Real-Parameter Black-Box Optimization Benchmarking: Experimental Setup

Nikolaus Hansen[*], Anne Auger[†], Steffen Finck[‡] and Raymond Ros[§]

compiled March 21, 2013

Abstract

Quantifying and comparing performance of numerical optimization algorithms is an important aspect of research in search and optimization. However, this task turns out to be tedious and difficult to realize even in the single-objective case – at least if one is willing to accomplish it in a scientifically decent and rigorous way. The COCO software used for the BBOB workshops (2009, 2010 and 2012) furnishes most of this tedious task for its participants: (1) choice and implementation of a well-motivated single-objective benchmark function testbed, (2) design of an experimental set-up, (3) generation of data output for (4) post-processing and presentation of the results in graphs and tables. What remains to be done for practitioners is to allocate CPU-time, run their favorite black-box real-parameter optimizer in a few dimensions a few hundreds of times and execute the provided post-processing scripts. Two testbeds are provided,

- noise-free functions
- noisy functions

and practitioners can freely choose any or all of them. The post-processing provides a quantitative performance assessment in graphs and tables, categorized by function properties like multi-modality, ill-conditioning, global structure, separability,...

This document describes the experimental setup and touches the question of how the results are displayed.

The benchmark function definitions, source code of the benchmark functions and for the post-processing and this report are available at http://coco.gforge.inria.fr/.

[*]NH is with the TAO Team of INRIA Saclay–Île-de-France at the LRI, Université-Paris Sud, 91405 Orsay cedex, France

[†]AA is with the TAO Team of INRIA Saclay–Île-de-France at the LRI, Université-Paris Sud, 91405 Orsay cedex, France

[‡]SF is with the Research Center PPE, University of Applied Science Vorarlberg, Hochschulstrasse 1, 6850 Dornbirn, Austria

[§]RR has been with the TAO Team of INRIA Saclay–Île-de-France at the LRI, Université-Paris Sud, 91405 Orsay cedex, France

Contents

1 Introduction **2**
1.1 Symbols, Constants, and Parameters 2

2 Benchmarking Experiment **3**
2.1 Input to the Algorithm and Initialization 3
2.2 Termination Criteria and Restarts 5

3 Time Complexity Experiment **5**

4 Parameter setting and tuning of algorithms **6**

5 Data to Provide **7**

6 Post-Processing and Data Presentation **7**
6.1 Processing the data 7
6.2 Compiling a LaTeX document 8

A Example Optimizer with Multistarts **11**

B How to Resume an Experiment **11**

C Rationales Behind the Parameter Settings **13**

D Rationale Behind the Data Presentation **13**
D.1 Performance Measure: Expected Running Time 13
D.2 Bootstrapping 14
D.3 Fixed-Cost versus Fixed-Target Scenario 14
D.4 Empirical Cumulative Distribution Functions 16

E Data and File Formats **17**
E.1 Introduction 17
E.2 General Settings 17
E.3 Output Files 17
E.3.1 Index File 17
E.3.2 Data Files 18

1 Introduction

This document is largely based on the BBOB-2009 experimental setup [5] and describes the experimental setup and the data presentation for BBOB-2012.

The definition of the benchmark functions and the technical documentation for the provided software are accessible at http://coco.gforge.inria.fr/.

1.1 Symbols, Constants, and Parameters

f_{opt} : optimal function value, defined for each benchmark function individually.

Δf : precision to reach, that is, a difference to the smallest possible function value f_{opt}.

$f_{\text{target}} = f_{\text{opt}} + \Delta f$: target function value to reach. The final, smallest considered target function value is $f_{\text{target}} = f_{\text{opt}} + 10^{-8}$, but also larger values for f_{target} are evaluated.

`Ntrial` = 15 is the number of trials for each single setup, i.e. each function and dimensionality (see also Appendix C). A different function instance is used in each trial. Performance is evaluated over all `Ntrial` trials.

$D = 2; 3; 5; 10; 20; 40$ search space dimensionalities used for all functions. Dimensionality 40 is optional and can be omitted.

2 Benchmarking Experiment

The real-parameter search algorithm under consideration is run on a testbed of benchmark functions to be minimized (the implementation of the functions is provided in C, Java, MATLAB/Octave and Python). On each function and for each dimensionality `Ntrial` trials are carried out (see also Appendix C). Different function *instances* are used (the instantiation numbers $1, \ldots, 5, 21, \ldots, 30$). A MATLAB example script for this procedure is given in Figure 1 (similar scripts are provided in C, Java and Python). The algorithm is run on *all* functions of the testbed under consideration.

2.1 Input to the Algorithm and Initialization

An algorithm can use the following input.

1. the search space dimensionality D
2. the search domain; all functions are defined everywhere in $\mathcal{R}^D$ and have their global optimum in $[-5, 5]^D$. Most functions have their global optimum in $[-4, 4]^D$ which can be a reasonable setting for initial solutions.
3. indication of the testbed under consideration, i.e. different algorithms and/or parameter settings might well be used for the noise-free and the noisy testbed.
4. the final target precision value *difference* $\Delta f = 10^{-8}$, in order to implement effective termination mechanisms and restarts (which should also prevent too early termination).
5. the target function value f_{target} can only be used for conclusive (final) termination of trials, in order to reduce the overall CPU requirements. The target function value must not be utilized as algorithm input otherwise.

Based on these input parameters, the parameter setting and initialization of the algorithm is entirely left to the participants. The setting however must be identical for all benchmark functions and all function instances (in other words for all trials) of one testbed. The function identifier or instance number or any known characteristics of the function must not be input to the algorithm, see also Section 4.

Figure 1: exampleexperiment.m: example for benchmarking MY_OPTIMIZER on the noise-free function testbed in MATLAB/Octave. An example for the function MY_OPTIMIZER is given in Appendix A. Similar code is available in C, Java and Python

---

```
% runs an entire experiment for benchmarking MY_OPTIMIZER
% on the noise-free testbed. fgeneric.m and benchmarks.m
% must be in the path of Matlab/Octave
% CAPITALIZATION indicates code adaptations to be made

addpath('PUT_PATH_TO_BBOB/matlab');  % should point to fgeneric.m etc.
datapath = 'PUT_MY_BBOB_DATA_PATH';  % different folder for each experiment
% opt.inputFormat = 'row';
opt.algName = 'PUT ALGORITHM NAME';
opt.comments = 'PUT MORE DETAILED INFORMATION, PARAMETER SETTINGS ETC';
maxfunevals = '10 * dim'; % 10*dim is a short test-experiment taking a few minutes
                          % INCREMENT maxfunevals successively to larger value(s)
minfunevals = 'dim + 2';  % PUT MINIMAL SENSIBLE NUMBER OF EVALUATIONS for a restart
maxrestarts = 1e4;        % SET to zero for an entirely deterministic algorithm

more off;  % in octave pagination is on by default

t0 = clock;
rand('state', sum(100 * t0));

for dim = [2,3,5,10,20,40]  % small dimensions first, for CPU reasons
  for ifun = benchmarks('FunctionIndices')  % or benchmarksnoisy(...)
    for iinstance = [1:5, 31:40]  % 15 function instances
      fgeneric('initialize', ifun, iinstance, datapath, opt);

      % independent restarts until maxfunevals or ftarget is reached
      for restarts = 0:maxrestarts
        if restarts > 0  % write additional restarted info
          fgeneric('restart', 'independent restart')
        end
        MY_OPTIMIZER('fgeneric', dim, fgeneric('ftarget'), ...
                     eval(maxfunevals) - fgeneric('evaluations'));
        if fgeneric('fbest') < fgeneric('ftarget') || ...
           fgeneric('evaluations') + eval(minfunevals) > eval(maxfunevals)
          break;
        end
      end

      disp(sprintf(['  f%d in %d-D, instance %d: FEs=%d with %d restarts,' ...
                    ' fbest-ftarget=%.4e, elapsed time [h]: %.2f'], ...
                   ifun, dim, iinstance, ...
                   fgeneric('evaluations'), ...
                   restarts, ...
                   fgeneric('fbest') - fgeneric('ftarget'), ...
                   etime(clock, t0)/60/60));

      fgeneric('finalize');
    end
    disp(['      date and time: ' num2str(clock, ' %.0f')]);
  end
  disp(sprintf('---- dimension %d-D done ----', dim));
end
```

2.2 Termination Criteria and Restarts

Algorithms with any budget of function evaluations, small or large, are considered in the analysis of the results. Exploiting a larger number of function evaluations increases the chance to achieve better function values or even to solve the functions up to the final f_{target}[1]. In any case, a trial can be conclusively terminated if f_{target} is reached. Otherwise, the choice of termination is a relevant part of the algorithm: the termination of unsuccessful trials can significantly effect the performance. To exploit a large number of function evaluations effectively, we suggest considering a multistart procedure, which relies on an interim termination of the algorithm, see also Figure 1.

Independent restarts, as implemented in Figure 1, do not change the main performance measure *expected running time* (ERT, see Appendix D.1). Independent restarts mainly improve the reliability and "visibility" of the measured value. For example, when using a fast algorithm with a small success probability, say 5% (or 1%), not a single of 15 trials might be successful. With 10 (or 90) independent restarts, the success probability will increase to 40% and the performance of the algorithm will become visible. At least 4–5 successful trials (here out of 15) are desirable to accomplish a stable performance measurement. This reasoning remains valid for any target function value (different values are considered in the evaluation).

Restarts either from a previous solution, or with a different parameter setup, for example with different (increasing) population size, might be considered as well, as it has been applied quite successfully [2, 6, 9]. Choosing different setups mimics what might be done in practice. All restart mechanisms are finally considered as part of the algorithm under consideration. Though this information is not used in the post processing, the function evaluations at which restarts occur are logged in the corresponding `.rdat` file, compare Figure 1.

3 Time Complexity Experiment

In order to get a rough measurement of the time complexity of the algorithm, the overall CPU time is measured when running the algorithm on f_8 (Rosenbrock function) for at least a few tens of seconds (and at least a few iterations). The chosen setup should reflect a "realistic average scenario". If another termination criterion is reached, the algorithm is restarted (like for a new trial). The *CPU-time per function evaluation* is reported for each dimension. The time complexity experiment is conducted in the same dimensions as the benchmarking experiment. The chosen setup, coding language, compiler and computational architecture for conducting these experiments are described. Figure 2 shows a respective MATLAB/Octave code example for launching this experiment. For CPU-inexpensive algorithms the timing might mainly reflect the time spent in function fgeneric.

[1] The easiest functions can be solved in less than $10D$ function evaluations, while on the most difficult functions a budget of more than $1000D^2$ function evaluations to reach *the final* $f_{\text{target}} = f_{\text{opt}} + 10^{-8}$ is expected.

Figure 2: exampletiming.m: example for measuring the time complexity of MY_OPTIMIZER given in MATLAB/Octave. An example for MY_OPTIMIZER is given in Appendix A

```
% runs the timing experiment for MY_OPTIMIZER. fgeneric.m
% and benchmarks.m must be in the path of MATLAB/Octave

addpath('PUT_PATH_TO_BBOB/matlab');  % should point to fgeneric.m etc.

more off;  % in octave pagination is on by default

timings = [];
runs = [];
dims = [];
for dim = [2,3,5,10,20,40]
  nbrun = 0;
  ftarget = fgeneric('initialize', 8, 1, 'tmp');
  tic;
  while toc < 30  % at least 30 seconds
    MY_OPTIMIZER(@fgeneric, dim, ftarget, 1e5);  % adjust maxfunevals
    nbrun = nbrun + 1;
  end  % while
  timings(end+1) = toc / fgeneric('evaluations');
  dims(end+1) = dim;    % not really needed
  runs(end+1) = nbrun;  % not really needed
  fgeneric('finalize');
  disp([['Dimensions:' sprintf(' %11d ', dims)]; ...
        ['      runs:' sprintf(' %11d ', runs)]; ...
        [' times [s]:' sprintf(' %11.1e ', timings)]]);
end
```

4 Parameter setting and tuning of algorithms

The algorithm and the used parameter setting for the algorithm should be described thoroughly. Any tuning of parameters to the testbed must be described and the approximate number of tested parameter settings should be given.

All functions in a testbed must be approached with the very same parameter setting (which might only depend on the dimensionality, see Section 2.1).[2]

In other words, we do not consider the function ID or any function characteristics (like separability, multi-modality, . . .) as input parameter to the algorithm (see also Section 2.1). Instead, we encourage benchmarking different parameter settings as "different algorithms" on the entire testbed. In order to combine different parameter settings, one might use either multiple runs with different parameters (for example restarts, see also Section 2.2), or use (other) probing techniques for identifying function-wise the appropriate parameters online. The underlying assumption in this experimental setup is that also in practice we do

[2]Thus, the *crafting effort* which was used in previous editions of BBOB to quantify the effort to tune to particular functions must be zero and is not considered as a parameter any more.)

not know in advance whether the algorithm will face f_1 or f_2, a unimodal or a multimodal function... therefore we cannot adjust algorithm parameters *a priori*[3].

5 Data to Provide

The provided implementations of the benchmark functions generate data for reporting and analysis. Since one goal is the comparison of different algorithms, the data from the experiments shall be submitted to `http://coco.gforge.inria.fr/`. All submitted data will be made available to the public. We strongly encourage to also submit the source code or libraries that allow to reproduce the submitted data.

6 Post-Processing and Data Presentation

Python scripts are provided to produce tables and figures reporting the outcome of the benchmarking experiment that can be compiled into a paper in the following way. A more detailed description of the post-processing software is accessible at `http://coco.gforge.inria.fr/doku.php?id=downloads`.

6.1 Processing the data

Given the output data from an experiment are in the folder(s) mydata0 (mydata1...mydataN), the command

```
python path_to_postproc_code/bbob_pproc/rungeneric.py mydata0 [mydata1 ... mydataN]
```

processes the data (depending on the number of arguments) and stores the results in folder `ppdata`. For more than one argument the individual figures and tables of ALG0 to ALGN will be in `ppdata/mydata0` to `ppdata/mydataN`.

Alternatively, the specific commands can be accessed for processing one, two, or many algorithms.

Single Algorithm Given the output data from an experiment are in the folder `mydata` of the current directory[4], the command line[5]

```
python path_to_postproc_code_folder/bbob_pproc/rungeneric1.py mydata
```

creates the folder/subfolder `ppdata/` in the working directory containing figures and tables.

[3]In contrast to most other function properties, the property of having noise can usually be verified easily. Therefore, for noisy functions a *second* testbed has been defined. The two testbeds can be approached *a priori* with different parameter settings or different algorithms.

[4]The data can be distributed over several folders. In this case several folders are given as trailing arguments.

[5] Under Windows the path separator '\' instead of '/' must be used in the command line. Python 2.X, Numpy, and Matplotlib must be installed. For higher Python versions, e.g. 3.X, the necessary libraries are not (yet) available. Python 2.X, Numpy and Matplotlib are freely available on all platforms. Python can be downloaded from `http://www.python.org/download/releases/`, Numpy from `http://numpy.scipy.org` and Matplotlib from `http://matplotlib.sourceforge.net`.

Comparison of Two Algorithms Let the output data from two experiments with two different algorithms be in the folders `mydata0` and `mydata1` respectively for the algorithms ALG0 and ALG1.

```
python path_to_postproc_code/bbob_pproc/rungeneric2.py mydata0 mydata1
```

generates the folder `ppdata` containing the comparison figures and tables[6]. The data are displayed in this fashion: `mydata1` (the second argument) is for the data of the "algorithm under consideration" or "algorithm of interest", while `mydata0` (first argument) is for the data of the "algorithm to compare with".

Comparison of More Than Two Algorithms Let the output data from experiments with N different algorithms be in the folders `mydata0` ... `mydataN` respectively for the algorithms ALG0 to ALGN.

```
python path_to_postproc_code/bbob_pproc/rungenericmany.py \
      mydata0 mydata1 ... mydataN
```

generates the folder `ppdata` containing the comparison figures and tables[7].

Expensive setting If the maximum budget for the experiments conducted is below a certain threshold (typically a few hundred times D function evaluations), a different presentation of the data is generated (where targets are chosen based on function values reached by the best 2009 algorithm after a fixed budget). However the same latex template needs to be compiled.

To enforce the "expensive" post-processing for data obtained with a "normal" budget we can use the command

```
python path_to_pproc/bbob_pproc/rungeneric.py --expensive mydata
```

This option is explain via

```
python path_to_bbob_pproc/rungeneric1.py --help
```

6.2 Compiling a LaTeX document

Finally, the paper template is compiled by one of the commands

```
pdflatex templateBBOBarticle
pdflatex templateBBOBcmp
pdflatex templateBBBmany
pdflatex templateBBOBnoisyarticle
pdflatex templateBBOBnoisycmp
pdflatex templateBBOBnoisymany
```

[6] The same can be accomplished within a Python shell by typing "`from bbob_pproc import rungeneric; bbob_pproc.generic.main('mydata0 mydata1'.split())`". The first command requires that the path to the package `bbob_pproc` is in the Python search path.

[7] The same can be accomplished within a Python shell by typing "`from bbob_pproc import rungeneric; bbob_pproc.generic.main('mydata0 mydata1 ... mydataN'.split())`". The first command requires that the path to the package `bbob_pproc` is in the Python search path.

that uses tables, captions and figures from `ppdata`. Note that the same template can be used if run with low ("expensive") or "normal" budget as figures, tables and captions are adapted automatically during the postprocessing.

The generic templates

```
pdflatex template1generic
pdflatex template2generic
pdflatex template3generic
pdflatex noisytemplate1generic
pdflatex noisytemplate2generic
pdflatex noisytemplate3generic
```

are not specific to the BBOB workshop but follow the same syntax.

The folder `ppdata` and the respective template `.tex` file have to be in the working directory. Compiled examples are accessible at http://coco.gforge.inria.fr/.

Acknowledgments

The authors would like to thank Petr Pošík and Arnold Neumaier for the inspiring and helpful discussion. Steffen Finck was supported by the Austrian Science Fund (FWF) under grant P19069-N18.

References

[1] 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)*, pages 1777–1784, 2005.

[2] A Auger and N Hansen. A restart CMA evolution strategy with increasing population size. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2005)*, pages 1769–1776. IEEE Press, 2005.

[3] Anne Auger and Raymond Ros. Benchmarking the pure random search on the BBOB-2009 testbed. In Franz Rothlauf, editor, *GECCO (Companion)*, pages 2479–2484. ACM, 2009.

[4] B. Efron and R. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall/CRC, 1993.

[5] N. Hansen, A. Auger, S. Finck, and R. Ros. Real-parameter black-box optimization benchmarking 2009: Experimental setup. Technical Report RR-6828, INRIA, 2009.

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

[7] H.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)*, pages 238–245, 1998.

[8] Kenneth Price. Differential evolution vs. the functions of the second ICEO. In *Proceedings of the IEEE International Congress on Evolutionary Computation*, pages 153–157, 1997.

[9] Raymond Ros. Black-box optimization benchmarking the IPOP-CMA-ES on the noiseless testbed: comparison to the BIPOP-CMA-ES. In *GECCO '10: Proceedings of the 12th annual conference comp on Genetic and evolutionary computation*, pages 1503–1510, New York, NY, USA, 2010. ACM.

APPENDIX

A Example Optimizer with Multistarts

The optimizer used in Fig. 1 and 2 is given in Fig. 3.

B How to Resume an Experiment

We give a short description of how to cleanly resume an experiment that was aborted before its completion.

1. Find the last modified `.info` file (see Appendix E). Function number and dimension, where the experiment was aborted, are given in the third to last line of the file. For example:

```
funcId = 13, DIM = 40, Precision = 1.000e-08, algId = 'my optimizer'
% all default parameters
data_f13/bbobexp_f13_DIM40.dat, 1:5387|-4.4e-09, 2:5147|-3.9e-09, 3
```

   The last line points to the written data file and the last number in the last line contains the function instance number of the unfinished trial (see also Appendix E).

Now, there are two options: either restarting by rerunning all experiments for f_{13} in 40-D, or restarting from the very instance 3, which is more involved.

Option 1 (rerun the complete "line" in info file)

2. (optional) delete the respective three(!) data files, in our example

```
data_f13/bbobexp_f13_DIM40.dat
data_f13/bbobexp_f13_DIM40.tdat
data_f13/bbobexp_f13_DIM40.rdat
```

3. delete the last three lines in the info file
4. modify your experiment script (see e.g. Fig. 1) to restart with the respective function and dimension, here f_{13} in 40-D

Option 2 (rerun from the broken trial)

2. remove the last characters, in the above example, "`, 3`" from the last line of the info file. If the first entry is already the unfinished one, refer to Option 1.
3. remove the respective data of the unfinished last trial in *all three* data files, `.dat`, `.tdat` and `.rdat`, in our example

```
data_f13/bbobexp_f13_DIM40.dat
data_f13/bbobexp_f13_DIM40.tdat
data_f13/bbobexp_f13_DIM40.rdat
```

4. modify you experiment script to restart your experiment from this very function instance (which can be a bit tricky).

Figure 3: Example optimizer used in Fig. 1 and 2

```
function [x, ilaunch] = MY_OPTIMIZER(FUN, DIM, ftarget, maxfunevals)
% minimizes FUN in DIM dimensions by multistarts of fminsearch.
% ftarget and maxfunevals are additional external termination conditions,
% where at most 2 * maxfunevals function evaluations are conducted.
% fminsearch was modified to take as input variable usual_delta to
% generate the first simplex.

  % set options, make sure we always terminate
  % with restarts up to 2*maxfunevals are allowed
  options = optimset('MaxFunEvals', min(1e8*DIM, maxfunevals), ...
                     'MaxIter', 2e3*DIM, ...
                     'Tolfun', 1e-11, ...
                     'TolX', 1e-11, ...
                     'OutputFcn', @callback, ...
                     'Display', 'off');

  % set initial conditions
  xstart = 8 * rand(DIM, 1) - 4;  % random start solution
  usual_delta = 2;
  % refining multistarts
  for ilaunch = 1:1e4;  % up to 1e4 times
    % try fminsearch from Matlab, modified to take usual_delta as arg
    x = fminsearch_mod(FUN, xstart, usual_delta, options);
    % terminate if ftarget or maxfunevals reached
    if feval(FUN, 'fbest') < ftarget || ...
          feval(FUN, 'evaluations') >= maxfunevals
      break;
    end
    % terminate with some probability
    if rand(1,1) > 0.90/sqrt(ilaunch)
       break;
    end
    xstart = x;  % try to improve found solution
    usual_delta = 0.1 * 0.1^rand(1,1);  % with small "radius"
    % if useful, modify more options here for next launch
  end

  function stop = callback(x, optimValues, state)
    stop = false;
    if optimValues.fval < ftarget
      stop = true;
    end
  end  % function callback

end  % function
```

C Rationales Behind the Parameter Settings

Rationale for the choice of `Ntrial` = 15 (see also Section 2.2) Parameter `Ntrial` determines the minimal measurable success rate and influences the minimally necessary CPU time. Compared to a standard setup for testing stochastic search procedures, we have chosen a small value for `Ntrial`. If the algorithm terminates before f_{target} is reached, longer trials can be trivially achieved by independent multistarts *within* each trial as implemented in Figure 1. Therefore, `Ntrial` effectively determines only a lower limit for the number of trials. With restarts within single trials, the success rate can, in principle, be raised to any desired level, if the algorithm is "(globally) stochastic"[8].

Because these multistarts are conducted within each trial, more sophisticated (dependent) restart strategies are feasible. Finally, 15 trials are sufficient to make relevant performance differences statistically significant[9].

Rationale for the choice of f_{target} The initial search domain and the target function value are an essential part of the benchmark function definition. Different target function values might lead to different characteristics of the problem to be solved. Smaller target values on the same function are invariably more difficult to reach. Functions can be easy to solve up to a function value of, say, 1 and become intricate for smaller target values. The actually chosen value for the final f_{target} is somewhat arbitrary (however numerical precision limits this "arbitrariness") and reasonable values change by simple modifications in the function definition (e.g. an affine linear transformation of the function value). The performance evaluation will consider a wide range of different target function values to reach, all being larger or equal to the final $f_{\text{target}} = f_{\text{opt}} + 10^{-8}$.

D Rationale Behind the Data Presentation

D.1 Performance Measure: Expected Running Time

We advocate performance measures that are

- quantitative, ideally with a ratio scale (opposed to interval or ordinal scale)[10] and with a wide variation (i.e. for example with values ranging not only between 0.98 and 1)
- well-interpretable, in particular by having a meaning and semantics attached to the number
- relevant with respect to the "real world"
- as simple as possible

[8] That is, any subset with non-zero volume is hit with positive probability, for example trivially achieved during initialization.

[9] A much larger number of trials has the undesirable effect that even tiny, irrelevant performance differences become statistically significant.

[10] `http://en.wikipedia.org/w/index.php?title=Level_of_measurement&oldid=261754099` gives an introduction to scale types.

For these reasons we use the *expected running time* (ERT, introduced in [8] as ENES and analyzed in [1] as success performance) as most prominent performance measure, more precisely, the expected number of function evaluations to reach a target function value for the first time. For a non-zero success rate p_s, the ERT computes to

$$\begin{aligned} \mathrm{ERT}(f_\mathrm{target}) &= \mathrm{RT}_\mathrm{S} + \frac{1-p_\mathrm{s}}{p_\mathrm{s}}\,\mathrm{RT}_\mathrm{US} && (1)\\ &= \frac{p_\mathrm{s}\mathrm{RT}_\mathrm{S} + (1-p_\mathrm{s})\mathrm{RT}_\mathrm{US}}{p_\mathrm{s}} && (2)\\ &= \frac{\#\mathrm{FEs}(f_\mathrm{best} \geq f_\mathrm{target})}{\#\mathrm{succ}} && (3) \end{aligned}$$

where the *running times* RT_S and RT_US denote the average number of function evaluations for successful and unsuccessful trials, respectively (zero for none respective trial), and p_s denotes the fraction of successful trials. Successful trials are those that reached f_target and evaluations after f_target was reached are disregarded. The $\#\mathrm{FEs}(f_\mathrm{best} \geq f_\mathrm{target})$ is the number of function evaluations conducted in all trials, while the best function value was not smaller than f_target during the trial, i.e. the sum over all trials of

$$\max\{\mathrm{FE\ s.t.}\ f_\mathrm{best}(\mathrm{FE}) \geq f_\mathrm{target}\}\ .$$

The #succ denotes the number of successful trials. ERT estimates the expected running time to reach f_target [1], as a function of f_target. In particular, RT_S and p_s depend on the f_target value. Whenever not all trials were successful, ERT also depends (strongly) on the termination criteria of the algorithm.

D.2 Bootstrapping

The ERT computes a single measurement from a data sample set (in our case from `Ntrial` optimization runs). Bootstrapping [4] can provide a dispersion measure for this aggregated measurement: here, a “single data sample” is derived from the original data by repeatedly drawing single trials with replacement until a successful trial is drawn. The running time of the single sample is computed as the sum of function evaluations in the drawn trials (for the last trial up to where the target function value was reached) [1, 3]. The distribution of the bootstrapped running times is, besides its displacement, a good approximation of the true distribution. Bootstrapped runs are used to display the empirical cumulative distribution of run-lengths and to provide a dispersion measure as percentiles of the bootstrapped distribution in tables.

D.3 Fixed-Cost versus Fixed-Target Scenario

Two different approaches for collecting data and making measurements from experiments are schematically depicted in Figure 4.

Fixed-cost scenario (vertical cuts). Fixing a number of function evaluations (this corresponds to fixing a cost) and measuring the function values reached for this given number of function evaluations. Fixing search costs

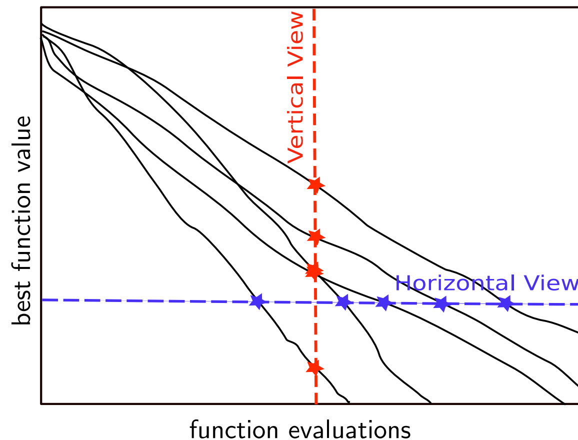

Figure 4: Illustration of fixed-cost view (vertical cuts) and fixed-target view (horizontal cuts). Black lines depict the best function value plotted versus number of function evaluations. Results obtained are (i) in the fixed-cost (vertical) view five objective function values, one for each graph, and the assignment which of them is better (a ranking). The resulting values can be interpreted (or used in computations) only with a thorough understand of the underlying test function. Values from different test functions are in general incompatible; (ii) in the fixed-target (horizontal) view five runtime values (number of function evaluations), one for each graph, thereof the quotient between two is the factor of *how much* slower/faster one value is compared to the other (or how much slower/faster *at least*). The interpretation of the values (as runtimes) is independent of the test function. For example, from the figure we can conclude that the worst run is about two times slower than the best one (given the x-axis is displayed on a linear scale with zero on the left). Values from different test functions are compatible and can be aggregated.

can be pictured as drawing a vertical line on the convergence graphs (see Figure 4 where the line is depicted in red).

Fixed-target scenario (horizontal cuts). Fixing a target function value and measuring the number of function evaluations needed to reach this target function value. Fixing a target can be pictured as drawing a horizontal line in the convergence graphs (Figure 4 where the line is depicted in blue).

It is often argued that the fixed-cost approach is close to what is needed for real word applications where the total number of function evaluations is limited. On the other hand, also a minimum target requirement needs to be achieved in real world applications, for example, getting (noticeably) better than the currently available best solution or than a competitor.

For benchmarking algorithms we prefer the fixed-target scenario over the fixed-cost scenario since it gives *quantitative and interpretable* data: the fixed-target scenario (horizontal cut) *measures a time* needed to reach a target function value and allows therefore conclusions of the type: Algorithm A is (at least) two/ten/hundred times faster than Algorithm B in solving this problem (i.e.

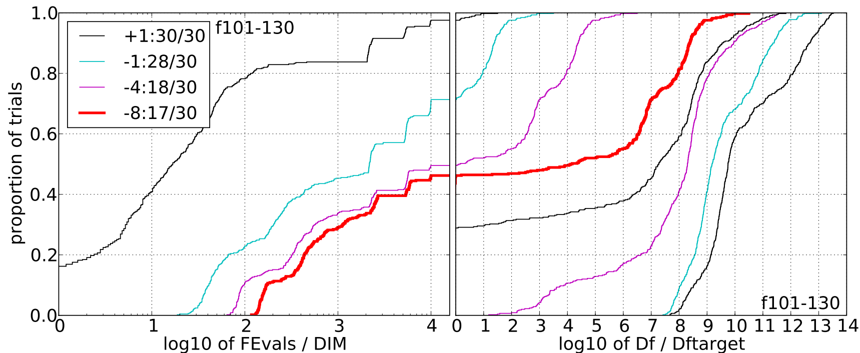

Figure 5: Illustration of empirical (cumulative) distribution functions (ECDF) of running length (left) and precision (right) arising respectively from the fixed-target and the fixed-cost scenarios in Fig. 4. In each graph the data of 450 trials are shown. Left subplot: ECDF of the running time (number of function evaluations), divided by search space dimension D, to fall below $f_{\mathrm{opt}} + \Delta f$ with $\Delta f = 10^k$, where $k = 1, -1, -4, -8$ is the first value in the legend. Right subplot: ECDF of the best achieved precision Δf divided by 10^k (thick red and upper left lines in continuation of the left subplot), and best achieved precision divided by 10^{-8} for running times of $D, 10\,D, 100\,D$, $1000\,D$... function evaluations (from the rightmost line to the left cycling through black-cyan-magenta-black).

reaching the given target function value). The fixed-cost scenario (vertical cut) does not give *quantitatively interpretable* data: there is no interpretable meaning to the fact that Algorithm A reaches a function value that is two/ten/hundred times smaller than the one reached by Algorithm B, mainly because there is no *a priori* evidence *how much* more difficult it is to reach a function value that is two/ten/hundred times smaller. Furthermore, for algorithms invariant under transformations of the function value (for example order-preserving transformations for algorithms based on comparisons like DE, ES, PSO), fixed-target measures can be made invariant to these transformations by simply transforming the chosen target function value while for fixed-cost measures all resulting data need to be transformed.

D.4 Empirical Cumulative Distribution Functions

We exploit the "horizontal and vertical" viewpoints introduced in the last Section D.3. In Figure 5 we plot the empirical cumulative distribution function[11] (ECDF) of the intersection point values (stars in Figure 4). A cutting line in Figure 4 corresponds to a "data" line in Figure 5, where 450 (30 × 15) convergence graphs are evaluated. For example, the thick red graph in Figure 5 shows on the left the distribution of the running length (number of function evaluations) [7] for reaching precision $\Delta f = 10^{-8}$ (horizontal cut). The graph

[11] The empirical (cumulative) distribution function $F : \mathcal{R} \to [0, 1]$ is defined for a given set of real-valued data S, such that $F(x)$ equals the fraction of elements in S which are smaller than x. The function F is monotonous and a lossless representation of the (unordered) set S.

continues on the right as a vertical cut for the maximum number of function evaluations, showing the distribution of the best achieved Δf values, divided by 10^{-8}. Run length distributions are shown for different target precisions Δf on the left (by moving the horizontal cutting line up- or downwards). Precision distributions are shown for different fixed number of function evaluations on the right. Graphs never cross each other. The y-value at the transition between left and right subplot corresponds to the success probability. In the example, just under 50% for precision 10^{-8} (thick red) and just above 70% for precision 10^{-1} (cyan).

E Data and File Formats

E.1 Introduction

This section specifies the format for the output data files and the content of the files, as they are written by the provided benchmark functions implementations. The goal is to obtain format-identical files which can be analyzed with the provided post-processing tools. The first section explains the general settings. Afterwards the format for the different output files will be given in detail and with examples.

E.2 General Settings

The output from `Ntrial` optimization runs for a given function-dimension pair is written in a folder which path is decided by the user and consists of at least one index file and at least two data files. The output files contain all necessary data for post-processing. The extensions are '*.info' for the index file and '*.dat', '*.tdat', '*.rdat' for the data files. An example of the folder/file structure can be found in Fig 6. After performing all simulations, the user can use the data files with the provided post-processing tool to obtain LaTeX files, including tables and figures of the results.

E.3 Output Files

E.3.1 Index File

The index file contains meta information on the optimization runs and the location of the corresponding data files. The user is free to choose any prefix for the index file name. The function identifier will be appended to it and the extension will be '.info'. The contents of the index file are the concatenation of 3-line index entries (output format is specified in brackets):

- *1st* line - function identifier (%d), search space dimension (%d), precision to reach (%4.3e) and the identifier of the used algorithm (%s)
- *2nd* line - comments of the user (e.g. important parameter or used internal methods)
- *3rd* line - relative location and name of data file(s) followed by a colon and information on a single run: the instance of the test function, final number of function evaluations, a vertical bar and the final best function value minus target function value.

```
↪ container_folder
      ↪ fileprefix_f1.info
      ↪ data_f1
        ↪ fileprefix_f1_DIM5.dat
        ↪ fileprefix_f1_DIM5.tdat
        ↪ fileprefix_f1_DIM5.rdat
        ↪ fileprefix_f1_DIM10.dat
        ↪ fileprefix_f1_DIM10.tdat
        ↪ fileprefix_f1_DIM10.rdat
      ↪ fileprefix_f2.info
      ↪ data_f2
        ↪ fileprefix_f2_DIM5.dat
        ↪ fileprefix_f2_DIM5.tdat
        ↪ fileprefix_f2_DIM5.rdat
↪ container_folder2
↪ ...
```

Figure 6: Example data file structures obtained with the BBOB experiment software.

All entries in the $1st$ line and the $3rd$ line are separated by commas.

For each function-dimension pair the provided data-writing tools generate one index file with the respective entries. All index files have to be included in the archive for the submission and will contain the *relative* location of the data files *within* the archive. Thus, it is necessary to archive all files in the same folder-subfolder structure as obtained. An example of an index file is given in Fig 7. An entry of the index file is written at the start of the first sample run

```
funcId = 12, DIM = 5, Precision = 1.000e-08, algId = 'ALG-A'
% parameterA = 2, parameterB = 3.34, ...
data_f12\test_f12_DIM5.dat, 1:387|-2.9e-009, 2:450|-2.8e-009, 3:422|-2.1e-009, data_f12\test-01_f12_DIM5.dat, 1:5000000|1.8e-008,
...
funcId = 12, DIM = 10, Precision = 1.000e-08, algId = 'ALG-A'
% parameterA = 2, parameterB = 3.34, ...
data_f12\test1_f12_DIM10.dat, 1:307|-8.6e-008, 2:321|-3.5e-008, ...
...
```

Figure 7: Example of an index file

for a given function and dimension.

E.3.2 Data Files

A data file contains the numerical output of an optimization run on a given objective function. The content of the data file is given in the following. Data files will be placed in subfolders at the location of their corresponding index file. At the start of each sample run the header for the data file is written. The header is one line with the titles for each data column:

- function evaluation
- noise-free fitness - Fopt (and its value)
- best noise-free fitness - Fopt
- measured fitness
- best measured fitness
- x1, x2, ... (one column for each dimension)

Fopt is the optimum of the test function considered. In the header, each of these entries are separated by the |-symbol. Each data line in the data file contains the following information:

- $1st$ column - recent number of function evaluation in format %d
- $2nd$ column - recent noise-free function value in format %+10.9e
- $3rd$ column - best noise-free function value so far in format %+10.9e
- $4th$ column - recent measured (noisy) function value in format %+10.9e
- $5th$ column - best measured (noisy) function value so far in format %+10.9e
- (5+d)th column - value of the dth (d $= 1, 2, \ldots, DIM$) object parameter of the best so far noise-free function value ($3rd$ column) in format %+5.4e

An example is given in Fig 8.

```
% function evaluation | noise-free fitness - Fopt (6.671000000000e+01) | best noise-free fitness - Fopt | measured fitness | best
measured fitness | x1 | x2 |...
1 +9.324567891e+05 +9.324567891e+05 +1.867342122e+06 +1.867342122e+06 +4.2345e+01 ...
2 +9.636565611e+05 +9.324567891e+05 +8.987623162e+05 +8.987623162e+05 +3.8745e+01 ...
...
31623 9.232667823e+01 9.576575761e+01 -6.624783627e+01 -1.657621581e+02 +5.1234e-02 ...
32478 1.000043784e+02 9.576575761e+01 -4.432869272e+01 -1.657621581e+02 +3.8932e-02 ...
35481 ...
...
```

Figure 8: Example of a data file

Each entry in the index files is associated to at least two data files: one for the function value-aligned data and another for the number of function evaluations-aligned data. The data file names are identical except for the file extension being '*.dat', '*.tdat' and '*.rdat' respectively. The '*.rdat' files contains two lines for each restart. The first contains a restart comment (which must be provided by the algorithm) and the second reflects the state of the algorithm before the restart in the same format as in the other two data files.

The writing to the function value aligned data file happens only each time the noise-free function value minus the optimum function value is less than $10^{i/5}$, for all integer i, for the first time (note, that the provided software does not return this difference to the algorithm).

The writing to the number of function evaluations aligned data file happens:

- in the first file each time the function evaluation number is equal to $\lfloor 10^{i/20} \rfloor$ for at least one $i = 1, 2, \ldots$ This means, that writing happens after about 12.2% additional function evaluations have been conducted. In particular the first 8 evaluations are written and also evaluations ..., 89, 100, 112, 125, 141, ..., 707, 794, 891, 1000, 1122, ...

- when any termination criterion is fulfilled (writing the recent evaluation and the current best so far values)

The prefix for the data file names of one function-dimension pair will be the same as the prefix of the corresponding index file. The function identifier and the dimension of the object parameters will be appended to this prefix. All data files will be saved in subfolders `data_fX`, where `X` is the function identifier, located at the same location as their index file.