Advanced Exercise #2: Benchmarking (Blackbox) Optimizers

Advanced Optimization Lecture - CentraleSupélec/ESSEC

Dimo Brockhoff

November 10, 2021

Background:

This exercise is meant to be an addition to the course Advanced Optimization for those students who have heard the main lecture content already in earlier optimization courses or who want to dig deeper into the topic. This exercise is provided to self-teach some of the concepts, discussed later in class.

Learning Objective:

Understand how to experimentally compare (=benchmark) optimization algorithms with the help of the Comparing Continuous Optimizers Platform (COCO, `https://github.com/numbbo/coco`).

Note: The instructions and questions are kept short, vague, and/or open on purpose like for the first advanced exercise. Also here, the purpose is that you explore and have ideas on your own in terms of both implementation details and explanations of what is going on. This openness of the instructions and questions simulates (as realistically as possible in our context) an actual research project on the topic, for example in the form of a Master's thesis.

1 Preparation/Setting

When approaching a new (real-world) optimization problem (for example in industry), one of the fundamental questions is which algorithm to use. As theoretical analyses of most optimization algorithms on the problem at hand are not feasible, one has to rely on recommendations from past experience on other, similar problems—either from theoretical or experimental results. For most practically relevant optimization algorithms, a theoretical analysis, even on the simplest functions is difficult. Moreover, we are typically more interested in the performance of a *concrete implementation* of an algorithm than in an *abstract version* of it. Hence, we have to rely on experiments. This experimental comparison of algorithms is also known under the term benchmarking and an ongoing topic of research.

2 General Considerations of Benchmarking

As such, benchmarking of two or more optimization algorithms seems easy at first sight: pick one or more objective functions of interest, run the algorithms on those and report the results.

However, as usual, the devil lies in the details. Let us consider the easiest case here: algorithms to minimize functions $f : \mathbb{R}^n \to \mathbb{R}$ without any constraints.

Think about the following questions first and try to come up with a concrete experimental setting before to continue reading the next sections:

- Which functions shall you choose? How many? Why more than one at all?
- What is an experiment? What can we record during an optimization run? Which of those measurements are practically relevant?
- Because we are optimizing in $\mathbb{R}^n$: when shall we stop the algorithms? Is there a single criterion? What about multi-modal functions, i.e., functions with multiple local optima?
- Given that most algorithms have internal parameters and might need some starting conditions (initial search point, initial step size, ...): what is actually "an algorithm" here?

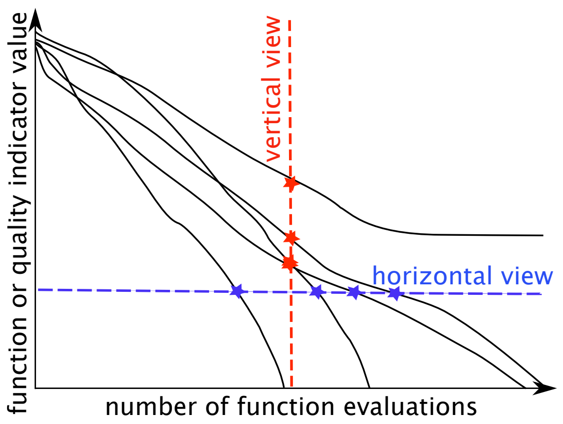
Advanced question How can we compare deterministic and stochastic algorithms?

Advanced question How to compare (stochastic) algorithms fairly when one is only sometimes coming close to the optimum, but then quickly, while another one is always finding good search points but (much) slower?

You see already here, that the originally thought trivial setup is not that trivial anymore in practice.

3 Convergence Graphs is (Kind of) All We Have to Start With

If we think about an algorithm, minimizing the function $f : \mathbb{R}^n \to \mathbb{R}$, the only data we can record (besides internal parameters of the algorithm) are the function values (and potentially gradients of f) at certain times during the experiment. These time stamps can either be iterations of the algorithm, function evaluations, or real time (for example CPU time). The latter, however, is very much related to the actual experimental setup of computer hardware, other threads running in the background, etc. and thus not easy to compare over different computer architectures and consequently when years or even decades lie between the experiments. Comparing algorithms with respect to the number of function/gradient/constraint/... evaluations is much easier to interpret when the experiments from different algorithms come from different people and/or are



vertical view
horizontal view
function or quality indicator value
number of function evaluations

Figure 1: Example of convergence graphs from five difference algorithms or from five runs of a stochastic algorithm on a given problem.

done over a wide range of time. Therefore, we will fall back on those comparisons in the following.

If we compare algorithms for blackbox problems (where the algorithm can query f (or the gradient of f or the constraint values) but does not have further information about how f looks like, this is the natural setup. The only data, we have available from a benchmarking experiment, in this case, is a so-called convergence graph, showing the evolution of the f-values from evaluated solutions over the number of function evaluations (typically in a semi-log or a log-log plot), see Figure 1.

4 Horizontal vs. Vertical View of Performance Assessment

If we do not want to store the function values of each and every solution evaluated, we have two choices: either we fix certain budgets (in terms of function evaluations) and record the corresponding function values reached at those times (vertical view of performance assessment) or we fix certain f-value precisions and report how long the algorithm needs to reach those f-values (horizontal view of performance assessment, see also Figure 1). Which choice is the better one and why? [hint: how easy it is to interpret the reported values?] Advanced question: what happens if algorithms are too fast or too slow for the recorded budgets/precisions such that no data point is recorded as in 1 out of the 5 cases in Figure 1?

5 Empirical Cumulative Distribution Functions: A Standard Way to Display Distributions of Values

If we have recorded certain f-value/number of function evaluations pairs, a common way to display them is to plot its empirical cumulative distribution function (ECDF), see `https://en.wikipedia.org/wiki/Empirical_distribution_function` for details. The ECDF is a step function that jumps by $1/n$ at each of the n data points.

If the (possible) data points are the runtimes of an algorithm of interest to reach a certain set of predefined t targets (typically, the number of displayed targets in COCO is a few dozens), the corresponding ECDF goes from 0 to 1 if and only if the last, most difficult target was actually reached by the algorithm (and correspondingly from 0 to the percentage of achieved target values otherwise).

Implement in the following the ECDF display of an algorithm run with the percentage of solved target precisions on the y-axis for each budget of number of function evaluations on the x-axis.

You might want to run an optimization algorithm (e.g. one from `scipy.optimize`) on a simple function (e.g. on the sphere function or on another convex quadratic function from `cma.ff`, see the last advanced exercise) and record the function values of all evaluated solutions to get some actual data to feed your ECDF plot. The easiest way to create such a real data set is probably to write a wrapper around the objective function that writes results to disk whenever the function is called.

6 ECDFs and Aggregation of Data

The nice property of ECDFs in our scenario of benchmarking is that they just display some distributions of runtimes and it is possible that those data recordings, displayed in an ECDF, can come from different experiments. Aggregation is therefore easily possible.

Those different experiments can be from different runs of a stochastic algorithm such as CMA-ES on the same function, or even from different functions (or from different instances of the same function). A function instance, here, can be seen as a variant of a generic function like the sphere function, where each instance is for example a sphere function with a shifted optimum.

Run an algorithm on two different functions and display the aggregated ECDF.

7 Using COCO

The Comparing Continuous Optimizers platform (COCO) has been developed to spare the users the time consuming setup, execution, and analyses of benchmarking experiments by implementing a standardized methodology for automated benchmarking. Besides providing pre-defined sets of benchmarking functions, so-called test suites, automated data acquisition during the experiment, and a well-thought benchmarking methodolody including visual and tabular output in HTML and LaTeX such as the above ECDF plots, the main contribution of COCO is to provide benchmarking data from previous numerical benchmarking experiments for easy comparison. This postprocessing part, implemented in the python module `cocopp`[1] provides easy access to more than 200 algorithm data sets ranging from classical algorithms like Quasi-Newton BFGS, Nelder-Mead (Simplex Downhill), NEWUOA, or SLSQP over population-based methods such as Differential Evolution, Ant Colony Optimization, Particle Swarm Optimization, Evolution Strategies including CMA-ES and Natural Evolution Strategy variants to surrogate-based algorithms based on Bayesian Optimization or quadratic approximation. Portfolios of algorithms or hybrids are equally available. For an overview of all algorithms, benchmarked on the classical, unconstrained, noiseless `bbob` test suite, please see here: `https://numbbo.github.io/data-archive/bbob/`.

If you want to explore the data, please go ahead and run the postprocessing (after installing via `pip install cocopp`) with a command like

```
python -m cocopp ALG1 ALG2 ... ALGN
```

where ALG1, ALG2, etc. are the algorithms, you want to compare. In case your choice is not unique, the postprocessing will let you know and you should be either more specific in your choice or choose the first entry of the provided list by appending a `!` after the algorithm name.

Please note that it might take a few minutes to run the postprocessing, in particular if you have not yet downloaded any data and/or if you specify too many algorithms at a time. Note equally that after the postprocessing is finished, your default browser will typically open and show you the results.

Have a look at what COCO provides as visual comparisons, in particular at the ECDF plots.

What do you observe? Which of the chosen algorithms performs best, over all functions, on specific functions, on groups of functions with similar properties? Does this depend on the problem dimension or other properties of the functions such as separability or multi-modality?

What about invariances? How can you gain insights about whether an algorithm is for example invariant under rotations of the search space?

[1] Installed easily via `pip install cocopp`.