Black-Box Optimization Benchmarking: Simulated Annealing

master AIC 2018/2019

Issa Hammoud Salim Tabarani Trung Vu-Thanh

Madji Youcef Yakoubi Yacine

ABSTRACT

The objective of our project is to implement simulated annealing algorithm proposed in the paper from Anton Dekkers and Emile Aarts: "Global optimization and simulated annealing". We use COCO platform to compare our results with standard optimizers and optimizers of other groups who run the same algorithm.

CCS CONCEPTS

• **Computing methodologies → Continuous space search**;

KEYWORDS

Benchmarking, Black-box optimization

ACM Reference format:

Issa Hammoud Salim Tabarani Trung Vu-Thanh Madji Youcef Yakoubi Yacine . 2018. Black-Box Optimization Benchmarking: Simulated Annealing. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[t]c@c Issa Hammoud tabular tabular[t]c@c Salim Tabarani tabular tabular[t]c@c Trung Vu-Thanh tabular tabular[t]c@c Madji Youcef tabular tabular[t]c@c Yakoubi Yacine tabular

1 INTRODUCTION

Global optimization issues emerge in numerous regions such as technical sciences and industrial engineering. The paper detailed a method which apply a probabilistic mechanism that permits search procedures to flee from local minima. In the followings, we'll show the concepts related to the paper. We'll depict the algorithm and our steps of implementation. Thanks to COCO, we'll talk about the results we gotten and look at different benchmarks given by the algorithm. We'll begin by presenting the two important concepts within the paper that are stochastic method and simulated annealing.

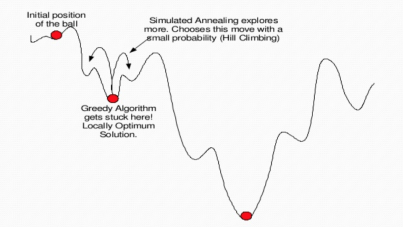
1.1 Stochastic method in global optimization

Global optimization algorithms are usually divided into two main classes, the deterministic and stochastic one. Deterministic methods provide a theoretical guarantee of locating the global minimizer, or at least an approximate global minimizer to within a prescribed tolerance, in a finite number of steps. It lead to larger computational burdens. In contrast, stochastic methods incorporate probabilistic (stochastic) elements, either in the problem data, for instance in the objective function and constraints, or in the algorithm itself, or in both. The convergence proofs for this type of methods involve the use of probability theory. Stochastic methods are usually faster in locating a global optimum than deterministic ones. Moreover, stochastic methods adapt better to black-box formulations and extremely ill-behaved functions. For these reasons. the paper focus on stochastic methods.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Conference'17, July 2017, Washington, DC, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Simulated annealing

Annealing is a process in metallurgy where metals are gradually cooled to form them reach a state of low energy where they are exceptionally solid. Simulated annealing is an analogous method for optimization. It is typically described in terms of thermodynamics. The random movement corresponds to high temperature; at low temperature, there is little randomness. Simulated annealing is a process where the temperature is reduced slowly, starting from a random search at high temperature eventually becoming pure greedy descent as it approaches zero temperature. The randomness should tend to jump out of local minima and find regions that have a low heuristic value; greedy descent will lead to local minima. At high temperatures, worsening steps are more likely than at lower temperatures.



In spite the fact that some algorithms can be surprisingly effective at finding a good solution, they also have a tendency to get stuck in local optimums, for example greedy or hill climber. As mentioned, the simulated annealing algorithm is excellent at avoiding this problem and is much better on average at finding an approximate global optimum. In the example above, We can see that these algorithms will simply accept neighbour solutions that are better than the current solution. When they can't find any better neighbours, it stops. We can clearly see that it's stuck in a local optimum. Simulated annealing works slightly differently than this and will occasionally accept worse solutions. This characteristic of simulated annealing helps it to jump out of any local optimums it might have.

2 ALGORITHM PRESENTATION

Simulated annealing is composed of 2 essential elements: cooling schedule and generations of random solution. Cooling schedule is a set of parameters must be specified that governs the convergence of the algorithm. We can consider that a finite-time implementation of the simulated annealing algorithm as generating homogeneous Markov chains of finite length at a finite sequence of descending values of the control parameter. The uniform distribution is chosen to generate a new point.

Algorithm 1 Procedure simulated annealing

```
begin
  "initialize(c,x)";
  stopcriterion := false
  while stopcriterion = false do
        begin
          for i := 1 to L do
              begin
                "generate y from x";
                if f(y) − f(x) ≤ 0
                then accept
                else if exp(−f(y) − f(x))/c) > random[0, 1)
                then accept;
                if accept then x:=y
              end;
              "lower c"
          end
        end
```

The algorithm looks quite simple. At first, we initialize generate a random solution x a value of control parameter c, called c_0. c_0 need to be sufficiently large, such that approximately all transitions are accepted at this value. With a value of control parameter, then three following steps are repeated number of L times: we generate a random neighboring solution, we calculate the new objective function value and we compare the old and new one. If $f_{new} < f_{old}$, we move to the new solution, otherwise we maybe move to the new solution in case the condition is satisfied. The the control parameter is lowed and a new loop is restarted if stop criterion is true, in other words, c is greater than a value c_{min}. c is regarded as the temperature in the method of heating and cooling metals. L is length of Markov chain of accepted solution points and L should be sufficiently large because simulated annealing does better when the neighbor comparing move process is carried about many times at each temperature.

3 DESCRIPTION OF THE IMPLEMENTATION

The programming language used is python, with the librairy numpy.

3.1 Initialisation of paramaters

```python
def generate_rand_vector(S):
    """
    generate a random vector x belonging to set S
    with a uniform distribution.
    """

    # generate a random value following a uniform
    # distribution over each dimension of S.
    x = [np.random.uniform(s[0],s[1]) for s in S]

    return x
```

The function receive argument S being numpy array of size nx2. The function return n random uniform value $(x_1, x_2, ..., x_n)$ which will be the input of objective function. The real function de initilize parameters of the algorithm is below:

```python
def init_c(f, S, m, ksi = 0.9):
    """
    This function initializes the value of the
    control parameter c.
    It returns the initial control paramter c0 and a
    realizable value x.
    f is the objective function.
    S is a 2D array representing the realizable values.
    m is the number of trials.
    ksi is the initial acceptance ratio.
    """

    # a list that should contain the positive
    # deltas (f(y) - f(x))
    delta = []

    # generate a value belonging to S
    x = generate_rand_vector(S)

    # m is the number of trials
    for i in range(m):

        # generate a value belonging to S
        y = generate_rand_vector(S)

        deltaf = f(y) - f(x)

        if(deltaf > 0):
            delta.append(deltaf)

        x = y

    delta = np.array(delta)

    # m2 is the number of trials with deltaf > 0
    m2 = len(delta)

    # m1 is the number of trials with deltaf <= 0
    m1 = m - m2

    delta_mean = np.mean(delta)

    # if m1 == m2, c0 will be infinity, and we are
    # interested to have m1 and m2 very near so c is high
```

```
    if(m1 == m2):
        m2+=1
        m1-=1

    # if m1 greater than m2, we have c negative, so we
    #inverse them

    if(m1 > m2):
        tmp = m1
        m1 = m2
        m2 = tmp


    c0 = delta_mean/np.log(m2/(m2*ksi + m1*(1 - ksi)))

    return c0
```

à écrire qqch

3.2 Algorithm core implementation

```
def SimulatedAnnealing(f,n,x):
    """
    This function calculate the global optimun of
    a function f defined over a set S, using the
    simulated annealing algorithm.
    """

    S = np.ones((n,2))

    S[:,0] = -100
    S[:,1] = 100

    # get c0 for m = 50
    c0 = init_c(f,S, 50)

    stopcriterion = False

    L0 = 20

    eps_s = 0.0001

    delta = 0.1

    c = c0

    # a list to stock the initial evaluations f(y's) for
    # the first Markov chain. It is used for the stop criterion.
    inital_Markov_chain_values = []

    inital_Markov_chain_values.append(f(x))

    # a boolean to indicate if the algorithm is in the first
    # Markov chain or not.
    first_chain = True

    previous_accepted_values = []


    while(stopcriterion == False):

        # list of accepted values for each round.
        accepted_values = []

    #         accepted_values.append(f(x))

        for i in range(n*L0):

            y = generate_rand_vector(S)

            # This condition is used when the value of c = 0,
            # because we can't divide by zero in the
            #exponential term.
            if(c == 0):
                acc = 0
            else :
                acc = np.exp(-(f(y) - f(x))/c)


            # acceptance criterion
            if(f(y) - f(x) <= 0 or acc > \
                      np.random.rand()):

                x = y

                accepted_values.append(f(y))

                # Save the first Markov chain
                if(first_chain == True):

                    inital_Markov_chain_values.append(f(y))

        # Test for the stop criterion if we are not in the
        #first Markov chain
        if(first_chain == False):

            # the term f(c0)(bar) in the paper
            f_0= np.mean(inital_Markov_chain_values)

            # check if there are accpeted values because
            # the np.mean will return nan in this case
            if(len(accepted_values) == 0):
                f_mean = 0
            else :
                f_mean = np.mean(accepted_values)

            # f_ is the term delta fs(c)/dc in the paper
            f_ = (f_mean - np.mean(previous_accepted_values)) \
            /(c - previous_c)

            # check stop criterion
            if(np.abs(f_ * c / f_0) < eps_s ):
                stopcriterion = True
```

```
        if(len(accepted_values) != 0):

            previous_accepted_values = accepted_values

            previous_c = c

        # check if there are accpeted values because
        # the np.std will return nan in this case
        if(len(accepted_values) <= 1 or np.std(accepted_values) \
                    == 0):
            c = 0
        else:
            c = c/(1 + (c*np.log(1+delta))/(3*np.std \
            (accepted_values)))


        # a boolean to indicate that the algo terminated the
        #first Markov chain
        first_chain = False



    return np.mean(previous_accepted_values)
```

4 CPU TIMING

In order to evaluate the CPU timing of the algorithm, we have run the MY-ALGORITHM-NAME on the bbob test suite [?] with restarts for a maximum budget equal to $400(D + 2)$ function evaluations according to [?]. The C/Java/Python/Matlab/Octave code was run on a Mac Intel(R) Core(TM) i5-2400S CPU @ 2.50GHz with 1 processor and 4 cores and (compile) options xxx. The time per function evaluation for dimensions 2, 3, 5, 10, 20, 40 equals $x.x$, $x.x$, $x.x$, xx, xxx, and xxx seconds respectively.

5 RESULTS

Results of scipy-optimize-fmin from experiments according to [?] and [?] on the benchmark functions given in [? ?] are presented in Figures 1, 2, 3, and 4 and in Tables ??. The experiments were performed with COCO [?], version 2.0, the plots were produced with version 2.0.

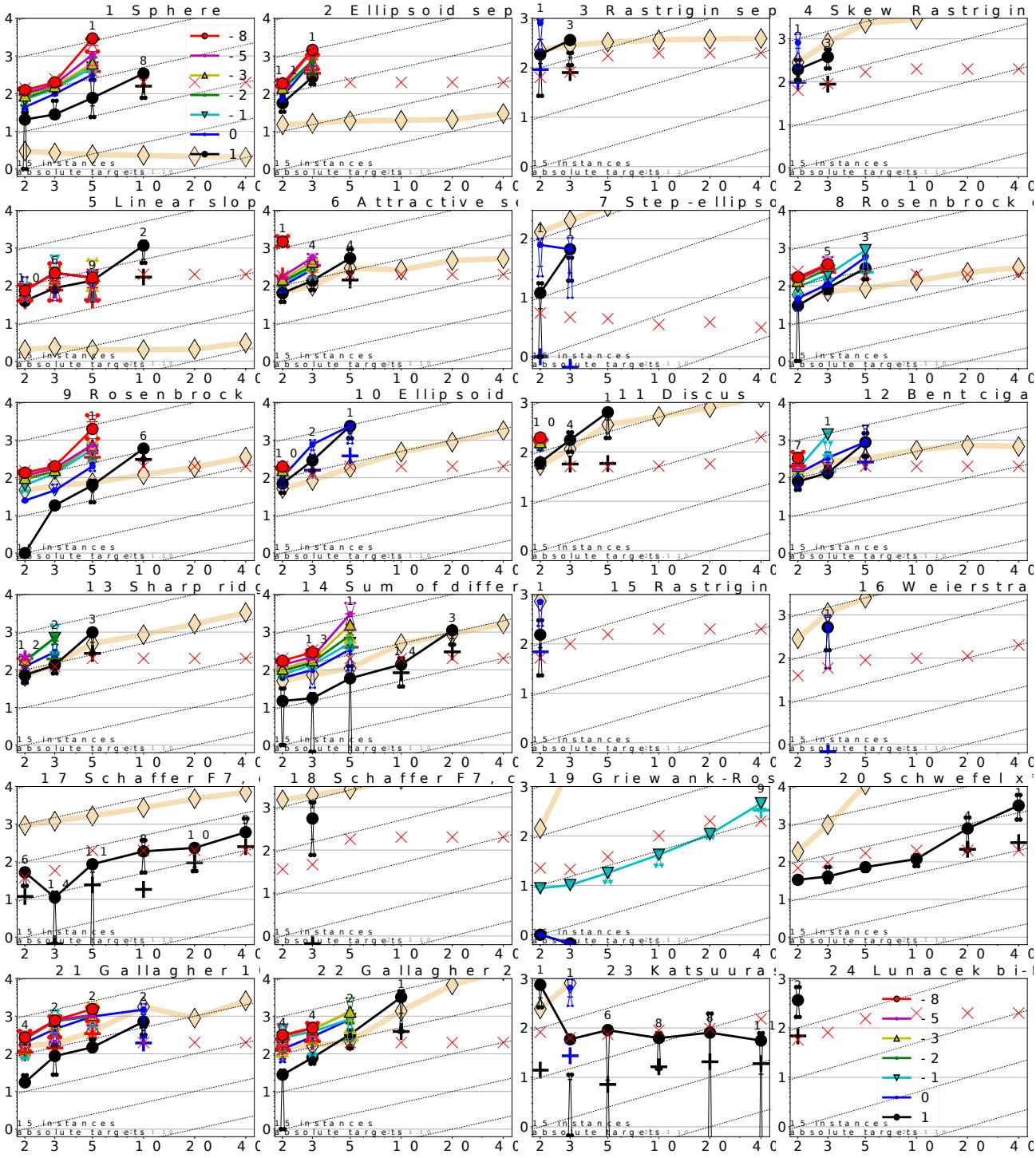


Figure 1: Scaling of runtime with dimension to reach certain target values Δf. Lines: average runtime (aRT); Cross (+): median runtime of successful runs to reach the most difficult target that was reached at least once (but not always); Cross (×): maximum number of f-evaluations in any trial. Notched boxes: interquartile range with median of simulated runs; All values are divided by dimension and plotted as $\log_{10}$ values versus dimension. Shown is the aRT for fixed values of $\Delta f = 10^k$ with k given in the legend. Numbers above aRT-symbols (if appearing) indicate the number of trials reaching the respective target. The light thick line with diamonds indicates the best algorithm from BBOB 2009 for the most difficult target. Horizontal lines mean linear scaling, slanted grid lines depict quadratic scaling.

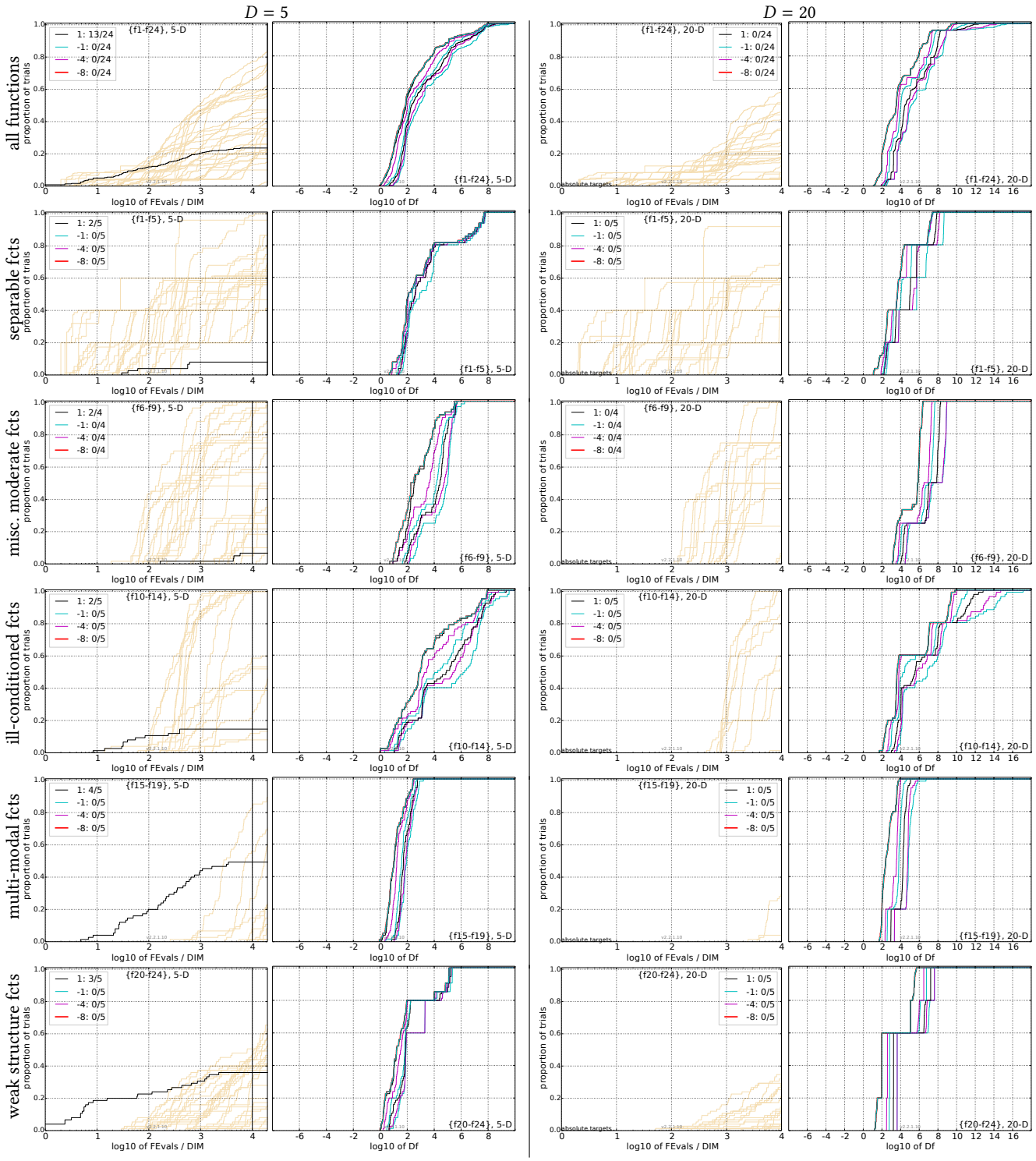


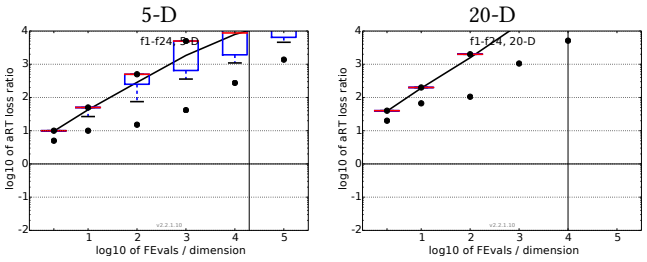
Figure 2: Empirical cumulative distribution functions (ECDF), plotting the fraction of trials with an outcome not larger than the respective value on the x-axis. Left subplots: ECDF of the number of function evaluations (FEvals) divided by search space dimension D, to fall below $f_{\text{opt}} + \Delta f$ with $\Delta f = 10^k$, where k is the first value in the legend. The thick red line represents the most difficult target value $f_{\text{opt}} + 10^{-8}$. Legends indicate for each target the number of functions that were solved in at least one trial within the displayed budget. Right subplots: ECDF of the best achieved Δf for running times of $0.5D, 1.2D, 3D, 10D, 100D, 1000D, \ldots$ function evaluations (from right to left cycling cyan-magenta-black…) and final Δf-value (red), where Δf and Df denote the difference to the optimal function value. Light brown lines in the background show ECDFs for the most difficult target of all algorithms benchmarked during BBOB-2009.

Δf	1e+1	1e+0	1e-1	1e-2	1e-3	1e-5	1e-7	#succ
f_1	11	12	12	12	12	12	12	15/15
	12610(17078)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_2	83	87	88	89	90	92	94	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_3	716	1622	1637	1642	1646	1650	1654	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_4	809	1633	1688	1758	1817	1886	1903	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_5	10	10	10	10	10	10	10	15/15
	37649(36269)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_6	114	214	281	404	580	1038	1332	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_7	24	324	1171	1451	1572	1572	1597	15/15
	14045(8129)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_8	73	273	336	372	391	410	422	15/15
	4808(3827)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_9	35	127	214	263	300	335	369	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{10}	349	500	574	607	626	829	880	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{11}	143	202	763	977	1177	1467	1673	15/15
	4919(8671)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{12}	108	268	371	413	461	1303	1494	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{13}	132	195	250	319	1310	1752	2255	15/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{14}	10	41	58	90	139	251	476	15/15
	2552(2551)	17112(25355)	∞	∞	∞	∞	∞*5e4*	0/15
f_{15}	511	9310	19369	19743	20073	20769	21359	14/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{16}	120	612	2662	10163	10449	11644	12095	15/15
	15(15)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{17}	5.0	215	899	2861	3669	6351	7934	15/15
	3777(15038)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{18}	103	378	3968	8451	9280	10905	12469	15/15
	3322(3290)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{19}	1	1	242	1.0e5	1.2e5	1.2e5	1.2e5	15/15
	35832(87736)	7.5e5(1e6)	∞	∞	∞	∞	∞*5e4*	0/15
f_{20}	16	851	38111	51362	54470	54861	55313	14/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{21}	41	1157	1674	1692	1705	1729	1757	14/15
	1129(2424)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{22}	71	386	938	980	1008	1040	1068	14/15
	2030(1446)	∞	∞	∞	∞	∞	∞*5e4*	0/15
f_{23}	3.0	518	14249	27890	31654	33030	34256	15/15
	8.0(6)	1358(1499)	∞	∞	∞	∞	∞*5e4*	0/15
f_{24}	1622	2.2e5	6.4e6	9.6e6	9.6e6	1.3e7	1.3e7	3/15
	∞	∞	∞	∞	∞	∞	∞*5e4*	0/15

Table 1: Average running time (aRT in number of function evaluations) divided by the aRT of the best algorithm from BBOB 2009 in dimension 5. The aRT and in braces, as dispersion measure, the half difference between 90 and 10%-tile of bootstrapped run lengths appear in the second row of each cell, the best aRT (preceded by the target Δf-value in *italics*) in the first. #succ is the number of trials that reached the target value of the last column. The median number of conducted function evaluations is additionally given in *italics*, if the target in the last column was never reached. Bold entries are statistically significantly better (according to the rank-sum test) compared to the best algorithm from BBOB 2009, with $p = 0.05$ or $p = 10^{-k}$ when the number $k > 1$ is following the ↓ symbol, with Bonferroni correction by the number of functions (24).
Data produced with COCO v2.2.1.10

Δf	1e+1	1e+0	1e-1	1e-2	1e-3	1e-5	1e-7	#succ
f_1	43	43	43	43	43	43	43	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_2	385	386	387	388	390	391	393	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_3	5066	7626	7635	7637	7643	7646	7651	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_4	4722	7628	7666	7686	7700	7758	1.4e5	9/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_5	41	41	41	41	41	41	41	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_6	1296	2343	3413	4255	5220	6728	8409	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_7	1351	4274	9503	16523	16524	16524	16969	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_8	2039	3871	4040	4148	4219	4371	4484	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_9	1716	3102	3277	3379	3455	3594	3727	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{10}	7413	8661	10735	13641	14920	17073	17476	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{11}	1002	2228	6278	8586	9762	12285	14831	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{12}	1042	1938	2740	3156	4140	12407	13827	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{13}	652	2021	2751	3507	18749	24455	30201	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{14}	75	239	304	451	932	1648	15661	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{15}	30378	1.5e5	3.1e5	3.2e5	3.2e5	4.5e5	4.6e5	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{16}	1384	27265	77015	1.4e5	1.9e5	2.0e5	2.2e5	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{17}	63	1030	4005	12242	30677	56288	80472	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{18}	621	3972	19561	28555	67569	1.3e5	1.5e5	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{19}	1	1	3.4e5	4.7e6	6.2e6	6.7e6	6.7e6	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{20}	82	46150	3.1e6	5.5e6	5.5e6	5.6e6	5.6e6	14/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{21}	561	6541	14103	14318	14643	15567	17589	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{22}	467	5580	23491	24163	24948	26847	1.3e5	12/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{23}	3.0	1614	67457	3.7e5	4.9e5	8.1e5	8.4e5	15/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15
f_{24}	1.3e6	7.5e6	5.2e7	5.2e7	5.2e7	5.2e7	5.2e7	3/15
	∞	∞	∞	∞	∞	∞	∞*2e5*	0/15

Table 2: Average running time (aRT in number of function evaluations) divided by the aRT of the best algorithm from BBOB 2009 in dimension 20. The aRT and in braces, as dispersion measure, the half difference between 90 and 10%-tile of bootstrapped run lengths appear in the second row of each cell, the best aRT (preceded by the target Δf-value in *italics*) in the first. #succ is the number of trials that reached the target value of the last column. The median number of conducted function evaluations is additionally given in *italics*, if the target in the last column was never reached. Bold entries are statistically significantly better (according to the rank-sum test) compared to the best algorithm from BBOB 2009, with $p = 0.05$ or $p = 10^{-k}$ when the number $k > 1$ is following the ↓ symbol, with Bonferroni correction by the number of functions (24).
Data produced with COCO v2.2.1.10



f_1–f_{24} in 5-D, maxFE/D=19630						
#FEs/D	best	10%	25%	**med**	75%	90%
2	5.0	10	10	10	10	10
10	10	17	50	50	50	50
100	15	67	2.5e2	5.0e2	5.0e2	5.0e2
1e3	42	3.1e2	5.8e2	5.0e3	5.0e3	5.0e3
1e4	2.7e2	1.0e3	1.9e3	8.8e3	5.0e4	5.0e4
1e5	1.4e3	4.3e3	6.2e3	1.8e4	4.6e4	5.0e5
RL_{US}/D	1e4	1e4	1e4	1e4	1e4	1e4

f_1–f_{24} in 20-D, maxFE/D=10005						
#FEs/D	best	10%	25%	**med**	75%	90%
2	20	40	40	40	40	40
10	67	2.0e2	2.0e2	2.0e2	2.0e2	2.0e2
100	1.1e2	1.8e3	2.0e3	2.0e3	2.0e3	2.0e3
1e3	1.1e3	1.8e4	2.0e4	2.0e4	2.0e4	2.0e4
1e4	5.1e3	6.1e4	2.0e5	2.0e5	2.0e5	2.0e5
1e5	3.1e4	8.0e4	1.3e6	2.0e6	2.0e6	2.0e6
RL_{US}/D	1e4	1e4	1e4	1e4	1e4	1e4

Figure 3: aRT **loss ratio versus the budget in number of f-evaluations divided by dimension. For each given budget** FEvals, **the target value f_t is computed as the best target f-value reached within the budget by the given algorithm. Shown is then the** aRT **to reach f_t for the given algorithm or the budget, if the best algorithm from BBOB 2009 reached a better target within the budget, divided by the** aRT **of the best algorithm from BBOB 2009 to reach f_t. Line: geometric mean. Box-Whisker error bar: 25-75%-ile with median (box), 10-90%-ile (caps), and minimum and maximum** aRT **loss ratio (points). The vertical line gives the maximal number of function evaluations in a single trial in this function subset. See also Figure 4 for results on each function subgroup.**
Data produced with COCO v2.2.1.10

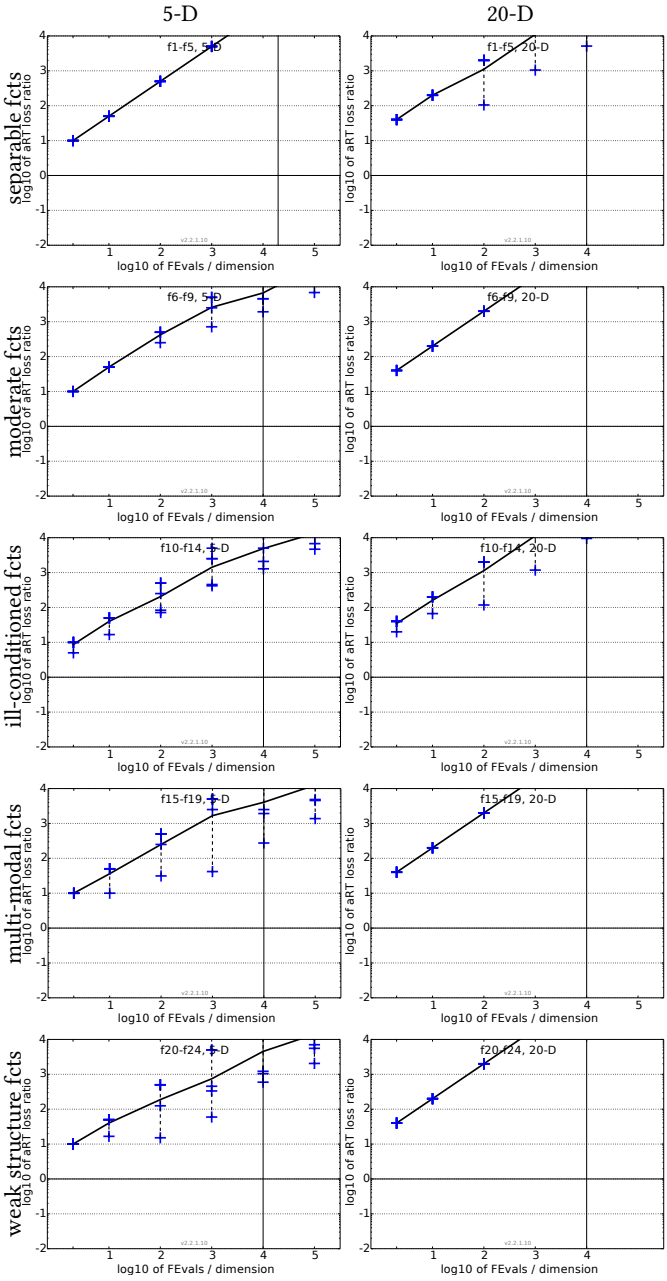


Figure 4: aRT **loss ratios (see Figure 3 for details). Each cross (+) represents a single function, the line is the geometric mean.**