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

Nikolaus Hansen, Anne Auger, Olaf Mersmann, Tea Tusar, Dimo Brockhoff

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

Nikolaus Hansen[1,2], Anne Auger[1,2], Olaf Mersmann[3], Tea Tusar[4], Dimo Brockhoff[4]
[1]Inria, research centre Saclay, France
[2]Université Paris-Saclay, LRI, France
[3]TU Dortmund University, Chair of Computational Statistics, Germany
[4]Inria, research centre Lille, France

March 27, 2016

Abstract

COCO is a platform for Comparing Continuous Optimizers in a black-box setting. It aims at automatizing the tedious and repetitive task of benchmarking numerical optimization algorithms to the greatest possible extent. We present the rationals behind the development of the platform as a general proposition for a guideline towards better benchmarking. We detail underlying fundamental concepts of COCO such as its definition of a problem, the idea of instances, the relevance of target values, and runtime as central performance measure. Finally, we give a quick overview of the basic code structure and the available test suites.

Contents

1 Introduction

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

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

such that for the l constraints

$$g : X \subset \mathbb{R}^n \to \mathbb{R}^l \qquad l \geq 0$$

we have $g_i(\boldsymbol{x}) \leq 0$ for all $i = 1 \ldots l$. 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 $f(\boldsymbol{x}) \in \mathbb{R}^m$ that satisfy all above constraints g. We consider *time* to be defined as the number of calls to the function f.

A continuous optimization algorithm, also known as *solver*, addresses the above problem. Here, we assume that X is known, but no prior knowledge about f or g is available to the algorithm. That is, f and g are considered as a black-box which the algorithm can query with solutions $\boldsymbol{x} \in \mathbb{R}^n$ to get the respective values $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$.

From these prerequisits, benchmarking optimization algorithms seems to be a rather simple and straightforward task. We run an algorithm on a collection of problems and display the results. However, under closer inspection, benchmarking turns out to be surprisingly tedious, and it appears to be difficult to get results that can be meaningfully interpreted beyond the standard claim that one algorithm is better than another on some problems. [1] Here, we offer a conceptual guideline for benchmarking continuous optimization algorithms which tries to address this challenge and has been implemented within the COCO framework. [2]

The COCO framework provides the practical means for an automatized benchmarking procedure. Installing COCO (in a shell) and benchmarking an optimization algorithm, say, implemented in the function `fmin` in Python, becomes as simple as in Figure 1. The COCO framework provides currently

- an interface to several languages in which the benchmarked optimizer can be written, currently C/C++, Java, Matlab/Octave, Python
- several benchmark suites or testbeds, currently all written in C
- data logging facilities via the `Observer`
- data post-processing in Python and data display facilities in `html`
- article LaTeX templates

The underlying philosophy of COCO is to provide everything which otherwise most experimenters needed to setup and implement themselves, if they wanted to benchmark an algorithm properly.

[1] One major flaw is that we often get no indication of *how much* better an algorithm is. That is, the results of benchmarking often provide no indication of *relevance*; the main output often consists of hundreds of tabulated numbers only interpretable on an *ordinal scale [STE1946]*. Addressing a point of a common confusion, *statistical significance* is only a secondary, and by no means a *sufficient* condition for *relevance*.

[2] See https://www.github.com/numbbo/coco or https://numbbo.github.io for implementation details.

```
```

```
#!/usr/bin/env python
import cocoex
import cocopp  # or: import bbob_pproc as cocopp
from myoptimizer import

      =        .      "bbob"  "year: 2016"  ""
         =        .         "bbob"  "result_folder: myoptimizer-on-bbob"

for   in          # loop over all problems
            .            # prepare logging of necessary data
             .

      .     'exdata/myoptimizer-on-bbob'   # invoke data post-processing
```

Fig. 1: Shell code for installation of COCO (above), and Python code to benchmark `fmin` on the `bbob` suite and display the results. Now the file `ppdata/ppdata.html` can be used to browse the resulting data.

So far, the framework has been used successfully for benchmarking far over a hundred algorithms by many researchers.

1.1 Why COCO?

Appart from diminishing the burden (time) and the pitfalls (and bugs or omissions) of the repetitive coding task by many experimenters, our aim is to provide a *conceptual guideline for better benchmarking*. Our guideline has the following defining features.

1. **Benchmark functions are**
   (a) used as black boxes for the algorithm, 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", that is, they do not have artificial regularities that can be (intentionally or unintentionally) exploited by an algorithm. [3]
   (d) scalable with the input dimension *[WHI1996]*.

[3] For example, the optimum is not in all-zeros, optima are not placed on a regular grid, most functions are not separable *[WHI1996]*. 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.

2. There is no predefined budget (number of f-evaluations) for running an experiment, the experimental procedure is *budget-free [HAN2016ex]*.

3. A single performance measure is used — and thereafter aggregated and displayed in several ways — namely **runtime**, *measured in number of f-evaluations [BBO2016perf]*. Runtime has the advantages to

   - be independent of the computational platform, language, compiler, coding styles, and other specific experimental conditions [4]
   - be relevant, meaningful and easily interpretable without expert domain knowledge
   - be quantitative on the ratio scale *[STE1946]* [5]
   - assume a wide range of values
   - aggregate over a collection of values in a meaningful way

   A *missing* runtime value is considered as possible outcome (see below).

4. The display is as comprehensible, intuitive and informative as possible, We believe that details matter. Aggregation over dimension is avoided, because dimension is an a priori known parameter that can and should be used for algorithm design or selection decisions.

1.2 Terminology

We specify a few terms which are used later.

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

problem We talk about a *problem*, `coco_problem_t`, as a specific *function instance* on which an optimization algorithm is run. A problem can be evaluated and returns an f-value or -vector and, in case, a g-vector. In the context of performance assessment, a target f- or indicator-value is added to define a problem.

runtime We define *runtime*, or *run-length [HOO1998]* as the *number of evaluations* conducted on a given problem until a prescribed target value is hit, also referred to as number of *function* evaluations or f-evaluations. Runtime is our central performance measure.

suite A test- or benchmark-suite is a collection of problems, typically between twenty and a hundred, where the number of objectives m is fixed.

[4] Runtimes measured in f-evaluations are widely comparable and designed to stay. The experimental procedure *[HAN2016ex]* includes however a timing experiment which records the internal computational effort of the algorithm in CPU or wall clock time.

[5] As opposed to a ranking of algorithm based on their solution quality achieved after a given budget.

2 Functions, Instances, Problems, and Targets

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, [6] that is, for a given m we 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. By fixing n and j for function f_i, we define an optimization **problem** $(n, i, j) \equiv (f_i, n, j)$ that can be presented to the optimization algorithm. Each problem receives again an index in the suite, mapping the triple (n, i, j) to a single number.

As the formalization above suggests, the differentiation between function (index) and instance index is of purely semantic nature. This semantics however is important in how we display and interpret the results. We interpret **varying the instance** parameter as a natural randomization for experiments [7] in order to

- generate repetitions on a function and
- **average away irrelevant aspects of a function thereby providing**
  - generality which alleviates the problem of overfitting, and
  - a fair setup which prevents intentional or unintentional exploitation of irrelevant or artificial function properties.

For example, we consider the absolute location of the optimum not a defining function feature. Consequently, in a typical COCO benchmark suite, instances with randomized search space translations are presented to the optimizer. [8]

3 Runtime and Target Values

In order to measure the runtime of an algorithm on a problem, we establish a hitting time condition. We prescribe a **target value**, t, which is an f- or indicator-value *[TUS2016]*. For a single run, when an algorithm reaches or surpasses the target value t on problem (f_i, n, j), we say it has *solved the problem* (f_i, n, j, t) — it was successful. [9]

[6] We can think of j as a continuous parameter vector, as it parametrizes, among others things, translations and rotations. In practice, j is a discrete identifier for single instantiations of these parameters.

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

[8] Conducting either several trials on instances with randomized search space translations or with a randomized initial solution is equivalent, given that the optimizer behaves translation invariant (disregarding domain boundaries).

[9] Note the use of the term *problem* in two meanings: as the problem the algorithm is benchmarked on, (f_i, n, j), and as the problem, (f_i, n, j, t), an algorithm can solve by hitting the target t with the runtime, $\mathrm{RT}(f_i, n, j, t)$, or may fail to solve. Each problem (f_i, n, j) gives raise to a collection of dependent problems (f_i, n, j, t). Viewed as random variables, the events $\mathrm{RT}(f_i, n, j, t)$ given (f_i, n, j) are not independent events for different values of t.

Now, 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) (but see also Recommendations in *[HAN2016ex]*). [10] *Measured runtimes are the only way of how we assess the performance of an algorithm.* [11]

If an algorithm does not hit the target in a single run, the runtime remains undefined — while it has been bound to be at least $k + 1$, where k is the number of evaluations in this unsuccessful run. The number of defined runtime values depends on the budget the algorithm has explored. Therefore, larger budgets are preferable — however they should not come at the expense of abandoning reasonable termination conditions. Instead, restarts should be done.

3.1 Restarts and Simulated Restarts

An optimization algorithm is bound to terminate and, in the single-objective case, return a recommended solution, $\boldsymbol{x}$, for the problem, (f_i, n, j). It solves thereby all problems (f_i, n, j, t) for which $f(\boldsymbol{x}) \leq t$. Independent restarts from different, randomized initial solutions are a simple but powerful tool to increase the number of solved problems *[HAR1999]* — namely by increasing the number of t-values, for which the problem (f_i, n, j) was solved. [12] Independent restarts tend to increase the success rate, but they generally do not *change* the performance *assessment*, because the successes materialize at greater runtimes. Therefore, we call our approach *budget-free*. Restarts however "*improve the reliability, comparability, precision, and "visibility" of the measured results*" *[HAN2016ex]*.

Simulated restarts [HAN2010ex] [HAN2010] [BBO2016perf] are used to determine a runtime for unsuccessful runs. Semantically, this is only valid if we interpret different instances as random repetitions. Resembling the bootstrapping method *[EFR1993]*, when we face an unsolved problem, we draw uniformly at random a new j until we find an instance such that (f_i, n, j, t) was solved. [13] The evaluations done on the first unsolved problem and on all subsequently drawn unsolved problems are added to the runtime on the last problem and are considered as runtime on the original unsolved problem. This method is applied if a problem instance was not solved and is (only) available if at least one problem instance was solved.

3.2 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.

[10] Target values are directly linked to a problem, leaving the burden to properly define the targets with the designer of the benchmark suite. The alternative is to present final f- or indicator-values as results, leaving the (rather unsurmountable) burden to interpret these values to the reader. Fortunately, there is an automatized generic way to generate target values from observed runtimes, the so-called run-length based target values *[BBO2016perf]*.

[11] Observed success rates can (and should) be translated into lower bounds on runtimes on a subset of problems.

[12] For a given problem (f_i, n, j), the number of acquired runtime values, $\mathrm{RT}(f_i, n, j, t)$ is monotonously increasing with the budget used. Considered as random variables, these runtimes are not independent.

[13] More specifically, we consider the problems $(f_i, n, j, t(j))$ for all benchmarked instances j. The targets $t(j)$ depend on the instance in a way to make the problems comparable *[BBO2016perf]*.

This means we want to consider at least $20 \times 5 = 100$, and up to $100 \times 15 \times 100 = 150\,000$ runtimes for the performance assessment. To make them amenable to the experimenter, we need to summarize these data.

Our idea behind an aggregation is to make a statistical summary over a set or subset of *problems of interest* over which we assume a uniform distribution *[BBO2016perf]*. From a practical perspective this means to have no simple way to distinguish between these problems and to select an optimization algorithm accordingly—in which case an aggregation would have no significance—and that we are likely to face each problem with similar probability. We do not aggregate over dimension, because dimension can and should be used for algorithm selection.

We have several ways to aggregate the resulting runtimes.

- Empirical cumulative distribution functions (ECDF). In the domain of optimization, ECDF are also known as *data profiles [MOR2009]*. We prefer the simple ECDF over the more innovative performance profiles *[MOR2002]* for two reasons. ECDF (i) do not depend on other presented algorithms, that is, they are entirely comparable across different publications, and (ii) let us distinguish in a natural way easy problems from difficult problems for the considered algorithm. We usually display ECDF on the log scale, which makes the area above the curve and the *difference area* between two curves a meaningful conception *[BBO2016perf]*.
- Averaging, as an estimator of the expected runtime. The average runtime, that is the estimated expected runtime, is often plotted against dimension to indicate scaling with dimension. The *arithmetic* average is only meaningful if the underlying distribution of the values is similar. Otherwise, the average of log-runtimes, or *geometric* average, is useful.
- Restarts and simulated restarts, see Section *Restarts and Simulated Restarts*, do not literally aggregate runtimes (which are literally defined only when t was hit). They aggregate, however, time data to eventually supplement missing runtime values, see also *[BBO2016perf]*.

4 General Code Structure

The code basis of the COCO code consists of two parts.

The *Experiments* part defines test suites, allows to conduct experiments, and provides the output data. The code base is written in C, and wrapped in different languages (currently Java, Python, Matlab/Octave). An amalgamation technique is used that outputs two files `coco.h` and `coco.c` which suffice to run experiments within the COCO framework.

The *post-processing* part processes the data and displays the resulting runtimes. This part is entirely written in Python and heavily depends on `matplotlib` *[HUN2007]*.

5 Test Suites

Currently, the COCO framework provides three different test suites.

bbob contains 24 functions in five subgroups *[HAN2009fun]*.

bbob-noisy contains 30 noisy problems in three subgroups *[HAN2009noi]*, currently only implemented in the old code basis.

bbob-biobj contains 55 bi-objective ($m = 2$) functions in 15 subgroups *[TUS2016]*.

Acknowledgments

The authors would like to thank Raymond Ros, Steffen Finck, Marc Schoenauer, Petr Posik and Dejan Tusar 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.

References

[BBO2016perf] The BBOBies (2016). COCO: Performance Assessment.

[HAN2010ex] N. Hansen, A. Auger, S. Finck, and R. Ros (2010). Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup, *Inria Research Report* RR-7215 http://hal.inria.fr/inria-00362649/en

[HAN2010] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Posik (2010). Comparing Results of 31 Algorithms from the Black-Box Optimization Benchmarking BBOB-2009. Workshop Proceedings of the GECCO Genetic and Evolutionary Computation Conference 2010, ACM, pp. 1689-1696

[HAN2009fun] 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, updated February 2010.

[HAN2009noi] N. Hansen, S. Finck, R. Ros, and A. Auger (2009). Real-Parameter Black-Box Optimization Benchmarking 2009: Noisy Functions Definitions. Technical Report RR-6869, Inria, updated February 2010.

[HAN2016ex] N. Hansen, T. Tusar, A. Auger, D. Brockhoff, O. Mersmann (2016). COCO: Experimental Procedure.

[HUN2007] J. D. Hunter (2007). Matplotlib: A 2D graphics environment, *Computing In Science & Engineering*, 9(3): 90-95.

[EFR1993] B. Efron and R. Tibshirani (1993). An introduction to the bootstrap. Chapman & Hall/CRC.

[HAR1999] G. R. Harik and F. G. Lobo (1999). A parameter-less genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, volume 1, pages 258-265. ACM.

[HOO1998] H. H. Hoos and T. Stützle (1998). Evaluating Las Vegas algorithms: pitfalls and remedies. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 238-245.

[MOR2009] J. Moré and S. Wild (2009). Benchmarking Derivative-Free Optimization Algorithms. *SIAM J. Optimization*, 20(1):172-191.

[MOR2002] D. Dolan and J. J. Moré (2002). Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91:201-213.

[STE1946] S.S. Stevens (1946). On the theory of scales of measurement. *Science* 103(2684), pp. 677-680.

[TUS2016] T. Tusar, D. Brockhoff, N. Hansen, A. Auger (2016). COCO: The Bi-objective Black Box Optimization Benchmarking (bbob-biobj) Test Suite.

[WHI1996] D. Whitley, S. Rana, J. Dzubera, K. E. Mathias (1996). Evaluating evolutionary algorithms. *Artificial intelligence*, 85(1), 245-276.