COCO - COmparing Continuous Optimizers : The Documentation

Nikolaus Hansen, Steffen Finck, Raymond Ros

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

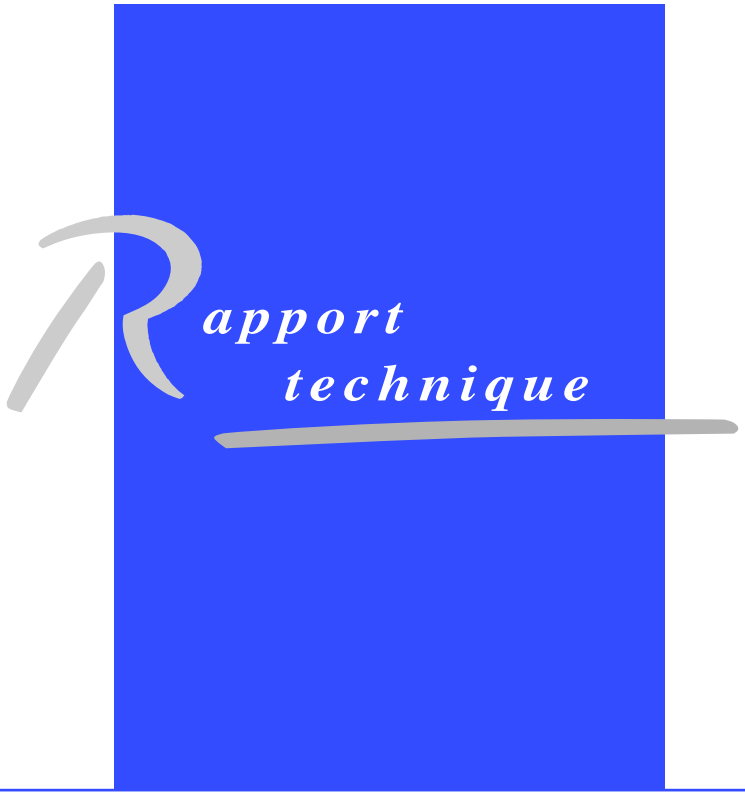
COCO – COmparing Continuous Optimizers : The Documentation

Nikolaus Hansen — Steffen Finck — Raymond Ros

N° 0409

May 2011

Thème COG



ISSN 0249-0803 ISRN INRIA/RT--0409--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

COCO – COmparing Continuous Optimizers : The Documentation

Nikolaus Hansen*, Steffen Finck†, Raymond Ros‡

Thème COG — Systèmes cognitifs
Équipes-Projets Adaptive Combinatorial Search et TAO

Rapport technique n° 0409 — May 2011 — 57 pages

Abstract: COmparing Continuous Optimisers (COCO) is a tool for benchmarking algorithms for black-box optimisation. COCO facilitates systematic experimentation in the field of continuous optimization.

COCO provides an experimental framework for testing the algorithms, post-processing facilities for generating publication-quality figures and tables, LaTeX templates of articles which present the figures and tables in a single document.

Key-words: software, optimization, evolutionary algorithms, benchmarking, black-box

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

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

COCO – COmparing Continuous Optimizers : documentation

Résumé : COmparing Continuous Optimisers (COCO) est un outil pour la comparaison d'algorithmes pour l'optimisation boîte noire. COCO facilite l'expérimentation systématique dans le domaine de l'optimisation continue.

COCO fournit un cadre expérimental pour tester des algorithmes, des outils de post-traitement pour générer des figures et tableaux de qualité pour la publication scientifique. COCO fournit aussi des modèles d'articles LaTeX qui présentent l'ensemble des figures et tableaux en un seul document.

Mots-clés : logiciel, optimisation, algorithmes évolutionnaires, banc d'essai, boîte noire

COCO Documentation

Release 10.7

S. Finck, N. Hansen, R. Ros

May 31, 2011

CONTENTS

This document serves as documentation for the COCO (COmparing Continuous Optimizers) software.

This software was mainly developed in the TAO team.

More information and the full documentation can be found at http://coco.gforge.inria.fr/

CHAPTER

ONE

INTRODUCTION

COmparing Continuous Optimisers (**COCO**) is a tool for benchmarking algorithms for black-box optimisation. COCO facilitates systematic experimentation in the field of continuous optimization.

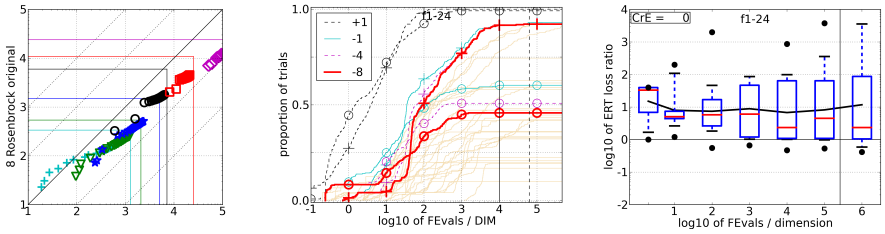


Figure 1.1: The pictures show (a) a scatterplot of run lengths (log10-scale) comparing two algorithms on a single function, (b) the run length distribution of two algorithms for different target difficulties (1e1, 1e-1, 1e-4, 1e-8) on a set of 24 functions and (c) box-whisker plots of the loss ratios of the expected run length compared to best (shortest) observed expected run length in BBOB-2009.

COCO has been used in two consecutive workshops of the international conference **GECCO** in 2009 (BBOB-2009) and 2010 (BBOB-2010).

These Black-Box Optimisation Benchmarking (BBOB) workshops compile:

- the submissions of *83* articles
- presenting the results of *55* algorithms.

COCO provides:

1. an **experimental framework** for testing the algorithms,
2. **post-processing facilities** for generating publication quality figures and tables,
3. LaTeX templates of articles which present the figures and tables in a single document.

The practitioner in continuous optimization who wants to benchmark one or many algorithms has to download COCO, interface the algorithm(s) to call the test functions and use the post-processing tools.

The COCO software is composed of two parts:

1. an interface available in different programming languages which allows to run and log experiments on multiple test functions; testbeds of functions (noisy and noiseless) are provided
2. a Python tool for generating figures and tables

The typical user of COCO simply has to render the interface of the considered algorithm compatible with COCOs interface of the objective/fitness function implementation, run the provided main script (most presumably several

times with increasing allowed maxfunevals until the waiting time becomes infeasible) and invoke the postprocessing on the generated data.

CHAPTER

TWO

INSTALLATION

To install COCO, provided that the minimal requirements are met (see *Requirements* below), you only need to:

1. download the *first* archive `bbobexpXX.XX.tar.gz` from the COCO download page.
2. extract the archive

To check that COCO is running correctly, please proceed to *First Time Using COCO*.

2.1 Requirements

2.1.1 Running Experiments

The interface for running experiments is provided in C/C++, Matlab/Gnu Octave, Java and Python. An algorithm to be tested with COCO has to be adapted to work with either of these versions.

Our code has been tested with:

- C/C++ gcc 4.2.1 and higher, the code provided is ANSI C,
- Matlab R2008a
- Java 1.6, the code uses the Java Native Interface to call a library created from the C/C++ version of the code
- Python 2.6

2.1.2 Python and Post-Processing

The requirements below are mainly for the post-processing tool in COCO.

Note: All packages are easily available for 32-bit Python 2.X.

Using 64-bit Python 2.X is possible but not straightforward. Numpy and Matplotlib do not work with Python 3.X.

- python 2.6 or later but **not python 3!**
- numpy 1.2.1 or later
- matplotlib 0.99.1.1 or later

Tables and articles aggregating figures and tables are generated using LaTeX

How do I install on Windows?

For installing Python under Windows, please go to the download page and download `python-2.X.X.msi` (the default Windows is for Win 32 architecture). This file requires the Microsoft Installer, which is a part of Windows XP and later releases. Numpy and Matplotlib can be installed with the standard `.exe` files which are respectively:

- `numpy-X-win32-superpack-python2.X.exe` and,
- `matplotlib-X.win32-py2.X.exe`.

How do I install on Linux?

In most Linux distributions, Python is already part of the installation. If not, use your favorite package manager to install Python (package name: python). The same needs to be done for Numpy (python-numpy) and Matplotlib (python-matplotlib) and their dependencies.

How do I install on Mac OS X?

Mac OS X comes with Python pre-installed but if your version of Python is 2.4 or earlier, you might need to upgrade.

You then need to download and install Numpy and Matplotlib:

- `numpy-X-py2.X-python.org-macosx10.X.dmg`
- `matplotlib-X-python.org-32bit-py2.X-macosx10.X.dmg`

Distributions exist: they provide a Python installation and a bundle of packages which may be relevant for your use. An example of such distribution is the Enthought python distribution.

CHAPTER

THREE

UNDERSTANDING COCO

This section presents and explains the motives behind COCO. Also it presents the guidelines to produce an article using COCO (provided for the submissions to the BBOB workshops, see *Black-Box Optimization Benchmarking Workshops*).

3.1 What is the purpose of COCO?

Quantifying and comparing performance of optimization algorithms is one 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.

COCO provides tools for most of this tedious task:

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.

3.2 Symbols, Constants, and Parameters

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

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

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

`Ntrial` is the number of trials for each single setup, i.e. each function and dimensionality. The performance is evaluated over all trials.

D search space dimensionalities used for all functions.

3.3 Black-Box Optimization Benchmarking Workshops

COCO has been used in two consecutive workshops of the international conference **GECCO** in 2009 (BBOB-2009) and 2010 (BBOB-2010).

For these workshops, a testbed of 24 noiseless functions and another of 30 noisy functions were provided. We provide documents describing all functions at http://coco.gforge.inria.fr/doku.php?id=downloads

For the workshops, some constants were set:

$\texttt{Ntrial} = 15$

$D = 2; 3; 5; 10; 20; 40$

$\Delta f = 10^{-8}$

Also, the `Ntrial` runs were done on different instances of the functions.

3.3.1 Rationale for the Choice of Ntrial = 15

The parameter `Ntrial` determines the minimal measurable success rate and influences the overall necessary CPU time. Compared to a standard setup for testing stochastic search procedures, we have chosen a small value for nruns. Consequently, within the same CPU-time budget, single trials can be longer and conduct more function evaluations (until f_{target} is reached). If the algorithm terminates before f_{target} is reached, longer trials can be trivially achieved by independent multistarts. Because these multistarts are conducted within each trial, more sophisticated restart strategies are feasible. Finally, 15 trials are sufficient to make relevant performance differences statistically significant. [1]

3.3.2 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, besides that larger target values are invariably less difficult to reach. Functions might be easy to solve up to a function value of 1 and become intricate for smaller target values. The chosen value for the final f_{target} is somewhat arbitrary and reasonable values would change by simple modifications in the function definition. The performance evaluation will consider a wide range of different target function values to reach, all being larger or equal to the final $f_{\mathrm{target}} = f_{\mathrm{opt}} + 10^{-8}$.

3.4 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/C++, Java, MATLAB/Octave and Python). On each function and for each dimensionality `Ntrial` trials are carried out (see also *Rationale for the Choice of Ntrial = 15*). Different function *instances* can be used.

[1] If the number of trials is chosen much larger, even tiny, irrelevant performance differences become statistically significant.

3.4.1 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 of BBOB are defined everywhere in $\mathbb{R}^D$ and have their global optimum in $[-5, 5]^D$. Most BBOB functions have their global optimum in the range $[-4, 4]^D$ 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 function value difference Δf (final target precision), in order to implement effective termination mechanisms (which should also prevent early termination)
5. the target function value f_{target} is provided for conclusive termination of trials, in order to reduce the overall CPU requirements. The target function value is not intended to be used as algorithm input otherwise.

Based on these input parameters, the parameter setting and initialization of the algorithm is entirely left to the user. As a consequence, the setting shall be identical for all benchmark functions of one testbed (the function identifier or any known characteristics of the function are not meant to be input to the algorithm, see also Section *Parameter Setting and Tuning of Algorithms*).

3.4.2 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 function up to the final f_{target} [2]. 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 affects 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.

Independent restarts do not change the main performance measure, *expected running time*, (ERT, see Appendix *Performance Measure: Expected Running Time*). Independent restarts mainly improve the reliability and "visibility" of the measured value. For example, using a fast algorithm with a small success probability, say 5% (or 1%), chances are that not a single of 15 trials is successful. With 10 (or 90) independent restarts, the success probability will increase to 40% and the performance of (here out of 15) are desirable to accomplish a stable performance measurement. This reasoning remains valid for any target function value (different values will be 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 in [Auger:2005b], [harik1999parameter].

Choosing different setups mimics what might be done in practice. All restart mechanisms are finally considered as part of the algorithm under consideration.

3.5 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) of the BBOB testbed 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.

[2] The easiest functions of BBOB 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 $f_{\mathrm{target}} = f_{\mathrm{opt}} + 10^{-8}$ is expected.

The chosen setup, coding language, compiler and computational architecture for conducting these experiments are described.

For CPU-inexpensive algorithms the timing might mainly reflect the time spent in function `fgeneric`.

3.6 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 should be described and the approximate number of tested parameter settings should be given. Whether or not all functions were approached with the very same parameter setting (which might well depend on the dimensionality, see Section *Input to the Algorithm and Initialization*) should be stated clearly and the *crafting effort* should be given for each dimension (see below). The crafting effort is zero, if the setting was identical for all functions.

In general, we strongly discourage the *a priori* use of function-dependent parameter settings (crafting effort larger than zero). In other words, we do not consider the function ID or any function characteristics (like separability, multimodality, ...) as input parameter to the algorithm (see also Section *Input to the Algorithm and Initialization*). 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 *Termination Criteria and Restarts*), 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 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].

Nevertheless, in the case that, in one dimension $K > 1$ *different* parameter settings were used, the procedure of how to come up with the different settings should be explained. The settings and the functions, where they were used, should be given together with the entropy measure *crafting effort* (see also [DEbook2006], [price1997dev] [4]) for each dimensionality D:

$$\mathrm{CrE} = -\sum_{k=1}^{K} \frac{n_k}{n} \ln\left(\frac{n_k}{n}\right)$$

$n = \sum_{k=1}^{K} n_k$ is the number of functions in the testbed

n_k is the number of functions, where the parameter setting with index k was used, for $k = 1, \ldots, K$.

When a single parameter setting was used for all functions, as recommended, the crafting effort is $\mathrm{CrE} = \sum_{k=1}^{1} \frac{n}{n} \ln\left(\frac{n}{n}\right) = 0$ [5]

3.6.1 Performance Measure: Expected Running Time

We advocate performance measures that are:

- quantitative, ideally with a ratio scale (opposed to interval or ordinal scale) [6] and with a wide variation (i.e. for example with values ranging not only between 0.98 and 1)

[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] Our definition differs from [DEbook2006], [price1997dev] in that it is independent of the number of adjusted parameters. Only the number of the different settings used is relevant.}

[5] We give another example: say, in 5-D all functions were optimized with the same parameter setting. In 10-D the first 14 functions were approached with one parameter setting and the remaining 10 functions with a second one (no matter how many parameters were changed). In 20-D the first 10 functions were optimized with one parameter setting, functions 11–13 and functions 23–24 were optimized with a second setting, and the remaining 9 functions 14–22 were optimized with a third setting. The crafting effort computes independently for each dimension in 5-D to $\mathrm{CrE}_5 = 0$, in 10-D to $\mathrm{CrE}_{10} = -\left(\frac{14}{24}\ln\frac{14}{24} + \frac{10}{24}\ln\frac{10}{24}\right) \approx 0.679$, and in 20-D to $\mathrm{CrE}_{20} = -\left(\frac{10}{24}\ln\frac{10}{24} + \frac{5}{24}\ln\frac{5}{24} + \frac{9}{24}\ln\frac{9}{24}\right) \approx 1.06$

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

- 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

For these reasons we use the *expected running time* (ERT, introduced in [price1997dev] as ENES and analyzed in [Auger:2005] 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:

$$\begin{aligned} \mathrm{ERT}(f_\mathrm{target}) &= \mathrm{RT_S} + \frac{1-p_\mathrm{s}}{p_\mathrm{s}}\,\mathrm{RT_{US}} && (3.1)\\ &= \frac{p_\mathrm{s}\mathrm{RT_S} + (1-p_\mathrm{s})\mathrm{RT_{US}}}{p_\mathrm{s}} && (3.2)\\ &= \frac{\#\mathrm{FEs}(f_\mathrm{best} \geq f_\mathrm{target})}{\#\mathrm{succ}} && (3.3) \end{aligned}$$

where the *running times* $\mathrm{RT_S}$ and $\mathrm{RT_{US}}$ 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 $\#\mathrm{FEs}(f_\mathrm{best}(\mathrm{FE}) \geq f_\mathrm{target})$ 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:

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

The $\#\mathrm{succ}$ denotes the number of successful trials. ERT estimates the expected running time to reach f_target [Auger:2005], as a function of f_target. In particular, $\mathrm{RT_S}$ 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.

3.6.2 Bootstrapping

The ERT computes a single measurement from a data sample set (in our case from `Ntrial` optimization runs). Bootstrapping [efron1993ib] 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) [Auger:2005], [DBLP:conf/gecco/AugerR09]_. The distribution of the bootstrapped running times is, besides its displacement, a good approximation of the true distribution. We provide some percentiles of the bootstrapped distribution.

3.6.3 Fixed-Cost versus Fixed-Target Scenario

Two different approaches for collecting data and making measurements from experiments are schematically depicted in Figure *Horizontal vs Vertical View*.

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 can be pictured as drawing a vertical line on the convergence graphs (see Figure *Horizontal vs Vertical View* 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 *Horizontal vs Vertical View* where the line is depicted in blue).

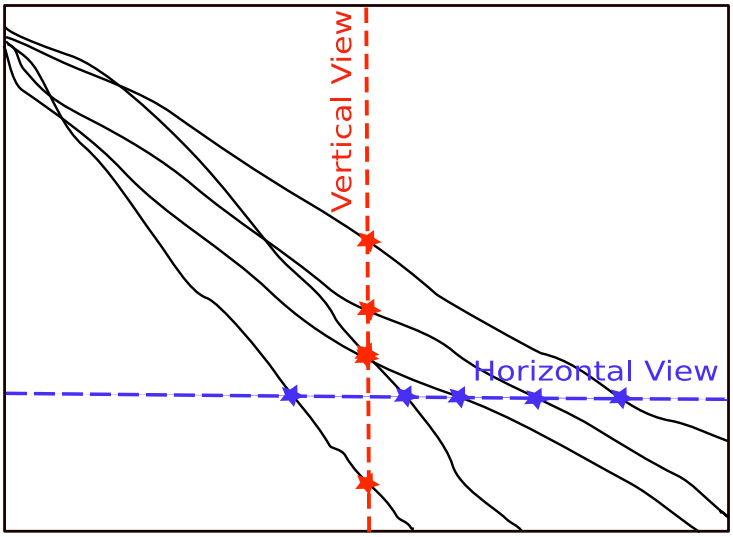


Figure 3.1: Horizontal vs Vertical View
Illustration of fixed-cost view (vertical cuts) and fixed-target view (horizontal cuts). Black lines depict the best function value plotted versus number of function evaluations.

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 two/ten/hundred times faster than Algorithm B in solving this problem (i.e. 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.

3.6.4 Empirical Cumulative Distribution Functions

We exploit the "horizontal and vertical" viewpoints introduced in the last Section *Fixed-Cost versus Fixed-Target Scenario*. In Figure *ECDF* we plot the ECDF (Empirical Cumulative Distribution Function) [7] of the intersection point values (stars in Figure *Horizontal vs Vertical View*).

A cutting line in Figure *Horizontal vs Vertical View* corresponds to a "data" line in Figure *ECDF*, where 450 (30 x 15) convergence graphs are evaluated. For example, the thick red graph in Figure *ECDF* shows on the left the distribution of the running length (number of function evaluations) [hoos1998eva] for reaching precision $\Delta f = 10^{-8}$ (horizontal cut). The graph 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

[7] The empirical (cumulative) distribution function $F : \mathbb{R} \to [0, 1]$ 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.

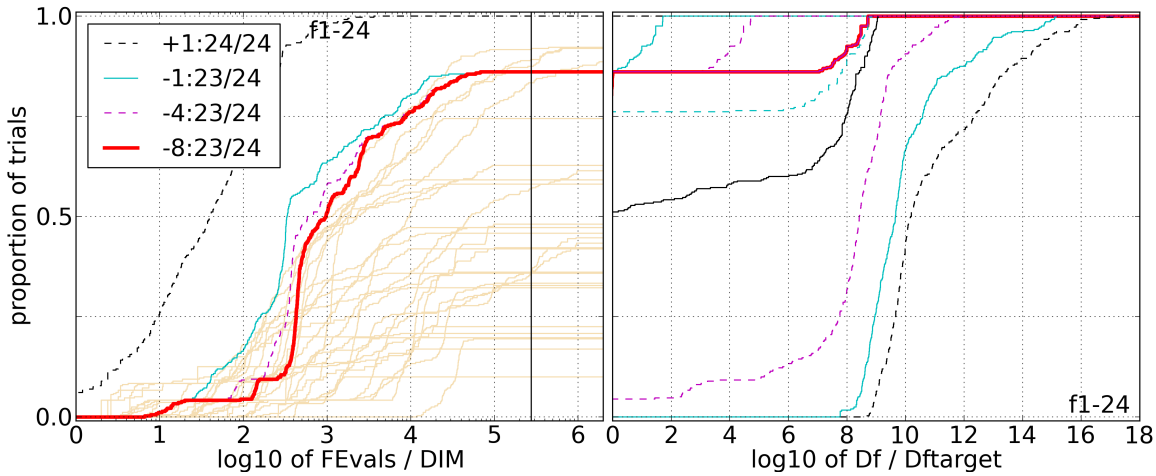


Figure 3.2: ECDF

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 Figure *Horizontal vs Vertical View*. 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. Data for algorithms submitted for BBOB 2009 and $\Delta f = 10^{-8}$ are represented in the background in light brown. 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).

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).

CHAPTER

FOUR

FIRST TIME USING COCO

The following describes the workflow with COCO:

4.1 Running Experiments

For more details, see *Running Experiments with COCO*.

The code for running experiments with COCO is available in C/C++, Matlab/GNU Octave, Java and Python.

`exampleexperiment` and `exampletiming` are provided in each language. These files demonstrate how the code is supposed to be run: the first for doing an experiment over a set of functions, the second for measuring the time complexity of an optimizer.

Content of `exampleexperiment.py` (Python):

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Runs an entire experiment for benchmarking PURE_RANDOM_SEARCH on a testbed.

CAPITALIZATION indicates code adaptations to be made.
This script as well as files bbobbenchmarks.py and cocoexp.py need to be
in the current working directory.

"""

import time
import numpy as np
import cocoexp
import bbobbenchmarks as bn

datapath = 'PUT_MY_BBOB_DATA_PATH'
opt = dict(algid='PUT ALGORITHM NAME',
           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)
```

```
def run_optimizer(fun, dim, maxfunevals, ftarget=-np.Inf):
    """start the optimizer, allowing for some preparation.
    This implementation is an empty template to be filled

    """
    # prepare
    x_start = 8. * np.random.rand(dim) - 4

    # call
    PURE_RANDOM_SEARCH(fun, x_start, maxfunevals, ftarget)

def PURE_RANDOM_SEARCH(fun, x, maxfunevals, ftarget):
    """samples new points uniformly randomly in [-5,5]^dim and evaluates
    them on fun until maxfunevals or ftarget is reached, or until
    1e8 * dim function evaluations are conducted.

    """
    dim = len(x)
    maxfunevals = min(1e8 * dim, maxfunevals)
    popsize = min(maxfunevals, 200)
    fbest = np.inf

    for iter in range(0, int(np.ceil(maxfunevals / popsize))):
        xpop = 10. * np.random.rand(popsize, dim) - 5.
        fvalues = fun(xpop)
        idx = np.argsort(fvalues)
        if fbest > fvalues[idx[0]]:
            fbest = fvalues[idx[0]]
            xbest = xpop[idx[0]]
        if fbest < ftarget:  # task achieved
            break

    return xbest

minfunevals = 'dim + 2' # PUT MINIMAL SENSIBLE NUMBER OF EVALUATIONS for a restart
maxrestarts = 10000     # SET to zero if algorithm is entirely deterministic

t0 = time.time()
np.random.seed(int(t0))

e = cocoexp.Logger(datapath, **opt)
for dim in (2, 3, 5, 10, 20, 40):  # small dimensions first, for CPU reasons
    for f_name in bn.nfreenames: # or bn.noisynames
        for iinstance in range(1, 16):
            e.setfun(*bn.instantiate(f_name, iinstance=iinstance))

            # independent restarts until maxfunevals or ftarget is reached
            for restarts in range(0, maxrestarts + 1):
                run_optimizer(e.evalfun, dim,  eval(maxfunevals) - e.evaluations,
                              e.ftarget)
                if (e.fbest < e.ftarget
                    or e.evaluations + eval(minfunevals) > eval(maxfunevals)):
                    break

            e.finalizerun()

            print('  f%d in %d-D, instance %d: FEs=%d with %d restarts, '
                  'fbest-ftarget=%.4e, elapsed time [h]: %.2f'
```

```
                % (f_name, dim, iinstance, e.evaluations, restarts,
                   e.fbest - e.ftarget, (time.time()-t0)/60./60.))

        print '      date and time: %s' % (time.asctime())
    print '---- dimension %d-D done ----' % dim
```

To run the script in `exampleexperiment.py` (it takes only about a minute), execute the following command from a command line of your operating system:

```
$ python path_to_postproc_code/exampleexperiment.py

    f1 in 2-D, instance 1: FEs=20 with 0 restarts, fbest-ftarget=1.6051e+00, elapsed time [h]: 0.00
  ...
    f24 in 40-D, instance 15: FEs=400 with 0 restarts, fbest-ftarget=1.1969e+03, elapsed time [h]: 0
      date and time: ...
  ---- dimension 40-D done ----

$
```

This generates experimental data in folder `PUT_MY_BBOB_DATA_PATH`.

To run your own experiments, you will need to:

1. modify the method `run_optimizer` to call your optimizer.
2. customize the experiment information: put the optimizer name instead of `'PUT ALGORITHM NAME'`, add a description instead of `'PUT MORE DETAILED INFORMATION, PARAMETER SETTINGS ETC'`
3. after a first test run, successively increase the maximum number of function evaluations `maxfunevals`, subject to the available CPU resources

4.2 Post-Processing

For more details, see *Post-Processing with COCO*.

To post-process the experimental data in folder `PUT_MY_BBOB_DATA_PATH`, execute the python post-processing script from a command line of your operating system:

```
$ python path_to_postproc_code/bbob_pproc/rungeneric.py PUT_MY_BBOB_DATA_PATH

  BBOB Post-processing: will generate output data in folder ppdata
    this might take several minutes.
  BBOB Post-processing: will generate post-processing data in folder ppdata/PUT_MY_BBOB_DATA_PATH
    this might take several minutes.
  Loading best algorithm data from BBOB-2009...  done.
  Scaling figures done.
  TeX tables (draft) done. To get final version tables, please use the -f option
  ECDF graphs done.
  Please input crafting effort value for noiseless testbed:
    CrE = 0.
  ERT loss ratio figures and tables done.
  Output data written to folder ppdata/PUT_MY_BBOB_DATA_PATH.

$
```

This generates output figure and table files in `ppdata/PUT_MY_BBOB_DATA_PATH`.

4.3 Write/Compile an Article

LaTeX files are provided as template articles. These files include the figures and tables generated by the post-processing during LaTeX compilation.

The template corresponding to the post-processed figures and tables for the data of *one* optimizer on the *noiseless* testbed is `template1generic.tex`.

We assume that the output folder of the post-processing and the template file are in the current working directory. The path of the figures and tables is set by editing the file and completing the following line:

```
\newcommand{\algfolder}{}
```

with the name of the subfolder in `ppdata`:

```
\newcommand{\algfolder}{PUT_MY_BBOB_DATA_PATH}
```

Then, compile the file using LaTeX, for example:

```
$ latex template1generic

  This is pdfTeX...
  entering extended mode
  (./template1generic.tex
  ...
  Output written on template1generic.dvi (...).
  Transcript written on template1generic.log.

$ dvipdf template1generic
$
```

This generates document `template1generic.pdf` in the current working directory.

CHAPTER

FIVE

RUNNING EXPERIMENTS WITH COCO

COCO provides an interface for running experiments. This interface has been ported in different languages:

- `fgeneric` in Matlab/GNU Octave, C/C++ and Java,
- `cocoexp` in Python.

5.1 `exampleexperiment` and `exampletiming`

In each language, two example scripts are provided. Below are the example scripts in Matlab/GNU Octave:

- `exampleexperiment` runs an experiment on one testbed,

```
% 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.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 = [1:15]  % first 15 function instances
      fgeneric('initialize', ifun, iinstance, datapath, opt);

      % independent restarts until maxfunevals or ftarget is reached
      for restarts = 0:maxrestarts
        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
```

- `exampletiming` runs the CPU-timing experiment.

```
% 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', 24, 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
```

5.1.1 Matlab/GNU Octave

The above example scripts run a complete experiment. The whole interface for running experiments is defined via the function `fgeneric.m`. `fgeneric` is *initialized* with a function number, instance number and data path. `fgeneric` can then be called to evaluate the test function. The end of a run is signaled by calling `fgeneric` with the 'finalize' keyword.

5.1.2 C/C++

The interface for running experiments relies on functions such as `fgeneric_initialize`, `fgeneric_finalize`, `fgeneric_ftarget`. Also, the evaluation function is `fgeneric_evaluate` for a single vector and `fgeneric_evaluate_vector` for an array of vectors with input arguments `XX` is the concatenation of the `np` candidate vectors, `np` the number of individual vectors, and `result` is an array of size `np` which will contain the resulting function values.

A specific folder structure is needed for running an experiment. This folder structure can be obtained by un-tarring the archive `createfolders.tar.gz` and renaming the output folder or alternatively by executing the Python script `createfolders.py` before executing any experiment program. Make sure `createfolders.py` is in your current working directory and from the command-line simply execute:

```
$ python createfolders.py FOLDERNAME
FOLDERNAME was created.
```

The code provided can be compiled in C or C++.

5.1.3 Python

The interface for running an experiment is `cocoexp` which is used within `exampleexperiment.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Runs an entire experiment for benchmarking PURE_RANDOM_SEARCH on a testbed.

CAPITALIZATION indicates code adaptations to be made.
This script as well as files bbobbenchmarks.py and cocoexp.py need to be
in the current working directory.

"""

import time
import numpy as np
import cocoexp
import bbobbenchmarks as bn

datapath = 'PUT_MY_BBOB_DATA_PATH'
opt = dict(algid='PUT ALGORITHM NAME',
           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)

def run_optimizer(fun, dim, maxfunevals, ftarget=-np.Inf):
    """start the optimizer, allowing for some preparation.
    This implementation is an empty template to be filled

    """
```

```
    # prepare
    x_start = 8. * np.random.rand(dim) - 4

    # call
    PURE_RANDOM_SEARCH(fun, x_start, maxfunevals, ftarget)

def PURE_RANDOM_SEARCH(fun, x, maxfunevals, ftarget):
    """samples new points uniformly randomly in [-5,5]^dim and evaluates
    them on fun until maxfunevals or ftarget is reached, or until
    1e8 * dim function evaluations are conducted.

    """
    dim = len(x)
    maxfunevals = min(1e8 * dim, maxfunevals)
    popsize = min(maxfunevals, 200)
    fbest = np.inf

    for iter in range(0, int(np.ceil(maxfunevals / popsize))):
        xpop = 10. * np.random.rand(popsize, dim) - 5.
        fvalues = fun(xpop)
        idx = np.argsort(fvalues)
        if fbest > fvalues[idx[0]]:
            fbest = fvalues[idx[0]]
            xbest = xpop[idx[0]]
        if fbest < ftarget:  # task achieved
            break

    return xbest

minfunevals = 'dim + 2' # PUT MINIMAL SENSIBLE NUMBER OF EVALUATIONS for a restart
maxrestarts = 10000     # SET to zero if algorithm is entirely deterministic

t0 = time.time()
np.random.seed(int(t0))

e = cocoexp.Logger(datapath, **opt)
for dim in (2, 3, 5, 10, 20, 40):  # small dimensions first, for CPU reasons
    for f_name in bn.nfreenames: # or bn.noisynames
        for iinstance in range(1, 16):
            e.setfun(*bn.instantiate(f_name, iinstance=iinstance))

            # independent restarts until maxfunevals or ftarget is reached
            for restarts in range(0, maxrestarts + 1):
                run_optimizer(e.evalfun, dim,  eval(maxfunevals) - e.evaluations,
                              e.ftarget)
                if (e.fbest < e.ftarget
                    or e.evaluations + eval(minfunevals) > eval(maxfunevals)):
                    break

            e.finalizerun()

            print('  f%d in %d-D, instance %d: FEs=%d with %d restarts, '
                  'fbest-ftarget=%.4e, elapsed time [h]: %.2f'
                  % (f_name, dim, iinstance, e.evaluations, restarts,
                     e.fbest - e.ftarget, (time.time()-t0)/60./60.))

        print '      date and time: %s' % (time.asctime())
    print '---- dimension %d-D done ----' % dim
```

5.2 Testing New Functions

We describe here how to use `cocoexp` to record experiment on functions outside of the BBOB testbeds.

Note: This feature is only available in Python for the moment.

Example: log experiment using the Nelder-Mead simplex algorithm (`scipy.optimize.fmin`) on the sphere function. The following commands from the Python Interpreter does 15 runs of the Nelder-Mead simplex algorithm on the 2-D sphere functions. The data is recorded in folder `data` in the current working directory.

```
>>> from pylab import *
>>> import cocoexp as co
>>> import scipy.optimize as so
>>> f = lambda x: sum(i ** 2 for i in x) # function definition
>>> e = co.Logger(datapath='data', algid='Nelder-Mead simplex',
                  comments='x0 uniformly sampled in [0, 1]^2, '
                           'default settings')
>>> for i in range(15): # 15 repetitions
...     e.setfun(fun=f, fopt=0., funId='sphere', iinstance='0')
...     so.fmin(e.evalfun, x0=rand(2)) # algorithm call
...     e.finalizerun()
(<bound method Logger.evalfun of <cocoexp.Logger object at [...]>>, 1e-08)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: [...]
         Function evaluations: [...]
array([...])
[...]
```

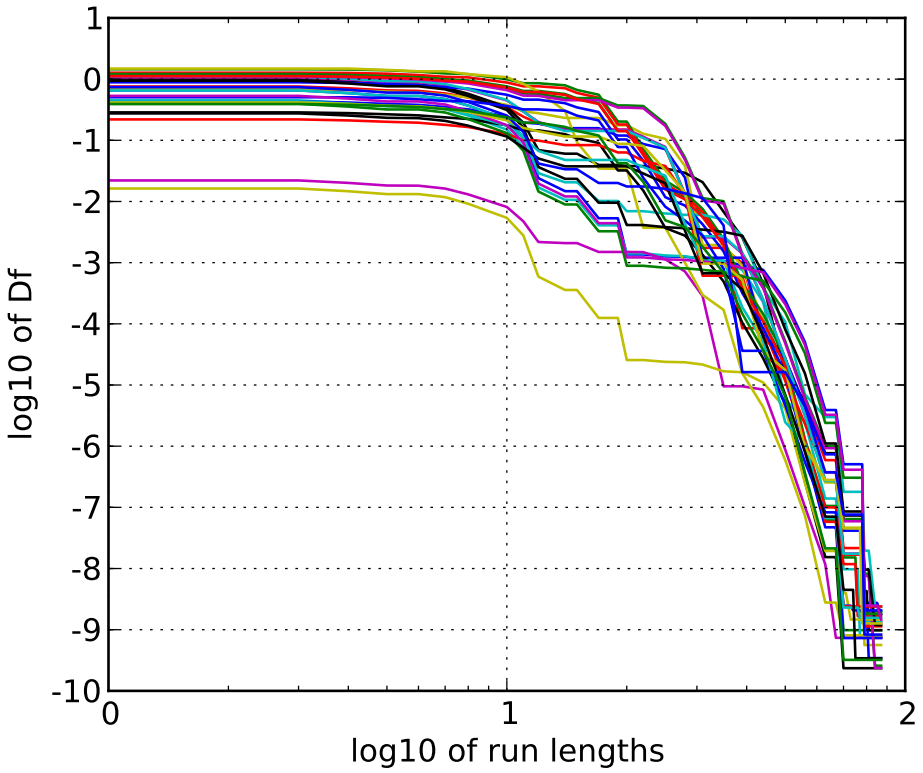
5.2.1 Testing Functions with Parameter

Note: This feature is only available in Python for the moment.

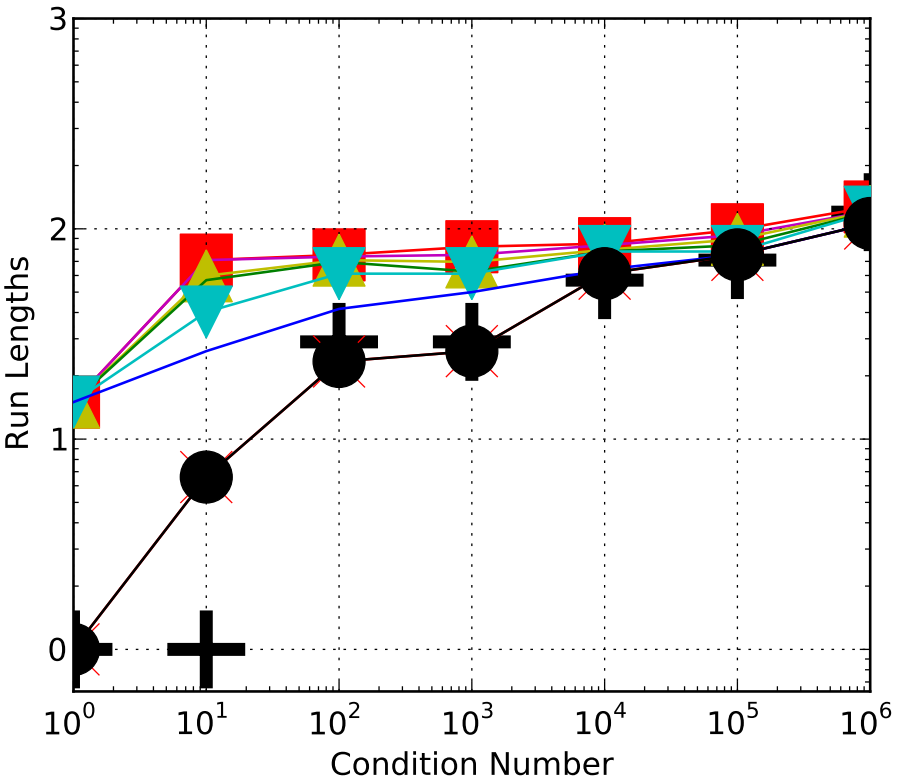
Example: log experiment using the BFGS algorithm (`scipy.optimize.fmin`) on the ellipsoid function with different condition numbers.

The following commands from the Python Interpreter does 15 runs of the BFGS algorithm on the 2-D sphere functions. The data is recorded in folder `data` in the current working directory and generate

```
>>> from pylab import *
>>> import cocoexp as co
>>> import scipy.optimize as so
>>> import numpy as np

>>> e = co.Logger(datapath='ellipsoid', algid='BFGS',
                  comments='x0 uniformly sampled in [0, 1]^5, default settings')
>>> cond_num = 10 ** np.arange(0, 7)
>>> for c in cond_num:
...     f = lambda x: np.sum(c ** np.linspace(0, 1, len(x)) * x ** 2)
...     # function definition: these are term-by-term operations
...     for i in range(5): # 5 repetitions
...         e.setfun(fun=f, fopt=0., funId='ellipsoid', iinstance=0,
...                  condnum=c)
...         so.fmin_bfgs(e.evalfun, x0=np.random.rand(5)) # algorithm call
```




```
...         e.finalizerun()
(<bound method Logger.evalfun of <cocoexp.Logger object at [...]>>, 1e-08)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: [...]
         Function evaluations: [...]
         Gradient evaluations: [...]
array([...])
[...]

>>>
```


3
2
1
0
Run Lengths
10^0
10^1
10^2
10^3
10^4
10^5
10^6
Condition Number

CHAPTER

SIX

POST-PROCESSING WITH COCO

6.1 Overview of the bbob_pproc Package

We present here the content of the latest version of the bbob_pproc package (version 10.7).

rungeneric.py is the main interface of the package that performs different routines listed below,

rungeneric1.py post-processes data from *one* single algorithm and outputs figures and tables included in the templates template1generic.tex, noisytemplate1generic.tex,

rungeneric2.py post-processes data from *two* algorithms using modules from bbob_pproc.comp2 and outputs comparison figures and tables included in the template template2generic.tex, noisytemplate2generic.tex,

rungenericmany.py post-processes data from 1 or more algorithms [1] using modules from bbob_pproc.compall and outputs comparison figures and tables included in the template, template3generic.tex, noisytemplate3generic.tex

genericsettings.py defines generic settings for the output figures and tables,

bwsettings.py defines generic settings with figures in black and white,

grayscalesettings.py defines generic settings with grayscale figures,

bbob_pproc.pproc defines the classes bbob_pproc.pproc.DataSetList and bbob_pproc.pproc.DataSet which are the main data structures that we use to gather the experimental raw data,

bbob_pproc.dataoutput contain routine to output instances of bbob_pproc.pproc.DataSet in Python-formatted data files,

bbob_pproc.readalign, bbob_pproc.bootstrap contain routines for the post-processing of the raw experimental data,

bbob_pproc.pptex defines some routines for generating TeXtables,

bbob_pproc.ppfig defines some routines for generating figures,

bbob_pproc.ppfigdim, bbob_pproc.pptable, bbob_pproc.pprldistr, bbob_pproc.pplogloss are used to produce figures and tables presenting the results of one algorithm,

bbob_pproc.compall is a sub-package which contains modules for the comparison of the performances of algorithms, routines in this package can be called using the interface of rungenericmany.py,

bbob_pproc.comp2 is a sub-package which contains modules for the comparison of the performances of two algorithms, routines in this package can be called using the interface of rungeneric2.py.

[1] for more than ten algorithm, the template shall be modified,

6.2 Standard Use of the bbob_pproc Package

The main interface is in `rungeneric.py` and behaves differently depending on the number of folders given as input arguments (each corresponding to the data of a different algorithm).

If one folder, `DATAPATH`, containing all data generated by the experiments for *one* algorithm is in the current working directory, the following command executes the post-processing:

```
$ python path_to_postproc_code/bbob_pproc/rungeneric.py DATAPATH

BBOB Post-processing: will generate output data in folder ppdata
  this might take several minutes.
BBOB Post-processing: will generate post-processing data in folder ppdata/DATAPATH
  this might take several minutes.
Loading best algorithm data from BBOB-2009...  done.
Scaling figures done.
TeX tables (draft) done. To get final version tables, please use the -f option
ECDF graphs done.
Please input crafting effort value for noiseless testbed:
  CrE = 0.
ERT loss ratio figures and tables done.
Output data written to folder ppdata/DATAPATH.

$
```

The above command create the folder with the default name `ppdata/DATAPATH` in the current working directory, which contain the post-processed data in the form of figures and LaTeX files for the tables. This process might take a few minutes. Help on the use of the `rungeneric.py` script, which corresponds to the execution of method `bbob_pproc.rungeneric.main`, can be obtained by the command `python path_to_postproc_code/bbob_pproc/rungeneric.py -h`.

To run the post-processing directly from a Python shell, the following commands need to be executed:

```
>>> import bbob_pproc as bb
>>> bb.rungeneric.main('DATAPATH'.split())

 BBOB Post-processing: will generate output data in folder ppdata
   this might take several minutes.
 ...
 Output data written to folder ppdata/DATAPATH.

>>>
```

This first command requires that the path to the package `bbob_pproc` is in the search path of Python (e.g. the current folder).

The resulting folder `ppdata/DATAPATH` now contains a number of `tex`, `eps`, `pdf` files.

6.2.1 Comparison of Algorithms

The sub-packages `bbob_pproc.compall` and `bbob_pproc.comp2` provide facilities for the generation of tables and figures comparing the performances of algorithms.

The post-processing works with data folders as input argument, with each folder corresponding to the data of an algorithm. Supposing you have the folders `ALG1`, `ALG2` and `ALG3` containing the data of algorithms ALG1, ALG2 and ALG3, you will need to execute from the command line:

```
$ python path_to_postproc_code/bbob_pproc/rungeneric.py ALG1 ALG2 ALG3

 BBOB Post-processing: will generate output data in folder ppdata
   this might take several minutes.
 ...
 Output data written to folder ppdata.
```

This assumes the folders `ALG1`, `ALG2` and `ALG3` are in the current working directory. In this case, the folders contain a number of files with the `pickle` extension which contain Python-formatted data or the raw experiment data with the `info`, `dat` and `tdat` extensions. Running the aforementioned command will generate the folder `ppdata` containing comparison figures and tables.

Outputs appropriate to the comparison of only two algorithms can be obtained by executing the following from the command line:

```
$ python path_to_postproc_code/bbob_pproc/rungeneric.py ALG0 ALG1

 BBOB Post-processing: will generate output data in folder ppdata
   this might take several minutes.
 ...
 Output data written to folder ppdata.
```

This assumes the folders `ALG0` and `ALG1` are in the current working directory. Running the aforementioned command will generate the folder `ppdata` containing the comparison figures.

Note: Instead of using the `rungeneric.py` interface, the user can directly use the sub-routines `rungeneric1.py`, `rungeneric2.py` or `rungenericmany.py` to generate some post-processing output. These sub-routines are to be used in the same way as `rungeneric.py`.

6.2.2 Use of the LaTeX Templates

LaTeX files are provided as template articles. These files compile the figures and tables generated by the post-processing in a single document.

- `template1generic.tex`
- `template2generic.tex`
- `template3generic.tex`
- `noisytemplate1generic.tex`
- `noisytemplate2generic.tex`
- `noisytemplate3generic.tex`
- `template1ecj.tex`
- `template2ecj.tex`
- `template3ecj.tex`
- `noisytemplate1ecj.tex`
- `noisytemplate2ecj.tex`
- `noisytemplate3ecj.tex`

The templates have different layout and do not present different figures and tables. The `noisytemplate*` and `template*` files aggregate results respectively on the noisy and noiseless testbeds from BBOB.

`*template*generic.tex` use the default article class from LaTeX, whereas `*template*ecj.tex` use the style for submissions to the Evolutionary Computation Journal.

The purpose of `*template1*` templates is to present the results of one algorithm on a BBOB testbed (obtained with `bbob_pproc.rungeneric1`). The templates `*template2*` and `*template3*` respectively for the comparison of two (`bbob_pproc.rungeneric2`) and two or more algorithms (`bbob_pproc.rungenericmany`).

The usual way of using these templates would be to:

1. copy the necessary template and associated style files (if necessary) in the current working directory, ie. in the same location as the output folder of the post-processing,
2. edit the template, in particular by pointing LaTeX to the output folder of the post-processing,
3. compile with LaTeX.

The path of the figures and tables is set by editing the file and completing the following line:

```
\newcommand{\bbobdatapath}{ppdata/}
```

and replace `ppdata/` with the name of the folder containing the output figures and tables from the post-processing.

Note: `bbob_pproc.rungeneric` will generate comparison figures and tables at the base of the output folder. It will **also generate individual results** of each algorithms which data were provided. Those will be found in subfolders.

The `*template1*` templates can be used by pointing to one of those subfolders. **Alternatively**, the line `\newcommand{\bbobdatapath}{ppdata/}` can be set to the main output folder and the line `\newcommand{\algfolder}{}` to the name of one of the subfolders.

Example: If the output folder is `ppdata`, the subfolder is `ppdata/ALG1`, The template can be edited so that we have: `\newcommand{\bbobdatapath}{ppdata/}` and `\newcommand{\algfolder}{ALG1}`

The templates are compiled using LaTeX, for instance:

```
$ latex template1generic

  This is pdfTeX...
  entering extended mode
  (./template1generic.tex
  ...
  Output written on template1generic.dvi (...).
  Transcript written on template1generic.log.

$ dvipdf template1generic
$
```

This generates document `template1generic.pdf` in the current working directory.

6.3 `bbob_pproc` from the Python Interpreter

The package `bbob_pproc` can also be used interactively. A Matlab-like interaction is even when using the Python interpreter in Pylab mode (see below).

6.3.1 Pylab

Installing `matplotlib` also installs `pylab`. `pylab` provides a Matlab-style interface to the functionalities in `numpy` and `matplotlib`. To start pylab mode from the Python interpreter, typing:

```
>>> from pylab import *
>>>
```

Some differences with the Python interpreter are:

- the `matplotlib.pyplot` and `numpy` packages are already imported,
- the interactive mode of matplotlib is already on.

Note: We can but recommend installing and using iPython which is an environment enhancing the Python interpreter for an interactive use. The pylab mode can then be started from the command line by typing:

```
$ ipython -pylab

Python 2...
Type "copyright", "credits" or "license" for more information.

IPython 0.X -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]:
```

Some differences with the Python interpreter are:

- the prompt is different, the iPython prompt is: In [1]: whereas the Python prompt is >>>
- some system commands are directly available: ls, cd, ...

6.3.2 Pre-requisites

For this session, we assume that:

- the current working directory is the folder `bbob.vXX.XXX/python` extracted from one of the archives containing `bbob_pproc`,
- the following files are downloaded and unarchived in the current working directory:
  - http://coco.lri.fr/BBOB2009/rawdata/BIPOP-CMA-ES_hansen_noiseless.tar.gz
  - http://coco.lri.fr/BBOB2009/pythondata/BIPOP-CMA-ES.tar.gz
  - http://coco.lri.fr/BBOB2009/pythondata/NEWUOA.tar.gz

The following describes a step-by-step interactive session with `bbob_pproc`.

6.3.3 Start of the Session

First, we start the Python interpreter. This is usually done by typing python from the command line:

```
$ python
Python 2....
...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To start pylab mode, type:

```
>>> from pylab import *
>>>
```

The `bbob_pproc` package is then loaded into memory.

```
>>> import bbob_pproc as bb
>>> help(bb)
Help on package bbob_pproc:

NAME
    bbob_pproc - COmparing Continuous Optimisers (COCO) post-processing software

FILE
    [...]/python/bbob_pproc/__init__.py

DESCRIPTION
    This package is meant to generate output figures and tables for the
    benchmarking of continuous optimisers in the case of black-box
    optimisation.
    The post-processing tool takes as input data from experiments and
    generates outputs that will be used in the generation of the LaTeX-
    formatted article summarizing the experiments.

    The 'main' function of this package is the main in bbob_pproc.rungeneric
    This function allows to use the post-processing through a command-line
    interface.

    To obtain more information on the use of this package from the python
    interpreter, assuming this package has been imported as bb, type:
    help(bb.cococommands)

PACKAGE CONTENTS
    bbob2010 (package)
    bestalg
    bootstrap
    bwsettings
    changeAlgIdAndComment
    cococommands
    comp2 (package)
    compall (package)
    dataoutput
    determineFtarget
    determineFtarget3
    findfiles
    firstsession
    genericsettings
    grayscalesettings
    ppfig
    ppfigdim
    pplogloss
    pprldistr
```

```
    pproc
    ppsingle
    pptable
    pptex
    readalign
    readindexfiles
    rungeneric
    rungeneric1
    rungeneric2
    rungenericmany

FUNCTIONS
    main(argv=None)
        Main routine for post-processing data from COCO.

        This routine will call sub-routine rungeneric1 for each input
        arguments and either sub-routines rungeneric2 (2 input arguments) or
        rungenericmany (more than 2) for the input arguments altogether.

        The output figures and tables are included in
        * template1generic.tex, template1ecj.tex, noisytemplate1generic.tex,
        noisytemplate1ecj.tex for single algorithm results on the noise-free
        and noisy testbeds respectively
        * template2generic.tex, template2ecj.tex, noisytemplate2generic.tex,
        noisytemplate2ecj.tex for showing the comparison of 2 algorithms
        * template3generic.tex, template3ecj.tex, noisytemplate3generic.tex,
        noisytemplate3ecj.tex for showing the comparison of more than 2
        algorithms.
        These files needs to be copied in the current working directory and
        edited so that the LaTeX commands \bbobdatapath and \algfolder (for
        xxxtemplate1xxx.tex) need to be set to the output folder of the
        post-processing. Compiling the template file with LaTeX should then
        produce a document.

        Keyword arguments:
        argv -- list of strings containing options and arguments. If not
        provided, sys.argv is accessed.

        argv must list folders containing COCO data files. Each of these
        folders should correspond to the data of ONE algorithm.

        Furthermore, argv can begin with facultative option flags.

            -h, --help

                display this message

            -v, --verbose

                verbose mode, prints out operations.

            -o, --output-dir OUTPUTDIR

                change the default output directory ('ppdata') to OUTPUTDIR

        Exceptions raised:
        Usage -- Gives back a usage message.
```

```
        Examples:

        * Calling the rungeneric.py interface from the command line:

            $ python bbob_pproc/rungeneric.py -v AMALGAM BIPOP-CMA-ES

        * Loading this package and calling the main from the command line
          (requires that the path to this package is in python search path):

            $ python -m bbob_pproc.rungeneric -h

        This will print out this help message.

        * From the python interpreter (requires that the path to this
          package is in python search path):

            >> import bbob_pproc as bb
            >> bb.rungeneric.main('-o outputfolder folder1 folder2'.split())

        This will execute the post-processing on the data found in folder1
        and folder2. The -o option changes the output folder from the
        default ppdata to outputfolder.

DATA
    __all__ = ['comp2', 'compall', 'main', 'ppfigdim', 'pplogloss', 'pprld...
    __version__ = '10.7'

VERSION
    10.7

>>>
```

Commands in `bbob_pproc` can now be accessed by prepending `bb.` to command names.

Typing `help(bb.cococommands)` provides some help on the use of `bbob_pproc` in the Python interpreter.

```
>>> help(bb.cococommands)
Help on module bbob_pproc.cococommands in bbob_pproc:

NAME
    bbob_pproc.cococommands - Module for using COCO from the (i)Python interpreter.

FILE
    [...]/python/bbob_pproc/cococommands.py

DESCRIPTION
    For all operations in the Python interpreter, it will be assumed
    that the package has been imported as bb, just like it is done
    in the first line of the examples below.

    The main data structures used in COCO are DataSet, which corresponds
    to data of one algorithm on one problem, and DataSetList, which is
    for collections of DataSet instances. Both classes DataSetList and
    DataSet are implemented in module bb.pproc.

    Examples:

    >>> import bbob_pproc as bb # load bbob_pproc
```

```
    * Load a data set, assign to variable ds:

    >>> ds = bb.load('BIPOP-CMA-ES_hansen_noiseless/bbobexp_f2.info')
    Processing BIPOP-CMA-ES_hansen_noiseless/bbobexp_f2.info.
    ...
    Processing ['BIPOP-CMA-ES_hansen_noiseless/data_f2/bbobexp_f2_DIM40.tdat']: 15/15 trials found.

    * Get some information on a DataSetList instance:

    >>> print ds
    [DataSet(cmaes V3.30.beta on f2 2-D), ..., DataSet(cmaes V3.30.beta on f2 40-D)]
    >>> bb.info(ds)
    6 data set(s)
    Algorithm(s): cmaes V3.30.beta
    Dimension(s): 2, 3, 5, 10, 20, 40
    Function(s): 2
    Max evals: 75017
    Df       |     min       10      med       90      max
    --------|----------------------------------------------
    1.0e+01 |      55      151     2182    49207    55065
    1.0e+00 |     124      396     2820    56879    59765
    1.0e-01 |     309      466     2972    61036    66182
    1.0e-03 |     386      519     3401    67530    72091
    1.0e-05 |     446      601     3685    70739    73472
    1.0e-08 |     538      688     4052    72540    75010

CLASSES
    [...]

FUNCTIONS
    info(dsList)
        Display more info on an instance of DatasetList.

    load(filename)
        Create a DataSetList instance from a file which filename is provided.

        Input argument filename can be a single info filename, a single pickle
        filename or a folder name.

    pickle(dsList)
        Pickle a DataSetList.

    systeminfo()
        Display information on the system.

DATA
    __all__ = ['load', 'info', 'pickle', 'systeminfo', 'DataSetList', 'Dat...

>>>
```

Data is loaded into memory with the `bbob_pproc.cococommands.load` function:

```
>>> ds = bb.load('BIPOP-CMA-ES_hansen_noiseless/bbobexp_f2.info')
Processing BIPOP-CMA-ES_hansen_noiseless/bbobexp_f2.info.
    [...]
Processing ['BIPOP-CMA-ES_hansen_noiseless/data_f2/bbobexp_f2_DIM40.tdat']: 15/15 trials found.
>>> ds
[DataSet(cmaes V3.30.beta on f2 2-D), ..., DataSet(cmaes V3.30.beta on f2 40-D)]
>>>
```

The variable `ds` stores an instance of class `bbob_pproc.pproc.DataSetList` represented between square brackets. This instance in `ds` has a single element which is an instance of `bbob_pproc.pproc.DataSet`.

```
>>> help(ds)
Help on DataSetList in module bbob_pproc.pproc object:

class DataSetList(__builtin__.list)
 |  List of instances of DataSet with some useful slicing functions.
 |
 |  Will merge data of DataSet instances that are identical (according
 |  to function __eq__ of DataSet).
 |
 |  Method resolution order:
 |      DataSetList
 |      __builtin__.list
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, args=[], verbose=True)
 |      Instantiate self from a list of inputs.
 |
 |      Keyword arguments:
 |      args -- list of strings being either info file names, folder containing
 |              info files or pickled data files.
 |      verbose -- controls verbosity.
 |
 |      Exception:
 |      Warning -- Unexpected user input.
 |      pickle.UnpicklingError
 |
 |  append(self, o)
 |      Redefines the append method to check for unicity.
 |
 |  dictByAlg(self)
 |      Returns a dictionary of DataSetList instances by algorithm.
 |
 |      The resulting dict uses algId and comment as keys and the
 |      corresponding slices of DataSetList as values.
 |
 |  dictByDim(self)
 |      Returns a dictionary of DataSetList instances by dimensions.
 |
 |      Returns a dictionary with dimension as keys and the
 |      corresponding slices of DataSetList as values.
 |
 |  dictByFunc(self)
 |      Returns a dictionary of DataSetList instances by functions.
 |
 |      Returns a dictionary with the function id as keys and the
 |      corresponding slices of DataSetList as values.
 |
 |  dictByFuncGroup(self)
 |      Returns a dictionary of DataSetList instances by function groups.
 |
 |      Returns a dictionary with function group names as keys and the
 |      corresponding slices of DataSetList as values.
 |
 |  dictByNoise(self)
```

```
 |      Returns a dictionary splitting noisy and non-noisy entries.
 |
 |  extend(self, o)
 |      Extend a DataSetList with elements.
 |
 |      This method is implemented to prevent problems since append was
 |      superseded. This method could be the origin of efficiency issue.
 |
 |  info(self, opt='all')
 |      Display some information onscreen.
 |
 |      Keyword arguments:
 |      opt -- changes size of output, can be 'all' (default), 'short'
 |
 |  pickle(self, outputdir=None, verbose=True)
 |      Loop over self to pickle each elements.
 |
 |  processIndexFile(self, indexFile, verbose=True)
 |      Reads in an index file information on the different runs.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from __builtin__.list:
 |
    [...]

>>> help(ds[0])
Help on instance of DataSet in module bbob_pproc.pproc:

class DataSet
 |  Unit element for the BBOB post-processing.
 |
 |  One unit element corresponds to data with given algId and comment, a
 |  funcId, and dimension.
 |
 |  Class attributes:
 |      funcId -- function Id (integer)
 |      dim -- dimension (integer)
 |      indexFiles -- associated index files (list of strings)
 |      dataFiles -- associated data files (list of strings)
 |      comment -- comment for the setting (string)
 |      targetFuncValue -- target function value (float)
 |      algId -- algorithm name (string)
 |      evals -- data aligned by function values (array)
 |      funvals -- data aligned by function evaluations (array)
 |      maxevals -- maximum number of function evaluations (array)
 |      finalfunvals -- final function values (array)
 |      readmaxevals -- maximum number of function evaluations read from
 |                      index file (array)
 |      readfinalFminusFtarget -- final function values - ftarget read
```

```
|                                from index file (array)
|      pickleFile -- associated pickle file name (string)
|      target -- target function values attained (array)
|      ert -- ert for reaching the target values in target (array)
|      itrials -- list of numbers corresponding to the instances of the
|                 test function considered (list of int)
|      isFinalized -- list of bool for if runs were properly finalized
|
|  evals and funvals are arrays of data collected from N data sets.
|  Both have the same format: zero-th column is the value on which the
|  data of a row is aligned, the N subsequent columns are either the
|  numbers of function evaluations for evals or function values for
|  funvals.
|
|  Methods defined here:
|
|  __eq__(self, other)
|      Compare indexEntry instances.
|
|  __init__(self, header, comment, data, indexfile, verbose=True)
|      Instantiate a DataSet.
|
|      The first three input argument corresponds to three consecutive
|      lines of an index file (info extension).
|
|      Keyword argument:
|      header -- string presenting the information of the experiment
|      comment -- more information on the experiment
|      data -- information on the runs of the experiment
|      indexfile -- string for the file name from where the information come
|      verbose -- controls verbosity
|
|  __ne__(self, other)
|
|  __repr__(self)
|
|  computeERTfromEvals(self)
|      Sets the attributes ert and target from the attribute evals.
|
|  createDictInstance(self)
|      Returns a dictionary of the instances.
|
|      The key is the instance id, the value is a list of index.
|
|  detERT(self, targets)
|      Determine the expected running time to reach target values.
|
|      Keyword arguments:
|      targets -- list of target function values of interest
|
|      Output:
|      list of expected running times corresponding to the targets
|
|  detEvals(self, targets)
|      Determine the number of evaluations to reach target values.
|
|      Keyword arguments:
|      targets -- list of target function values of interest
```

```
 |
 |      Output:
 |      list of arrays each corresponding to one value in targets
 |
 |  generateRLData(self, targets)
 |      Determine the running lengths for reaching the target values.
 |
 |      Keyword arguments:
 |      targets -- list of target function values of interest
 |
 |      Output:
 |      dict of arrays, one array has for first element a target
 |      function value smaller or equal to the element of inputtargets
 |      considered and has for other consecutive elements the
 |      corresponding number of function evaluations.
 |
 |  info(self)
 |      Return some text info to display onscreen.
 |
 |  mMaxEvals(self)
 |      Returns the maximum number of function evaluations.
 |
 |  nbRuns(self)
 |      Returns the number of runs.
 |
 |  pickle(self, outputdir=None, verbose=True)
 |      Save DataSet instance to a pickle file.
 |
 |      Saves the instance of DataSet to a pickle file. If not specified
 |      by argument outputdir, the location of the pickle is given by
 |      the location of the first index file associated to the DataSet.
 |      This method will overwrite existing files.
 |
 |  splitByTrials(self, whichdata=None)
 |      Splits the post-processed data arrays by trials.
 |
 |      Returns a two-element list of dictionaries of arrays, the key of
 |      the dictionary being the instance id, the value being a smaller
 |      post-processed data array corresponding to the instance id.

>>>
```

The `bbob_pproc.cococommands.load` function also works on pickle files or folders. For instance:

```
>>> ds = bb.load('BIPOP-CMA-ES/ppdata_f002_20.pickle')
Unpickled BIPOP-CMA-ES/ppdata_f002_20.pickle.
>>> ds = bb.load('BIPOP-CMA-ES_hansen_noiseless')
Searching in BIPOP-CMA-ES_hansen_noiseless ...
    [...]
Processing ['BIPOP-CMA-ES_hansen_noiseless/data_f9/bbobexp_f9_DIM40.tdat']: 15/15 trials found.
>>>
```

To use wildcards for loading only part of the files in a folder, the `glob` package included in Python standard library can be used:

```
>>> import glob
>>> ds = bb.load(glob.glob('BIPOP-CMA-ES/ppdata_f002_*.pickle'))
Unpickled BIPOP-CMA-ES/ppdata_f002_02.pickle.
Unpickled BIPOP-CMA-ES/ppdata_f002_03.pickle.
```

```
Unpickled BIPOP-CMA-ES/ppdata_f002_05.pickle.
Unpickled BIPOP-CMA-ES/ppdata_f002_10.pickle.
Unpickled BIPOP-CMA-ES/ppdata_f002_20.pickle.
Unpickled BIPOP-CMA-ES/ppdata_f002_40.pickle.
>>>
```

Some information on loaded data can be obtained with the `bbob_pproc.cococommands.info` function:

```
>>> ds = bb.load('BIPOP-CMA-ES/ppdata_f002_20.pickle')
Unpickled BIPOP-CMA-ES/ppdata_f002_20.pickle.
>>> bb.info(ds) # display information on DataSetList ds
1 data set(s)
Algorithm(s): cmaes V3.30.beta
Dimension(s): 20
Function(s): 2
Max evals: 20690
Df      |     min       10      med       90      max
--------|----------------------------------------------
1.0e+01 |   10875    11049    13483    16152    16826
1.0e+00 |   13329    14061    15155    17421    17675
1.0e-01 |   15392    15581    16481    18382    18463
1.0e-03 |   16706    17282    18162    19081    19083
1.0e-05 |   17854    17855    18939    19629    19831
1.0e-08 |   18726    18749    19757    20599    20678
>>>
```

The actual data in a `bbob_pproc.pproc.DataSet` instance can be displayed on-screen:

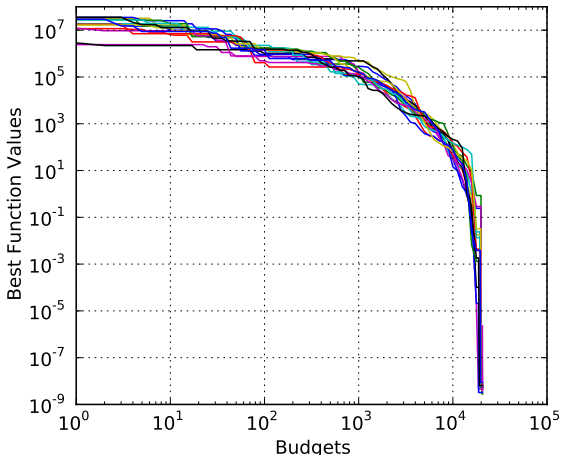
```
>>> d = ds[0] # store the first element of ds in d for convenience
>>> d.funvals
array([[  1.00000000e+00,   1.91046048e+07,   3.40744851e+07, ...,
          1.65690802e+07,   3.57969327e+07,   2.94479644e+07],
       [  2.00000000e+00,   1.83130834e+07,   3.40744851e+07, ...,
          1.65690802e+07,   3.57969327e+07,   2.94479644e+07],
       [  3.00000000e+00,   1.83130834e+07,   3.40744851e+07, ...,
          1.65690802e+07,   3.57969327e+07,   1.51653130e+07],
       ...,
       [  2.04740000e+04,   4.83164257e-07,   2.78740231e-09, ...,
          8.42007353e-09,   6.33018260e-09,   8.59924398e-09],
       [  2.06060000e+04,   4.79855089e-09,   2.78740231e-09, ...,
          8.42007353e-09,   6.33018260e-09,   8.59924398e-09],
       [  2.06900000e+04,   4.79855089e-09,   2.78740231e-09, ...,
          8.42007353e-09,   6.33018260e-09,   8.59924398e-09]])
>>> budgets = d.funvals[:, 0] # stores first column in budgets
>>> funvals = d.funvals[:, 1:] # stores all other columns in funvals
>>>
```

The columns from one to last in attribute `bbob_pproc.pproc.DataSet.funvals` of each correspond to a different run of one algorithm on a given test problem. The values in each of these column are the best precision attained over time. The zero-th column are the function evaluations for which the values were attained.

The graphs of the evolution of the best precision over time for each single run can be displayed as follows :

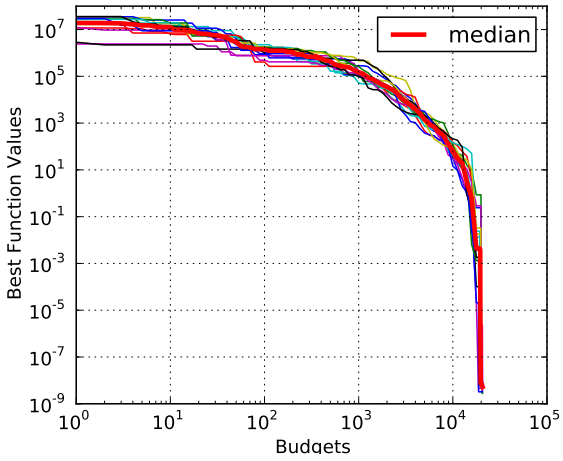
```
>>> nbrows, nbruns = funvals.shape
>>> for i in range(0, nbruns):
...     loglog(budgets, funvals[:, i])
...
[<matplotlib.lines.Line2D object at 0x1[...]>]
    [...]
[<matplotlib.lines.Line2D object at 0x1[...]>]
```

```
>>> grid()
>>> xlabel('Budgets')
<matplotlib.text.Text object at 0x1[...]>
>>> ylabel('Best Function Values')
<matplotlib.text.Text object at 0x1[...]>
```



The median of the function values for each number of function evaluations versus the number of function evaluations can be displayed as well.

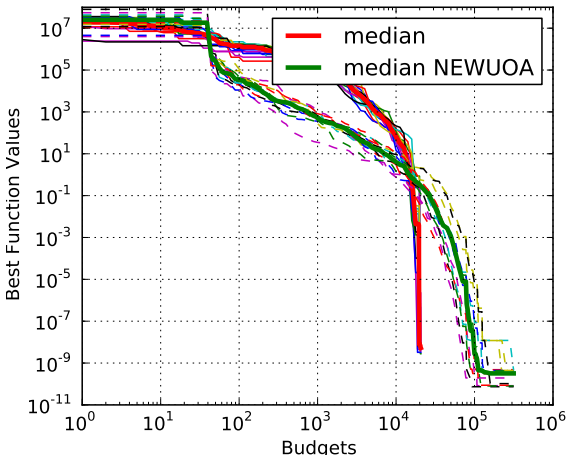
```
>>> loglog(budgets, median(funvals, axis=1), linewidth=3, color='r',
...         label='median CMA-ES')
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> legend() # display legend
<matplotlib.legend.Legend object at 0x1[...]>
```



The options `linewidth` and `color` makes the line bold and red. The `matplotlib.pyplot.legend` function displays a legend for the figure which uses the `label` option.

Another data set can be displayed for comparison:

```
>>> ds1 = bb.load('NEWUOA/ppdata_f002_20.pickle')
Unpickled NEWUOA/ppdata_f002_20.pickle.
>>> d1 = ds1[0]
>>> budgets1 = d1.funvals[:, 0]
>>> funvals1 = d1.funvals[:, 1:]
>>> for i in range(0, funvals1.shape[1]):
...     loglog(budgets1, funvals1[:, i], linestyle='--')
...
[<matplotlib.lines.Line2D object at 0x1[...]>]
    [...]
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> loglog(budgets1, median(funvals1, axis=1), linewidth=3,
...        color='g', label='median NEWUOA')
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> legend() # updates legend
<matplotlib.legend.Legend object at 0x1[...]>
```



A figure can be saved using the `matplotlib.pyplot.savefig` function:

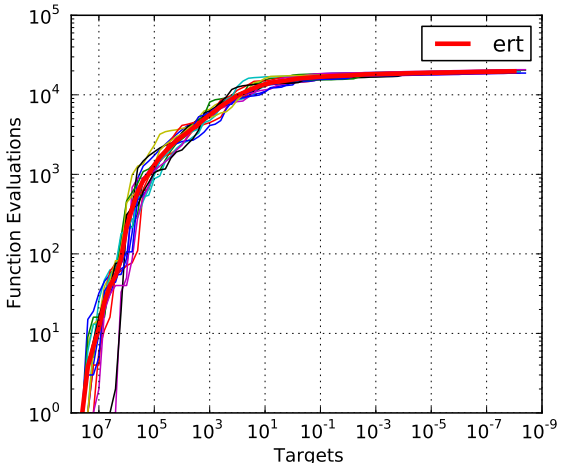
```
>>> savefig('examplefigure')  # save active figure as image
```

An image file is then created. The format of the output image file depends on the extension of the file name provided to `matplotlib.pyplot.savefig` or the default settings of `matplotlib`.

In the previous figures, it is the evolution of the best function values versus time which is considered, in other words the best function values given a budget (vertical view). Instead, we can consider the horizontal view: determining the run lengths for attaining a target precision. This defines the Expected Run Time performance measure. Please refer to Experimental Set-Up Document for more details.

```
>>> targets = d.evals[:, 0]
>>> evals =  d.evals[:, 1:]
>>> nbrows, nbruns = evals.shape
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]
>>> for i in range(0, nbruns):
...     loglog(targets, evals[:, i])
...
[<matplotlib.lines.Line2D object at 0x1[...]
    [...]
[<matplotlib.lines.Line2D object at 0x1[...]>]
```

```
>>> grid()
>>> xlabel('Targets')
<matplotlib.text.Text object at 0x1[...]>
>>> ylabel('Function Evaluations')
<matplotlib.text.Text object at 0x1[...]>
>>> loglog(d.target[d.target>=1e-8], d.ert[d.target>=1e-8], lw=3,
...         color='r', label='ert')
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> gca().invert_xaxis() # xaxis from the easiest to the hardest
>>> legend() # this operation updates the figure with the inverse axis.
<matplotlib.legend.Legend object at 0xa84592c>
```

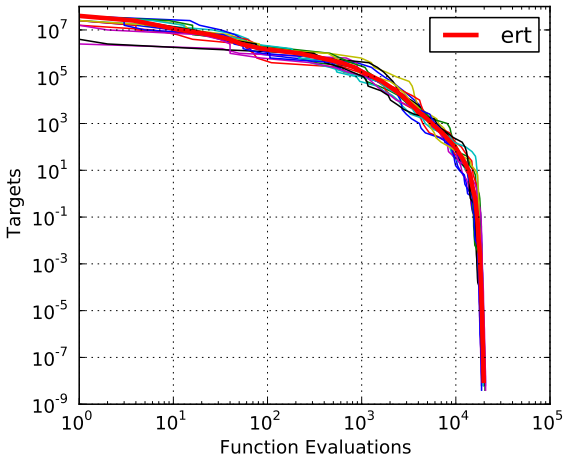


If we swap the abscissa and the ordinate of the previous figure, we can directly compare the horizontal view and vertical view figures:

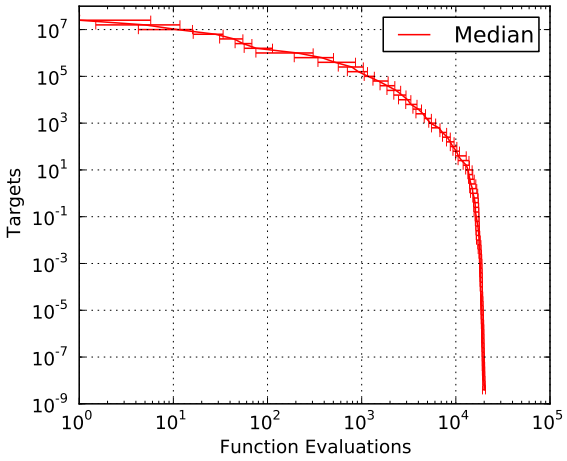
```
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]>
>>> for i in range(0, nbruns):
...     loglog(evals[:, i], targets)
...
[<matplotlib.lines.Line2D object at 0x1[...]>]
    [...]
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> grid()
>>> xlabel('Function Evaluations')
<matplotlib.text.Text object at 0x1[...]>
>>> ylabel('Targets')
<matplotlib.text.Text object at 0x1[...]>
>>> loglog(d.ert[d.target>=1e-8], d.target[d.target>=1e-8], lw=3,
...         color='r', label='ert')
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> legend()
```

The process for displaying the median function values and interquartile ranges for different targets is described thereafter:

```
>>> figure() # open a new figure
<matplotlib.figure.Figure object at 0x1[...]>
>>> from bbob_pproc.bootstrap import prctile
>>> q = array(list(prctile(i, [25, 50, 75]) for i in evals))
```




```
>>> xmed = q[:, 1]
>>> xlow = xmed - q[:, 0]
>>> xhig = q[:, 2] - xmed
>>> xerr = vstack((xlow, xhig))
>>> errorbar(xmed, targets, xerr=xerr, color='r', label='CMA-ES')
(<matplotlib.lines.Line2D object at 0x1[...]>, [...], [<matplotlib.collections.LineCollection object
>>> grid()
>>> xscale('log')
>>> yscale('log')
>>> xlabel('Function Evaluations')
<matplotlib.text.Text object at 0x1[...]>
>>> ylabel('Targets')
<matplotlib.text.Text object at 0x1[...]>
```

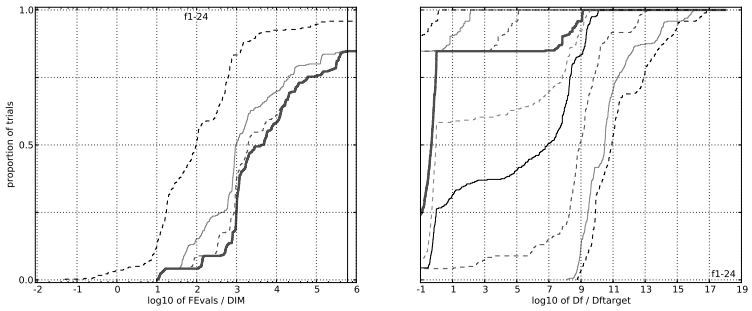


Other types of figures are generated by different modules in `bbob_pproc`. Each of these modules has a `plot` function and a `beautify` function. The `plot` function is used for generating a type of graph. The `beautify` function accommodates the general presentation of the figure to its content.

The modules for generating figures in this manner are `bbob_pproc.pprldistr`, `bbob_pproc.compall.ppperfprof`, `bbob_pproc.ppfigdim`.

The bbob_pproc.pprldistr module generates Empirical Cumulative Distribution Function (ECDF) graphs of the number of evaluations for attaining target precisions and the function values for a given budget:

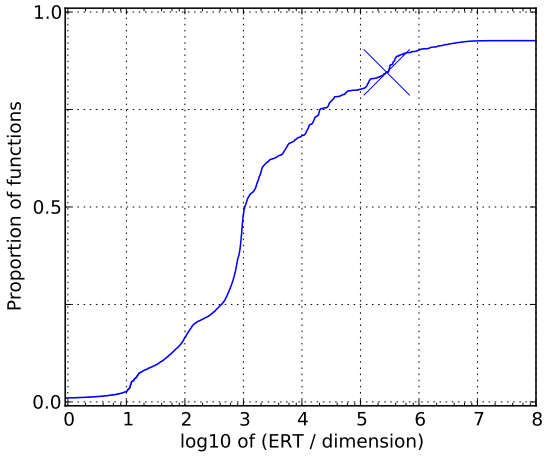
```
>>> help(bb.pprldistr)
Help on module bbob_pproc.pprldistr in bbob_pproc:

NAME
    bbob_pproc.pprldistr - Creates run length distribution figures.

FILE
    [...]/python/bbob_pproc/pprldistr.py

FUNCTIONS
    beautify()
        Format the figure of the run length distribution.

    comp(dsList0, dsList1, valuesOfInterest, isStoringXMax=False, outputdir='', info='default', verbo
        Generate figures of empirical cumulative distribution functions.
        Dashed lines will correspond to ALG0 and solid lines to ALG1.

        Keyword arguments:
        dsList0 -- list of DataSet instances for ALG0.
        dsList1 -- list of DataSet instances for ALG1
        valuesOfInterest -- target function values to be displayed.
        isStoringXMax -- if set to True, the first call BeautifyVD sets the
          globals fmax and maxEvals and all subsequent calls will use these
          values as rightmost xlim in the generated figures.
        outputdir -- output directory (must exist)
        info --- string suffix for output file names.

    main(dsList, valuesOfInterest, isStoringXMax=False, outputdir='', info='default', verbose=True)
        Generate figures of empirical cumulative distribution functions.

        Keyword arguments:
        dsList -- list of DataSet instances to process.
        valuesOfInterest -- target function values to be displayed.
        isStoringXMax -- if set to True, the first call BeautifyVD sets the
          globals fmax and maxEvals and all subsequent calls will use these
          values as rightmost xlim in the generated figures.
        outputdir -- output directory (must exist)
        info --- string suffix for output file names.

        Outputs:
        Image files of the empirical cumulative distribution functions.

    plot(dsList, valuesOfInterest=(10.0, 0.10000000000000001, 0.0001, 1e-08), kwargs={})
        Plot ECDF of final function values and evaluations.
DATA
    __all__ = ['beautify', 'comp', 'main', 'plot']

>>> ds = bb.load(glob.glob('BIPOP-CMA-ES/ppdata_f0*_20.pickle'))
Unpickled BIPOP-CMA-ES/ppdata_f001_20.pickle.
    [...]
Unpickled BIPOP-CMA-ES/ppdata_f024_20.pickle.
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]>
>>> pprldistr.plot(ds)
>>> pprldistr.beautify() # resize the window to view whole figure
```



The `bbob_pproc.compall.ppperfprof` module generates ECDF graphs of the bootstrap distribution of the Expected Running Time for target precisions:

```
>>> help(bb.compall.ppperfprof)
Help on module bbob_pproc.compall.ppperfprof in bbob_pproc.compall:

NAME
    bbob_pproc.compall.ppperfprof - Generates figure of the bootstrap distribution of ERT.

FILE
    [...]/python/bbob_pproc/compall/ppperfprof.py

DESCRIPTION
    The main method in this module generates figures of Empirical
    Cumulative Distribution Functions of the bootstrap distribution of
    the Expected Running Time (ERT) divided by the dimension for many
    algorithms.

FUNCTIONS
    beautify()
        Customize figure presentation.

    main(dictAlg, targets, order=None, plotArgs={}, outputdir='', info='default', verbose=True)
        Generates a figure showing the performance of algorithms.
        From a dictionary of DataSetList sorted by algorithms, generates the
        cumulative distribution function of the bootstrap distribution of
        ERT for algorithms on multiple functions for multiple targets
        altogether.

        Keyword arguments:
        dictAlg -- dictionary of dataSetList instances containing all data
        to be represented in the figure
        targets -- list of target function values
        order -- sorted list of keys to dictAlg for plotting order

    plot(dsList, targets=(1e-08, [...]), rhleg=False, kwargs={})
        Generates a plot showing the performance of an algorithm.

        Keyword arguments:
        dsList -- a DataSetList instance
        targets -- list of target function values
        kwargs

DATA
    __all__ = ['beautify', 'main', 'plot']
```

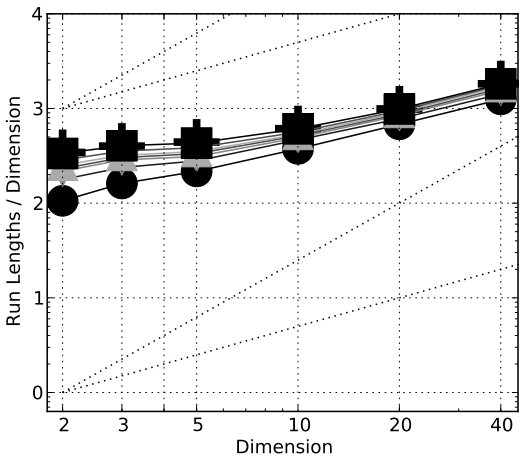
```
>>> ds = bb.load(glob.glob('BIPOP-CMA-ES/ppdata_f0*_20.pickle'))
Unpickled BIPOP-CMA-ES/ppdata_f001_20.pickle.
   [...]
Unpickled BIPOP-CMA-ES/ppdata_f024_20.pickle.
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]>
>>> ppperfprof.plot(ds)
[<matplotlib.lines.Line2D object at 0x1[...]>, <matplotlib.lines.Line2D object at 0x1[...]>]
>>> ppperfprof.beautify()
```



The `bbob_pproc.ppfigdim` module generates scaling figures.

```
>>> from bbob_pproc import ppfigdim
>>> help(ppfigdim)
Help on module bbob_pproc.ppfigdim in bbob_pproc:

NAME
    bbob_pproc.ppfigdim - Generate performance scaling figures.

FILE
    [...]/python/bbob_pproc/ppfigdim.py

FUNCTIONS
    beautify()
        Customize figure presentation.

    main(dsList, _valuesOfInterest, outputdir, verbose=True)
        From a DataSetList, returns a convergence and ERT/dim figure vs dim.

    plot(dsList, _valuesOfInterest=(10, 1, [...], 1e-08))
        From a DataSetList, plot a figure of ERT/dim vs dim.

DATA
    __all__ = ['beautify', 'plot', 'main']

>>> ds = bb.load(glob.glob('BIPOP-CMA-ES/ppdata_f002_*.pickle'))
Unpickled BIPOP-CMA-ES/ppdata_f002_02.pickle.
[...]
Unpickled BIPOP-CMA-ES/ppdata_f002_40.pickle.
```

```
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]>
>>> ppperfprof.plot(ds)
[<matplotlib.lines.Line2D object at 0x1[...]>, [...], <matplotlib.lines.Line2D object at 0x1[...]>]
>>> ppperfprof.beautify()
```



The post-processing tool `bbob_pproc` generates image files and LaTeX tables from the raw experimental data obtained with COCO.

Its main usage is through the command line interface. To generate custom figures, `bbob_pproc` can be used alternatively from the Python interpreter.

CHAPTER

SEVEN

ACKNOWLEDGMENTS

Steffen Finck was supported by the Austrian Science Fund (FWF) under grant P19069-N18. The BBOBies would like to acknowledge:

- Miguel Nicolau for his insights and the help he has provided on the implementation of the C-code,
- Mike Preuss for his implementation of the JNI for using the C-code in Java,
- Petr Posik for his help and feedback overall,
- Alvaro Fialho for providing the C++ version of the code.

BIBLIOGRAPHY

[Auger:2005b] 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.

[harik1999parameter] 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.

[DEbook2006] Vitaliy Feoktistov. *Differential Evolution: In Search of Solutions*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[price1997dev] 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.

[Auger:2005] 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.

[efron1993ib] B. Efron and R. Tibshirani. *An introduction to the bootstrap.* Chapman & Hall/CRC, 1993.

[hoos1998eva] 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.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803