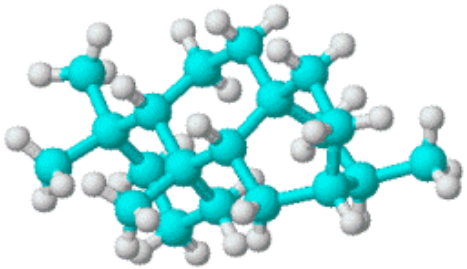
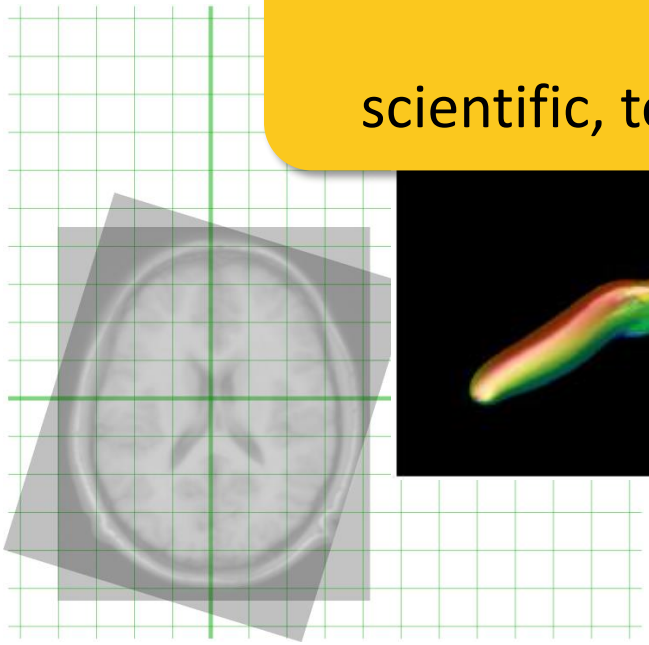
COCO: A Platform for Comparing Continuous Optimizers Effortlessly

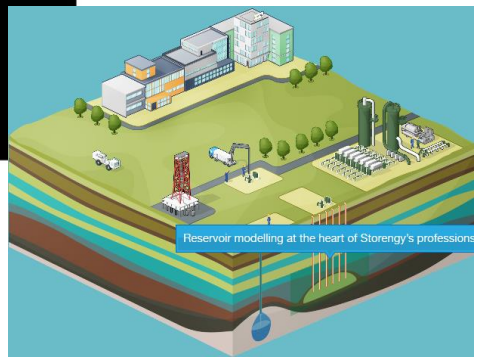
Dimo Brockhoff
dimo.brockhoff@inria.fr

optimization problems are ubiquitous



challenging optimization problems
appear in many
scientific, technological and industrial domains





Numerical Blackbox Optimization

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

derivatives not available or not useful

Practical Blackbox Optimization

Given:

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 which difficulties their problem has (e.g. multi-modality, non-separability, ...)

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):

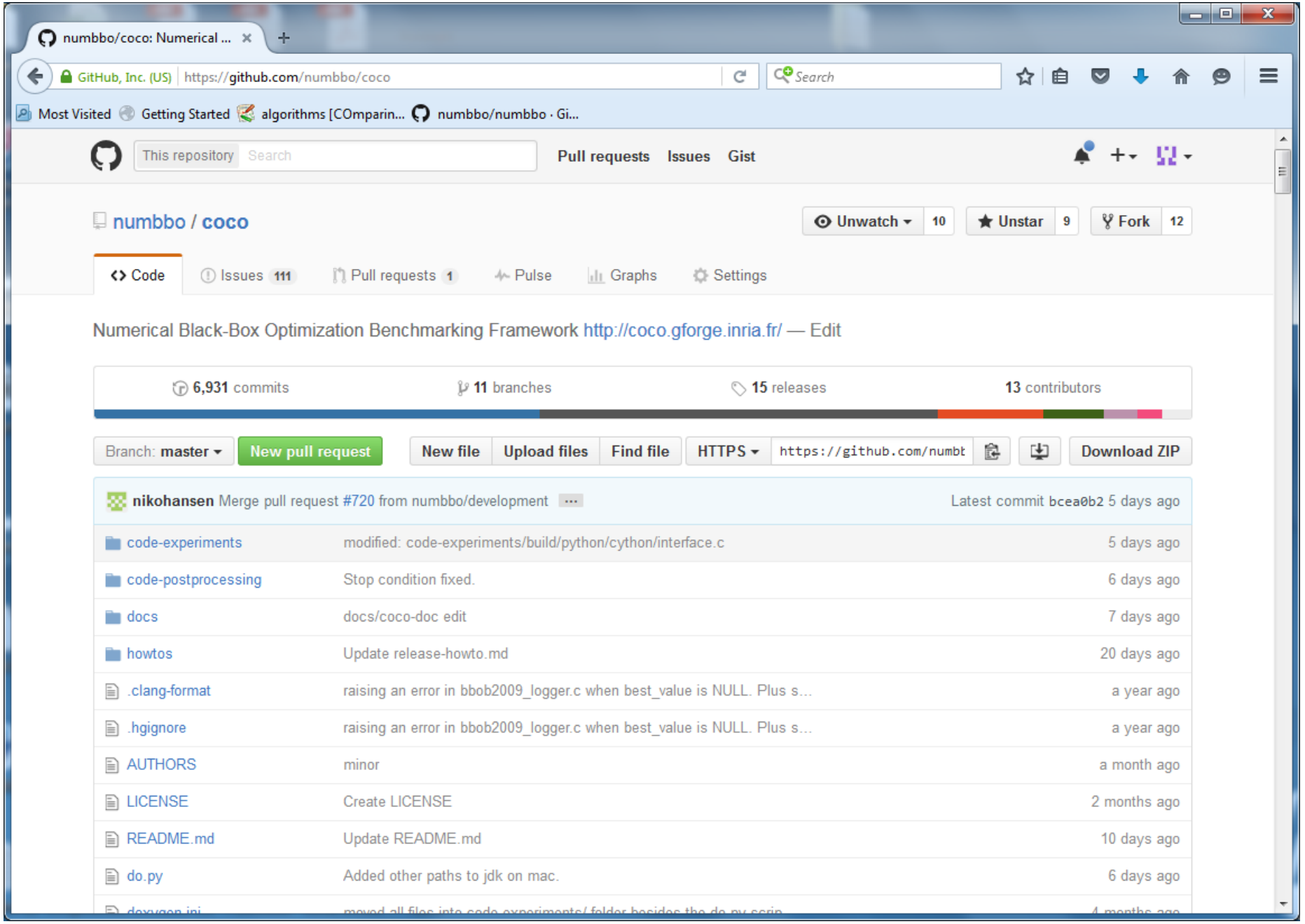
- choosing (and implementing) problems, performance measures, visualization, stat. tests, ...
- running a set of algorithms

that's where COCO comes into play

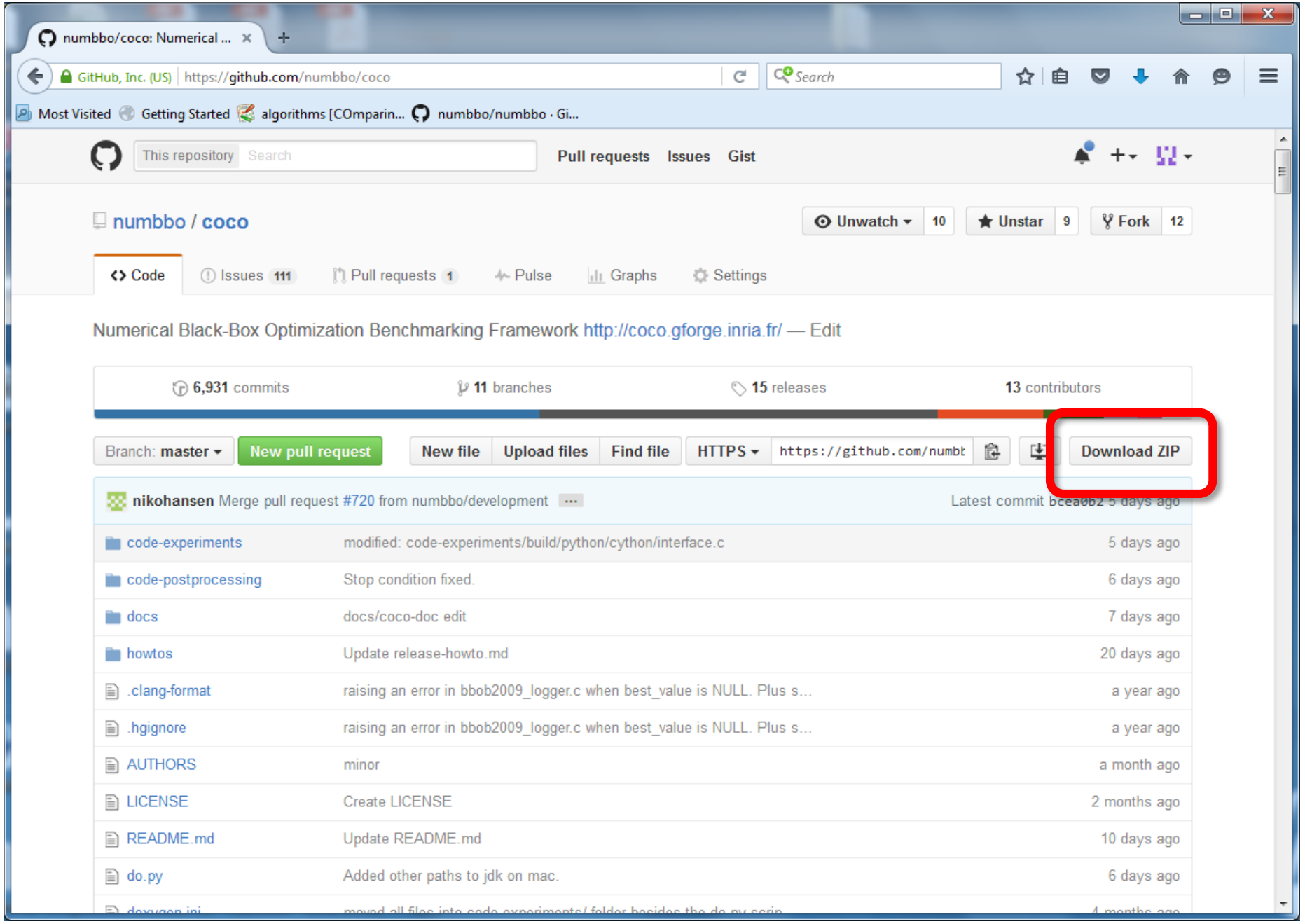
automatized benchmarking

How to benchmark algorithms with COCO?

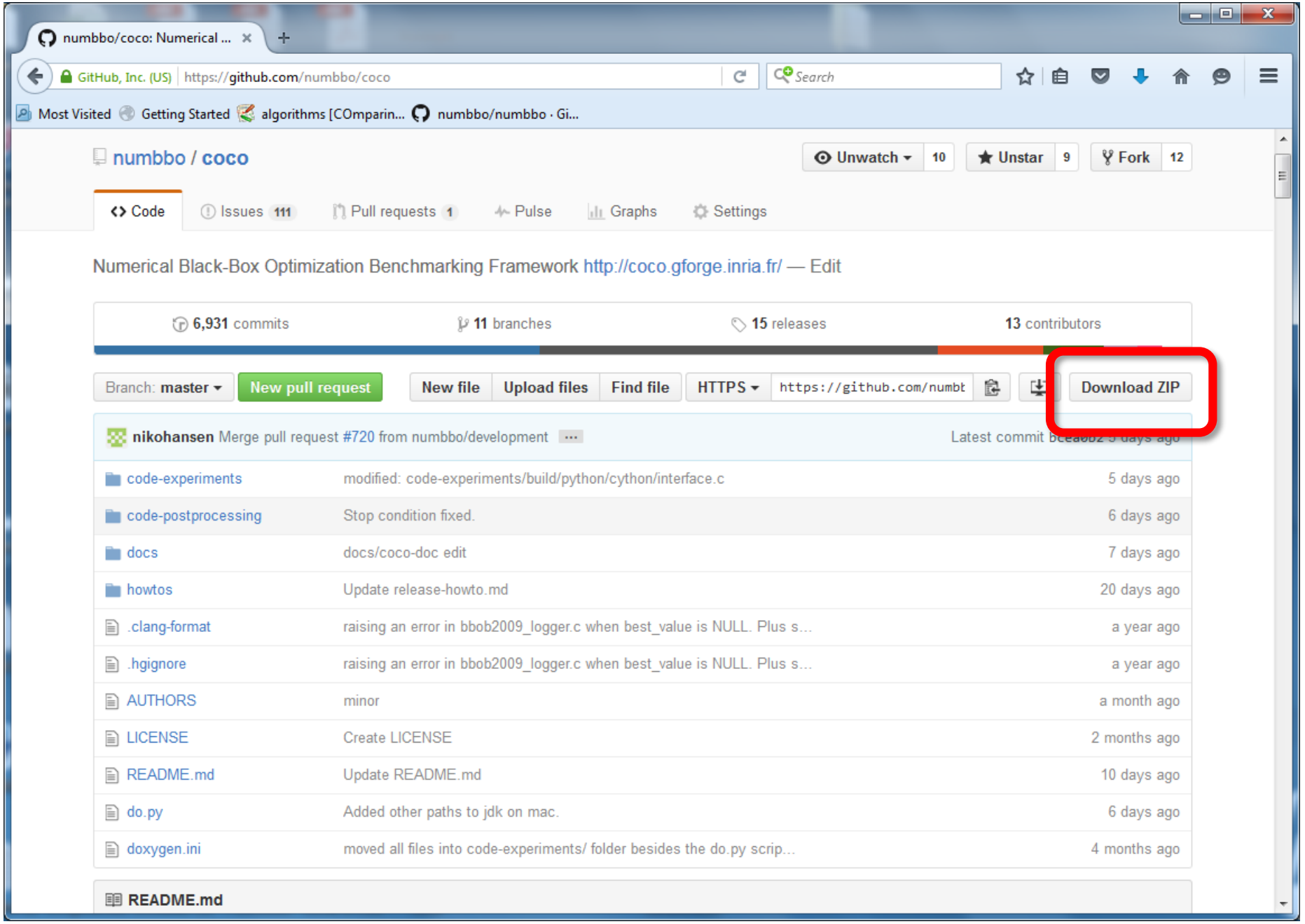
https://github.com/numbbo/coco



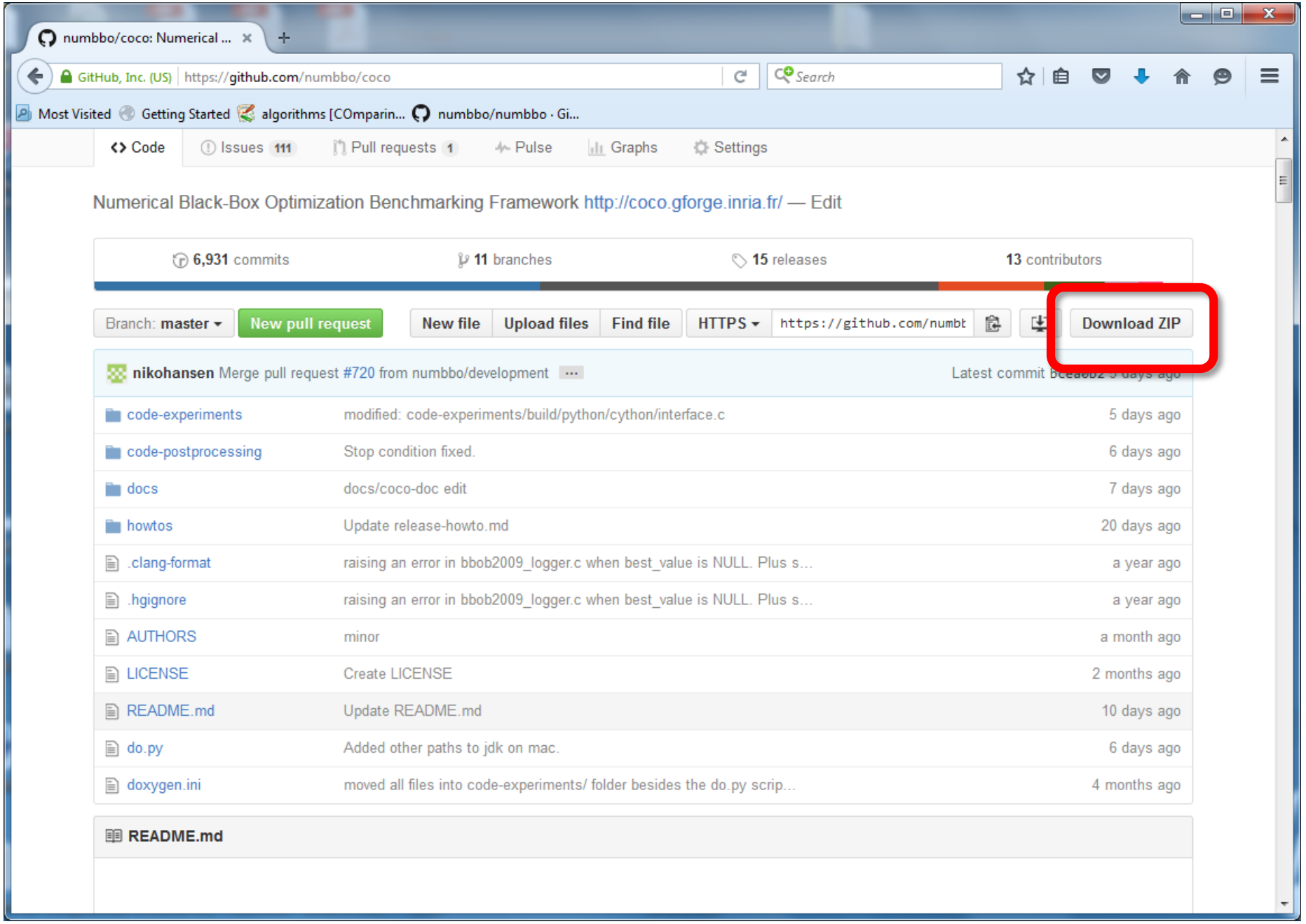
https://github.com/numbbo/coco



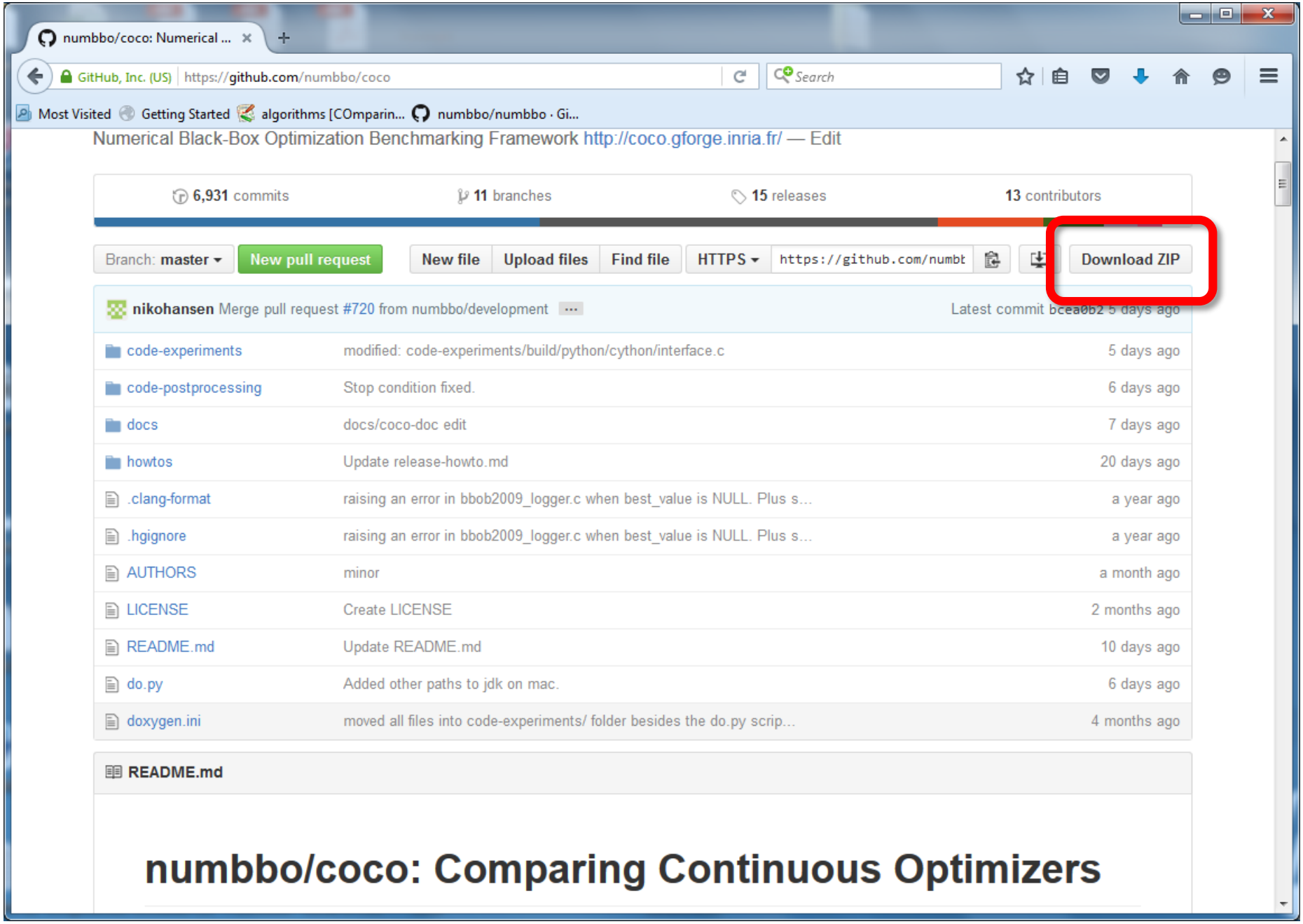
https://github.com/numbbo/coco



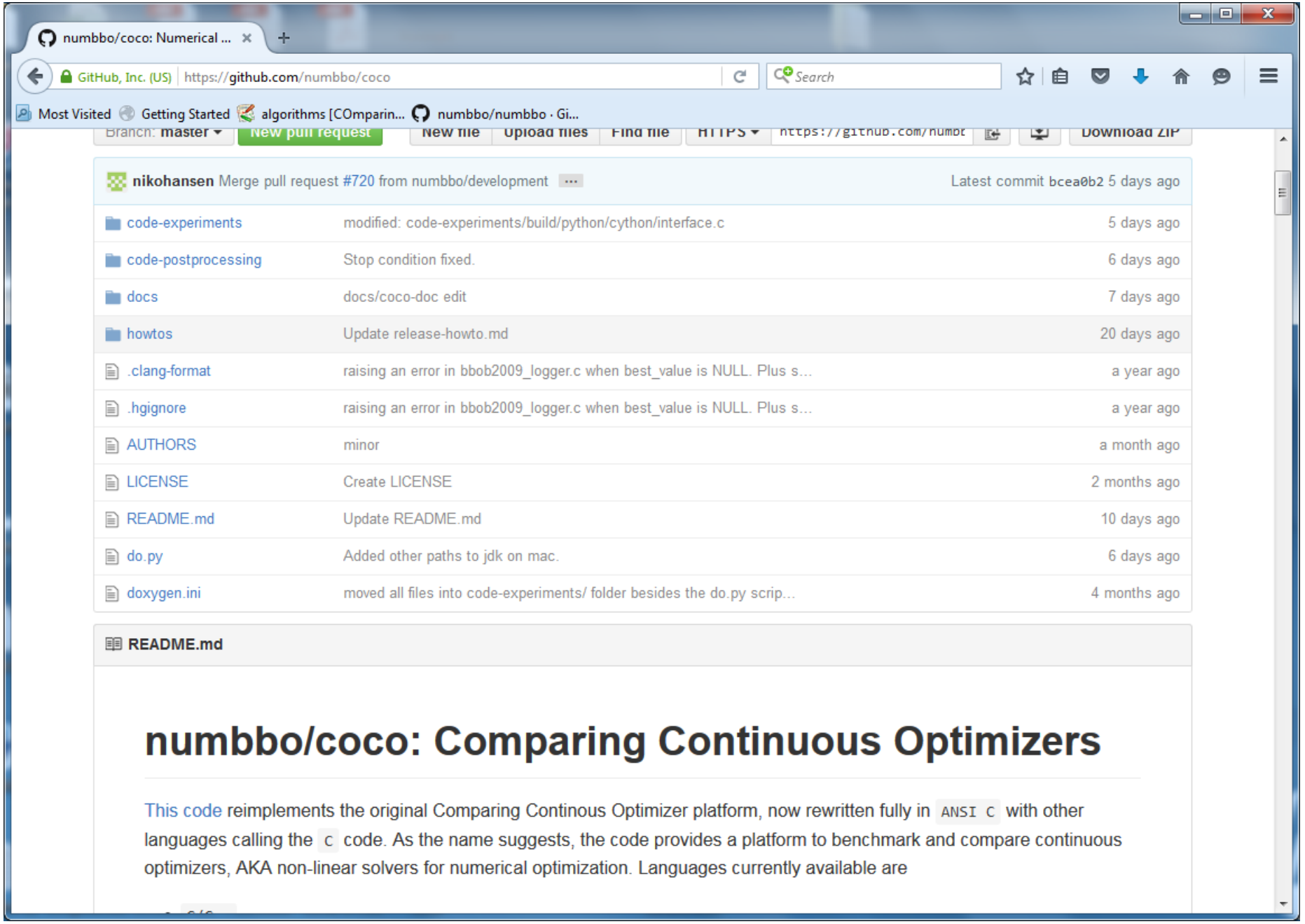
https://github.com/numbbo/coco



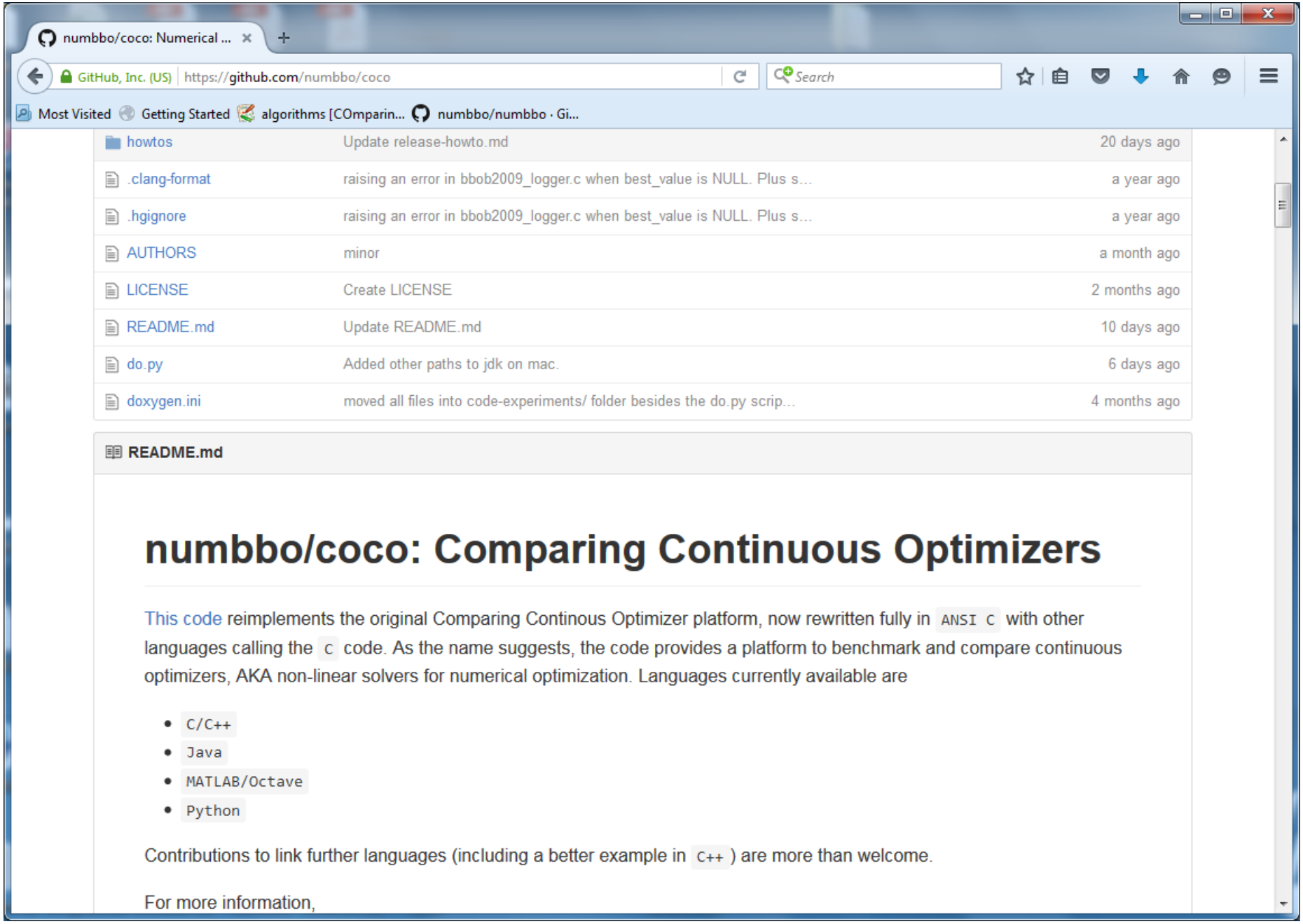
https://github.com/numbbo/coco



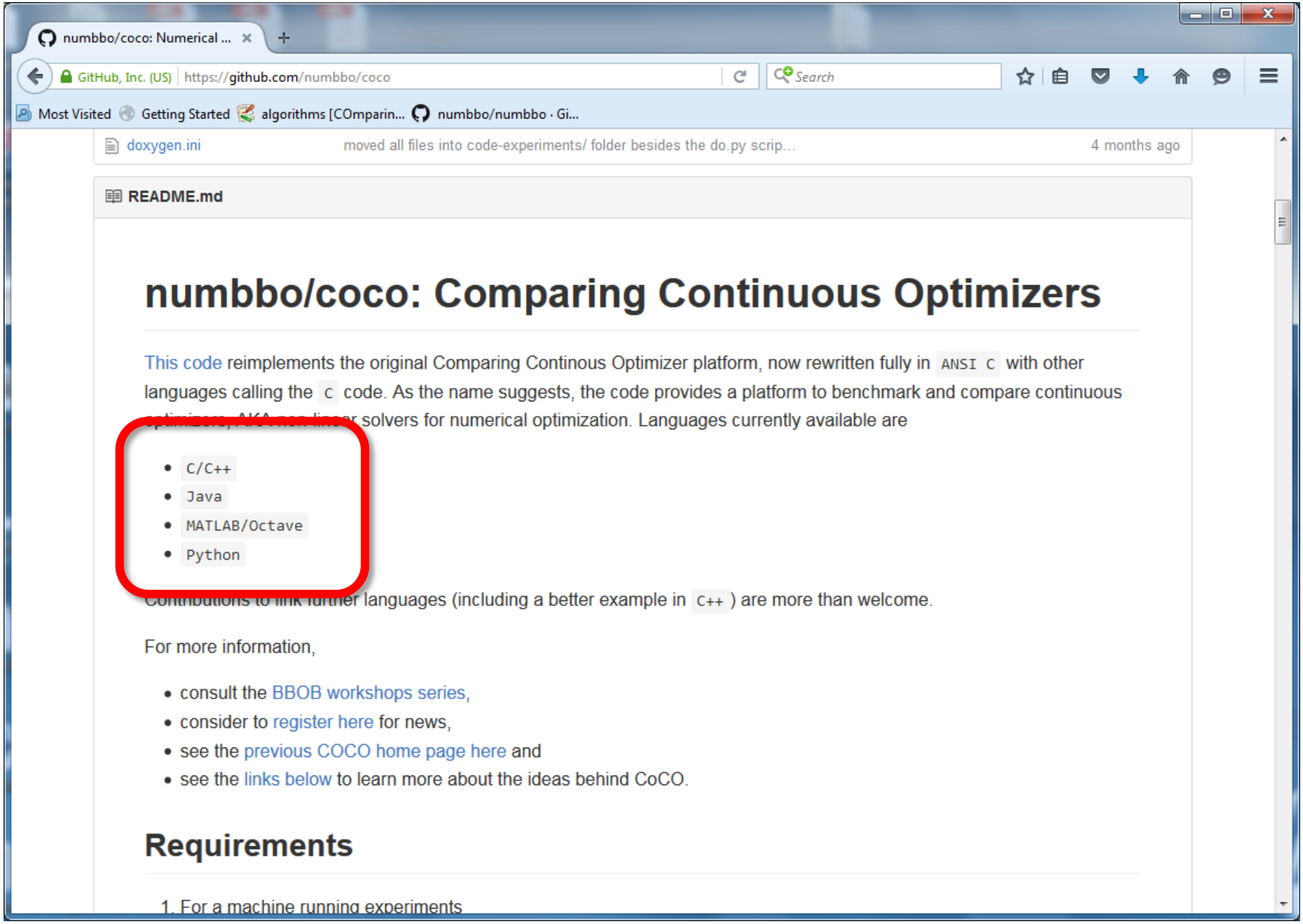
https://github.com/numbbo/coco



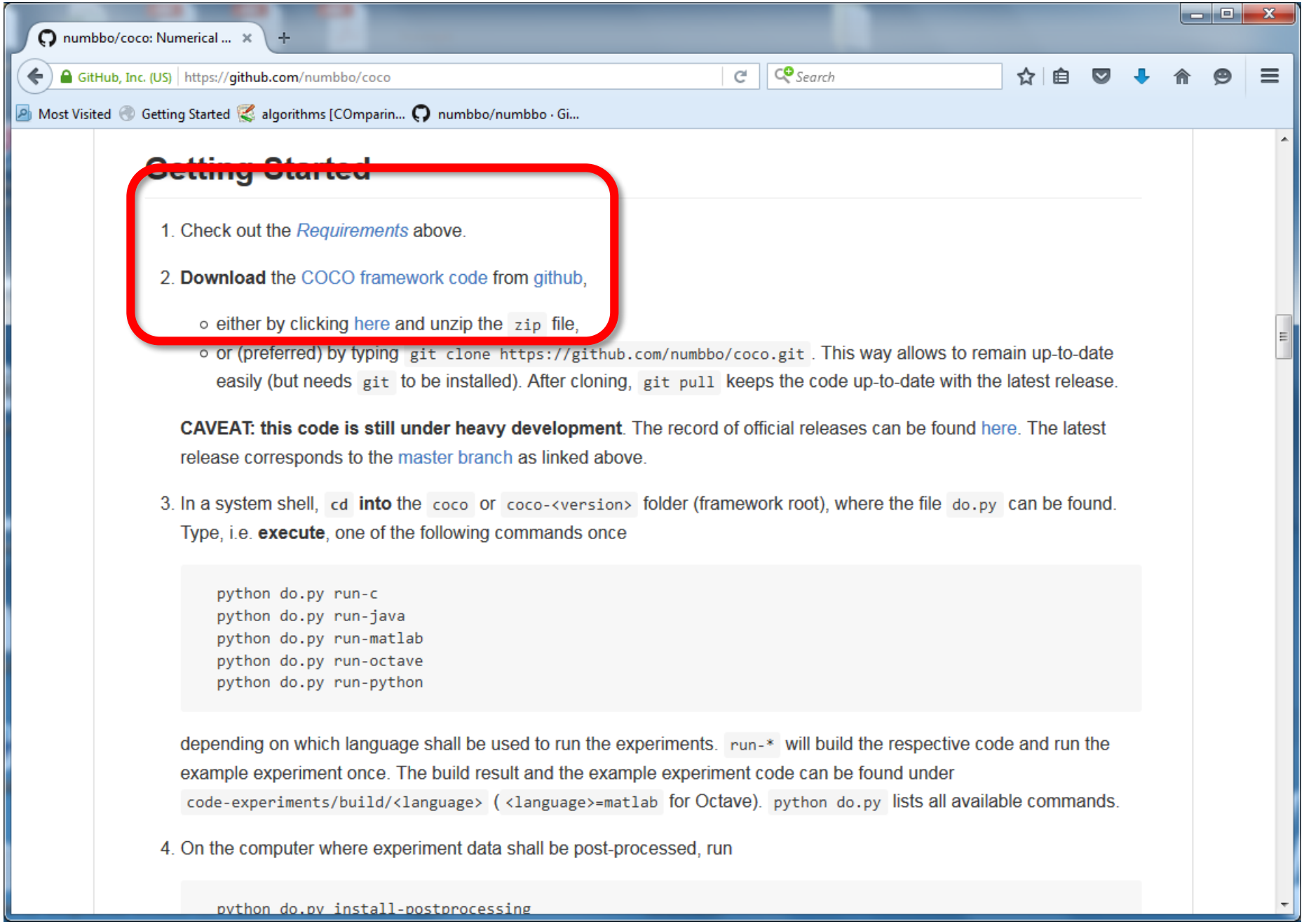
https://github.com/numbbo/coco



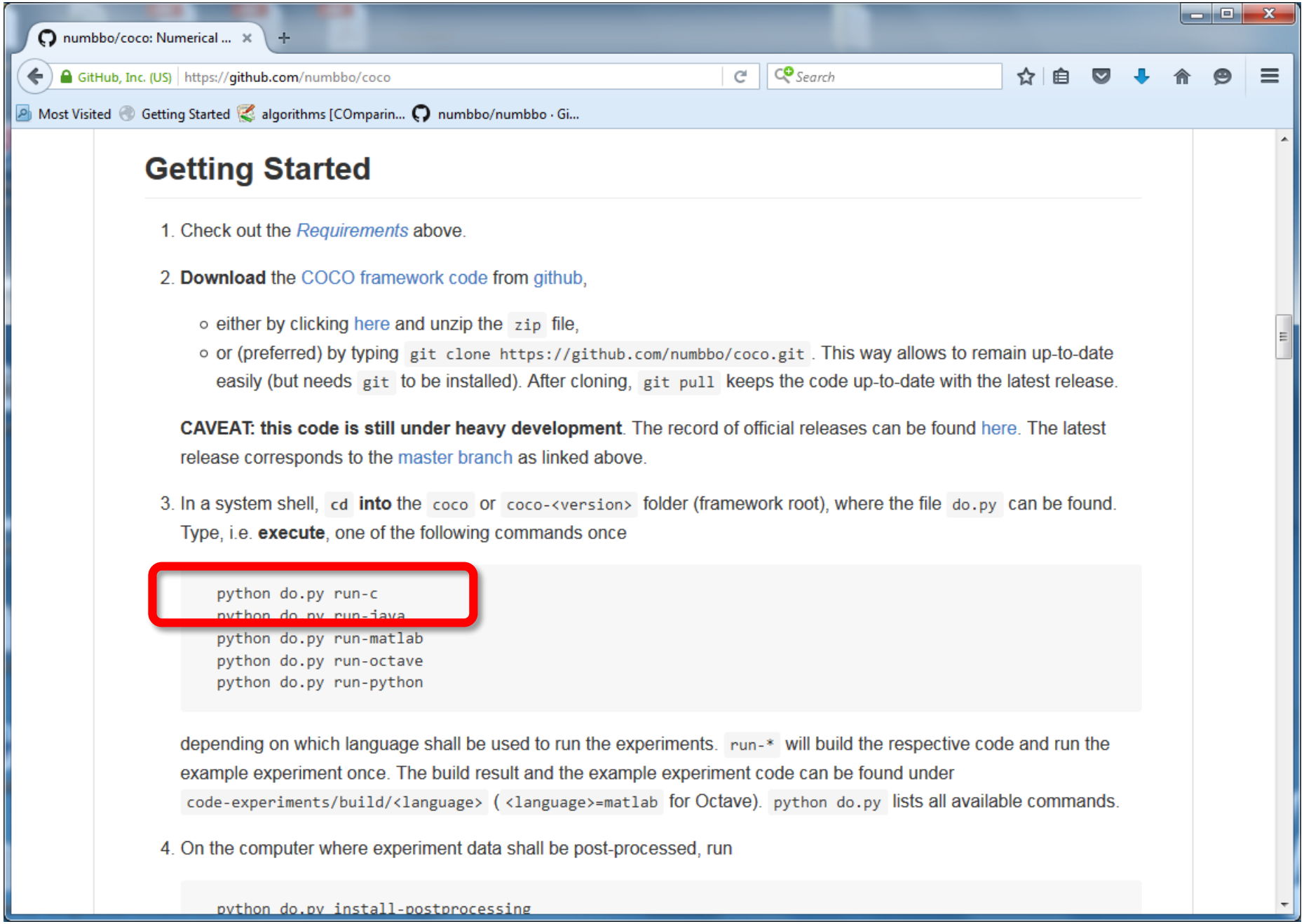
https://github.com/numbbo/coco



https://github.com/numbbo/coco



https://github.com/numbbo/coco



Getting Started

1. Check out the *Requirements* above.
2. **Download** the COCO framework code from github,
   - either by clicking here and unzip the `zip` file,
   - or (preferred) 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.

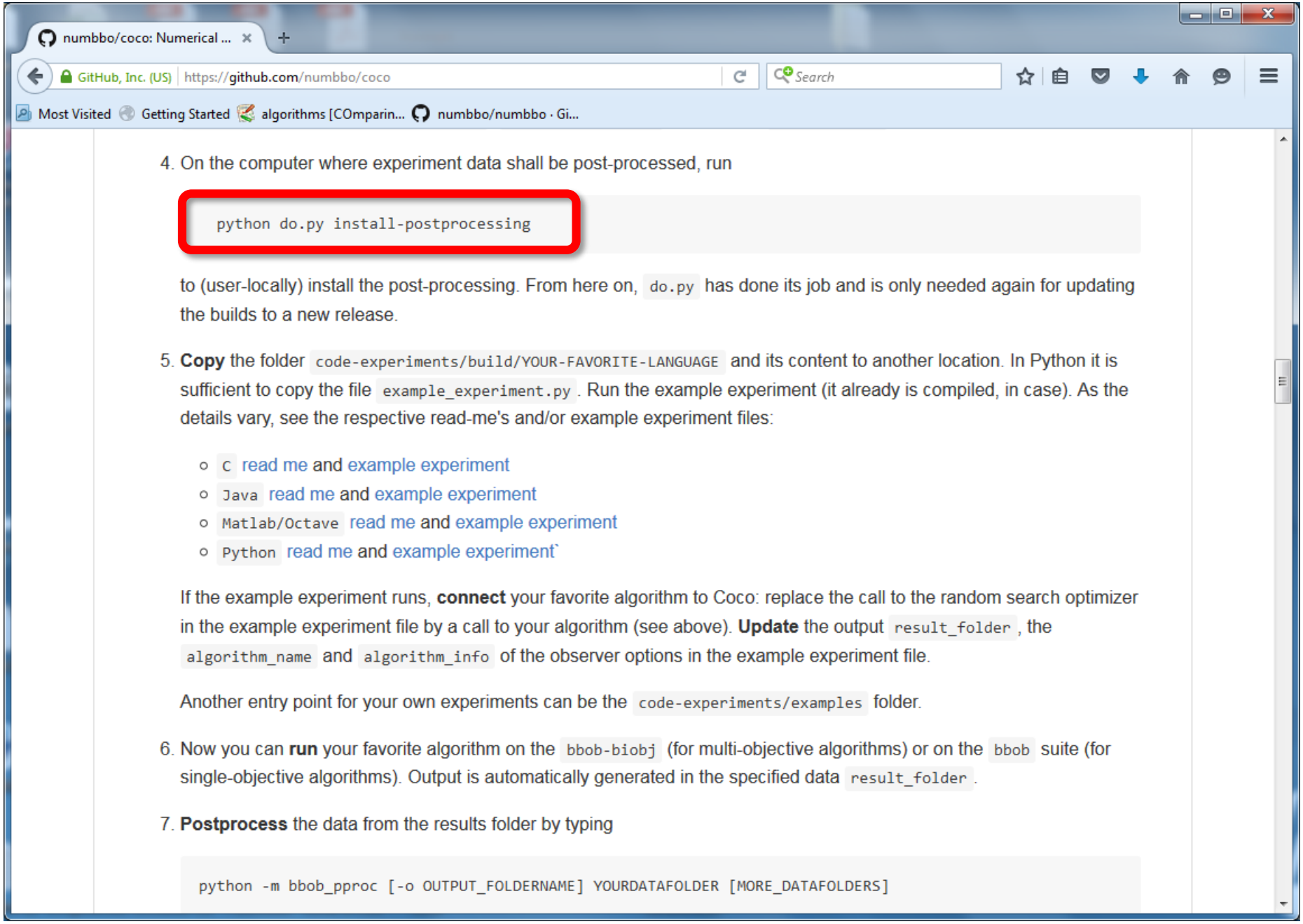
   CAVEAT: this code is still under heavy development. The record of official releases can be found here. The latest release corresponds to the master branch as linked above.
3. In a system shell, `cd` **into** the `coco` or `coco-<version>` folder (framework root), where the file `do.py` can be found. Type, i.e. **execute**, one of the following commands once

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

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

https://github.com/numbbo/coco



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, `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 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.

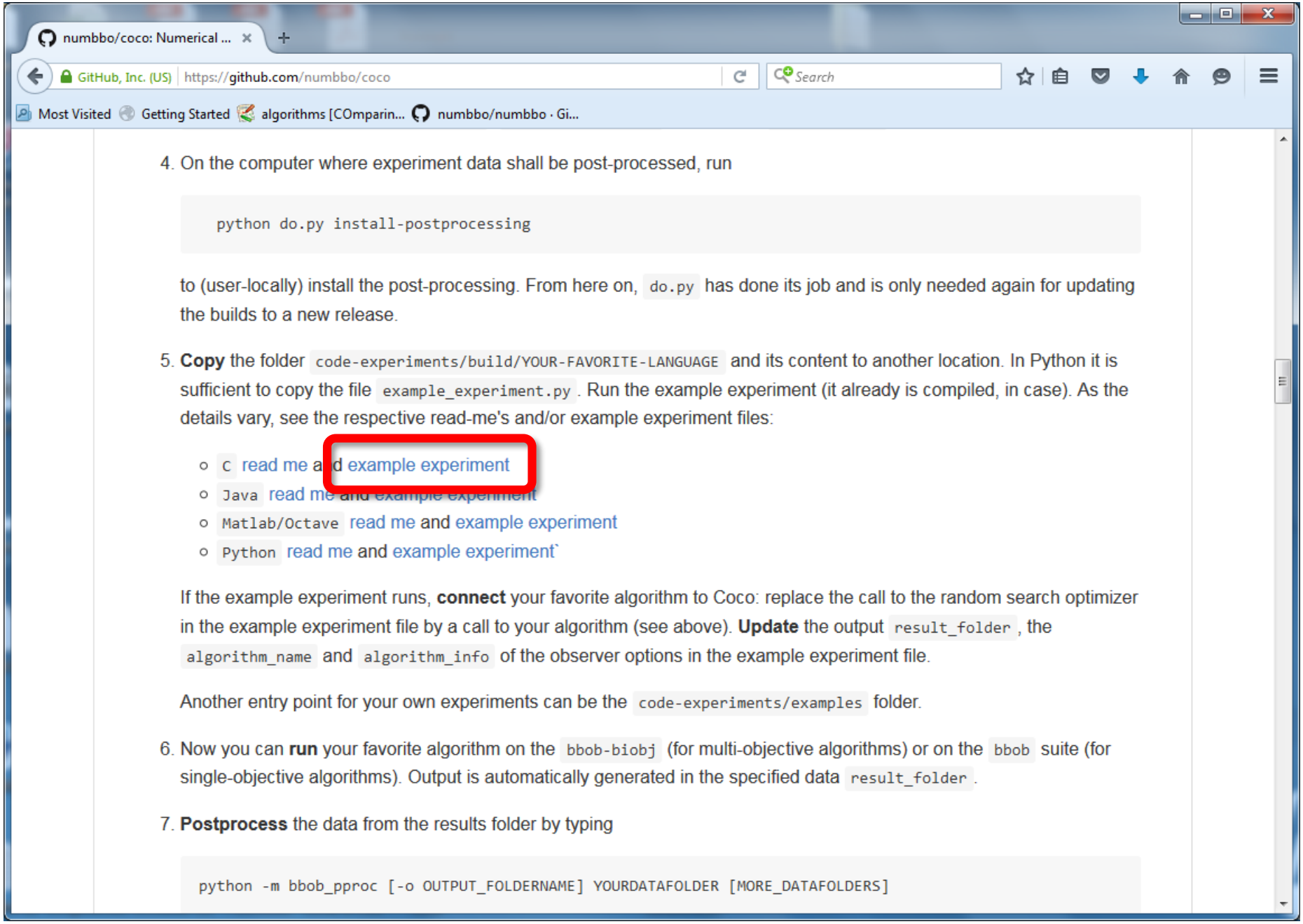
   Another entry point for your own experiments can be the `code-experiments/examples` folder.

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

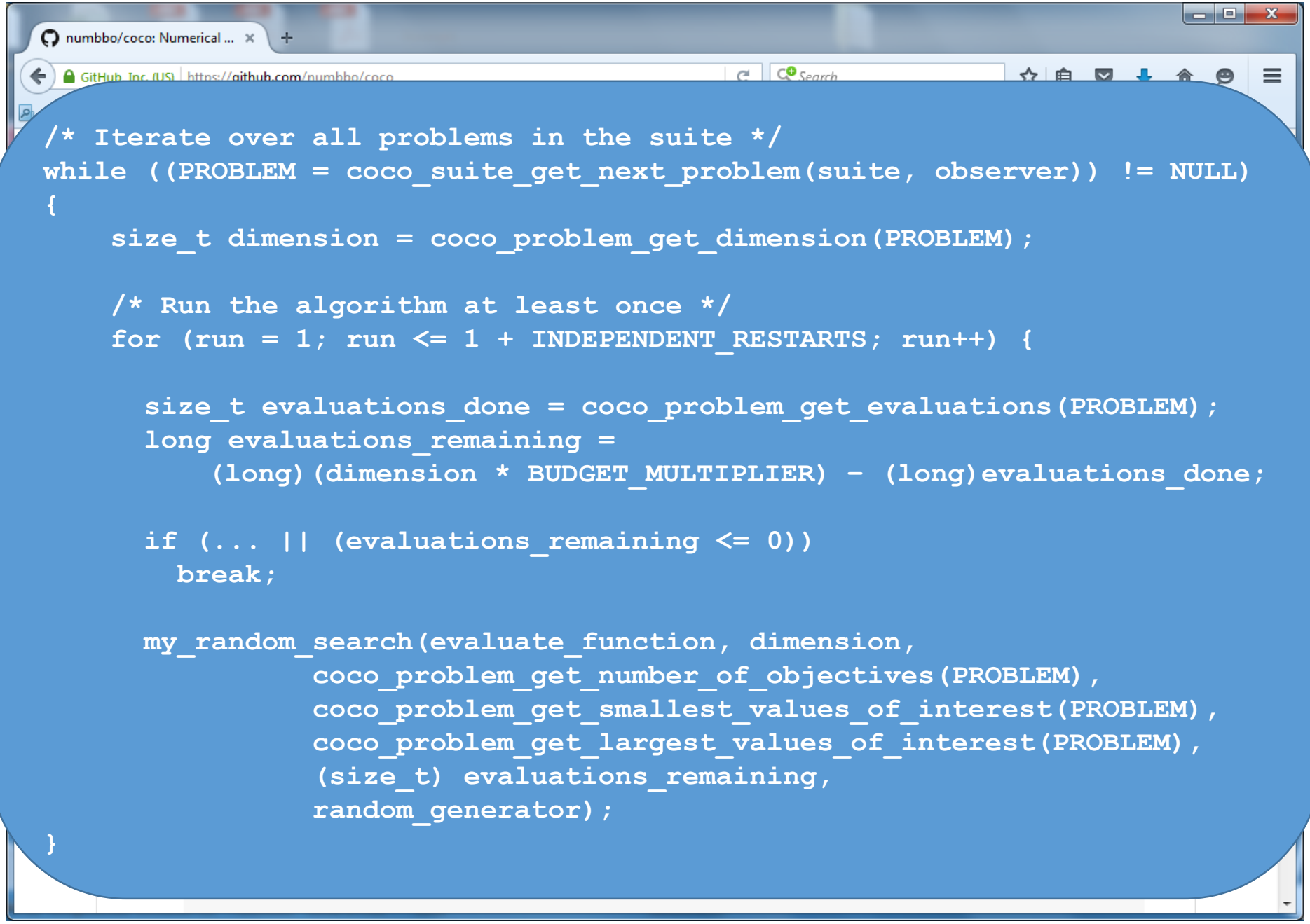
7. **Postprocess** the data from the results folder by typing

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

https://github.com/numbbo/coco

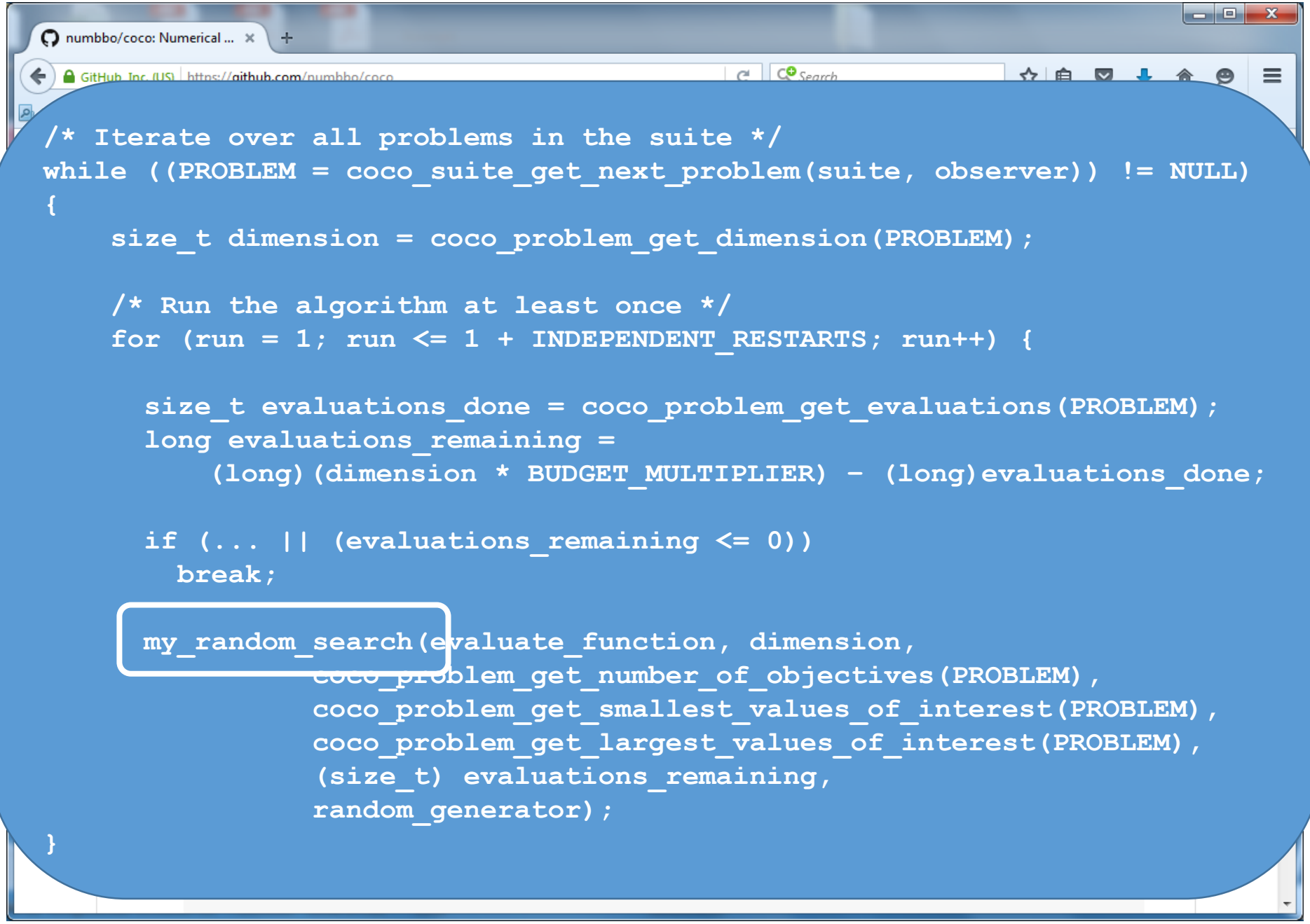


example_experiment.c



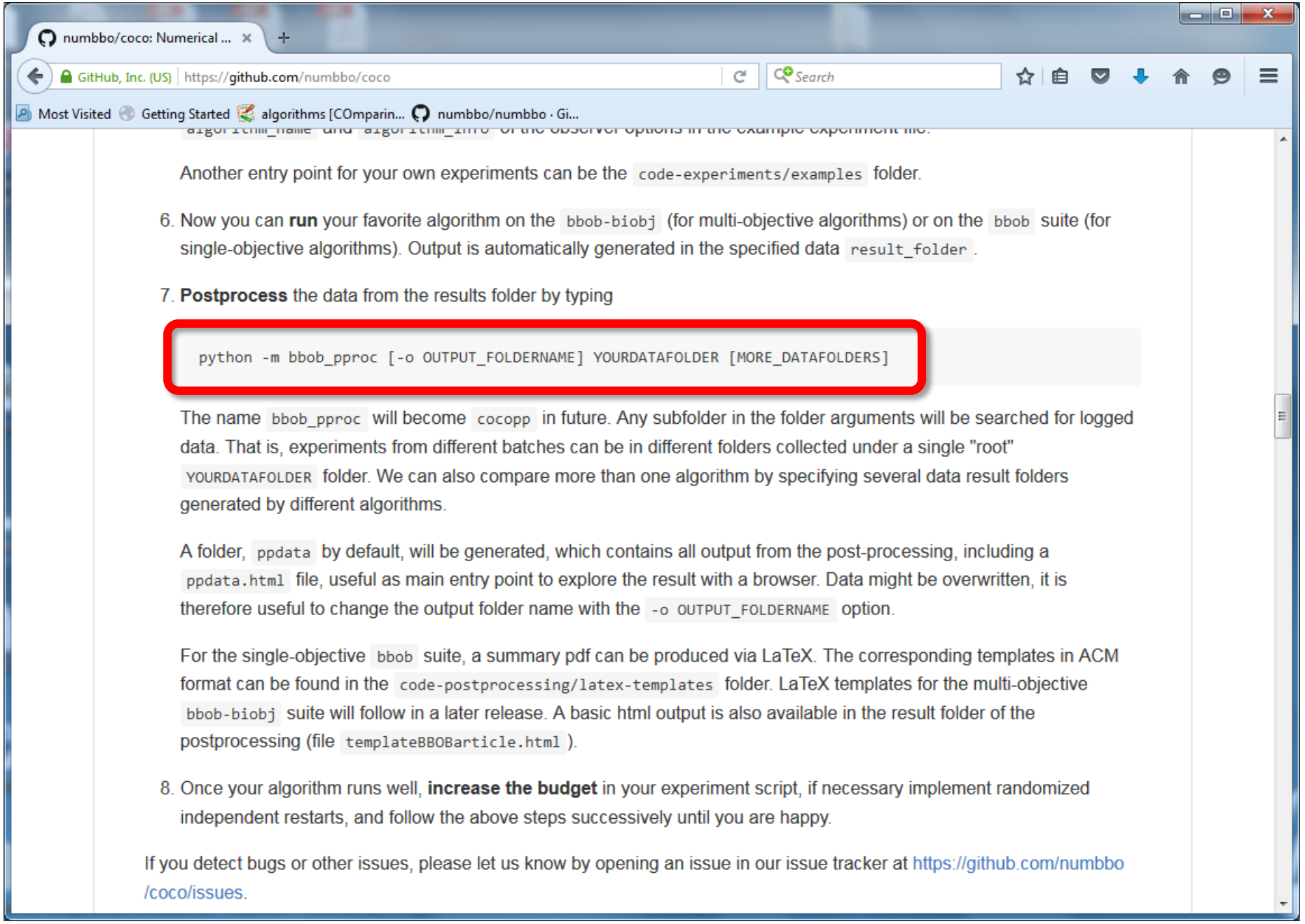

```
/* Iterate over all problems in the suite */
while ((PROBLEM = coco_suite_get_next_problem(suite, observer)) != NULL)
{
    size_t dimension = coco_problem_get_dimension(PROBLEM);

    /* 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);
}
```

example_experiment.c

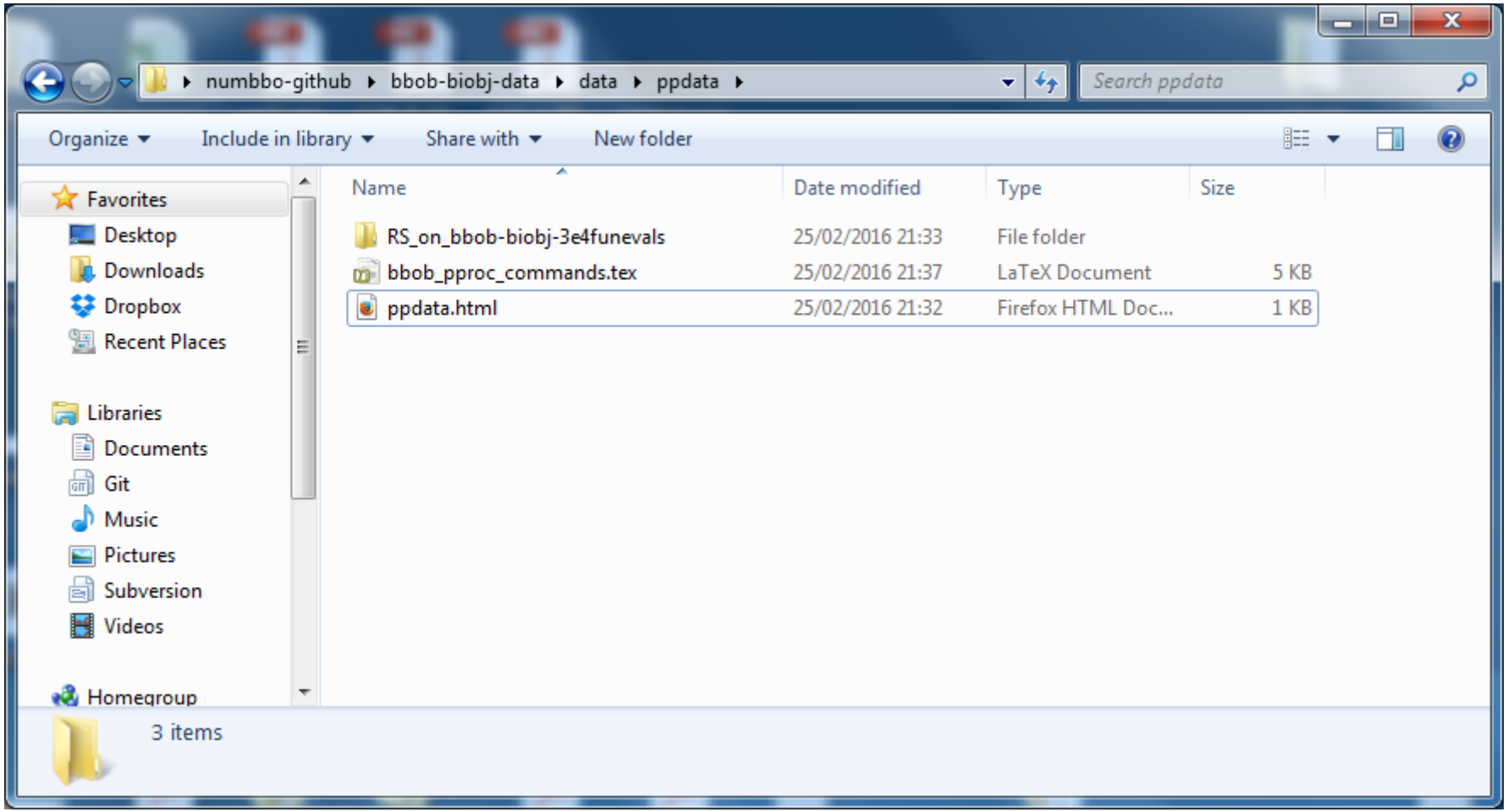



```
/* Iterate over all problems in the suite */
while ((PROBLEM = coco_suite_get_next_problem(suite, observer)) != NULL)
{
    size_t dimension = coco_problem_get_dimension(PROBLEM);

    /* 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);
}
```

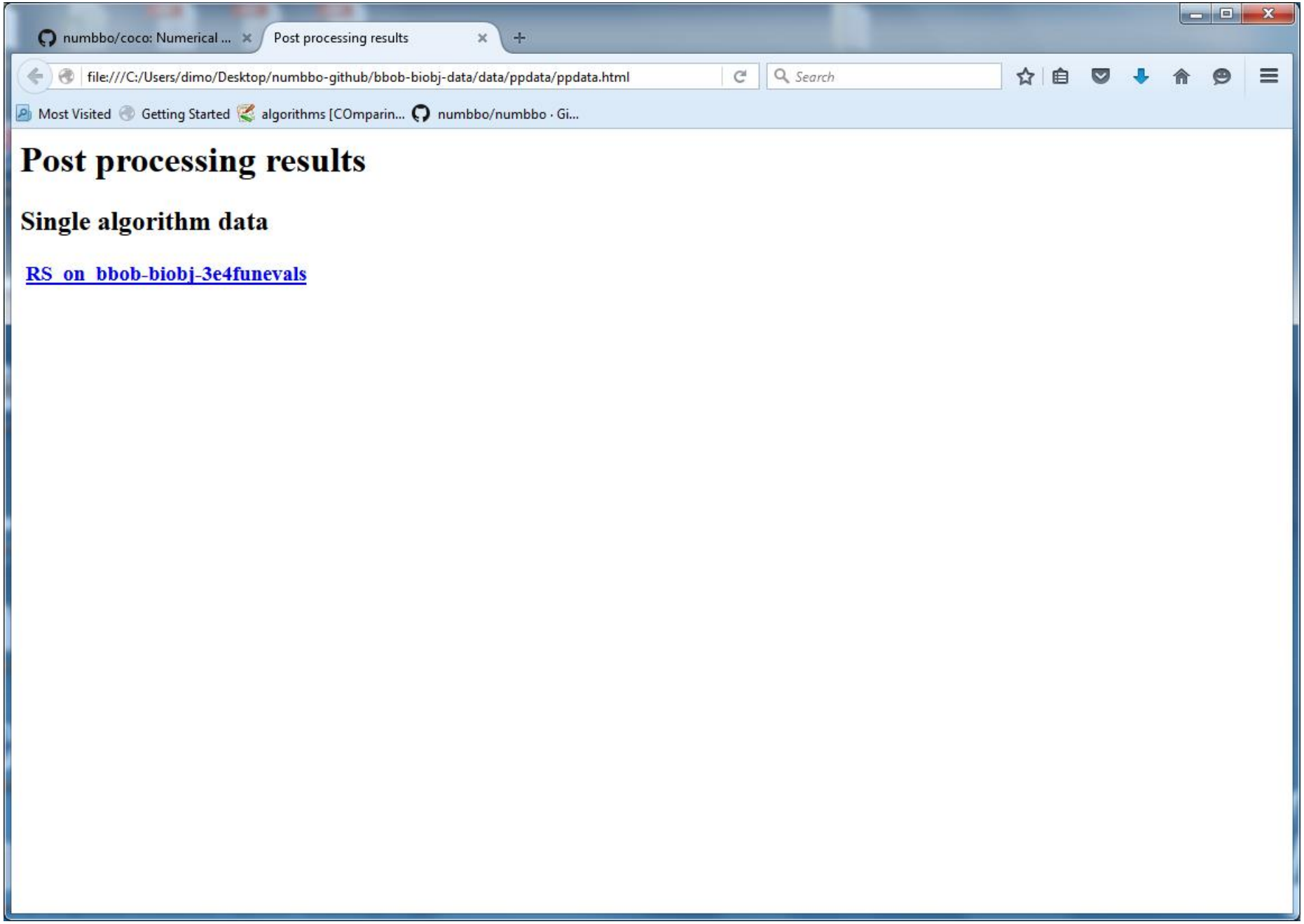
https://github.com/numbbo/coco



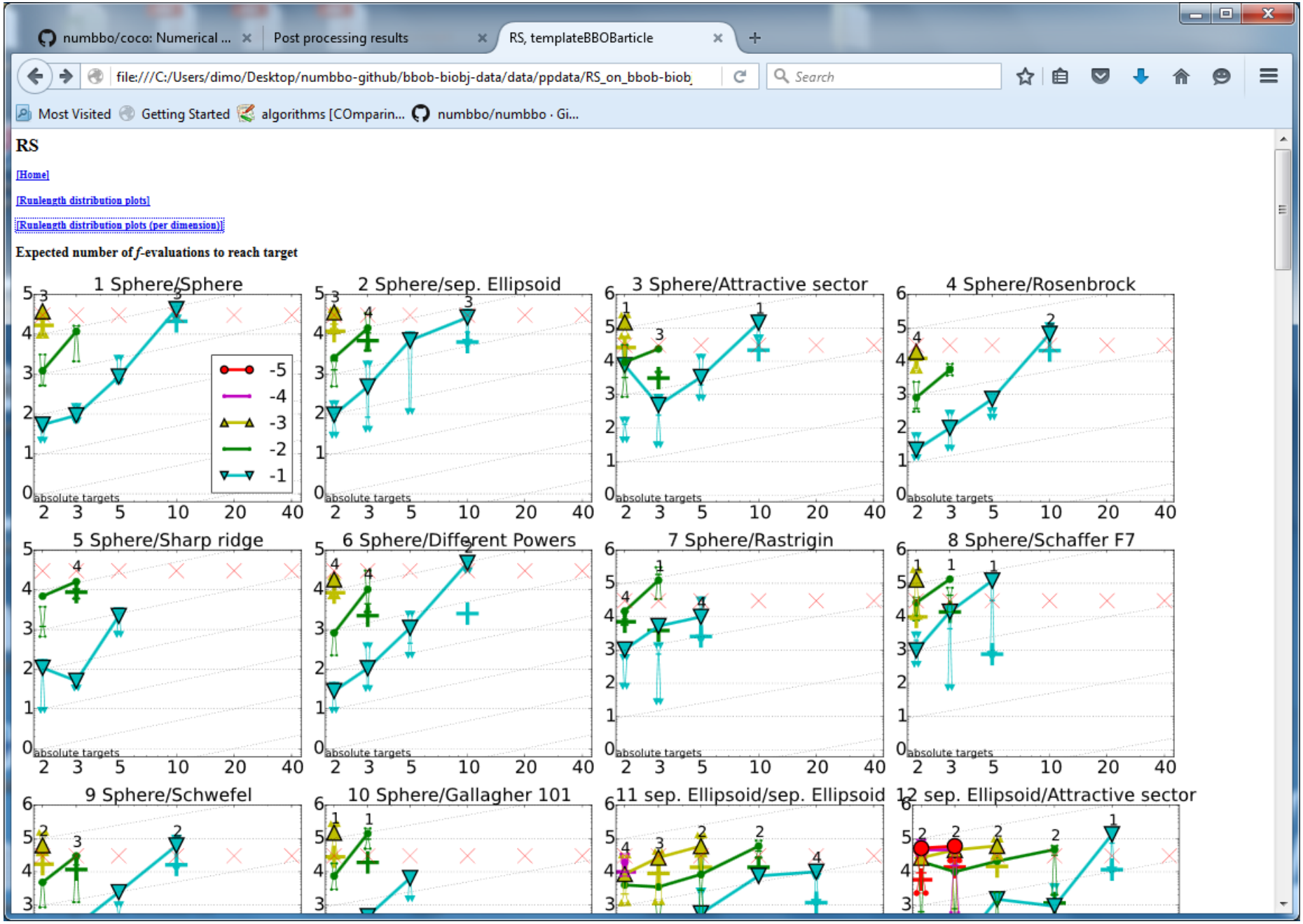
result folder



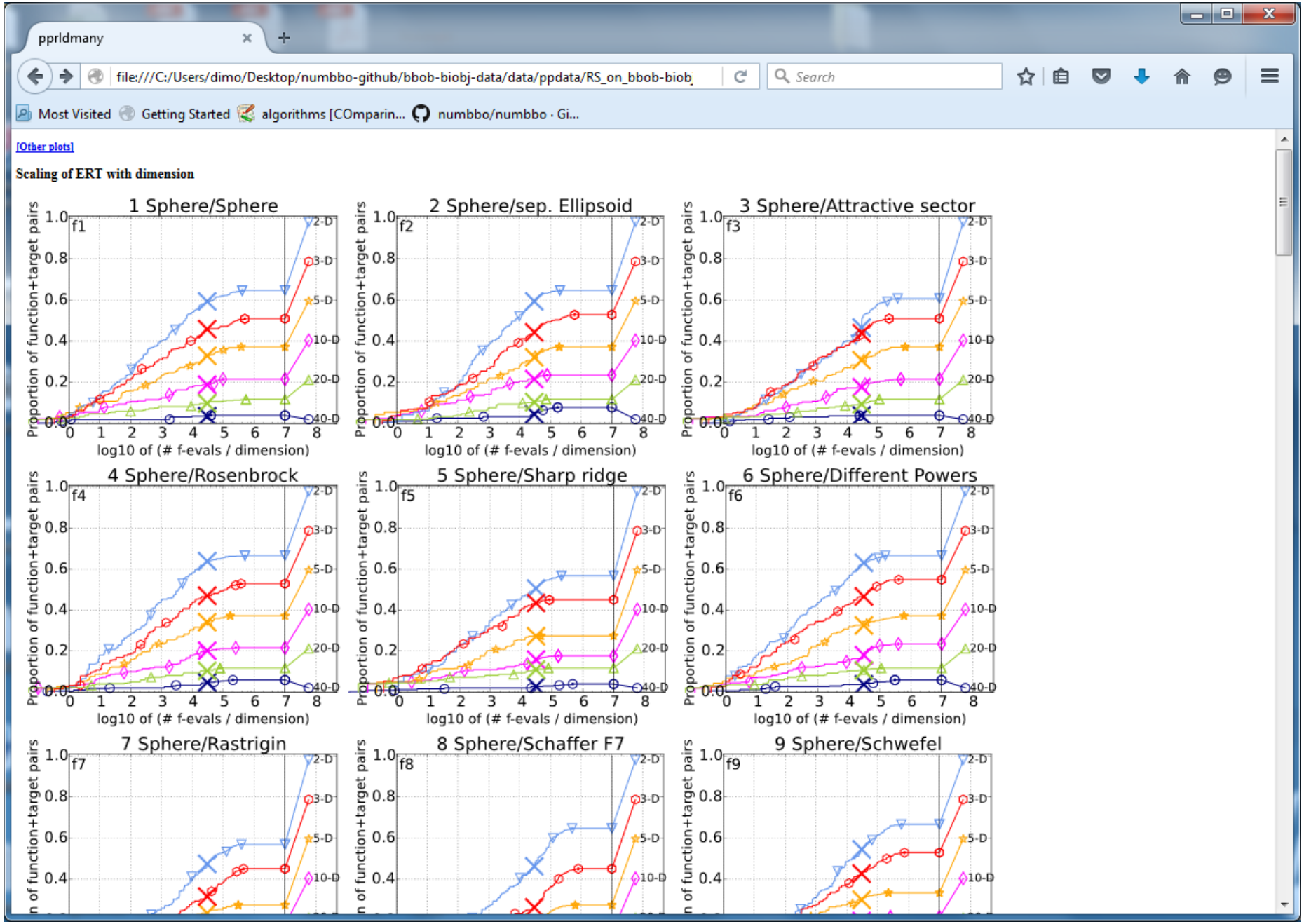
automatically generated results



automatically generated results



automatically generated results



doesn't look too complicated, does it?

[the devil is in the details ☺]

so far:

data for about 150 algorithm variants

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?

remind separability

- a number of testbeds are around

Available Test Suites in COCO

• bbob	24 noiseless fcts	140+ algo data sets
• bbob-noisy	30 noisy fcts	40+ algo data sets
• bbob-biobj	55 bi-objective fcts	new in 2016

Under development:

- large-scale versions
- linearly constrained test suite

Long-term goals:

- combining difficulties
- almost real-world problems
- real-world problems

How Do We Measure Performance?

- Account for invariance properties

  prediction of performance is based on “similarity”, ideally equivalence classes of functions

- 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?

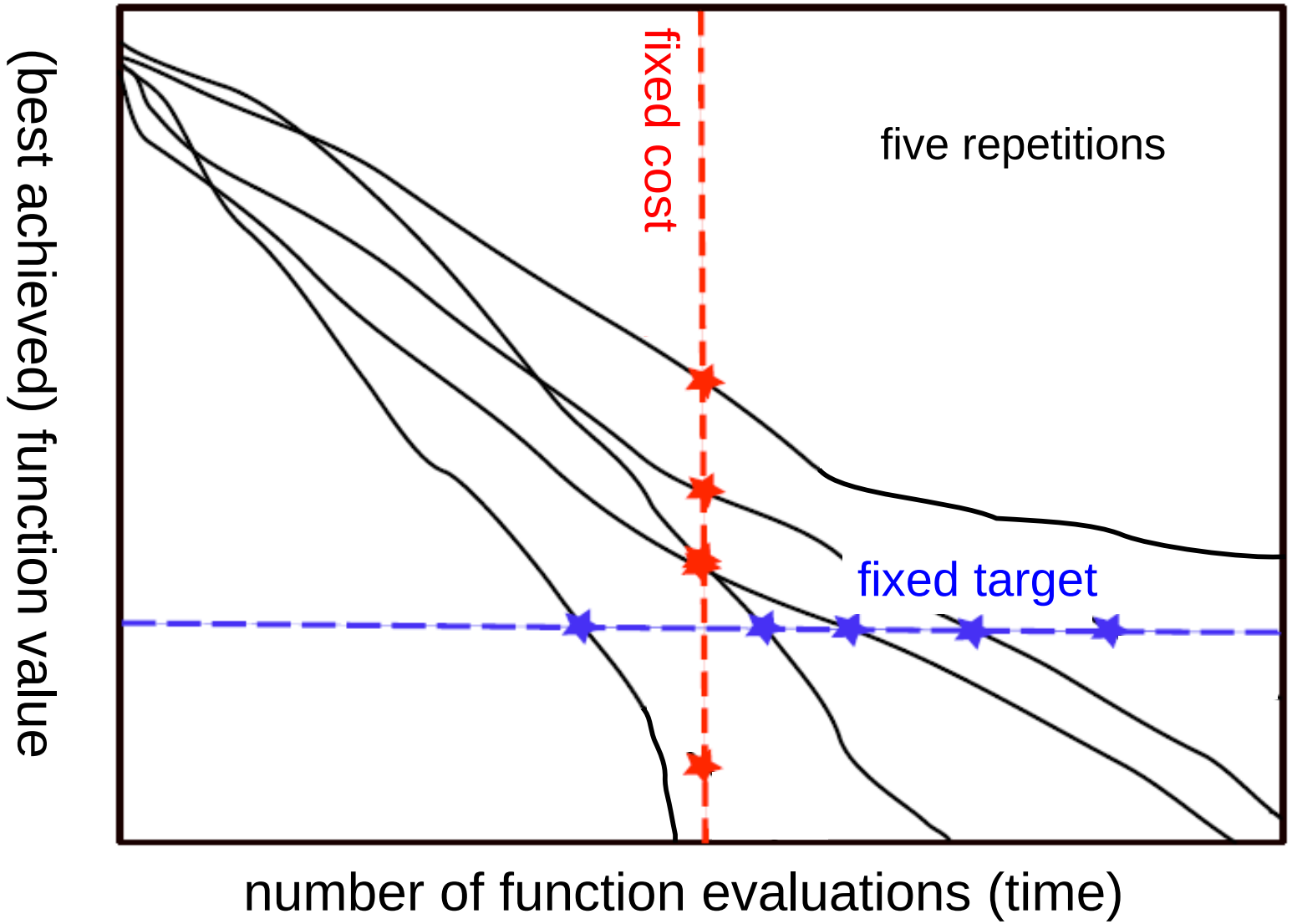
Two objectives:

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

For measuring performance: fix one and measure the other

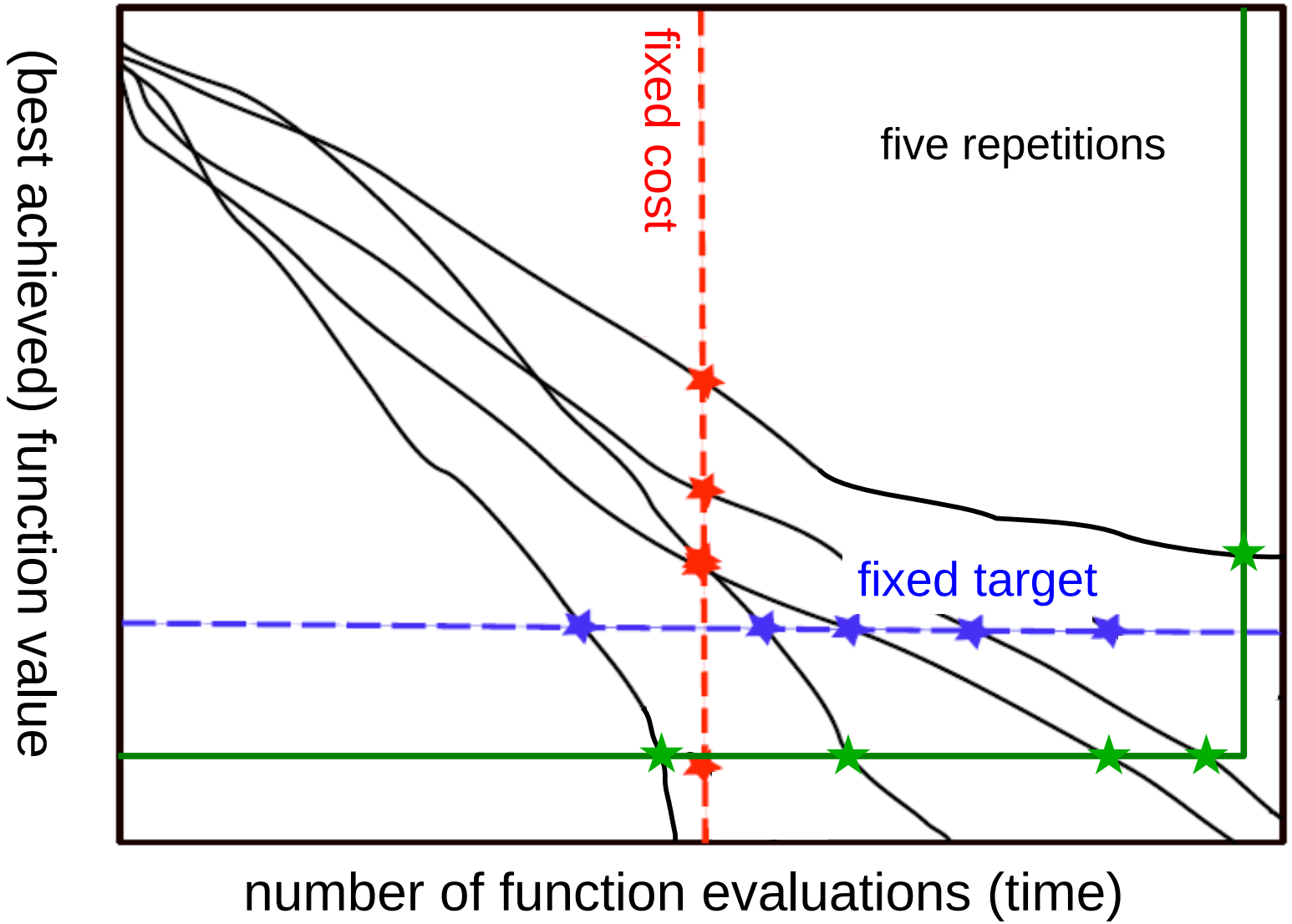
Measuring Performance Empirically

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



Measuring Performance Empirically

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

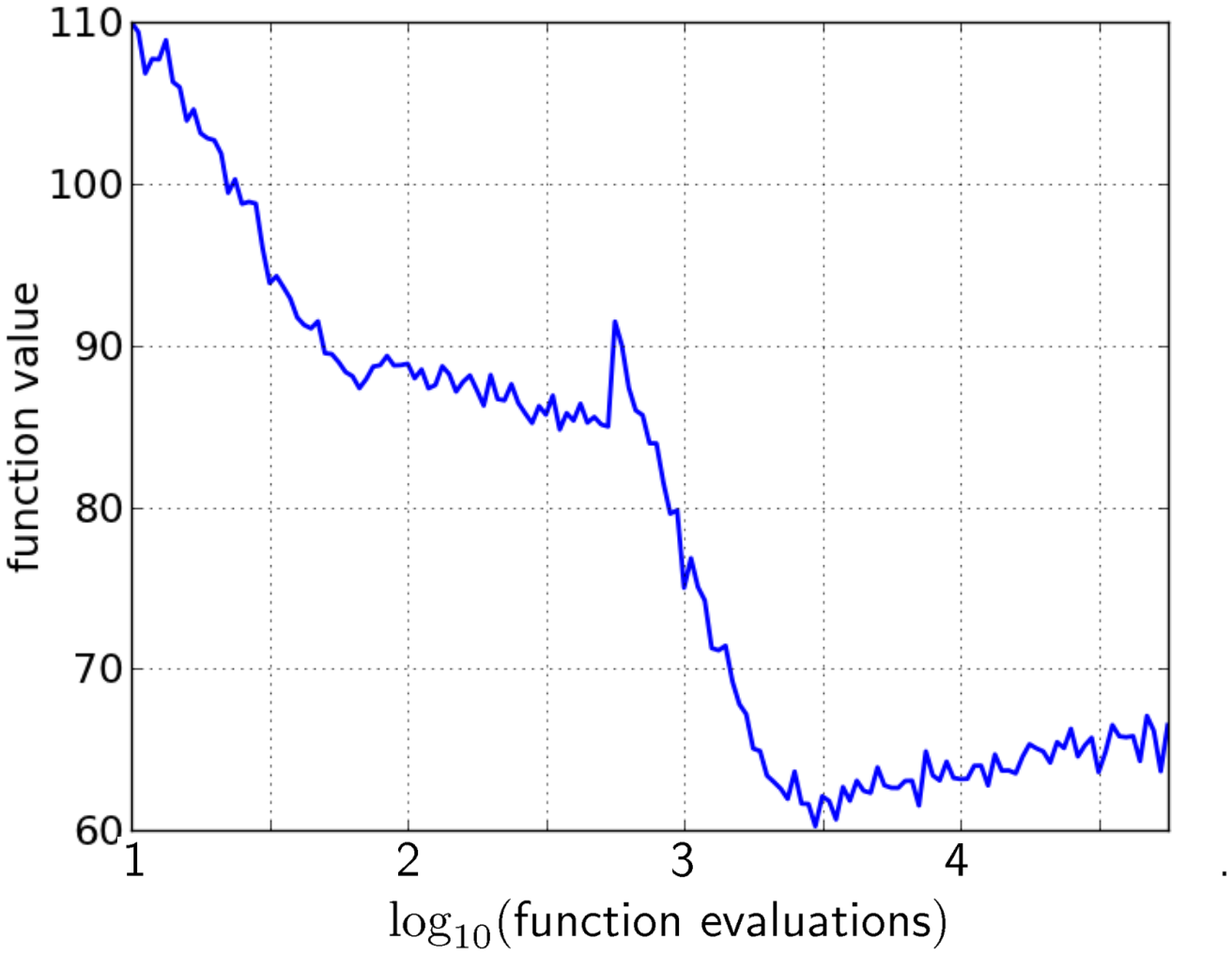


ECDF:

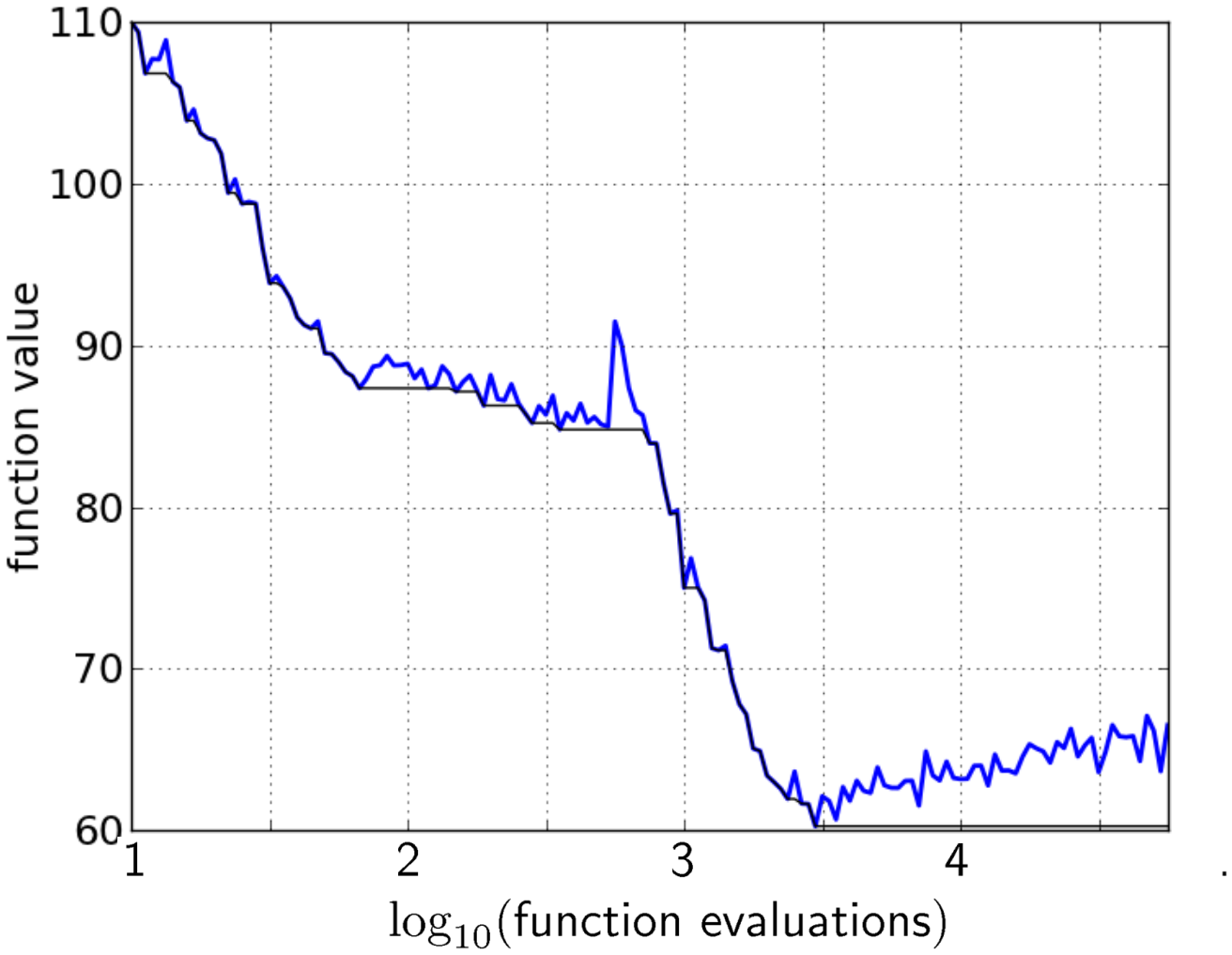
Empirical Cumulative Distribution Function of the Runtime

[aka data profile]

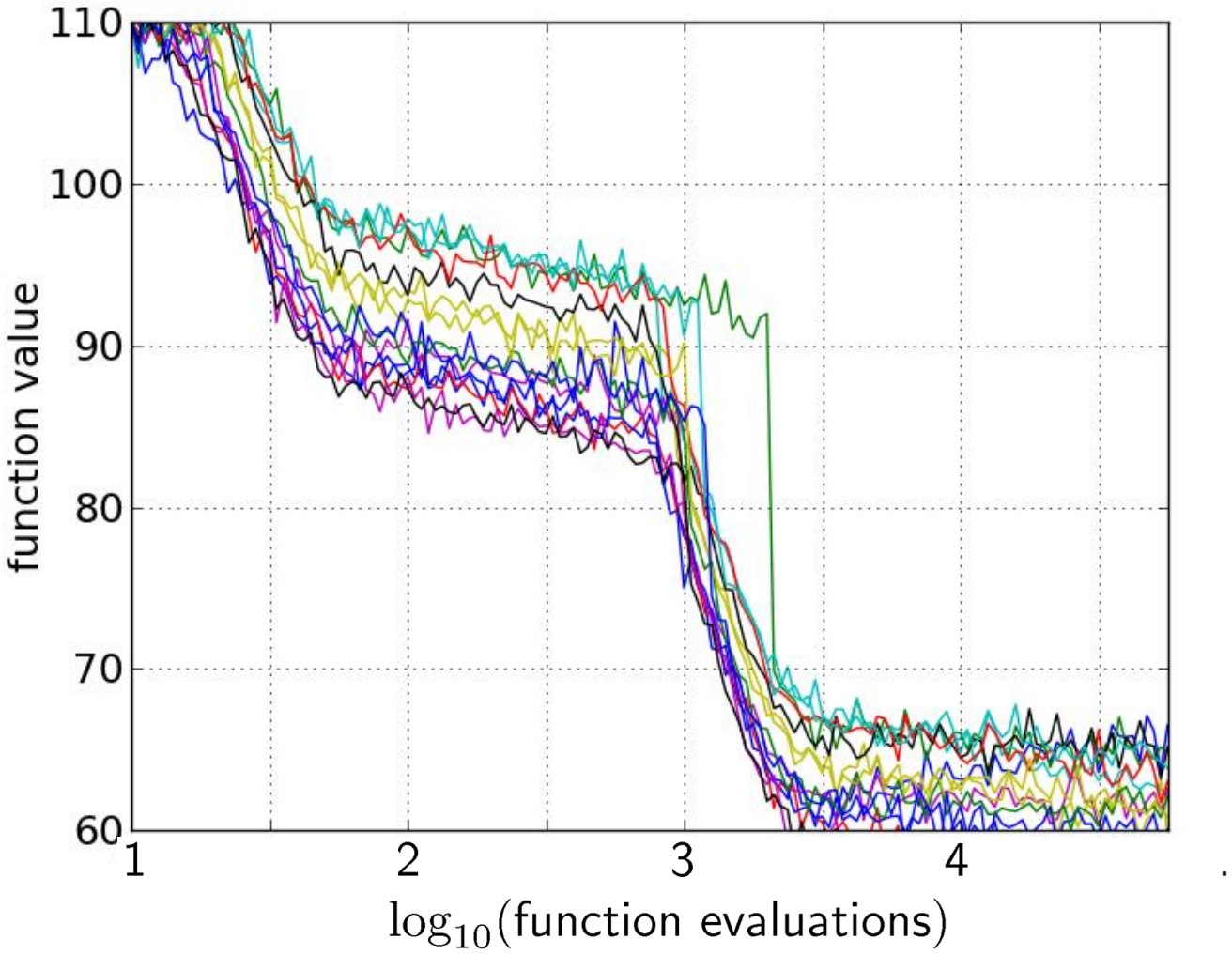
A Convergence Graph



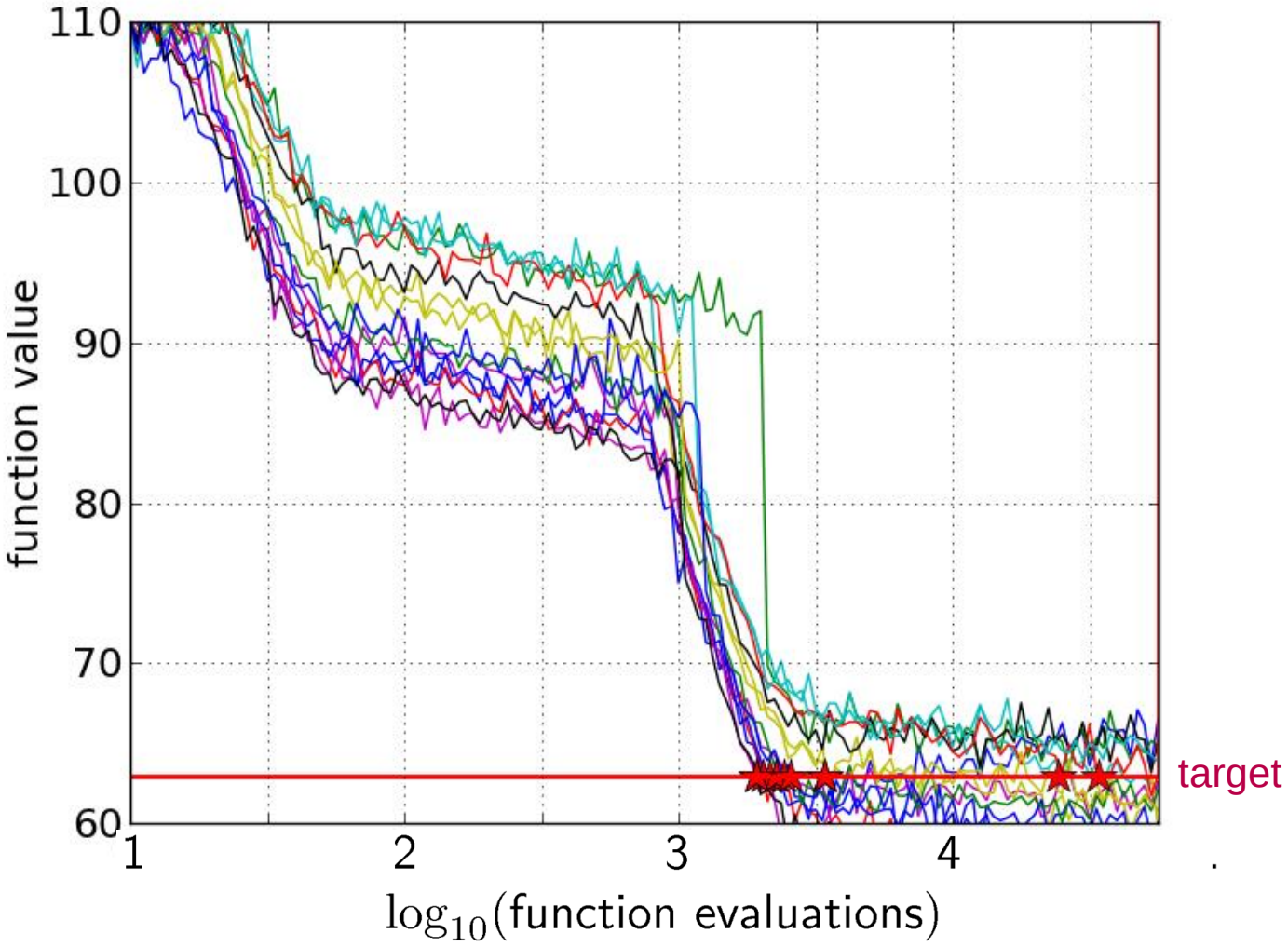
First Hitting Time is Monotonous



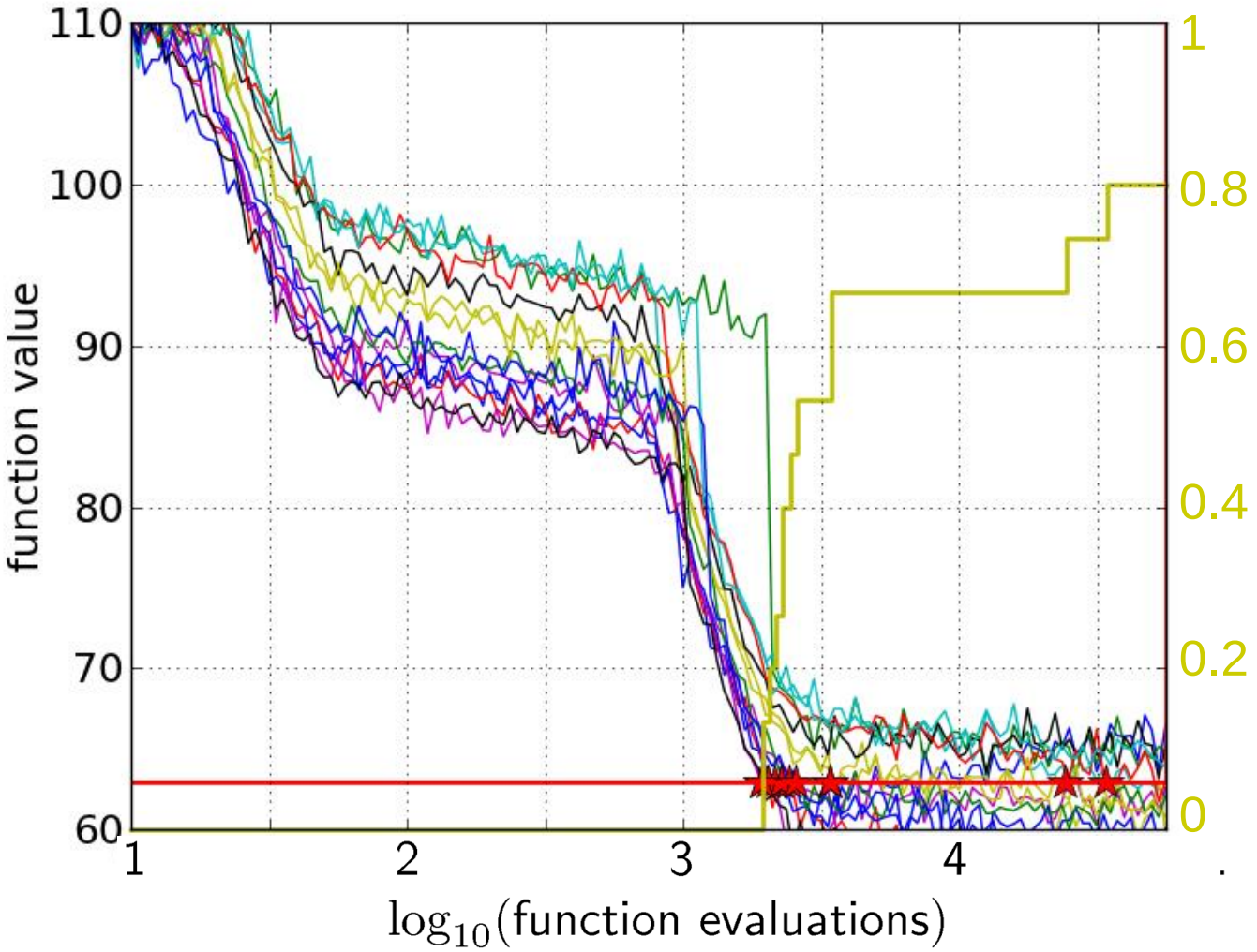
15 Runs


110
100
90
80
70
60
function value
1
2
3
4
$\log_{10}$(function evaluations)

15 Runs ≤ 15 Runtime Data Points



Empirical Cumulative Distribution

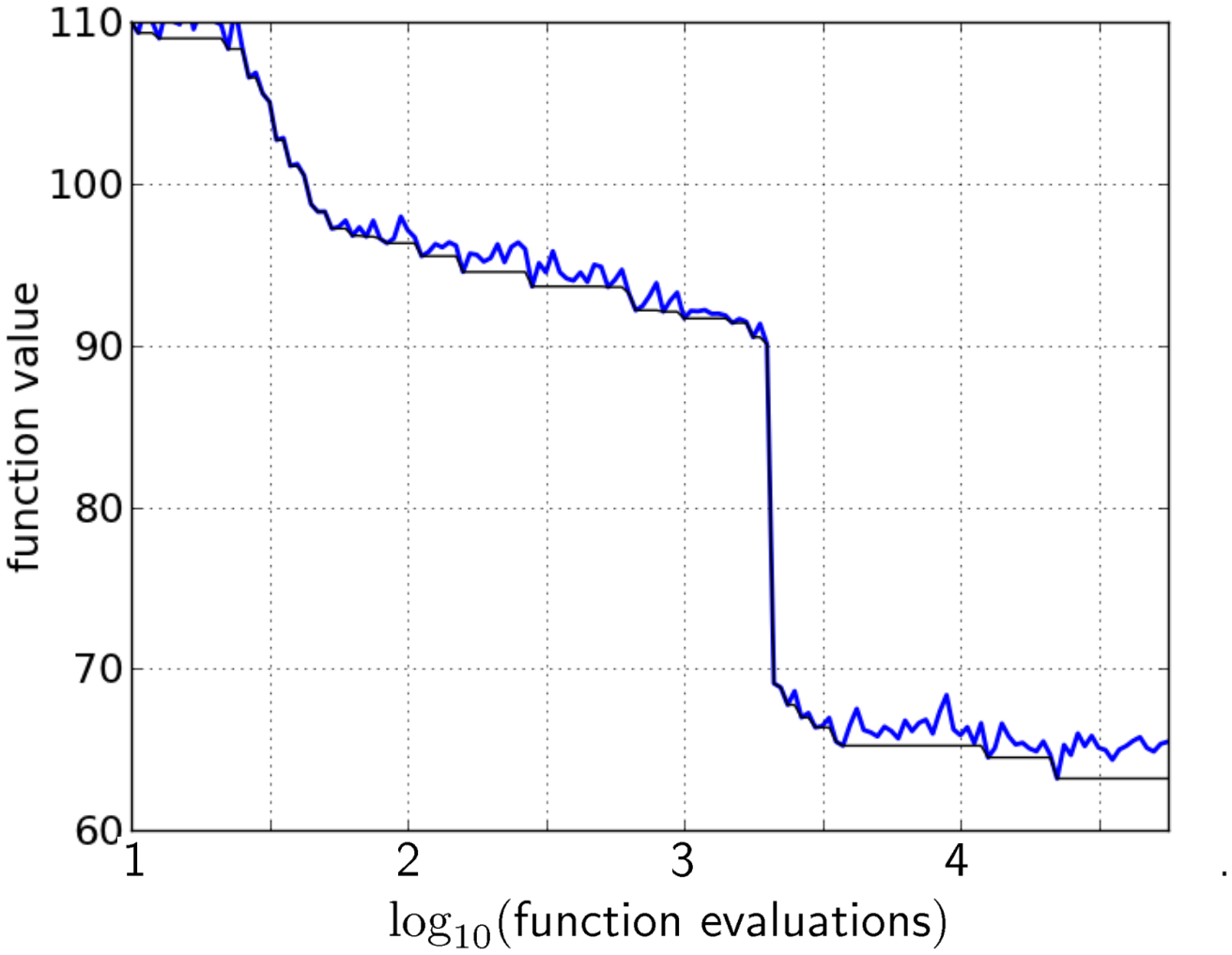


the ECDF of run lengths to reach the target

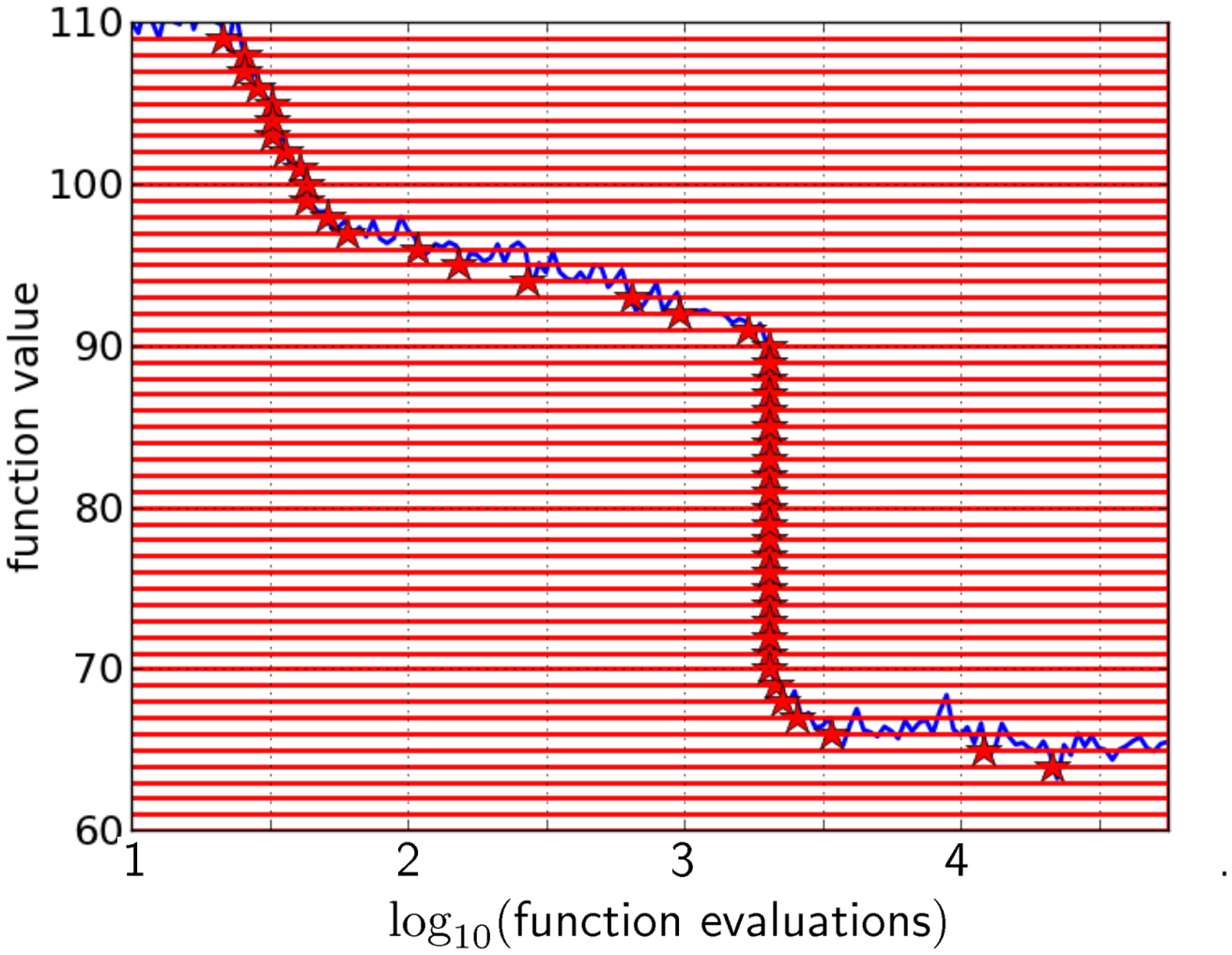
- has for each data point a vertical step of constant size
- displays for each x-value (budget) the count of observations to the left (first hitting times)

e.g. 60% of the runs need between 2000 and 4000 evaluations
80% of the runs reached the target

Reconstructing A Single Run

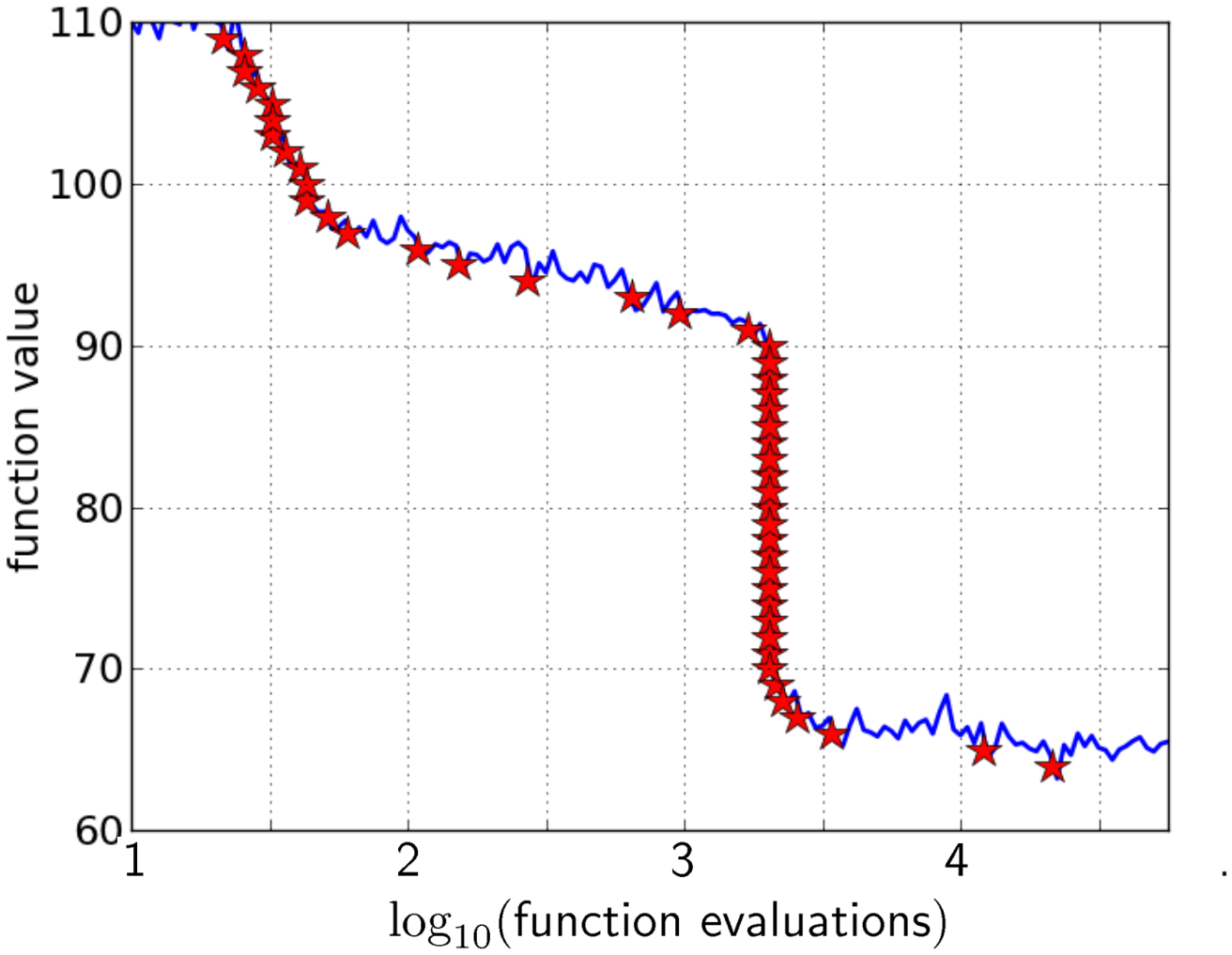


Reconstructing A Single Run

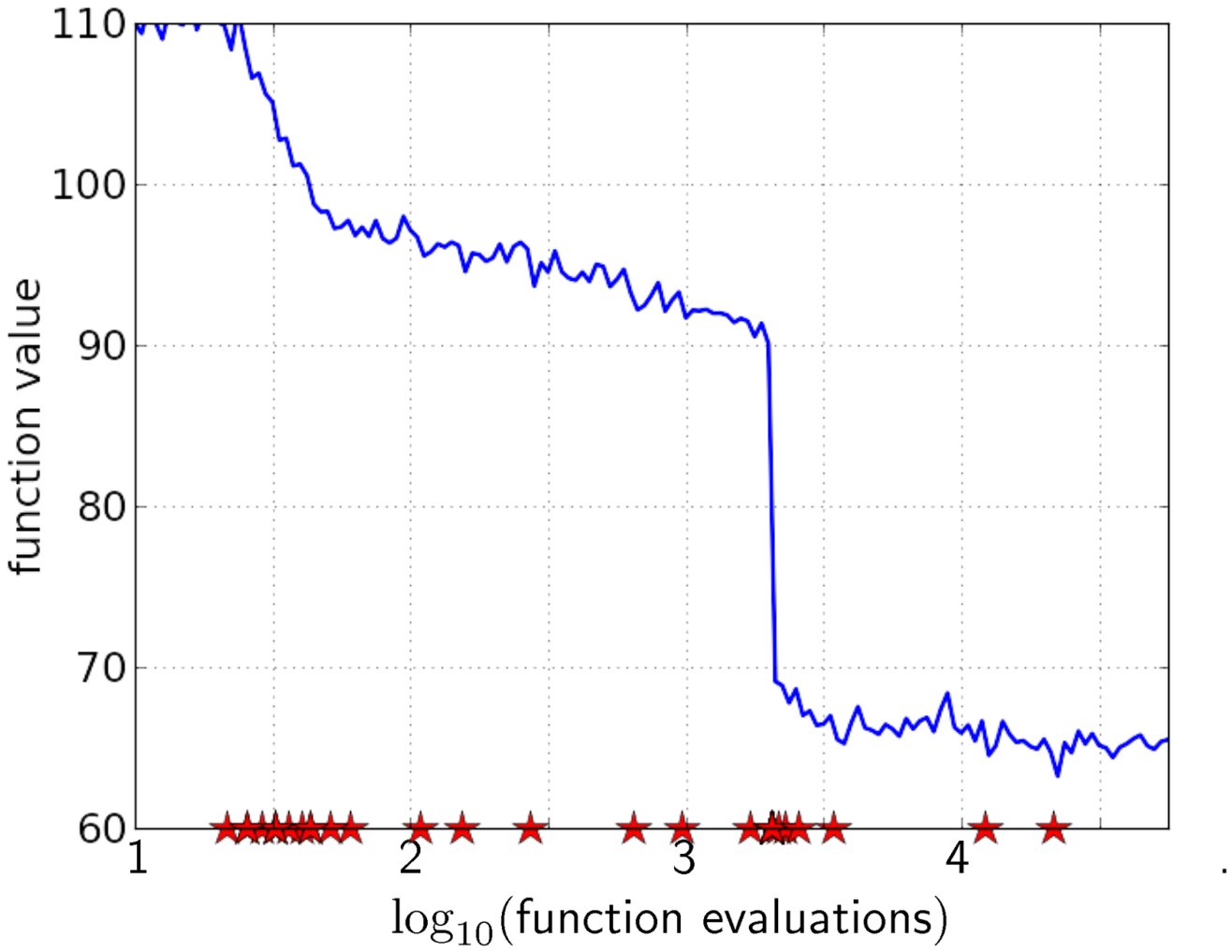


50 equally spaced targets

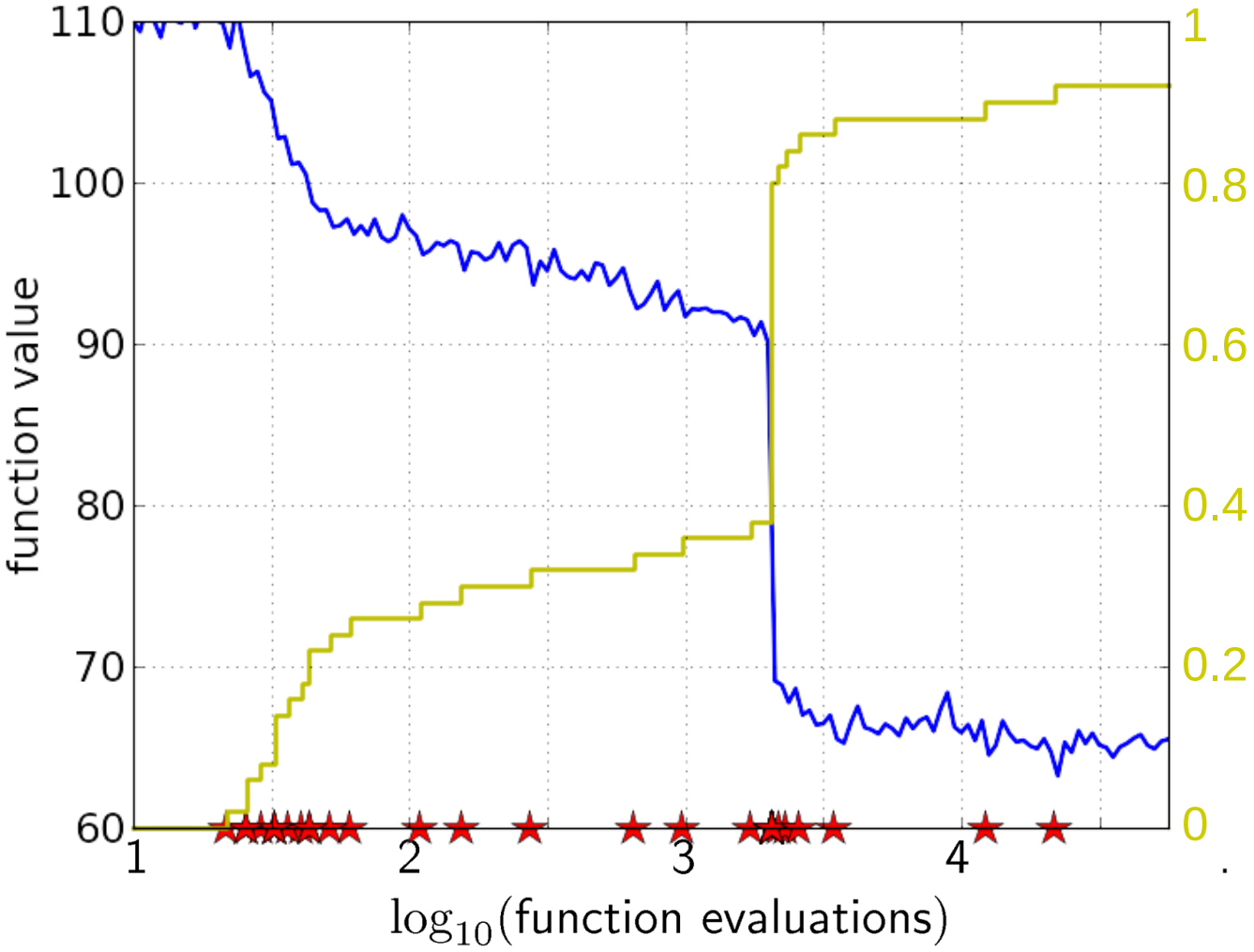
Reconstructing A Single Run



Reconstructing A Single Run

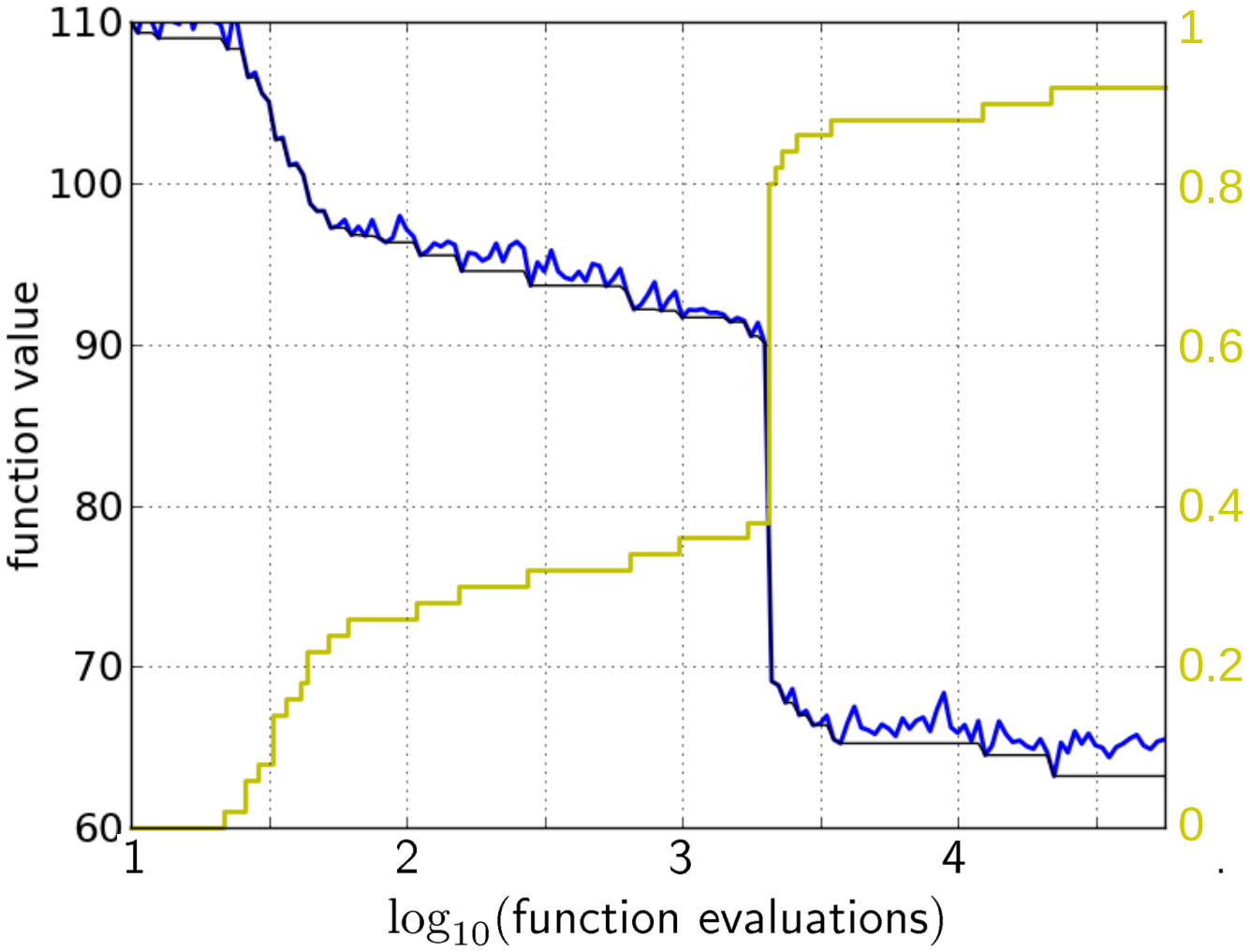


Reconstructing A Single Run



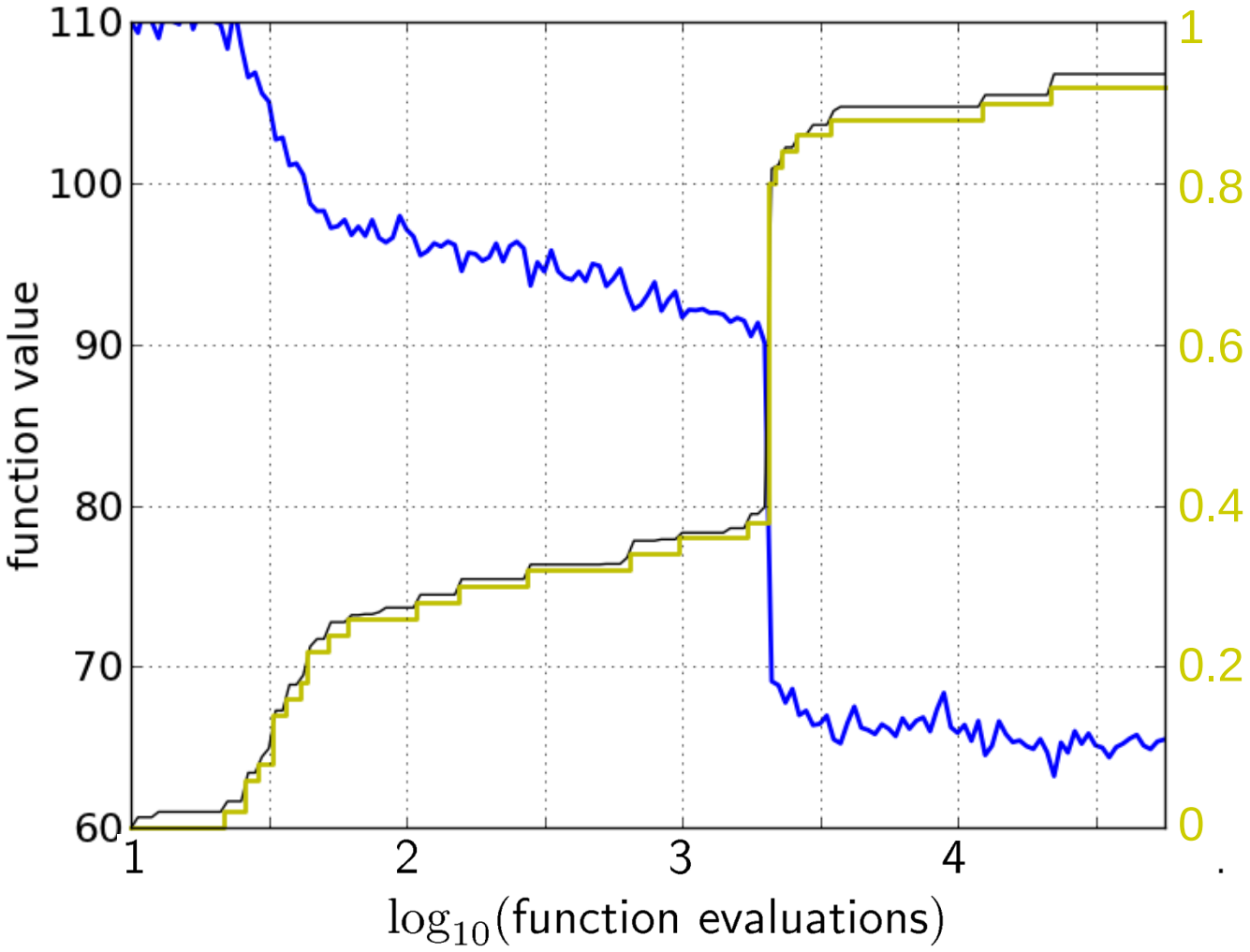
the empirical CDF
- makes a step for each star, is monotonous and displays for each budget the fraction of targets achieved within the budget

Reconstructing A Single Run



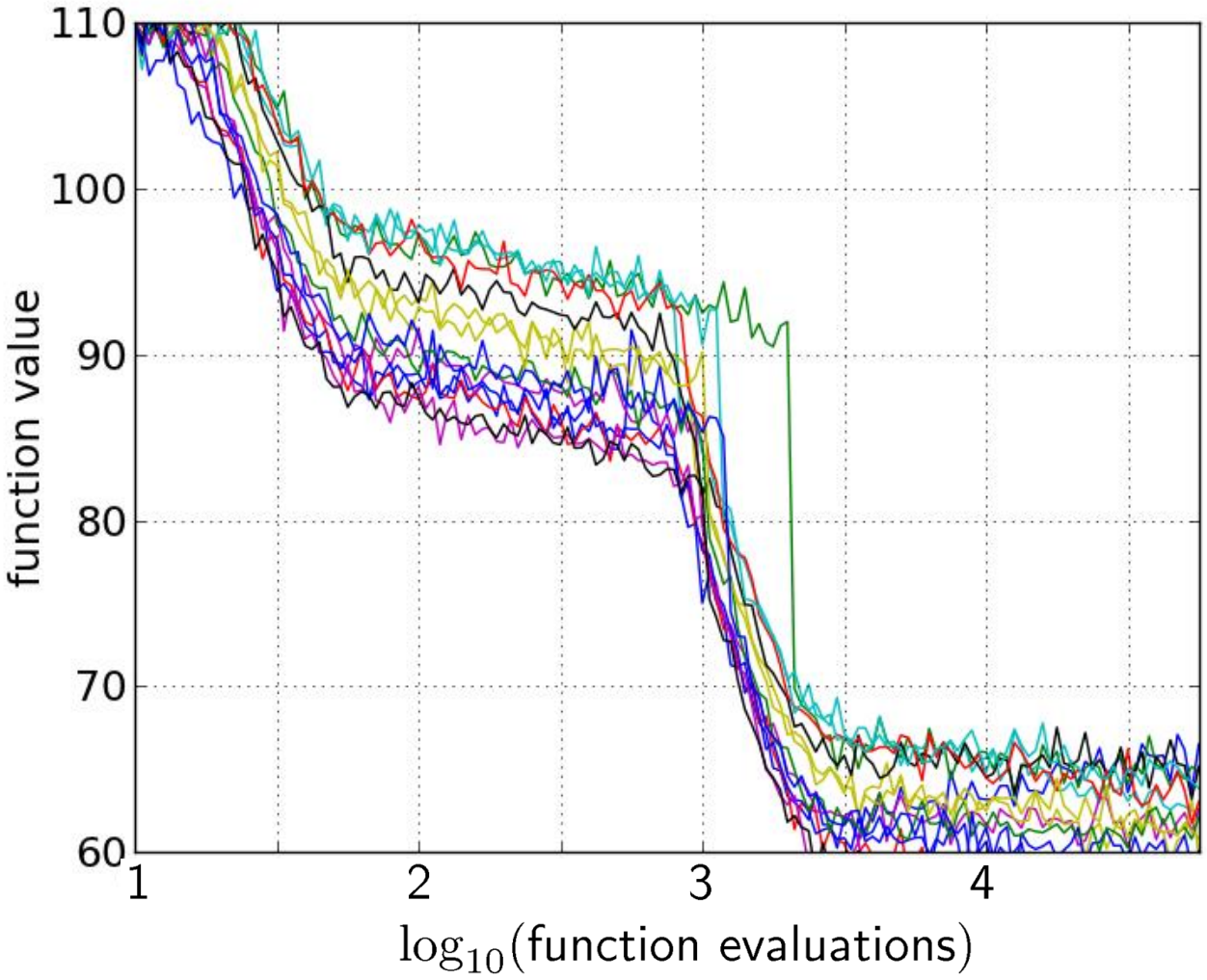
the ECDF recovers the monotonous graph, discretised and flipped

Reconstructing A Single Run



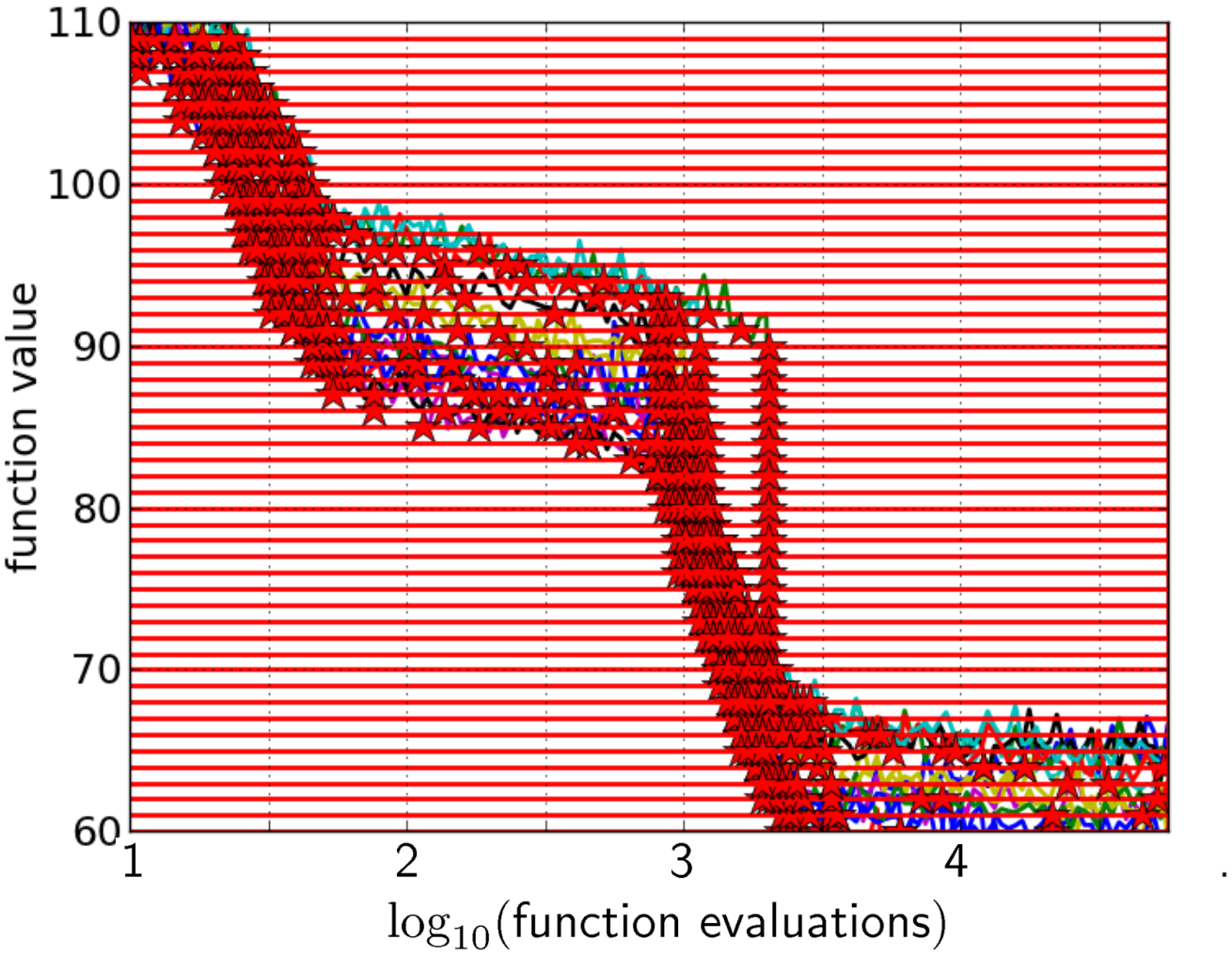
the ECDF recovers the monotonous graph, discretised and flipped

Aggregation



15 runs

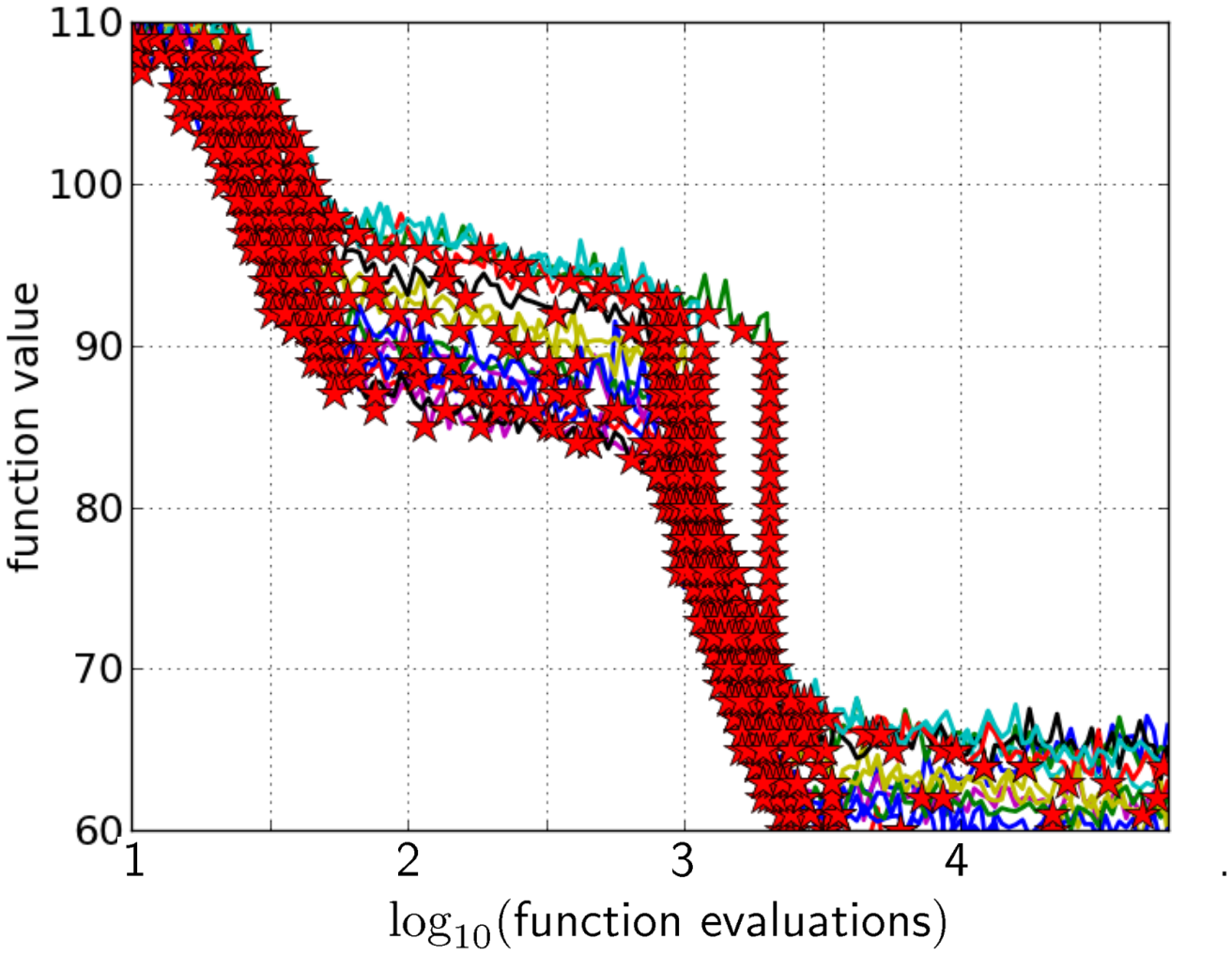
Aggregation



15 runs

50 targets

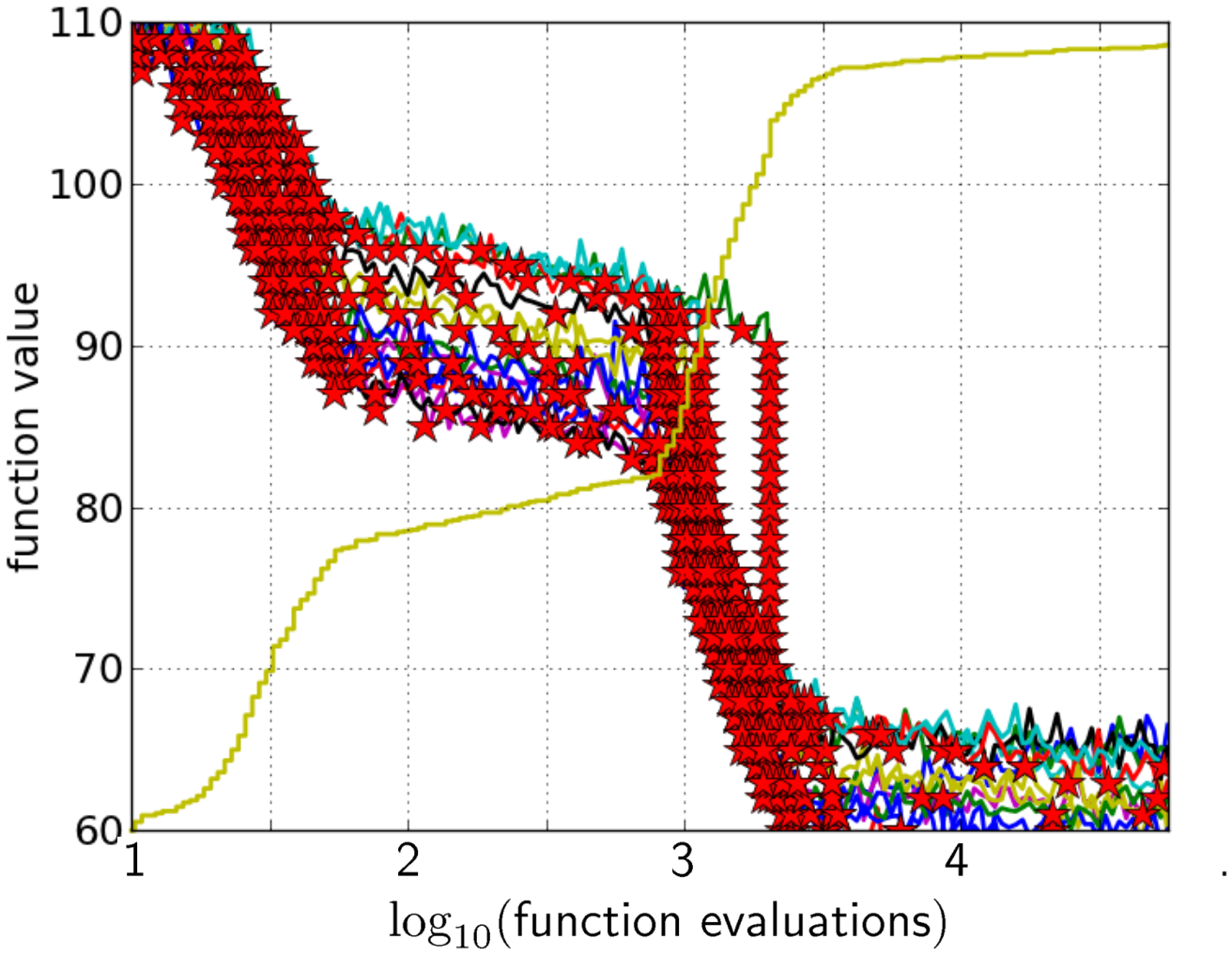
Aggregation



15 runs

50 targets

Aggregation

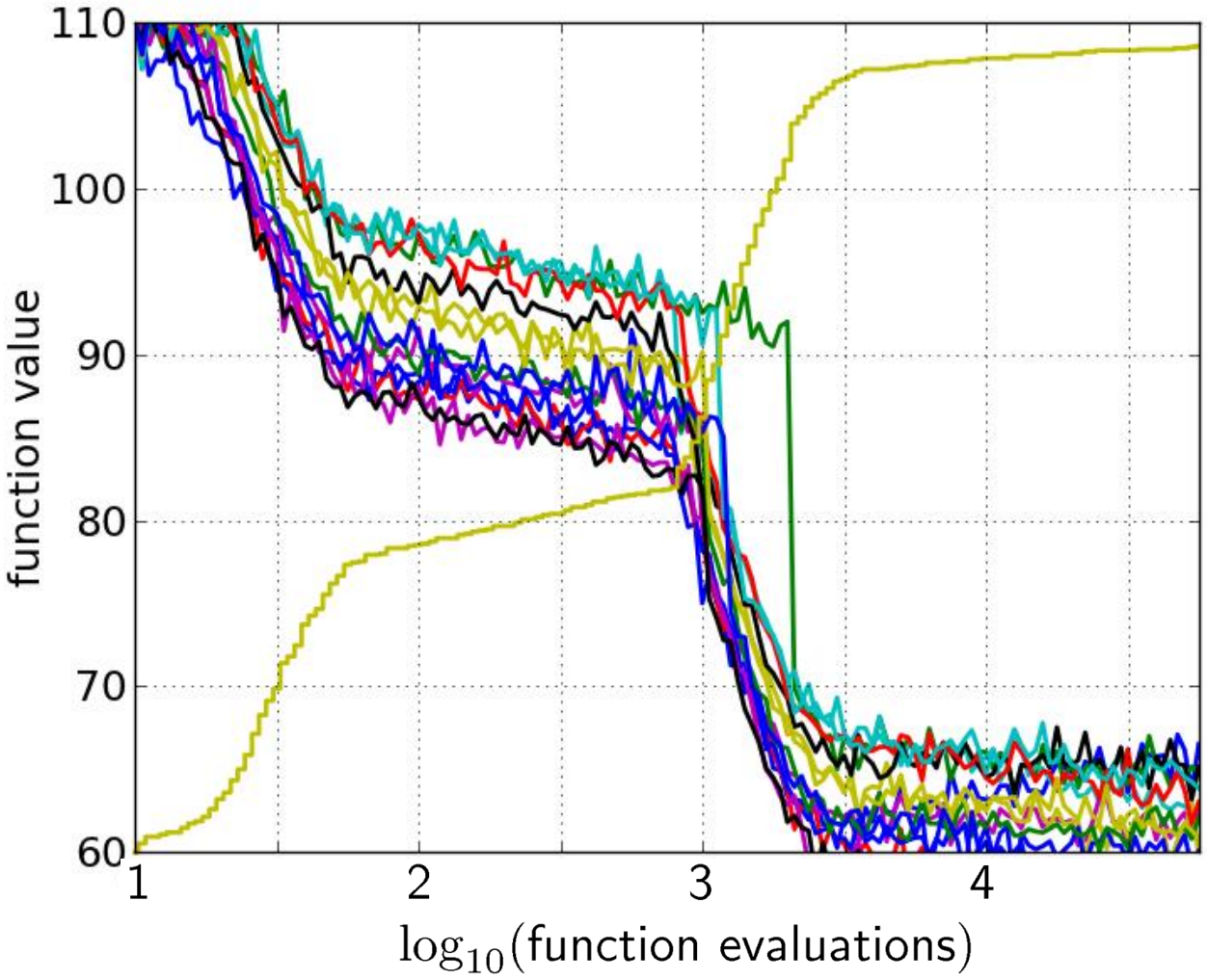


15 runs

50 targets

ECDF with 750 steps

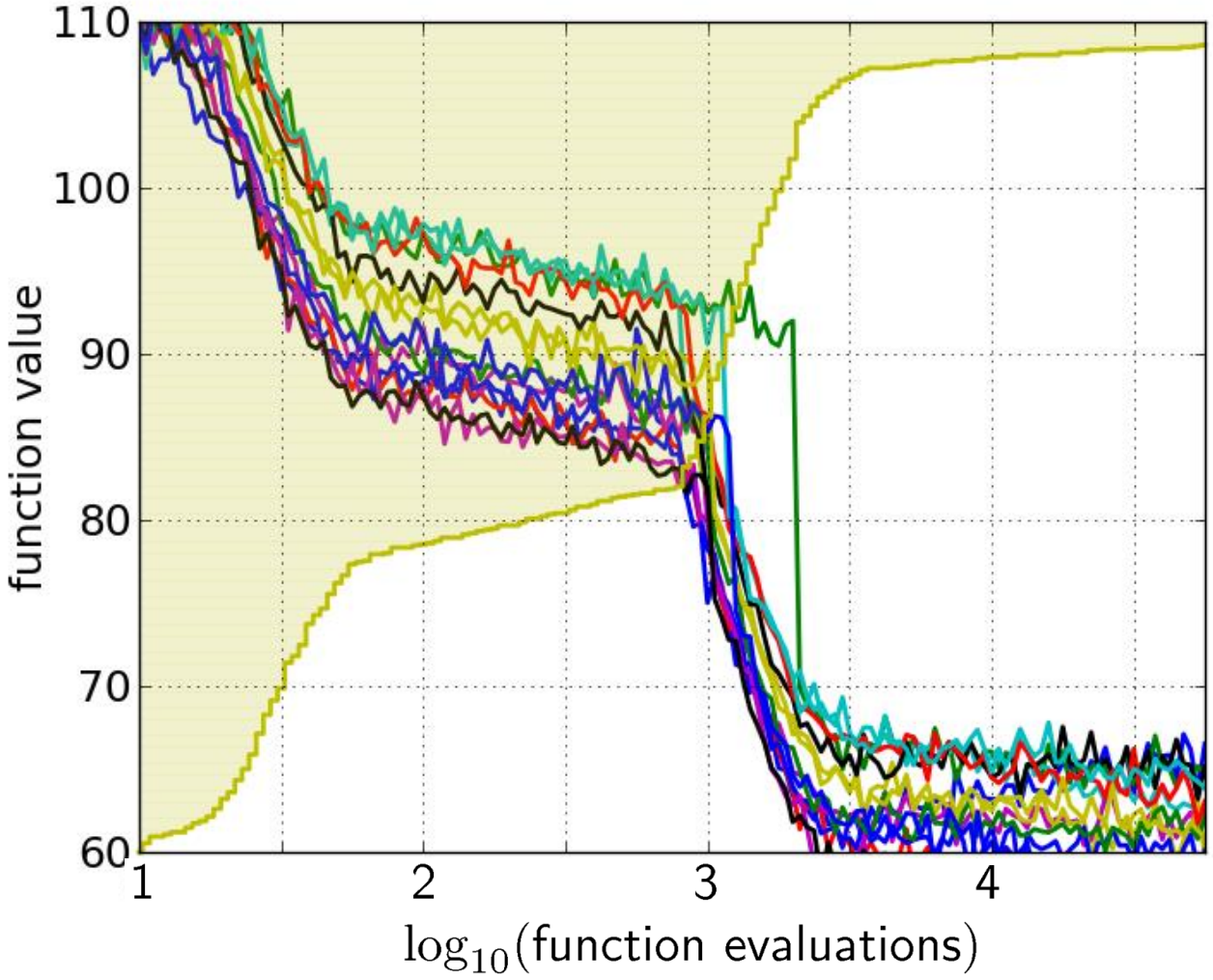
Aggregation



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

phew ;-)

http://coco.gforge.inria.fr/

start [COmparing Continu...

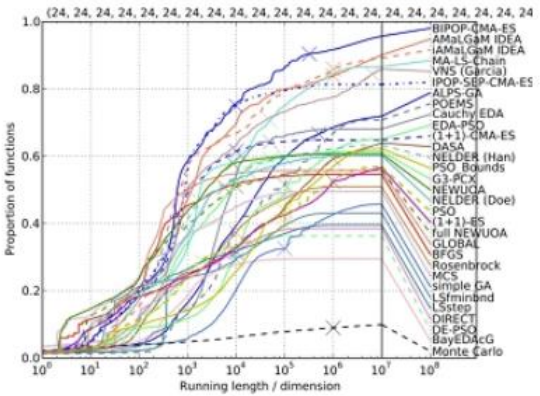
coco.gforge.inria.fr

Most Visited Getting Started algorithms [COmparin... numbbo/numbbo · Gi...

[[start]] COMPARING CONTINUOUS OPTIMISERS: COCO

Show pagesource Old revisions Recent changes Sitemap Login

COCO (COmparing Continuous Optimisers) is a platform for systematic and sound comparisons of real-parameter global optimisers. COCO provides benchmark function testbeds, experimentation templates which are easy to parallelize, and tools for processing and visualizing data generated by one or several optimizers. The COCO platform has been used for the Black-Box-Optimization-Benchmarking (BBOB) workshops that took place during the GECCO conference in 2009, 2010, 2012, 2013 and 2015. It was also used at the IEEE Congress on Evolutionary Computation (CEC'2015) in Sendai, Japan. The COCO source code is available at the downloads page.



- Black-Box Optimization Benchmarking (BBOB) 2016
- Black-Box Optimization Benchmarking (BBOB) 2015
- CEC'2015 special session on Black-Box Optimization Benchmarking (CEC-BBOB 2013)
- Black-Box Optimization Benchmarking (BBOB) 2013
- Black-Box Optimization Benchmarking (BBOB) 2012
- Black-Box Optimization Benchmarking (BBOB) 2010
- Black-Box Optimization Benchmarking (BBOB) 2009
- Downloads and documentations

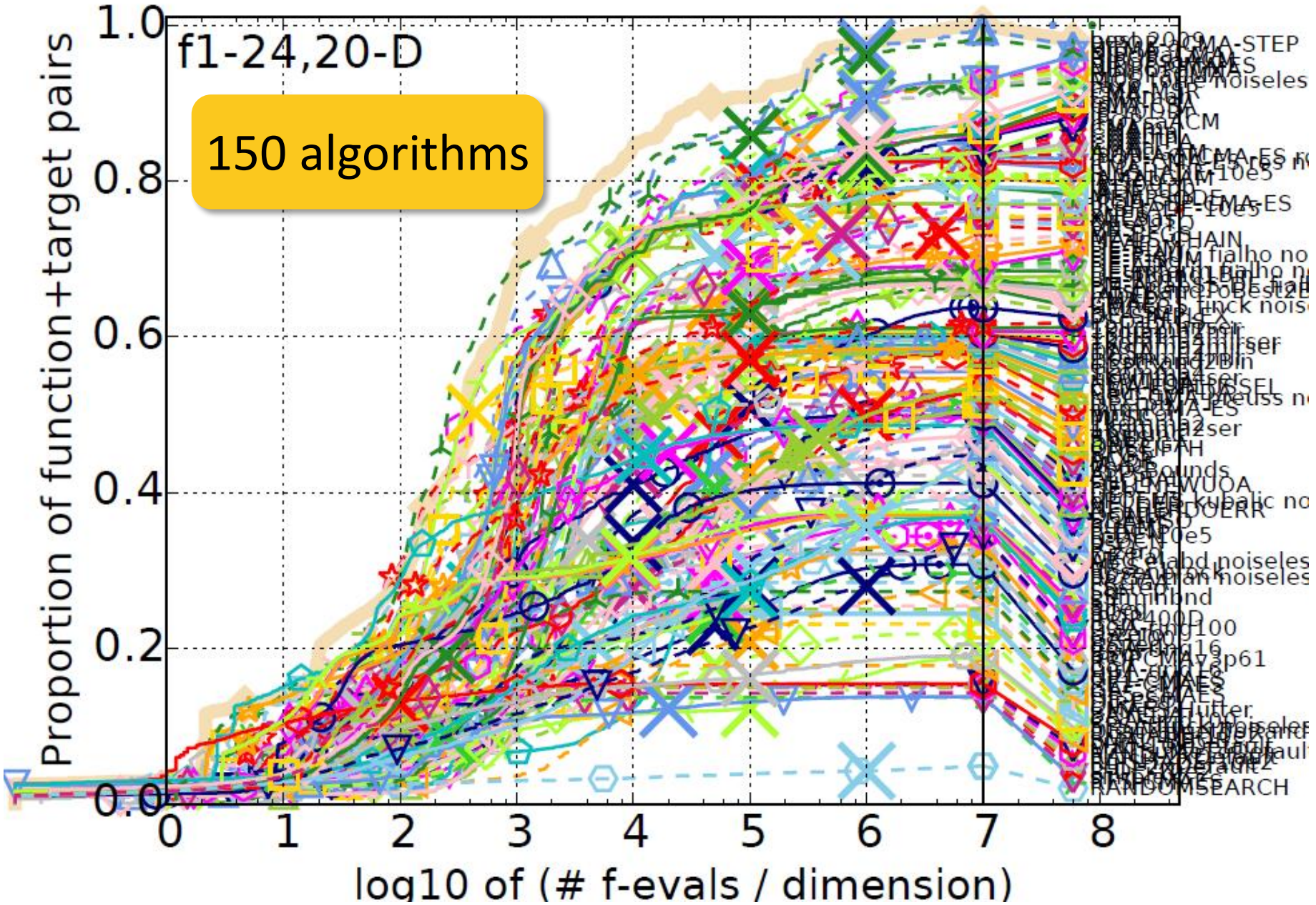
To subscribe to (or unsubscribe from) the bbob discussion mailing list follow this link http://lists.lri.fr/cgi-bin/mailman/listinfo/bbob-discuss .

To receive announcements related to the BBOB workshops simply send an email to BBOB team

Search

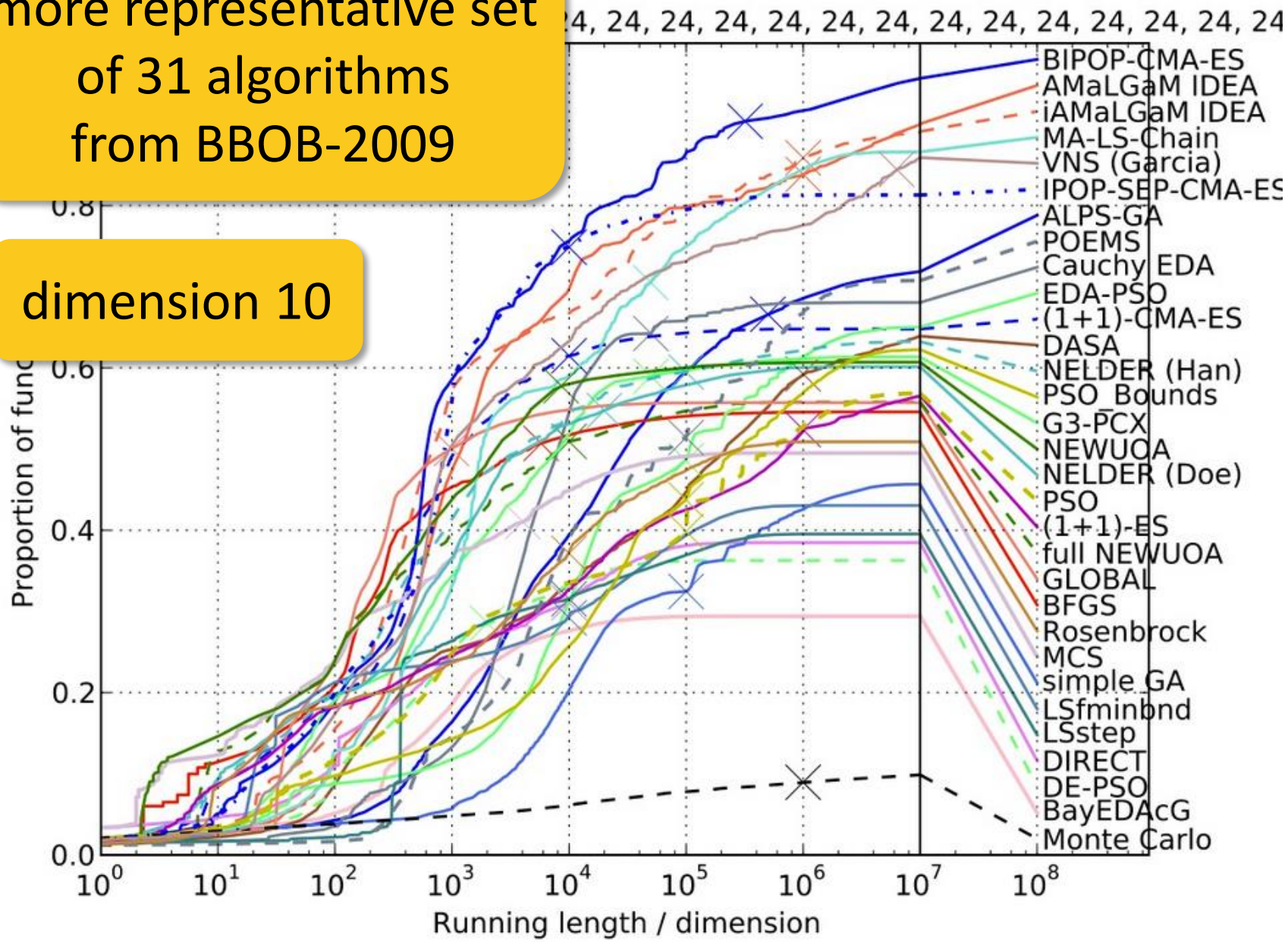
Navigation

- Home
- Documentation
- download latest old code
- new code homepage
- download new code directly
- BBOB 2016
- BBOB 2015 @ GECCO
  - Algorithms
  - Results
  - Schedule
  - Downloads
- BBOB 2015 @ CEC
  - Algorithms
  - Results
  - Downloads
- BBOB 2013
  - Algorithms
  - Results
  - Schedule
  - Downloads
- BBOB 2012
  - Algorithms
  - Results
  - Downloads
- BBOB 2010
  - Results
  - Downloads


f1-24,20-D
150 algorithms
Proportion of function+target pairs
log10 of (# f-evals / dimension)
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5
6
7
8

more representative set of 31 algorithms from BBOB-2009

dimension 10



Recommendations

Results of BBOB 2009-2015 (noiseless bbob test suite)

- low dimension (2-D, 3-D), small budget: Nelder Mead (implementation by B. Doerr et al.)
- very small budget (<50D funevals): BBOB-SMAC (an EGO variant)
- larger dimension, small budget (<500D funevals): NEWUOA, MCS
- larger dimension, large budget (>500D funevals): CMA-ES variants
- portfolio algorithms can combine the best of all worlds, e.g. HCMA

Further Observations

Invariance

- many algorithms are not invariant under affine transformations of the search space (for example PSO), others like CMA-ES variants are
- as a consequence, other types of algorithms are good for separable and non-separable problems

Robustness of Algorithms

- CMA-ES was tested in several variants, in several programming languages and with several, slightly changing settings
- nevertheless, it showed throughout its variants consistently good behavior and can be considered as a robust algorithm if the budget is not too low

The Future of COCO

- bi-objective data coming up (BBOB workshop@GECCO'16)
- towards more realistic problems
  - constraints
  - "almost real-world" problems
- online visualization of data

Thank you.

GECCO workshop on

Black Box Optimization Benchmarking (BBOB'2016) with a focus on bi-objective problems

organizers:

Anne Auger, Dimo Brockhoff, Nikolaus Hansen,

Tea Tušar, Dejan Tušar, Tobias Wagner

submission deadline: **April 17, 2016**

http://numbbo.github.io/workshops/BBOB-2016/

https://github.com/numbbo/coco