COCO (COmparing Continuous Optimizers) Software: User Documentation

Steffen Finck* and Raymond Ros†

compiled March 22, 2010

Contents

1 Purpose

The COmparison of Continuous Optimizers (COCO) software[1] is a benchmarking software to render easier experiments in the field of continuous optimization. A post-processing Python package generates tables and figures to be included in a research paper template presenting all results.

The COCO software was used for the GECCO 2009 workshop named Black-Box Optimization Benchmarking (BBOB-2009). The efforts of BBOB-2009 resulted in thirty-eight accepted workshop papers presenting results of state-of-the-art algorithms.

*SF is with the Research Center PPE, University of Applied Sciene 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

[1] Available at http://coco.gforge.inria.fr

The same efforts will be pursued for the workshop BBOB-2010[2] to be held during GECCO 2010[3].

The COCO software provides:

1. a single generic function interface fgeneric to the benchmark functions of BBOB-2010, coded in MATLAB/GNU OCTAVE and C,
2. Java Native Interface classes to use fgeneric in JAVA,
3. the PYTHON post-processing module bbob_pproc,
4. LaTeX templates to generate papers, and
5. the corresponding documentation.

The practitioner in BBO who wants to benchmark one or many algorithms on the BBOB-2010 testbeds has to download COCO, interface the algorithms to call the test functions in the testbed and use the post-processing tools. The most substantial part is to render the considered algorithms compatible with our software implementation.

We describe the different steps for obtaining a complete workshop paper for an algorithm, thus allowing us to present the architecture of COCO. We also present additional facilities implemented for the comparison of the results of the many algorithms submitted. Section 2 presents the experimental framework software used to generate benchmarking data. Section 3 describes the post-processing facilities of COCO, namely the PYTHON package bbob_pproc. Section 4 briefly describes the process of compiling a paper regrouping all the post-processed results.

2 Experimental Framework Software

The experimental framework software mainly consists in the implementation of the methodology presented in [2]. The software is centered on the interface function, fgeneric.

We describe the format of the output data files and the content of the files as they are written by fgeneric. These files are to be analysed with the provided post-processing tools that are described in Section 3. To display an example of the use of fgeneric, we provide two example scripts. Executing the MATLAB scripts provided in Listings 2 and 3 results in testing an algorithm —MY_OPTIMIZER in the examples, see Listing 1— on the noiseless testbed of BBOB-2010 and displaying measures of the time complexity of an algorithm respectively. In Listing 2, lines 6 to 10 set variables used by fgeneric. The whole set of experiment on the noiseless testbed is done by looping over the lines 18 to 36.

The function fgeneric outputs the results of the experiments, also it provides a single interface to any of the test functions of the BBOB-2010 testbeds. Once fgeneric is loaded into memory, the initialization process, see line 21 in Listing 2, sets all variables internal to fgeneric: the test function considered, the instance considered, the output directory. Later calls to fgeneric evaluate the chosen test

[2] http://coco.gforge.inria.fr/doku.php?id=bbob-2010
[3] http://www.sigevo.org/gecco-2010/workshops.html#bbob

Listing 1: `MY_OPTIMIZER.m`: Monte Carlo search in MATLAB. At each iteration, 200 points are sampled and stored in a matrix of size `DIM` $\times$ 200 so as to reduce loops and function calls within MATLAB and therefore improve its efficiency

```
1  function MY_OPTIMIZER(FUN, DIM, ftarget, maxfunevals)
2  % MY_OPTIMIZER(FUN, DIM, ftarget, maxfunevals)
3  % samples new points uniformly randomly in [-5,5]^DIM
4  % and evaluates them on FUN until ftarget of maxfunevals
5  % is reached, or until 1e8 * DIM fevals are conducted.
6  % Relies on FUN to keep track of the best point.
7
8    maxfunevals = min(1e8 * DIM, maxfunevals);
9    popsize = min(maxfunevals, 200);
10   for iter = 1:ceil(maxfunevals/popsize)
11     feval(FUN, 10 * rand(DIM, popsize) - 5);
12     if feval(FUN, 'fbest') < ftarget  % task achieved
13       break;
14     end
15     % if useful, modify more options here for next start
16   end
```

function at the point $\vec{x}$ given as input argument, see line 11 of Listing 1. Necessary finalization operations are effected by using the command fgeneric('finalize') in MATLAB, see line 31 in Listing 2.

In Listing 2, the function f_8 is tested in 2, 3, 5, 10, 20, and 40-D. The `while` loop from line 15 to 18 make the runs last thirty seconds.

2.1 Running Experiments in C

The interface to fgeneric differs from the MATLAB example given in [2], we provide in Listing 4 the equivalent example script in C. A specific folder structure is needed for running an experiment. While creating the folder structure was handled by running fgeneric in MATLAB, this is not the case using the C code. 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 module createfolders before executing any experiment program. Make sure `createfolders.py` is in your current working directory and from the command-line simply do:

```
python createfolders.py PUT_MY_BBOB_DATA_PATH
```

Calls to fgeneric specified by a string first argument in MATLAB, are replaced by fgeneric_*string* in C, e.g. fgeneric('ftarget') is replaced with fgeneric_ftarget. Also, the generic call to fgeneric(X) to evaluate candidate vectors is replaced by fgeneric_evaluate(double * X) for a single vector and fgeneric_evaluate_vector(double * XX, unsigned int np, double * result) for an array of vectors where XX is the concatenation of the np candidate vectors and result is an array of size np which contains the resulting function values.

2.2 Running Experiments in Java

The class JNIfgeneric implements an interface for using the C-implementation of fgeneric. Methods fgeneric_*string* in C are replaced by JNIfgeneric.*string*,

Listing 2: `exampleexperiment.m`: script for benchmarking MY_OPTIMIZER, see Listing 1, for BBOB-2010 on the noiseless function testbed in MATLAB/GNU OCTAVE

```
% 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 = '20 * dim';  % SHORT EXPERIMENT, takes overall three minutes

more off;  % in octave pagination is on by default

t0 = clock;
rand('state', sum(100 * t0)); % initialises the pseudo-random generator
                              % in MY_OPTIMIZER

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

      MY_OPTIMIZER('fgeneric', dim, fgeneric('ftarget'), eval(maxfunevals));

      disp(sprintf(['  f%d in %d-D, instance %d: FEs=%d,' ...
                    ' fbest-ftarget=%.4e, elapsed time [h]: %.2f'], ...
                   ifun, dim, iinstance, ...
                   fgeneric('evaluations'), ...
                   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
```

except for the initialization JNIfgeneric.initBBOB(...) and finalization JNIfgeneric.exitBBOB().

3 Post-Processing the Experimental Data

The PYTHON post-processing tool, called bbob_pproc in BBOB-2010 generates image files and LaTeX tables from the raw experimental data obtained as described previously in Section 2.

The entire post-processing tool requires that PYTHON is installed on your machine. The minimal software requirements for using the post-processing tool are Python (2.5.2), Matplotlib (0.91.2) and Numpy (1.0.4). The installation of the software is described in Appendix A.

Listing 3: `exampletiming.m`: script for measuring the time complexity of MY_OPTIMIZER, see Listing 1, for BBOB-2010 in MATLAB/GNU OCTAVE

```
% 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
```

Overview of the bbob_pproc Package

We present here the content of the latest version of the bbob_pproc package (version 10.0).

`run.py` is the main interface of the package that calls the different routines listed below,

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

`ppfigdim.py`, `pptex.py`, `pprldistr.py` are used to produce figures and tables that we describe further down,

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

`dataoutput.py` contain routine to output instances of DataSet in PYTHON-formatted data files,

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 `runcompall.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 `runcomp2.py`.

Listing 4: `exampleexperiment.c`: script for benchmarking MY_OPTIMIZER, for BBOB-2010 on the noiseless function testbed in C

```
/*runs an entire experiment benchmarking MY_OPTIMIZER on the noise-free testbed*/

#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "bbobStructures.h" /* Include all declarations for BBOB calls */

/* include all declarations for your own optimizer here */
void MY_OPTIMIZER(double(*fitnessfunction)(double*), unsigned int dim,
                  double ftarget, unsigned int maxfunevals);

int main()
{
    unsigned int dim[6] = {2, 3, 5, 10, 20, 40};
    unsigned int idx_dim, ifun, instance;
    clock_t t0 = clock(); time_t Tval;
    ParamStruct params = fgeneric_getDefaultPARAMS();

    srand(time(NULL)); /* used by MY_OPTIMIZER */
    strcpy(params.dataPath, "PUT_MY_BBOB_DATA_PATH");
    /* please run 'python createfolders.py PUT_MY_BBOB_DATA_PATH' beforehand */
    strcpy(params.algName, "PUT ALGORITHM NAME");
    strcpy(params.comments, "PUT MORE DETAILED INFORMATION, SETTINGS ETC");

    for (idx_dim = 0; idx_dim < 6; idx_dim++)
    {
        /*Function indices are from 1 to 24 (noiseless) or from 101 to 130 (noisy)*/
        for (ifun = 1; ifun <= 24; ifun++)
        {
            for (instance = 1; instance <= 15; instance++)
            {
                /* Mandatory for each experiment: set DIM, funcId, instanceId*/
                params.DIM = dim[idx_dim];
                params.funcId = ifun;
                params.instanceId = instance;
                fgeneric_initialize(params);

                MY_OPTIMIZER(&fgeneric_evaluate, dim[idx_dim], fgeneric_ftarget(),
                             20*dim[idx_dim]); /* SHORT EXPERIMENTS. */

                printf("  f%d in %d-D, instance %d: FEs=%lu,", ifun, dim[idx_dim],
                       instance, fgeneric_evaluations());
                printf(" fbest-ftarget=%.4e, elapsed time [h]: %.2f\n",
                       fgeneric_best() - fgeneric_ftarget(),
                       (double)(clock()-t0)/CLOCKS_PER_SEC/60./60.);

                fgeneric_finalize();
            }
            Tval = time(NULL); printf("    date and time: %s", ctime(&Tval));
        }
        printf("---- dimension %d-D done ----\n", dim[idx_dim]);
    }
    return 0;
}
```

3.1 Using the bbob_pproc Package

To perform the post-processing on the experimental data, the bbob_pproc package needs to be downloaded[4] and un-archived. Then, to post-process the data, the data folder `DATAPATH` containing all data generated by the experiments needs to be in the current working directory before executing the following command:

```
python path_to_postproc_code/bbob_pproc/run.py DATAPATH
```

from a shell[5], the folder `path_to_postproc_code` is the one where the provided post-processing software was un-archived.

The above command create the folder with the default name `ppdata` 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.

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

```
>>> import bbob_pproc
>>> bbob_pproc.main('DATAPATH')
```

This first command loads bbob_pproc into memory and requires that the path to the package is in the Python search path.

The resulting `ppdata` folder now contains a number of TeX, eps, png files.

Additional help for the bbob_pproc package can be obtained by executing the following command in a shell:

```
python path_to_postproc_code/bbob_pproc/run.py -h
```

In particular, this command describes the additional options for the execution of the post-processing. The code documentation can be found in the folder `path_to_postproc_code/pydoc` within the provided software package.

3.2 Comparison of Algorithms

The sub-package bbob_pproc.compall and bbob_pproc.comp2 (v10.0) from bbob_pproc provide facilities for the generation of tables and figures comparing the performances of algorithms tested using COCO.

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/runcompall.py ALG1 ALG2 ALG3
```

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 `cmpalldata` containing comparison figures and tables.

Outputs appropriate to the comparison of only two algorithms can be obtained using bbob_pproc.comp2 by executing from the command line:

[4] The package can be obtained from http://coco.gforge.inria.fr/doku.php?id=bbob-2010.

[5] Note that in Windows the path separator '\' must be used instead of '/'

```
python path_to_postproc_code/bbob_pproc/runcomp2.py ALG0 ALG1
```

This assumes the folders `ALG0` and `ALG1` are in the current working directory. Running the aforementioned command will generate the folder `cmp2data` containing the comparison figures.

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

```
>>> from bbob_pproc import runcompall
>>> bbob_pproc.runcompall.main('ALG1 ALG2 ALG3'.split())
```

or:

```
>>> from bbob_pproc import runcomp2
>>> bbob_pproc.runcomp2.main('ALG0 ALG1'.split())
```

The from... import... command loads package into memory and requires that the path to the package is in the PYTHON search path. Call to the main method runs the whole post-processing script.

4 Generating a Paper

`templateBBOBarticle.tex` and `templateBBOBnoisyarticle.tex` are the template LaTeX files that include all the figures and tables presenting the result of an algorithm on the noiseless and noisy testbeds of BBOB-2010. If compiled correctly using LaTeX, it generates documents collecting and organizing the output from bbob_pproc. Each of the templates has a given page organization optimized for the presentation of the results on each testbed.

To compile a document, one needs:

1. to have a working LaTeX distribution[6],
2. to be in the correct working directory (containing the folder `ppdata` that includes all the output from the bbob_pproc),
3. that `templateBBOBarticle.tex`[7], `bbob.bib` and `sig-alternate.cls` are in the working directory (all files are provided with the software),

Then the following commands needs to be executed in a shell:

```
latex templateBBOBarticle
bibtex templateBBOBarticle
latex templateBBOBarticle
latex templateBBOBarticle
```

The document `templateBBOBarticle.dvi` is then generated in the format required for a GECCO workshop paper. An example of the resulting template document obtained by compiling the LaTeX template paper is provided here[8].

[6] http://www.latex-project.org/

[7] or `templateBBOBnoisyarticle.tex` for the noisy testbed of BBOB-2010.

[8] The figures and tables show the data of the Monte Carlo search on the noiseless testbed of BBOB-2009 [1].

Black-Box Optimization Benchmarking Template for Noiseless Function Testbed

Draft version [*]

Forename Name

ABSTRACT

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization—*global optimization, unconstrained optimization*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems

General Terms

Algorithms

Keywords

Benchmarking, Black-box optimization, Evolutionary computation

1. RESULTS

Results from experiments according to [2] on the benchmark functions given in [1, 3] are presented in Figures 1 and 2 and in Table 1.

2. REFERENCES

[1] S. Finck, N. Hansen, R. Ros, and A. Auger. Real-parameter black-box optimization benchmarking 2009: Presentation of the noiseless functions. Technical Report 2009/20, Research Center PPE, 2009.

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

[3] N. Hansen, S. Finck, R. Ros, and A. Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions. Technical Report RR-6829, INRIA, 2009.

[*]Camera-ready paper due April 17th.

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

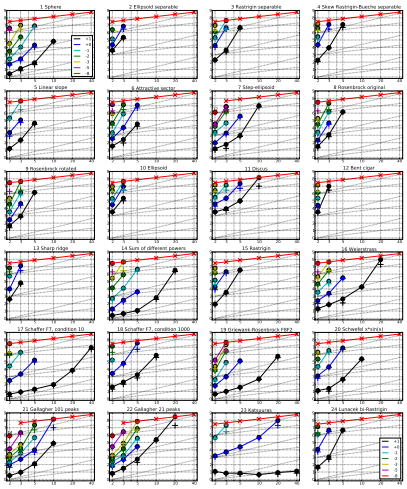

Figure 1: Expected Running Time (ERT, ●) to reach $f_{\mathrm{opt}} + \Delta f$ and median number of function evaluations of successful trials (+), shown for $\Delta f = 10, 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-5}, 10^{-8}$ (the exponent is given in the legend of f_1 and f_{24}) versus dimension in log-log presentation. The $\mathrm{ERT}(\Delta f)$ equals to $\#\mathrm{FEs}(\Delta f)$ divided by the number of successful trials, where a trial is successful if $f_{\mathrm{opt}} + \Delta f$ was surpassed during the trial. The $\#\mathrm{FEs}(\Delta f)$ are the total number of function evaluations while $f_{\mathrm{opt}} + \Delta f$ was not surpassed during the trial from all respective trials (successful and unsuccessful), and f_{opt} denotes the optimal function value. Crosses (×) indicate the total number of function evaluations $\#\mathrm{FEs}(-\infty)$. Numbers above ERT-symbols indicate the number of successful trials. Annotated numbers on the ordinate are decimal logarithms. Additional grid lines show linear and quadratic scaling.

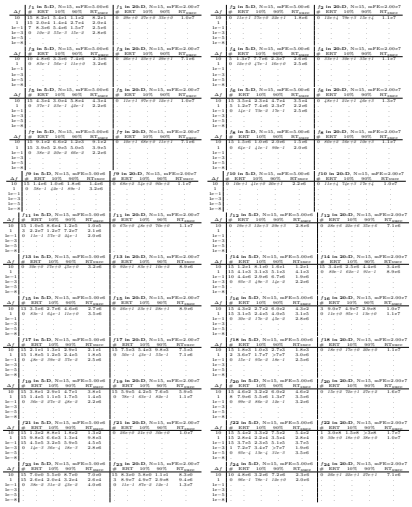

Table 1: Shown are, for a given target difference to the optimal function value Δf: the number of successful trials (#); the expected running time to surpass $f_{\mathrm{opt}} + \Delta f$ (ERT, see Figure 1); the 10%-tile and 90%-tile of the bootstrap distribution of ERT; the average number of function evaluations in successful trials or, if none was successful, as last entry the median number of function evaluations to reach the best function value ($\mathrm{RT}_{\mathrm{succ}}$). If $f_{\mathrm{opt}} + \Delta f$ was never reached, figures in *italics* denote the best achieved Δf-value of the median trial and the 10% and 90%-tile trial. Furthermore, N denotes the number of trials, and mFE denotes the maximum of number of function evaluations executed in one trial. See Figure 1 for the names of functions.

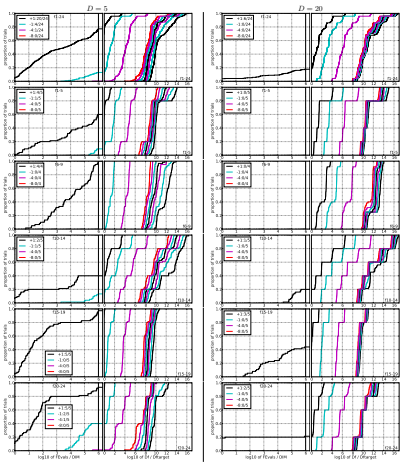

Figure 2: Empirical cumulative distribution functions (ECDFs), plotting the fraction of trials versus running time (left subplots) or versus Δf (right subplots). The thick red line represents the best achieved results. Left subplots: ECDF of the running time (number of function evaluations), divided by search space dimension D, to fall below $f_{\mathrm{opt}} + \Delta f$ with $\Delta f = 10^k$, where k is the first value in the legend. Right subplots: ECDF of the best achieved Δf divided by 10^k (upper left lines in continuation of the left subplot), and best achieved Δf divided by 10^{-8} for running times of $D, 10\,D, 100\,D \ldots$ function evaluations (from right to left cycling black-cyan-magenta). Top row: all functions; second row: separable functions; third row: misc. moderate functions; fourth row: ill-conditioned functions; fifth row: multi-modal functions with adequate structure; last row: multi-modal functions with weak structure. The legends indicate the number of functions that were solved in at least one trial. FEvals denotes number of function evaluations, D and DIM denote search space dimension, and Δf and Df denote the difference to the optimal function value.

The participants of BBOB-2010 are expected to fill in the template with all of their information, the description of their algorithm and their parameter settings [2], their source code or a reference to it, their results on the timing experiment. The BibTeX file `bbob.bib` includes the references to the BBOB-2010 experimental set-up and documentation.

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. The BBOBies would also like to acknowledge Mike Preuss for his implementation of the JNI for using the C-code in Java, and Petr Pošík for his help and feedback in the beta-tests.

References

[1] 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.

[2] N. Hansen, A. Auger, S. Finck, and R. Ros. Real-parameter black-box optimization benchmarking 2010: Experimental setup. Technical Report RR-7215, INRIA, 2010.

A Installing bbob_pproc

The entire post-processing tool is written in Python and requires Python to be installed on your machine. The minimal software requirements for using the post-processing tool are Python (2.5.2), Matplotlib (0.91.2) and Numpy (1.0.4). In the following, we explain how to obtain and install the required software for different systems (Linux, Windows, Mac OS) and which steps you have to perform to run the post-processing on your data.

While the `bbob_pproc` source files are provided, you need to install Python and its libraries Matplotlib and Numpy. We recommend using Python 2.6 and not a higher version (3.0, 3.1) since the necessary libraries are not (yet) available and the code is not verified.

A.1 Downloading the Packages

For all operating systems the packages can be found at the following locations:

- Python: http://www.python.org/download/releases/,
- Numpy: http://sourceforge.net/projects/numpy/,
- Matplotlib: http://sourceforge.net/projects/matplotlib/.

We recommend the use of the latest versions of Matplotlib (0.99.1.2), Python (2.6.4) and Numpy (1.4.0).

A.2 Installing on Linux

In most common Linux distributions Python (not Numpy or Matplotlib) is already part of the installation. If not, use your favorite package manager to install Python (package name: python), Numpy (python-numpy) and Matplotlib (package name: python-matplotlib) and their dependencies. If your distribution and repositories are up-to-date, you should have at least Python (2.6.4), Matplotlib (0.99.0) and Numpy (1.3.0). Though those are not the most recent versions of each package, they meet the minimal software requirements to make the BBOB-2010 software work. If needed, you can alternatively download sources and compile binaries. Python and the latest versions of Matplotlib and Numpy can be downloaded from the links in Section A.1. A dependency for the Linux version of Matplotlib is libpng, which can be obtained at http://www.libpng.org/. You then need to properly install the downloaded packages before you can use them. Please refer to the corresponding package installation pages.

A.3 Installing on Windows

For installing Python under Windows, please go to the Python link in Section A.1 and download `python-2.6.4.msi`. This file requires the Microsoft Installer, which is a part of Windows XP and later releases. If you don't have the Microsoft Installer, there is a link for the download provided at the same page. After installing Python, it is recommended to first install Numpy and then Matplotlib. Both can be installed with the standard .exe files which are respectively

- `numpy-1.4.0-win32-superpack-python2.6.exe` and,
- `matplotlib-0.99.1.win32-py2.6.exe`.

These files can be obtained from the provided SourceForge links in Section A.1.

A.4 Installing on Mac OS

Mac OS X comes with Python pre-installed, the version might be older than 2.6 though. It is recommended to upgrade Python by downloading and installing a newer version. To do this, if you have Mac OS X 10.3 and later you can download the disk image file `python-2.6.4_macosx10.3.dmg` containing universal binaries from the Python download page, see Section A.1. More information on the update of Python on Mac OS can be found at this location: http://www.python.org/download/mac/[9]. Open the disk image and use the installer[10]. You then need to download and install Numpy and Matplotlib from the SourceForge links listed in Sect A.1.

[9] The discussion over IDLE for Leopard user (http://wiki.python.org/moin/MacPython/Leopard) is not relevant for the use of bbob_pproc package.

[10] Following this step leave the pre-installed Python on the system and install the MacPython 2.6.4 distribution. MacPython contains a Python installation as well as some Mac-specific extras.