COCO Documentation

Release 15.03

S. Finck, N. Hansen, R. Ros, and A. Auger

November 17, 2015

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.

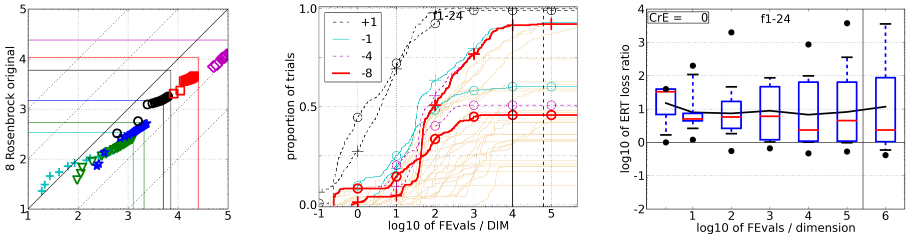


Fig. 1.1: The pictures show (a) a scatterplot of run lengths (log10-scale) comparing two algorithms in different dimensions on a single function, (b) the run length distribution of two algorithms for different target difficulties $\{10, 10^{-1}, 10^{-4}, 10^{-8}\}$ on a set of 24 functions and (c) box-whisker 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 several Black-Box Optimization Benchmarking (BBOB) workshops at the GECCO conference, the first in 2009 (BBOB-2009). Overall around a 100 articles and algorithms have been benchmarked.

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, plug the algorithm(s) into the provided experimental template 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 that can be used in the LaTeX templates.

The typical user of COCO renders the interface of the considered algorithm compatible with COCOs interface of the objective/fitness function, runs the provided main script (most presumably several times with increasing allowed maxfunevals until the waiting time becomes infeasible), invokes the postprocessing on the generated data and latexs the provided template.

An extensive documentation of the **test functions** can be found here.

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 What is the purpose of COCO?.

2.1 Requirements

2.1.1 Running Experiments

The interface for running experiments is provided in C/C++, Matlab/Gnu Octave, Java, R 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 and higher
- 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, 2.7
- R 2.14.2

2.1.2 Python and Post-Processing

The requirements below are for the post-processing tool of COCO that runs in Python.

Needed are

- python 2.6.x or 2.7.x (**not python 3.x**)
- numpy 1.2.1 or later (the version must fit to the python version)
- matplotlib 1.0.1 or later (the version must fit to the python version)

All three are part of

- Anaconda (first choice)
- Enthought python distribution and
- Sage (then `sage` instead of `python` must be used).

Documents and articles that aggregate tables and figures are generated using LaTeX

How do I install on Windows?

For installing Python under Windows, either install Anaconda or Enthought or go to the Python download page and download the Python Windows Installer `python-2.7.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?

Either install Anaconda or Sage or Enthought. Otherwise, Python is usually 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?

Either install Anaconda or Sage or Enthought. Otherwise, Mac OS X comes with Python pre-installed but if your version of Python is 2.5 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`

2.2 Interactive Python

In order to fully enjoy a Python shell, we enable tab-completion on Unix systems like

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
```

or we use IPython. In order to have a matlab-like environment with hundreds of additional commands directly available (including the `plot` command) use

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

or

```
>>> import pylab as pl
```

which provides `pl.plot` etc.

See also *Pylab*.

CHAPTER

THREE

WHAT IS THE PURPOSE OF COCO?

Quantifying and comparing performance of numerical 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.

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, R 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.

The content of `exampleexperiment.py` shows how the experiment is invoked (see Running Experiments with COCO for an example in Matlab/Octave). The last 20-or-so lines define the experimentation loop over all dimension, all functions, and all instances:

```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 fgeneric.py need to be
in the current working directory.

Under unix-like systems:
    nohup nice python exampleexperiment.py [data_path [dimensions [functions [instances]]]] > output

"""
import sys # in case we want to control what to run via command line args
import time
import numpy as np
import fgeneric
import bbobbenchmarks

argv = sys.argv[1:] # shortcut for input arguments
```

```
datapath = 'PUT_MY_BBOB_DATA_PATH' if len(argv) < 1 else argv[0]

dimensions = (2, 3, 5, 10, 20, 40) if len(argv) < 2 else eval(argv[1])
function_ids = bbobbenchmarks.nfreeIDs if len(argv) < 3 else eval(argv[2])
# function_ids = bbobbenchmarks.noisyIDs if len(argv) < 3 else eval(argv[2])
instances = range(1, 6) + range(41, 51) if len(argv) < 4 else eval(argv[3])

opts = 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)
minfunevals = 'dim + 2'  # PUT MINIMAL sensible number of EVALUATIONS before to restart
maxrestarts = 10000      # SET to zero if algorithm is entirely deterministic


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, REPLACE with optimizer to be tested
    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 _ 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

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

f = fgeneric.LoggingFunction(datapath, **opts)
for dim in dimensions:  # small dimensions first, for CPU reasons
    for fun_id in function_ids:
        for iinstance in instances:
            f.setfun(*bbobbenchmarks.instantiate(fun_id, iinstance=iinstance))
            # independent restarts until maxfunevals or ftarget is reached
            for restarts in xrange(maxrestarts + 1):
                if restarts > 0:
                    f.restart('independent restart')  # additional info
                run_optimizer(f.evalfun, dim,  eval(maxfunevals) - f.evaluations,
                              f.ftarget)
                if (f.fbest < f.ftarget
                    or f.evaluations + eval(minfunevals) > eval(maxfunevals)):
                    break

            f.finalizerun()

            print('  f%d in %d-D, instance %d: FEs=%d with %d restarts, '
                  'fbest-ftarget=%.4e, elapsed time [h]: %.2f'
                  % (fun_id, dim, iinstance, f.evaluations, restarts,
                     f.fbest - f.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 datapath, say `'advanced-random-search'` instead of `'PUT_MY_BBOB_DATA_PATH'`, 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 data in folder `advanced-random-search`, execute the python post-processing script from a command line of your operating system:

```
$ python path_to_bbob_pproc/bbob_pproc/rungeneric.py advanced-random-search

  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/advanced-random-search
```

```
    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.
  ERT loss ratio figures and tables done.
  Output data written to folder ppdata/advanced-random-search.

$
```

The `bbob_pproc` folder is located in the python folder of the COCO distribution. This command generates output figures and tables in the folder `ppdata/advanced-random-search`.

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, by default `ppdata`, and the template file are in the current working directory. The path to the figures and tables is set to the default but can be edited in the LaTeX file:

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

Also the name of the algorithm as it appears in figure captions can be edited (by default it is implied from the original data folder).

```
\renewcommand{\algname}{Advanced Random Search}
```

Then, compile the file using LaTeX, for example:

```
$ pdflatex template1generic

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

This generates document `template1generic.pdf` in the current working directory. On some systems, the combination latex and dvipdf might produce much better results.

CHAPTER

FIVE

BLACK-BOX OPTIMIZATION BENCHMARKING PROCEDURE

COCO has been used in several workshops during the GECCO conference since 2009 (BBOB-2009).

For these workshops, a testbed of 24 noiseless functions and another of 30 noisy functions are provided. Descriptions of the functions can be found at http://coco.gforge.inria.fr/doku.php?id=downloads

This section describes the setup of the experimental procedures and their rationales, giving the guidelines to produce an article for a GECCO-BBOB workshop using COCO.

5.1 Symbols, Constants, and Parameters

For the workshops, some constants were set:

$D = 2; 3; 5; 10; 20; 40$ is search space dimensionalities used for all functions.

Ntrial $= 15$ is the number of trials for each single setup, i.e. each function and dimensionality. The performance is evaluated over all trials.

$\Delta f = 10^{-8}$ precision to reach, that is, a difference to the smallest possible function value f_{opt}.

$f_{\mathrm{target}} = f_{\mathrm{opt}} + \Delta f$ is target function value to reach for different Δf values. 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.

The `Ntrial` runs are conducted on different instances of the functions.

5.1.1 Rationale for the Choice of Ntrial = 15

All functions can be instantiated in different "versions" (with different location of the global optimum and different optimal function value). Overall `Ntrial` runs are conducted on different instantiations (in the 2009 setup each instance was repeated three times). The parameter `Ntrial`, the overall number of trials on each function/dimension pair, 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, `Ntrial` $= 15$. Consequently, within the same CPU-time budget, single trials can be longer and conduct more function evaluations (until f_target is reached). If an algorithm terminates before f_target is reached, longer trials can simply be achieved by independent multistarts. Because these multistarts are conducted within each trial, more sophisticated restart strategies are feasible. **Within-trial multistarts never impair the used performance measures and are encouraged.** Finally, 15 trials are sufficient to make relevant performance differences statistically significant. [1]

5.1.2 Rationale for the Choice of f$_\mathrm{target}$

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. We take records for a larger number of predefined target values, defined relative to the known optimal function value f_opt and in principle unbounded from above. The chosen value for the final (smallest) f_target is somewhat arbitrary. Reasonable values can change by simple modifications in the function definition. In order to safely prevent numerical precision problems, the final target is $f_\mathrm{target} = f_\mathrm{opt} + 10^{-8}$.

5.2 Benchmarking Experiment

The real-parameter search algorithm under consideration is run on a testbed of benchmark functions to be minimized (the implementation of the functions is provided in C/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.

The `exampleexperiment.*` code template is provided to run this experiment. For BBOB-2012, the instances are 1 to 5 and 21 to 30.

5.2.1 Input to the Algorithm and Initialization

An algorithm can use the following input:

1. the search space dimensionality D
2. the search domain; all functions 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 can be used for the noise-free and the noisy testbed.
4. the final target precision delta-value $\Delta f = 10^{-8}$ (see above), in order to implement effective termination and restart mechanisms (which should also prevent early termination)
5. the target function value f_target, however provided *only* for conclusive (final) termination of trials, in order to reduce the overall CPU requirements. The target function value must not be used as algorithm input otherwise (not even to trigger within-trial restarts).

[1] If the number of trials is chosen *much* larger, small and therefore irrelevant performance differences become statistically significant.

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, for natural reasons, not allowed as input to the algorithm, see also Section *Parameter Setting and Tuning of Algorithms*).

5.2.2 Termination Criteria and Restarts

Algorithms with any budget of function evaluations, small or large, are considered in the analysis of the results. Exploiting a larger number of function evaluations increases the chance to achieve better function values or even to solve the 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 *Expected Running Time*) to hit a given target. 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 will become visible. At least four to five (here out of 15) successful trials are desirable to accomplish a stable performance measurement. This reasoning remains valid for any target function value (different values are considered in the evaluation).

Restarts either from a previous solution, or with a different parameter setup, for example with different (increasing) population sizes, might be considered, as it has been applied quite successful *[Auger:2005a] [Harik:1999]*.

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

5.3 Time Complexity Experiment

In order to get a rough measurement of the time complexity of the algorithm, the overall CPU time is measured when running the algorithm on f_8 (Rosenbrock function) 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. The chosen setup, coding language, compiler and computational architecture for conducting these experiments are described.

The `exampletiming.*` code template is provided to run this experiment. For CPU-inexpensive algorithms the timing might mainly reflect the time spent in function `fgeneric`.

5.4 Parameter Setting and Tuning of Algorithms

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

On all functions the very same parameter setting must be used (which might well depend on the dimensionality, see Section *Input to the Algorithm and Initialization*). That means, *a priori* use of function-dependent parameter settings is prohibited (since 2012). In other words, the function ID or any function characteristics (like separability, multi-modality, ...) cannot be considered as input parameter to the algorithm. Instead, we encourage benchmarking different

[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.

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].

5.5 Performance Measurement

We advocate performance measures that are:

- quantitative, ideally with a ratio scale (opposed to interval or ordinal scale) [4] and with a wide variation (i.e., for example, with values ranging not only between 0.98 and 1.0)
- well-interpretable, in particular by having a meaning and semantics attached to the numbers
- relevant with respect to the "real world"
- as simple as possible

For these reasons we measure "running times" to reach a target function value, denoted as fixed-target scenario in the following.

5.5.1 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*.

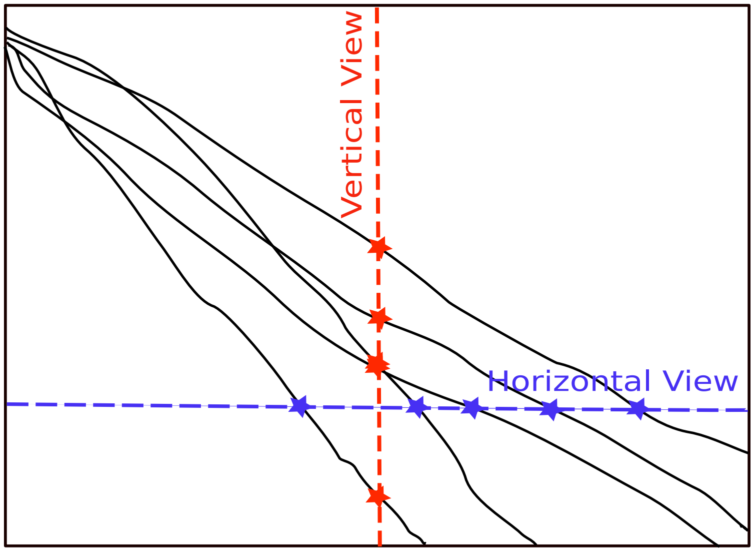


Fig. 5.1: Horizontal vs Vertical View

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.

[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] Wikipedia gives a reasonable introduction to scale types.

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).

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. This, indeed, largely depends on the specific function and on the specific function value reached. Furthermore, for algorithms that are invariant under certain transformations of the function value (for example under order-preserving transformations as algorithms based on comparisons like DE, ES, PSO), fixed-target measures can be made invariant under these transformations by simply choosing different target values while fixed-cost measures require the transformation of all resulting data.

5.5.2 Expected Running Time

We use the *expected running time* (ERT, introduced in *[Price:1997]* as ENES and analyzed in *[Auger:2005b]* as success performance) as most prominent performance measure. The Expected Running Time is defined as 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:

$$\begin{aligned}\mathrm{ERT}(f_\mathrm{target}) &= \mathrm{RT_S} + \frac{1-p_\mathrm{s}}{p_\mathrm{s}}\,\mathrm{RT_{US}} &\quad (5.1)\\ &= \frac{p_\mathrm{s}\mathrm{RT_S} + (1-p_\mathrm{s})\mathrm{RT_{US}}}{p_\mathrm{s}} &\quad (5.2)\\ &= \frac{\#\mathrm{FEs}(f_\mathrm{best} \ge f_\mathrm{target})}{\#\mathrm{succ}} &\quad (5.3)\end{aligned}$$

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; evaluations after f_target was reached are disregarded. The $\#\mathrm{FEs}(f_\mathrm{best}(\mathrm{FE}) \ge f_\mathrm{target})$ is the number of function evaluations conducted in all trials, while the best function value was not smaller than f_target during the trial, i.e. the sum over all trials of:

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

The $\#\mathrm{succ}$ denotes the number of successful trials. ERT estimates the expected running time to reach f_target *[Auger:2005b]*, 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.

5.5.3 Bootstrapping

The ERT computes a single measurement from a data sample set (in our case from `Ntrial` optimization runs). Bootstrapping *[Efron:1993]* 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 is reached) *[Auger:2005b]* *[Auger:2009]*. 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.

5.5.4 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) [5] of the intersection point values (stars in Figure *Horizontal vs Vertical View*) for 450 trials.

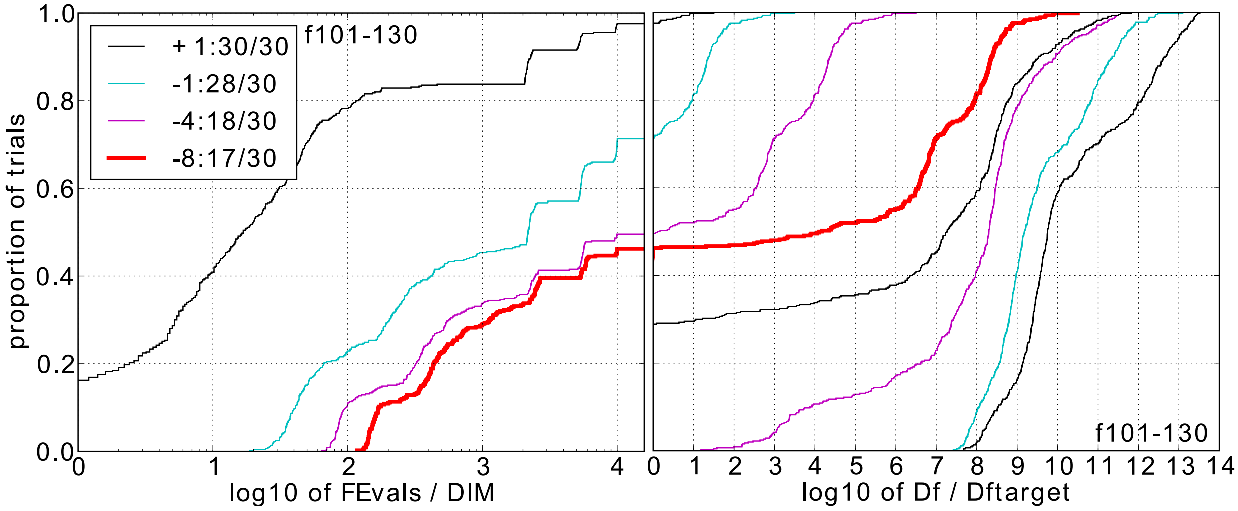


Fig. 5.2: ECDF

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 $f_{\rm opt} + \Delta f$ 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).

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) *[Hoos:1998]* 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

[5] 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.

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

CHAPTER

SIX

RUNNING EXPERIMENTS WITH COCO

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

- in Python, Matlab/GNU Octave, C/C++, R and Java,

6.1 `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.inputFormat = 'row';
opt.algName = 'PUT ALGORITHM NAME';
opt.comments = 'PUT MORE DETAILED INFORMATION, PARAMETER SETTINGS ETC';
maxfunevals = '10 * dim'; % 10*dim is a short test-experiment taking a few minutes
                          % INCREMENT maxfunevals successively to larger value(s)
minfunevals = 'dim + 2';  % PUT MINIMAL SENSIBLE NUMBER OF EVALUATIONS for a restart
maxrestarts = 1e4;        % SET to zero for an entirely deterministic algorithm

dimensions = [2, 3, 5, 10, 20, 40];  % small dimensions first, for CPU reasons
functions = benchmarks('FunctionIndices');  % or benchmarksnoisy(...)
```

```
instances = [1:5, 41:50];  % 15 function instances

more off;  % in octave pagination is on by default

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

for dim = dimensions
  for ifun = functions
    for iinstance = instances
      fgeneric('initialize', ifun, iinstance, datapath, opt);

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

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

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

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

6.1.1 Matlab/GNU Octave

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

6.1.2 C/C++

The interface for running experiments relies on functions such as `fgeneric_initialize`, `fgeneric_finalize`, `fgeneric_ftarget`. The evaluation function is `fgeneric_evaluate` for a single vector or `fgeneric_evaluate_vector` for an array of vectors as input.

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++.

6.1.3 R

Quick Installation

Run the following commands in your R session:

```
install.packages(c("BBmisc", "stringr"),
                 repos="http://cran.at.r-project.org")
fn <- file.path(tempdir(), "bbob_current.tar.gz")
download.file("http://coco.lri.fr/downloads/download15.03/bbobr.tar.gz",
              destfile=fn)
install.packages(fn, repos=NULL)
file.remove(fn)
```

You should now be able to load the package by running

```
library("bbob")
```

If all went well, you can skip down to the next section. Otherwise consulte the detailed instructions which follow.

Detailed Installation

Before you start, install the required dependencies. At the R prompt enter

```
install.packages(c("BBmisc", "stringr"))
```

This should download and install the two packages. Now download the current BBOB R source package and save it somewhere on your hard drive. The most up-to-date version is always available here, but there should also be a stable version available on the COCO website. Now it's time to install the package. In R, run the following command:

```
install.packages("/path/to/bbob_current.tar.gz", repos=NULL)
```

Note that you will have to adjust the path and possibly the filename to match the location where you stored the downloaded package on your hard drive. On Windows you will also need to have the Rtools installed for this to work since the package contains C code. If you have any problems with this step, do not hesitate to contact me for assistance.

After completing the above steps, you should be able to load the package in R

```
library("bbob")
```

The help page for the *bbo_benchmark* function should get you started if you do not want to continue reading this introduction.

```
?bbo_benchmark
```

Simple experiment

If you already have an optimizer ready to go in R and all you want to do is produce a BBOB dataset for post-processing, then this section will walk you through the required steps. We will use the high-level interface *bbo_benchmark* in this example. If you want to parallelize your optimization runs, need to perform complex initializations or just want full control, skip down to the next section for a brief tour of the low-level interface.

In this example we will use the L-BFGS-B optimizer included in base R. You will need to adapt parts of the code to suit your optimizer. The first step is to wrap your optimization algorithm in a function with the following signature

```
function(par, fun, lower, upper, max_eval)
```

where *par* will be a numeric vector with the starting point for the optimization, *fun* is the function to be minimized, *lower* and *upper* are the bounds of the box constraints and *max_eval* is the number of function evaluations left in the allocated budget. What these five parameters mean is best conveyed by an example. Here we wrap the *optim* function included in base R

```
my_optimizer <- function(par, fun, lower, upper, max_eval) {
  optim(par, fun, method="L-BFGS-B",
        lower=lower, upper=upper, control=list(maxit=max_eval))
}
```

If your algorithm does not have the notion of an initial parameter setting, you can safely ignore the *par* parameter in your implementation. You might also notice, that we do not strictly adhere to the *max_eval* limit because the number of iterations is generally not equal to the number of function evaluations for L-BFGS-B. This is OK. *max_eval* is only a hint to the optimizer how much effort it should put into optimizing the function.

Should your algorithm perform restarts internally it is possible to log these using the *bbob_log_restart* function. The function takes exactly one argument, a string describing the reason for the restart.

We are now ready to run our experiment. This is done by calling the *bbo_benchmark* function

```
bbo_benchmark(my_optimizer, "l-bfgs-b", "optim_l-bfgs-b")
```

which will perform the BBOB experiments (caution, may take many hours). The first argument passed to *bbo_benchmark* is our optimization wrapper from the previous step. Make sure that it has the correct function signature! Next we supply the so called *algorithm id*. This is a short descriptive name for our optimization algorithm. Ideally it should include the package name and version which contains the algorithm. So for for *genoud* from the *rgenoud* package, we might use *rgenoud::genoud (5.7-3)* as the algorithm id. The last required argument is the name of the base directory where the result files will be stored. Again, it is a good idea to include the algorithm name in the directory name. If no other arguments are given, this runs a complete BBO benchmark on the noiseless test functions. This includes all instances (1-5 and 21-30), all dimensions (2, 3, 5, 10, 20, 40). If you do not want to include runs in 40 dimensions or want to use different instances you can change the defaults using the *dimensions* and *instances* arguments to *bbo_benchmark*. For details, see the manual page for *bbo_benchmark*.

If no other budget is specified, *bbo_benchmark* will perform random independent restarts of your algorithm until the desired target precision (1e-8) is reached or the default budget of 10000000 function evaluations is exhausted. If you want to reduce the budget, you can by specifiying it as the *budget* argument to *bbo_benchmark*.

To run the required timing experiment, execute the following code:

```
bbo_timing(my_optimizer)
```

It will return a data frame with the relevant timing information.

Low-level interface

Will follow soon.

6.1.4 Python

The interface for running an experiment is `fgeneric` which is used within `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 fgeneric.py need to be
in the current working directory.

Under unix-like systems:
    nohup nice python exampleexperiment.py [data_path [dimensions [functions [instances]]]] > output

"""
import sys # in case we want to control what to run via command line args
import time
import numpy as np
import fgeneric
import bbobbenchmarks

argv = sys.argv[1:] # shortcut for input arguments

datapath = 'PUT_MY_BBOB_DATA_PATH' if len(argv) < 1 else argv[0]

dimensions = (2, 3, 5, 10, 20, 40) if len(argv) < 2 else eval(argv[1])
function_ids = bbobbenchmarks.nfreeIDs if len(argv) < 3 else eval(argv[2])
# function_ids = bbobbenchmarks.noisyIDs if len(argv) < 3 else eval(argv[2])
instances = range(1, 6) + range(41, 51) if len(argv) < 4 else eval(argv[3])
```

```

opts = 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)
minfunevals = 'dim + 2'  # PUT MINIMAL sensible number of EVALUATIONS before to restart
maxrestarts = 10000      # SET to zero if algorithm is entirely deterministic


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, REPLACE with optimizer to be tested
    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 _ 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

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

f = fgeneric.LoggingFunction(datapath, **opts)
for dim in dimensions:  # small dimensions first, for CPU reasons
    for fun_id in function_ids:
        for iinstance in instances:
            f.setfun(*bbobbenchmarks.instantiate(fun_id, iinstance=iinstance))

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

            f.finalizerun()

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

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

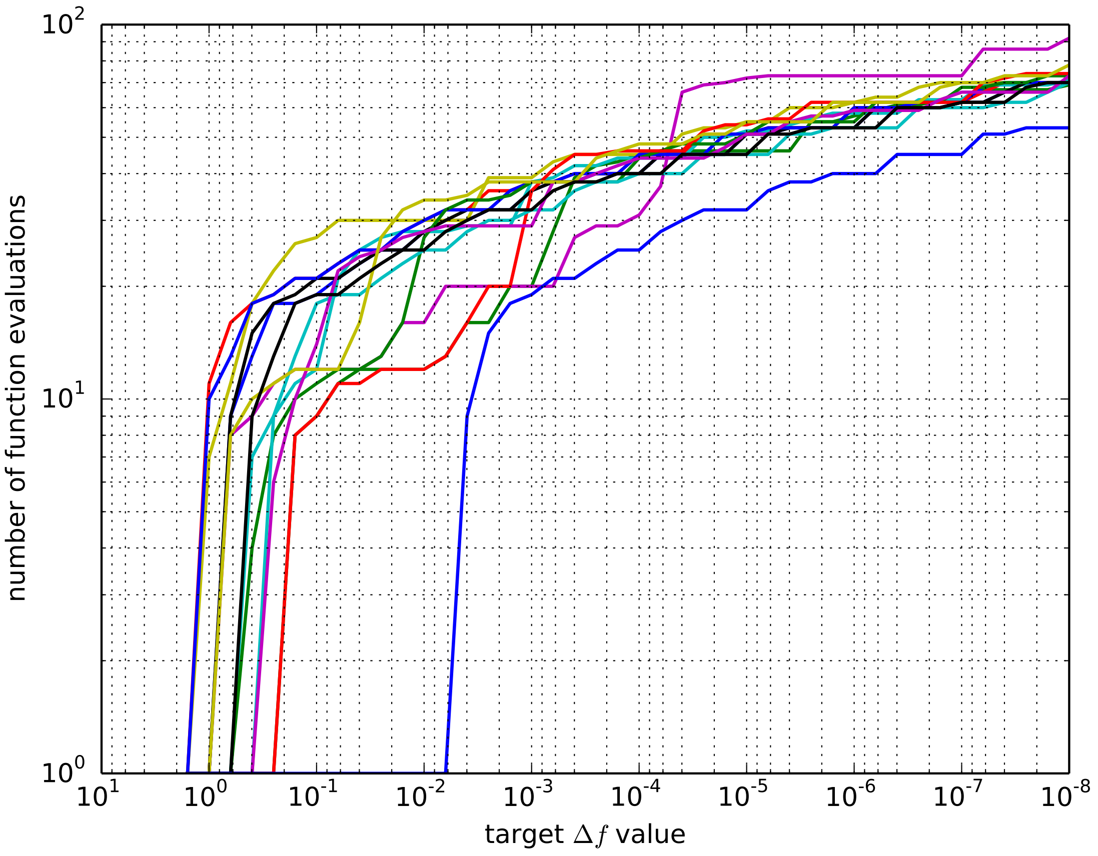
6.2 Testing New Functions

We describe here how to use `fgeneric` to record experiments on functions that are not part 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 fgeneric as fg
>>> import scipy.optimize as so
>>> f = lambda x: sum(i**2 for i in x) # function definition
>>> e = fg.LoggingFunction(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 LoggingFunction.evalfun of <fgeneric.LoggingFunction object at [...]>>, 1e-08)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: [...]
         Function evaluations: [...]
array([...])
[...]
>>> # Display convergence graphs
>>> import bbob_pproc as bb
>>> ds = bb.load('data')
>>> ds.plot()
```



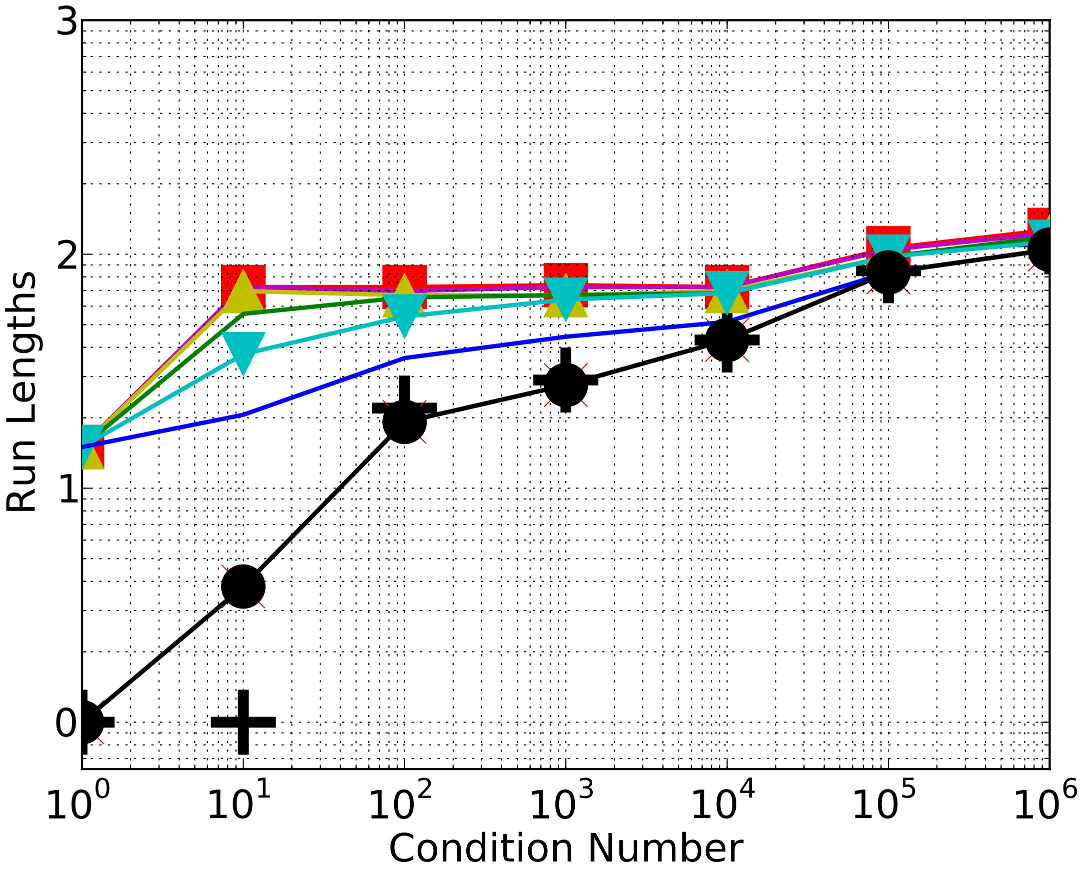
6.2.1 Testing Functions with Parameter

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 fgeneric as fg
>>> import scipy.optimize as so
>>> import numpy as np

>>> e = fg.LoggingFunction(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 LoggingFunction.evalfun of <fgeneric.LoggingFunction object at [...]>>, 1e-08)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: [...]
         Function evaluations: [...]
         Gradient evaluations: [...]
array([...])
[...]
>>> # plot data
>>> import bbob_pproc as bb
>>> import bbob_pproc.ppfigparam
>>> ds = bb.load('ellipsoid')
>>> bb.ppfigparam.plot(ds, param='condnum')
>>> bb.ppfigparam.beautify()
>>> import matplotlib.pyplot as plt
>>> plt.xlabel('Condition Number')
```



CHAPTER

SEVEN

POST-PROCESSING WITH COCO

7.1 Overview of the `bbob_pproc` Package

We present here the content of version 10.7 of the `bbob_pproc` package.

`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 during module import,

`config.py` configuration and modification of settings depending on input parameters while the module is running

`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.toolsstats` 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,

7.2 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.
ERT loss ratio figures and tables done.
Output data written to folder ppdata/DATAPATH.

$
```

The above command creates the folder with the default name `ppdata/DATAPATH` in the current working directory, which contains 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 import command requires that the path to the package `bbob_pproc` is in the search path of Python (e.g. the current folder or use `sys.path.append` to append the path).

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

7.2.1 Expensive Setting

`rungeneric.py` accepts quite a few parameters, consider

```
$ python path_to_postproc_code/bbob_pproc/rungeneric.py --help
$ python path_to_postproc_code/bbob_pproc/rungeneric1.py --help
```

Since version 13.01, data with a smaller overall budget (number of overall evaluations less than about 1000 times dimension) are processed differently. This processing style can be enforced using the `--expensive` option. In this case the targets are chosen based on function values reached by the best 2009 algorithm after some fixed budgets. At this time, the option only applies to the single algorithm processing (via `rungeneric.py` or `rungeneric1.py`).

7.2.2 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 that ALG1, ALG2 and ALG3 are valid directories and containing either pickle files (which contain Python-formatted data) or the raw experiment data, in particular files with 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.

7.2.3 Use of the LaTeX Templates

Several LaTeX files are provided as template articles. These files compile the figures and tables generated by the post-processing in a single document. The templates have different layouts. The files with *noisy* in their names aggregate results on the noisy testbed from BBOB while the others compile results on the noiseless testbed.

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). Similarly, names with ‘cmp’ refer to LaTeX templates comparing two algorithms and templates with ‘many’ in their names can be used to compare two or more algorithms.

The usual way of using these LaTeX templates is to:

1. copy the necessary template and associated style files (if necessary) in the current working directory, i.e. in the same location as the output folder of the post-processing ppdata.
2. (optional) edit the template, in particular the algorithm name. The default data folder name is defined as \newcommand{\bbobdatapath}{ppdata/}. In ppdata the file bbob_pproc_commands.tex defines relevant LaTeX commands, in particular the folder in ppdata where the latest post-processed data are found.

3. compile with LaTeX.

Note: bbob_pproc.rungeneric will generate *comparison* figures and tables at the base of the output folder and individual results of each algorithm for which data were provided in subfolders. The *template1* templates can be used by pointing to any of these subfolders with \newcommand{\algfolder}{desired-subfolder}.

The templates are compiled using LaTeX, for instance:

```
$ pdflatex template1generic

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

$
```

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

7.3 Post-Processing from the Python Interpreter

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

7.3.1 Pylab

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

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

which

- imports the matplotlib.pyplot and numpy packages, and
- switches the interactive mode of matplotlib on.

Note: We recommend using iPython, an environment enhancing the Python interpreter for interactive use:

```
$ ipython

Python 2.7.9...
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- 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.

In [1]:
```

Next, we activate pylab mode, a MATLAB-like environment.

```
In [1]: %pylab
Using matplotlib backend: MacOSX
Populating the interactive namespace from numpy and matplotlib
```

Some differences with the Python interpreter are:

- the prompt is different: `In [1]:` versus `>>>`
- system commands are directly available, like `ls`, `cd`, ...
- the excellent tab completion is directly available
- `run myfile.py` runs a python script, just as `python myfile.py` from the OS-shell command line, `help plot` displays help, and many other so-called magic commands are available
- pasting code into the shell usually works (`%doctest_mode` allows to paste code preceded with a `>>>` prompt).
- typing `debug` after an error occurs jumps into the execution code into the state before the error was raised, where all variables can be inspected and commands executed like in an interactive python shell.

7.3.2 Pre-requisites

For this session, we assume that:

- the folder `~/code` contains the folder `bbob.v10.74/python/bbob_pproc` extracted from one of the archives. The latter folder contains the Python module `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`.

7.3.3 Start of the Session

First, we start the Python interpreter here by typing `python` from the command line:

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

On Unix systems, enabling the tab-completion in the python shell makes life much more convenient:

```
>>> import rlcompleter
>>> import readline  # only available on Unix systems
>>> readline.parse_and_bind("tab: complete")
>>>
```

Note: One might like to write these commands into a `startup.py` file and type

```
>>> exec open('startup.py')
>>>
```

or even better put them into the file pointed to by the `PYTHONSTARTUP` system enviroment variable, e.g. `~/.pythonrc`.

An alternative is to use `ipython` or `IDLE`.

In order to have a matlab-like enviroment we can “start pylab mode”:

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

An alternative is to use `ipython` as noted above.

The `bbob_pproc` package is then loaded into memory:

```
>>> import sys
>>> sys.path.append('~/code/bbob.v10.74/python')   # where bbob_proc can be found
>>> import bbob_pproc as bb
>>> help(bb)
Help on package bbob_pproc:
[...]
```

will list the `bbob_pproc` documentation.

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.

Data is loaded into memory with the `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)]
>>> type(ds)
<class 'bbob_pproc.pproc.DataSetList'>
>>> len(ds)
324
>>>
```

The variable `ds` stores an instance of class `pproc.DataSetList` represented between square brackets. This instance in `ds` is a list of `pproc.DataSet` instances.

```
>>> 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:
 |
    [...]
```

The central data structure in COCO is the `DataSet` that holds all trials on a single function on a given dimension.

```
>>> 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)
 |      instancenumbers -- 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 `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'))
>>>
```

Some information on loaded data can be obtained with the `info` method:

```
>>> ds = bb.load('BIPOP-CMA-ES/ppdata_f002_20.pickle')
>>> ds.info() # 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 `pproc.DataSet` instance can be displayed on-screen:

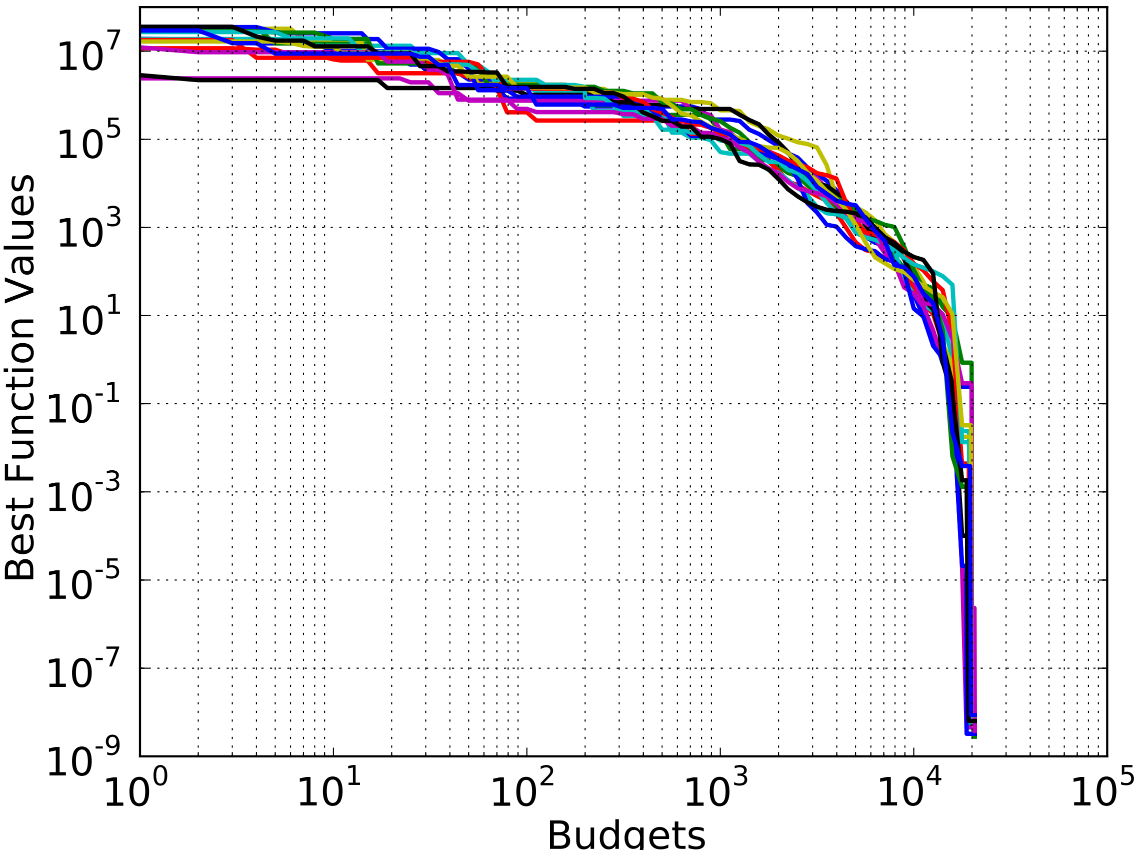
```
>>> d = ds[0] # store the first element of ds in d for convenience
>>> d.funcId  # in IPython d.<TAB> displays all available attributes and methods
2
>>> d.instancenumbers # instance numbers of the available trials
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]
>> type(d.funvals)  # numpy array type is fundamental
<type 'numpy.ndarray'>
>>> d.funvals.shape  # we find 95 rows in function value data
(95, 16)
>>> d.funvals[[0,1,-1], :]  # let's see the first two and last row
array([[  1.00000000e+00,   1.91046048e+07,   3.40744851e+07,
          1.16378912e+07,   1.88965407e+07,   2.41571279e+06,
          3.22847770e+07,   2.85601634e+06,   3.51908185e+07,
          2.82405771e+07,   1.79074770e+07,   2.71439178e+07,
          1.22112898e+07,   1.65690802e+07,   3.57969327e+07,
          2.94479644e+07],
       [  2.00000000e+00,   1.83130834e+07,   3.40744851e+07,
          1.16378912e+07,   1.88965407e+07,   2.41571279e+06,
          3.22847770e+07,   2.20298276e+06,   3.51908185e+07,
          2.82405771e+07,   1.79074770e+07,   2.71439178e+07,
          9.46258565e+06,   1.65690802e+07,   3.57969327e+07,
          2.94479644e+07],
       [  2.06900000e+04,   4.79855089e-09,   2.78740231e-09,
          5.28442001e-09,   5.17276533e-09,   4.54151916e-09,
          5.04810771e-09,   5.50441826e-09,   3.25114513e-09,
          7.20403648e-09,   6.95291646e-09,   5.91512617e-09,
          3.79441190e-09,   8.42007353e-09,   6.33018260e-09,
          8.59924398e-09]])
>>> budgets = d.funvals[:, 0] # first column contains budgets
>>> funvals = d.funvals[:, 1:] # stores columns 1,...,15 in funvals
>>>
```

The first column (with index 0) of `DataSet` attribute `funvals` contains number of function evaluations. The remaining columns contain the best function value precision (difference to the optimal function value) in the respective trial or run.

Graphs of the evolution of the best precision over time for each single run can be displayed as follows [2]:

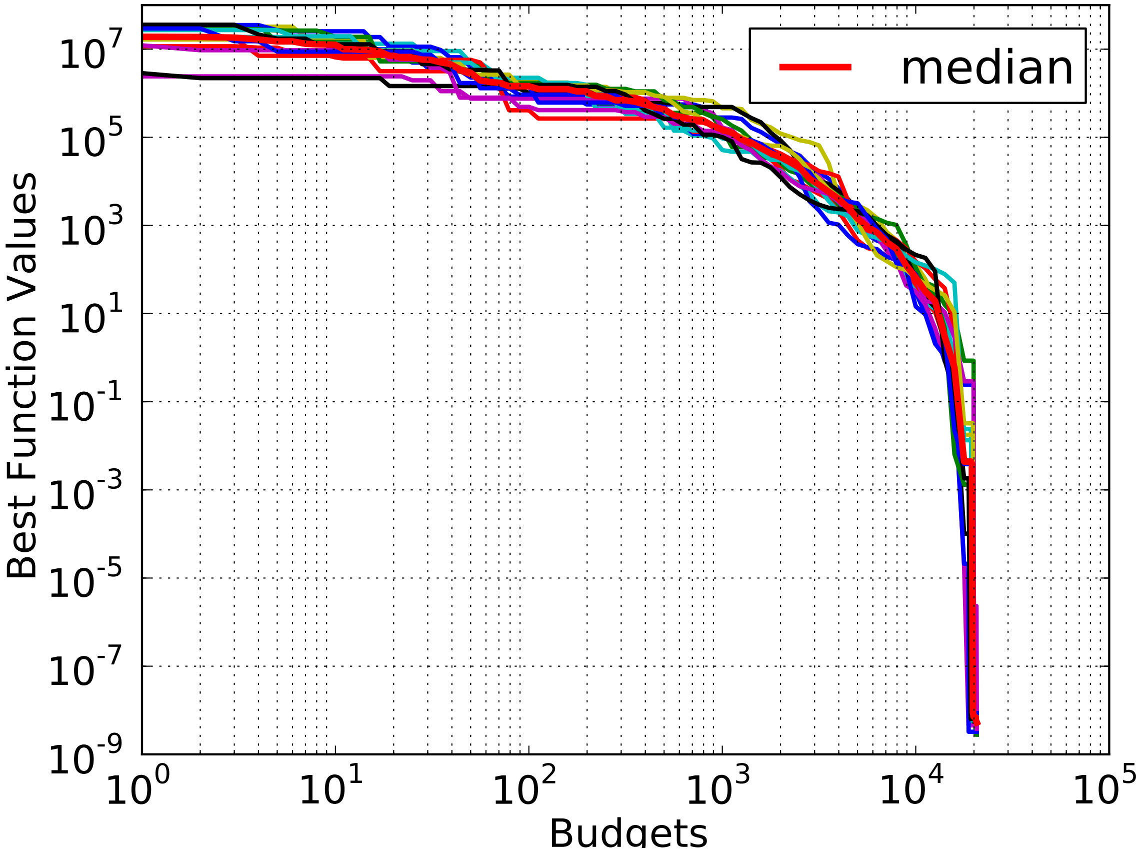
```
>>> for f in funvals.T:  # T==transpose: iterate over columns of funvals
...     loglog(budgets, f)
...
[<matplotlib.lines.Line2D object at 0x1[...]>]
    [...]
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> grid()
>>> xlabel('Budgets')
<matplotlib.text.Text object at 0x1[...]>
>>> ylabel('Best Function Values')
<matplotlib.text.Text object at 0x1[...]>
```

[2] More information on plot functions from http://matplotlib.sourceforge.net/users/pyplot_tutorial.html



We might add the median data points.

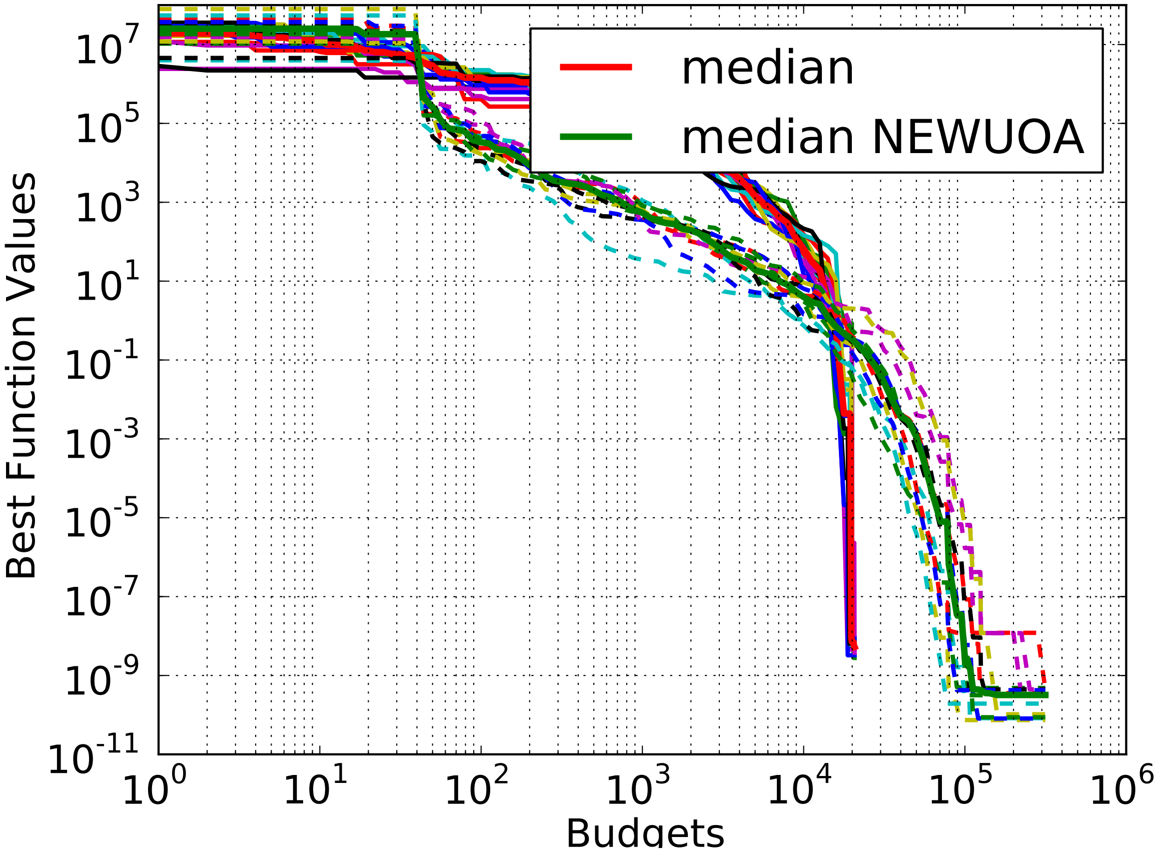
```
>>> loglog(budgets, median(funvals, axis=1), linewidth=3, color='r',
...         label='median CMA-ES')
[<matplotlib.lines.Line2D object at 0x1[...]>]
>>> legend() # display legend
<matplotlib.legend.Legend object at 0x1[...]>
```



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.

We display another data set for comparison:

```
>>> ds1 = bb.load('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:

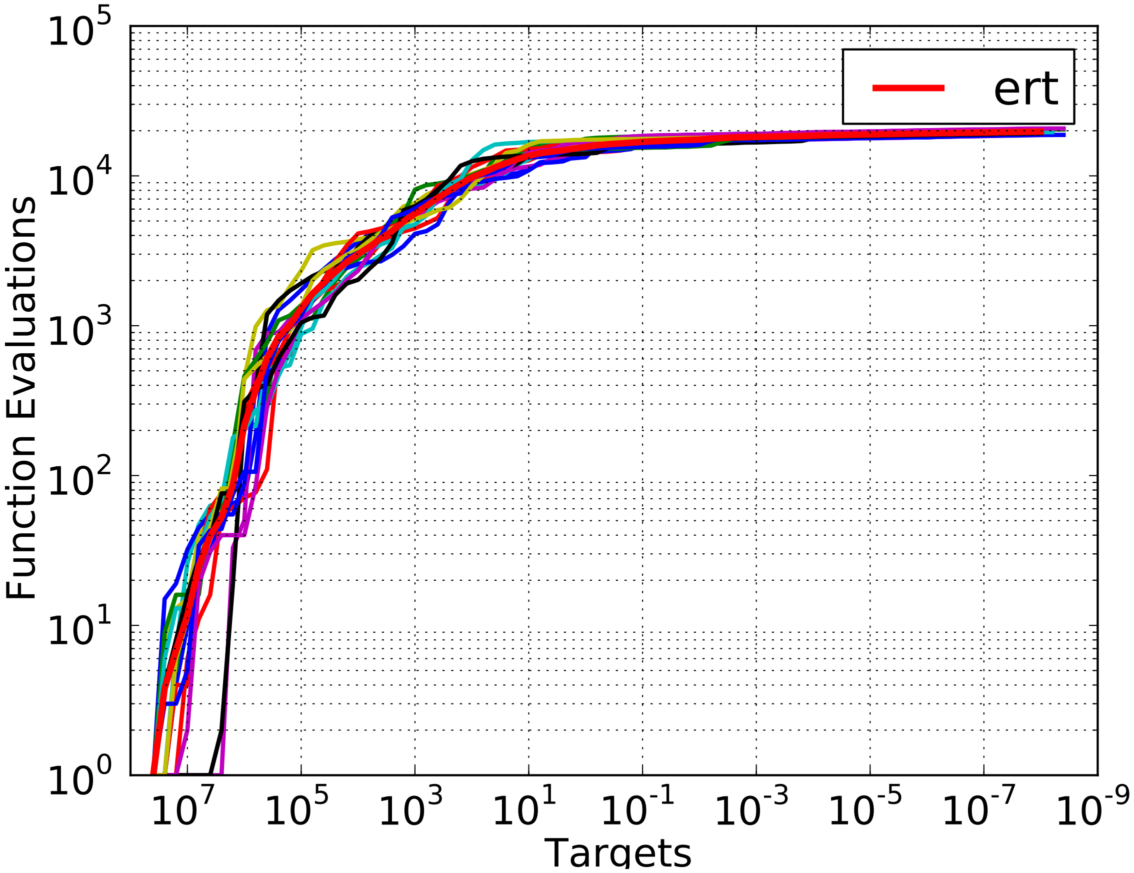
```
>>> savefig('examplefigure')  # save active figure as image
```

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.

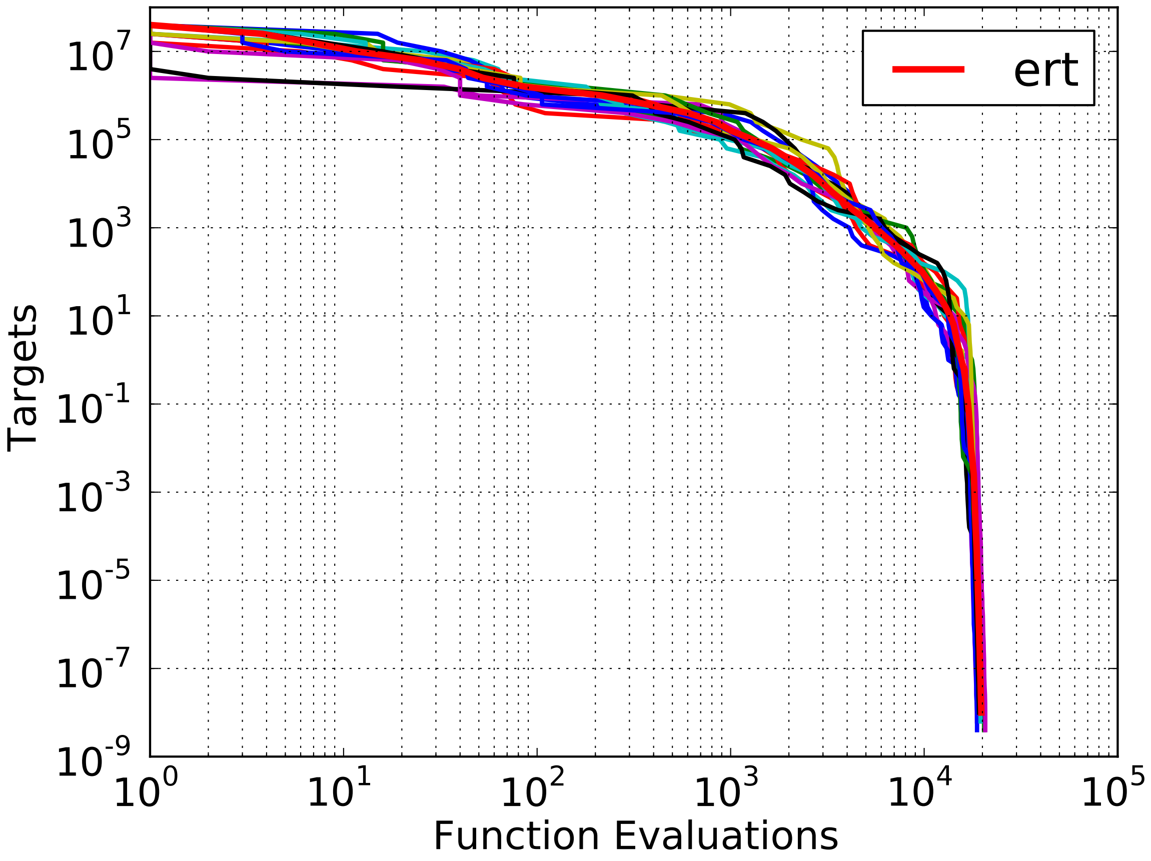
```
>>> targets = d.evals[:, 0]
>>> evals =  d.evals[:, 1:]
>>> nruns = evals.shape[1]
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]
>>> for i in range(0, nruns):
...     loglog(targets, evals[:, i])
...
[<matplotlib.lines.Line2D object at 0x1[...]
    [...]
[<matplotlib.lines.Line2D object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



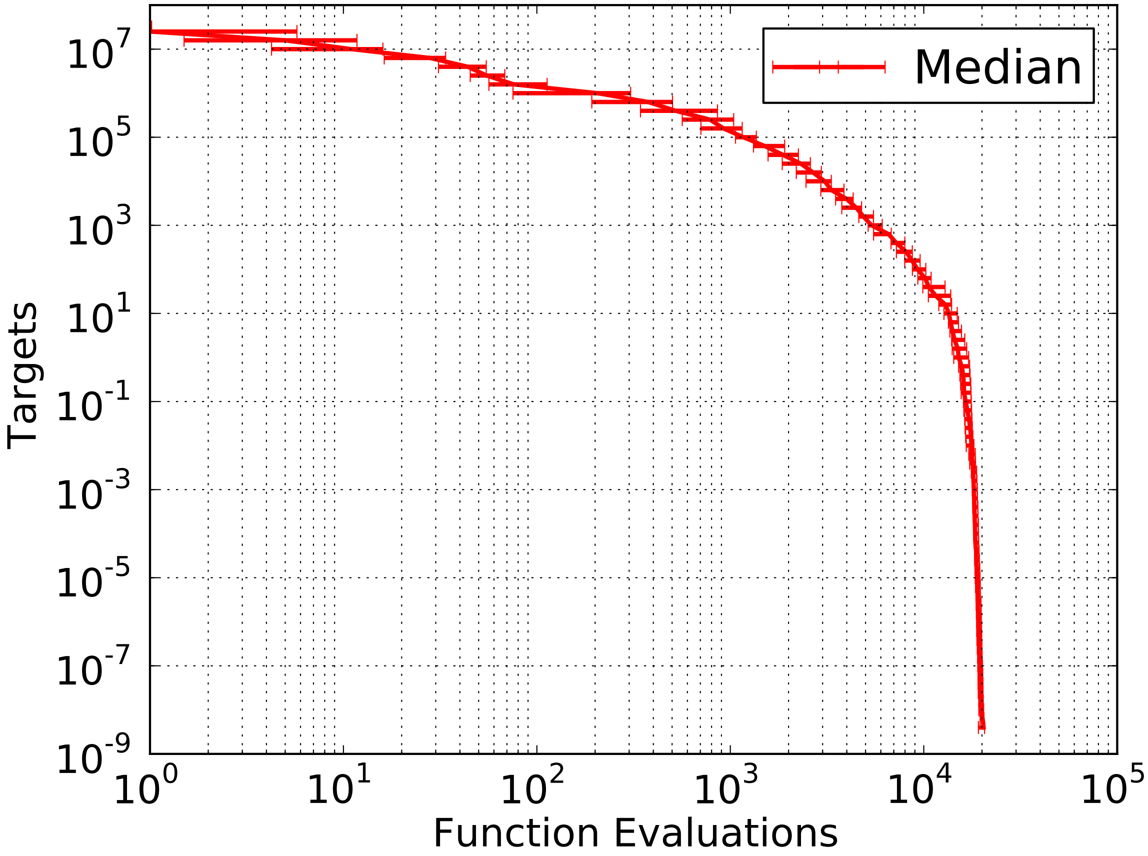
If we swap the abscissa and the ordinate of the previous figure, we can directly compare the horizontal view and vertical view figures:

```
>>> figure()
<matplotlib.figure.Figure object at 0x1[...]>
>>> for i in range(0, nruns):
...     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[...]>
```

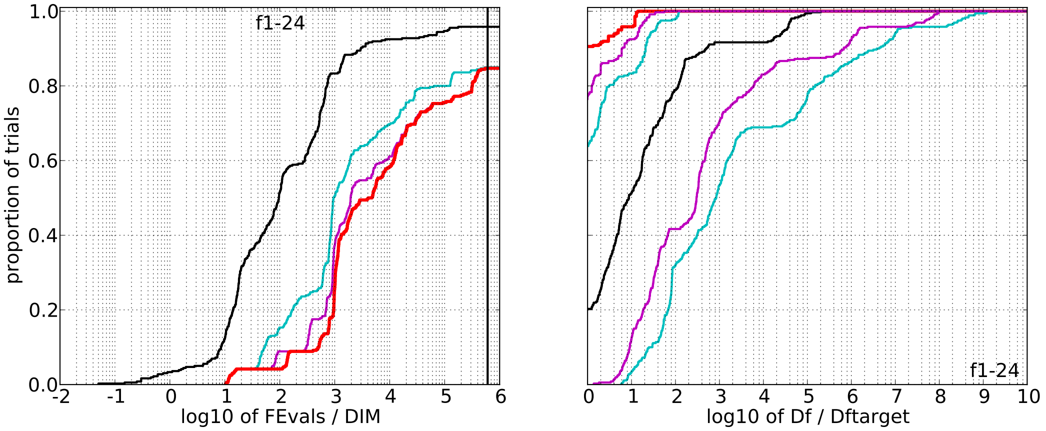


Other types of figures are generated by different modules in `bbob_pproc`. Each of these modules has a `plot` function and a `beautify` function. The `plot` function is used for generating a type of graph. The `beautify` function accommodates the general presentation of the figure to its content.

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:

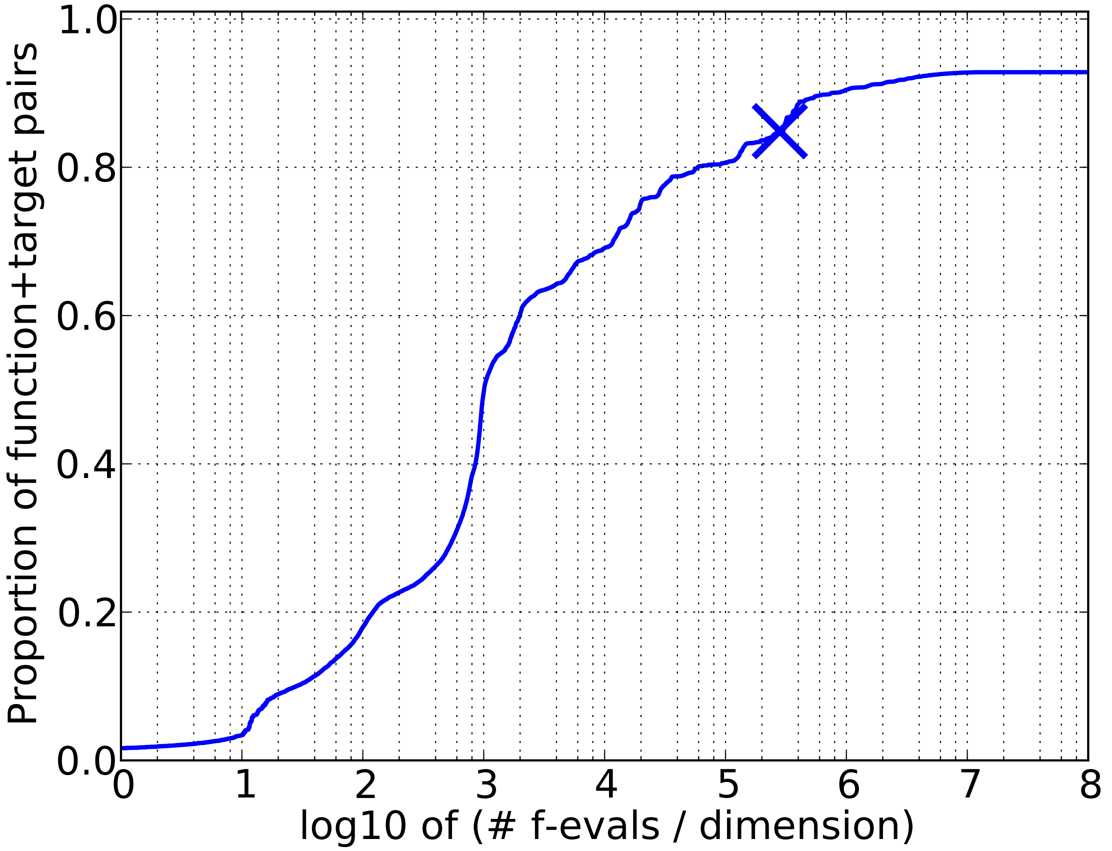
```
>>> help(bb.pprldistr)
Help on module bbob_pproc.pprldistr in bbob_pproc:

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'))
>>> 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:
```

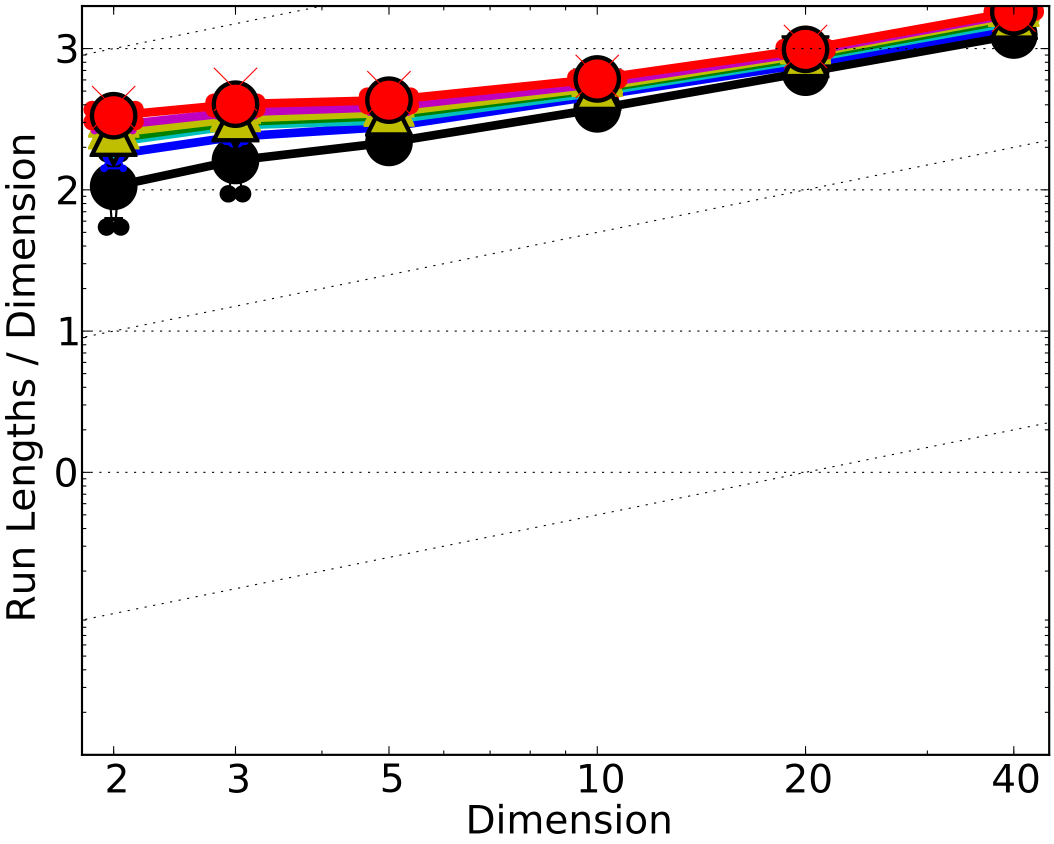
```
NAME
    bbob_pproc.compall.ppperfprof - Generates figure of the bootstrap distribution of ERT.

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'))
>>> 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()
>>> ppfigdim.plot(ds)
>>> ppfigdim.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

EIGHT

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:2005a] 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.

[Auger:2005b] 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.

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

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

[Harik:1999] 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.

[Hoos:1998] 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.

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