Tutorial: Algorithms for Multiobjective Optimization and How to Benchmark Them

Dimo Brockhoff
dimo.brockhoff@inria.fr
http://www.cmap.polytechnique.fr/~dimo.brockhoff/

SSBSE'2017, Paderborn, Germany
September 11, 2017
Understanding Multiobjective Optimization Algorithms using Benchmarks

... an alternative title suggested by Tim Menzies

Dimo Brockhoff
dimo.brockhoff@inria.fr
http://www.cmap.polytechnique.fr/~dimo.brockhoff/

SSBSE'2017, Paderborn, Germany
September 11, 2017
How to Benchmark Blackbox Optimization Algorithms
(and how to do it concretely for Multiobjective Optimization)

Dimo Brockhoff
dimo.brockhoff@inria.fr
http://www.cmap.polytechnique.fr/~dimo.brockhoff/

SSBSE'2017, Paderborn, Germany
September 11, 2017
Disclaimer

I am not an expert in search based software engineering 😊
Given:

\[ x \in \mathbb{R}^n \quad \rightarrow \quad f(x) \in \mathbb{R}^k \]

\[ g(x) \in \mathbb{R}^m \]

Main practical question:

Which of the many algorithms should I use on my problem?
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking? hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms incl. visual results and first recommendations
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking?  
   hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms  
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms  
   incl. visual results and first recommendations
Why do we need Benchmarking?

Goals:

- understanding of algorithms
- algorithm selection
- putting algorithms to a standardized test
  - simplify judgement
  - simplify comparison
  - regression test under algorithm changes

Kind of everybody has to do it (and it is tedious):

- choosing (and implementing) problems, performance measures, visualization, stat. tests, ...
- running a set of algorithms
that's why we have COCO 😊

Comparing Continuous Optimizers Platform

https://github.com/numbbo/coco
automatized benchmarking
benchmarking is non-trivial
[remember the tutorial of Antonio]
hence, COCO implements a reasonable, well-founded, and well-documented pre-chosen methodology
How to benchmark algorithms with COCO?
https://github.com/numbbo/coco
https://github.com/numbbo/coco
https://github.com/numbbo/coco
**numbbo/coco: Comparing Continuous Optimizers**

This code repository is initialized to support Continuous Optimization Algorithms (COO). It provides a suite of tools and algorithms for comparing continuous optimization methods. The repository includes various components:

- **code-experiments**: A little more verbose error message when suite regression test fails.
- **code-postprocessing**: Hashes are back on the plots.
- **code-preprocessing**: Fixed preprocessing to work correctly with the extended biobjective.
- **howtos**: Update create-a-suite-howto.md
- **.clang-format**: raising an error in bbo2009_logger.c when best_value is NULL. Plus s...
- **.hgignore**: raising an error in bbo2009_logger.c when best_value is NULL. Plus s...
- **AUTHORS**: small correction in AUTHORS
- **LICENSE**: Update LICENSE
- **README.md**: Added link to #1335 before closing.
- **do.py**: refactoring here and there in do.py to get closer to PEP8 specifications
- **doxygen.ini**: moved all files into code-experiments/ folder besides the do.py script

This repository is actively maintained, and contributions are encouraged for the continuous improvement of continuous optimization algorithms.
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers. AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
- Python

Contributions to link further languages (including a better example in C++) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description
numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continuous Optimizer platform, now rewritten fully in ANSI C with other languages calling the C code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- C/C++
- Java
- MATLAB/Octave
- Python

Contributions to link further languages (including a better example in C++) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description
- see the bbob-biobj and bbob-biobj-ext COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbeds.
- consult the BBOB workshops series,
- consider to register here for news,
- see the previous COCO home page here and
- see the links below to learn more about the ideas behind CoCO.
0. Check out the Requirements above.

1. Download the COCO framework code from github,
   - either by clicking the Download ZIP button and unzip the zip file,
   - or by typing `git clone https://github.com/numbbo/coco.git`. This way allows to remain up-to-date easily (but needs git to be installed). After cloning, `git pull` keeps the code up-to-date with the latest release.

The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

2. In a system shell, cd into the coco or coco-<version> folder (framework root), where the file do.py can be found.
   Type, i.e. execute, one of the following commands once

   python do.py run-c
   python do.py run-java
   python do.py run-matlab
   python do.py run-octave
   python do.py run-python

   depending on which language shall be used to run the experiments. run-* will build the respective code and run the example experiment once. The build result and the example experiment code can be found under code-experiments/build/<language> ( <language>=matlab for Octave). python do.py lists all available commands.

3. On the computer where experiment data shall be post-processed, run

   python do.py install-postprocessing
Getting Started

0. Check out the Requirements above.

1. Download the COCO framework code from github,
   - either by clicking the Download ZIP button and unzip the zip file,
   - or by typing git clone https://github.com/numbbo/coco.git. This way allows to remain up-to-date easily (but needs git to be installed). After cloning, git pull keeps the code up-to-date with the latest release.

The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

2. In a system shell, cd into the coco or coco-<version> folder (framework root), where the file do.py can be found. Type, i.e. execute, one of the following commands:

```
python do.py run-c
python do.py run-java
python do.py run-matlab
python do.py run-octave
python do.py run-python
```

depending on which language shall be used to run the experiments. run-* will build the respective code and run the example experiment once. The build result and the example experiment code can be found under code-experiments/build/<language> (in Octave: python do.py lists all available commands.

3. On the computer where experiment data shall be post-processed, run

```
python do.py install-postprocessing
```
installation II: postprocessing

python do.py install-postprocessing

4. Copy the folder code-experiments/build/YOUR-FAVORITE-LANGUAGE and its content to another location. In Python it is sufficient to copy the file example_experiment.py. Run the example experiment (it already is compiled). As the details vary, see the respective read-me's and/or example experiment files:

- C read me and example experiment
- Java read me and example experiment
- Matlab/Octave read me and example experiment
- Python read me and example experiment

If the example experiment runs, connect your favorite algorithm to Coco: replace the call to the random search optimizer in the example experiment file by a call to your algorithm (see above). Update the output result_folder, the algorithm_name and algorithm_info of the observer options in the example experiment file.

Another entry point for your own experiments can be the code-experiments/examples folder.

5. Now you can run your favorite algorithm on the bbo suite (for single-objective algorithms) or on the bbo-biobj and bbo-biobj-ext suites (for multi-objective algorithms). Output is automatically generated in the specified data result_folder. By now, more suites might be available, see below.
https://github.com/numbbo/coco

coupling algo + COCO
import cocoex
import scipy.optimize

### input
suite_name = "bbob"
output_folder = "scipy-optimize-fmin"
fmin = scipy.optimize.fmin

### prepare
suite = cocoex.Suite(suite_name, "", "")
observer = cocoex.Observer(suite_name,
                           "result_folder: " + output_folder)

### go
for problem in suite:  # this loop will take several minutes
    problem.observe_with(observer)  # generates the data for
    # cocopp post-processing
    fmin(problem, problem.initial_solution)

Note: the actual example_experiment.py contains more advanced things like restarts, batch experiments, other algorithms (e.g. CMA-ES), etc.
Another entry point for your own experiments can be the code-experiments/examples folder.

5. Now you can run your favorite algorithm on the bbo suite (for single-objective algorithms) or on the bbo-biobj and bbo-biobj-ext suites (for multi-objective algorithms). Output is automatically generated in the specified data result_folder. By now, more suites might be available, see below.

6. Postprocess the data from the results folder by typing

   python -m cocopp [-o OUTPUT_FOLDERNAME] YOURDATAFOLDER

Any subfolder in the folder arguments will be searched for, resulting in different folders collected under a single "root" YOURDATAFOLDER folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.

A folder, pdata by default, will be generated, which contains all output from the post-processing, including an index.html file, useful as main entry point to explore the result with a browser. Data might be overwritten, it is therefore useful to change the output folder name with the -o OUTPUT_FOLDERNAME option.

A summary pdf can be produced via LaTeX. The corresponding templates can be found in the code-postprocessing/latex-templates folder. Basic html output is also available in the result folder of the postprocessing (file templateBBOBarticle.html).

7. Once your algorithm runs well, increase the budget in your experiment script, if necessary implement randomized independent restarts, and follow the above steps successively until you are happy.

8. The experiments can be parallelized with any re-distribution of single problem instances to batches (see example_experiment.py for an example). Each batch must write in a different target folder (this should happen automatically). Results of each batch must be kept under their separate folder as is. These folders then must be stored in the root folder.
Another entry point for your own experiments can be the `code-experiments/examples` folder.

5. Now you can run your favorite algorithm on the `bbob` suite (for single-objective algorithms) or on the `bbob-biobj` and `bbob-biobj-xt` suites (for multi-objective algorithms). Output is automatically generated in the specified data result folder. By now, more suites might be available, see below.

6. Postprocess the data from the results folder by typing

```
python -m cocopp [-o OUTPUT_FOLDERNAME] YOURDATAFOLDER [MORE_DATAFOLDERS]
```

Any subfolder in the folder arguments will be searched for logged data. That is, experiments from different batches can be in different folders collected under a single "root" YOURDATAFOLDER specifying several data result folders generated by different algorithms.

A folder, `ppdata`, by default, will be generated, which contains a single file, useful as main entry point to explore the result with a browser. The output folder name with the -o OUTPUT_FOLDERNAME option.

A summary pdf can be produced via LaTeX. The corresponding templates can be found in the `code-postprocessing/latex-templates` folder. Basic html output is also available in the result folder of the postprocessing (file templateBBOBarticle.html).

7. Once your algorithm runs well, increase the budget in your experiment script, if necessary implement randomized independent restarts, and follow the above steps successively until you are happy.

8. The experiments can be parallelized with any redistribution of single problem instances to batches (see `example_experiment.py` for an example). Each batch must write in a different target folder (this should happen automatically). Results of each batch must be kept under their separate folder as is. These folders then must be moved in (in the `batches/batch1/...` folder) following the same naming scheme (e.g., `batch1/2023-04-01` for the folder).
## Result Folder

![Folder Contents](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIPOP-CMA-ES_hansen_noiseless</td>
<td>03/06/2017 11:33</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>cocopp_commands.tex</td>
<td>03/06/2017 11:33</td>
<td>LaTeX Document</td>
<td>7 KB</td>
</tr>
<tr>
<td>index.html</td>
<td>03/06/2017 11:33</td>
<td>Firefox HTML Doc...</td>
<td>1 KB</td>
</tr>
<tr>
<td>ppdata.html</td>
<td>03/06/2017 11:33</td>
<td>Firefox HTML Doc...</td>
<td>1 KB</td>
</tr>
</tbody>
</table>
_post processing results_

Single algorithm data

BIPOP-CMA-ES hansen noiseless
BIPOP-CMA-ES

Home

Runtime distributions (ECDFs) per function

Runtime distributions (ECDFs) summary and function groups

Scaling with dimension for selected targets

Tables for selected targets

Runtime distribution for selected targets and f-distributions

Runtime loss ratios

Runtime distributions (ECDFs) over all targets

- bobb - f1-f24
- 51 targets in 100.1e-08
- 15 instances
Overview page

Runtime distributions (ECDFs) per function

1. Sphere
2. Ellipsoid separable
3. Rastrigin separable
4. Skew Rastrigin-Bueche separable
5. Linear slope
6. Attractive sector
7. Steepest ellipsoid
8. Rosenbrock original
9. Rosenbrock rotated
10. Ellipsoid
11. Discus
12. Bent cigar
13. Sharp ridge
14. Sum of different powers
15. Rastrigin
16. Weierstrass
Overview page

Average number of $f$-evaluations to reach target

1. Sphere
2. Ellipsoid separable
3. Rastrigin separable
4. Skew Rastrigin-Bueche separable
5. Linear slope
6. Attractive sector
7. Step-ellipsoid
8. Rosenbrock original
9. Rosenbrock rotated
10. Ellipsoid
11. Discus
12. Bent cigar
13. Sharp ridge
14. Sum of different powers
15. Rastrigin
16. Weierstrass
17. Schaffer F7, condition 10
18. Schaffer F7, condition 1000
19. Griewank-Rosenbrock F8F2
20. Schwefel $x \sin(x)$
doesn't look too complicated, does it?

[the devil is in the details 😊]
so far:

data for about 170 algorithm variants
(some of which on noisy or multiobjective test functions)
132 workshop papers
by 101 authors from 28 countries
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking? hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms
   incl. visual results and first recommendations
Talk Outline

1. Why do we do benchmarking?
2. What **pitfalls** to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking? hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms
   incl. visual results and first recommendations
All of the following is...

- not restricted to numerical optimization
- but assumes blackbox optimization

pitfall #0:
we don't benchmark algorithms but algorithm implementations!
Measuring Performance

On

- real world problems
  - expensive
  - comparison typically limited to certain domains
  - experts have limited interest to publish

- "artificial" benchmark functions
  - cheap
  - controlled
  - data acquisition is comparatively easy
  - problem of representativeness
Test Functions

- define the "scientific question"
  - the relevance can hardly be overestimated
- should represent "reality"
- are often too simple?
  - remind separability
- a number of test suites are around

- account for invariance properties
  - prediction of performance is based on “similarity”, ideally equivalence classes of functions
Available Test Suites in COCO

<table>
<thead>
<tr>
<th>Suite</th>
<th>No. of Noiseless Functions</th>
<th>No. of Noisy Functions</th>
<th>No. of Algorithm Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbob</td>
<td>24</td>
<td>30</td>
<td>150+</td>
</tr>
<tr>
<td>bbob-noisy</td>
<td></td>
<td></td>
<td>40+</td>
</tr>
<tr>
<td>bbob-biobj</td>
<td>55</td>
<td>55</td>
<td>16</td>
</tr>
</tbody>
</table>

**Pitfall #3:**
- too few functions
  - overfitting

**Pitfall #4:**
- no baselines
  - improving over bad algorithms is easy
Available Test Suites in COCO

- **bbob**: 24 noiseless fcts, 150+ algo data sets
- **bbob-noisy**: 30 noisy fcts, 40+ algo data sets
- **bbob-biobj**: 55 bi-objective fcts, 16 algo data sets

**Under development:**
- extended bi-objective suite (bbob-biobj-ext)
- large-scale version of bbob (bbob-largescale)
- constrained test suite (bbob-constrained)

**Long-term goals:**
- combining difficulties
- almost real-world problems
- real-world problems
Examples of Scientific Questions

Sphere function (bbob f1)
- What is the optimal convergence rate of an algorithm?

Ellipsoid function (bbob f2)
- In comparison to f1: Is symmetry exploited?
- In comparison to f10: Is separability exploited?

Rastrigin function (bbob f3)
- compared to e.g. f2: What is the effect of multimodality?

Büche-Rastrigin function (bbob f4)
- In comparison to f3: What is the effect of asymmetry?

Linear Slope (bbob f5)
- Can the search go outside the initial convex hull of solutions into the domain boundary?
- Can the step size be increased accordingly?
How Do We Measure Performance?

We want a meaningful quantitative measure:

- quantitative on the ratio scale (highest possible)
  "algo A is two times better than algo B" shall be a meaningful statement
- assume a wide range of values
- meaningful (interpretable) with regard to the real world
  possible to transfer from benchmarking to real world
How Do We Measure Performance?

Actually two objectives:

- Find solution with small(est possible) function/indicator value
- With the least possible search costs (number of function evaluations)

For measuring performance:
fix one and measure the other

pitfall #5 [in blackbox opt. at least]: measure cpu times
Measuring Performance Empirically

convergence graphs is all we have to start with...

![Graph showing convergence graphs with fixed target and fixed budget]
Runtime is the prime candidate

Why?

- runtimes are easier to interpret than f-values
- runtimes are interpretable in a meaningful way
- runtimes have a natural zero (hence: ratio scale)

Another reason:

- in both cases, we have missing values
- in the fixed budget view, we miss values of "too good" algorithms
- in the fixed target view, we miss values of "bad/slow" algorithms
we (should) display ECDFs:
Empirical Cumulative Distribution Functions of the Runtime
[aka data profiles*]

A Convergence Graph

function value vs. $\log_{10}(\text{function evaluations})$
Runtime to hit target is Monotonous

\[ \log_{10}(\text{function evaluations}) \]
15 Runs

The graph shows the function value on the y-axis, with a logarithmic scale, against the number of function evaluations on the x-axis, also on a logarithmic scale. The data points are represented in various colors, indicating multiple runs or trials. The graph illustrates the convergence of the algorithms as the number of evaluations increases.
15 Runs ≤ 15 Runtime Data Points

function value vs. \( \log_{10}(\text{function evaluations}) \)

Target
Empirical Cumulative Distribution

the ECDF of run lengths to reach a target

- has for each data point a vertical step of constant size
- displays for each x-value (budget) the count of observations to the left (first hitting times)
Empirical Cumulative Distribution

In the ECDF we can read that e.g. 60% of the runs need between 2000 and 4000 evaluations. e.g. 80% of the runs (=12/15) reached the target.
Reconstructing A Single Run

![Graph showing function value vs. log10(function evaluations)]
Reconstructing A Single Run

50 equally spaced targets
Reconstructing A Single Run
Reconstructing A Single Run
Reconstructing A Single Run

The empirical CDF makes a step for each star, is monotonous and displays for each budget the fraction of targets achieved within the budget.
Reconstructing A Single Run

The ECDF recovers the monotonous graph, discretized and flipped.
Reconstructing A Single Run

the ECDF recovers the monotonous graph, discretized and flipped
Aggregation

function value

$\log_{10}(\text{function evaluations})$
Aggregation

15 runs
50 targets
Aggregation

15 runs
50 targets
Aggregation

- Pitfall #7a: target choice
- Pitfall #7b: single target/single budget

15 runs
50 targets
ECDF with 750 steps
Interpretation

area over the ECDF curve
  =
average log runtime
  (or geometric avg. runtime)
over all targets (difficult and easy)
  and all runs
Fixed-target: Measuring Runtime

\[ p_s(\text{Algo A}) < 1, \text{ fast convergence} \]

\[ p_s(\text{Algo B}) \approx 1, \text{ slow convergence} \]

pitfall #8
Fixed-target: Measuring Runtime

• Algo Restart A:

• Algo Restart B:
Fixed-target: Measuring Runtime

- Expected running time of the restarted algorithm:

\[ E[RT^r] = \frac{1 - p_s}{p_s} E[RT_{unsuccessful}] + E[RT_{successful}] \]

- Estimator average running time (aRT):

\[ \hat{p}_s = \frac{\#successes}{\#runs} \]

\[ RT_{unsucc} = \text{Average evals of unsuccessful runs} \]

\[ RT_{succ} = \text{Average evals of successful runs} \]

\[ aRT = \frac{\text{total \#evals}}{\#successes} \]
ECDFs with Simulated Restarts

What we typically plot are ECDFs of the simulated restarted algorithms:

![Graph showing ECDFs with Simulated Restarts](image)
ECDFs with Simulated Restarts

What we typically plot are ECDFs of the simulated restarted algorithms:

pitfall #9: aggregating over dimension
→ dimension is input to the algorithm
Last Pitfall #10 😊

Other advantages of ECDF plots: we can also
- aggregate over functions
- show data of more than 1 algorithm at a time

150 algorithms from BBOB-2009 till BBOB-2015
More Automated Plots with COCO

…but no time to explain them here 😞
More Automated Plots with COCO

...but no time to explain them here
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking?
   hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms
   incl. visual results and first recommendations
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking? hands-on tutorial
4. Available and Interesting Multiobjective Optimization Algorithms
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms
   incl. visual results and first recommendations
Multiobjective Blackbox Problems

Given:

\[ x \in \mathbb{R}^n \quad \rightarrow \quad f(x) \in \mathbb{R}^k \]

\[ k \geq 2 \]

Same practical question to ask:

Which of the many algorithms should I use on my problem?
Many, Many Algorithms Available

...and everyday people propose more

The "Classics":
- NSGA-II
- SPEA2

Indicator-Based:
- IBEA
- HypE
- SMS-EMOA
- MO-CMA-ES

Decomposition-Based:
- MOEA/D
- ParEGO

Others:
- RM-MEDA
- Hybrids/Portfolios/
- Random search
- Derivative-Free
  Optimization methods
Algorithms in jMetal

- NSGA-II
  - variants: ssNSGAII, NSGAIIadaptive, NSGAIIrandom
- SPEA2
- PAES
- PESA-II
- OMOPSO
- MOCell
- AbYSS
- MOEA/D
- GDE3
- IBEA
- SMPSO
- SMPSOhv
- SMS-EMOA

- WASF-GA
- GWASF-GA
- MOEA/D-STM (> v5.0)
- MOEA/D-DE (> v5.0)
- MOCHC (> v5.0)
- MOMBI (> v5.0)
- NSGA-III (> v5.0)

Further (only in jmetal v4.5):

- FastPGA
- dMOPSO
- Densea
- CellDE
### Available bbob-biobj Data Sets

The following table lists all submitted algorithm data sets on the bbob-biobj test suite, related to the BBOB-2016 and BBOB-2017 workshops together with links to their corresponding papers. In order to sort the table according to some columns, please click on the corresponding table header. If available, the source codes of the algorithms can be downloaded by clicking on the link with the corresponding algorithm name in the second column.

<table>
<thead>
<tr>
<th>No</th>
<th>Algorithm</th>
<th>Year</th>
<th>Author(s)</th>
<th>COCO data</th>
<th>related PDFs and Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>biobj-001</td>
<td>DEMO</td>
<td>2016</td>
<td>Tušar and Filipić</td>
<td>data</td>
<td>slides</td>
</tr>
<tr>
<td>biobj-002</td>
<td>GA-MULTIOBJ-NSGA-II</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-003</td>
<td>HMO-CMA-ES</td>
<td>2016</td>
<td>Loshchilov and Glasmachers</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-004</td>
<td>MAT-DIRECT</td>
<td>2016</td>
<td>Al-Dujaili and Sundaram</td>
<td>data</td>
<td>slides</td>
</tr>
<tr>
<td>biobj-005</td>
<td>MAT-SMS</td>
<td>2016</td>
<td>Al-Dujaili and Sundaram</td>
<td>data</td>
<td>slides</td>
</tr>
<tr>
<td>biobj-006</td>
<td>MO-DIRECT-HV-Rank</td>
<td>2016</td>
<td>Wong et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-007</td>
<td>MO-DIRECT-ND</td>
<td>2016</td>
<td>Wong et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-008</td>
<td>MO-DIRECT-Rank</td>
<td>2016</td>
<td>Wong et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-009</td>
<td>RM-MEDA</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-010</td>
<td>RS-4</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-011</td>
<td>RS-5</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-012</td>
<td>RS-100</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-013</td>
<td>SMS-EMOA-DE</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-014</td>
<td>SMS-EMOA-PM</td>
<td>2016</td>
<td>Auger et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-014</td>
<td>UP-MO-CMA-ES</td>
<td>2016</td>
<td>Krause et al.</td>
<td>data</td>
<td>PDF</td>
</tr>
<tr>
<td>biobj-015</td>
<td>SMS-EMOA-SA</td>
<td>2017</td>
<td>Wessing</td>
<td>data</td>
<td>n/a yet</td>
</tr>
</tbody>
</table>
Talk Outline

1. Why do we do benchmarking?
2. What pitfalls to avoid in blackbox benchmarking?
3. How to use the COCO platform for benchmarking? (hands-on tutorial)
4. Available and Interesting Multiobjective Optimization Algorithms
   only shortly, refer to Antonio's talk from Saturday
5. Case Study of Benchmarking 16 Multiobjective Optimization Algorithms
   incl. visual results and first recommendations
The **bbob-biobj Testbed**

- **55 functions** by combining 2 **bbob** functions

<table>
<thead>
<tr>
<th>1 Separable Functions</th>
<th>4 Multi-modal functions with adequate global structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1 Sphere Function ✓</td>
<td>f15 Rastrigin Function ✓</td>
</tr>
<tr>
<td>f2 Ellipsoidal Function ✓</td>
<td>f16 Weierstrass Function</td>
</tr>
<tr>
<td>f3 Rastrigin Function</td>
<td>f17 Schaffers F7 Function ✓</td>
</tr>
<tr>
<td>f4 Büche-Rastrigin Function</td>
<td>f18 Schaffers F7 Functions, moderately ill-conditioned</td>
</tr>
<tr>
<td>f5 Linear Slope</td>
<td>f19 Composite Griewank-Rosenbrock Function F8F2</td>
</tr>
<tr>
<td>2 Functions with low or moderate conditioning</td>
<td><strong>reasoning:</strong> have problems with practically relevant features</td>
</tr>
<tr>
<td>f6 Attractive Sector Function ✓</td>
<td></td>
</tr>
<tr>
<td>f7 Step Ellipsoidal Function</td>
<td></td>
</tr>
<tr>
<td>f8 Rosenbrock Function, original ✓</td>
<td></td>
</tr>
<tr>
<td>f9 Rosenbrock Function, rotated</td>
<td></td>
</tr>
<tr>
<td>3 Functions with high conditioning and unimodal</td>
<td></td>
</tr>
<tr>
<td>f10 Ellipsoidal Function</td>
<td></td>
</tr>
<tr>
<td>f11 Discus Function</td>
<td></td>
</tr>
<tr>
<td>f12 Bent Cigar Function</td>
<td></td>
</tr>
<tr>
<td>f13 Sharp Ridge Function ✓</td>
<td></td>
</tr>
<tr>
<td>f14 Different Powers Function ✓</td>
<td></td>
</tr>
</tbody>
</table>
The bbob-biobj Testbed

- **55 functions** by combining 2 bbob functions

- **6 dimensions**: 2, 3, 5, 10, 20, (40 optional)
Notion of Instances

- All COCO problems come in form of instances
  - e.g. as translated/rotated versions of the same function

- Avoids to step in another pitfall:
  - how to compare deterministic and stochastic algorithms?

- Prescribed instances typically change over the years
  - avoid overfitting
  - 5 instances are always kept the same

Plus:
- the bbob functions are locally perturbed by non-linear transformations
Notion of Instances

- All COCO problems come in form of instances

$f_{10}$ (Ellipsoid)

$f_{15}$ (Rastrigin)
Bi-objective Performance Assessment

algorithm quality =

\[
\text{normalized* hypervolume (HV) of all non-dominated solutions}
\]

\text{if a point dominates nadir}

\[
\text{closest normalized* negative distance to region of interest [0,1]^2}
\]

\text{if no point dominates nadir}

* such that ideal=[0,0] and nadir=[1,1]
Bi-objective Performance Assessment

We measure runtimes to reach (HV indicator) targets:

- relative to a reference set, given as the best Pareto front approximation known (since exact Pareto set not known)
  
  current_best values incl. all non-dominated points found by the 15 workshop algos of BBOB-2016 (expected to improve over time)

- absolute hypervolume targetprecisions
  
  \[ \text{HV(refset)} - \text{targetprecision} \]

  are used with 58 fixed targetprecisions in
  \{ -10^{-4}, -10^{-4.2}, ..., -10^{-5}, 0, 10^{-5}, 10^{-4.9}, 10^{-4.8}, ..., 1 \}

  same for all dimensions, functions, and instances
Remarks

- in principle the same for many objectives as well, but...
- not yet implemented in COCO 😊
- and also slower of course than for 2 objectives
  - logarithmic insert and hv update for 2-objectives
  - linear in 2-objectives
  - quadratic in 3-objectives
  - follow http://simco.gforge.inria.fr/
- and hypervolume might not be meaningful in higher dimensions (to be discussed)
- will in principle also work with non-continuous problems (despite the abbreviation "COCO")
link to bbo-biobj data
link to bboib-biobj data (expensive)
Conclusions

- Numerical **benchmarking** is a **compulsory task** to assess performance of practical algorithms
- Many **pitfalls** need to be avoided to provide **meaningful comparisons** and recommendations
- **COCO** is one way to **automatize** all this

We need your help 😊

- compare algorithms and send us the data
- send papers to our workshops (e.g. BBOB-2017 in Kyoto)
- find bugs or solve issues, give us feedback
- contribute new test suites
- contribute new tests 😊 and your knowledge on SBSE
- [https://github.com/numbbo/coco/](https://github.com/numbbo/coco/)

Thank you!