Introduction to Optimization: Benchmarking

September 13, 2016
TC2 - Optimisation
Université Paris-Saclay, Orsay, France

Anne Auger
Inria Saclay – Ile-de-France

Dimo Brockhoff
Inria Saclay – Ile-de-France
# Course Overview

<table>
<thead>
<tr>
<th>Day</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fri, 16.9.2016</td>
<td>Introduction to Optimization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>groups defined via wiki</td>
</tr>
<tr>
<td></td>
<td></td>
<td>everybody went (actively!) through the Getting Started part of github.com/numbbo/coco</td>
</tr>
<tr>
<td>2</td>
<td>Fri, 23.9.2016</td>
<td>Today's lecture: Benchmarking; <strong>final adjustments of groups</strong> everybody can run and postprocess the example experiment (~1h for final questions/help during the lecture)</td>
</tr>
<tr>
<td>3</td>
<td>Fri, 30.9.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>4</td>
<td>Fri, 7.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>5</td>
<td>Fri, 14.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>6</td>
<td>Tue, 18.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>Tue, 18.10.2016</td>
<td>deadline for submitting data sets</td>
</tr>
<tr>
<td></td>
<td>Fri, 21.10.2016</td>
<td>deadline for paper submission</td>
</tr>
<tr>
<td></td>
<td>Fri, 4.11.2016</td>
<td>vacation</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Final lecture</td>
</tr>
<tr>
<td></td>
<td>7.-11.11.2016</td>
<td>oral presentations (individual time slots)</td>
</tr>
<tr>
<td></td>
<td>14 - 18.11.2016</td>
<td>Exam (exact date to be confirmed)</td>
</tr>
</tbody>
</table>

**All deadlines:** 23:59pm Paris time
## Course Overview

<table>
<thead>
<tr>
<th></th>
<th>Fri, 16.9.2016</th>
<th>Introduction to Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wed, 21.9.2016</td>
<td>groups defined via wiki</td>
</tr>
<tr>
<td></td>
<td>Thu, 22.9.2016</td>
<td>everybody went (actively!) through the Getting Started part of <a href="http://github.com/numbbo/coco">github.com/numbbo/coco</a></td>
</tr>
<tr>
<td>2</td>
<td>Fri, 23.9.2016</td>
<td>Today's lecture: ❶ Benchmarking; ❷ final adjustments of groups</td>
</tr>
<tr>
<td></td>
<td></td>
<td>everybody can run and postprocess the example experiment (~1h for final ❸ questions/help during the lecture)</td>
</tr>
<tr>
<td>3</td>
<td>Fri, 30.9.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>4</td>
<td>Fri, 7.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>Mon, 10.10.2016</td>
<td>deadline for intermediate wiki report: what has been done and what remains to be done?</td>
</tr>
<tr>
<td>5</td>
<td>Fri, 14.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>6</td>
<td>Tue, 18.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>Tue, 18.10.2016</td>
<td>deadline for submitting data sets</td>
</tr>
<tr>
<td></td>
<td>Fri, 21.10.2016</td>
<td>deadline for paper submission</td>
</tr>
<tr>
<td>7</td>
<td>Fri, 4.11.2016</td>
<td>vacation</td>
</tr>
<tr>
<td></td>
<td>Final lecture</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.-11.11.2016</td>
<td>oral presentations (individual time slots)</td>
</tr>
<tr>
<td></td>
<td>14 - 18.11.2016</td>
<td>Exam (exact date to be confirmed)</td>
</tr>
</tbody>
</table>

**All deadlines: 23:59pm Paris time**
challenging optimization problems appear in many scientific, technological and industrial domains
Numerical Blackbox Optimization

Optimize $f: \Omega \subset \mathbb{R}^n \mapsto \mathbb{R}^k$

$\mathbf{x} \in \mathbb{R}^n \quad \Rightarrow \quad f(\mathbf{x}) \in \mathbb{R}^k$

derivatives not available or not useful
Practical Blackbox Optimization

Given:

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

Not clear:
which of the many algorithms should I use on my problem?
Numerical Blackbox Optimizers

Deterministic algorithms
Quasi-Newton with estimation of gradient (BFGS) [Broyden et al. 1970]
Simplex downhill [Nelder & Mead 1965]
Pattern search [Hooke and Jeeves 1961]
Trust-region methods (NEWUOA, BOBYQA) [Powell 2006, 2009]

Stochastic (randomized) search methods
Evolutionary Algorithms (continuous domain)
  • Differential Evolution [Storn & Price 1997]
  • Particle Swarm Optimization [Kennedy & Eberhart 1995]
  • Evolution Strategies, CMA-ES [Rechenberg 1965, Hansen & Ostermeier 2001]
  • Estimation of Distribution Algorithms (EDAs) [Larrañaga, Lozano, 2002]
  • Cross Entropy Method (same as EDA) [Rubinstein, Kroese, 2004]
  • Genetic Algorithms [Holland 1975, Goldberg 1989]
Simulated annealing [Kirkpatrick et al. 1983]
Simultaneous perturbation stochastic approx. (SPSA) [Spall 2000]
Numerical Blackbox Optimizers

Deterministic algorithms
Quasi-Newton with estimation of gradient (BFGS) [Broyden et al. 1970]
Simplex downhill [Nelder & Mead 1965]
Pattern search [Hooke and Jeeves 1961]
Trust-region methods (NEWUOA, BOBYQA) [Powell 2006, 2009]

Stochastic (randomized) search methods
Evolutionary Algorithms (continuous domain)
• Differential Evolution [Storn & Price 1997]
• Particle Swarm Optimization [Kennedy & Eberhart 1995]
• Evolution Strategies, CMA-ES [Rechenberg 1965, Hansen&Ostermeier 2001]
• Estimation of Distribution Algorithms (EDAs) [Larrañaga, Lozano, 2002]
• Cross Entropy Method (same as EDA) [Rubinstein, Kroese, 2004]
• Genetic Algorithms [Holland 1975, Goldberg 1989]
Simulated annealing [Kirkpatrick et al. 1983]
Simultaneous perturbation stochastic approx. (SPSA) [Spall 2000]

• choice typically not immediately clear
• although practitioners have knowledge about problem difficulties (e.g. multi-modality, non-separability, ...)

Numerical Blackbox Optimizers
Need: Benchmarking

• 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 where COCO comes into play

Comparing Continuous Optimizers Platform
https://github.com/numbbo/coco
automatized benchmarking
How to benchmark algorithms with COCO?
Numerical Black-Box Optimization Benchmarking Framework http://coco.gforge.inria.fr/

- 7,902 commits
- 12 branches
- 25 releases
- 13 contributors

**brockho committed on GitHub** Merge pull request #1075 from numbbo/development

- `code-experiments` Merge pull request #1071 from ttusar/debug
- `code-postprocessing` further clean up of postprocessing output.
- `code-preprocessing/archive-update` Added empty last lines.
- `docs` updated reference to biobjective perf-assessment paper on arXiv in ge...
- `howtos` Update documentation howto.md
- `.clang-format` raising an error in bobb09_logger.c when best_value is NULL. Plus s...
- `.hgignore` raising an error in bobb09_logger.c when best_value is NULL. Plus s...
- `AUTHORS` small correction in AUTHORS
- `LICENSE` Added acknowledgements to external collaborators

Latest commit 0:cb7db on 10 Jun

<table>
<thead>
<tr>
<th>Branch</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **7,902 commits**
- **12 branches**
- **25 releases**
- **13 contributors**

<table>
<thead>
<tr>
<th>Commit</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>brockho</td>
<td>Merge pull request #1075 from numbbo/development</td>
<td></td>
</tr>
<tr>
<td>code-experiments</td>
<td></td>
<td>2 months ago</td>
</tr>
<tr>
<td>code-postprocessing</td>
<td>further clean up of postprocessing output,</td>
<td>2 months ago</td>
</tr>
<tr>
<td>code-preprocessing/archive-update</td>
<td>Added empty last lines.</td>
<td>2 months ago</td>
</tr>
<tr>
<td>docs</td>
<td>updated reference to biobjective perf-assessment paper on arXiv in gen...</td>
<td>3 months ago</td>
</tr>
<tr>
<td>howtos</td>
<td>Update documentation howto.md</td>
<td>5 months ago</td>
</tr>
<tr>
<td>.clang-format</td>
<td>raising an error in bbo2009_logger.c when best_value is NULL. Plus s...</td>
<td>a year ago</td>
</tr>
<tr>
<td>.hgignore</td>
<td>raising an error in bbo2009_logger.c when best_value is NULL. Plus s...</td>
<td>a year ago</td>
</tr>
<tr>
<td>AUTHORS</td>
<td>small correction in AUTHORS</td>
<td>4 months ago</td>
</tr>
<tr>
<td>LICENSE</td>
<td>Added acknowledgements to external collaborators...</td>
<td>5 months ago</td>
</tr>
<tr>
<td>README.md</td>
<td>Update README.md</td>
<td>2 months ago</td>
</tr>
<tr>
<td>do.py</td>
<td>Merge branch 'development' of <a href="https://github.com/numbbo/coco">https://github.com/numbbo/coco</a> into pp...</td>
<td>3 months ago</td>
</tr>
<tr>
<td>doxygen.ini</td>
<td>moved all files into code-experiments/ folder besides the do.py script...</td>
<td>9 months ago</td>
</tr>
</tbody>
</table>
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 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,
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 bbo-b1obj COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbed
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 bbo-blob COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbed.
- 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.
Getting Started

1. Check out the Requirements above.

2. Download the COCO framework code from github either by clicking the Download ZIP button or (preferred) by typing git clone https://github.com/numbbo/coco. After cloning, git pull keeps the code up-to-date with the latest release.

CAVEAT: this code is still under heavy development. The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

3. 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
don.py run-c
don.py run-java
don.py run-matlab
don.py run-octave
don.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.

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

```python
don.py install-postprocessing
```
3. 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
```

Depending on which language shall be used to run the experiments, `run-<language>` 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.

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

```
python do.py install-postprocessing
```

To (user-locally) install the post-processing. From here on the builds are not necessary and can be skipped for new releases.

5. 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, in case). As the details vary, see the respective read-me’s and/or example experiment files:

- Read me and example experiment
- Java read me and example experiment
- Matlab/Octave read me and example experiment
to (user-locally) install the post-processing. From here on, do.py has done its job and is only needed again for updating the builds to a new release.

5. 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, in case). As the details vary, see the respective read-me’s and/or:

- 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.

6. Now you can run your favorite algorithm on the bbo-bbobj (for multi-objective algorithms) or on the bbob suite (for single-objective algorithms). Output is automatically generated in the specified data result_folder.

7. Postprocess the data from the results folder by typing

   python -m bbo_pproc [-o OUTPUT_FOLDERNAME] YOURDATAFOLDER [MORE_DATAFOLDERS]

The name bbo_pproc will become cocopp in future. 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 folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.
/* Iterate over all problems in the suite */
while ((PROBLEM = coco_suite_get_next_problem(suite, observer)) != NULL) {

    size_t dimension = coco_problem_get_dimension(PROBLEM);

    /* Run the algorithm at least once */
    for (run = 1; run <= 1 + INDEPENDENT_RESTARTS; run++) {

        size_t evaluations_done = coco_problem_get_evaluations(PROBLEM);
        long evaluations_remaining =
            (long)(dimension * BUDGET_MULTIPLIER) - (long)evaluations_done;

        if (... || (evaluations_remaining <= 0))
            break;

        my_random_search(evaluate_function, dimension,
                         coco_problem_get_number_of_objectives(PROBLEM),
                         coco_problem_get_smallest_values_of_interest(PROBLEM),
                         coco_problem_get_largest_values_of_interest(PROBLEM),
                         (size_t) evaluations_remaining,
                         random_generator);
}
6. Now you can run your favorite algorithm on the bboB single-objective algorithms. Output is automatically saved in the results folder.

7. Postprocess the data from the results folder by typing:

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

The name `bbob_pproc` will become `cocode` in future. 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` folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.

A folder, `ppdata` by default, will be generated, which contains all output from the post-processing, including a `ppdata.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.

For the single-objective `bbob` suite, a summary `pdf` can be produced via `LaTeX`. The corresponding templates in `ACM` format can be found in the `code-postprocessing/latex-templates` folder. `LaTeX` templates for the multi-objective `bbob-biobj` suite will follow in a later release. A basic `html` output is also available in the result folder of the postprocessing (file `templateBBOBarticle.html`).

8. 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.

If you detect bugs or other issues, please let us know by opening an issue in our issue tracker at https://github.com/numbbo/coco/issues.

**Description by Folder**
6. Now you can run your favorite algorithm on the bbo-bobj (for multi-objective algorithms) or on the bbo suite (for single-objective algorithms). Output is automatically generated in the specified data result_folder.

7. Postprocess the data from the results folder by typing

```
python -m bbo_postproc [-o OUTPUT_FOLDERNAME] YOURDATA_FOLDER [MORE_DATAFOLDERS]
```

The name bbo_postproc will become coco in future. Any subfolder in the folder arg data. That is, experiments from different batches can be in different folders collected in YOURDATA_FOLDER 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 a pdata.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.

For the single-objective bbo suite, a summary pdf can be produced via LaTeX. The corresponding templates in ACM form.

8. On success, return to step 5.

If you discover any bugs:

https://github.com/numbbo/coco

Description by Folder

tip: start with small #funevals (until bugs fixed 😊) then increase budget to get a feeling how long a "long run" will take
result folder
Post processing results

Single algorithm data

RS_on_bbob-biobj-3e4funevals
Automatically generated results

Average number of \(f\) evaluations to reach target

1 Sphere/Sphere

2 Sphere/sep. Ellipsoid

3 Sphere/Attractive sector

4 Sphere/Rosenbrock

5 Sphere/Sharp ridge

6 Sphere/Different Powers

7 Sphere/Restric"
Automatically generated results

Runtime distributions (ECDFs) per function

1. Sphere/Sphere
2. Sphere/sep. Ellipsoid
3. Sphere/Attractive sector
4. Sphere/Rosenbrock
5. Sphere/Sharp ridge
6. Sphere/Different Powers
7. Sphere/Rastrigin
8. Sphere/Schaffer F7
9. Sphere/Schwefel
so far:

data for about 165 algorithm variants
[in total on single- and multiobjective problems]
118 workshop papers
by 79 authors from 25 countries
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?

• a number of testbeds are around

• account for invariance properties

prediction of performance is based on “similarity”,
ideally equivalence classes of functions
<table>
<thead>
<tr>
<th>Test Suite</th>
<th>No. of Noiseless Functions</th>
<th>No. of Noisy Functions</th>
<th>No. of Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>bboob</td>
<td>24</td>
<td>30</td>
<td>140+</td>
</tr>
<tr>
<td>bboob-noisy</td>
<td>24</td>
<td>30</td>
<td>40+</td>
</tr>
<tr>
<td>bboob-biobj</td>
<td>55</td>
<td>55</td>
<td>15</td>
</tr>
</tbody>
</table>
How Do We Measure Performance?

Meaningful quantitative measure

- quantitative on the ratio scale (highest possible)
  "algo A is two times better than algo B" is 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
runtime or first hitting time is the prime candidate (we don't have many choices anyway)
How Do We Measure Performance?

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
Measuring Performance Empirically

convergence graphs is all we have to start with...
ECDF:
Empirical Cumulative Distribution Function of the Runtime
[aka data profile]
A Convergence Graph

function value

$\log_{10}(\text{function evaluations})$
First Hitting Time is Monotonous
15 Runs
15 Runs ≤ 15 Runtime Data Points
Empirical Cumulative Distribution

- The ECDF of run lengths to reach the 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

interpretations possible:

- 80% of the runs reached the target
- e.g. 60% of the runs need between 2000 and 4000 evaluations
Reconstructing A Single Run

function value vs. \log_{10}(\text{function evaluations})
Reconstructing A Single Run

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

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

15 runs
Aggregation

15 runs
50 targets
Aggregation

15 runs
50 targets
Aggregation

15 runs
50 targets
ECDF with 750 steps
Aggregation

50 targets from 15 runs
...integrated in a single graph
Interpretation

50 targets from 15 runs integrated in a single graph

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

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

\[ p_s(\text{Algo B}) \approx 1, \text{ slow convergence} \]
Fixed-target: Measuring Runtime

• Algo Restart A:

• Algo Restart B:

\[ p_s(\text{Algo Restart A}) = 1 \]

\[ p_s(\text{Algo Restart B}) = 1 \]
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):

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

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

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

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

What we typically plot are ECDFs of the simulated restarted algorithms:
Worth to Note: ECDFs in COCO

In COCO, ECDF graphs
- never aggregate over dimension
  - but often over targets and functions
- can show data of more than 1 algorithm at a time

150 algorithms from BBOB-2009 till BBOB-2015
Another Interesting Plot...

...comparing aRT values over several algorithms
Another Interesting Plot...

...comparing aRT values over several algorithms

- aRT value [if $< \infty$] to reach given target precision
- A star indicates statistically significant results compared to all other displayed algs
- Median runlength of unsuccessful runs
Another Interesting Plot...

...comparing aRT values over several algorithms

artificial best algorithm from BBOB-2016
Interesting for 2 Algorithms...

...are scatter plots
There are more Plots...

...but they are probably less interesting for us here
The single-objective BBOB functions
bboB Testbed

- 24 functions in 5 groups:

<table>
<thead>
<tr>
<th>1 Separable Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1 Sphere Function</td>
</tr>
<tr>
<td>f2 Ellipsoidal Function</td>
</tr>
<tr>
<td>f3 Rastrigin Function</td>
</tr>
<tr>
<td>f4 Büche-Rastrigin Function</td>
</tr>
<tr>
<td>f5 Linear Slope</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Functions with low or moderate conditioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>f6 Attractive Sector Function</td>
</tr>
<tr>
<td>f7 Step Ellipsoidal Function</td>
</tr>
<tr>
<td>f8 Rosenbrock Function, original</td>
</tr>
<tr>
<td>f9 Rosenbrock Function, rotated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Functions with high conditioning and unimodal</th>
</tr>
</thead>
<tbody>
<tr>
<td>f10 Ellipsoidal Function</td>
</tr>
<tr>
<td>f11 Discus Function</td>
</tr>
<tr>
<td>f12 Bent Cigar Function</td>
</tr>
<tr>
<td>f13 Sharp Ridge Function</td>
</tr>
<tr>
<td>f14 Different Powers Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Multi-modal functions with adequate global structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>f15 Rastrigin Function</td>
</tr>
<tr>
<td>f16 Weierstrass Function</td>
</tr>
<tr>
<td>f17 Schaffers F7 Function</td>
</tr>
<tr>
<td>f18 Schaffers F7 Functions, moderately ill-conditioned</td>
</tr>
<tr>
<td>f19 Composite Griewank-Rosenbrock Function F8F2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5 Multi-modal functions with weak global structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>f20 Schwefel Function</td>
</tr>
<tr>
<td>f21 Gallagher’s Gaussian 101-me Peaks Function</td>
</tr>
<tr>
<td>f22 Gallagher’s Gaussian 21-hi Peaks Function</td>
</tr>
<tr>
<td>f23 Katsuura Function</td>
</tr>
<tr>
<td>f24 Lunacek bi-Rastrigin Function</td>
</tr>
</tbody>
</table>

- 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
• Prescribed instances typically change from year to year
  • 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.

\[ f_{10} \text{ (Ellipsoid)} \]

\[ f_{15} \text{ (Rastrigin)} \]

linear transformations
the recent extension to multi-objective optimization
**Multiobjective Optimization (MOO)**

Multiple objectives that have to be optimized simultaneously.

---

**Diagram:**

- **Axes:**
  - **X-axis:** Cost
  - **Y-axis:** Performance

- **Legend:**
  - **Better**
  - **Worse**
  - **Incomparable**

---

© Anne Auger and Dimo Brockhoff, Inria
Observations: ① there is no single optimal solution, but
② some solutions (●) are better than others (○)
A Brief Introduction to Multiobjective Optimization

\[ u \text{ weakly Pareto dominates } v \ (u \leq_{\text{par}} v): \quad \forall 1 \leq i \leq k : f_i(u) \leq f_i(v) \]

\[ u \text{ Pareto dominates } v \ (u <_{\text{par}} v): \quad u \leq_{\text{par}} v \land v \nsubseteq_{\text{par}} u \]
u weakly Pareto dominates v ($u \leq_{\text{par}} v$): $\forall 1 \leq i \leq k : f_i(u) \leq f_i(v)$

u Pareto dominates v ($u <_{\text{par}} v$): $u \leq_{\text{par}} v \land v \not\leq_{\text{par}} u$
Pareto set: set of all non-dominated solutions (decision space)
Pareto front: its image in the objective space
A Brief Introduction to Multiobjective Optimization

**Pareto set:** set of all non-dominated solutions (decision space)

**Pareto front:** its image in the objective space

---

**true Pareto front**
(Pareto efficient frontier)
A Brief Introduction to Multiobjective Optimization

Decision space

Objective space

Solution of Pareto-optimal set

Non-optimal decision vector

Vector of Pareto-optimal front

Non-optimal objective vector
ideal point: best values
nadir point: worst values

\{ \text{obtained for } \textit{Pareto-optimal} \text{ points} \}
Idea:
- transfer multiobjective problem into a set problem
- define an objective function ("quality indicator") on sets

Important:
⇒ Underlying dominance relation (on sets) should be reflected by the resulting set comparisons!

\[ A \preceq B :\iff \forall y \in B \exists x \in A x \preceq \text{par } y \]
Examples of Quality Indicators

I(\text{A}) = \text{volume of the weakly dominated area in objective space}

I(\text{A}, \text{B}) = \text{how much needs A to be moved to weakly dominate B}

ref \ A \preceq \ B : \iff I(\text{A}) \geq I(\text{B})

ref \ A \preceq \ B : \iff I(\text{A}, \text{B}) \leq I(\text{B}, \text{A})
Examples of Quality Indicators II

$A \preceq B : \iff I(A, R) \leq I(B, R)$

$I(A, R) = \text{how much needs } A \text{ to be moved to weakly dominate } R$

$A \preceq B : \iff I(A) \leq I(B)$

$I(A) = \frac{1}{|\Lambda|} \sum_{\lambda \in \Lambda} \min_{a \in A} \left( \max_{j=1..m} \lambda_j |z_j^* - a_j| \right)$

unary epsilon indicator

unary R2 indicator
Examples of Quality Indicators II

ref

\[ A \preceq B :\iff I(A,R) \leq I(B,R) \]

\[ A \preceq B :\iff I(A) \leq I(B) \]

\[ I(A,R) = \text{how much needs } A \text{ to be moved to weakly dominate } R \]

\[ I(A) = \frac{1}{|\Lambda|} \sum_{\lambda \in \Lambda} \min_{a \in A} \left( \max_{j=1..m} \lambda_j |z_j^* - a_j| \right) \]

unary epsilon indicator

unary R2 indicator

© Anne Auger and Dimo Brockhoff, Inria

### bbob-biobj Testbed

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

#### 1 Separable Functions
- f1: Sphere Function ✓
- f2: Ellipsoidal Function ✓
- f3: Rastrigin Function
- f4: Büche-Rastrigin Function
- f5: Linear Slope

#### 2 Functions with low or moderate conditioning
- f6: Attractive Sector Function ✓
- f7: Step Ellipsoidal Function
- f8: Rosenbrock Function, original ✓
- f9: Rosenbrock Function, rotated

#### 3 Functions with high conditioning and unimodal
- f10: Ellipsoidal Function
- f11: Discus Function
- f12: Bent Cigar Function
- f13: Sharp Ridge Function ✓
- f14: Different Powers Function ✓

#### 4 Multi-modal functions with adequate global structure
- f15: Rastrigin Function ✓
- f16: Weierstrass Function
- f17: Schaffers F7 Function ✓
- f18: Schaffers F7 Functions, moderately ill-conditioned
- f19: Composite Griewank-Rosenbrock Function F8F2

#### 5 Multi-modal functions with weak global structure
- f20: Schwefel Function ✓
- f21: Gallagher’s Gaussian 101-me Peaks Function ✓
- f22: Gallagher’s Gaussian 21-hi Peaks Function
- f23: Katsuura Function
- f24: Lunacek bi-Rastrigin Function
**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></td>
</tr>
<tr>
<td>f5 Linear Slope</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Functions with low or moderate conditioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>f6 Attractive Sector Function ✓</td>
</tr>
<tr>
<td>f7 Step Ellipsoidal Function</td>
</tr>
<tr>
<td>f8 Rosenbrock Function, original ✓</td>
</tr>
<tr>
<td>f9 Rosenbrock Function, rotated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Functions with high conditioning and unimo</th>
</tr>
</thead>
<tbody>
<tr>
<td>f10 Ellipsoidal Function</td>
</tr>
<tr>
<td>f11 Discus Function</td>
</tr>
<tr>
<td>f12 Bent Cigar Function</td>
</tr>
<tr>
<td>f13 Sharp Ridge Function ✓</td>
</tr>
<tr>
<td>f14 Different Powers Function ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
<th>f8</th>
<th>f9</th>
<th>f10</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td>f11</td>
<td>f12</td>
<td>f13</td>
<td>f14</td>
<td>f15</td>
<td>f16</td>
<td>f17</td>
<td>f18</td>
<td>f19</td>
</tr>
<tr>
<td>f6</td>
<td>f20</td>
<td>f21</td>
<td>f22</td>
<td>f23</td>
<td>f24</td>
<td>f25</td>
<td>f26</td>
<td>f27</td>
<td></td>
</tr>
<tr>
<td>f8</td>
<td>f28</td>
<td>f29</td>
<td>f30</td>
<td>f31</td>
<td>f32</td>
<td>f33</td>
<td>f34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f13</td>
<td>f35</td>
<td>f36</td>
<td>f37</td>
<td>f38</td>
<td>f39</td>
<td>f40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f42</td>
<td>f43</td>
<td>f44</td>
<td>f45</td>
<td></td>
</tr>
<tr>
<td>f17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f46</td>
<td>f47</td>
<td>f48</td>
<td>f49</td>
</tr>
<tr>
<td>f20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f50</td>
<td>f51</td>
<td>f52</td>
</tr>
<tr>
<td>f21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f53</td>
<td>f54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f55</td>
</tr>
</tbody>
</table>
bbob-biobj Testbed

- 55 functions by combining 2 bbob functions
- 15 function groups with 3-4 functions each
  - separable – separable, separable – moderate, separable - ill-conditioned, ...
- 6 dimensions: 2, 3, 5, 10, 20, (40 optional)
- instances derived from bbob instances:
- no normalization (algo has to cope with different orders of magnitude)
- for performance assessment: ideal/nadir points known
bbob-biobj Testbed (cont'd)

• Pareto set and Pareto front unknown
  • but we have a good idea of where they are by running quite some algorithms and keeping track of all non-dominated points found so far

• Various types of shapes
bbob-biobj Testbed (cont'd)

projection of decision space for bbob-biobj $f_{10}$ (5-D, instance 1)
- reference set (1230 of 1722826 points)
- cuts through single optima
- cut through both optima
- two random directions

projection of decision space for bbob-biobj $f_{10}$ (5-D, instance 1)
- reference set (1095 of 1378108 points)
- cuts through single optima
- cut through both optima
- two random directions

projection of decision space for bbob-biobj $f_{10}$ (5-D, instance 1)
- reference set (585 of 444135 points)
- cuts through single optima
- cut through both optima
- two random directions

search space
- connected
- uni-modal

objective space
- disconnected
- multi-modal

bbob-biobj $f_{10}$ along linear search space directions (5-D, instance 1)
- cuts through single optima
- cut through both optima
- two random directions

bbob-biobj $f_{10}$ along linear search space directions (5-D, instance 1)
- cuts through single optima
- cut through both optima
- two random directions

bbob-biobj $f_{10}$ along linear search space directions (5-D, instance 1)
- cuts through single optima
- cut through both optima
- two random directions
Bi-objective Performance Assessment

algorithm quality =

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

if a point dominates nadir

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

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)

• actual absolute hypervolume targets used are

  \[ \text{HV(refset)} - \text{target precision} \]

  with 58 fixed target precisions between 1 and \(-10^{-4}\) (same for all functions, dimensions, and instances) in the displays
## Course Overview

<table>
<thead>
<tr>
<th>No.</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fri, 16.9.2016</td>
<td>Introduction to Optimization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Fri, 21.9.2016</td>
<td>groups defined via wiki</td>
</tr>
<tr>
<td>3</td>
<td>Thu, 22.9.2016</td>
<td>everybody went (actively!) through the Getting Started part of github.com/numbbo/coco</td>
</tr>
<tr>
<td></td>
<td>Fri, 23.9.2016</td>
<td>Today's lecture: ❶ Benchmarking; ❷ final adjustments of groups</td>
</tr>
<tr>
<td></td>
<td></td>
<td>everybody can run and postprocess the example experiment (~1h for final ❸ questions/help during the lecture)</td>
</tr>
<tr>
<td></td>
<td>Fri, 30.9.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>4</td>
<td>Fri, 7.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>5</td>
<td>Mon, 10.10.2016</td>
<td>deadline for intermediate wiki report: what has been done and what remains to be done?</td>
</tr>
<tr>
<td>6</td>
<td>Fri, 14.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td>7</td>
<td>Tue, 18.10.2016</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>Tue, 18.10.2016</td>
<td>deadline for submitting data sets</td>
</tr>
<tr>
<td></td>
<td>Fri, 21.10.2016</td>
<td>deadline for paper submission</td>
</tr>
<tr>
<td>8</td>
<td>Fri, 4.11.2016</td>
<td>vacation</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Final lecture</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>oral presentations (individual time slots)</td>
</tr>
<tr>
<td></td>
<td>7.-11.11.2016</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Exam (exact date to be confirmed)</td>
</tr>
<tr>
<td>12</td>
<td>14 - 18.11.2016</td>
<td></td>
</tr>
</tbody>
</table>

**All deadlines: 23:59pm Paris time**
I hope it became clear...

...what are the **important issues** in algorithm benchmarking
...which **functionality** is behind the COCO platform
...and **how to measure performance** in particular
...what are the basics of **multiobjective optimization**
...and what are the next important steps to do:

  read the assigned paper and implement the algorithm
  document everything on the wiki

**Monday in 2 weeks:** intermediate report on wiki
And now...

...time for your questions and problems around COCO