \quad \boldsymbol{x} \mapsto f_i^j(\boldsymbol{x}) = f[n, i, j](\boldsymbol{x}) \ .$$

Varying n or j leads to a variation of the same function i of a given suite. Fixing n and j of function f_i defines an optimization **problem** $(n, i, j) \equiv (f_i, n, j)$ that can be presented to the solver. Each problem receives again an index within the suite, mapping the triple (n, i, j) to a single number.

CPU or wall clock time [41].

[12] As opposed to a ranking of solvers based on their solution quality achieved after a given budget of evaluations.

As this formalization suggests, the differentiation between function and instance index is of purely semantic nature. This semantics is, however, important in how we interpret and display results. We interpret **varying the instance** parameter as a natural randomization for experiments[13] in order to

- generate repetitions on a single function for deterministic solvers, making deterministic and non-deterministic solvers directly comparable (both are benchmarked with the same experimental setup)[14],
- average away irrelevant aspects of the function definition,
- alleviate the problem of overfitting, and
- prevent exploitation of artificial function properties,

thereby providing a "fairer" and more robust setup. For example, we do not consider the absolute position of the optimum as a defining function feature. Therefore, in a typical COCO benchmark suite, instances with different (pseudo-randomized) search space translations are presented to the solver. If a solver is translation invariant (and hence ignores domain boundaries), this is equivalent to varying the initial solution.

3.2. Runtime and Target Values

In order to measure the runtime of a solver on a problem, we consider a hitting time condition. We define a non-increasing *quality indicator measure* and prescribe a set of **target values**, t [15, 17, 34]. In the single-objective unconstrained case, the quality indicator is the best so-far-seen f-value.[15] For a single run, when the quality indicator reaches or surpasses a target value t on problem (f_i, n, j), we say the solver has *solved the problem* (f_i, n, j, t)—the solver was successful.[16] We typically collect runtimes for around a hundred different target values from each *single* run.

Target values are directly linked to a problem, and we leave the burden of defining the targets with the designer of the benchmark suite.[17] More specifically, we consider the problems $(f_i, n, j, t(i, j))$ for all functions i and benchmarked instances j. The targets $t(i, j)$ depend on the problem instance in a way to make certain problems comparable. We typically define the (absolute) target values from a single set of *precision* values added to a reference offset which is the known (or estimated) optimal indicator value. Runtimes from the same subset of target precision values are aggre-

[13]Changing or sweeping through a relevant feature of the problem class, systematically or randomized, is another possible usage of instance parametrization.

[14]The number of instances and the number of runs per instance may be varied. By default, all solvers execute one run per instance and instances are interpreted as repetitions. We initially used a setup with 3 runs per instance for stochastic solvers which would allow to distinguish within- and between-instance variance. Either setup is available to the user and compatible with the data processing and aggregation.

[15]In the constrained, multiobjective and noisy cases, the quality indicator measure is more intricate. In the multiobjective case, for example, we use a version of the hypervolume indicator of all so-far evaluated solutions and approximate the optimal value from experimental data in order to derive informative target values [17]. For more information, please see Appendix C.

[16]Reflecting the *anytime* aspect of the experimental setup, we use the term *problem* in two meanings: (a) for the problem the solver is benchmarked on, (f_i, n, j), and (b) for the problem, (f_i, n, j, t), a solver may solve by hitting the target t with the runtime, $\mathrm{RT}(f_i, n, j, t)$, or may fail to solve within the experimentation budget. Each problem (f_i, n, j) gives raise to a collection of *dependent* problems (f_i, n, j, t). Viewed as random variables, given (f_i, n, j), the $\mathrm{RT}(f_i, n, j, t)$ are not independent for different values of t. In particular, the Cumulative Distribution Function (CDF) for any larger t dominates the CDF for any smaller (i.e., more difficult) t.

[17]The alternative, namely to present the obtained f- or indicator-values as results, leaves the (often rather insurmountable) burden to interpret the meaning of the indicator values to the experimenter or the final audience. Fortunately, budget-based targets are a generic way to generate target values from observed runtimes as discussed at the end of the section.

gated over problems or functions. Only the precision values and not the (by themselves meaningless) target values are exposed to the user.

The **runtime** is the evaluation count when the target value t was reached or surpassed for the first time. That is, runtime is the number of f-evaluations needed to solve the problem (f_i, n, j, t). *Measured or numerically bootstrapped runtime values are in essence the only way how we assess the performance of solvers.*

If a solver does not hit the target t in a given single run, the run is considered to be *unsuccessful*. The runtime of this single run remains undefined, but is bounded from below by the number of evaluations conducted during the run. Naturally, increasing the budget that the solver is allowed to use increases the number of successful runs and hence the number of available runtime measurements. Therefore, larger budgets are preferable. However, larger budgets should not come at the expense of abandoning prudent termination conditions. Instead, restarts should be conducted (see also Section 3.3).

As an alternative to predefined target precision values, we also propose to select target values based on the runtimes of a set of solvers similar to data profiles. For any *given budget*, we select the associated target in the following way [34]: from the finite set of recorded target values, we pick the easiest (i.e., largest) target for which the expected runtime of all solvers (ERT, see Section 3.4) exceeds the budget.[18] The resulting target value depends on a set of solvers and on the given function and dimension. Starting with a set of budgets of interest (e.g., relevant budgets for an application of interest), we compute this way a set of **budget-based target values** (AKA run-length-based targets) for each function and dimension. This choice of target values does not require any knowledge about the underlying functions and their indicator values, but it requires experimental data and depends on the chosen set of solvers.

3.3. Restarts and Simulated Restarts

Any solver is bound to terminate and, in the single-objective case, return a single recommended solution, $\boldsymbol{x}$, for the problem, (f_i, n, j). More generally, we assume an anytime and any-target scenario, considering a non-increasing quality indicator value computed in each time step from all evaluated solutions. Then, at any given time step, the solver solves all problems (f_i, n, j, t) for which the current indicator value hit or surpassed the target t.

Independent restarts from different, randomized initial solutions are a simple but powerful tool to increase the number of solved problems [43]—namely by increasing the number of t-values, for which the problem (f_i, n, j) was solved. Increasing the budget by independent restarts will increase the success rate, but generally does not change the performance assessment in our methodology, because the additional successes materialize at higher runtimes [34]. Therefore, we call our approach *budget-free*. Restarts, however, "*improve the reliability, comparability, precision, and 'visibility' of the measured results*" [41].

Simulated restarts [34, 42] are used to determine a runtime for unsuccessful runs. Semantically, simulated restarts are only meaningful if we can interpret different instances as *random repetitions* that could arise, for example, by restarting from different initial solutions on the same instance (hence we do not use simulated restarts for the multiobjective case). Resembling the bootstrapping method [26] when we face an un-

[18] That is, we take the best solver as reference to compute budget-based targets. Instead of the best solver, we could also take the median or (least promising) the worst solver or pure random search as reference.

successful run, we draw further runs from the set of tried problem instances, uniformly at random with replacement, until we find an instance, j, for which $(f_i, n, j, t(i, j))$ was solved. The evaluations done on the first and on all subsequently drawn unsolved problems are added to the runtime on the last, solved problem and are considered as the runtime on the originally unsolved problem instance. This method is only applicable if at least one problem instance was solved and is applied if at least one problem instance was not solved. By their nature, the success probability of "runs" with simulated restarts is either zero (if no problem instance was solved) or one. *Simulated restarts allow to directly compare solvers with vastly different success probabilities.*

3.4. Aggregation

A typical benchmark suite consists of about 20–100 functions with 5–15 instances for each function. For each instance, up to about 100 targets are considered for the performance assessment. This means we consider at least $20 \times 5 = 100$, and up to $100 \times 15 \times 100 = 150\,000$ runtimes to assess performance. To make them amenable to examination and interpretation, we need to summarize these data.

The semantic idea behind aggregation is to compute a statistical summary over a set or subset of problems of interest *over which we assume a uniform distribution.* From a practical perspective, this means we assume to face each problem with similar probability *and* we have no simple way to distinguish between these problems to select a solver accordingly. If we can distinguish between problems easily, for example, according to their input dimension, the aggregation of data (for a single solver) is rather counterproductive. Because the dimension is known in advance and can be used for solver selection, we never aggregate over dimension. This has no significant drawback when all functions are scalable in the dimension.

We use several ways to aggregate the measured runtimes.

- Empirical cumulative distribution functions of runtimes (runtime ECDFs), also denoted as (empirical) runtime distributions. In the domain of numerical optimization, ECDFs of runtimes to reach a given *single* target precision value are well-known as *data profiles* [55]. Performance profiles are ECDFs of these runtimes relative to the respective best solver [24]. We favour absolute runtime distributions over performance profiles for two reasons:
  (1) Runtime ECDFs are unconditionally comparable across different publications [51].[19] They are absolute performance measures (opposed to relative measures) and, in our case, do not depend on other solvers for normalization.[20] Performance profiles suffer from the deficiency of any relative comparison procedure: adding or removing a single entry may significantly affect the result of other pairwise comparisons [33, 53].
  (2) Runtime ECDFs allow to distinguish easy problems from difficult problems (for any given solver). In performance profiles, a small runtime ratio gives no

[19] While runtime distributions by themselves are comparable across different publications, this is generally not the case for data profiles. The latter compute the target value for each problem individually as $f_{\text{best}} + \tau(f_0 - f_{\text{best}})$, based on the best achieved f-value from a set of solvers within a given budget and for the precision parameter τ [55].

[20] This advantage partly disappears with budget-based target values, as considered in Section 3.2. Conceptually, performance profiles change the displayed measurement potentially depending on all displayed solvers. Budget-based target values, similar to the approach in data profiles, change the considered problems, namely the target precisions which define when a problem is solved, based on the performance of a set of solvers. Budget-based target values make the unconditionally comparable but somewhat arbitrary targets-to-reach setting somewhat less arbitrary.

indication of the problem difficulty at all and a large runtime ratio can still mainly be caused by a competitor which solves the problem very quickly. An easy-versus-difficult problem classification is, however, a powerful feature to select the best applicable solver [42].

We usually aggregate not only runtimes from a single target precision value, as in data profiles, but from several targets per function. Because we display the x-axis on a log scale, the area above the curve and the *difference area* between two curves are meaningful notions even when results from several problems are shown in a single graph. The geometric average runtime difference

$$\exp\left(\frac{1}{k}\sum_{i=1}^{k}\log\left(\frac{B_i}{A_i}\right)\right) = \exp\left(\frac{1}{k}\sum_{i=1}^{k}\log(B_i) - \frac{1}{k}\sum_{i=1}^{k}\log(A_i)\right) \qquad (2)$$

reflects this difference area between the graphs in log-display and is invariant under re-sorting of data.

The runtime distribution on a single problem over all targets is tightly related to the convergence graph that displays the so far best f-value against the number of f-evaluations. Consider the convergence graph with reversed y-axis (like for maximizing $-f$). Then the runtime distribution is a step function strictly below (but close to) this reversed convergence graph, where the maximal y-distance results from the target discretization. Hence, runtime ECDFs provide a single formalization for convergence graph data (when many targets are used) and data profiles (when a single target precision is used) with a smooth transition between the two.

By sorting the runtime values taken from several problems, runtime ECDFs disguise relevant information which can lead to a misinterpretation when one graph dominates another. Domination in each point of a data profile does not imply equal or better performance on *each problem* that is presented in the data profile. For example, consider the two runtimes on two problems, respectively, to be 50 and 500 for Solver A, and 1000 and 100 for Solver B. On the first problem, Solver A is 20 times faster than Solver B. On the second problem, Solver B is 5 times faster than Solver A. Yet, in the *sorted* data profiles, Solver A (50, 500) dominates Solver B (100, 1000) everywhere by a factor of two.

- Averaging, as an estimator of the expected runtime. The estimated expected runtime of the restarted solver, ERT, is often plotted against dimension to indicate scaling with dimension. The ERT, also known as Enes [60] or SP2 [7] or aRT, is computed by dividing the sum of all evaluations before the target was reached (from successful and unsuccessful runs) by the number of runs that reached the target. If all runs reached the target, this is the (plain) average number of evaluations to reach the target. Otherwise, the unsuccessful runs are fully taken into account as if restarts had been conducted until the target was reached. Like simulated restarts, ERT integrates out the observed success rate in the data. This also answers the question of how to weigh failure rate and/or solution quality versus speed [20] with a semantically and practically meaningful measure[21]. The *arithmetic* average is only meaningful if the underlying distribution of the values is similar and has light tails. Ideally, the values stem from the same distribution.

[21] A practically less meaningful but easier to obtain ad hoc measure, known as Q-measure [60] or SP1 [7], uses only successful runs to compute an analogous ratio. This can be useful to get a quick estimate of ERT without the need to bother with effective termination conditions, but is currently not used within COCO.

Otherwise, the average of log-runtimes, that is the *geometric* average (2) or a shifted geometric mean [31] are feasible alternatives.

- Simulated restarts, see Section 3.3, aggregate data from several runs into a "single run" to supplement a missing runtime value, similarly as in the computation of ERT. The same data can be used to simulate many runtimes via simulated restarts. These runtimes are usually plotted as empirical cumulative distribution function. The ERT is the expected runtime of these simulated restarts.

3.5. Budget-Dependent Benchmarking

The performance of some solvers depends on the total budget of function evaluations given as an additional parameter to the solver. Consider, for example, a hybrid solver that couples an explorative strategy with a local search method. A number of final function evaluations is typically reserved to additionally improve the best solutions [27]. Therefore, a single performance assessment of such solvers cannot be expected to faithfully predict their performance for budgets that are different from the one that was used in the experiments. The budgeted (non-anytime) performance of a budget-dependent solver can be better estimated by repeatedly running the solver with increasing budgets [70]. This overestimates the performance (underestimates the runtime), as if the "optimal" budget were known in advance and given as parameter to the solver. Depending on the number and size of the budgets, this can take a significant amount of extra time (in the worst case, the overhead is quadratic in the maximal budget). By using budgets that are equidistant on the logarithmic scale, however, the time complexity of the overhead depends linearly on the maximal budget, making the approach usable in practice [70].

4. Test Suites

An important feature of the COCO framework is that new test suites can be added with comparatively little effort, thereby getting all the benefits of the established benchmarking setup for the new suite. Currently, the COCO framework provides the following test suites (listed in order of their introduction):

`bbob` contains 24 functions in dimensions 2, 3, 5, 10, 20 and 40 in five subgroups: separable, moderate, ill-conditioned, multi-modal weakly structured, multi-modal with global structure [38].

`bbob-noisy` contains 30 noisy functions in dimensions 2, 3, 5, 10, 20 and 40 in three subgroups with three different noise models [39]. The code for this test suite is only available at coco.gforge.inria.fr.

`bbob-biobj` contains 55 bi-objective functions in dimensions 2, 3, 5, 10, 20 and 40 in 15 subgroups [16].

`bbob-biobj-ext` contains 92 bi-objective functions in dimensions 2, 3, 5, 10, 20, and 40, including all 55 `bbob-biobj` functions [16]. For the 37 new functions in the suite, the reference target values are not yet established.

`bbob-largescale` contains 24 functions in dimensions 20, 40, 80, 160, 320 and 640 and the same five subgroups as the `bbob` suite [71].

`bbob-mixint` contains 24 mixed-integer single-objective functions in dimensions 5, 10, 20, 40, 80 and 160 and the same five subgroups as the `bbob` suite [69]. Integer variables are embedded into the continuous space, that is, the problems

can be evaluated as continuous problems and appear in the integer variables as piecewise constant functions.

`bbob-biobj-mixint` contains 92 mixed-integer bi-objective functions in dimensions 5, 10, 20, 40, 80 and 160 and the same 15 subgroups as the `bbob-biobj` suite [69].

Test suites are crucial in that they ultimately define the anticipated purpose of the benchmarked solvers. For example, if we care about solving difficult problems more than about solving easy problems, test suites must contain more difficult than easy problems, etc.

The listed test suites were designed with the remarks from Section 2 in mind. In particular, they introduce a number of transformations in $\boldsymbol{x}$- and f-space in order to make the functions less regular and less symmetrical and hence less susceptible to exploits of the relatively simple underlying formulas.

The continuous variables of the functions are unbounded. The single-objective suites, however, guarantee that the global optimum is in a bounded domain that is known to the solver, hence bounded solvers can be, and frequently have been, benchmarked as well.

COCO allows to integrate new problem suites. The interface to integrate suites with arbitrary constraints is implemented and fully functional.[22] In order to play well with the performance measurement methodology, new test suites should feature

- definitions of all functions in various dimensions,
- pseudo-randomized function instances, for example, with different locations of the global optimum,
- the same target precision levels for all function instances (as used by the logger, however, disguised for the solver). A simple way to control the target levels function-wise is to multiply each "raw" function by a different positive scalar.
- In case of noisy (stochastic) functions, the f-distribution of any two solutions should either be the same or one distribution should dominate the other. This way, any ambiguity as to which solution is better can be avoided.

In the performance evaluation, the defined target levels become part of the problem definition (see Section 3.2).

5. Usage and Output Examples

The COCO platform implementation consists of two major parts (see also Figure 1):

The *experiments* part defines test suites, allows to conduct experiments, and provides the output data. The code base is written in C. COCO amalgamates as a pre-compilation step all C source code files into the files `coco.h` and `coco.c`. These two files suffice to compile and link to run all experiments. Interface wrappers for further languages are provided, currently for Java, Matlab/Octave, and Python.

The *post-processing* part processes the data and displays the benchmarking result. This is the central part of COCO. The code is written in Python and heavily depends on `matplotlib` [48], which might change in future versions. The post-processing part of COCO allows to compare experiments from multiple solvers,

[22] See https://github.com/numbbo/coco/blob/master/howtos/create-a-suite-howto.md and https://github.com/numbbo/coco/blob/master/code-experiments/src/coco.h for more details.

in particular, from hundreds of data sets provided by the framework.

Examples of using the two parts are presented in the following two sections.

5.1. Running an Experiment

Installing COCO in a linux system shell (with the prerequisites of git and Python installed) and benchmarking a solver in Python, say, the solver `fmin` from `scipy.optimize`, is as simple as shown in Figure 2, where the installation sequence is given above, an example to invoke the benchmarking from a shell is given in the middle, and a Python script for benchmarking is shown below.[23] The benchmarking scripts write experimental data into the `exdata` folder first, then invoke the post-processing, `cocopp`, writing figures and tables into the `ppdata` folder (by default). The last line of the Python script finally opens the file `ppdata/index.html` in the browser as shown in Figure 3, left, to visually investigate the resulting data.

5.2. Analysing the Results

We present an illustrative overview of the data analysis possibilities offered by the COCO platform. The outputs for the data analysis are generated by the `cocopp` Python module which is at the core of the COCO platform and is installed either as shown in Figure 2 (last line of the top box) or simply with the shell command

```
python -m pip install cocopp
```

The module can be accessed from the command line or within a Python/IPython shell or a Jupyter notebook.

The post-processing is directly invoked by the scripts in Figure 2, except when `example_experiment2.py` is run in several batches. In the latter case, we move the root output folders generated by each batch into a single folder, say `EXP_FOLDER`, and evoke the post-processing by the shell command

```
python -m cocopp EXP_FOLDER
```

or by typing

```
import cocopp
cocopp.main('EXP_FOLDER')
```

in a Python/IPython shell or a Jupyter notebook. These commands take the experimental data from the `EXP_FOLDER` folder and produce figures, tables and html-pages by default in the `ppdata` folder.

Data from multiple experiments can be post-processed together by adding folder names to these calls where each folder represents a full experiment. Additionally, the COCO data archive allows to access online data from previously run experiments. This archive can be explored interactively as shown in Figure 4.

If a given name is not a unique data set substring, COCO provides the list of all matching data sets. In this case, the user has three choices. Either specify the name further until it is unique; or, add an exclamation mark, like with `MCS!`, to retrieve the first entry of the matching list by the `get_first` method; or, add a star (like `JADE*`)

[23] See also `example_experiment_for_beginners.py` which runs out-of-the-box as a benchmarking Python script just as `example_experiment2.py` which also allows to run the experiment in separate batches.

```
$ ### get and install the code
$ git clone https://github.com/numbbo/coco.git  # get coco using git
$ cd coco
$ python do.py run-python  # install Python experimental module cocoex
$ python do.py install-postprocessing install-user # install postprocessing :-)
```

```
$ ### (optional) run an example from the shell
$ mkdir my-first-experiment
$ cd my-first-experiment
$ cp ../code-experiments/build/python/example_experiment2.py .
$ python example_experiment2.py  # run the current "default" experiment
$                                # and the post-processing
$                                # and open browser when finished
```

```
#!/usr/bin/env python
"""Python script to benchmark fmin of scipy.optimize"""
from __future__ import division  # not needed in Python 3
import cocoex, cocopp  # experimentation and post-processing modules
import scipy.optimize # to define the solver to be benchmarked

### input
suite_name = "bbob"
output_folder = "scipy-optimize-fmin"
fmin = scipy.optimize.fmin
budget_multiplier = 2  # increase to 10, 100, ...

### prepare
suite = cocoex.Suite(suite_name, "", "")
observer = cocoex.Observer(suite_name, "result_folder: " + output_folder)

### go
for problem in suite:  # this loop will take several minutes or longer
    problem.observe_with(observer)  # will generate the data for cocopp
    # restart until the problem is solved or the budget is exhausted
    while (not problem.final_target_hit and
           problem.evaluations < problem.dimension * budget_multiplier):
        fmin(problem, problem.initial_solution_proposal())
        # we assume that 'fmin' evaluates the final/returned solution

### post-process data
cocopp.main(observer.result_folder)  # re-run folders look like "...-001" etc
```

Figure 2. Shell code for user installation of COCO (above) and for running a benchmarking experiment from a shell via Python (middle), and Python code to benchmark `scipy.optimize.fmin` on the `bbob` suite (below).

to retrieve all matching entries. Typically, only results from the same function suite can be processed together.[24]

In the system shell, the last call to the post-processing in Figure 4 transcribes to

```
python -m cocopp 2009/NEWUOA! BFGS-P-StPt NELDERDOERR MCS! NIPOPaCMA lmm RANDOMSEARCH!
```

This call processes the data sets of seven (more-or-less representative) solvers benchmarked on the 24 unconstrained continuous test functions of the `bbob` test suite:

- NEWUOA, the NEW Unconstrained Optimization Algorithm [59] with the recommended number of $2n + 1$ interpolation points in the quadratic model. We show the results for a Scilab implementation, as benchmarked in [62].
- The BFGS algorithm [18, 29, 32, 65], as implemented in the Python function

[24] Exceptions of *compatible* function suites exist—for example when they contain the same functions over different dimensions (which is the case for the `bbob` and `bbob-largescale` suites) or when one is a subset of the other (like for the `bbob-biobj` and `bbob-biobj-ext` suites).

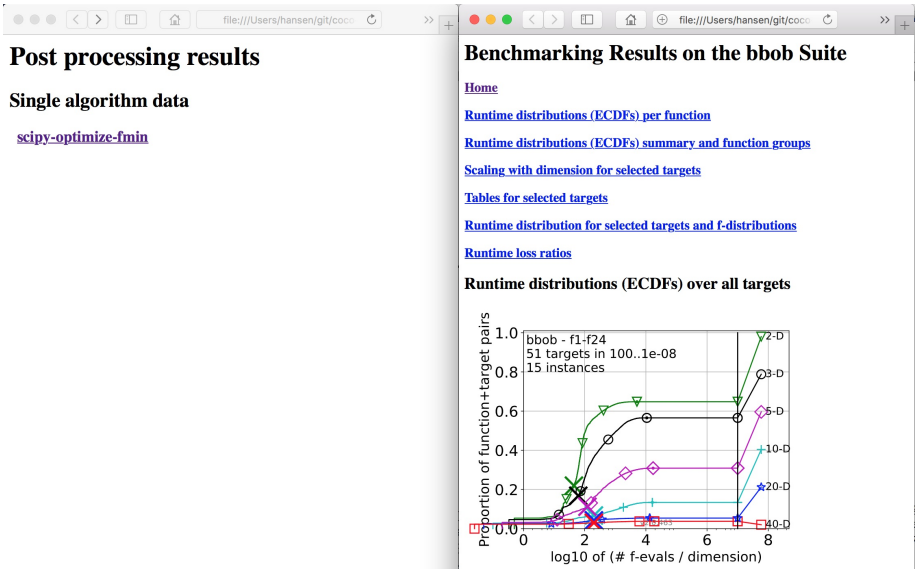


Figure 3. Benchmarking output in the browser. Left: homepage which opens when the post-processing has finished; right: page when clicking the scipy-optimize-fmin link on the left. The further links on the right open various types of graphs most of which are discussed below.

```
In [1]: import cocopp
In [2]: ars = cocopp.archives
In [3]: ars.all  # get the list of all available data sets
Out[3]:
['bbob/2009/ALPS_hornby_noiseless.tgz',
 'bbob/2009/AMALGAM_bosman_noiseless.tgz',
 [...]
 'bbob-noisy/2016/PSAaSmD-CMA-ES_Nishida_bbob-noisy.tgz',
 'test/N-II.tgz',
 'test/RS-4.zip']
In [4]: ars.bbob.find('bfgs')  # find all 'bbob' data sets
Out[4]:                        # with 'bfgs' in their name
['2009/BFGS_ros_noiseless.tgz',
 '2012/DE-BFGS_voglis_noiseless.tgz',
 '2012/PSO-BFGS_voglis_noiseless.tgz',
 '2014-others/BFGS-scipy-Baudis.tgz',
 '2014-others/L-BFGS-B-scipy-Baudis.tgz',
 '2018/BFGS-M-17.tgz',
 '2018/BFGS-P-09.tgz',
 '2018/BFGS-P-Instances.tgz',
 '2018/BFGS-P-range.tgz',
 '2018/BFGS-P-StPt.tgz',
 '2019/BFGS-scipy-2019_bbob_Varelas_Dahito.tgz',
 '2019/L-BFGS-B-scipy-2019_bbob_Varelas_Dahito.tgz']
In [5]: bfgs_data_link = ars.bbob.get_first('2018/BFGS-P-S')
  downloading [...]
In [6]: cocopp.main(bfgs_data_link)  # post-process data
Post-processing (1)
  Using:
    PATH-TO-USER-HOME/.cocopp/data-archive/bbob/2018/BFGS-P-StPt.tgz
  [...]
In [7]: cocopp.main("2009/NEWUOA! BFGS-P-StPt NELDERDOERR MCS! NIPOPaCMA "
                    "lmm RANDOMSEARCH!")  # post-process multiple data sets
Post-processing (2+)
  downloading http://coco.gforge.inria.fr/data-archive/...
  [...]
```

Figure 4. Example of searching in and post-processing of archived data.

`fmin_bfgs` from the `scipy.optimize` module with the origin as the initial point, named BFGS-P-StPt [12].
- The downhill simplex method by Nelder and Mead [56] with re-shaping and halfruns as presented in [22], denoted as NELDERDOERR.
- The multilevel coordinate search (MCS, [49]) by Huyer and Neumaier [50].
- Two variants of the Covariance Matrix Adaptation Evolution Strategy [40]: a version with increasing population size and negative recombination weights [52], entitled NIPOPaCMA, and a local meta-model assisted version benchmarked in [2], under the name lmm-CMA-ES.
- As a baseline, a simple random search (RANDOMSEARCH) with solutions sampled uniformly at random in the hyperbox $[-5, 5]^n$ [8].

The post-processing runs for a few minutes and produces an output folder `ppdata` where all visualizations, tables, HTML pages, etc. are written. Once finished, a web browser opens and displays the results.

Depending on the number of compared data sets (with cases $1, 2$, and >2), different visualizations will appear, as detailed in the following. Note that all plots, displayed here, have been made with COCO, version 2.3.2.

The main display in COCO are runtime distributions (ECDFs of number of function evaluations) to solve a given set of problems. Extending over so-called data profiles [55], COCO aggregates problems with different target precision values and displays the runtime distributions with simulated restarts. The default target precision values are 51 evenly log-spaced values between 10^{-8} and 10^2.

Figure 5 shows examples of runtime ECDFs from single functions, aggregated over function groups, and aggregated over all functions of a test suite. Plots of the first two rows were generated with the above `cocopp` calls. The last row shows results for five well-known multiobjective algorithms on the `bbob-biobj` suite, invoked by the command

```
python -m cocopp NSGA-II! DEMO RM-MEDA SMS-EMOA! bbob-biobj/2016/RANDOMSEARCH-5
```

The multiobjective algorithms are NSGA-II [3, 21], DEMO [61, 68], RM-MEDA [4, 74], SMS-EMOA-DE [6, 11], and a uniform random search within the hyperbox $[-5, 5]^n$ [5].

To demonstrate the influence of the target choice, Figure 6 shows runtime distributions for the same solvers as in Figure 5 for (i) the 51 default target values (first row) and (ii) for 31 budget-based target values, see Section 3.2, with budgets up to fifty times dimension evaluations. Budget-based targets are invoked by simply adding the optional argument "`--expensive`" to the `cocopp` call. Results are shown in dimension 20 on the function f_{10} (left column) and aggregated over all 24 `bbob` functions (right column). The absolute targets (first row) do not reveal the initially superior speed of MCS, NEWUOA, or Nelder-Mead. The budget-based targets (second row) do not reveal that the (disturbed) ellipsoid f_{10} is not even closely solved by MCS or Nelder-Mead.

When only a single solver is post-processed, COCO produces also single-target-precision runtime ECDFs (akin to data profiles) for four different target precision values, along with ECDFs of the precisions attained within a given budget, as shown in Figure 7. Obtained precision ECDFs extend data profile graphs naturally to the right (see also the caption).

Additionally, COCO generates scaling plots of the average runtime (ERT) over dimension. Figure 8 shows the scaling of BFGS-P-StPt for several target precisions (left plot) and the scaling of multiple solvers for a single target precision (right plot).

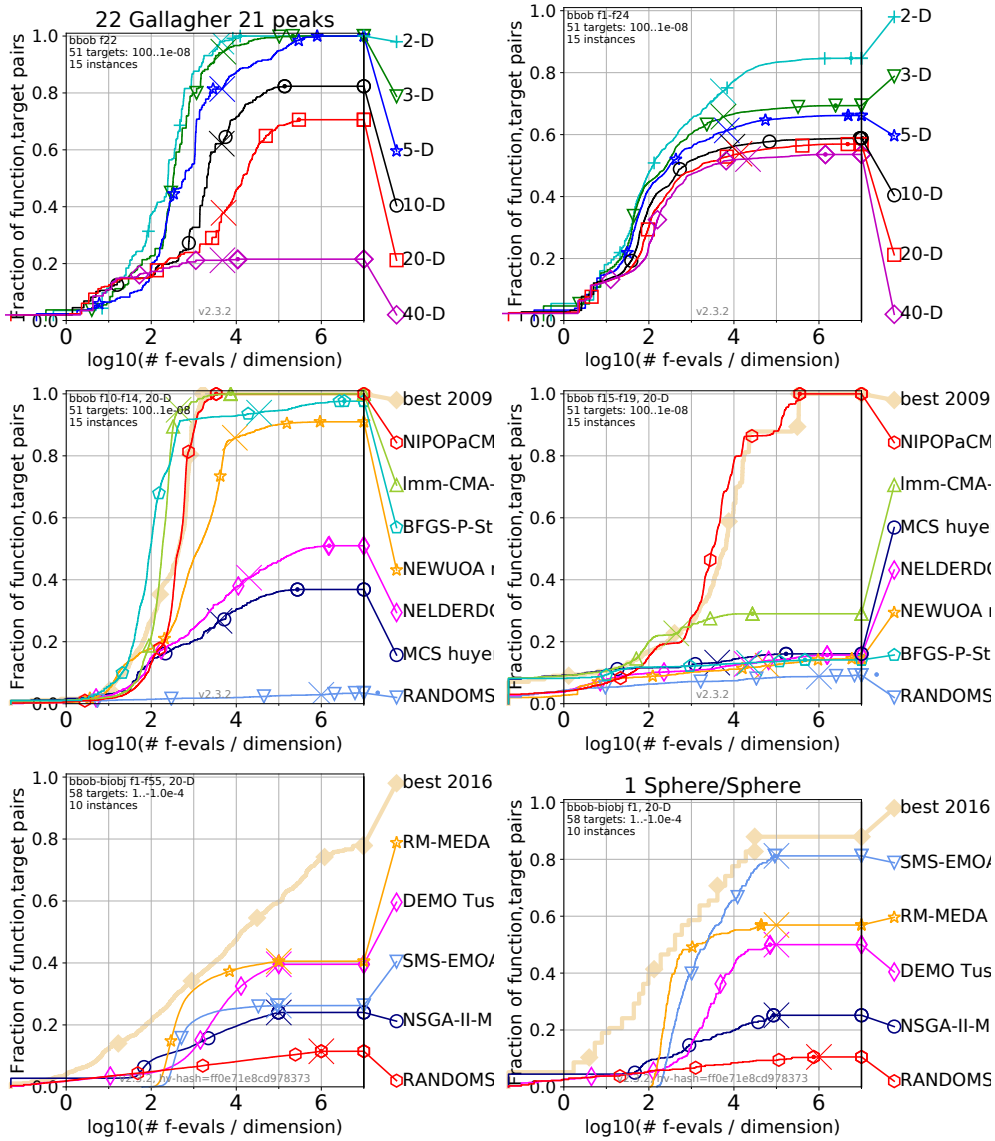


Figure 5. Examples of runtime ECDF plots in COCO. First row: plots for a single solver (here: BFGS-P-StPt) in dimensions between 2 and 40, to the left on the multi-modal function f_{22} and to the right aggregated over all 24 `bbob` functions. Second row: plots for seven solvers in dimension 20, aggregated over all problems from the highly ill-conditioned function group (left) and the multi-modal function group with global structure (right). Third row: plots for five multiobjective solvers in 20-D on the `bbob-biobj` test suite with data aggregated over all 55 functions (left) and for the double-sphere problem `bbob-biobj` F_1 (right). Long algorithm names are cut for readability.

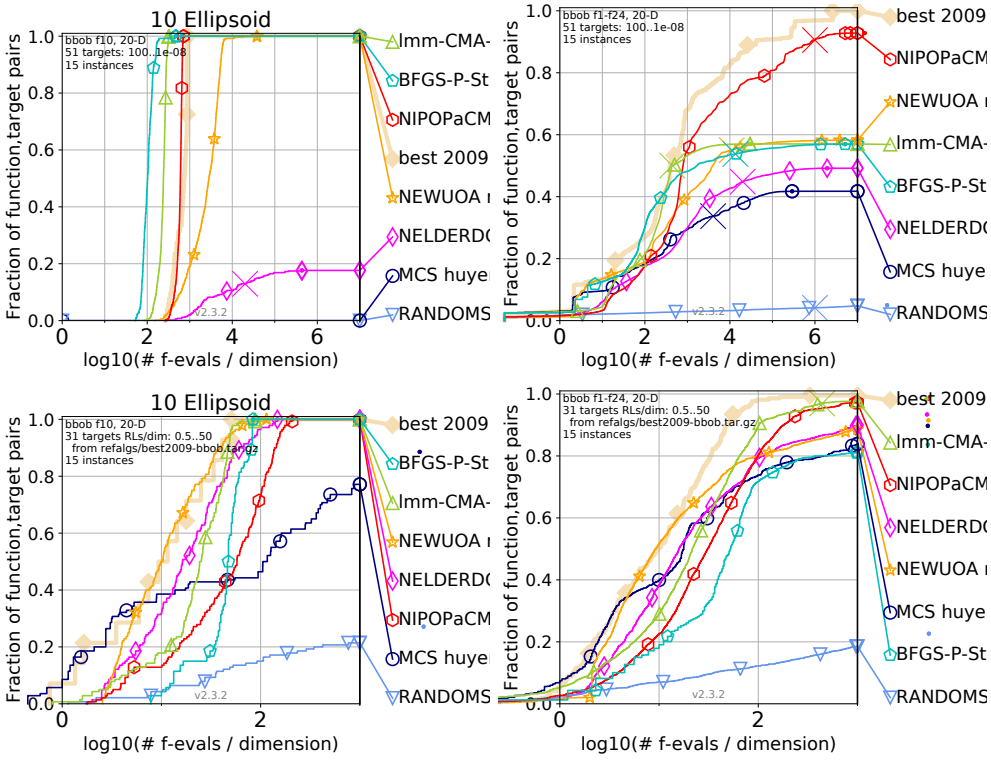


Figure 6. Examples of runtime ECDF plots in COCO and the impact of the target choice. First row: plots for seven solvers in dimension 20 on a single function (left) and aggregated over all 24 `bbob` functions (right) for the 51 default target precisions, equidistantly log-spaced between 100 and 10^{-8}. Second row: as first row, but using budget-based targets with the target-wise best result from the BBOB-2009 workshop as reference and with budgets between 0.5 and 50 times dimension function evaluations.

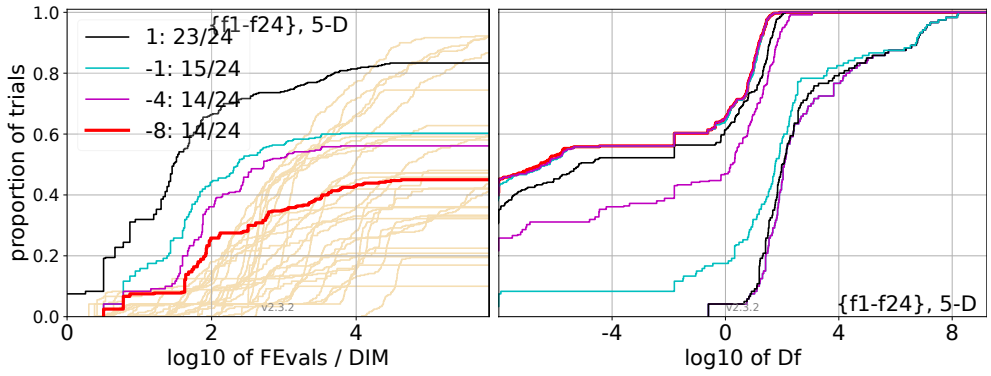


Figure 7. ECDFs of runtimes to reach a single target precision (left) and ECDFs of precisions reached for different budgets (right) for BFGS-P-StPt. Left: the target precisions for each graph are from left to right, respectively, 10^{1}, 10^{-1}, 10^{-4}, and 10^{-8}, and the legend provides the number of functions solved for each precision. Right: the budgets for each graph from right to left are, respectively, 0.5, 1.2, 3, 10, 100, 1000, and 10 000 times dimension and the maximal budget (thick red) indicating the final distribution of precisions values. The left most point of each precision graph on the right, representing precision 10^{-8}, coincides with the (thick red) runtime graph for precision 10^{-8} at the respective budgets and proportion of trials to the left. The same holds analogously for all shown target precisions.

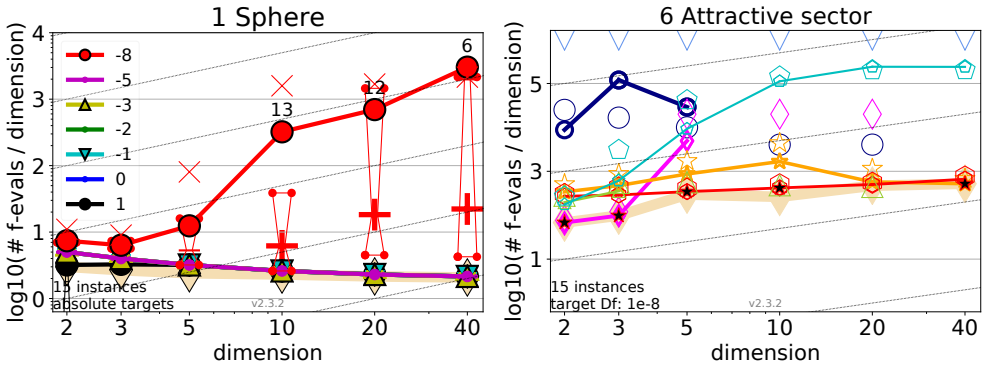
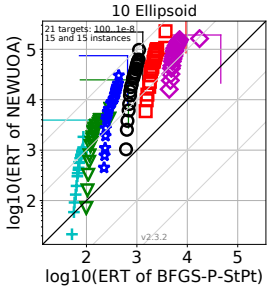


Figure 8. Scaling with the dimension of BFGS-P-StPt on the sphere function for different target precisions (left) and of different solvers on the attractive sector function for target precision 10^{-8} (right, the algorithms are distinguished by the same colors and markers as in Figure 6). Shown are average runtimes divided by the problem dimension (in log10-scale, only the exponent is annotated) to reach a given target, plotted against dimension. The corresponding target precisions are either mentioned in the legend with the number i indicating target precision 10^i (left plot) or at the bottom left of the figure as "target Df" (here 10^{-8} in the right plot).



Δf_{opt}	1e1	1e0	1e-1	1e-2	1e-3	1e-5	1e-7	#succ
f1, 5-D	11	12	12	12	12	12	12	15/15
MCS huyer noiseless	**1** (0.1)	1.9 (0.9)	2.5 (2)	2.6 (2)	2.6 (2)	2.6 (2)	2.6 (2)	15/15
NELDERDOERR doerr noiseless	1.5 (0.7)	3.5 (2)	5.6 (0.4)	7.6 (1.0)	10 (2)	14 (0.5)	18 (3)	15/15
NEWUOA ros noiseless	1.1 (0)	**1.0** (0)$^{\star 3}$	**1.0** (0)$^{\star 3}$	**1.0** (0)$^{\star 4}$	**1.0** (0.1)$^{\star 4}$	**1.0** (0)$^{\star 4}$	**1.0** (0.1)$^{\star 4}$	15/15
RANDOMSEARCH auger noiseless	7.5 (11)	1698 (1325)	6.9e5 (7e5)	∞	∞	∞	*∞5e6*	0/15
NIPOPaCMA loshchilov noiseless	2.7 (2)	7.5 (3)	14 (3)	20 (3)	26 (4)	37 (2)	51 (4)	15/15
lmm-CMA-ES auger noiseless	1.6 (0.9)	2.9 (0.4)	3.6 (0.2)	4.4 (0.6)	5.1 (0.5)	6.8 (0.6)	8.5 (0.6)	15/15
BFGS-P-StPt	1.5 (0)	1.3 (0)	1.3 (0)	1.3 (0)	1.3 (0)	1.3 (0)	1.4 (0.3)	15/15

Figure 9. Examples of the COCO scatter plots (left) and the platform's tabular output (right). The scatter plot shows, for 21 target precision values between 100 and 10^{-8} per dimension, the average runtime (ERT) in log10 of the number of function evaluations for the two solvers BFGS-P-St-Pt and NEWUOA. The thin colored axis-aligned line segments indicate the maximal budgets associated with the x- and y-axis. The table shows ERT ratios for the target precisions given in the first row. The ratios are computed by dividing ERT with the best ERT from 31 solvers from BBOB-2009 (given in the second line). The dispersion measure given in brackets is the semi-interdecile range (half the difference between the 10 and 90%-tile) of bootstrapped runtimes. If the last target was never reached, the median number of conducted function evaluations is given in italics. The last column (#succ) contains the number of trials that reached the (final) target $f_{opt} + 10^{-8}$. Entries succeeded by a star are statistically significantly better (according to the rank-sum test) when compared to all other solvers of the table, with $p = 0.05$ or $p = 10^{-k}$ when a number k follows the star, with Bonferroni correction by the number of functions (24). Best results are printed in bold.

These bring a direct visual aid to investigating how the performance of a solver scales with the problem dimension.

When only two data sets are compared, COCO also produces for each function a scatter plot of ERT values for 21 targets and all dimensions, as shown in Figure 9. Performance data are also available in a tabular format, an example thereof is shown on the right-hand side in Figure 9.

Besides the browser output, COCO also provides ACM-compliant LaTeX templates with already included main performance displays, which facilitates the publication of benchmarking results, see Figure 10. The showcased plots are non-exhaustive and COCO also provides more detailed descriptions.

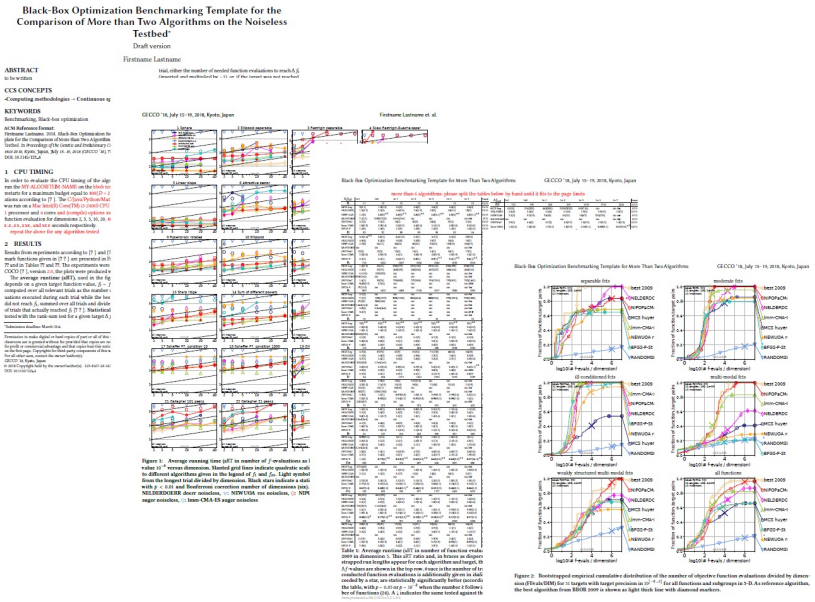


Figure 10. Compiled LaTeX-template for producing ACM-compliant papers from COCO data.

6. Usage Statistics

Since 2009, the COCO software framework has been used to facilitate submissions to the workshops on Black-Box Optimization Benchmarking (BBOB) at the ACM GECCO conference. With time, the platform has also been more and more used outside the scope of this workshop series. Table 1 summarizes some numbers of visibility and user base, including the number of citations to the documentation.

Table 1. Visibility of COCO. All citations as of November 19, 2019, in Google Scholar.

Data sets online	`bbob` suite	227
	`bbob-noisy` suite	45
	`bbob-biobj` suite	32
	`bbob-largescale` suite	11
	`bbob-mixint` suite	4
BBOB workshop papers using COCO		143
Unique authors on the workshop papers		109 from 28 countries
Papers in Google Scholar found with the search phrase *"comparing continuous optimizers" OR "black-box optimization benchmarking (BBOB)"*		559
Citations to the COCO documentation including [37] [38] [28] [39] [35] [36] [41] [34] [17] and [16]		1,455

7. Extensions under Development

The COCO software framework is under continuous development. In this section, we briefly discuss features which have been thoroughly specified for inclusion or are already under advanced development.

- Interface for the implementation of new suites under Python. In the current release, new suites can be implemented and (seamlessly) integrated only on the C level.[25] The new development shall allow to integrate new benchmark suites also written with Python, which will reduce the necessary development time considerably and make rapid prototyping of benchmark suites possible.
- Interface for enabling external evaluation of solutions based on socket communication. The interface where an external evaluator (the server) responds to requests for evaluations by COCO (the client) can be especially useful for supporting benchmarking on real-world problems that rely on particular software for solution evaluation.
- Benchmark suites for
  - constrained problems,
  - multiobjective problems with 3 objectives implemented in Python,
  - two types of real-world problems in games, the problem of forming a deck for the TopTrumps card game and the problem of generating a level for the Super Mario Bros. game [72] (both in single- and bi-objective variants).
- Use of recommendations, in order to address a (usually) small bias in the performance evaluation of noisy functions. Recommendations represent the current return value of the solver. The performance quality indicator is based on a short history of recommendations.
- Rewriting of the post-processing with interactive figures.

These features have not been released yet and their description and implementation may still undergo some relevant changes before their release.

8. Summary and Discussion

We have presented the open source zero-order black-box optimization benchmarking platform COCO that allows to benchmark numerical optimization algorithms automatically. The platform is composed of an interface to several languages (currently C/C++, Java, Matlab/Octave and Python) where the solvers can be plugged and run on a set of test functions. The ensuing collected data are then post-processed by a Python module and different graphs and tables are generated. All collected benchmarking data are open access, allowing to more easily reproduce and in particular seamlessly compare results. As of November 2019, more than 300 datasets are available. The platform also supports the implementation of new test suites.

The benchmarking methodology implemented within the platform is original in several key aspects:

- Each test function comes in several instances that typically differ by having different (pseudo-randomly sampled) optima, shifts of function value, and rotations. The underlying assumption when analysing the data is that different instances of the same test function have similar difficulties. This notion of function instances

[25] See https://github.com/numbbo/coco/blob/master/howtos/create-a-suite-howto.md.

allows to compare deterministic and stochastic solvers in a single framework and makes it harder for a solver to exploit specific function instance properties (like a specific position of the optimum).
- All test functions are scalable with respect to the input dimension.
- The `bbob` test suite for unconstrained optimization tries to reflect difficulties encountered in reality. A well-balanced set of difficulties and a scalable testbed is especially important when performance is aggregated, for example, through runtime ECDFs or data profiles.
- We never aggregate results over dimension because dimension is a *known* input to the solver which can and should be used when deciding which solver to apply on a (real-world) problem. We may aggregate, however, over many target values.
- The benchmarking methodology generalizes to multiobjective problems by using a quality indicator which maps all so-far evaluated solutions to a single value and defining targets for this indicator value. Currently, only benchmarking of bi-objective problems is supported.

As a final remark, we want to emphasize the importance of carefully scrutinizing the test functions used in aggregated results, for example, in empirical runtime distributions and data or performance profiles. Unbalanced test suites (for example, primarily low-dimensional, separable, ...) and the common approach to aggregate over all functions from a suite can lead to strong biases. This may disconnect solvers that perform well on test suites from those that perform well on real-world problems and hence seriously misguide research efforts in numerical optimization.

If we aim towards providing software that can address real-world difficulties, we should (i) include in our test suites mainly challenging, yet solvable problems and (ii) ensure that aggregated performance measures do not over-emphasize results on unimportant problem classes—like easy-to-solve problems.

Acknowledgments

The authors would like to thank Steffen Finck, Marc Schoenauer, Petr Pošik and Dejan Tušar for their many invaluable contributions to this work.

The authors also acknowledge support by the grant ANR-12-MONU-0009 (NumBBO) of the French National Research Agency. This work was furthermore supported by a public grant as part of the Investissement d'avenir project, reference ANR-11-LABX-0056-LMH, LabEx LMH, in a joint call with Gaspard Monge Program for optimization, operations research and their interactions with data sciences.

Tea Tušar acknowledges the financial support from the Slovenian Research Agency (research project No. Z2-8177 and research program No. P2-0209) and the European Commission's Horizon 2020 research and innovation program (grant agreement No. 692286).

References

[1] C. Audet and W. Hare, *Derivative-free and blackbox optimization*, Springer, 2017.

[2] A. Auger, D. Brockhoff, and N. Hansen, *Benchmarking the Local Metamodel CMA-ES on the Noiseless BBOB'2013 Test Bed*, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2013)*. ACM, 2013, pp. 1225–1232.

[3] A. Auger, D. Brockhoff, N. Hansen, D. Tušar, T. Tušar, and T. Wagner, *Benchmarking*

MATLAB's gamultiobj (NSGA-II) on the Bi-objective BBOB-2016 Test Suite, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2016)*. ACM, 2016, pp. 1233–1239.

[4] A. Auger, D. Brockhoff, N. Hansen, D. Tušar, T. Tušar, and T. Wagner, *Benchmarking RM-MEDA on the Bi-objective BBOB-2016 Test Suite*, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2016)*. ACM, 2016, pp. 1241–1247.

[5] A. Auger, D. Brockhoff, N. Hansen, D. Tušar, T. Tušar, and T. Wagner, *The Impact of Search Volume on the Performance of RANDOMSEARCH on the Bi-objective BBOB-2016 Test Suite*, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2016)*. ACM, 2016, pp. 1257–1264.

[6] A. Auger, D. Brockhoff, N. Hansen, D. Tušar, T. Tušar, and T. Wagner, *The Impact of Variation Operators on the Performance of SMS-EMOA on the Bi-objective BBOB-2016 Test Suite*, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2016)*. ACM, 2016, pp. 1225–1232.

[7] A. Auger and N. Hansen, *Performance Evaluation of an Advanced Local Search Evolutionary Algorithm*, in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2005)*. 2005, pp. 1777–1784.

[8] A. Auger and R. Ros, *Benchmarking the pure random search on the BBOB-2009 testbed*, in Rothlauf [63], 2009, pp. 2479–2484.

[9] R.S. Barr, B.L. Golden, J.P. Kelly, M.G. Resende, and W.R. Stewart, *Designing and reporting on computational experiments with heuristic methods*, Journal of heuristics 1 (1995), pp. 9–32.

[10] V. Beiranvand, W. Hare, and Y. Lucet, *Best practices for comparing optimization algorithms*, Optimization and Engineering 18 (2017), pp. 815–848.

[11] N. Beume, B. Naujoks, and M. Emmerich, *SMS-EMOA: Multiobjective Selection Based on Dominated Hypervolume*, European Journal of Operational Research 181 (2007), pp. 1653–1669.

[12] A. Blelly, M. Felipe-Gomes, A. Auger, and D. Brockhoff, *Stopping Criteria, Initialization, and Implementations of BFGS and their Effect on the BBOB Test Suite*, in *GECCO (Companion) workshop on Black-Box Optimization Benchmarking (BBOB'2009)*. 2018.

[13] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, *PISA—A Platform and Programming Language Independent Interface for Search Algorithms*, in *Conference on Evolutionary Multi-Criterion Optimization (EMO 2003)*, C.M. Fonseca, *et al.*, eds., LNCS Vol. 2632, Berlin. Springer, 2003, pp. 494–508.

[14] J. Bossek, *Performance assessment of multi-objective evolutionary algorithms with the R package ecr*, in *Companion Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018)*. ACM, 2018, pp. 1350–1356.

[15] D. Brockhoff, T.D. Tran, and N. Hansen, *Benchmarking Numerical Multiobjective Optimizers Revisited*, in *Genetic and Evolutionary Computation Conference (GECCO 2015)*. ACM, 2015, pp. 639–646.

[16] D. Brockhoff, T. Tušar, A. Auger, and N. Hansen, *Using well-understood single-objective functions in multiobjective black-box optimization test suites*, ArXiv e-prints arXiv:1604.00359v3 (2019). Last updated January 4, 2019.

[17] D. Brockhoff, T. Tušar, D. Tušar, T. Wagner, N. Hansen, and A. Auger, *Biobjective performance assessment with the COCO platform*, ArXiv e-prints arXiv:1605.01746 (2016).

[18] C.G. Broyden, *The convergence of a class of double-rank minimization algorithms*, Journal of the Institute of Mathematics and Its Applications 6 (1970), pp. 76–90.

[19] S. Bubeck, *Convex optimization: Algorithms and complexity* (2014).

[20] M.R. Bussieck, S.P. Dirkse, and S. Vigerske, *Paver 2.0: an open source environment for automated performance analysis of benchmarking data*, Journal of Global Optimization 59 (2014), pp. 259–275.

[21] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, *A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation 6 (2002),

pp. 182–197.
[22] B. Doerr, M. Fouz, M. Schmidt, and M. Wahlström, *BBOB: Nelder-Mead with resize and halfruns*, in Rothlauf [63], 2009, pp. 2239–2246.
[23] C. Doerr, H. Wang, F. Ye, S. van Rijn, and T. Bäck, *IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics*, ArXiv e-prints arXiv:1810.05281 (2018).
[24] E.D. Dolan and J.J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical programming 91 (2002), pp. 201–213.
[25] J.J. Durillo and A.J. Nebro, *jMetal: a Java Framework for Multi-Objective Optimization*, Advances in Engineering Software 42 (2011), pp. 760–771.
[26] B. Efron and R.J. Tibshirani, *An introduction to the bootstrap*, CRC press, 1994.
[27] T.A. El-Mihoub, A.A. Hopgood, L. Nolle, and A. Battersby, *Hybrid genetic algorithms: A review.*, Engineering Letters 13 (2006), pp. 124–137.
[28] S. Finck, N. Hansen, R. Ros, and A. Auger, *Real-parameter black-box optimization benchmarking 2009: Presentation of the noiseless functions*, Tech. Rep. 2009/20, Research Center PPE, 2009.
[29] R. Fletcher, *A new approach to variable metric algorithms*, Computer journal 13 (1970), pp. 317–322.
[30] M. Gaviano, D. Kvasov, D. Lera, and Y.D. Sergeyev, *Software for generation of classes of test functions with known local and global minima for global optimization*, ACM Transactions on Mathematical Software (2003), pp. 469–480.
[31] A. Georges, A. Gleixner, G. Gojic, R.L. Gottwald, D. Haley, G. Hendel, and B. Matejczyk, *Feature-based algorithm selection for mixed integer programming*, Tech. Rep. 18-17, ZIB, Takustr. 7, 14195 Berlin, 2018.
[32] D. Goldfarb, *A family of variable metric updates derived by variational means*, Mathematics of Computation 24 (1970), pp. 23–26.
[33] N. Gould and J. Scott, *A note on performance profiles for benchmarking software*, ACM Transactions on Mathematical Software (TOMS) 43 (2016).
[34] N. Hansen, A. Auger, D. Brockhoff, D. Tušar, and T. Tušar, *COCO: Performance assessment*, ArXiv e-prints arXiv:1605.03560 (2016).
[35] N. Hansen, A. Auger, S. Finck, and R. Ros, *Real-parameter black-box optimization benchmarking 2009: Experimental setup*, Tech. Rep. RR-6828, INRIA, 2009. Available at http://hal.inria.fr/inria-00362649/en/.
[36] N. Hansen, A. Auger, S. Finck, and R. Ros, *Real-parameter black-box optimization benchmarking 2010: Experimental setup*, Tech. Rep. RR-7215, INRIA, 2010. Available at http://coco.gforge.inria.fr/bbob2010-downloads.
[37] N. Hansen, A. Auger, O. Mersmann, T. Tušar, and D. Brockhoff, *COCO: A platform for comparing continuous optimizers in a black-box setting*, ArXiv e-prints arXiv:1603.08785 (2016).
[38] N. Hansen, S. Finck, R. Ros, and A. Auger, *Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions*, Tech. Rep. RR-6829, INRIA, 2009. Available at http://hal.inria.fr/inria-00362633/en/.
[39] N. Hansen, S. Finck, R. Ros, and A. Auger, *Real-parameter black-box optimization benchmarking 2009: Noisy functions definitions*, Tech. Rep. RR-6869, INRIA, 2009. Available at http://hal.inria.fr/inria-00369466/en.
[40] N. Hansen and A. Ostermeier, *Completely Derandomized Self-Adaptation in Evolution Strategies*, Evolutionary Computation 9 (2001), pp. 159–195.
[41] N. Hansen, T. Tušar, O. Mersmann, A. Auger, and D. Brockhoff, *COCO: The experimental procedure*, ArXiv e-prints arXiv:1603.08776 (2016).
[42] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík, *Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009*, in *GECCO '10: Proceedings of the 12th annual conference comp on Genetic and evolutionary computation*, New York, NY, USA. ACM, 2010, pp. 1689–1696.
[43] G. Harik and F. Lobo, *A parameter-less genetic algorithm*, in *Genetic and Evolutionary Computation Conference (GECCO 1999)*, Vol. 1. ACM, 1999, pp. 258–265.

[44] W. Hock and K. Schittkowski, *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems Vol. 187, Springer, 1981.
[45] A. Hoffman, M. Mannos, D. Sokolowsky, and N. Wiegmann, *Computational experience in solving linear programs*, Journal of the Society for Industrial and Applied Mathematics 1 (1953), pp. 17–33.
[46] J.N. Hooker, *Testing heuristics: We have it all wrong*, Journal of heuristics 1 (1995), pp. 33–42.
[47] H. Hoos and T. Stützle, *Evaluating Las Vegas Algorithms—Pitfalls and Remedies*, in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*. 1998, pp. 238–245.
[48] J.D. Hunter, *Matplotlib: A 2d graphics environment*, Computing in science & engineering 9 (2007), p. 90.
[49] W. Huyer and A. Neumaier, *Global optimization by multilevel coordinate search*, J. of Global Optimization 14 (1999), pp. 331–355.
[50] W. Huyer and A. Neumaier, *Benchmarking of MCS on the noiseless function testbed*, Manuscript available at http://www.mat.univie.ac.at/~neum/papers.html (2009).
[51] D.S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges 59 (2002), pp. 215–250.
[52] I. Loshchilov, M. Schoenauer, and M. Sebag, *Black-Box Optimization Benchmarking of NIPOP-aCMA-ES and NBIPOP-aCMA-ES on the BBOB-2012 Noiseless Testbed*, in *Companion Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2012)*, T. Soule, ed. ACM, 2012, pp. 269–276.
[53] O. Mersmann, M. Preuss, H. Trautmann, B. Bischl, and C. Weihs, *Analyzing the bbob results by means of benchmarking concepts*, Evolutionary computation 23 (2015), pp. 161–185.
[54] J.J. Moré, B.S. Garbow, and K.E. Hillstrom, *Testing unconstrained optimization software*, ACM Transactions on Mathematical Software (TOMS) 7 (1981), pp. 17–41.
[55] J.J. Moré and S.M. Wild, *Benchmarking derivative-free optimization algorithms*, SIAM Journal on Optimization 20 (2009), pp. 172–191.
[56] J. Nelder and R. Mead, *The downhill simplex method*, Computer Journal 7 (1965), pp. 308–313.
[57] A. Nemirovski, *Information-based complexity of convex programming*, Lecture Notes (1995).
[58] Y. Nesterov, *Lectures on convex optimization*, Vol. 137, Springer, 2018.
[59] M.J.D. Powell, *The NEWUOA software for unconstrained optimization without derivatives*, Large Scale Nonlinear Optimization (2006), pp. 255–297.
[60] K. Price, *Differential evolution vs. the functions of the second ICEO*, in *Proceedings of the IEEE International Congress on Evolutionary Computation*, Piscataway, NJ, USA. IEEE, 1997, pp. 153–157.
[61] T. Robič and B. Filipič, *Differential evolution for multiobjective optimization*, in *Evolutionary Multi-Criterion Optimization (EMO 2005)*. Springer, 2005, pp. 520–533.
[62] R. Ros, *Benchmarking the NEWUOA on the BBOB-2009 function testbed*, in Rothlauf [63], 2009, pp. 2421–2428.
[63] F. Rothlauf (ed.), *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009, Companion Material*, ACM, 2009.
[64] K. Schittkowski, *More test examples for nonlinear programming codes*, Lecture Nots in Economics and Mathematical Systems Vol. 282, Springer, 1987.
[65] D.F. Shanno, *Conditioning of quasi-newton methods for function minimization*, Mathematics of Computation 24 (1970), pp. 647–656.
[66] S.S. Stevens, *et al.*, *On the theory of scales of measurement*, Science (1946), pp. 677–680.
[67] Y. Tian, R. Cheng, X. Zhang, and Y. Jin, *Platemo: A matlab platform for evolutionary multi-objective optimization [educational forum]*, IEEE Computational Intelligence

Magazine 12 (2017), pp. 73–87.
[68] T. Tušar and B. Filipič, *Performance of the DEMO Algorithm on the Bi-objective BBOB Test Suite*, in *Companion Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2016)*. ACM, 2016, pp. 1249–1256.
[69] T. Tušar, D. Brockhoff, and N. Hansen, *Mixed-integer benchmark problems for single- and bi-objective optimization*, in *Genetic and Evolutionary Computation Conference (GECCO 2019)*. ACM, 2019, pp. 718–726.
[70] T. Tušar, N. Hansen, and D. Brockhoff, *Anytime Benchmarking of Budget-Dependent Algorithms with the COCO Platform*, in *International Multiconference Information Society (IS 2017)*. ACM, 2017, pp. 47–50.
[71] K. Varelas, A. Auger, D. Brockhoff, N. Hansen, O.A. ElHara, Y. Semet, R. Kassab, and F. Barbaresco, *A comparative study of large-scale variants of CMA-ES*, in *International Conference on Parallel Problem Solving from Nature*. Springer, 2018, pp. 3–15.
[72] V. Volz, B. Naujoks, P. Kerschke, and T. Tušar, *Single- and multi-objective game-benchmark for evolutionary algorithms*, in *Genetic and Evolutionary Computation Conference (GECCO 2019)*. ACM, 2019, pp. 647–655.
[73] D. Whitley, S. Rana, J. Dzubera, and K.E. Mathias, *Evaluating evolutionary algorithms*, Artificial intelligence 85 (1996), pp. 245–276.
[74] Q. Zhang, A. Zhou, and Y. Jin, *RM-MEDA: A regularity model-based multiobjective estimation of distribution algorithm*, IEEE Transactions on Evolutionary Computation 12 (2008), pp. 41–63.
[75] E. Zitzler and L. Thiele, *Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study*, in *Conference on Parallel Problem Solving from Nature (PPSN V)*, LNCS Vol. 1498, Amsterdam. 1998, pp. 292–301.

Appendix A. How We Chose Test Functions

Our first and main test suite, `bbob`, took only known and relatively simple test functions as a starting point. For choosing any specific test function to begin with, the function should model either

- a difficulty of continuous domain optimization known to be important in practice, as for example multimodality or ill-conditioning or a ridge-like topology, or
- a comprehensible difficulty that is likely to be relevant in practice at least sometimes, as for example ruggedness, or
- a rather simple topology that every search algorithm should be able to deal with, like for example a linear slope.

We also wanted the functions to be comprehensible, in order to facilitate interpretation, and we required them to be scalable with the dimension. These functions were then (slightly) modified, mainly to make them less amenable to simple exploits. We also paired up functions to understand the effect of a particular change on the algorithm performance. Furthermore, we chose to have only a two dozen of functions in order to be able to run repeated experiments over a range of dimensions in reasonable time and to incentivize manual checking of the results on each and every function.

Appendix B. COCO Versus Competitive Testing

The main motivation behind our benchmarking effort is to be able to generate and assess a comprehensive profile of the performance of solvers and understand why they

perform well on some functions and not so well on others. In order to understand the behaviour of a solver, it is of vital importance to understand the underlying function it has been run on as well as possible. Therefore, functions can not be presented as black boxes to the scientific community and should ideally be fully comprehensible. Additionally, if we want to be able to compare results with previously collected performance data, the functions can not substantially change over time.

The incentives for a competition are, however, different. The main goal in a competition is to perform well rather than to understand algorithm behaviour. Hence, the competition designer should, in particular, take precautions to prevent exploits and overtuning. The most effective way to prevent this is to present the functions as black boxes not only to the solver but also to the scientific community and to change them frequently.

We have taken precautions in the function definitions and in the experimental setup such that unintended exploits of trivial function properties are unlikely. For example, the function optima as well as the optimal function values are not easily accessible in the API.

However, intentional exploitation and thereby neglect of the prescribed experimental setup is not prevented this way.

Appendix C. Details on the Used Biobjective Performance Measure

When benchmarking multiobjective algorithms, we must choose a quality measure to compare algorithms. In COCO, this quality measure is based on the well-known hypervolume indicator, I_{HV}, also known as the $\mathcal{S}$-metric, and introduced as "the size of the space covered" in [75]. In the following, we assume a generic search space Ω, two objective functions, and that the ideal point

$$p^{\mathrm{ideal}} = \left(\min_{x \in \Omega} f_1(x), \min_{x \in \Omega} f_2(x) \right)$$

and the nadir point

$$p^{\mathrm{nadir}} = \left(\max_{x^* \text{ Pareto-optimal}} f_1(x^*), \max_{x^* \text{ Pareto-optimal}} f_2(x^*) \right)$$

are known, which is the case for all current biobjective test suites in COCO. In order to be able to compare indicator values over different functions, dimensions, and instances[26], the objective space is first normalized using the following transformation:

$$f_i^{\mathrm{N}} = \frac{f_i - p_i^{\mathrm{ideal}}}{p_i^{\mathrm{nadir}} - p_i^{\mathrm{ideal}}} \tag{C1}$$

for $i = 1, 2$. In the normalized objective space, the ideal and nadir points correspond

[26]The algorithms themselves see the raw objective values which can span several orders of magnitudes, but they have access to the nadir point as the upper bound of the region of interest in objective space.

to $(0,0)$ and $(1,1)$, respectively.

The quality indicator $I : 2^{\Omega} \to \mathbb{R}$ used in COCO to measure the quality of a biobjective algorithm $\mathcal{A}$ after t function evaluations depends on the set $S \subseteq \Omega$ of all non-dominated solutions found by $\mathcal{A}$ within the first t function evaluations and on whether the nadir point has been dominated in the first t function evaluations (see also Equation (C2)):

- If the nadir point is dominated by an objective vector $f(s)$ with $s \in S$, the quality indicator $I(S)$ equals the hypervolume indicator $I_{\mathrm{HV}}(S, r)$ of S on the normalized objective space with $(1,1)$ as the reference point r.
- If the nadir point has not been dominated in the first t evaluations, the quality indicator $I(S)$ is the smallest distance of a normalized objective vector $f^{\mathrm{N}}(s)$ with $s \in S$ to the normalized objective space that dominates the nadir point, namely $[0,1]^2$, multiplied by -1 to allow for maximization of the quality indicator.

Note that both parts of the quality indicator align with a zero value at the border between the (objective) space dominating the reference point and the (objective) space not dominating it. The quality is positive in the former and negative in the latter case. More formally:

$$I(S) = \begin{cases} I_{\mathrm{HV}}(S, (1,1)) & \text{if } \exists s \in S : f_1(s) \leq p_1^{\mathrm{nadir}}, f_2(s) \leq p_2^{\mathrm{nadir}} \\ -\min\limits_{\substack{s \in S \\ z \in [0,1]^2}} \mathrm{dist}(f^{\mathrm{N}}(s), z) & \text{otherwise} \end{cases} . \quad \text{(C2)}$$

Finally, we record the number of function evaluations to reach certain target indicator values

$$I_{\mathrm{HV}}(S^*, (1,1)) - \varepsilon,$$

where $I_{\mathrm{HV}}(S^*, (1,1)) \in [0,1]$ is a reference hypervolume value, obtained as the hypervolume of the best known Pareto set approximation S^* in the normalized objective space, and ε is a target precision such as 10^{-5}. Because S^* is only an estimation of the true Pareto set, we also record runtimes for $\varepsilon = 0$ and for a few negative ε values.

For a more detailed description and further illustrations, we refer the interested reader to [17].