A Practical Guide to Experimentation

Nikolaus Hansen
Inria
Research Centre Saclay, CMAP, Ecole polytechnique, Université Paris-Saclay

http://www.cmap.polytechnique.fr/~nikolaus.hansen/invitedtalks.html

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07.
https://doi.org/10.1145/3205651.3207850

Why Experimentation?

- The behaviour of many if not most interesting algorithms is
  - not amenable to a (full) theoretical analysis even when applied to simple problems
    calling for an alternative to theory for investigation
  - not fully comprehensible or even predictable without (extensive) empirical examinations
    even on simple problems
    comprehension is the main driving force for scientific progress
- Virtually all algorithms have parameters
  like most (physical/biological/…) models in science
  we rarely have explicit knowledge about the "right" choice
  this is a *big* obstacle in designing and benchmarking algorithms
- We are interested in solving *black-box* optimisation problems
  which may be "arbitrarily" complex

Scientific Experimentation

- What is the aim? *Answer a question*, ideally quickly and comprehensively
  consider in advance what the question is and in which way the experiment can answer the question
- do not (blindly) trust what one needs to rely on (code, claims, …) without *good* reasons
  check/test "everything" yourselves, practice stress testing, boosts also understanding
  one key element for success
  Why Most Published Research Findings Are False [Ioannidis 2005]
- run *rather many than few* experiments, as there are many questions to answer, practice *online experimentation*
  to run many experiments they must be *quick to implement and run*
  develops a feeling for the effect of setup changes
- run any experiment at least twice
  assuming that the outcome is stochastic
  get an estimator of variation
- display: *the more* the better, *the better* the better
  figures are *intuition pumps* (not only for presentation or publication)
  it is hardly possible to overestimate the value of a good figure
  data is the only way experimentation can help to answer questions, therefore look at them!

Scientific Experimentation

- don't make minimising CPU-time a primary objective
  avoid spending time in implementation details to tweak performance
- It is usually more important to know why algorithm A performs badly on function f, than to make A faster for unknown, unclear or trivial reasons
  mainly because an algorithm is applied to *unknown* functions
  and the "why" allows to predict the effect of design changes
- *Testing Heuristics: We Have it All Wrong* [Hooker 1995]
  "The emphasis on competition is fundamentally anti-intellectual and does not build the sort of insight that in the long run is conducive to more effective algorithms"
- there are many devils in the details, results or their interpretation may crucially depend on simple or intricate bugs or subtleties
  yet another reason to run many (slightly) different experiments
  check limit settings to give consistent results
- Invariance is a very powerful, almost indispensable tool

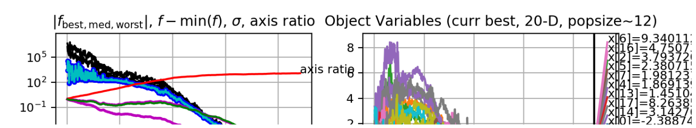
Jupyter IPython notebook

```
%pylab nbagg
import cma
cma.fmin(cma.ff.tablet, 20 * [1], 1);
```

```
Populating the interactive namespace from numpy and matplotlib
(6_w,12)-aCMA-ES (mu_w=3.7,w_1=40%) in dimension 20 (seed=344737, Wed Jul  5 16:09:44 2017)
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1     12 2.637846492377813e+03 1.0e+00 9.49e-01  9e-01  1e+00 0:00.0
    2     24 3.858353384747645e+04 1.1e+00 9.13e-01  9e-01  9e-01 0:00.0
    3     36 1.589934793439056e+04 1.2e+00 8.94e-01  9e-01  9e-01 0:00.0
  100   1200 1.805167565570186e+02 6.6e+00 2.52e-01  6e-02  3e-01 0:00.1
  200   2400 9.260486860109009e+01 4.2e+01 2.79e-01  1e-02  4e-01 0:00.3
  300   3600 8.460045942108286e+00 2.0e+02 3.20e-01  4e-03  4e-01 0:00.4
  400   4800 5.352841113616880e-02 5.2e+02 4.71e-02  2e-04  5e-02 0:00.5
  500   6000 1.169838413517761e-04 8.7e+02 2.61e-03  3e-06  2e-03 0:00.7
  600   7200 2.232682824828931e-08 9.9e+02 5.00e-05  4e-08  3e-05 0:00.8
  700   8400 1.483610308401096e-12 1.2e+03 4.61e-07  3e-10  2e-07 0:00.9
  736   8832 2.696542797455203e-14 1.2e+03 1.03e-07  5e-11  5e-08 0:01.0
termination on tolfun=1e-11 (Wed Jul  5 16:09:46 2017)
final/bestever f-value = 1.422957e-14 1.422957e-14
incumbent solution: [ -1.01044748e-11  -3.22608195e-08  -8.75163241e-10  -3.66834969e-08
   2.35485309e-08  -9.59521093e-10   4.23137381e-08   6.92049899e-09 ...]
std deviations: [  5.07976963e-11   4.52415829e-08   4.67529085e-08   4.36659472e-08
   4.04686177e-08   4.38294341e-08   4.65665203e-08   5.01580767e-08 ...]
```

```
cma.plot()
```

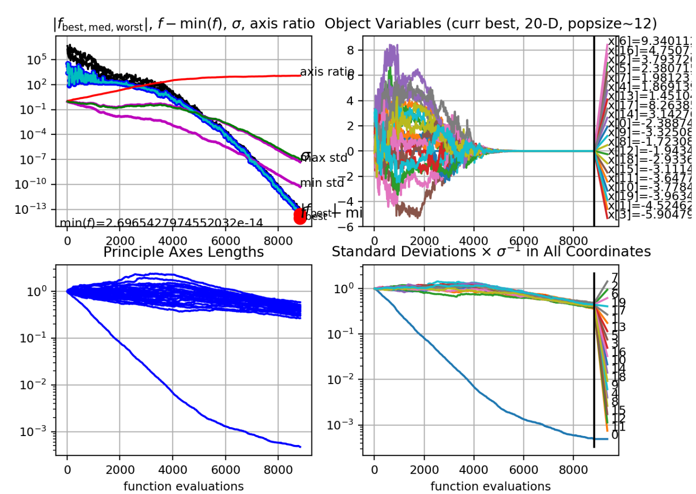
Figure 328



4.04686177e-08   4.38294341e-08   4.65665203e-08   5.01580767e-08 ...]

```
cma.plot()
```

Figure 328



Jupyter IPython notebook

```
# download&install anaconda python
# shell cmd "conda create" in case a different Python version is needed
# shell cmd "pip install cma" to install a CMA-ES module (or see github)
# shell cmd "jupyter-notebook" and click on compact-ga.ipynb
from __future__ import division, print_function
%pylab nbagg
```

```
Populating the interactive namespace from numpy and matplotlib
```

- Demonstration

Pure Random Search: Experimentation Summary

Results:

- the implementation seems consistent
  debugging of stochastic code is *really* tricky
  one possibility: compare two independent implementations (or with a reference implementation) with the same RNG and seeds
- scaling on onemax is indistinguishable from 1/2**n

Methodology:

- consider and exploit invariance
  one aspect: independence of *change of representation*
- run the quicker experiment first
  search space dimension is a simple control parameter
  taking a week of CPU-time in itself doesn't make the outcome more meaningful or informative
- adjust the *number of experiments* to the observed noise
  variation often decreases quickly with increasing dimension
  one can get away with single repetitions in a parameter sweep (two experiments per value)
- already *one single* repetition adds an estimator for variance
  any more repetitions only reduce the variance of this estimator

Invariance: onemax

Assigning 0/1

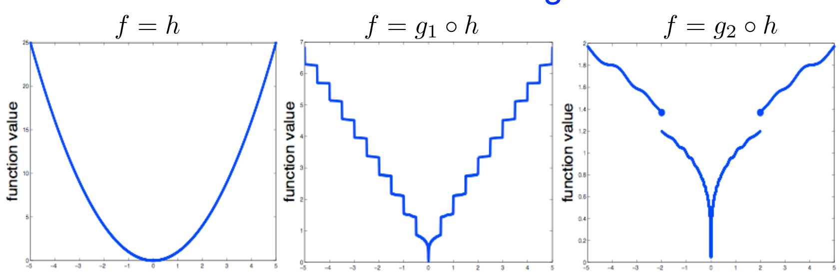
- is an “arbitrary” and “trivial” encoding choice and
- amounts to the affine linear transformation $x_i \mapsto -x_i + 1$

  the same transformation in each transformed variable
  continuous domain: isotropic (norm-preserving) transformation
- Does not change the function “structure”
  - all level sets $\{x \mid f(x) = \text{const}\}$ have the same size (number of elements, same volume)
  - no variable dependencies
  - same neighbourhood

Instead of 1 function, we now consider 2**n different but equivalent functions

2**n is non-trivial, it is the size of the search space itself

Invariance Under Order Preserving Transformations



Three functions belonging to the same equivalence class

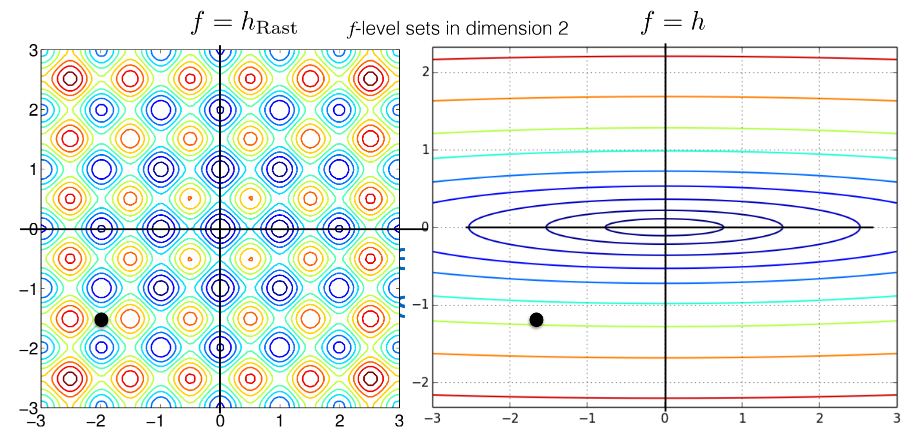
A *function-value free search algorithm* is invariant under the transformation with any order preserving (strictly increasing) g.

Invariances make

- observations meaningful
- algorithms predictable and/or ”robust”

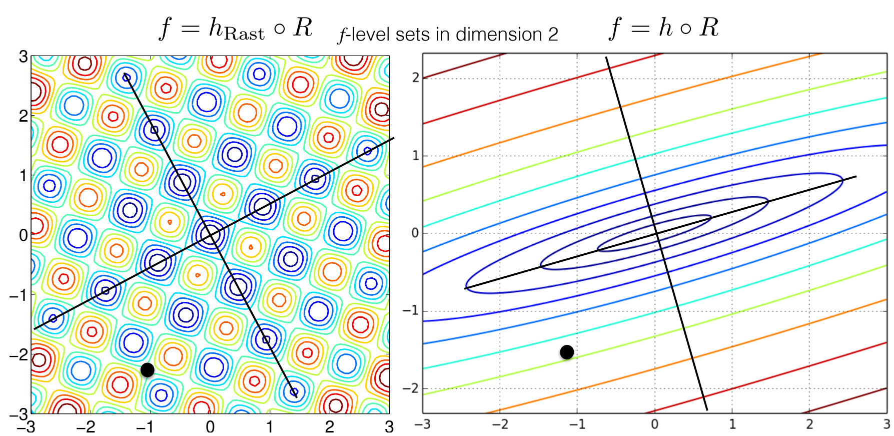
as a rigorous notion of generalization

Invariance Under Rigid Search Space Transformations



for example, invariance under search space rotation (separable vs non-separable)

Invariance Under Rigid Search Space Transformations



for example, invariance under search space rotation (separable vs non-separable)

Invariance

The grand aim of all science is to cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms.
— Albert Einstein

- Empirical performance results
  - from benchmark functions
  - from solved real world problems

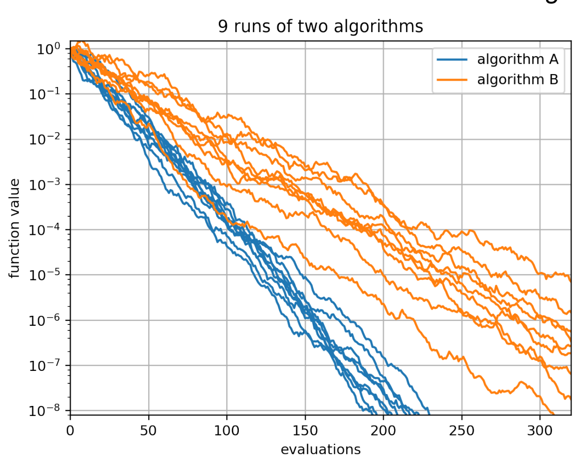
  are only useful if they do generalize to other problems

- Invariance is a strong non-empirical statement about generalization
  generalizing (identical) performance from a single function to a whole class of functions

Consequently, invariance is of greatest importance for the assessment of search algorithms.

Statistical Analysis

"experimental results lacking proper statistical analysis must be considered anecdotal at best, or even wholly inaccurate"
— M. Wineberg



Statistical Significance: General Prodecure

- *first*, check *the relevance* of the result, e.g., of the difference to be tested for statistical significance
  this also means: do not *explorative testing* (e.g. test *all* pairwise combinations)
  any ever so small difference can be made *statistically* significant with a simple trick,
  but *not made* significant in the sense of important or *meaningful*

- prefer "nonparametric" methods
  not based on a parametrised family of probability distributions

- p-value = significance level = probability of a false positive outcome
  smaller p-values are better
  <0.1% or <1% or <5% is usually considered as *statistically significant*

- for any found/observed p-value, *fewer data may be better*
  to achieve the same p-value with fewer data the *between*-difference must be larger than the *within*-variation

Statistical Significance: Methods

- use the rank-sum test (aka Wilcoxon or Mann-Whitney U test)

  - Assumption: all observations (data values) are independent
    The *lack of necessary preconditions* is the main reason to use the rank-sum test.
    yet, the rank-sum test is *nearly as efficient* as the t-test which requires normal distributions

  - Null hypothesis (nothing relevant is observed if): $\Pr(x < y) = \Pr(y < x)$
    the probability to be greater or smaller (better or worse) is the same
    the aim is to be able to reject the null hypothesis

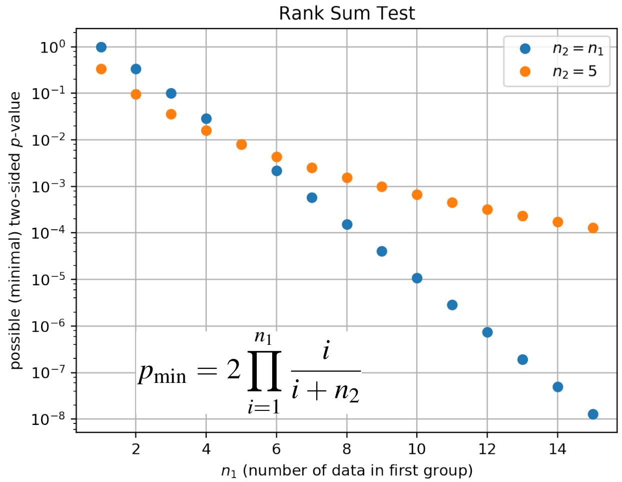
  - Procedure: compute the sum of ranks in the ranking of all (combined) data values

  - Outcome: a *p*-value
    the probability that this or a more extreme data set was generated under the null hypothesis
    the probability to *mistakenly* reject the null hypothesis

  - How many data do we need (two groups)? Five per group may suffice, *nine is plenty*.
    minimum number of data to possibly get two-sided
    $p < 1\%$: 5+5 or 4+6 or 3+9 or 2+19 or 1+200
    and $p < 5\%$: 4+4 or 3+5 or 2+8 or 1+40

Statistical Significance: How many data do we need?

AKA as test efficiency



- assumption: data are fully separated, i.e. x < y for all x, y
- observation: adding 2 data points in each group gives one additional order of magnitude

- use the Bonferroni correction for multiple tests

simple and conservative: multiply the computed p-value by the number of tests

Using Theory

"In the course of your work, you will from time to time encounter the situation where the facts and the theory do not coincide. In such circumstances, young gentlemen, it is my earnest advice to respect the facts."

— Igor Sikorsky, airplane and helicopter designer

Using Theory in Experimentation

- debugging / consistency checks

  theory may tell us what we *expect* to see

- knowing the limits (optimal bounds)

  e.g., we cannot converge faster than optimal
  trying to improve becomes a waste of time

- shape our expectations and objectives

Performance Assessment

- methodology: run an algorithm on a set of test functions and extract performance measures from the generated data

  choice of measure and aggregation

- display

  subtle display changes can make a huge difference

- there are surprisingly many devils in the details

Why do we want to measure performance?

- compare algorithms (the obvious)
  ideally we want standardised comparisons
- regression test after (small) changes
  as we may expect (small) changes in behaviour, conventional regression testing may not work
- algorithm selection (the obvious)
- understanding of algorithms
  very useful to improve algorithms
  non-standard experimentation is often preferable

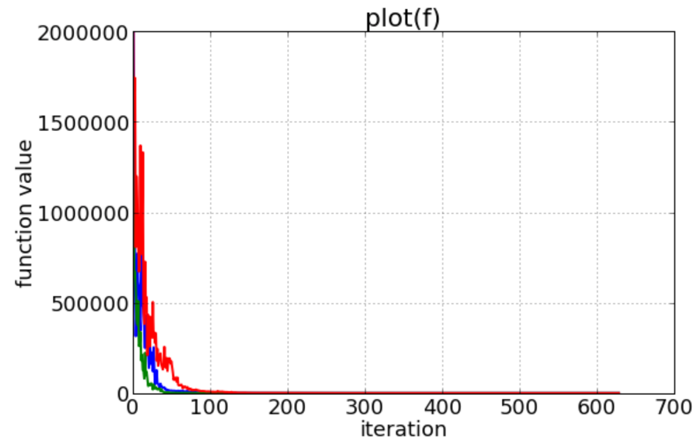
Measuring Performance

Empirically

convergence graphs is all we have to start with

having the right presentation is important

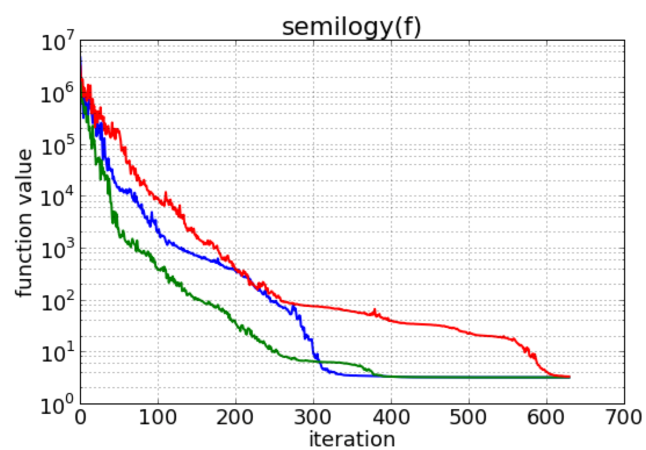
Displaying Three Runs



not like this (it's unfortunately not an uncommon picture)

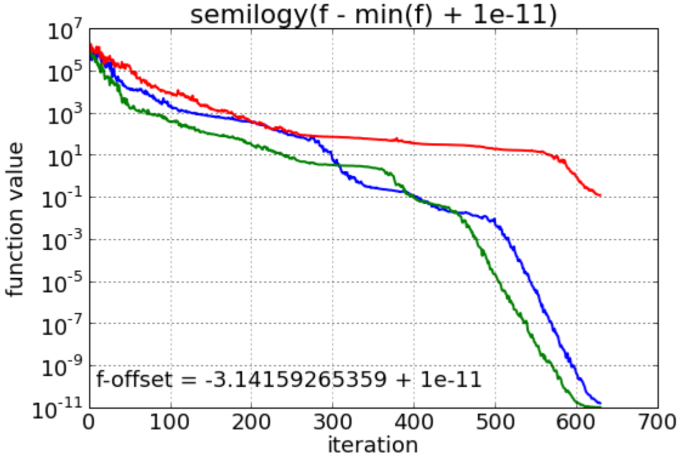
why not, what's wrong with it?

Displaying Three Runs



better like this (shown are the same data),
caveat: fails with negative f-values

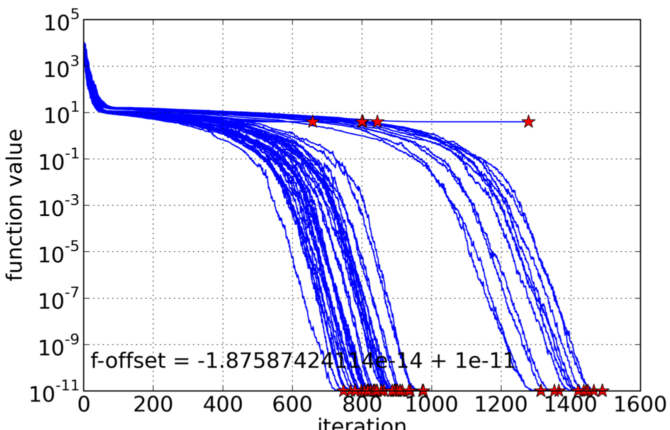
Displaying Three Runs



even better like this: subtract minimum value over all runs

Displaying 51 Runs

don't hesitate to display all data (the appendix is your friend)

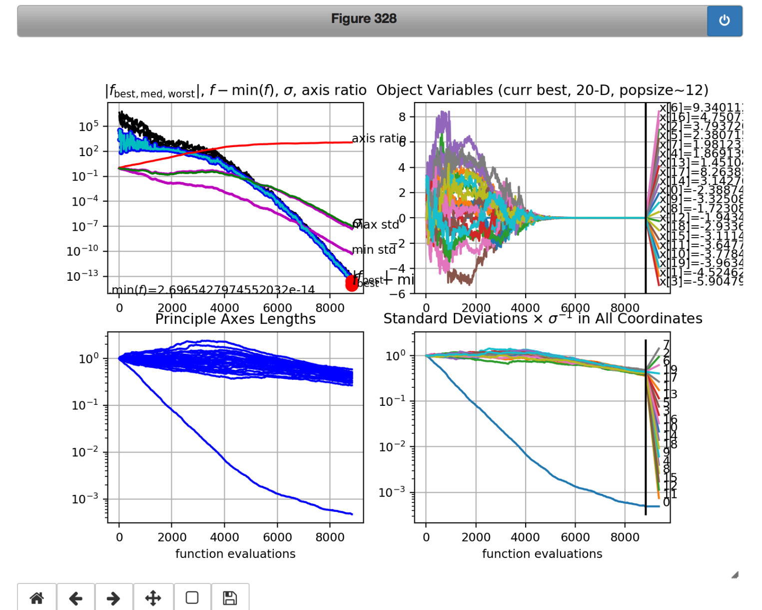


★ : final value

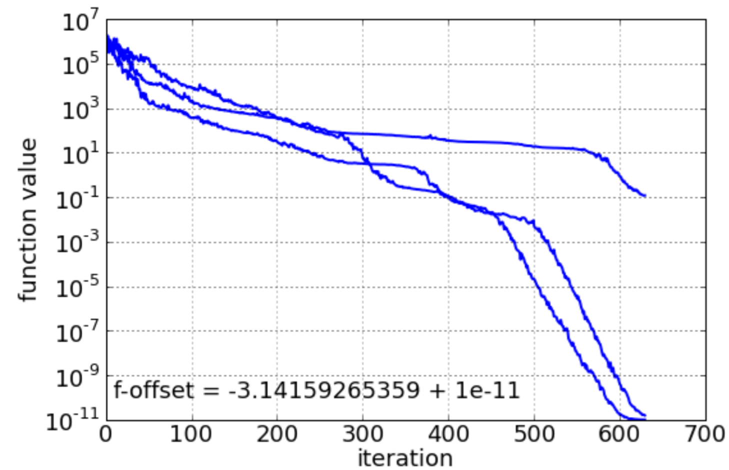
observation: three different "modes", which would be difficult to represent or recover in single statistics


```
4.04686177e-08   4.38294341e-08   4.65665203e-08   5.01580767e-08 ...]
cma.plot()
```

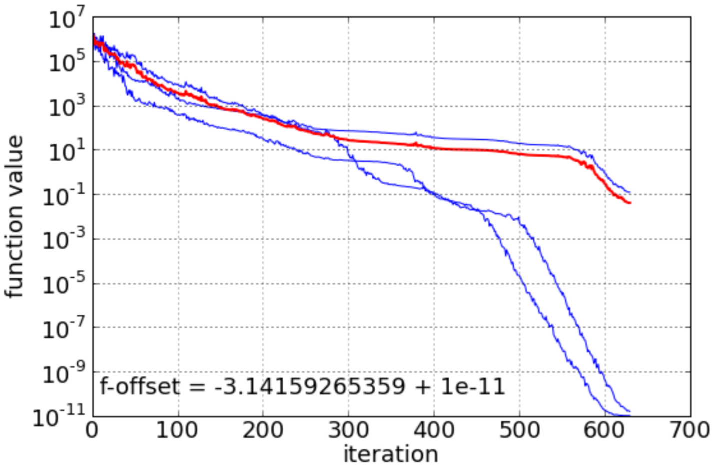
There is more to display than convergence graphs



Which Statistics?



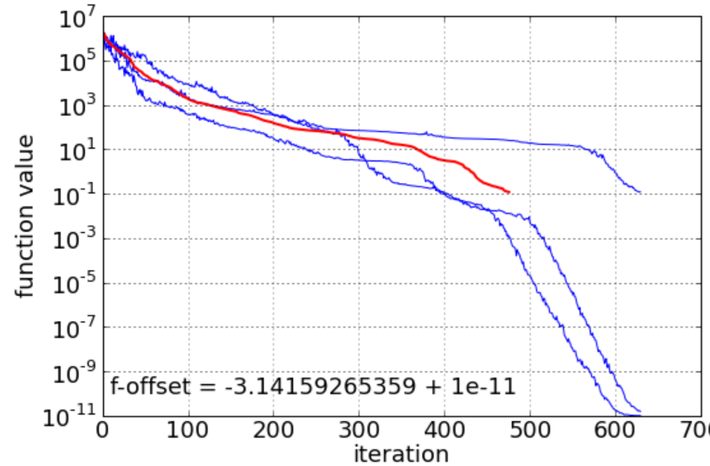
Which Statistics?



mean/average function value

- tends to emphasize large values

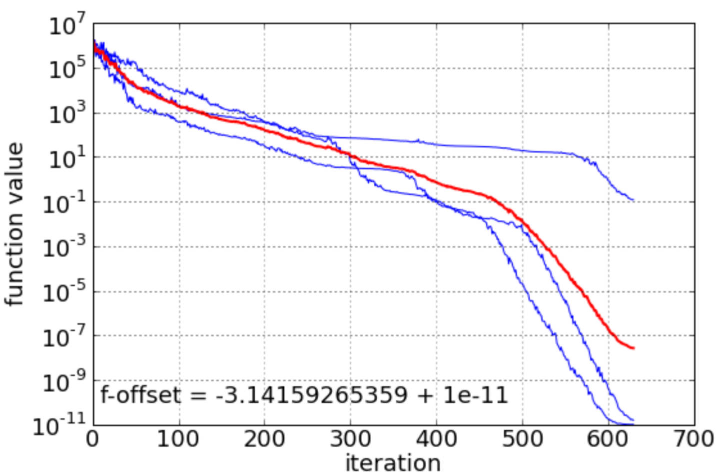
Which Statistics?



average iterations

- reflects "visual" average
- here: incomplete

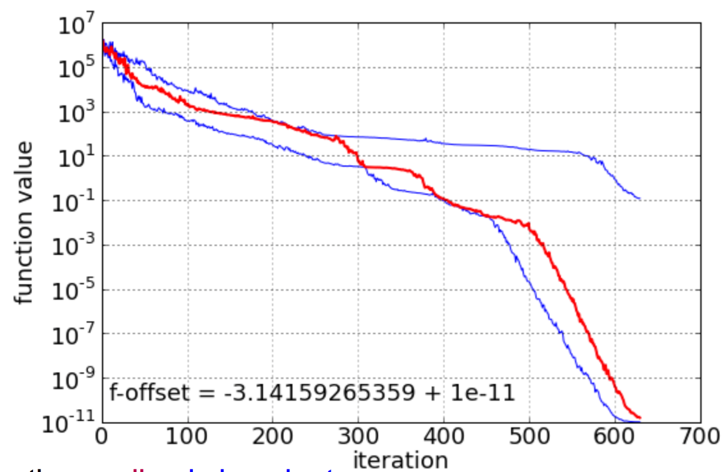
Which Statistics?



geometric average function value $\exp(\text{mean}_i(\log(f_i)))$

- reflects "visual" average
- depends on offset

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

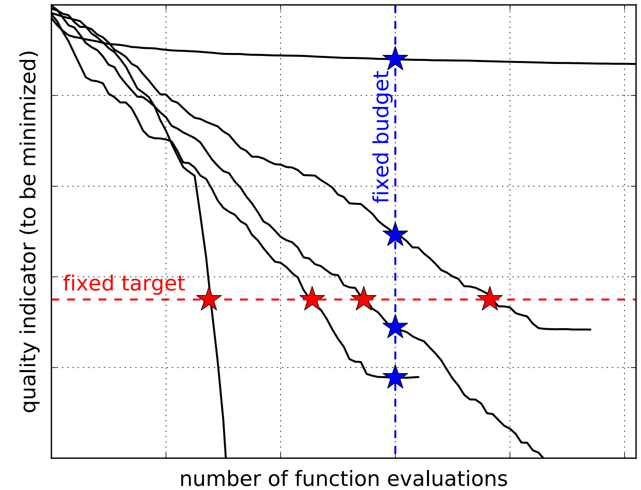
Implications

unless there are good reasons for a different statistics use the median as summary datum

more general: use quantiles as summary data

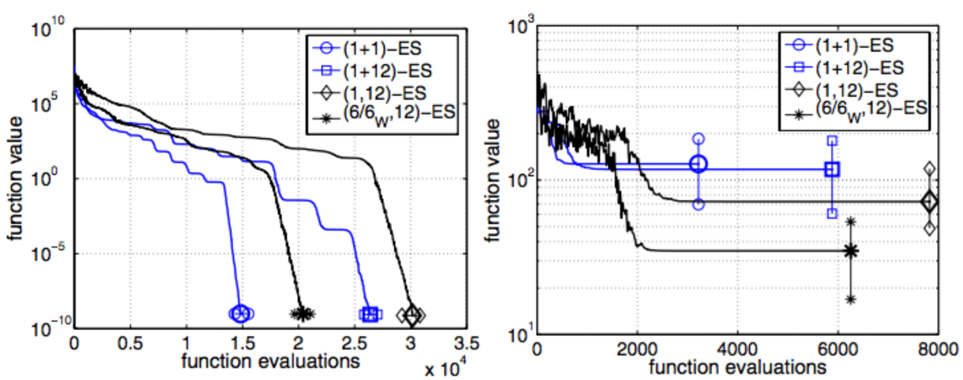
for example out of 15 data: 2nd, 8th, and 14th value represent the 10%, 50%, and 90%-tile

Aggregation: Fixed Budget vs Fixed Target



- for aggregation we need comparable data
- missing data: problematic when most or all runs lead to missing data
  - fixed target approach misses out on bad results (we may correct for this to some extend)
  - fixed budget approach misses out on good results

Examples



Comparison of 4 algorithms using the "median run" and the 90% central range of the final value on two different functions (Ellipsoid and Rastrigin)

caveat: this range display with simple error bars fails, if, e.g., 30% of all runs "converge"

Fixed Budget vs Fixed Target

Number of function evaluations are

- ***quantitatively*** comparable (on a ratio scale)
  ratio scale: "A is 3.5 times faster than B" (A/B = 1/3.5) is meaningful
- as measurement independent of the function
  time remains the same time

=> fixed target

Performance Measures for Evaluation

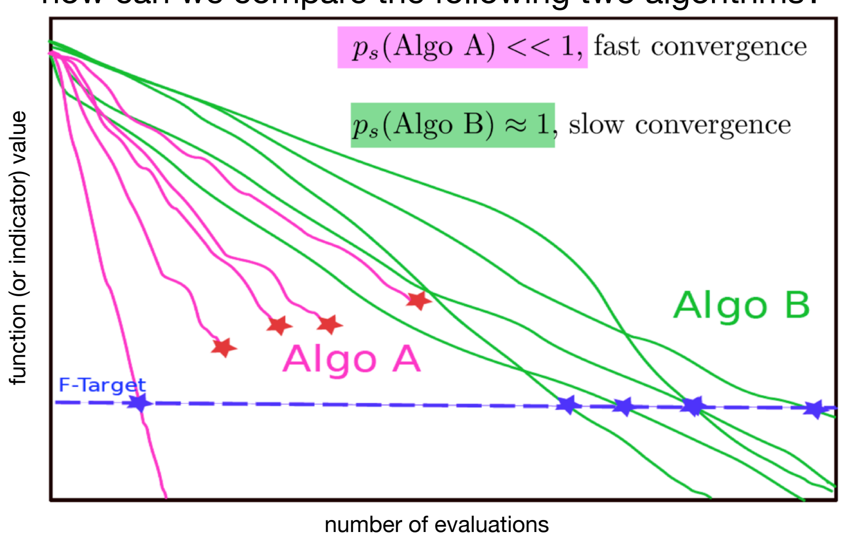
Generally, a performance measure should be

- quantitative on the ratio scale (highest possible)
  - "algorithm A is two *times* better than algorithm B" is a meaningful statement
- can 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, hence we use fixed targets

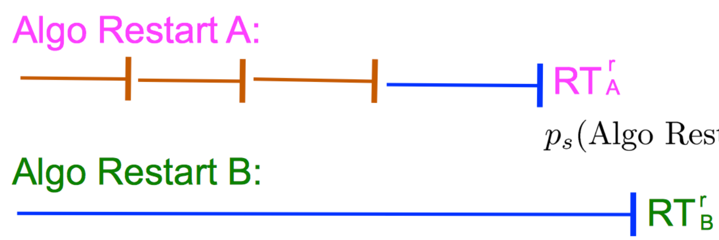
The Problem of Missing Values

how can we compare the following two algorithms?



The Problem of Missing Values

Consider simulated (artificial) restarts using the given independent runns



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

$p_s(\text{Algo Restart B}) = 1$

The Problem of Missing Values

The expected runtime (ERT, aka SP2, aRT) to hit a target value in #evaluations is computed (estimated) as:

$$\text{ERT} = \frac{\#\text{evaluations(until to hit the target)}}{\#\text{successes}}$$

unsuccessful runs count (only) in the nominator

$$= \text{mean}(\text{evals}_{\text{succ}}) + \overbrace{\frac{N_{\text{unsucc}}}{N_{\text{succ}}}}^{\text{odds ratio}} \times \text{mean}(\text{evals}_{\text{unsucc}})$$

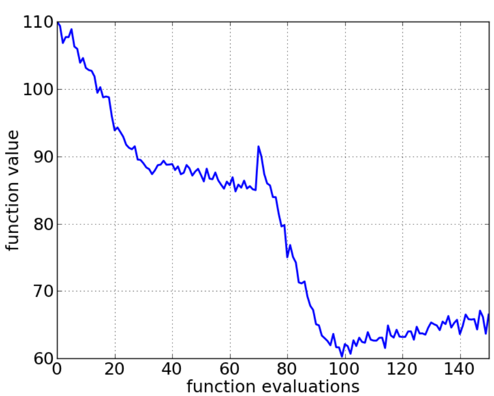
$$\approx \text{mean}(\text{evals}_{\text{succ}}) + \frac{N_{\text{unsucc}}}{N_{\text{succ}}} \times \text{mean}(\text{evals}_{\text{succ}})$$

$$= \frac{N_{\text{succ}} + N_{\text{unsucc}}}{N_{\text{succ}}} \times \text{mean}(\text{evals}_{\text{succ}})$$

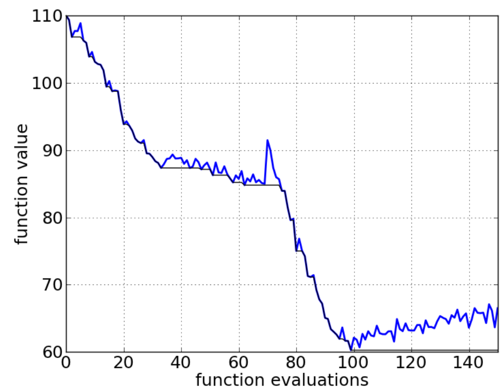
defined (only) for #successes > 0. The last two lines are aka Q-measure or SP1 (success performance).

Empirical Distribution Functions

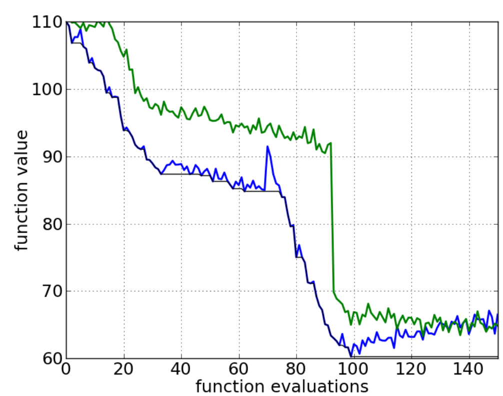
- Empirical cumulative distribution functions (ECDF) are arguably the single most powerful tool to display “aggregated” data.



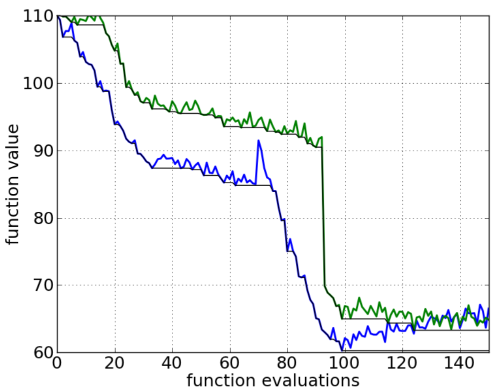
- a convergence graph



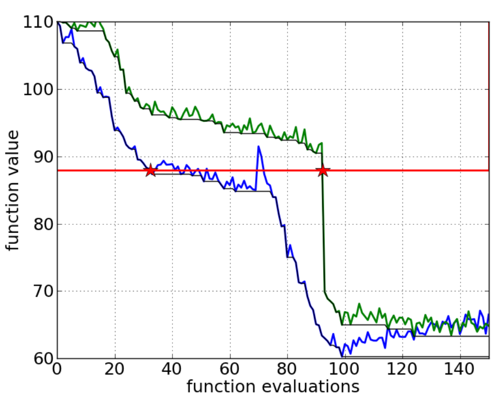
- a convergence graph
- first hitting time (black): lower envelope, a monotonous graph



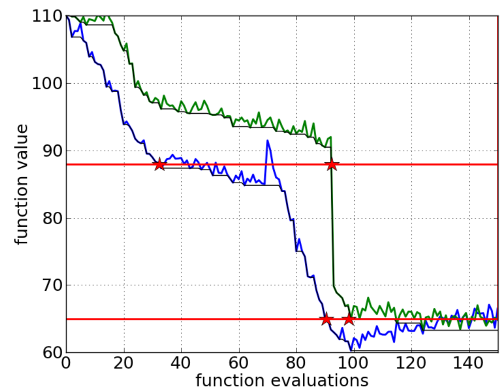
- another convergence graph



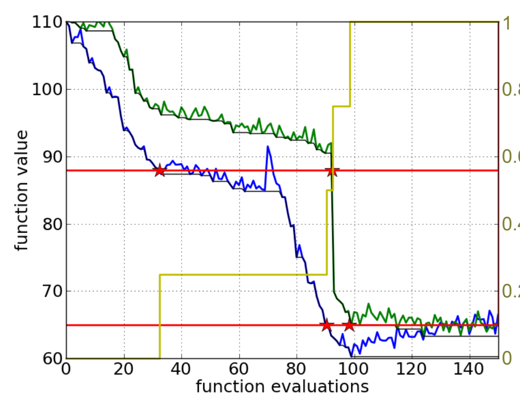
- another convergence graph with hitting time



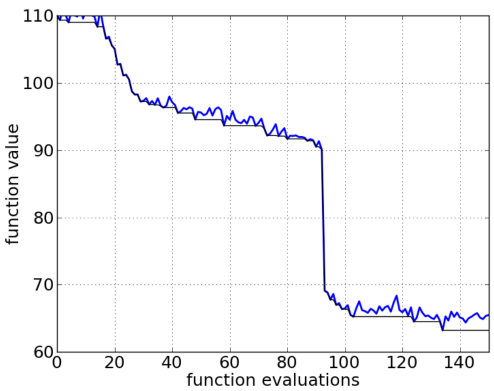
- a target value delivers two data points (possibly a missing value)



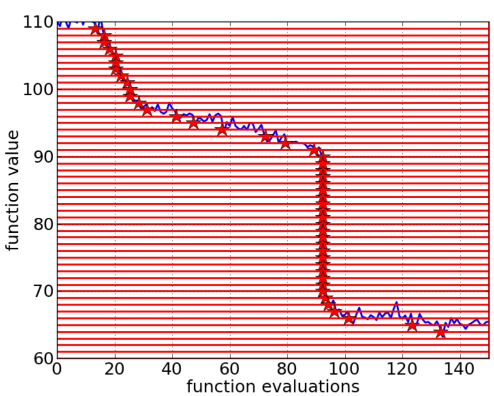
- a target value delivers two data points



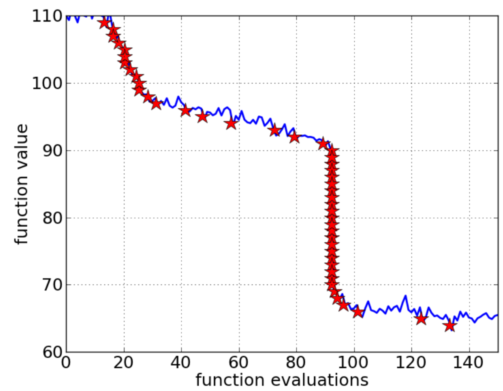
- the ECDF with four steps (between 0 and 1)

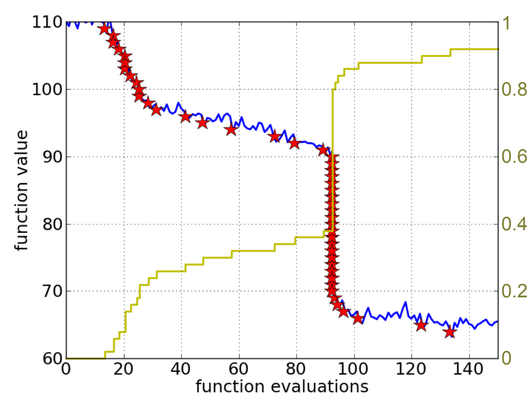


- reconstructing a single run

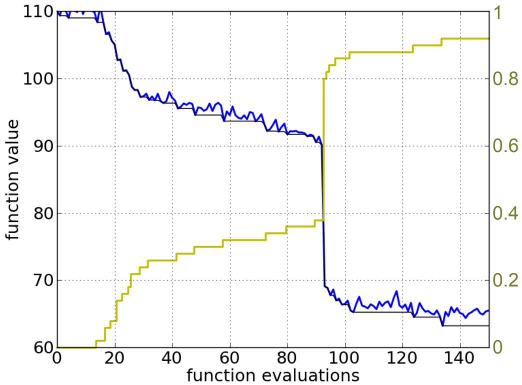


50 equally spaced targets

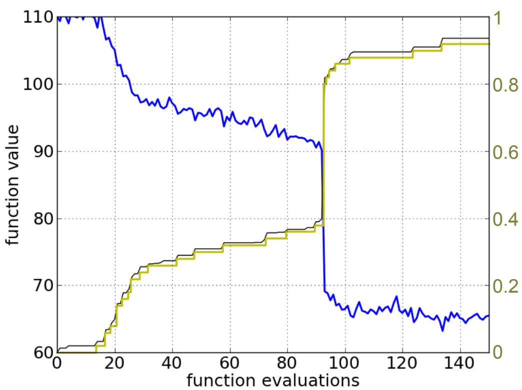




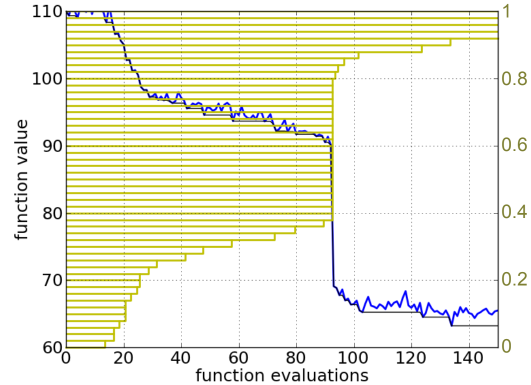
the ECDF recovers the monotonous graph



the ECDF recovers the monotonous graph, discretised and flipped



the ECDF recovers the monotonous graph, discretised and flipped



the ECDF recovers the monotonous graph, discretised and flipped

the area over the ECDF curve is the average runtime (the geometric average if the x-axis is in log scale)

Data and Performance Profiles

Benchmarking with COCO

COCO — Comparing Continuous Optimisers

- is a (software) platform for comparing continuous optimisers in a black-box scenario
  https://github.com/numbbo/coco
- *automatises* the tedious and repetitive task of *benchmarking numerical optimisation algorithms in a black-box setting*
- advantage: saves time and prevents common (and not so common) pitfalls

COCO provides

- experimental and measurement *methodology*
  main decision: what is the end point of measurement
- suites of benchmark functions
  single objective, bi-objective, noisy, constrained (in alpha stage)
- data of already benchmarked algorithms to compare with

COCO: Installation and Benchmarking in Python

```
$ ### get and install the code
$ git clone https://github.com/numbbo/coco.git  # get coco using git
$ cd coco
$ python do.py run-python  # install Python experimental module cocoex
$ python do.py install-postprocessing  # install post-processing :-)
```

```
import os, webbrowser
from scipy.optimize import fmin
import cocoex, cocopp

# prepare
output_folder = "scipy-optimize-fmin"
suite = cocoex.Suite("bbob", "", "")
observer = cocoex.Observer("bbob", "result_folder: " + output_folder)

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

# post-process and show data
cocopp.main(observer.result_folder)  # re-run folders look like "...-001" etc
webbrowser.open("file://" + os.getcwd() + "/ppdata/index.html")
```


Benchmark Functions

should be

- comprehensible
- difficult to defeat by “cheating”
  examples: optimum in zero, separable
- scalable with the input dimension
- reasonably quick to evaluate
  e.g. 12-36h for one full experiment
- reflect reality
  specifically, we model well-identified difficulties encountered also in real-world problems

The COCO Benchmarking Methodology

- budget-free
  larger budget means more data to investigate
  any budget is comparable
  termination and restarts are or become relevant
- using runtime as (almost) single performance measure
  measured in number of function evaluations
- runtimes are aggregated
  - in empirical (cumulative) distribution functions
  - by taking averages
    geometric average when aggregating over different problems

Benchmarking Results for Algorithm ALG on the bbob Suite

Home

Runtime distributions (ECDFs) per function

Runtime distributions (ECDFs) summary and function groups

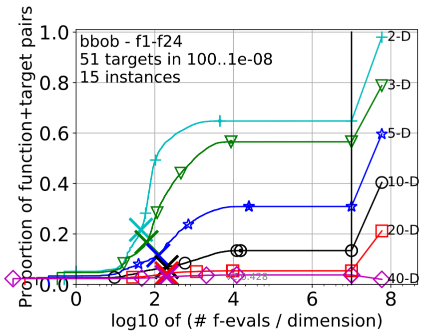
Scaling with dimension for selected targets

Tables for selected targets

Runtime distribution for selected targets and f-distributions

Runtime loss ratios

Runtime distributions (ECDFs) over all targets



FIN