COCO (COmparing Continuous Optimizers) Software: User Documentation

Steffen Finck[*] and Raymond Ros[†]

compiled November 17, 2015

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 workshops named Black-Box Optimization Benchmarking (BBOB-2009 and BBOB-2010). 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 Science Vorarlberg, Hochschulstrasse 1, 6850 Dornbirn, Austria

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

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

The COCO software provides:

1. a single generic function interface fgeneric to the benchmark functions of BBOB-2010, coded in MATLAB/GNU OCTAVE, C/C++ and PYTHON,

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 interface of the considered algorithms compatible with the interface of our objective/fitness function implementation in fgeneric.

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 [1]. 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 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.

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

```matlab
function MY_OPTIMIZER(FUN, DIM, ftarget, maxfunevals)
% MY_OPTIMIZER(FUN, DIM, ftarget, maxfunevals)
% samples new points uniformly randomly in [-5,5]^DIM
% and evaluates them on FUN until ftarget of maxfunevals
% is reached, or until 1e8 * DIM fevals are conducted.
% Relies on FUN to keep track of the best point.

  maxfunevals = min(1e8 * DIM, maxfunevals);
  popsize = min(maxfunevals, 200);
  for iter = 1:ceil(maxfunevals/popsize)
    feval(FUN, 10 * rand(DIM, popsize) - 5);
    if feval(FUN, 'fbest') < ftarget  % task achieved
      break;
    end
    % if useful, modify more options here for next start
  end
```

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 or C++

The interface to fgeneric differs from the MATLAB example given in [1], 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.

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

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

3

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

```matlab
% 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
```

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 (2.6), Matplotlib (0.99) and Numpy (1.2). Please note that the packages do not work with Python 3.X but only with Python 2.X. The installation of the software is described in Appendix A.

Overview of the **bbob_pproc** Package

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

Listing 3: `exampletiming.m`: script for measuring the time complexity of MY_OPTIMIZER, see Listing 1, for BBOB-2010 in Matlab/Gnu Octave

```matlab
1   % runs the timing experiment for MY_OPTIMIZER. fgeneric.m
2   % and benchmarks.m must be in the path of MATLAB/Octave
3
4   addpath('PUT_PATH_TO_BBOB/matlab');  % should point to fgeneric.m etc.
5
6   more off;  % in octave pagination is on by default
7
8   timings = [];
9   runs = [];
10  dims = [];
11  for dim = [2,3,5,10,20,40]
12    nbrun = 0;
13    ftarget = fgeneric('initialize', 8, 1, 'tmp');
14    tic;
15    while toc < 30  % at least 30 seconds
16      MY_OPTIMIZER(@fgeneric, dim, ftarget, 1e5);  % adjust maxfunevals
17      nbrun = nbrun + 1;
18    end  % while
19    timings(end+1) = toc / fgeneric('evaluations');
20    dims(end+1) = dim;      % not really needed
21    runs(end+1) = nbrun;  % not really needed
22    fgeneric('finalize');
23    disp([['Dimensions:' sprintf(' %11d ', dims)]; ...
24           ['     runs:' sprintf(' %11d ', runs)]; ...
25           [' times [s]:' sprintf(' %11.1e ', timings)]]);
26  end
```

`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 $n > 0$ algorithms using modules from bbob_pproc.compall and outputs comparison figures and tables included in the template, `template3generic.tex`, `noisytemplate3generic.tex` [2],

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

`bwsettings.py` defines generic settings with figures in grayscale,

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

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

---

[2] for n larger than ten, modifications of the template will be required

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

```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;
}
```

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

pptex.py defines some routines for generating TeX tables,

ppfig.py defines some routines for generating figures,

ppfigdim.py, pptable.py, pprldistr.py, pplogloss.py 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.

3.1 Using the **bbob_pproc** Package

To perform the post-processing on the experimental data, the bbob_pproc package needs to be downloaded[3] and un-archived.

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

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

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

from a shell[4], 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/DATAPATH in the current working directory, which contain the post-processed data in the form of figures and LaTeX files for the tables. This process might take a few minutes.

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())
```

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

The resulting ppdatapath/DATAPATH folder now contains a number of TeX, eps, pdf 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/rungeneric.py -h
```

---

[3]The package can be obtained from http://coco.gforge.inria.fr/.

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

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 and more recent) 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/rungeneric.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 `ppdata` containing comparison figures and tables.

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

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

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.

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

```
 >>> import bbob_pproc as bb
 >>> bb.rungeneric.main('ALG1 ALG2 ALG3'.split())
```

or:

```
 >>> import bbob_pproc as bb
 >>> bb.rungeneric.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.

3.3 `rungeneric1.py`, `rungeneric2.py`, `rungenericmany.py`

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`:

```
python path_to_postproc_code/bbob_pproc/rungeneric1.py DATAPATH
```

or:

```
python path_to_postproc_code/bbob_pproc/rungeneric2.py ALG0 and ALG1
```

or:

```
python path_to_postproc_code/bbob_pproc/rungenericmany.py ALG1 ALG2 ALG3...
```

4 Generating a Paper

`template1generic.tex`, `template2generic.tex` and `template3generic.tex`
are the template LaTeX files that include all the figures and tables presenting
the result of an algorithm on the noiseless testbed of BBOB-2010.

`noisytemplate1generic.tex`, `noisytemplate2generic.tex` and
`noisytemplate3generic.tex` are the same template LaTeX files for the noisy
testbed of BBOB-2010. If compiled correctly using LaTeX, it generates documents collecting and organizing the output from bbob_pproc.

To compile a document, one needs:

1. to have a working LaTeX distribution[5],

2. to be in the correct working directory (containing the folder `ppdata` that
   includes all the output from the bbob_pproc),

3. that `templateXgeneric.tex`, where X is `1`, `2` or `3`[6], `bbob.bib` are in the
   working directory (all files are provided with the software),

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

```
latex templateXgeneric
bibtex templateXgeneric
latex templateXgeneric
latex templateXgeneric
```

The document `templateXgeneric.dvi` is then generated in a generic article
format. The BibTeX file `bbob.bib` includes the references to the BBOB-2010
experimental set-up and documentation.

Acknowledgments

---

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

[6] `template1.tex` for showing the results of one algorithm, `template2.tex`, `template3.tex`
for showing comparison results.

References

[1] 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 (2.6), Matplotlib (0.98) and Numpy (1.2). 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 or 2.7 and not a higher version (3.0, 3.1) since the necessary libraries are not available and the code is not verified. The code is currently tested using Python (2.6.5), Matplotlib (0.99.1.1) and Numpy (1.3.0).

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/.

Not all packages are distributed for all operating systems with 64-bit architecture. Installing 32-bit versions of the packages whether your architecture is 32 or 64-bit is possible though.

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 (2.6.5), Matplotlib (0.99.1.1) 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.X.msi`. This file requires the Microsoft Installer, which is a part of Windows XP and later releases. 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-1.5.0b1-win32-superpack-python2.6.exe` and,

- `matplotlib-1.0.0.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.X_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/[7]. Open the disk image and use the installer[8]. You then need to download and install Numpy and Matplotlib from the SourceForge links listed in Sect A.1.

---

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

[8]Following this step leave the pre-installed Python on the system and install the MacPython 2.X distribution. MacPython contains a Python installation as well as some Mac-specific extras.