Derivative-Free Optimization
Benchmarking and Performance Assessment

December 13, 2024

M2 Optimization, Université Paris-Saclay

Dimo Brockhoff

Inria Saclay – Ile-de-France

Practical Group Project

What do we want to achieve?

- understand some (new or not so new) algorithms
- and how they behave subject to *noise/outliers*
- learn how to conduct scientific testing, how to assess performance, and how to draw scientific conclusions from benchmarking data

How does it work?

- Today:
  - introduction to benchmarking
  - installation of COCO on your machines
  - first exercise to interpret (existing) benchmarking data
- In groups of 3
  - please use this to organize yourself: http://tinyurl.com/cocoproject2024

Practical Group Project

- We propose some possible algorithms (max. 1 group/algo)
  - but you can take others (e.g. your own)
  - benchmark this algorithm with the help of COCO on noisy functions
- 24th of January:
  - report (PDF) sent by email, 8 pages (based on COCO LaTeX template?)
- 31st of January:
  - 12 minutes presentation per group
  - + 10 minutes questions
  - about the algorithm
  - about its performance (compared to other algos)
  - order of the talks to be decided early next year
- 10th of January:
  - send a short progress statement by email (<10 lines)

The Group Project's Context

A real science project

- Teaching how to do research
- Results will be new
- "Expect the unexpected"
- Anne and I are open for discussions along the project

We are very curious about your results

Scientific Context: Noisy optimization / outliers

- How is the performance of algorithms "perturbed" if the objective function gets "perturbed"?
- We provide a wrapper around deterministic test functions (from the bbob suite of the COCO platform) and an experiment script to do the benchmarking for varying levels of noise/outlier probability

Outliers + Noise Level

The noise model is the following:

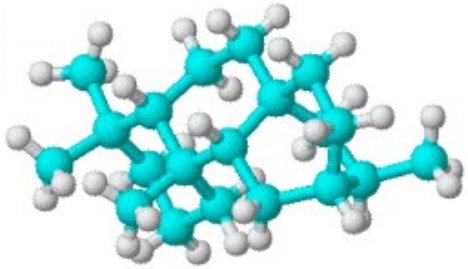
- deterministic noise ("outliers"):
  - if re-evaluated, a search point x will always result in the same $f(x)$
  - the points x which are "outliers" are fixed as well
- noise-level $\in \{0, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4\}$ (probability that a search point x is not the underlying objective function)
- if a point x is "noisy" or an "outlier", we add or substract a value to the original objective function
  - x becomes worse more often than it becomes better
  - the difference between observed and original function value is Cauchy distributed
  - more details can be found in the code

Excerpt from the Code

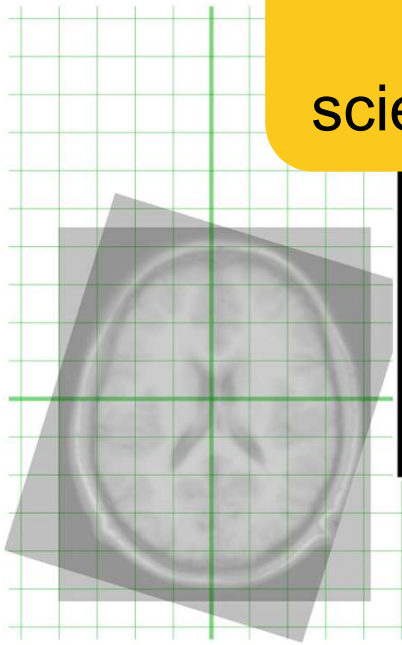
```
for added_noise in [0, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4]:
    output_folder = […]  # folder name must start with a number for cocopp
    observer = cocoex.Observer(suite_name, "result_folder: " +    =
                               output_folder)
    […]

    ### go
    while not repeater.done():  # while budget is left and successes are few
        for problem in suite:  # loop takes 2-3 minutes x budget_multiplier
            problem.observe_with(observer)  # generate data for cocopp
            problem = noiser.NoisifyProblem(problem, p_add=added_noise)
            problem(problem.dimension * [0])  # for better comparability
            ### input: implement/amend the next few lines for another fmin
            if fmin == scipy.optimize.fmin:
                res = fmin(problem,
                           repeater.initial_solution_proposal(problem),
                           disp=False, full_output=True)
                xopt = res[0]
            problem(xopt)  # make sure the returned solution is evaluated
```

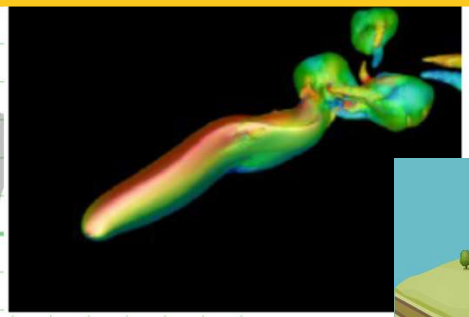
Benchmarking Optimization Algorithms

or: critical performance assessment



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





Practical (Numerical) Blackbox Optimization

Given:

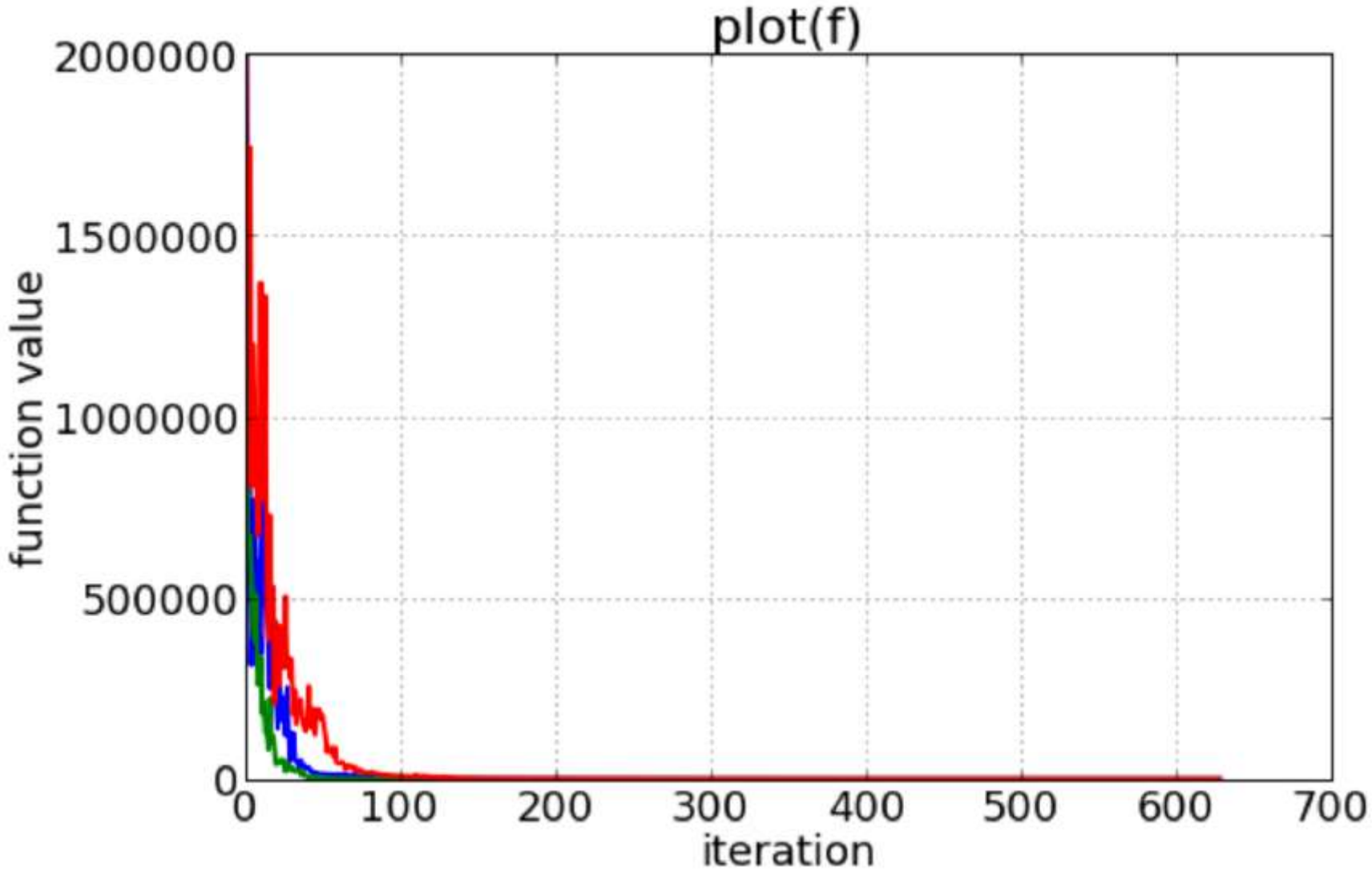
derivatives not available or not useful

Not clear:

which of the many algorithms should I use on my problem?

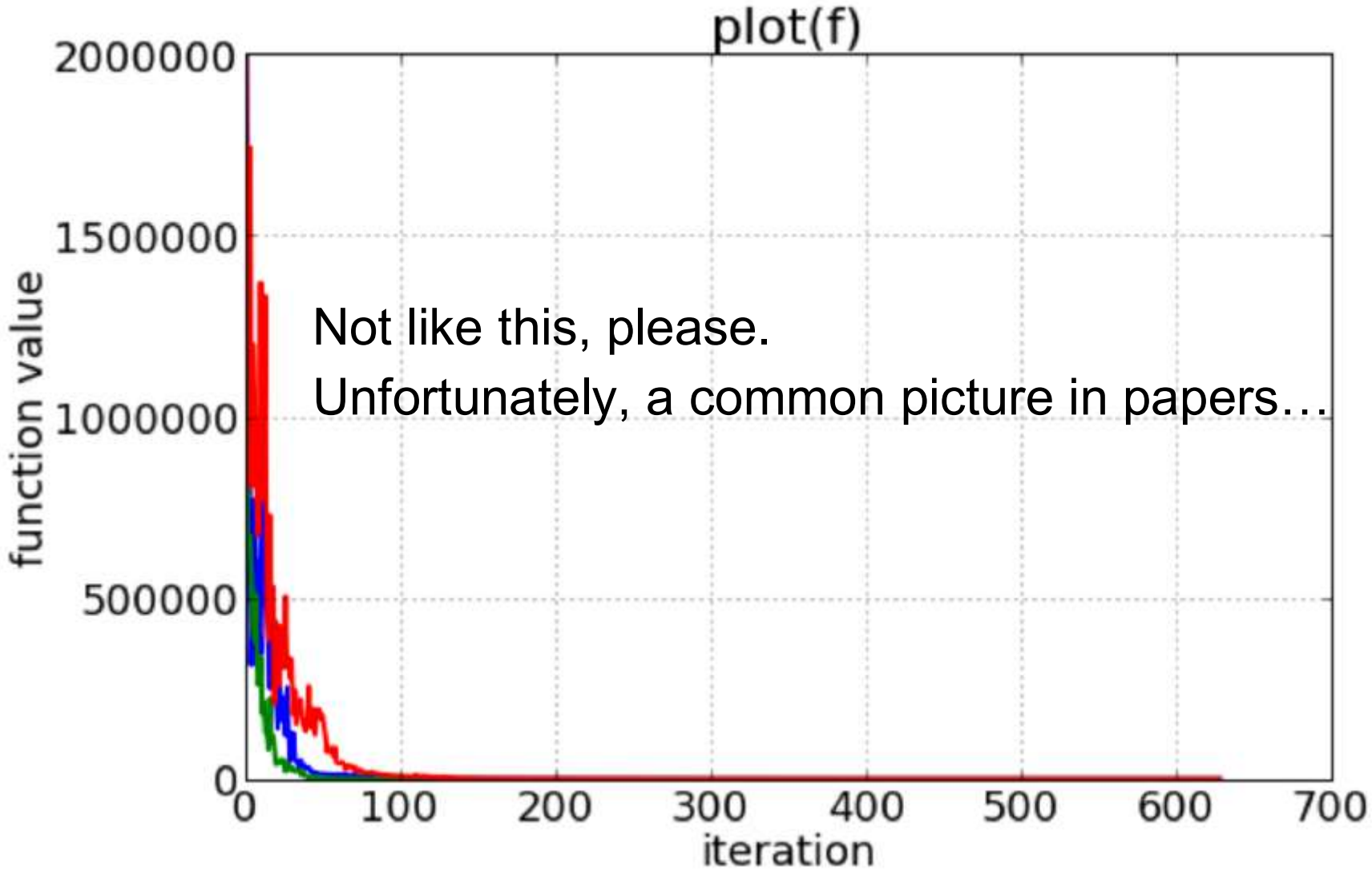
visualizing the raw data (single runs)

Displaying 3 runs (3 trials)



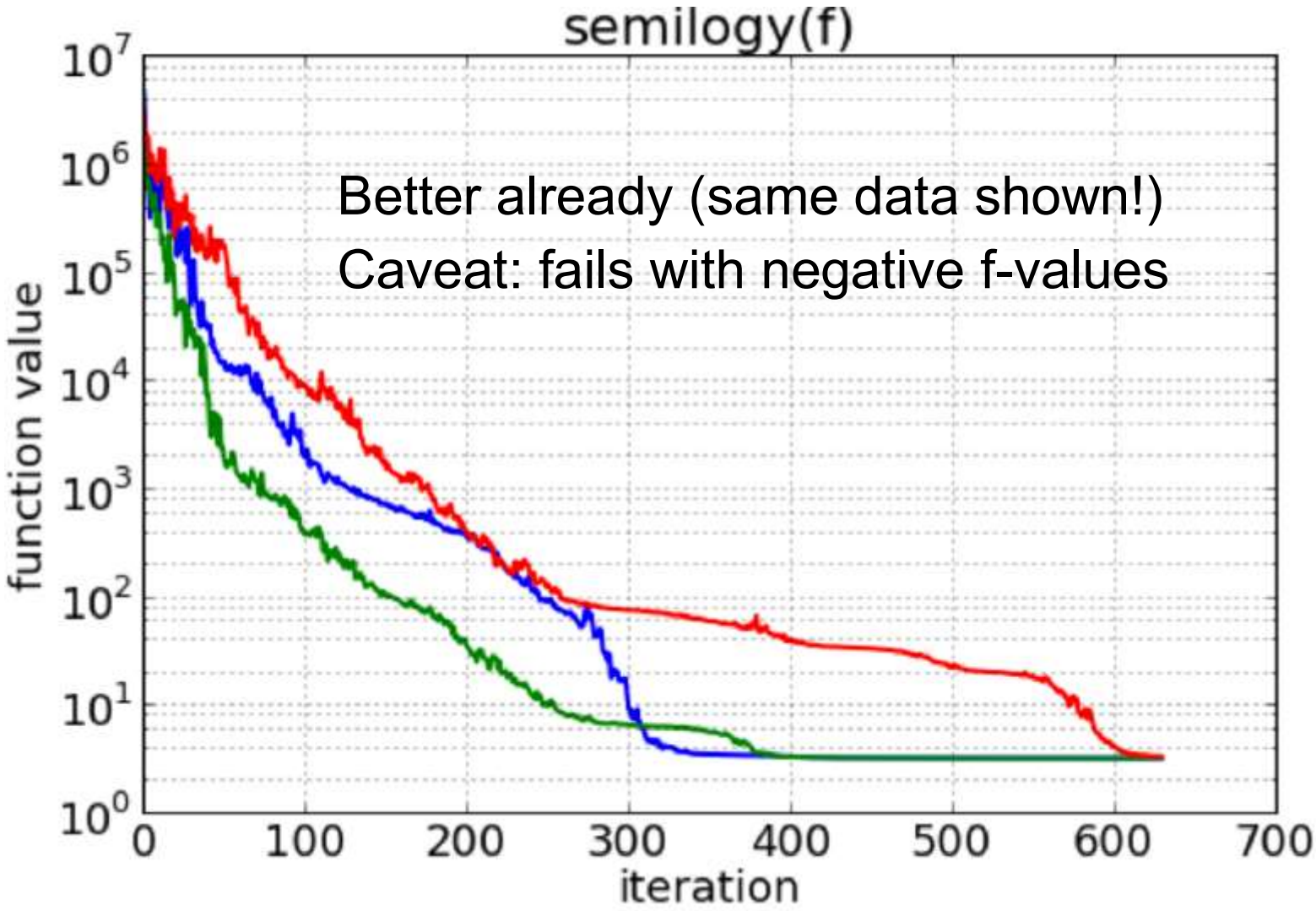
Copyright: A. Auger and N. Hansen

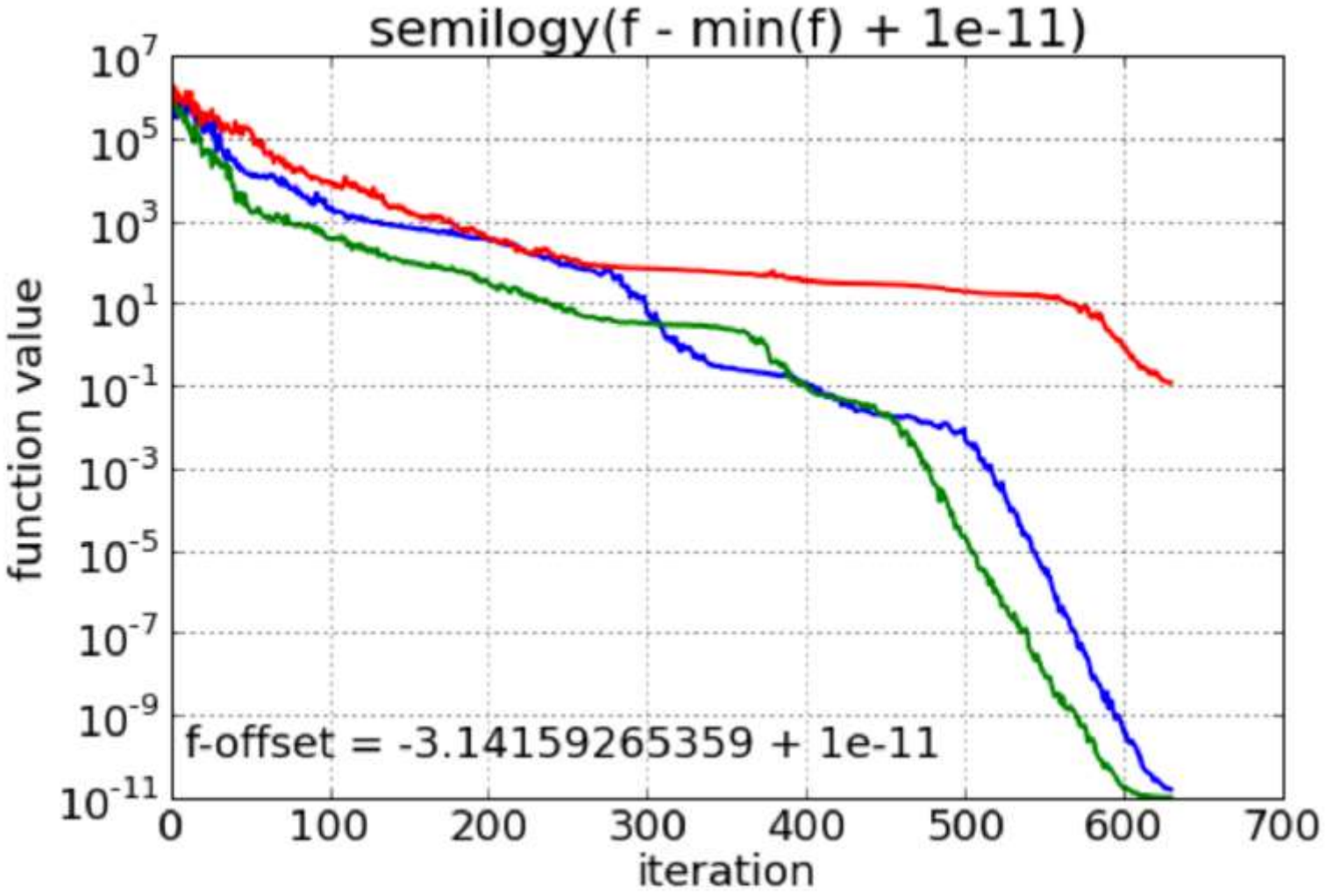
Displaying 3 runs (3 trials)



Copyright: A. Auger and N. Hansen

Displaying 3 runs (3 trials)



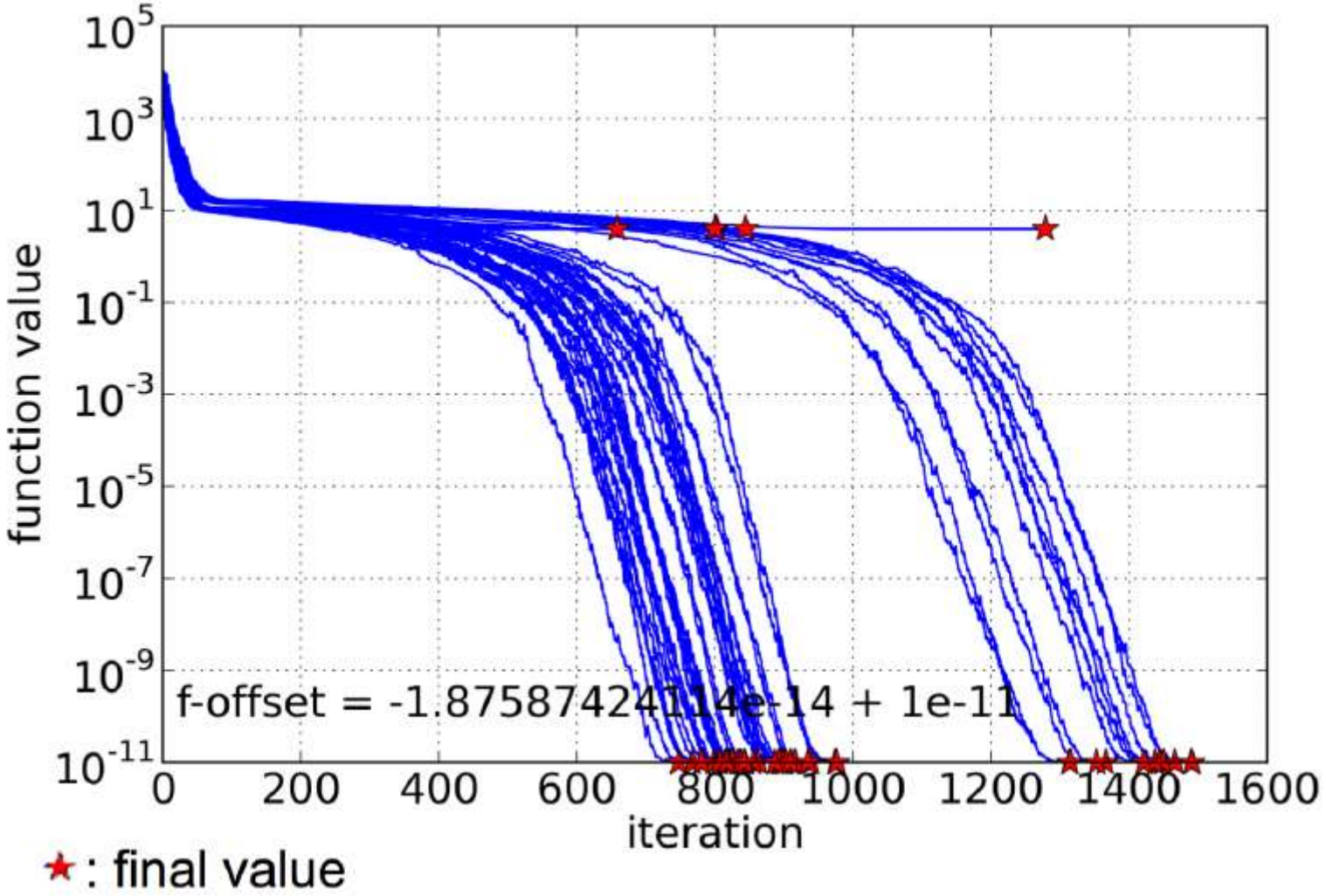


Even better like this: substract minimum over all runs

Copyright: A. Auger and N. Hansen

Displaying 51 runs

Don’t hesitate to display all data (the appendix is your friend)

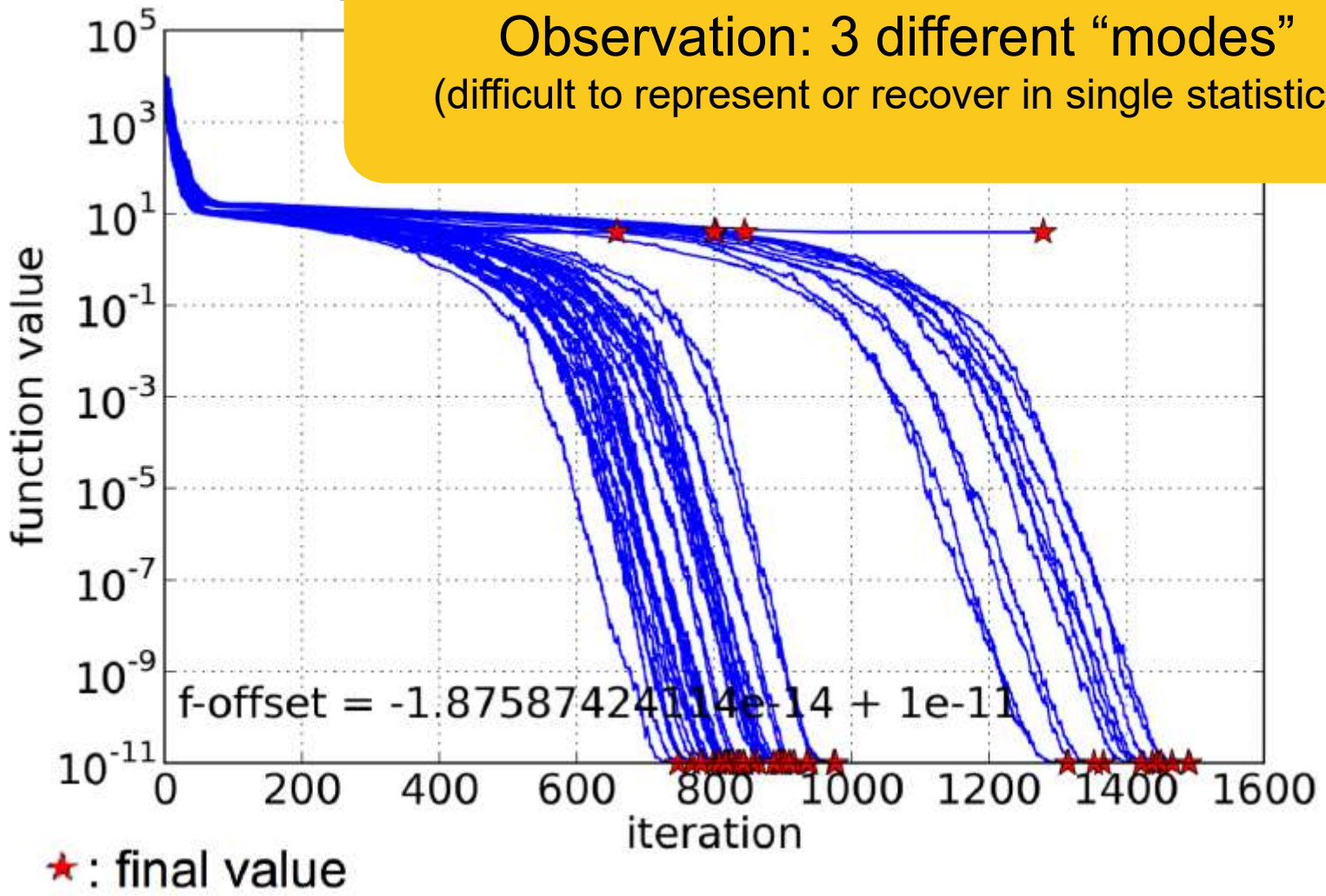


Copyright: A. Auger and N. Hansen

Displaying 51 runs

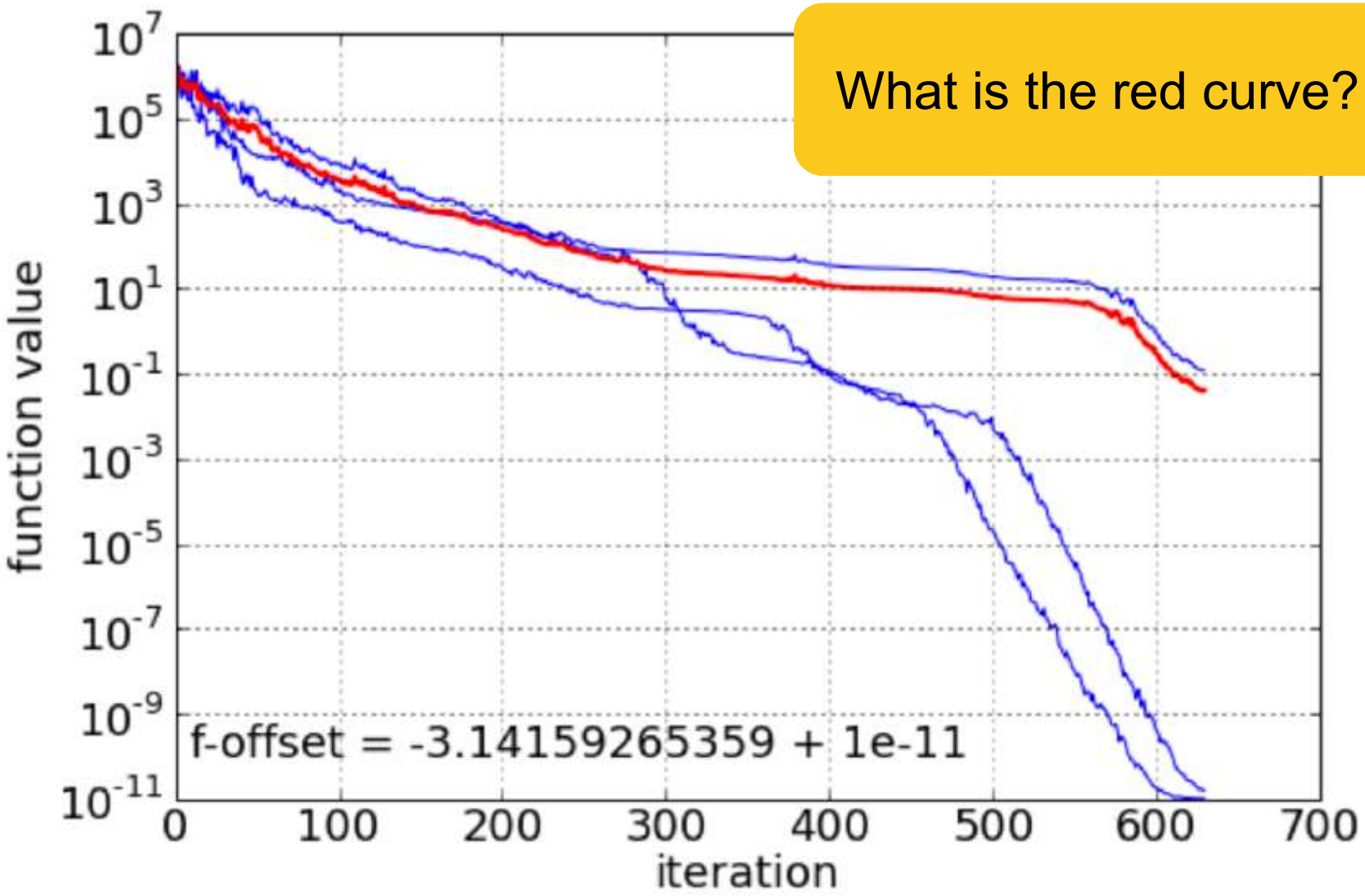
Don’t hesitate to disp

Observation: 3 different “modes”
(difficult to represent or recover in single statistics)

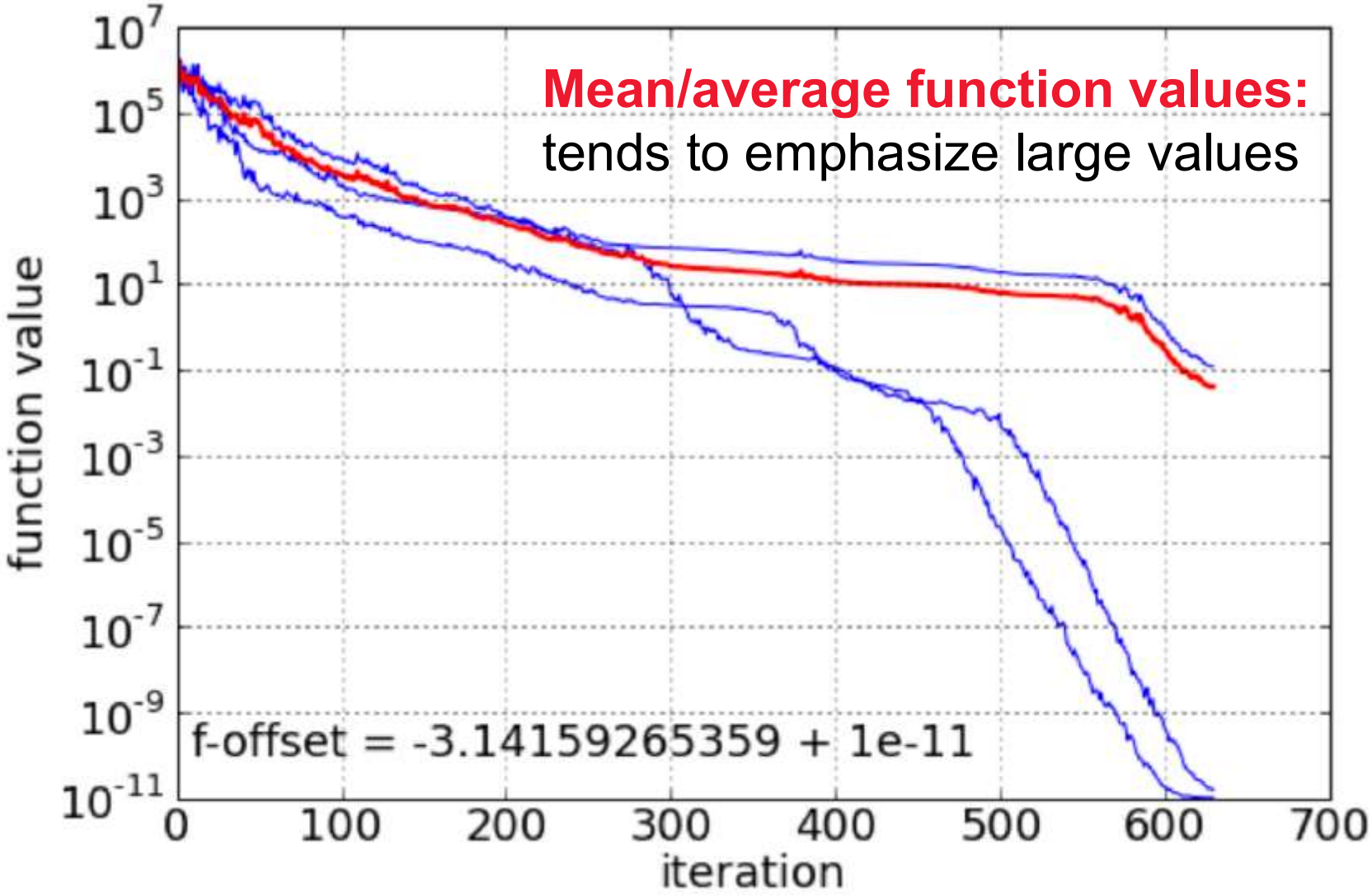


Copyright: A. Auger and N. Hansen

Which Statistics?



Which Statistics?

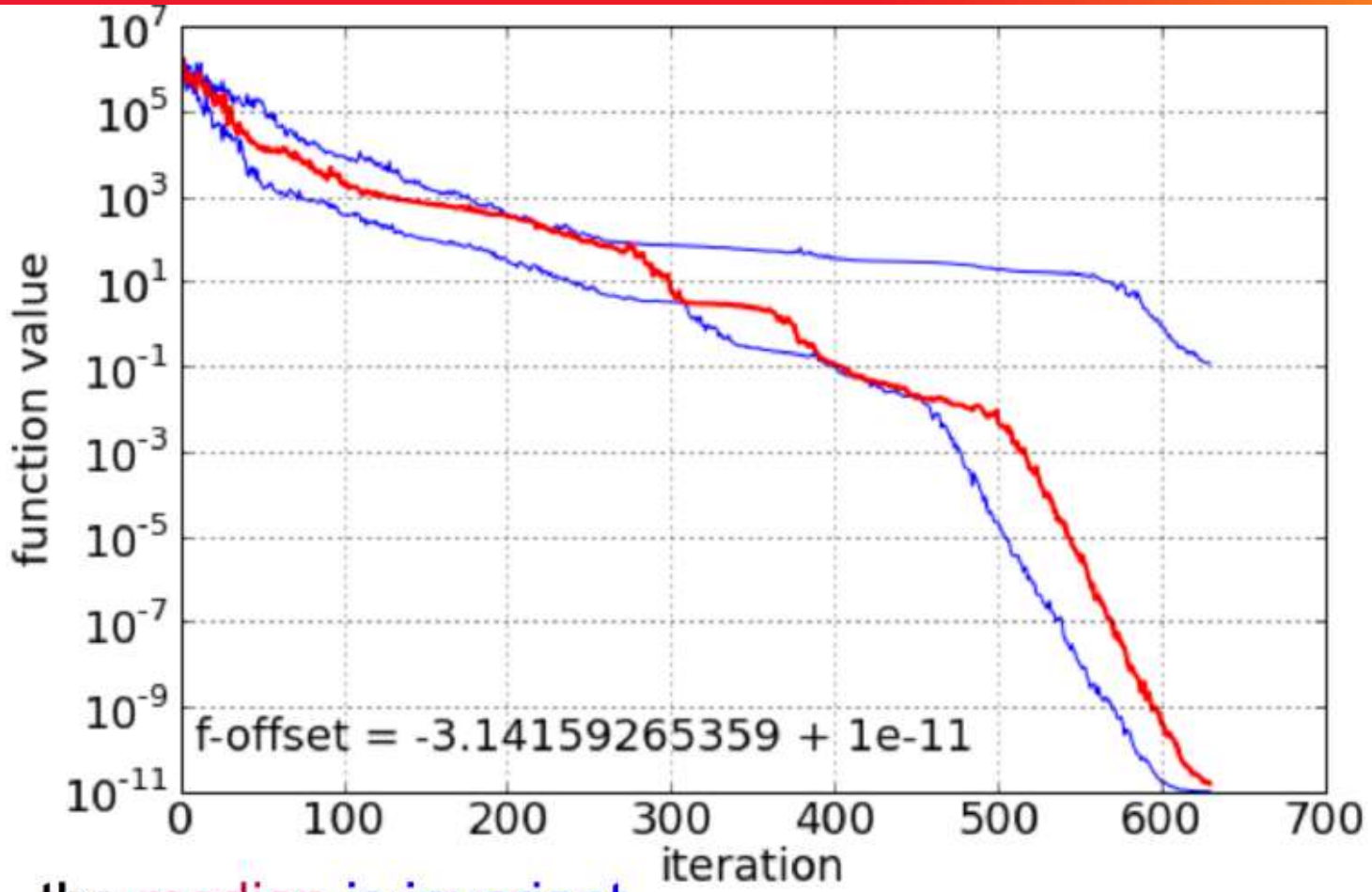


More Problems with Averages/Expectations

- to reliably estimate an expectation (from the *average*) we need to make *assumptions on the tail* of the underlying distribution
  - these can not be implied from the observed data
  - AKA: the average is well-known to be (highly) sensitive to outliers (extreme events)
- rare events can only be analyzed by collecting a *large enough number* of data

from Hansen GECCO 2019 Experimentation tutorial

Which Statistics?



the median is invariant

- unique for uneven number of data
- independent of log-scale, offset...

median(log(data))=log(median(data))

- same when taken over x- or y-direction

Implications

- use the median as summary datum

  unless there are good reasons for a different statistics
  out of practicality: use an odd number of repetitions

- more general: use quantiles as summary data

  for example out of 15 data: 2nd, 8th, and 14th
  value represent the 10%, 50%, and 90%-tile

from Hansen GECCO 2019 Experimentation tutorial

Benchmarking =

evaluate the performance of optimization algorithms

compare the performance of different algorithms

understand strengths and weaknesses of algorithms

help in design of new algorithms

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

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

CPU timing not a good candidate

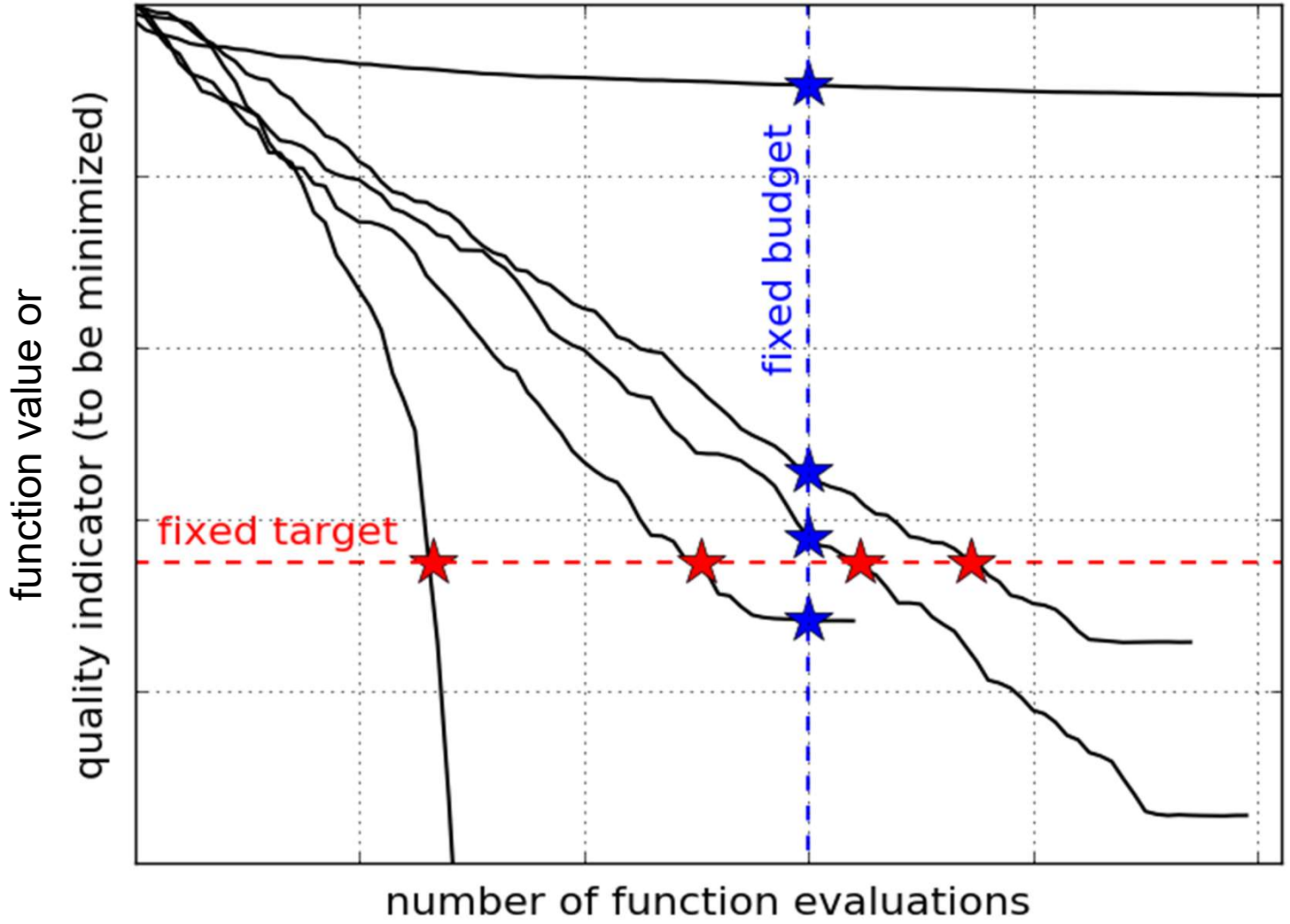
→depends on implementation/language/machine/…

time is spent on code optimization instead of science

J.N. Hooker: Testing heuristics, we have it all wrong, 1995, J. of Heuristics

Measuring Performance Empirically

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



How Do We Measure Performance?

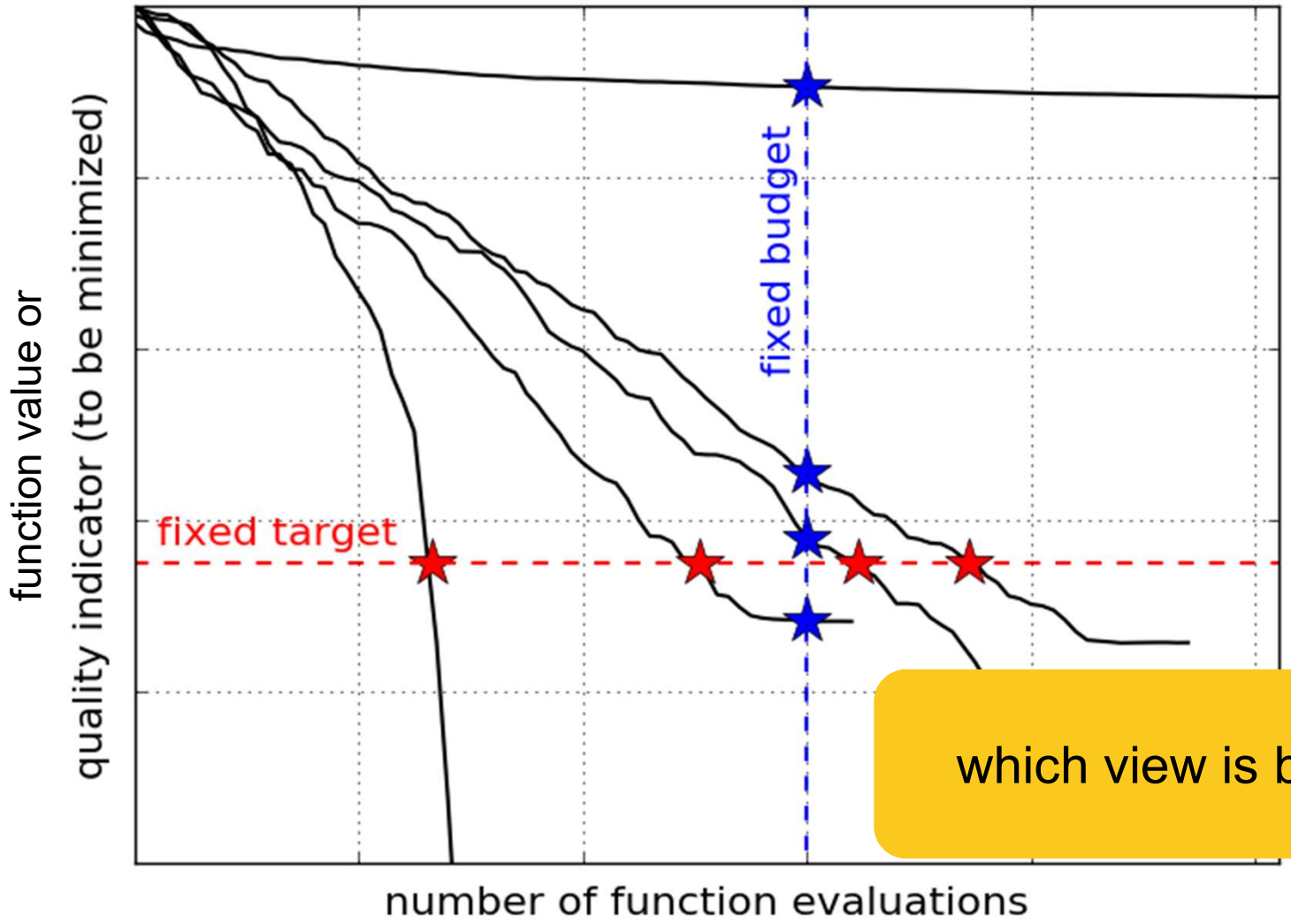
Two objectives:

- 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...



which view is better?

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

Collect for a given target (several target), the number of function evaluations needed to reach a target

Repeat several times:

if algorithms are stochastic, never draw a conclusion from a single run

if deterministic algorithm, repeat by changing (randomly) the initial conditions

ECDF:

Empirical Cumulative Distribution Function of the Runtime

[aka data profile]

Cumulative Distribution Function (CDF)

Given a random variable T, the cumulative distribution function (CDF) is defined as

$$\mathrm{CDF}_T(t) = Pr(T \leq t) \text{ for all } t \in \mathbb{R}$$

It characterizes the probability distribution of T

If two random variables have the same CDF, they have the same probability distribution

Copyright: A. Auger and N. Hansen

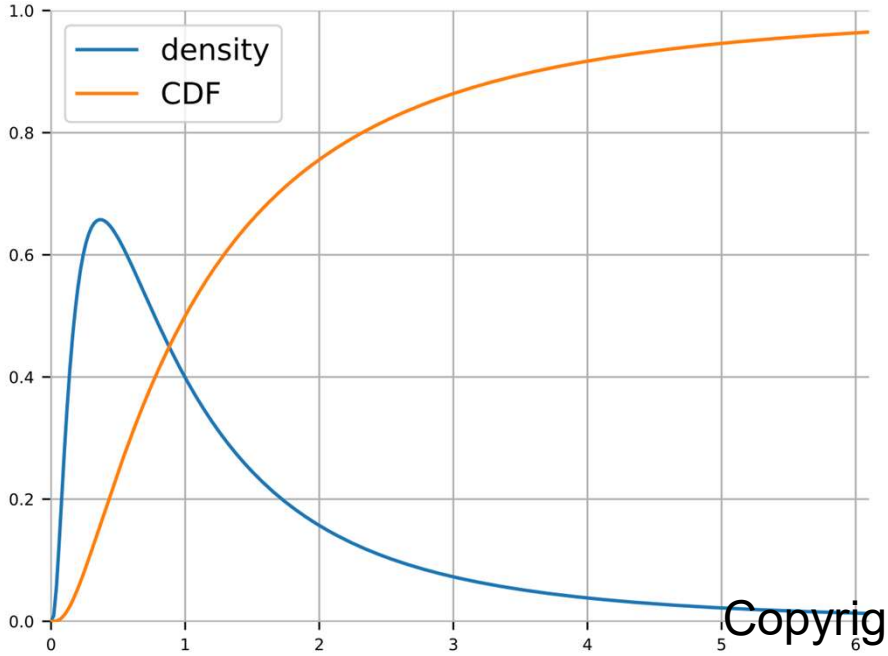
Cumulative Distribution Function (CDF)

Given a random variable T, the cumulative distribution function (CDF) is defined as

$$\mathrm{CDF}_T(t) = Pr(T \leq t) \text{ for all } t \in \mathbb{R}$$

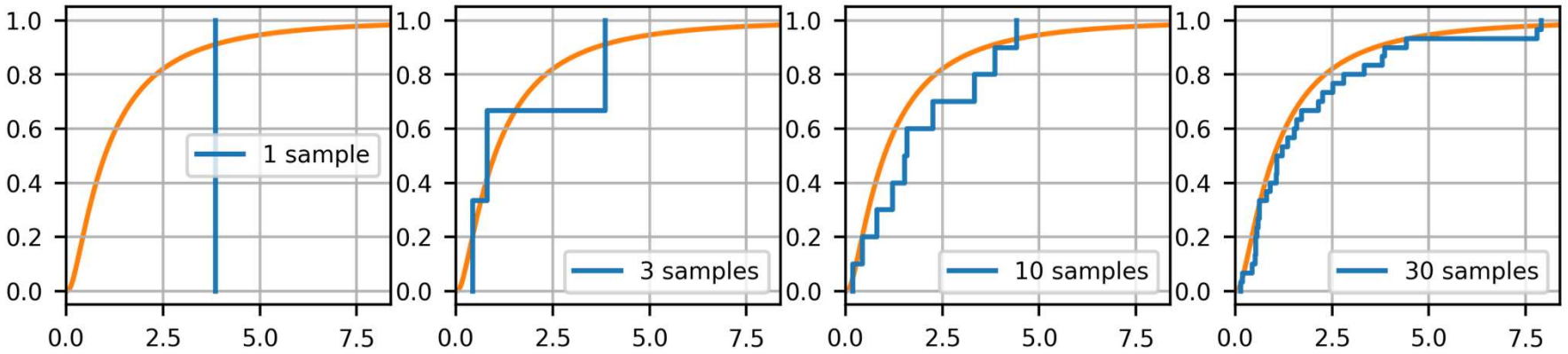
It characterizes the probability distribution of T

If two random variables have the same CDF, they have the same probability distribution



Empirical Cumulative Distribution Function

Given a collection of data $T_1, T_2, \ldots, T_k$ (e.g. an empirical sample of a random variable) the *empirical* cumulative distribution function (ECDF) is a step function that jumps by $1/k$ at each value in the data.



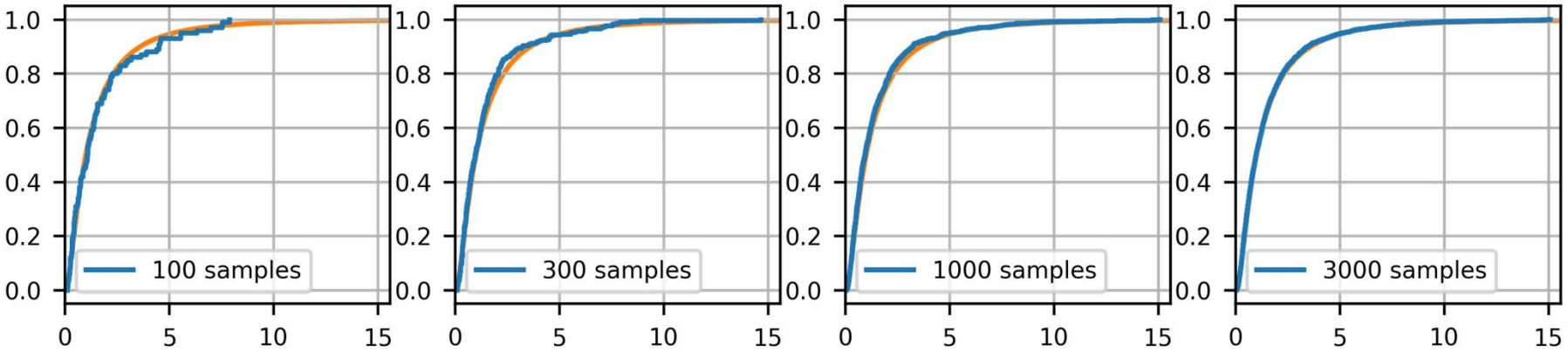
It is an estimate of the CDF that generated the points in the sample.

Empirical Cumulative Distribution Function

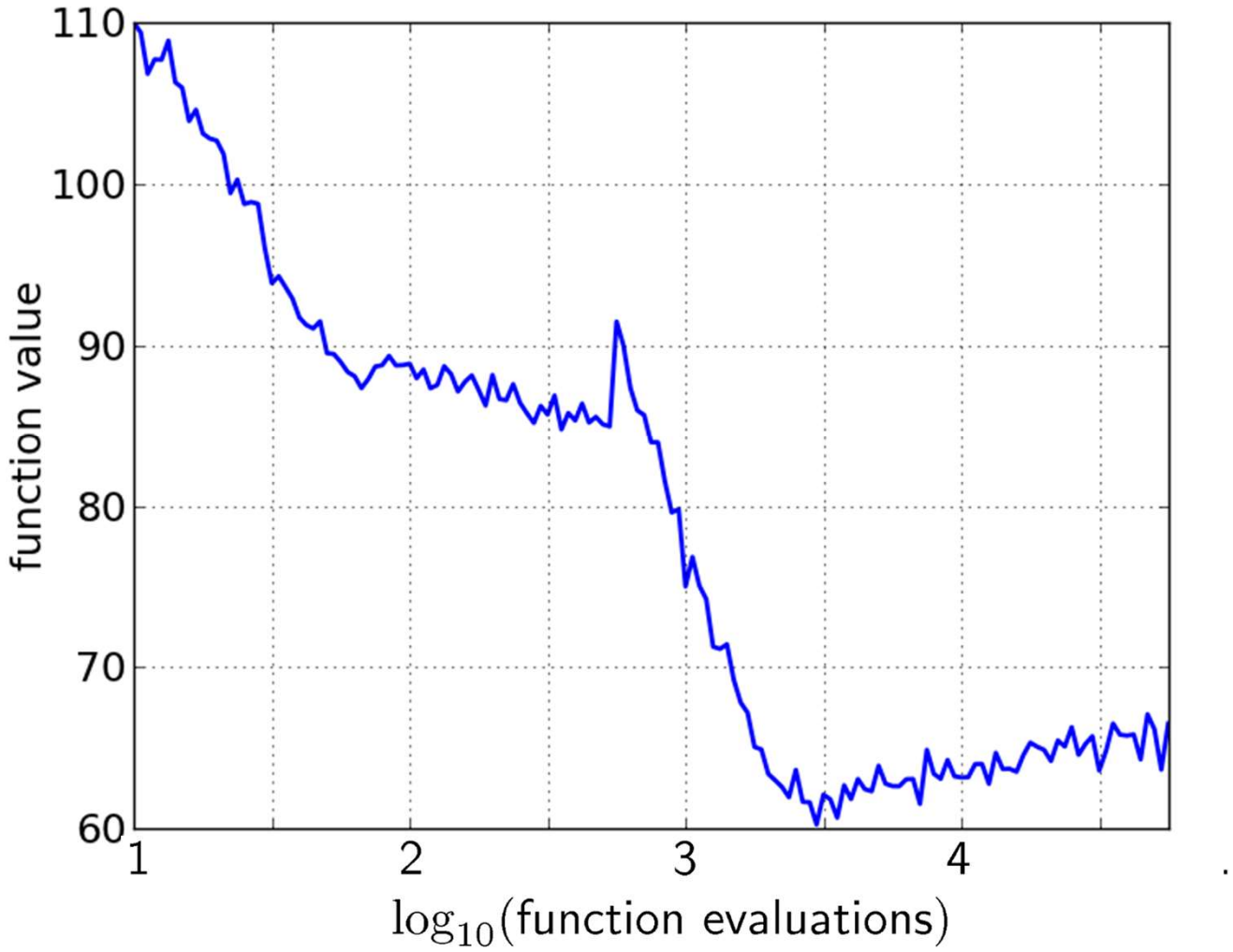
$$\mathrm{ECDF}_{(T_1,\ldots,T_k)}(t) = \frac{\text{number of } T_i \leq t}{k} = \frac{1}{k}\sum_{i=1}^{k} 1_{\{T_i \leq t\}}$$

For $\{T_i : i \geq 1\}$ i.i.d. realization of a random variable T, by the LLN

$$\mathrm{ECDF}_{T_1,\ldots,T_k}(t) \underset{k\to\infty}{\to} \mathrm{CDF}_T(t) \text{ a.s. for all } t$$

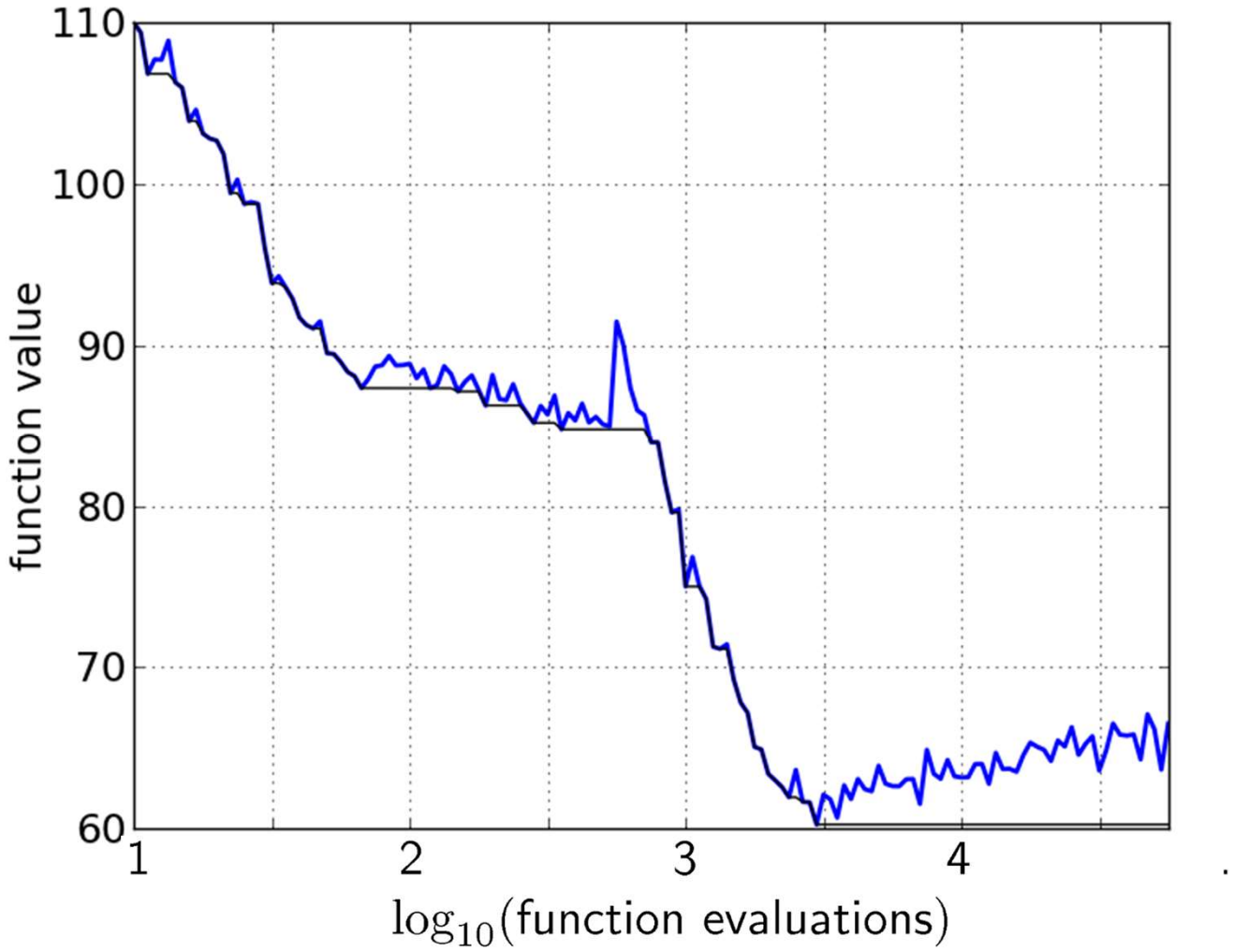


A Convergence Graph



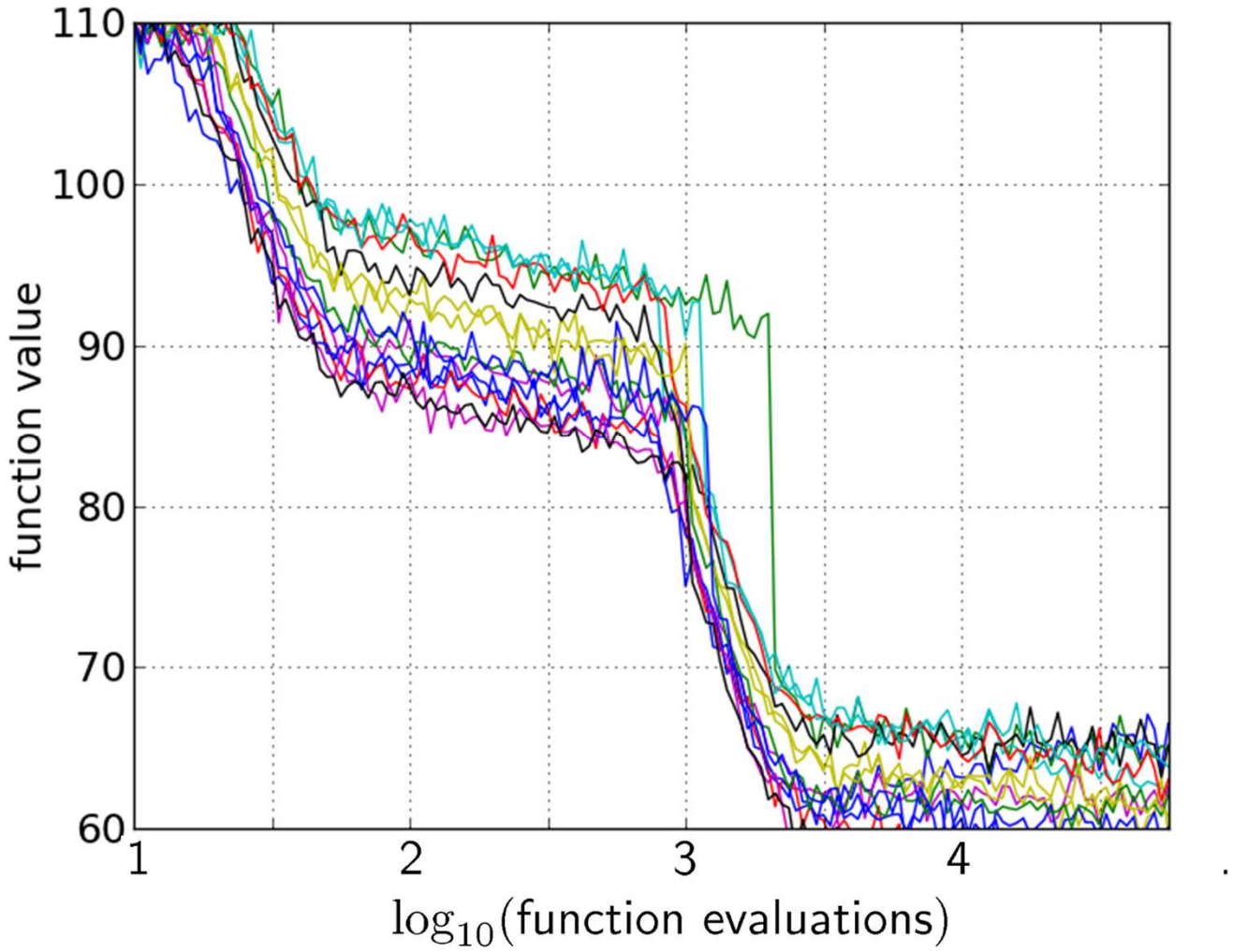
.

First Hitting Time is Monotonous



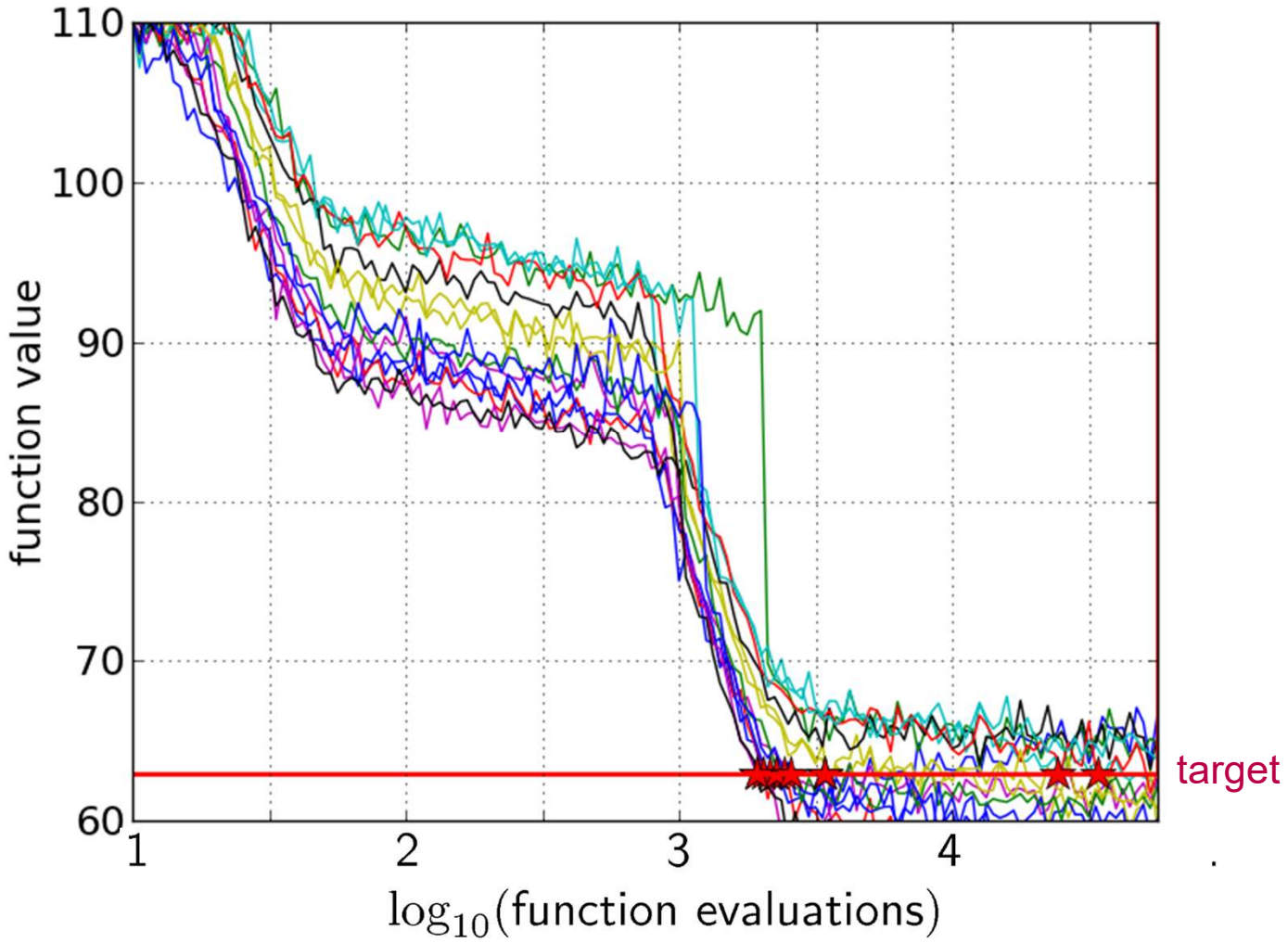
.

15 Runs


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

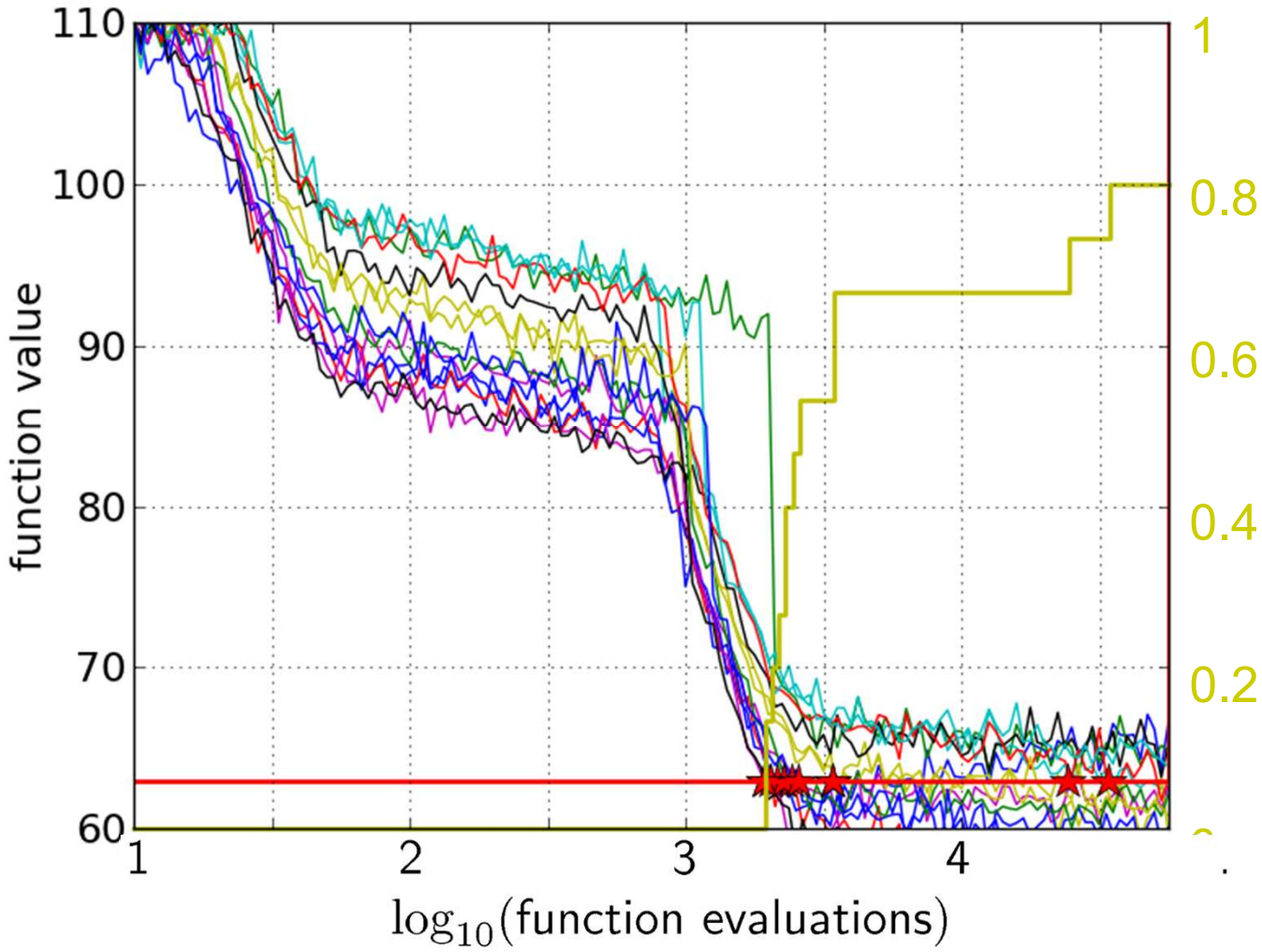
.

15 Runs ≤ 15 Runtime Data Points



.

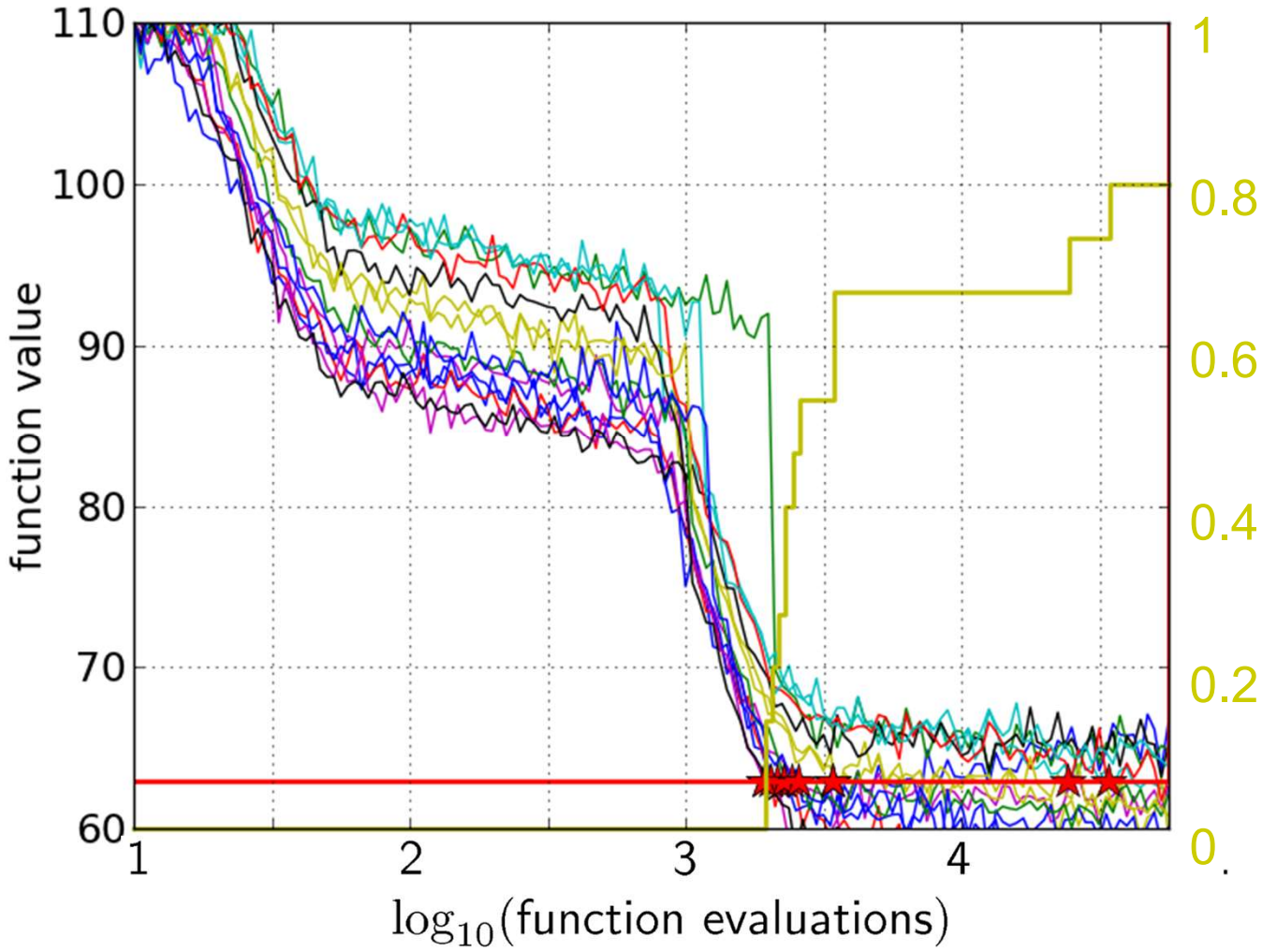
Empirical Cumulative Distribution



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)

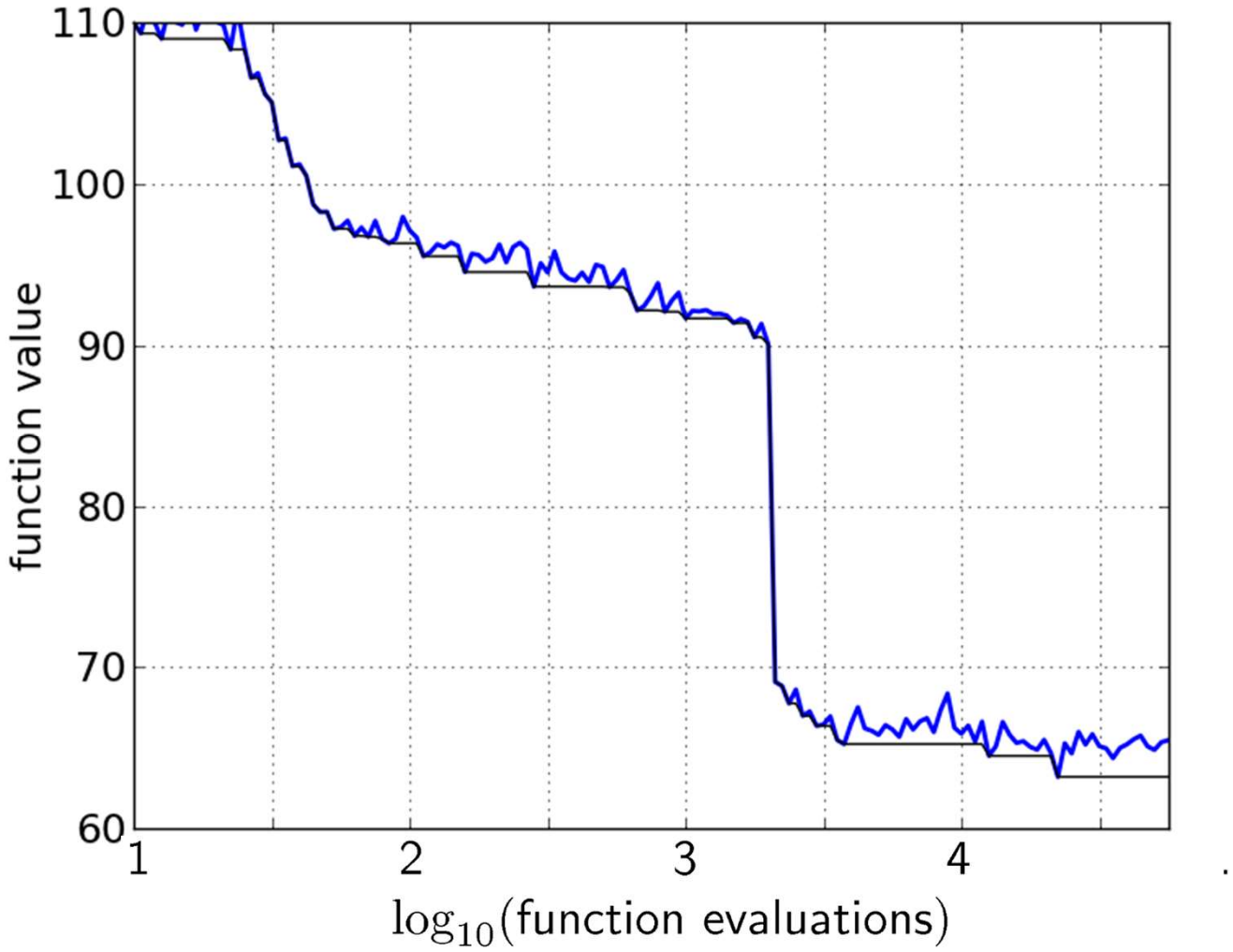
Empirical Cumulative Distribution



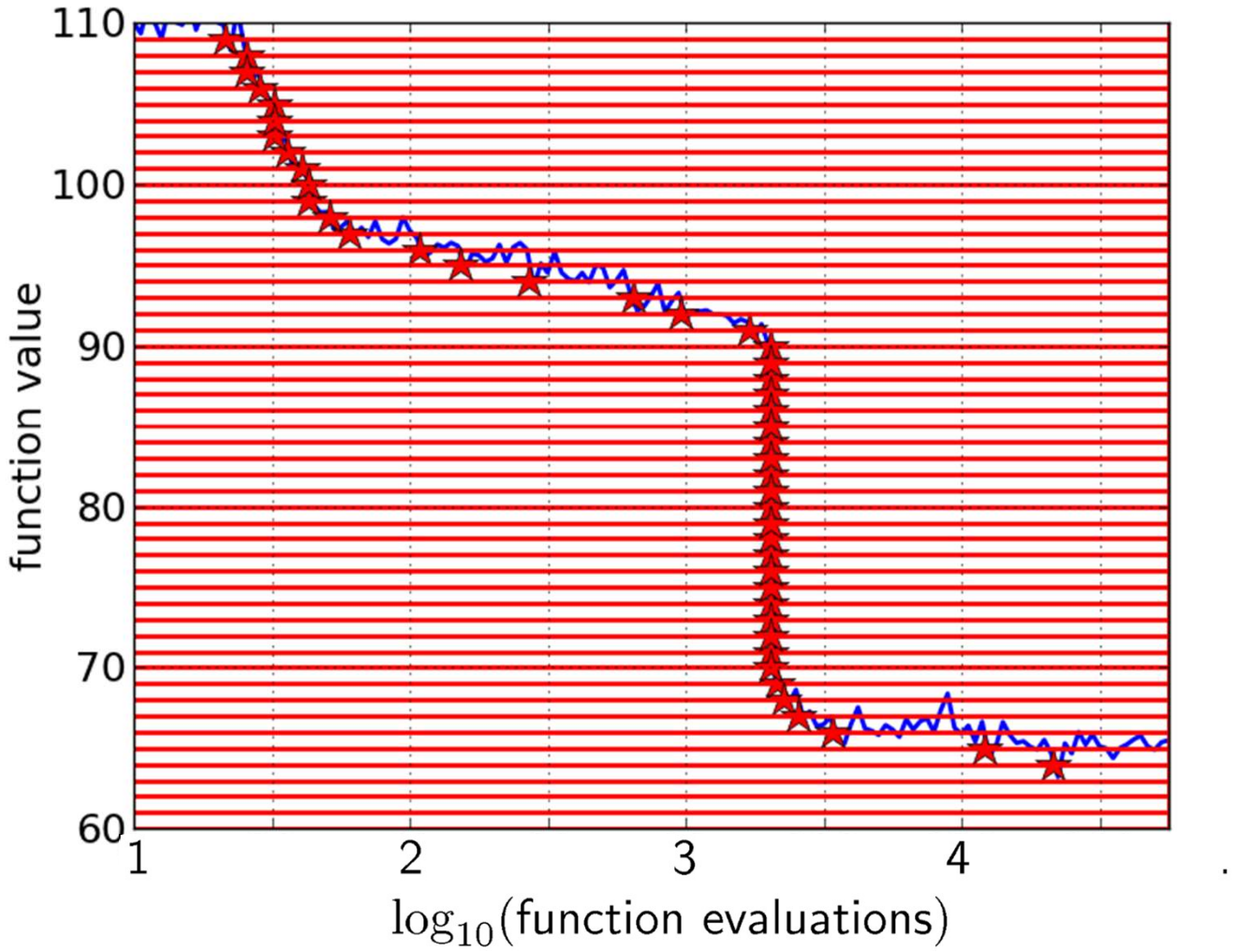
interpretations possible:

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

Reconstructing A Single Run

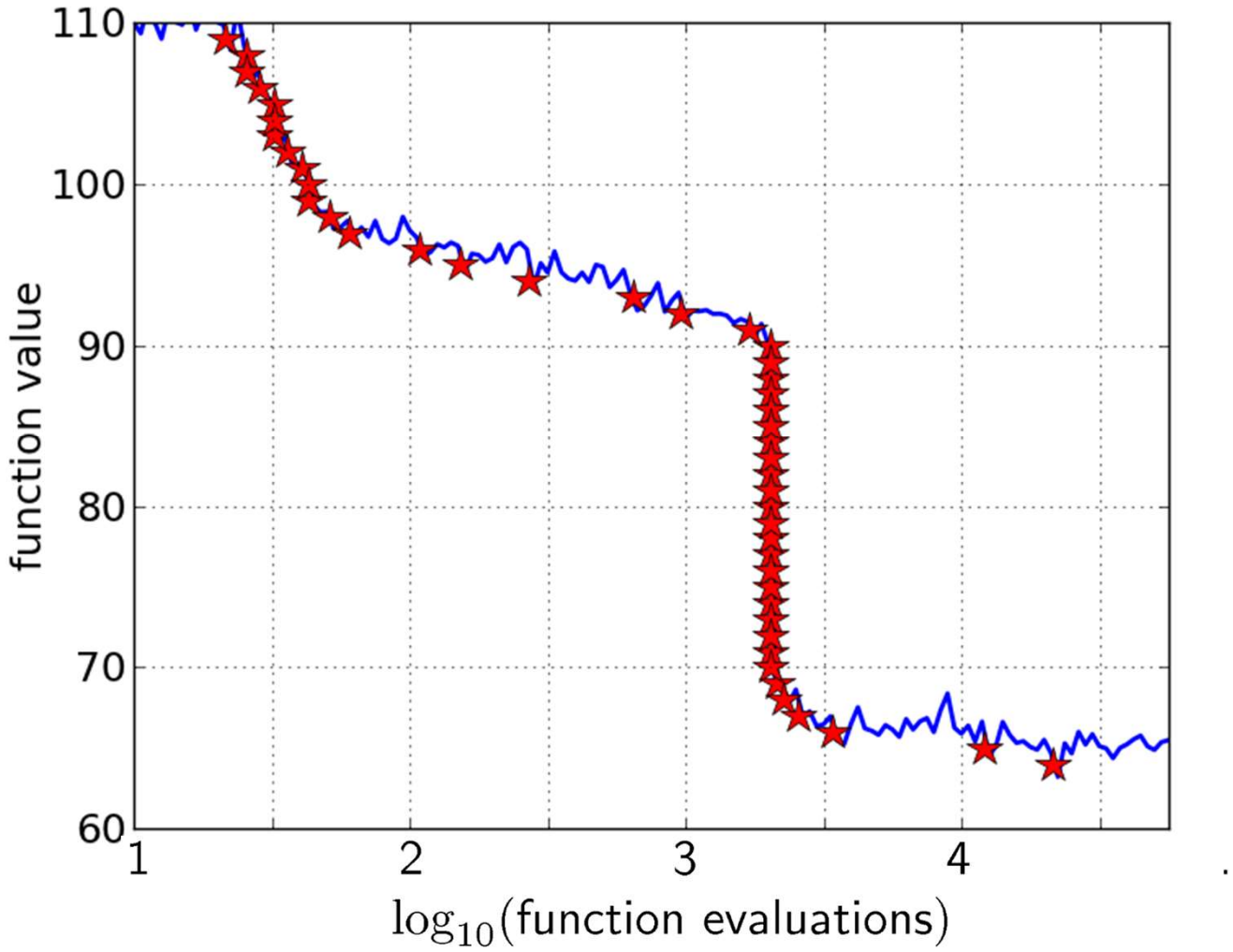


Reconstructing A Single Run

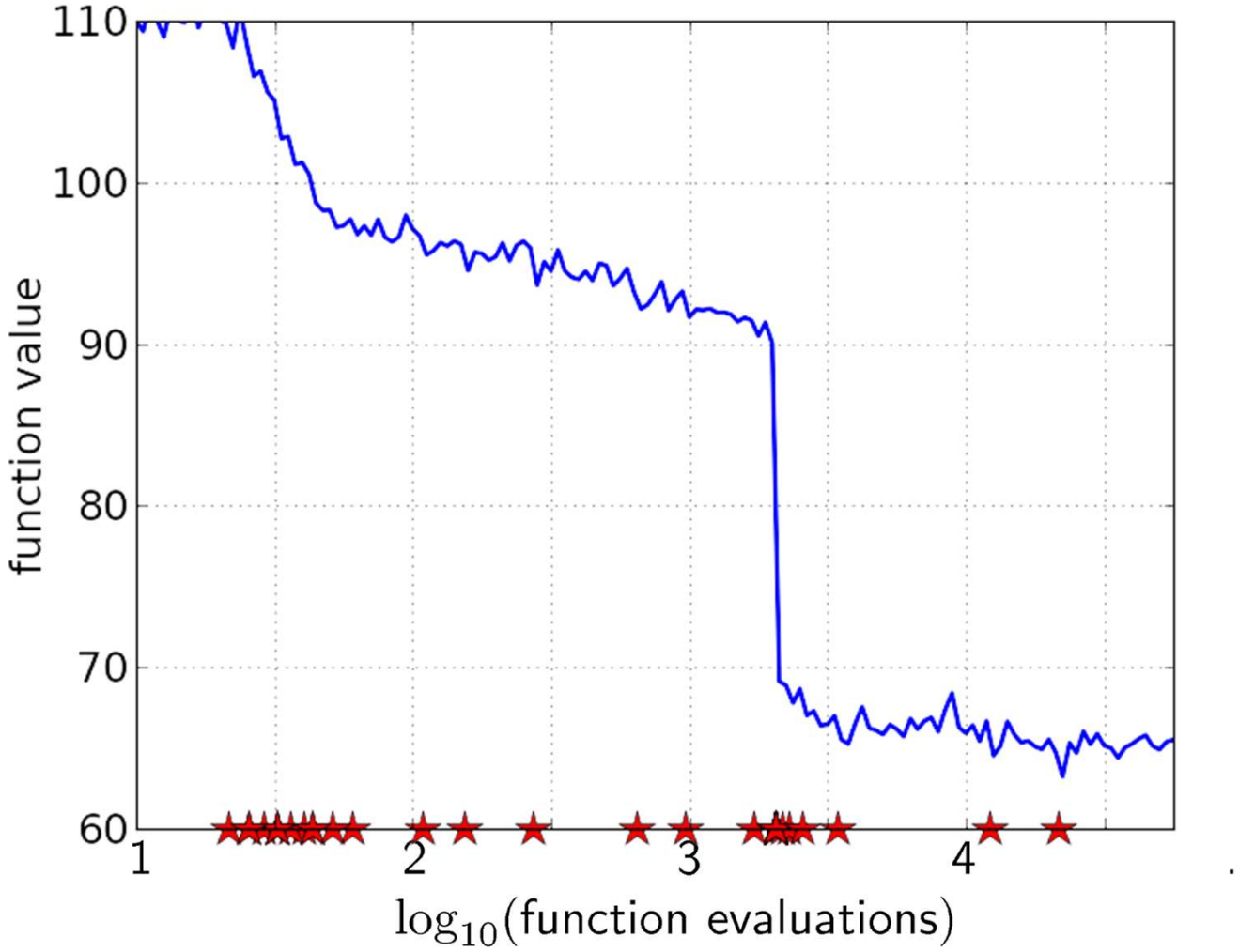


50 equally spaced targets

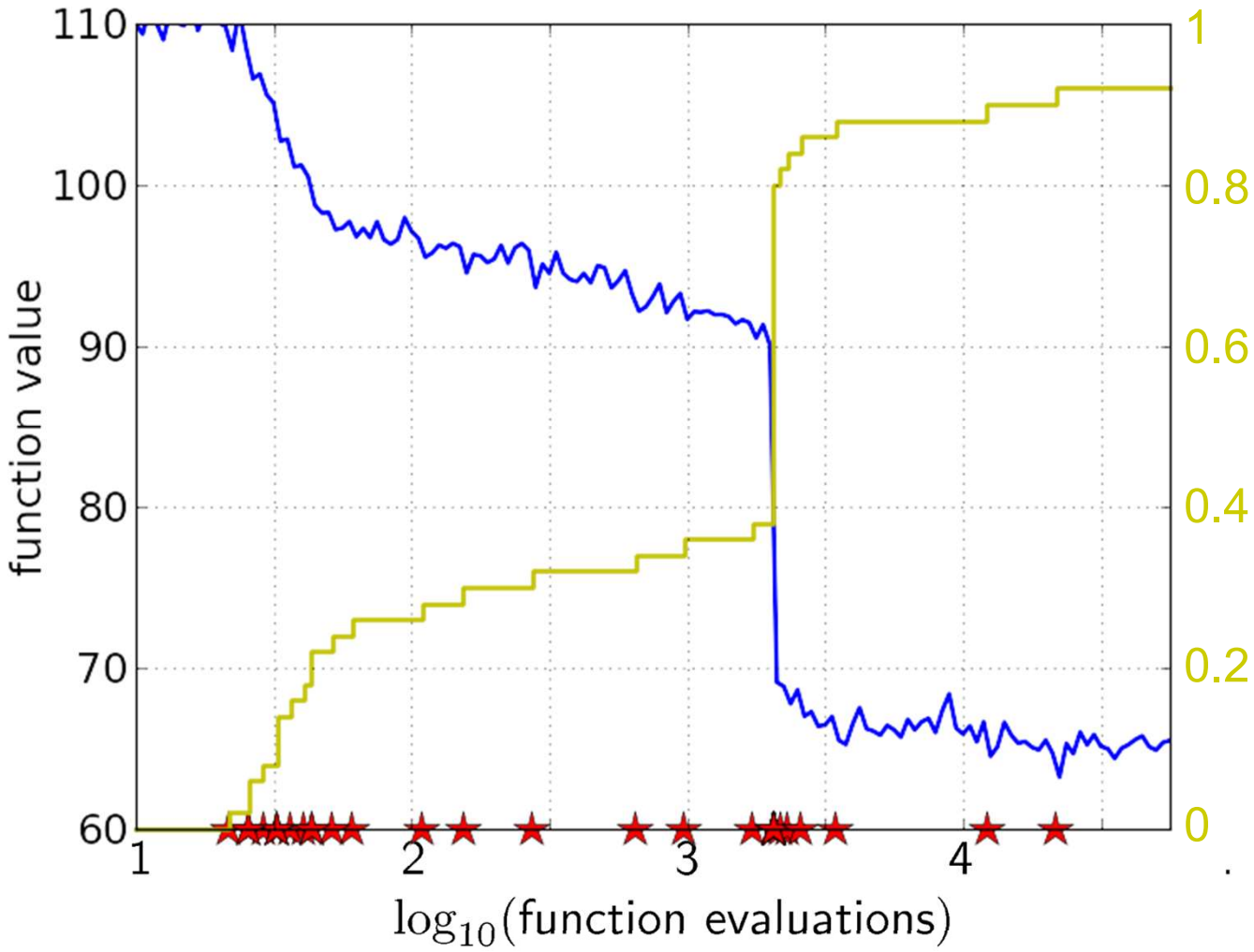
Reconstructing A Single Run



Reconstructing A Single Run

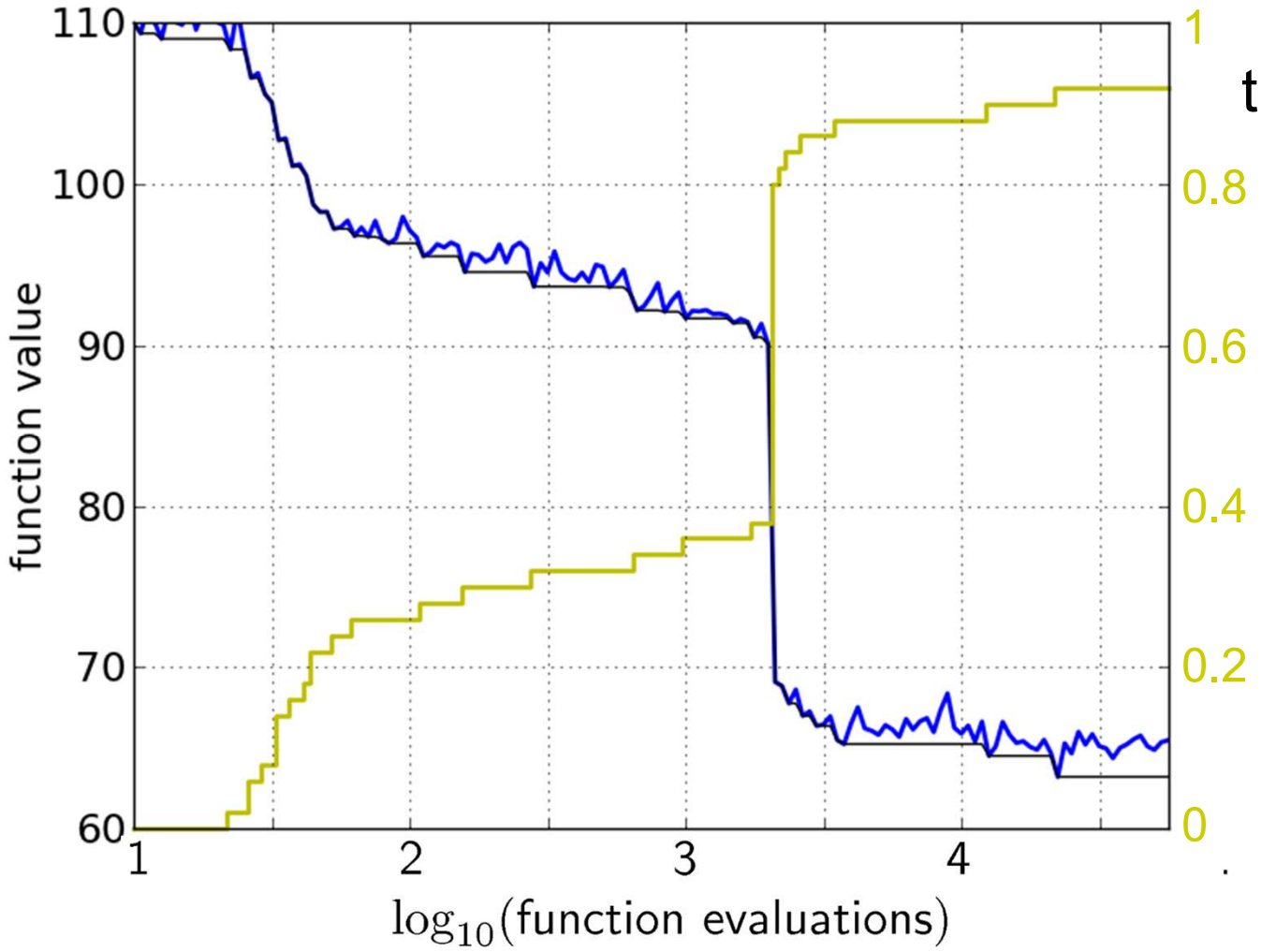


Reconstructing A Single Run



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

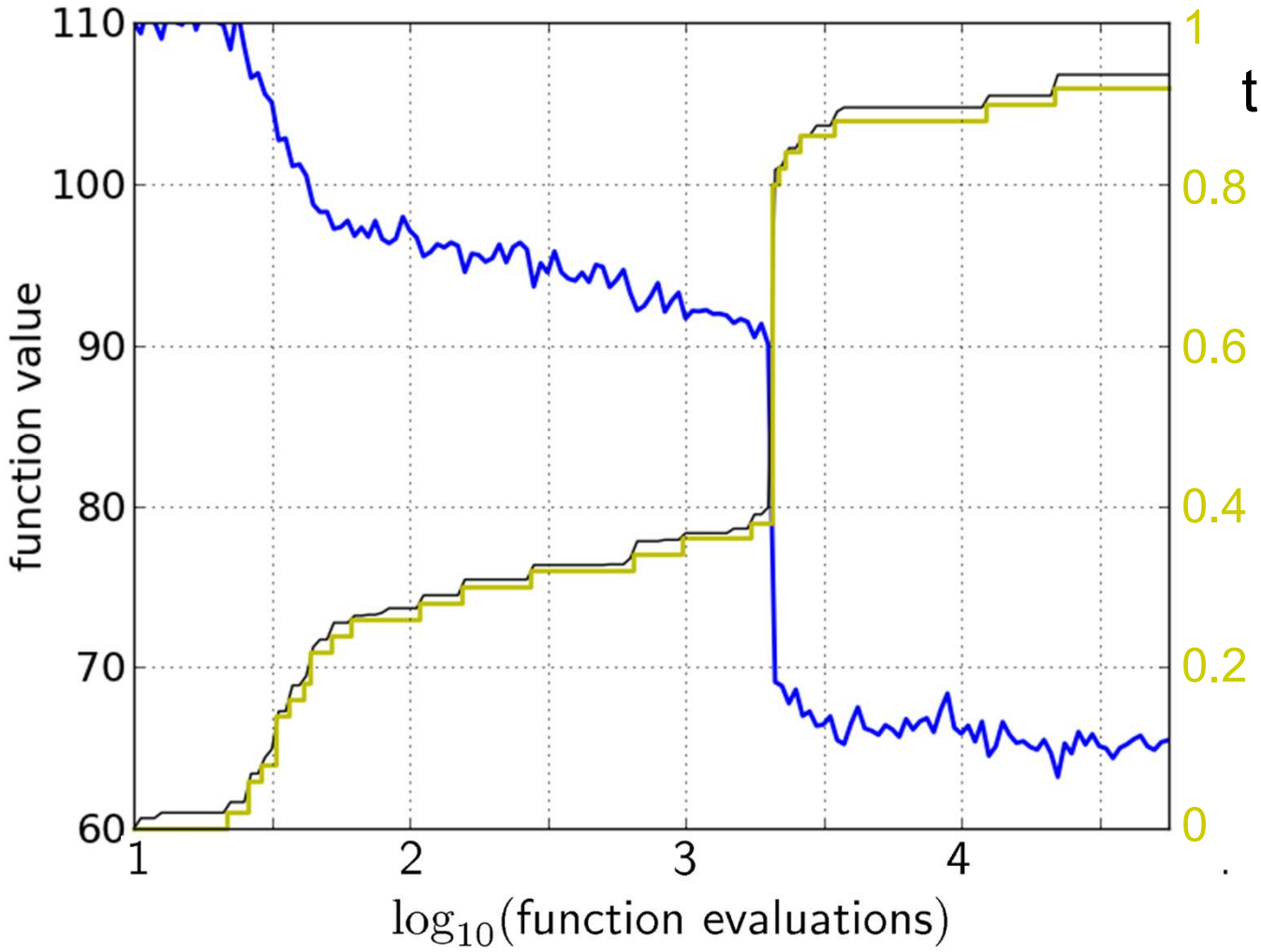
Reconstructing A Single Run



the ECDF recovers the monotonous graph, discretized and flipped

.

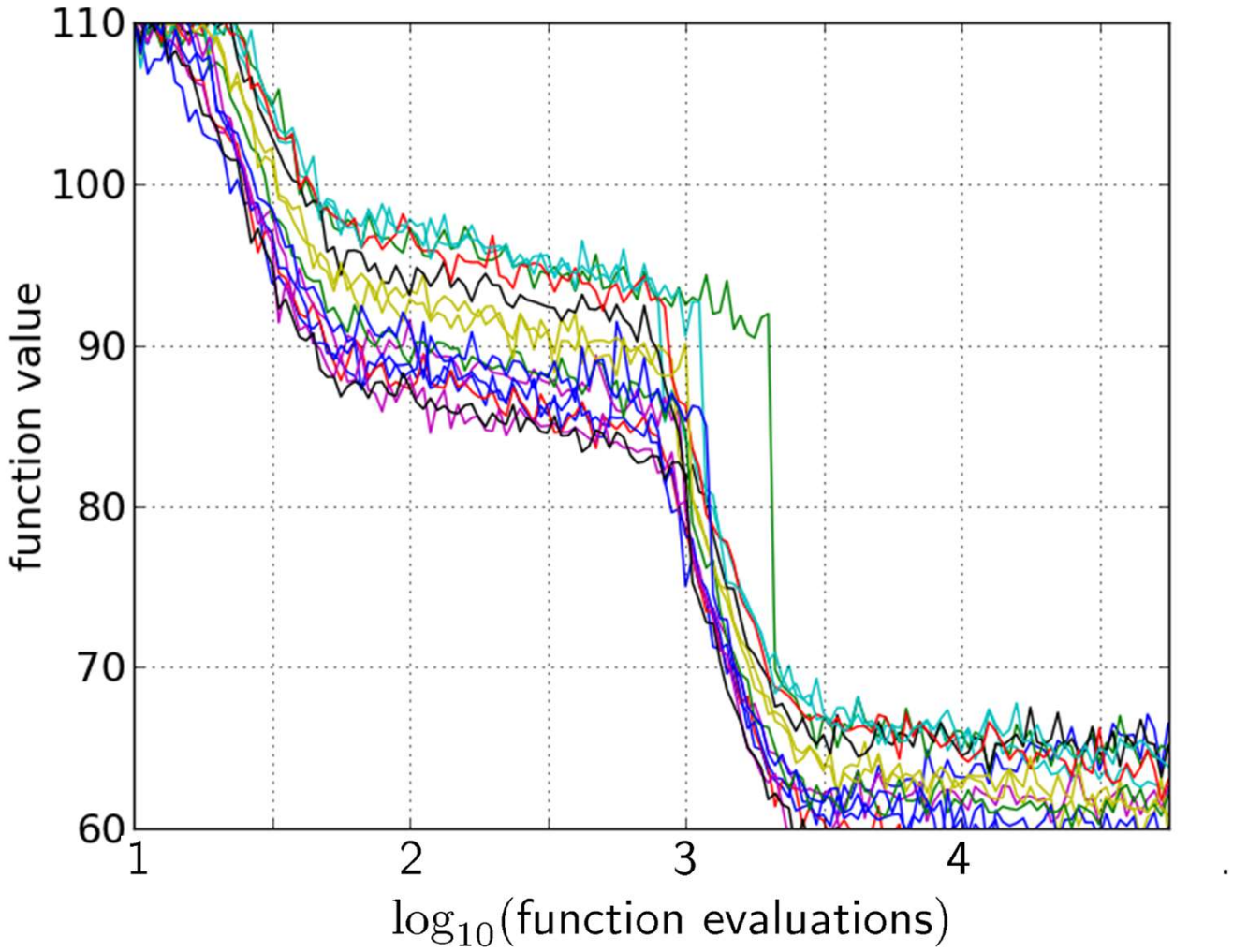
Reconstructing A Single Run



the ECDF recovers the monotonous graph, discretized and flipped

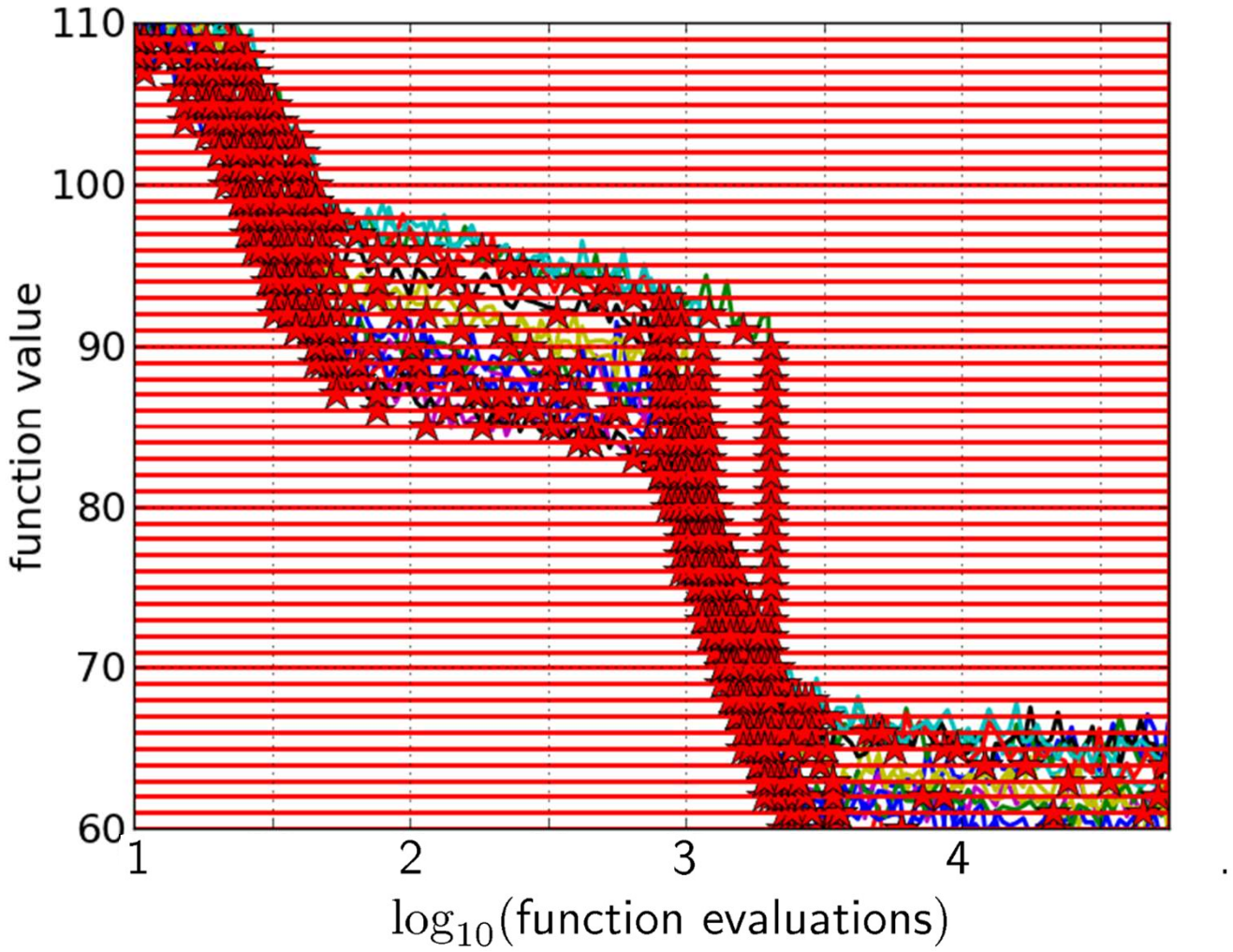
.

Aggregation



15 runs

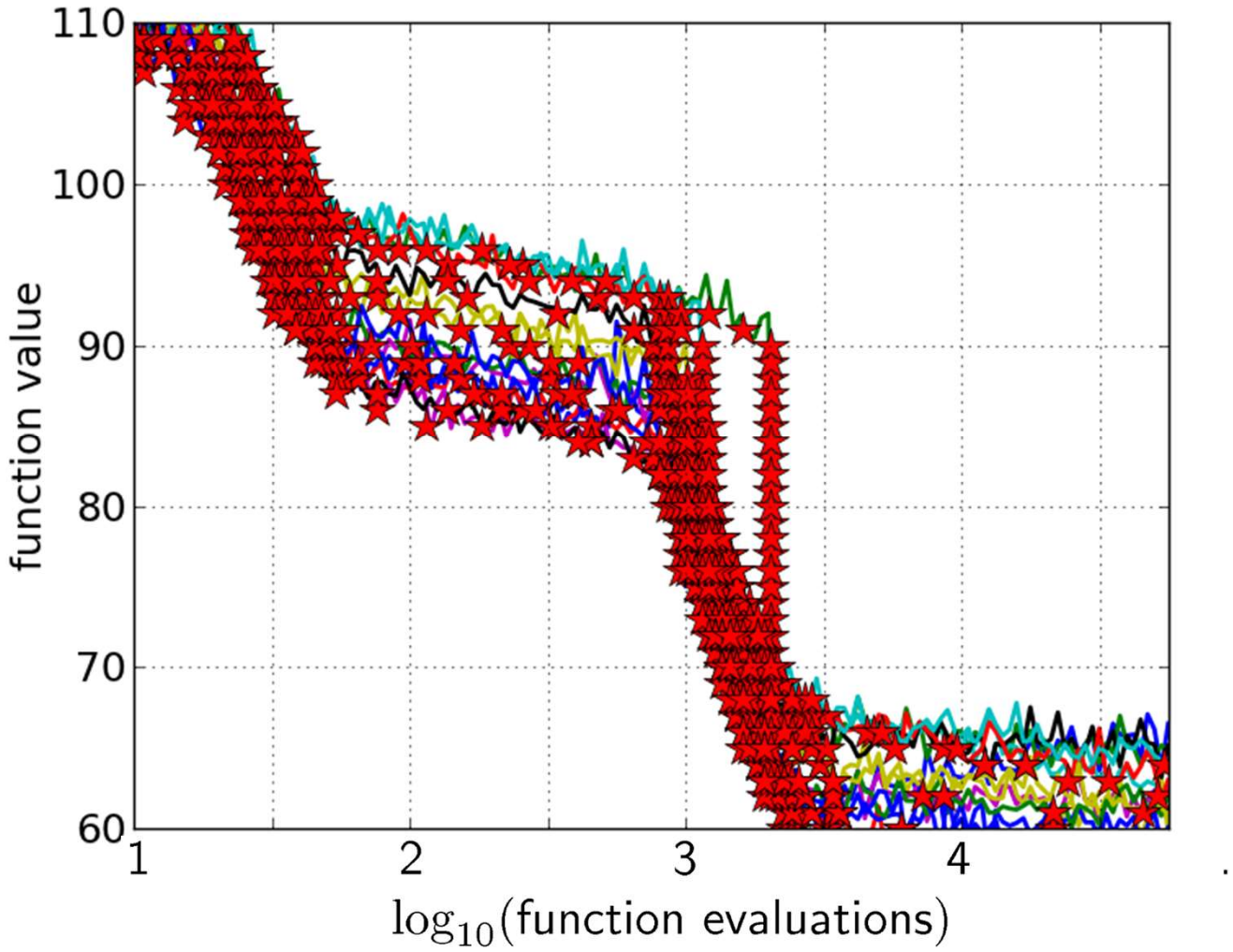
Aggregation



15 runs

50 targets

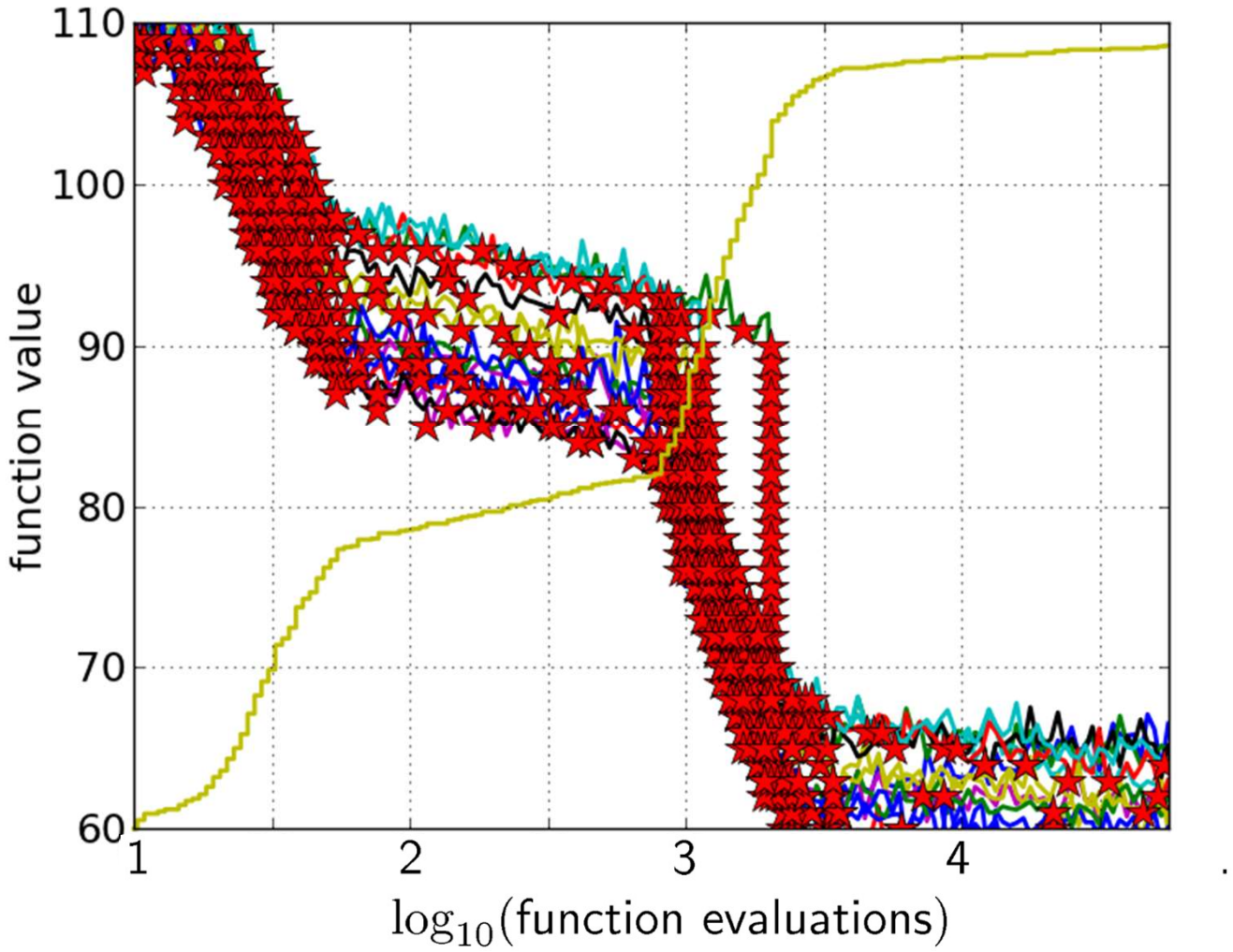
Aggregation



15 runs

50 targets

Aggregation

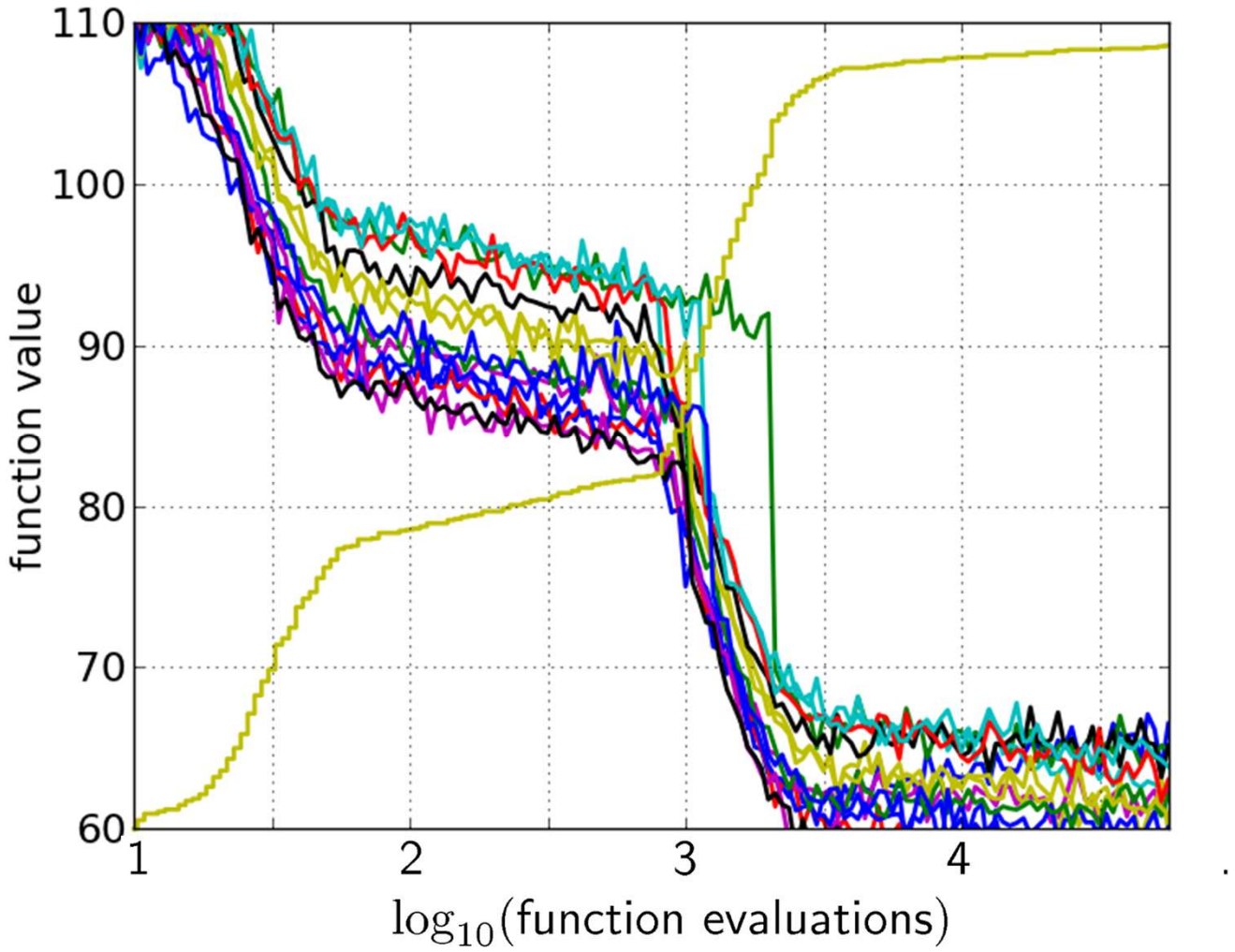


15 runs

50 targets

ECDF with
750 steps

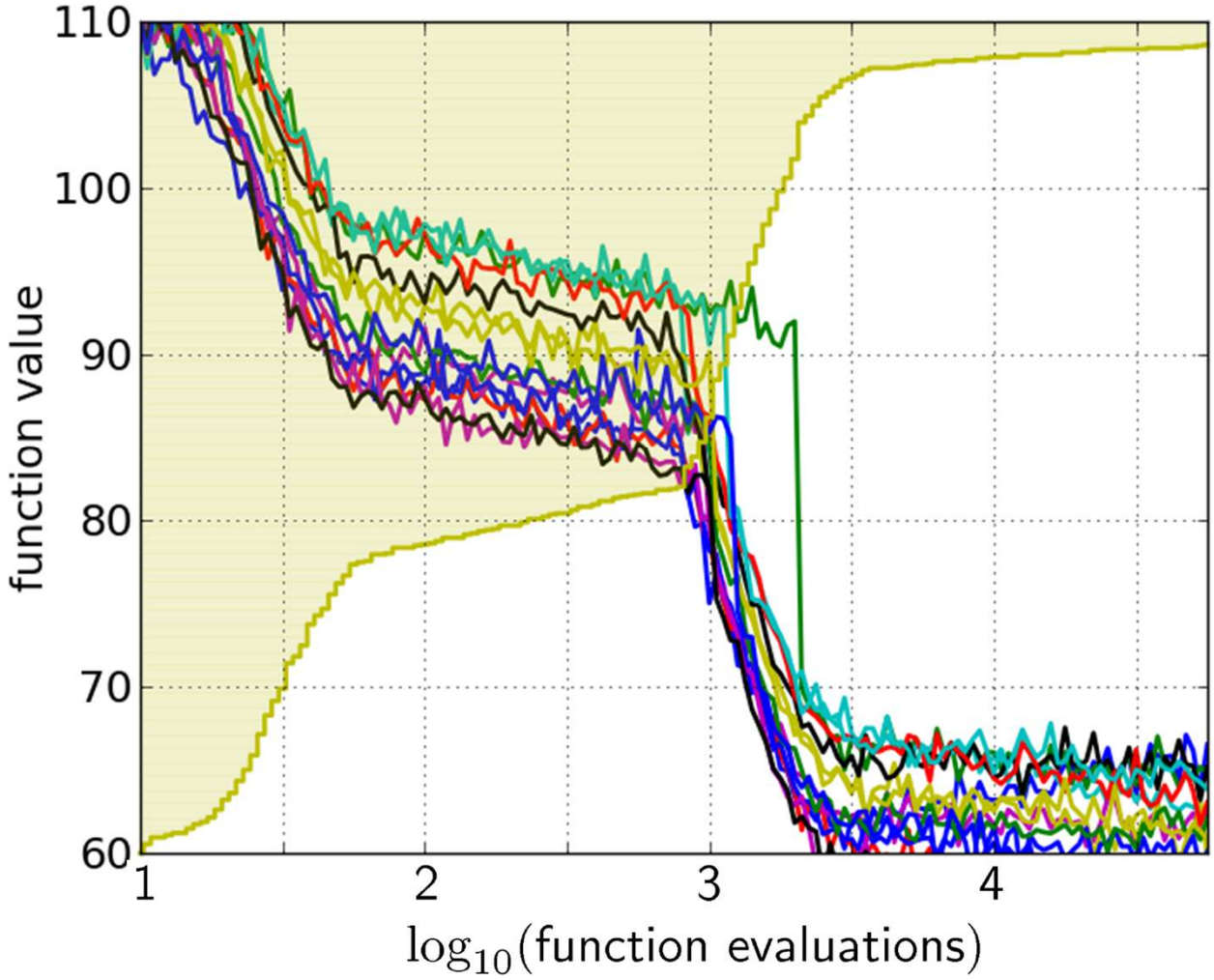
Aggregation



50 targets from 15 runs

...integrated in a single graph

Interpretation



area over the ECDF curve

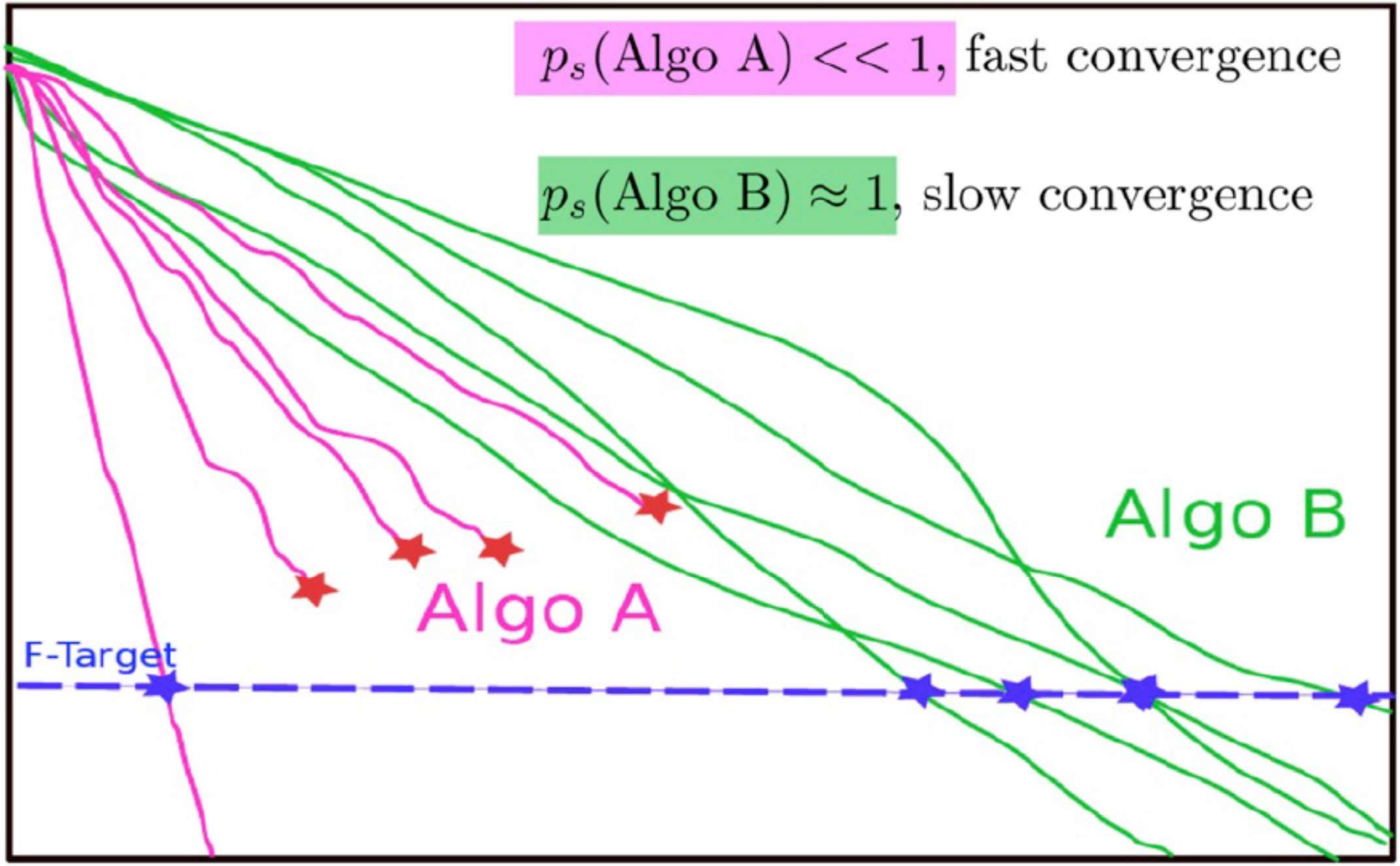
=

average log runtime

(or geometric avg. runtime) over all targets (difficult and easy) and all runs

.

Fixed-target: Measuring Runtime



Fixed-target: Measuring Runtime

Idea: Simulated restarts by bootstrapping from measured runtimes until we see a success

Algo Restart A:

p_s(Algo Restart A) = 1

Algo Restart B:

p_s(Algo Restart A) = 1

Fixed-target: Measuring Runtime

- Expected running time of the restarted algorithm:

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

- Estimator average running time (aRT/ERT/Enes/SP2*, w/o proof):

$$\hat{p_s} = \frac{\#\text{successes}}{\#\text{runs}}$$

$$\widehat{RT_{unsucc}} = \text{Average evals of unsuccessful runs}$$

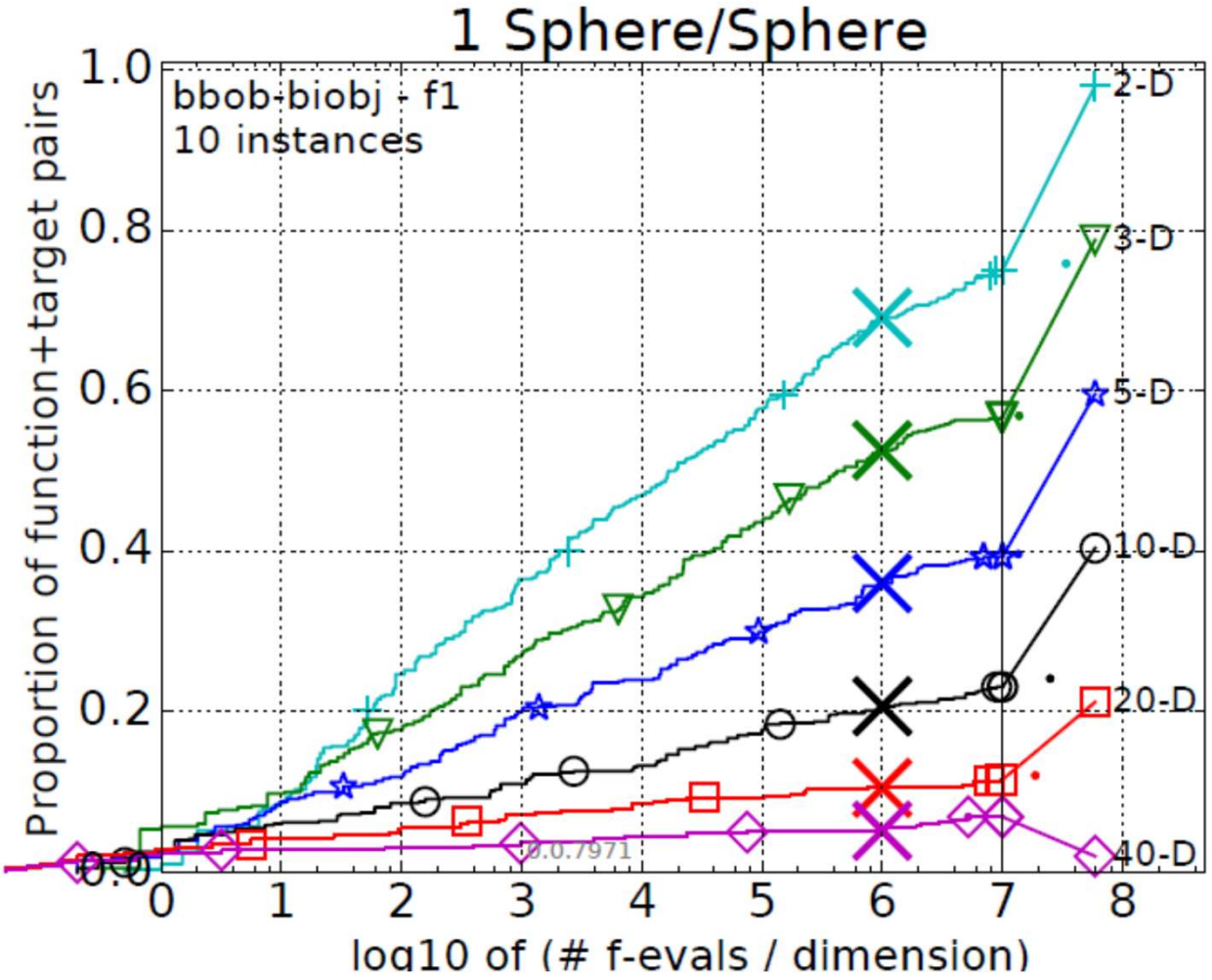
$$\widehat{RT_{succ}} = \text{Average evals of successful runs}$$

$$aRT = \frac{\text{total \#evals}}{\#\text{successes}}$$

* The concept is so essential that it has been discovered/proposed multiple times under different names in the past.

ECDFs with Simulated Restarts

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



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

- account for invariance properties

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

What to Benchmark?

Furious activity is no substitute for understanding (H.H. Williams)

- Taking all possible functions from a repository?

Copyright: A. Auger and N. Hansen

What to Benchmark?

Furious activity is no substitute for understanding (H.H. Williams)

- Taking all possible functions from a repository?
- Bad idea if
  - function difficulties are unbalanced
    too many small dimensional problems, convex problems…
  - and performance are aggregated
- Leads to bias in the performance assessment

What to Benchmark?

- test functions should be representative of difficulties we want to test

  therefore NFL has no relevance as assumption of being closed under permutation has no relevance wrt real world problems

- related to real-word difficulties

  for performance to be generalizable to RW

- scalable

  dimension plays a big role in performance
  curse of dimensionality

- comprehensible but not too easy

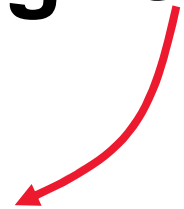
  BB optimization does not mean BB benchmarking

- we should still hide properties from the solver (hide optimum, …)

  solvers should not be able to exploit the benchmark intentionally or not

Copyright: A. Auger and N. Hansen

automated benchmarking: COCO



Comparing Continuous Optimizers Platform

`https://coco-platform.org`

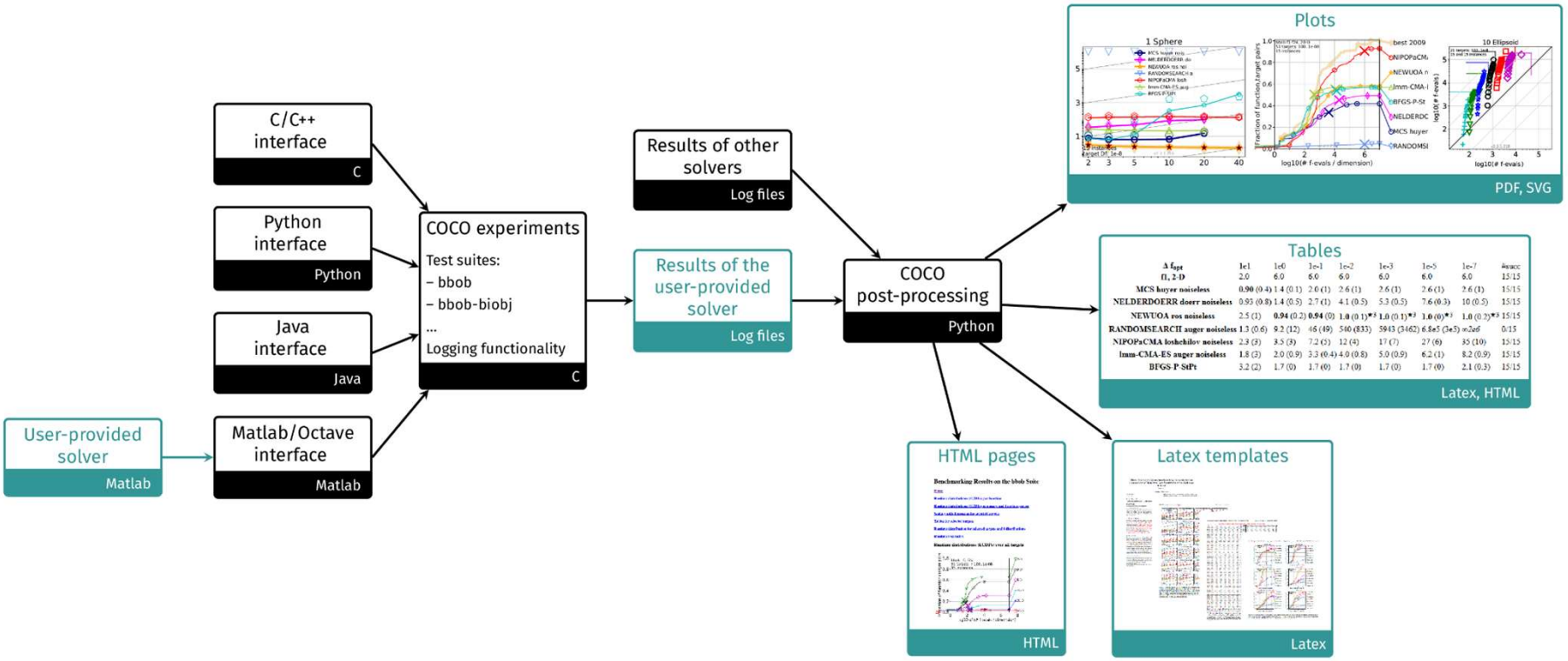
COCO implements a

reasonable, well-founded, and well-documented

pre-chosen methodology

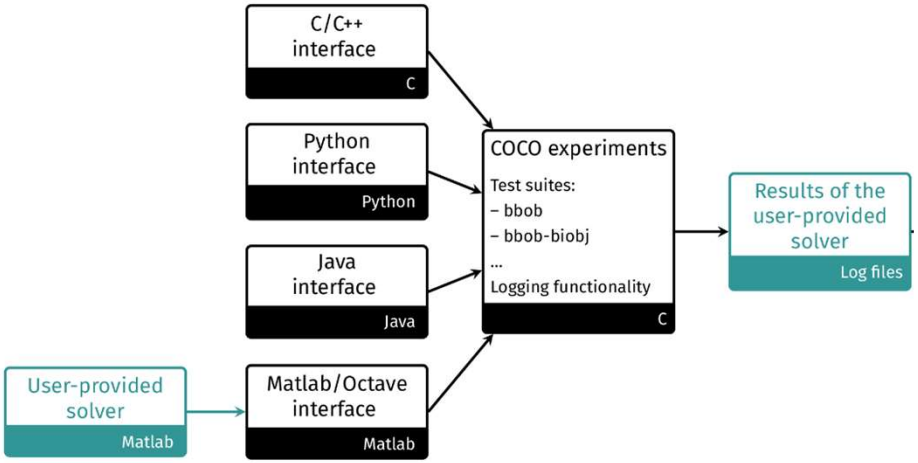
The Comparing Continuous Optimizers Platform

https://coco-platform.org



The Comparing Continuous Optimizers Platform

https://coco-platform.org

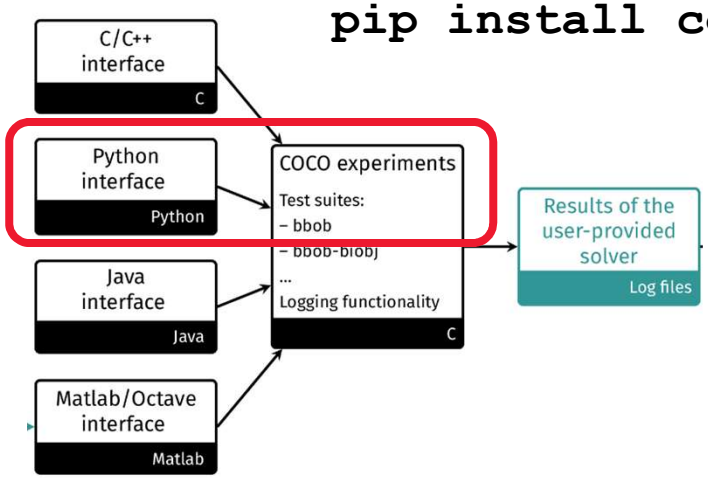


The Comparing Continuous Optimizers Platform

https://coco-platform.org

coco-experiment module:

```
pip install coco-experiment
```

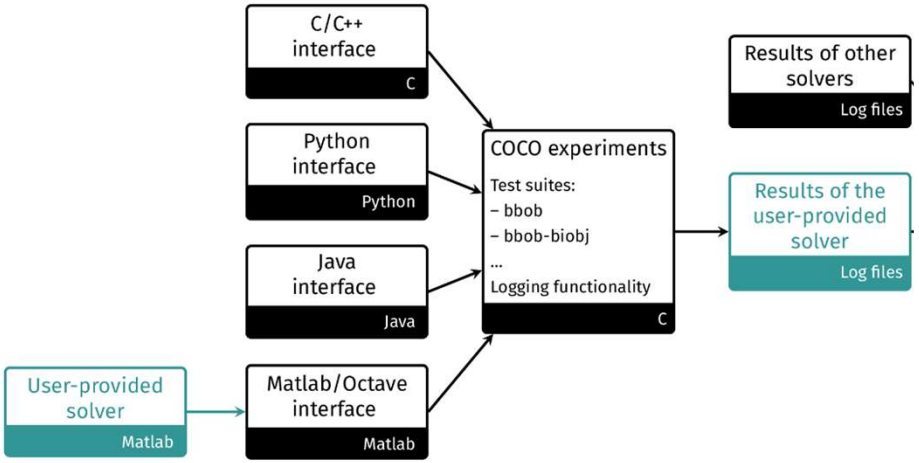


Experiments are simple (after `pip install coco-experiment`)

```
import cocoex
import scipy.optimize

### input
suite_name = "bbob"  # or "bbob-biobj" or ...
output_folder = "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
    scipy.optimize.fmin(problem, problem.initial_solution)
```

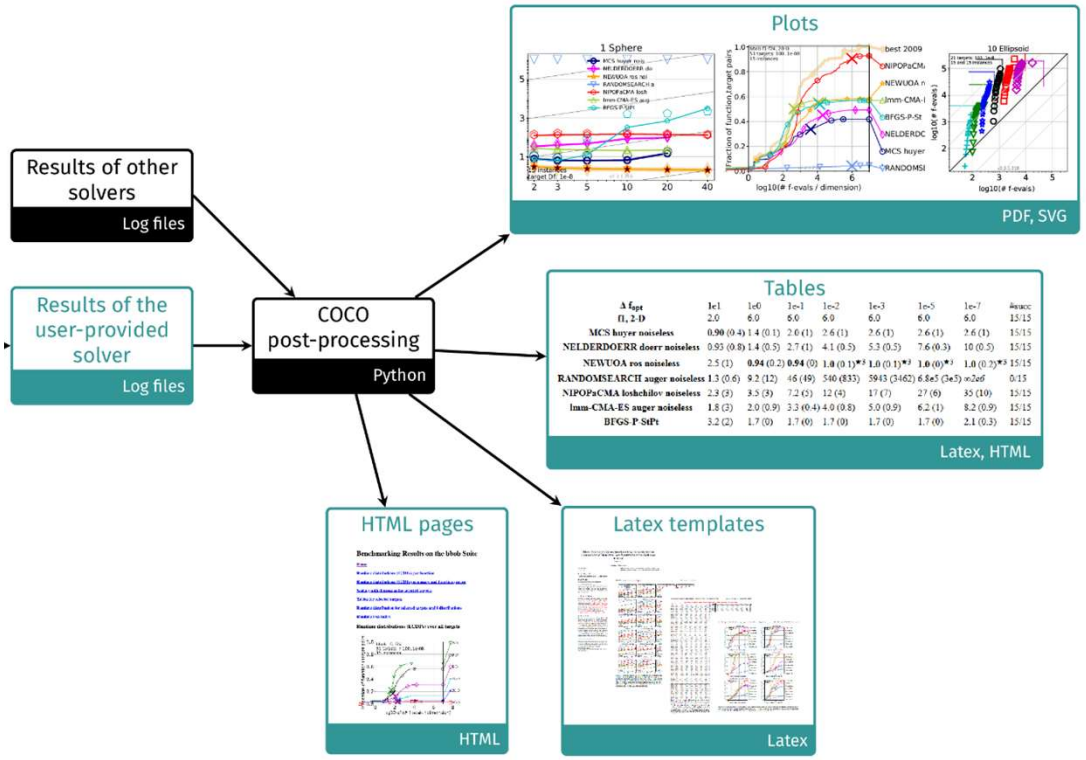
The Comparing Continuous Optimizers Platform

https://coco-platform.org



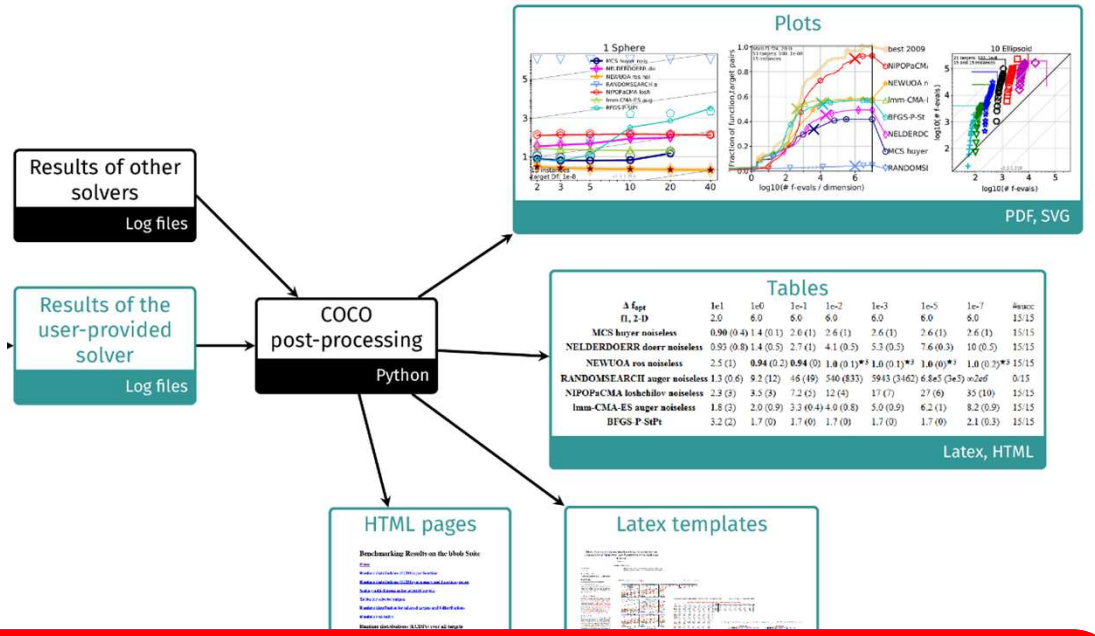
The Comparing Continuous Optimizers Platform

https://coco-platform.org



The Comparing Continuous Optimizers Platform

https://coco-platform.org



data from 300+ algorithms can be accessed directly through their name

(see https://coco-platform.org/data-archive/)

How to benchmark algorithms with COCO?

[in python]

https://coco-platform.org/getting-started/

installation I: experiments

As easy (in python) as:

```
pip install coco-experiment
```

installation II: postprocessing

```
pip install cocopp
```

for other languages, things might be slightly more involved…

Simplified Example Experiment in Python

```
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)
```

coupling algo + COCO

Note: the actual example_experiment_complete.py contains more advanced things like restarts, batch experiments, other algorithms (e.g. CMA-ES), etc.

https://coco-platform.org/getting-started/

running the experiment:

```
python example_experiments2.py 3
```

budget_multiplier:
experiments will be
run dimension times
this number many fevals

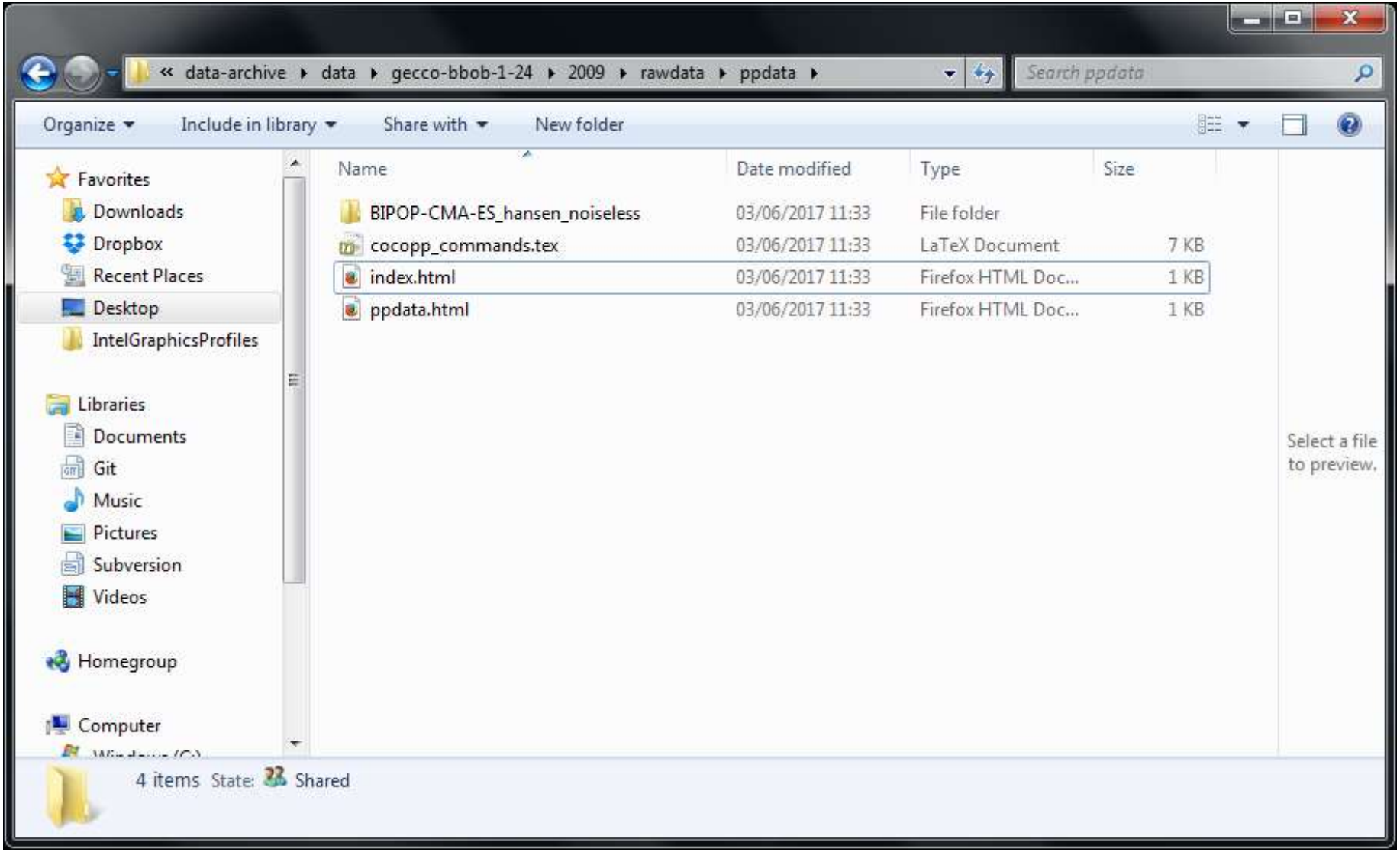
tip:
start with small #funevals (until bugs fixed ☺)
then increase budget to get a feeling
how long a "long run" will take

https://coco-platform.org/getting-started/

postprocessing

```
python -m cocopp YOURALGORITHMFOLDER ALG1 ALG2
```

ALG1, ALG2, … can be chosen
from any of the 300+ algorithms directly through its name
(see `https://coco-platform.org/data-archive/`)


« data-archive ▸ data ▸ gecco-bbob-1-24 ▸ 2009 ▸ rawdata ▸ ppdata ▸
Search ppdata
Organize
Include in library
Share with
New folder
Favorites
Downloads
Dropbox
Recent Places
Desktop
IntelGraphicsProfiles
Libraries
Documents
Git
Music
Pictures
Subversion
Videos
Homegroup
Computer
Name
Date modified
Type
Size
BIPOP-CMA-ES_hansen_noiseless 03/06/2017 11:33 File folder
cocopp_commands.tex 03/06/2017 11:33 LaTeX Document 7 KB
index.html 03/06/2017 11:33 Firefox HTML Doc... 1 KB
ppdata.html 03/06/2017 11:33 Firefox HTML Doc... 1 KB
Select a file to preview.
4 items State: Shared

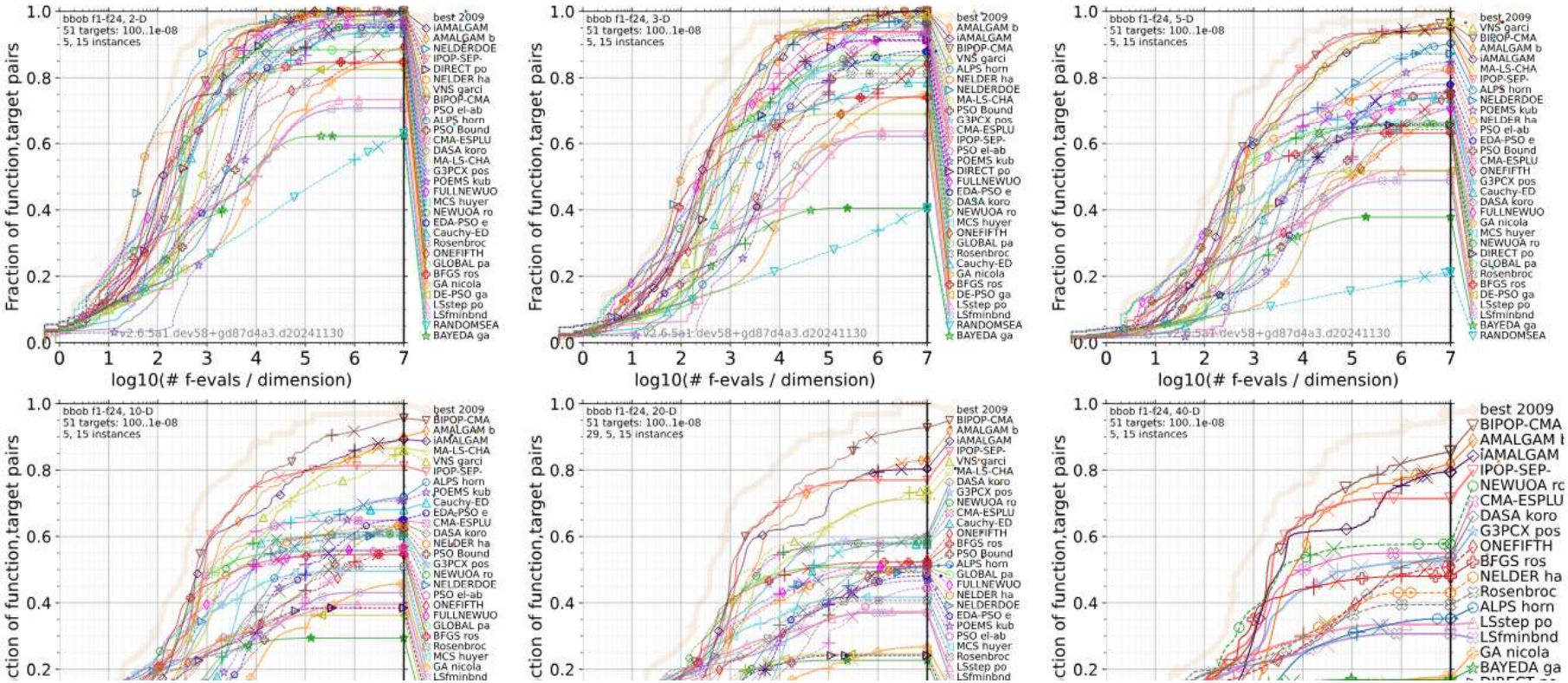
Automatically Generated Results

index

https://numbbo.github.io/ppdata-archive/bbob/2009/index.html

Home
Runtime profiles (dynamic navigation)
Runtime profiles per function
Runtime profiles summary and function groups
Scaling with dimension
Tables for selected targets

Runtime profiles over all targets



Available Test Suites in COCO

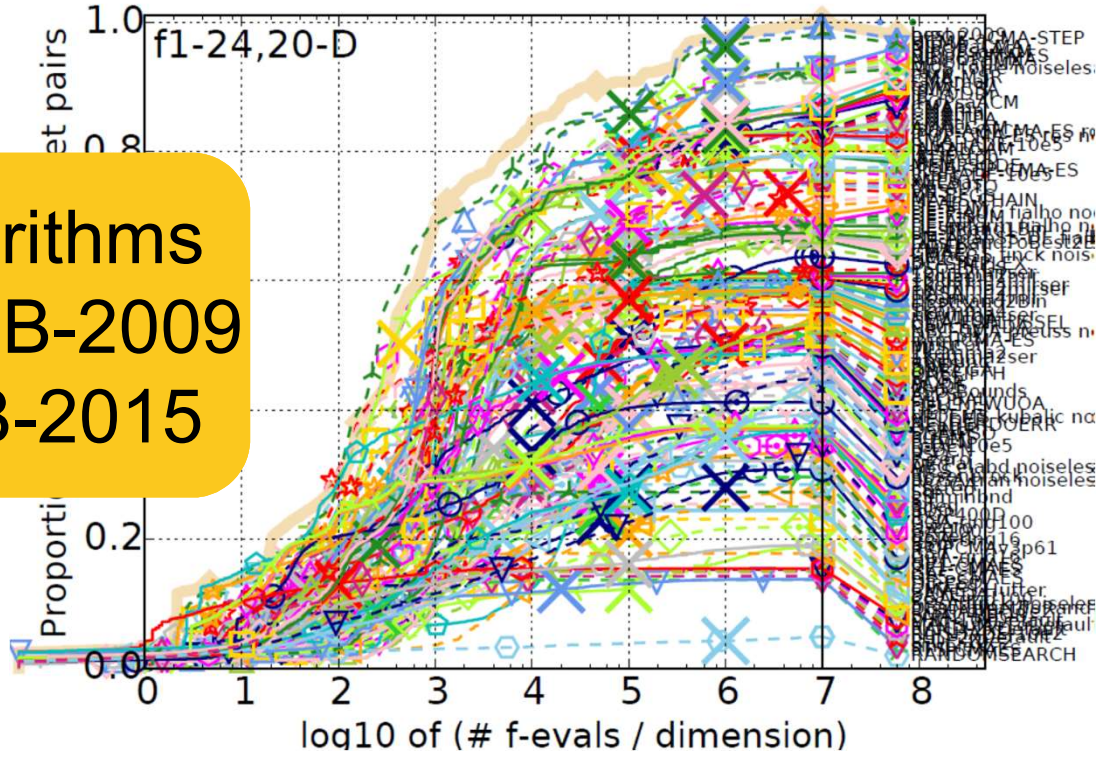
▪ bbob	24 noiseless fcts	250+ algo data sets
▪ bbob-noisy	30 noisy fcts	40+ algo data sets
▪ bbob-biobj	55 bi-objective fcts	30+ algo data sets
▪ bbob-largescale	24 noiseless fcts	16 algo data sets
▪ bbob-mixint	24 mixed-int. fcts	5 algo data sets
▪ bbob-constrained	54 fcts w/ varying constr.	9 algo data sets
▪ bbob-boxed	24 box-constr. fcts	3 algo data sets

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



The single-objective BBOB functions

https://numbbo.github.io/gforge/downloads/download16.00/bbobdocfunctions.pdf

The bbob Testbed

24 functions in 5 groups:

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

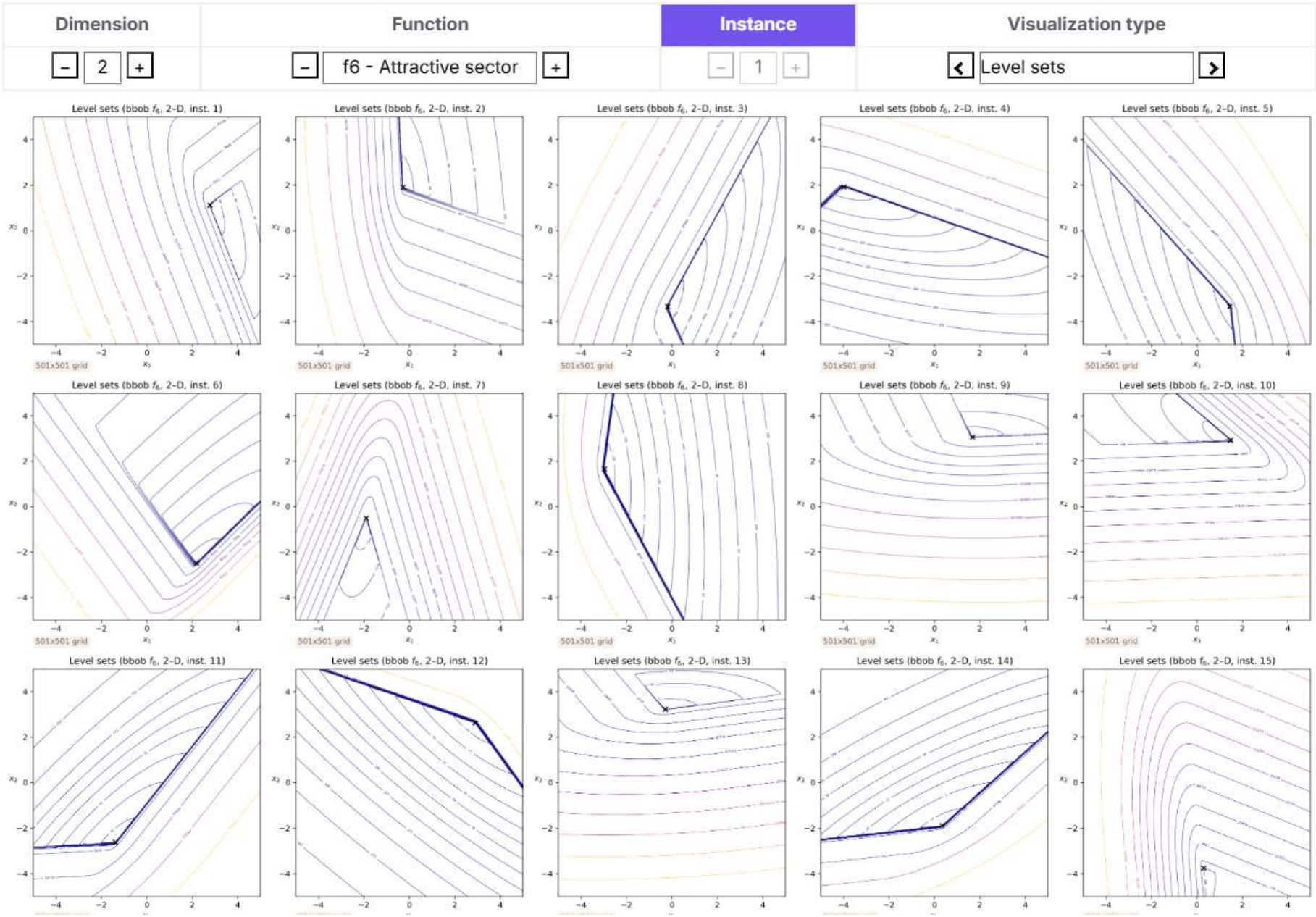
4 Multi-modal functions with adequate global structure	
f15	Rastrigin Function
f16	Weierstrass Function
f17	Schaffers F7 Function
f18	Schaffers F7 Functions, moderately ill-conditioned
f19	Composite Griewank-Rosenbrock Function F8F2
5 Multi-modal functions with weak global structure	
f20	Schwefel Function
f21	Gallagher's Gaussian 101-me Peaks Function
f22	Gallagher's Gaussian 21-hi Peaks Function
f23	Katsuura Function
f24	Lunacek bi-Rastrigin Function

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

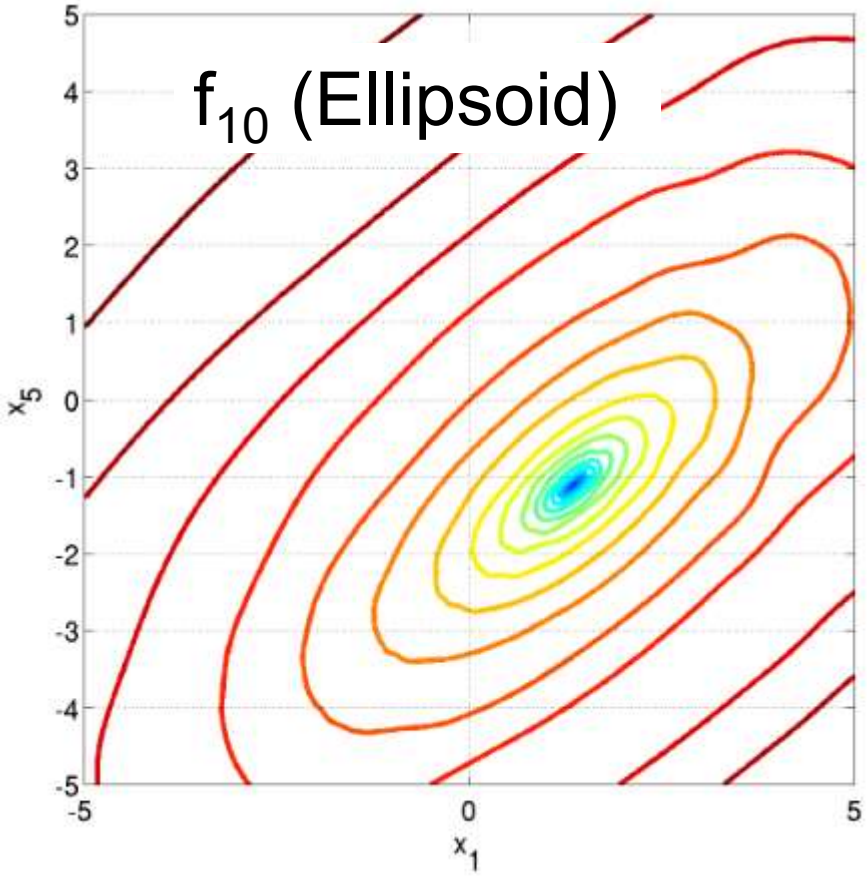
Notion of Instances

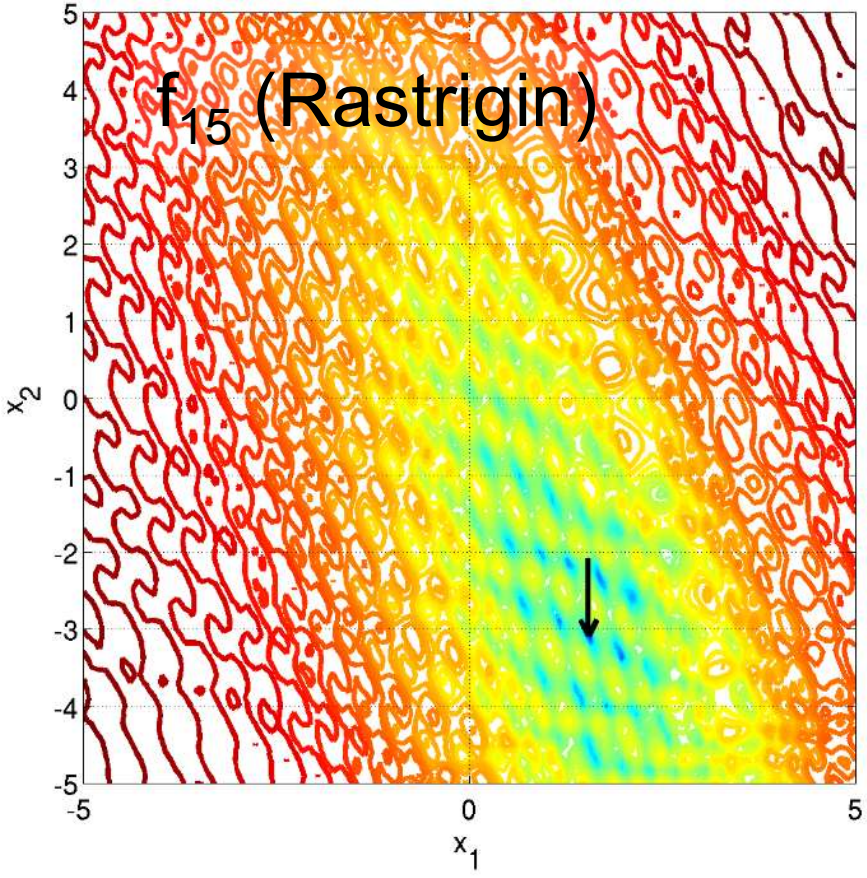


Notion of Instances

Plus:

- the bbob functions are locally perturbed by non-linear transformations





Exercise

Objectives:

- investigate the performance of these 6 algorithms:
  - CMA-ES ("IPOP-CMA-ES" version)
  - CMA-ES ("BIPOP-CMA-ES" version)
  - Nelder-Mead simplex (use "NelderDoerr" version here)
  - BFGS quasi-Newton
  - Genetic Algorithm: discretization of cont. variables ("GA")
  - ONEFIFTH: (1+1)-ES with 1/5 rule
- `pip install cocopp --pre`
- `python -m cocopp 2010/ipop! bipop! nelder! bfgs! 2009/GA ONEFIFTH!`
- postprocessed data available here: **http://www.cmap.polytechnique.fr/~dimo.brockhoff/advancedOptSaclay/2024/**
- so now: investigate the data!

Exercise

Objective:

investigate the data:

a) which algorithms are the best ones?

b) does this depend on the dimension? Or on other things?

c) look at single graphs: can we say something about the algorithms' invariances, e.g. wrt. rotations of the search space?

d) what's the impact of covariance-matrix-adaptation?

e) what do you think: are the displayed algorithms well-suited for problems with larger dimension?