

# 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

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).

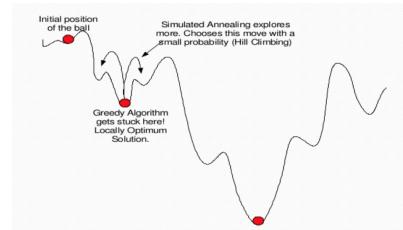
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

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.

### 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. In simulated annealing, we keep the temperature to simulate the thermal process. When the temperature is high, the algorithm allows to receive the worse value than current value. That's what makes the algorithm capable of jumping out of the local optimum in the early stages of the search process. When the temperature decreases, the ability to accept the bad value lower, so the algorithm can focus on one region that hopefully, is closed to optima solution. This makes the algorithm becomes effective when faced with the problem with numerous local optima.



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 optima, 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 optima 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
  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

---

```

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  $(x_1, x_2, \dots, x_n)$  which will be the input of objective function. The real function de initilize parameters of the algorithm is below:

---

```

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 optimum 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.
    initial_Markov_chain_values = []

    initial_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):
        initial_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(initial_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 accepted 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**.

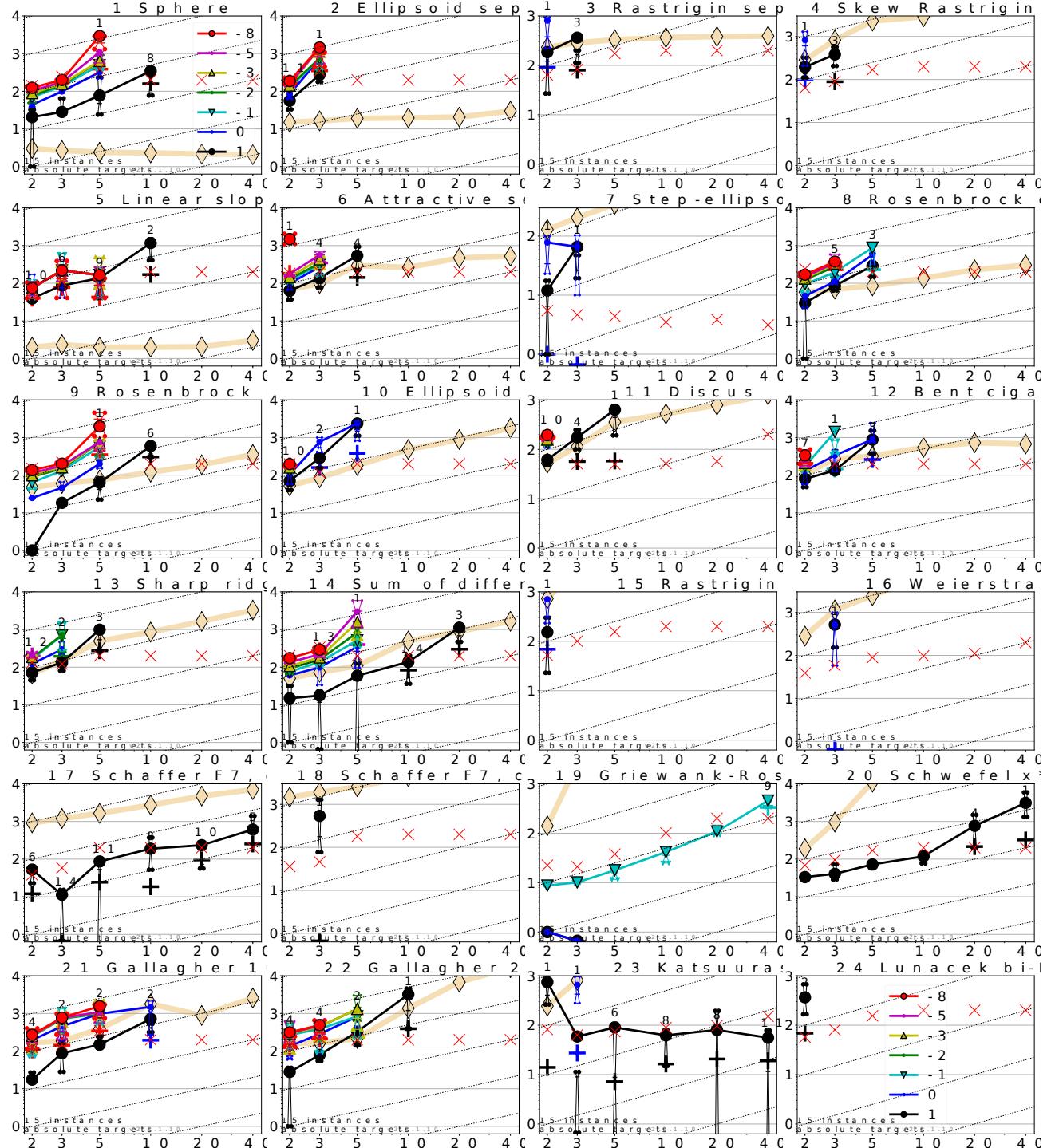
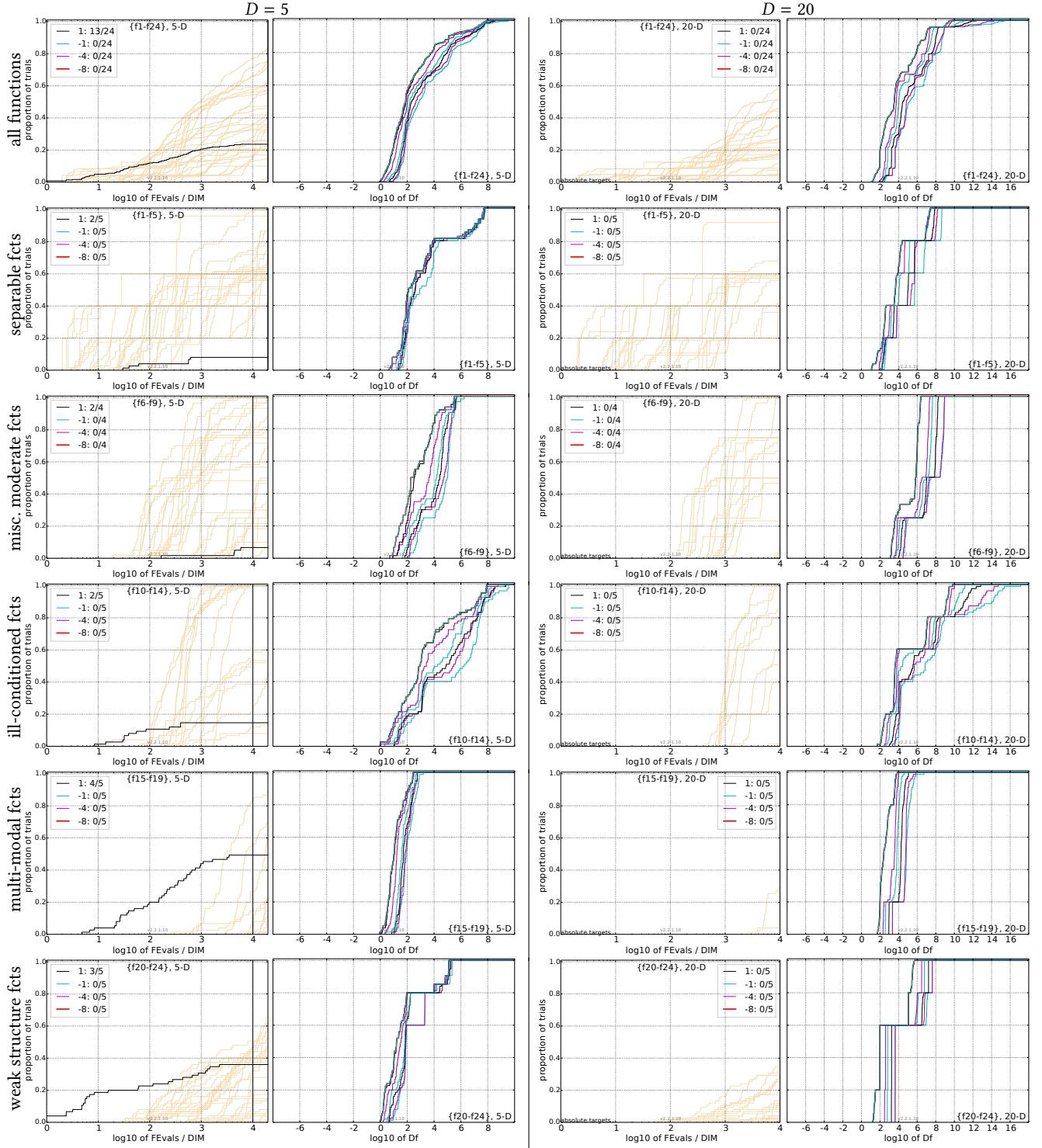


Figure 1: Scaling of runtime with dimension to reach certain target values  $\Delta f$ . Lines: average runtime (aRT); Cross (+): median runtime of successful runs to reach the most difficult target that was reached at least once (but not always); Cross (X): maximum number of  $f$ -evaluations in any 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  $\Delta f$  for running times of  $0.5D, 1.2D, 3D, 10D, 100D, 1000D, \dots$  function evaluations (from right to left cycling cyan-magenta-black...) and final  $\Delta f$ -value (red), where  $\Delta f$  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.

| $\Delta f$            | 1e+1               | 1e+0      | 1e-1      | 1e-2      | 1e-3      | 1e-5      | 1e-7      | #succ |
|-----------------------|--------------------|-----------|-----------|-----------|-----------|-----------|-----------|-------|
| <b>f<sub>1</sub></b>  | 11<br>12610(17078) | 12<br>∞   | 12<br>∞   | 12<br>∞   | 12<br>∞   | 12<br>∞   | 12<br>∞   | 15/15 |
| <b>f<sub>2</sub></b>  | 83<br>∞            | 87<br>∞   | 88<br>∞   | 89<br>∞   | 90<br>∞   | 92<br>∞   | 94<br>∞   | 15/15 |
| <b>f<sub>3</sub></b>  | 716<br>∞           | 1622<br>∞ | 1637<br>∞ | 1642<br>∞ | 1646<br>∞ | 1650<br>∞ | 1654<br>∞ | 15/15 |
| <b>f<sub>4</sub></b>  | 809<br>∞           | 1633<br>∞ | 1688<br>∞ | 1758<br>∞ | 1817<br>∞ | 1886<br>∞ | 1903<br>∞ | 15/15 |
| <b>f<sub>5</sub></b>  | 10<br>37649(36269) | 10<br>∞   | 10<br>∞   | 10<br>∞   | 10<br>∞   | 10<br>∞   | 10<br>∞   | 15/15 |
| <b>f<sub>6</sub></b>  | 114<br>∞           | 214<br>∞  | 281<br>∞  | 404<br>∞  | 580<br>∞  | 1038<br>∞ | 1332<br>∞ | 15/15 |
| <b>f<sub>7</sub></b>  | 24<br>14045(8129)  | 324<br>∞  | 1171<br>∞ | 1451<br>∞ | 1572<br>∞ | 1572<br>∞ | 1597<br>∞ | 15/15 |
| <b>f<sub>8</sub></b>  | 73<br>4808(3827)   | 273<br>∞  | 336<br>∞  | 372<br>∞  | 391<br>∞  | 410<br>∞  | 422<br>∞  | 15/15 |
| <b>f<sub>9</sub></b>  | 35<br>∞            | 127<br>∞  | 214<br>∞  | 263<br>∞  | 300<br>∞  | 335<br>∞  | 369<br>∞  | 15/15 |
| <b>f<sub>10</sub></b> | 349<br>∞           | 500<br>∞  | 574<br>∞  | 607<br>∞  | 626<br>∞  | 829<br>∞  | 880<br>∞  | 15/15 |
| <b>f<sub>11</sub></b> | 143<br>4919(8671)  | 202<br>∞  | 763<br>∞  | 977<br>∞  | 1177<br>∞ | 1467<br>∞ | 1673<br>∞ | 15/15 |
| <b>f<sub>12</sub></b> | 108<br>∞           | 268<br>∞  | 371<br>∞  | 413<br>∞  | 461<br>∞  | 1303<br>∞ | 1494<br>∞ | 15/15 |

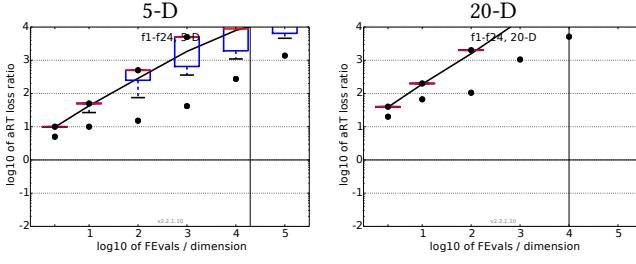
| $\Delta f$            | 1e+1               | 1e+0               | 1e-1           | 1e-2           | 1e-3           | 1e-5           | 1e-7           | #succ |
|-----------------------|--------------------|--------------------|----------------|----------------|----------------|----------------|----------------|-------|
| <b>f<sub>13</sub></b> | 132<br>∞           | 195<br>∞           | 250<br>∞       | 319<br>∞       | 1310<br>∞      | 1752<br>∞      | 2255<br>∞      | 15/15 |
| <b>f<sub>14</sub></b> | 10<br>2552(2551)   | 41<br>17112(25355) | 58<br>∞        | 90<br>∞        | 139<br>∞       | 251<br>∞       | 476<br>∞       | 15/15 |
| <b>f<sub>15</sub></b> | 511<br>∞           | 9310<br>∞          | 19369<br>∞     | 19743<br>∞     | 20073<br>∞     | 20769<br>∞     | 21359<br>∞     | 14/15 |
| <b>f<sub>16</sub></b> | 120<br>15(15)      | 612<br>∞           | 2662<br>∞      | 10163<br>∞     | 10449<br>∞     | 11644<br>∞     | 12095<br>∞     | 15/15 |
| <b>f<sub>17</sub></b> | 5.0<br>3777(15038) | 215<br>∞           | 899<br>∞       | 2861<br>∞      | 3669<br>∞      | 6351<br>∞      | 7934<br>∞      | 15/15 |
| <b>f<sub>18</sub></b> | 103<br>3322(3290)  | 378<br>∞           | 3968<br>∞      | 8451<br>∞      | 9280<br>∞      | 10905<br>∞     | 12469<br>∞     | 15/15 |
| <b>f<sub>19</sub></b> | 1<br>35832(87736)  | 1<br>7.5e5(1e6)    | 242<br>∞       | 1.0e5<br>∞     | 1.2e5<br>∞     | 1.2e5<br>∞     | 1.2e5<br>∞     | 15/15 |
| <b>f<sub>20</sub></b> | 16<br>∞            | 851<br>∞           | 38111<br>51362 | 54470<br>54861 | 55313<br>55313 | 54470<br>54861 | 55313<br>55313 | 14/15 |
| <b>f<sub>21</sub></b> | 41<br>1129(2424)   | 1157<br>∞          | 1674<br>∞      | 1692<br>∞      | 1705<br>∞      | 1720<br>∞      | 1757<br>∞      | 14/15 |
| <b>f<sub>22</sub></b> | 71<br>2030(1446)   | 386<br>∞           | 938<br>∞       | 980<br>∞       | 1008<br>∞      | 1040<br>∞      | 1068<br>∞      | 14/15 |
| <b>f<sub>23</sub></b> | 3.0<br>8.0(6)      | 518<br>1358(1499)  | 14249<br>∞     | 27890<br>∞     | 31654<br>∞     | 33030<br>∞     | 34256<br>∞     | 15/15 |
| <b>f<sub>24</sub></b> | 1622<br>∞          | 2.2e5<br>∞         | 6.4e6<br>∞     | 9.6e6<br>∞     | 9.6e6<br>∞     | 1.3e7<br>∞     | 1.3e7<br>∞     | 3/15  |

**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  $\Delta f$ -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

| $\Delta f$            | 1e+1      | 1e+0      | 1e-1       | 1e-2       | 1e-3       | 1e-5       | 1e-7       | #succ |
|-----------------------|-----------|-----------|------------|------------|------------|------------|------------|-------|
| <b>f<sub>1</sub></b>  | 43<br>∞   | 43<br>∞   | 43<br>∞    | 43<br>∞    | 43<br>∞    | 43<br>∞    | 43<br>∞    | 15/15 |
| <b>f<sub>2</sub></b>  | 385<br>∞  | 386<br>∞  | 387<br>∞   | 388<br>∞   | 390<br>∞   | 391<br>∞   | 393<br>∞   | 15/15 |
| <b>f<sub>3</sub></b>  | 5066<br>∞ | 7626<br>∞ | 7635<br>∞  | 7637<br>∞  | 7643<br>∞  | 7646<br>∞  | 7651<br>∞  | 15/15 |
| <b>f<sub>4</sub></b>  | 4722<br>∞ | 7628<br>∞ | 7666<br>∞  | 7686<br>∞  | 7700<br>∞  | 7758<br>∞  | 1.4e5<br>∞ | 9/15  |
| <b>f<sub>5</sub></b>  | 41<br>∞   | 41<br>∞   | 41<br>∞    | 41<br>∞    | 41<br>∞    | 41<br>∞    | 41<br>∞    | 15/15 |
| <b>f<sub>6</sub></b>  | 1296<br>∞ | 2343<br>∞ | 3413<br>∞  | 4255<br>∞  | 5220<br>∞  | 6728<br>∞  | 8409<br>∞  | 15/15 |
| <b>f<sub>7</sub></b>  | 1351<br>∞ | 4274<br>∞ | 9503<br>∞  | 16523<br>∞ | 16524<br>∞ | 16524<br>∞ | 16969<br>∞ | 15/15 |
| <b>f<sub>8</sub></b>  | 2039<br>∞ | 3871<br>∞ | 4040<br>∞  | 4148<br>∞  | 4219<br>∞  | 4371<br>∞  | 4484<br>∞  | 15/15 |
| <b>f<sub>9</sub></b>  | 1716<br>∞ | 3102<br>∞ | 3277<br>∞  | 3379<br>∞  | 3455<br>∞  | 3594<br>∞  | 3727<br>∞  | 15/15 |
| <b>f<sub>10</sub></b> | 7413<br>∞ | 8661<br>∞ | 10735<br>∞ | 13641<br>∞ | 14920<br>∞ | 17073<br>∞ | 17476<br>∞ | 15/15 |
| <b>f<sub>11</sub></b> | 1002<br>∞ | 2228<br>∞ | 6278<br>∞  | 8586<br>∞  | 9762<br>∞  | 12285<br>∞ | 14831<br>∞ | 15/15 |
| <b>f<sub>12</sub></b> | 1042<br>∞ | 1938<br>∞ | 2740<br>∞  | 3156<br>∞  | 4140<br>∞  | 12407<br>∞ | 13827<br>∞ | 15/15 |

| $\Delta f$            | 1e+1       | 1e+0       | 1e-1       | 1e-2       | 1e-3       | 1e-5       | 1e-7       | #succ |
|-----------------------|------------|------------|------------|------------|------------|------------|------------|-------|
| <b>f<sub>13</sub></b> | 652<br>∞   | 2021<br>∞  | 2751<br>∞  | 3507<br>∞  | 18749<br>∞ | 24455<br>∞ | 30201<br>∞ | 15/15 |
| <b>f<sub>14</sub></b> | 75<br>∞    | 239<br>∞   | 304<br>∞   | 451<br>∞   | 932<br>∞   | 1648<br>∞  | 15661<br>∞ | 15/15 |
| <b>f<sub>15</sub></b> | 30378<br>∞ | 1.5e5<br>∞ | 3.1e5<br>∞ | 3.2e5<br>∞ | 3.2e5<br>∞ | 4.5e5<br>∞ | 4.6e5<br>∞ | 15/15 |
| <b>f<sub>16</sub></b> | 1384<br>∞  | 27265<br>∞ | 77015<br>∞ | 1.4e5<br>∞ | 1.9e5<br>∞ | 2.0e5<br>∞ | 2.2e5<br>∞ | 15/15 |
| <b>f<sub>17</sub></b> | 63<br>∞    | 1030<br>∞  | 4005<br>∞  | 12242<br>∞ | 30677<br>∞ | 56288<br>∞ | 80472<br>∞ | 15/15 |
| <b>f<sub>18</sub></b> | 621<br>∞   | 3972<br>∞  | 19561<br>∞ | 28555<br>∞ | 67569<br>∞ | 1.3e5<br>∞ | 1.5e5<br>∞ | 15/15 |
| <b>f<sub>19</sub></b> | 1<br>∞     | 1<br>∞     | 3.4e5<br>∞ | 4.7e6<br>∞ | 6.2e6<br>∞ | 6.7e6<br>∞ | 6.7e6<br>∞ | 15/15 |
| <b>f<sub>20</sub></b> | 82<br>∞    | 46150<br>∞ | 3.1e6<br>∞ | 5.5e6<br>∞ | 5.5e6<br>∞ | 5.6e6<br>∞ | 5.6e6<br>∞ | 14/15 |
| <b>f<sub>21</sub></b> | 561<br>∞   | 6541<br>∞  | 14103<br>∞ | 14318<br>∞ | 14643<br>∞ | 15567<br>∞ | 17589<br>∞ | 15/15 |
| <b>f<sub>22</sub></b> | 467<br>∞   | 5580<br>∞  | 23491<br>∞ | 24163<br>∞ | 24948<br>∞ | 26847<br>∞ | 1.3e5<br>∞ | 12/15 |
| <b>f<sub>23</sub></b> | 3.0<br>∞   | 1614<br>∞  | 67457<br>∞ | 3.7e5<br>∞ | 4.9e5<br>∞ | 8.1e5<br>∞ | 8.4e5<br>∞ | 15/15 |
| <b>f<sub>24</sub></b> | 1.3e6<br>∞ | 7.5e6<br>∞ | 5.2e7<br>∞ | 5.2e7<br>∞ | 5.2e7<br>∞ | 5.2e7<br>∞ | 5.2e7<br>∞ | 3/15  |

**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  $\Delta f$ -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



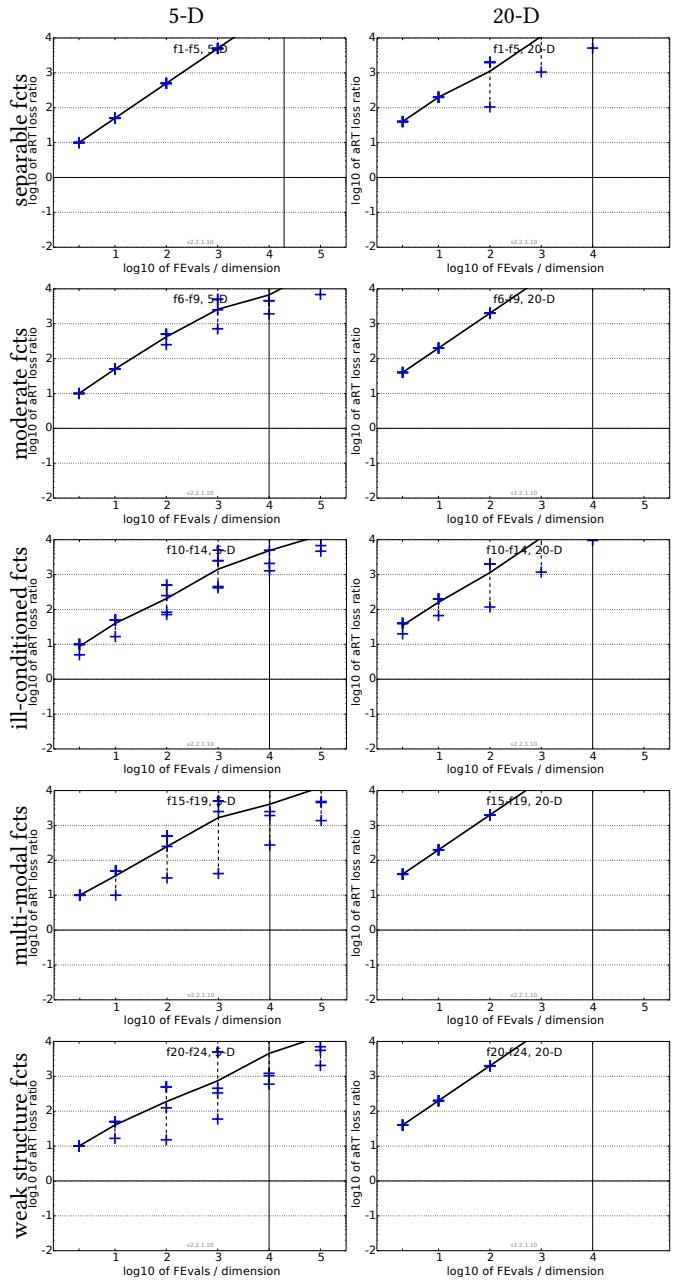
| #FEs/D | <i>f1-f24</i> in 5-D, maxFE/D=19630 |       |       |            |       |       |
|--------|-------------------------------------|-------|-------|------------|-------|-------|
|        | best                                | 10%   | 25%   | <b>med</b> | 75%   | 90%   |
| 2      | 5.0                                 | 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 |
| RLUS/D | 1e4                                 | 1e4   | 1e4   | 1e4        | 1e4   | 1e4   |

| #FEs/D | <i>f1-f24</i> in 20-D, maxFE/D=10005 |       |       |            |       |       |
|--------|--------------------------------------|-------|-------|------------|-------|-------|
|        | best                                 | 10%   | 25%   | <b>med</b> | 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 |
| RLUS/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.