Benchmarking the R2-EMOA for the Bi-Objective bbob-biobj Test Suite

Vincent Boyer, Ludovic Shiro-Kun, Guillaume Collin, Mariem Bouhaha

ABSTRACT

The aim behind this project is to contribute to the numerical benchmarking of a blackbox optimization algorithm via the COCO platform. In this study, we investigate results of an Evolutionary Multiobjective Optimisation Algorithm with *R2* indicator-based selection. We thus want to evaluate the ability of the *R2*-EMOA to converge to the optimal μ-distributions of the *R2* indicator, on problems wih different Pareto Front shapes.
We implemented the *R2*-EMOA with uniform weights and normalization of the objective space to garantee that values obtained per objective always lie in [0,1]. We used python as programming language and run it for the *bbob-biobj* test suite. We did not reach the number of function evaluations as described in the article for all function groups, due to the very high running time of the algorithm. However, we could run the algorithm using a budget of 200k evaluations for the first function group (seperable-seperable), only in 2D, to assess it's performance. We also used data of the NSGA-II and the Random Search as baseline algorithms for comparison.
On the basis of the obtained results, we noticed that, when sufficiently run, the R2-EMOA can approximate the optimal (10-)distibution regarding the R2 indicator after at least 150k evaluations. For a limited number of function evaluations, the algorithm showed a good performance for some test functions and poorer performance for others.

KEYWORDS

Benchmarking, Black-box optimization, Bi-objective optimization, R2 indicator

ACM Reference format:
Vincent Boyer, Ludovic Shiro-Kun, Guillaume Collin, Mariem Bouhaha . 2017. Benchmarking the R2-EMOA for the Bi-Objective bbob-biobj Test Suite. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 13 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Many fields of study, including economics, electric power systems and finance, often involve making decisions based on multiple criteria that have to be minimized (or maximized) simultaneously. This class of problems is called multi-objective optimization.

In mathematical terms, a multi-objective optimization problem can be formulated as:

$$\min f_1(x), f_2(x), ..., f_m(x), x \in X$$

where the integer $m \geq 2$ is the number of objectives and the set X is the feasible set of decision vectors.

The goal is to approximate the Pareto front by producing a set of non-dominated solutions. A solution $x \in X$ is called *non-dominated* when none of the objective functions can be improved in value without degrading some of the other objective values, ie $\forall y \in X, \forall i \in [\![1, ..., m]\!] : f_i(x) \leq f_i(y)$.

Some of the solvers for multiobjective problems are classified as Evolutionary Multiobjective Optimization Algorithms (EMOAs). These algorithms iteratively produce a set of Pareto optimal solutions by selecting some solutions with regard to a specific indicator among a population (randomized at the first step) and generating their "children" at each step.

In this paper, we will focus on an EMOA algorithm based on the R2 indicator. This indicator is not very documented, despite its similarities with the HyperVolume (HV) indicator which has been studied a lot more in the last years. Our work will be mostly based on [2] that presents a unary R2 indicator and investigates its properties and its differences to the HV indicator.

The main goal of this paper is to perform numerical benchmarking on the R2-EMOA algorithm in order to assess its behaviour and performance on some well-defined problems.

We will implement the algorithm using Python combined with the library Numpy. As we will describe in details, we will use uniform weights and normalize the objective space such that the minimum and maximum values obtained per objective always lie in [0,1] in order to facilitate the comparison of the objective values at each step of the algorithm.

In order to assess its performance, advantages and drawbacks, we will run this algorithm on a suite of well-known test functions and compare its results to those of two baseline algorithms: NGSA-II and a random search within $[-5, 5]^n$.

For this purpose, we will use the COCO platform, an open source application developed by Inria from 2007, that enables automated benchmarking tests.

2 ALGORITHM PRESENTATION

In the following we present the pseudo code of the R2-EMOA algorithm that we implemented and we will describe the related procedures in the following sub-sections.

We will start by explaining the rationale behind the SBX crossover and the gaussian mutation, used as variation operators for our population, to generate the offspring (line 5. in Algorithm 1) with some parameters that we mention later. Then, we present the non-dominated sorting and finally the procedure that computes the R2 indicator.

Conference'17, Washington, DC, USA

DOI: 10.475/123_4

Algorithm 1 Optimization algorithm

procedure OPTIMIZE($P, n, iteration$) ▷ return the population P optimized
  Initialize uniformly the n weights : w
  Initialize randomly the population of size μ : P
  while iteration ≥ 0 **do**
    generate offspring : z
    $P = P \cup z$
    $R = fast\_nondominated\_sort(P)$
    $Rh = normalize\_points(R)$ ▷ worst front normalised
    $ra = r2\_indicator(Rh, w, utopian\_points)$ ▷ r2 indicator value for each point inside the worst front
    $a* = min(ra)$
    $P = P - a*$ ▷ remove the worst point
    $iteration = iteration - 1$
  return F ▷ non-dominated front levels

2.1 SBX Crossover and gaussian mutation

Crossover and *Mutation* are genetic operators. A genetic operator is an operator used in genetic algorithms to guide the algorithm towards a solution to a given problem. These operators follow the principle of biological evolution and are comparable to the processes occuring during natural reproduction. *Crossover* is the process of taking more than one parent and producing a child solution from them. By recombining portions of good solutions, the genetic algorithm is more likely to create a better solution. With the principle of selection, parents will tend to become better, enhancing therefore the "quality" of their children. Each entity dimension is defined as a vector where each dimension corresponds to a gene in the biological analogy. Each gene of a child (one dimension of the vector) depends on two genes from two distinct parents. The *R2-EMOA* algorithm uses the *Simulated Binary Crossover* process to achieve the *crossover*. *Simulated binary crossover(SBX)* simulates the effect of one-point crossover on a string of binary alphabets (a vector) in a continuous domaine. A *swap* sub-operator is also added. It allows to reverse the genes between two individuals of the same generation of child in order to favor the diversity. At the end of the *crossover* process, we obtain two children and we keep only one, chosen at random.
The *Gaussian mutation* is a process that limits the risk of getting a local minimum to the problem. The mutation is obtained by modifying the value of the child gene by adding a random value resulting from a Gaussian distribution.

Algorithm 2 SBX

procedure SBX($P, \eta_c, \eta_m, p_{crossover_{variation}}, p_{crossover_{swap}}$) ($p_{mutation_{variation}}, limit_{high}, limit_{low}$)

  $Parent_{selected,1}, Parent_{selected,2} = random_{uniform}(P)$

  for each gene i of child **do**

    if $random_{uniform} < p_{crossover_{variation}}$ **then**
      $\beta_{crossover,i} = SpreadFactor(u_i, \eta_c)$ with u in $random_{uniform([0,1])}$

      $child_1, i = Crossover(Parent_{selected,1,i}, \beta_{crossover,i})$
      $child_2, i = Crossover(Parent_{selected,2,i}, \beta_{crossover,i})$
    else
      $child_1, i = Parent_{selected,1,i}$
      $child_2, i = Parent_{selected,2,i}$

    if $random_{uniform} < p_{crossover_{swap}}$ **then**
      $child_1, i = child_2, i$
      $child_2, i = child_1, i$
  $child_{selected} = random_{uniform}\ (child_1, child_2)$

  for each gene i of $child_{selected}$ **do**
    if $random_{uniform} < p_{mutation_{variation}}$ **then**
      $\beta_{mutation}, i = MutationFactor(u_i, \eta_m)$ with u in $random_{uniforme([0,1])}$
      $child_{selected}, i = Mutation(child_{selected}, i)$
    if $child_{selected}, i > limit_{high}$ **then**
      $child_{selected}, i = limit_{high}$
    if $child_{selected}, i < limit_{low}$ **then**
      $child_{selected}, i = limit_{low}$

2.2 Fast non-dominated sorting

We use this procedure within the R2 EMOA to rank the population in the objective space. The goal is to sort a population into different levels using domination as sorting criterion. The idea is simple, we find the first non-dominated front, then the second non-dominated front, etc. We identify : N as the size of the population, M as the number of objectives (in our bi-objective case, M = 2) To identify the solutions of the first non-dominated front, we compare each solution to all elements in the population set to see if it is dominated or not. This requires M*N comparisons for each solution. The process of finding the first non-dominated front has a complexity of O(MN^2). In order to find the individuals in the next non-dominated front, the solutions of the first front are removed temporarily and the above procedure is repeated. So the final computation is O(MN^3).
As an example, let's consider a population of size N = 5 of points A = (3, 4), B= (1,5), C= (4,4), D = (6,7) and E= (2,3). We search the different levels of non domination by minimization. Clearly, A is dominated by E, C is dominated by A and E, D is dominated by all the others; B and E are not dominated. Thus, the first non-dominated front is R1 = B, E. The next step would be to remove temporally B and E from the population set then we look for the new front. C is dominated by A, D is dominated by A and C and A is not dominated. So the second non-dominated front is R2 = A. Then we have R3 = C and R4 = D.

Algorithm 3 Non-dominated sorting algorithm

```
procedure FAST NON-DOMINATED SORTING(P)  ▷ P the population
    for p in P do
        S_p = ∅
        n_p = 0
        for q in P do

            if p dominates q then
                S_p = S_p ∪ q
            else if q dominates p then
                n_p = n_p + 1
        if n_p = 0 then
            F_1 = F_1 ∪ p                ▷ First Front
    i = 1                                ▷ initialize the front counter
    while F_i ≠ ∅ do
        H = ∅                            ▷ next front
        for p in P do
            for q in P do
                n_q = n_q - 1
                if n_q = 0 then
                    H = H ∪ q
        i = i + 1
        F_i = H
    return F                             ▷ non-dominated front levels
```

2.3 R2 indicator

We use the R2 indicator to assess the relative quality of two Pareto front approximation sets. The smaller the value of the indicator, the better the quality of the front. The R2 formula we used is the following:
R2(A, $\Lambda, \overrightarrow{r*}$)= $\frac{1}{|\Lambda|} \sum_{\lambda \in \Lambda} \min_{\overrightarrow{a} \in A} \max_{j \in 0,\ldots,m} (\lambda_j | r_j * -a_j|)$ where Λ is the set of weights, A is the solution set and $\overrightarrow{r*}$ is the utopian point. Algrithm 2 describes the implementation pseudo code to compute R2.

Algorithm 4 R2 indicator

```
procedure R2 INDICATOR(A, Λ, r*, m)  ▷ return the indicator value R2 = 0
    for λ in Λ do
        H = ∅
        for a in A do
            G = ∅
            for j=1 to m  do
                z = λ_j|r_j * -a_j|
                G = G ∪ z
            H = H ∪ max(G)
        R2 = R2 + min(H)
```

2.4 Generate Weights

We implemented the R2-EMOA with uniformly chosen weight vectors in the space $[0, 1]^2$. One way to do that for our bi-objective case is to compute the weights as : [(0, 1); ($\frac{1}{k-1}$, 1- $\frac{1}{k-1}$) ,. . ., (1, 0)]. We thus generated a set of 500 weights, using the mentioned method, for the implementation of the R2-EMOA.

2.5 Normalizing the objective space

The problems treated by the algorithm might have different scales. In order not to favor the problem with the highest scale, it is necessary to standardize the data. Moreover, in order to always consider [-1e-8, -1e-8] as a utopic point, normalization must be within the interval [0,1]. Normalization is performed at each iteration. The point [1,1] is defined by [fmax (f1), fmax (f2)] and the point [0,0] by [fmin (f1), fmin (f2)] where f1 and f2 are the objective functions and fmax and fmin hold for, respectively, the maximum and minimum value for the corresponding functions. Thus, for each point (x1, x2) in the population P, projected into the objective space, we want to have fmin (f1) ≤ f1(x1) ≤ fmax (f1) (and likewise for x2). To satisfy this condition, we will use this relation for normalization:

$$fi(xi) = \frac{fi(xi) - fmin(fi)}{fmin(fi) + fmax(fi)}$$

2.6 Discussing the parameter setting

The R2-EMOA algorithm has different parameters influencing its performance and results:

The number of iterations This is our stop criterion and corresponds to the number of times that the main loop (creation of a point, deletion of the least relevant) is carried out. The higher the number of iterations, the better the optimization result. However, the computation time increases correspondingly. The number of iterations is thus a compromise between the computation time and the optimality of the result. In our case, the number of iterations was limited to 1000, given the ong runtime of the algorithm and the limited processing power of the hardware.

The variable crossover probability It corresponds to the probability that the creation of the gene of a child is derived from a linear relationship linking the genes of two distinct parents (not just the "copy" of a parent gene). This probability must be high so that the crossover allows the creation of diversified children and not very correlated to their parents, otherwise we would favor the convergence towards a local minimum instead of a global one. In our case, it was set to 0.9.

The distribution index for crossover This is a necessary factor to compute the crossover of the population. A high value tends to create children close to their parents while a low value tends to create distant children. We set the index to 15.

The variable swap probability This is the probability that a gene exchange occurs between two children created by the genetic algorithm. This exchange is in favor of the diversity of the population created in order to tackle the problem of convergence towards a local minimum. This value must not be too high (or too low), otherwise there is a risk of not converging properly to an optimal solution. Indeed, this method allows the creation of a decorrelated population from the associated parents. This is detrimental when parents are solutions of the global minimum. On the

contrary, it is detrimental to have children too close to their parents if they are associated to a local minimum. Thus, it is necessary to find a compromise between scalability and convergence. In our case, we used a variable swap probability of 0.5.

The variable mutation probability It's the probability that a mutation occurs on a child's gene. For the R2-EMOA algorithm, the variation is Gaussian. As for the swap described above, the mutation serves to avoid the problem of local minimum by creating a population decorrelated to the parents. The mutation value should not be too high otherwise there is a risk of slowing convergence towards the optimal solution of the problem (because a parent can be associated with a global minimum, and in this case the child must remain close).

The distribution index for mutation It is a necessary factor to compute the mutation of the population. A high value tends to create children close to their parents while a low value tends to create children away from their parents. In our case, the index was set to 20.

3 RESULTS

Results of R2 EMOA from experiments according to [5], [3] and [1] on the benchmark functions given in [6] are presented in Figures 4, 5, 6, and 7, and in Tables 3 and 4. The experiments were performed with COCO [4], version 2.0, the plots were produced with version 2.0.

4 CPU TIMING

In order to evaluate the CPU timing of the algorithm, we have run the R2 EMOAwith restarts on the entire bbob-biobj test suite [6] for $2D$ function evaluations according to [5]. The Python code was run on a Mac Intel(R) Core(TM) i5-7360U CPU @ 2.50GHz with 1 processor and 4 cores. The time per function evaluation for all dimensions used equals 54 seconds.

5 DISCUSSION

ECDF runtime. We will start by looking at the empirical cumulative distribution of simulated runtimes, aggregated over all 55 functions. We remind that we were limited to 1000 iterations due to time issues, for the whole test suite. Howevere, we could run the algorithm for the first instance (seperable-seperable) using a maximum of 200k function evaluations and we obtained the graphs shown in figures 3

Therefore, we can extrapolate the runtime results for the limited case, as shown in the figure 2, assuming that the results we have for the first instance of problems using 100k,150k and 200k iterations are enough to estimate the trend. Therefore, we identify two tangents: one that is a constant line y=0.8 and the other having as equation y=$\frac{2*x}{7} - \frac{2}{5}$, where $x = log(t)$ and t is the number of function evaluations divided by dimension, which leads to an average log time distribution of 4.2. In table 1, we show the variation of the ECDF's asymptote with the problem dimension.

We can see that the problem dimension affects the average log time distribution. This is kind of obvious since there are $n = number - of - dimensions$ more computation per iteration. (i.e. there are 5 times more computation for a 10D problem than in a 2D problem). What is interesting though is that the difference is not linear (cf. 1). Therefore, it gives us the impression that there is a logarithmic relation between the number of function + target pairs and the problem dimension. This remains a hypothesis, since we would need more dimensions (1D, 15D and 30D) to distinguish the logarithmic (or the polynomial) behaviour.

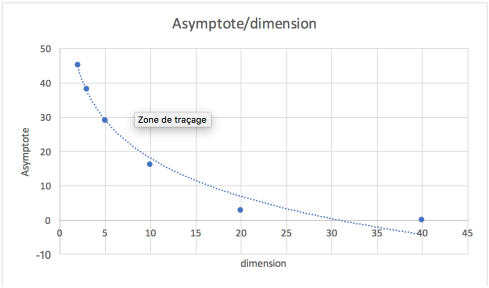

Figure 1: Asymptotic behaviour with dimension

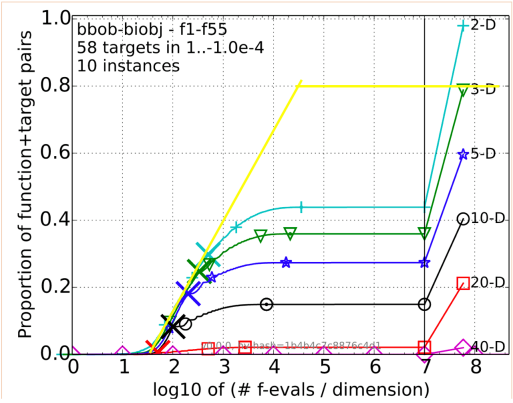

Figure 2: Extrapolation of simulated runtimes, budget = 1000

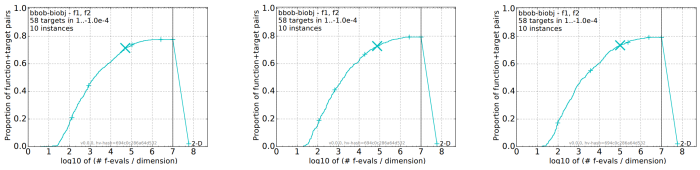

Figure 3: empirical cumulative distribution of simulated runtimes for the first instance in 2D using budgets of 100k, 150k and 200k function evaluations

Asymptote ($y = f(x)$)	dimension
0	40
0.03	20
0.16	10
0.29	5
0.38	3
0.45	2

Table 1: Variation of the ECDF's asymptote with dimension

Comparison to NSGA-II and RS-5. The main visual difference between these two algorithms is the slope. The slope of the NSGA-II is very steep, opposite to the random search slope which is gentle. Whereas both of them do not reach the maximum (100%) proportion of function + target pairs. The maximum achieved is around 80% even if we increase the number of operations.

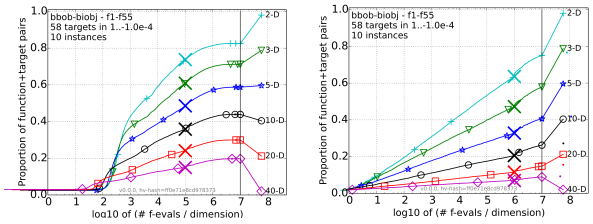

Table 2: comparing NSGA-II and RS-5

In sum, the best algorithm is: $\begin{cases} Random search for fi < 2.15 \\ NSGA-II for fi > 2.15 and fi < 3.1 \\ R2-EMOA for fi > 3.1 \end{cases}$

Efficiency. The algorithms'efficiency differs from one test function to another. If we look at the table of selected index in the post processing data, we can see that our algorithm shows a good performance, for example, for f13, f12 and f18. However the performance is poorer for functions like f53, f50, f31. Contrary to Random Search which performs mediocrely on f14 and f49 and NGSA which has a very good performance for f50.

Suggested improvement of the algorithm. We thought of different ways to improve the R2-EMOA algorithm:

(1) changing the utility function: The weighted Tchebycheff function might not be the most effective one since, first, for normalizition purposes, we require the knowledge of the minimum and maximum values of each objective. Second, it requires the knowledge of the ideal solution z*. It seems unlikely to have this information on completely unknown problems.
We could have used Benon's Method with u(x) = $\sum_{m=1}^{M}$ max(0,(z_m^0 - $f_m(x)$)), the weighted sum method or just dynamically change the ideal solution when we find a pareto-optimal solution. Other than the fact that we could adapt the utility function for the type of problems in hand, depending on whether they are convex/convex, differential/ non-differential.

(2) Parallel computing: For better performance, we could have parallelized the computation process, using for example the *concurrent.futures* package in python, although the ideal choice would be an implementation in C++, for optimal exploitation of data structures.

(3) Improving the SBX algorithm: First, a better (than random) method for choosing the parents can be used. An idea would be to choose one of the closest and one of the farthest points to the pareto front to generate the offspring. Second, with the SBX algorithm we eliminate randomly one offspring, consequently it is possible that we eliminate a good point. Thus, the random elimination should be revised.

6 CONCLUSION

In this project, we had the delight to work on the benchmarking of the R2-EMOA algorithm using the COCO platform. Although it was relatively uncompicated to implement the algorithm in python, its execution conditions were critical for performance assessment. We showed that, using only a limited number of function evaluations, the algorithm's performance is acceptable, for some functions of the BOBJ test suite. We even noted that for some cases, the algorithm outperforms the NSGA-II and the Random Search algorithms. The comparison being made "locally", a generalization is only possible after a sufficiently long run of the algorithm.
At the end of our work, we suggested some improvement vectors that one might take into consideration for future implementations. An important thing to keep in mind is that, in an optimization problem, the performance of the suggested algorithm highly depends on the parameters used and the implementation conditions.

REFERENCES

[1] D. Brockhoff, T. Tušar, D. Tušar, T. Wagner, N. Hansen, and A. Auger. 2016. Biobjective Performance Assessment with the COCO Platform. *ArXiv e-prints* arXiv:1605.01746 (2016).
[2] D. Brockhoff, T. Wagner, and H. Trautmann. 2016. R2 Indicator Based Multiobjective Search. *Evolutionary Computation* (2016), 369.
[3] N. Hansen, A Auger, D. Brockhoff, D. Tušar, and T. Tušar. 2016. COCO: Performance Assessment. *ArXiv e-prints* arXiv:1605.03560 (2016).
[4] N. Hansen, A. Auger, O. Mersmann, T. Tušar, and D. Brockhoff. 2016. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. *ArXiv e-prints* arXiv:1603.08785 (2016).
[5] N. Hansen, T. Tušar, O. Mersmann, A. Auger, and D. Brockhoff. 2016. COCO: The Experimental Procedure. *ArXiv e-prints* arXiv:1603.08776 (2016).
[6] T. Tušar, D. Brockhoff, N. Hansen, and A. Auger. 2016. COCO: The Bi-objective Black-Box Optimization Benchmarking (bbob-biobj) Test Suite. *ArXiv e-prints* arXiv:1604.00359 (2016).

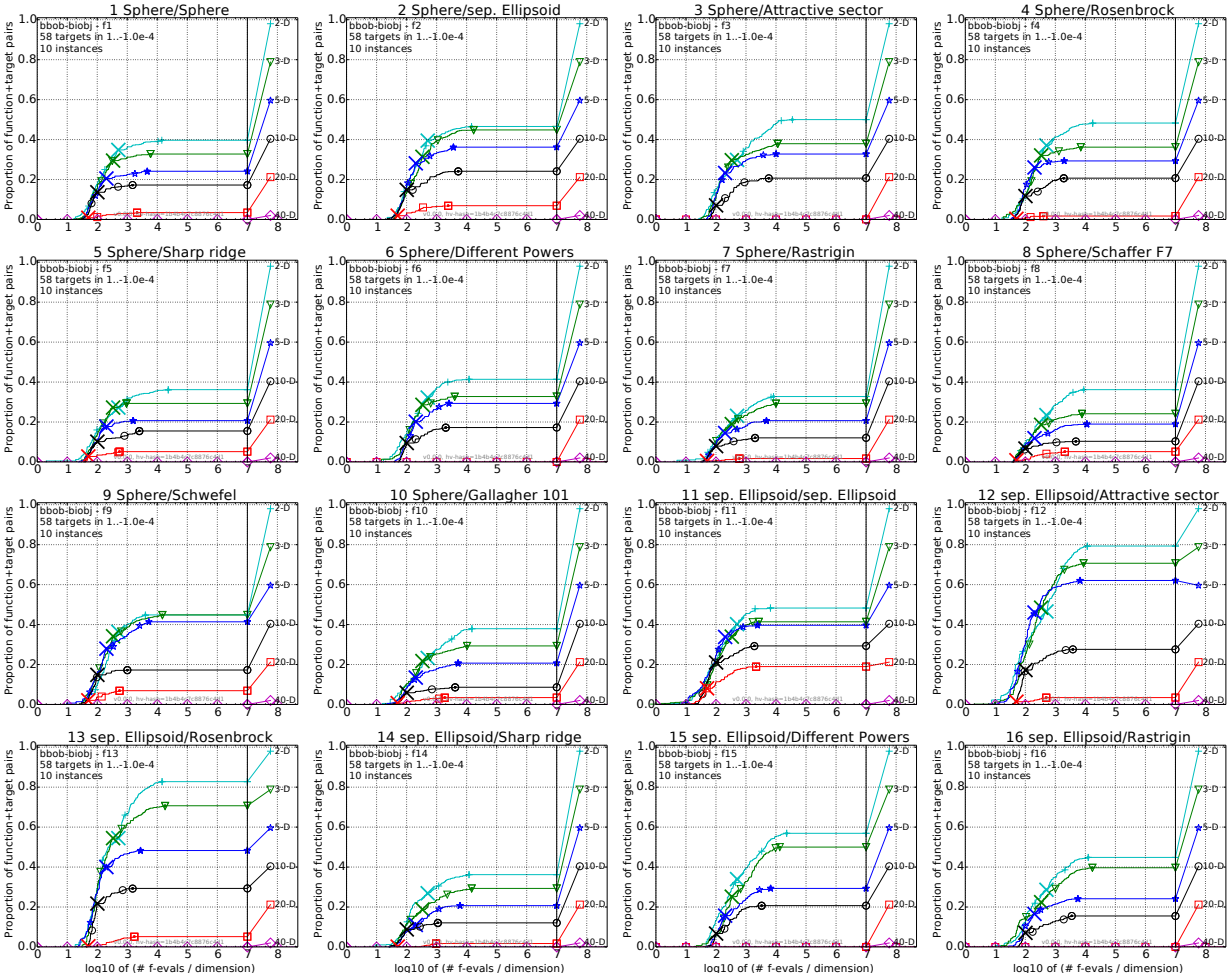

Figure 4: Empirical cumulative distribution of simulated (bootstrapped) runtimes in number of objective function evaluations divided by dimension (FEvals/DIM) for the 58 targets $\{-10^{-4}, -10^{-4.2}, -10^{-4.4}, -10^{-4.6}, -10^{-4.8}, -10^{-5}, 0, 10^{-5}, 10^{-4.9}, 10^{-4.8}, \ldots, 10^{-0.1}, 10^{0}\}$ for functions f_1 to f_{16} and all dimensions.

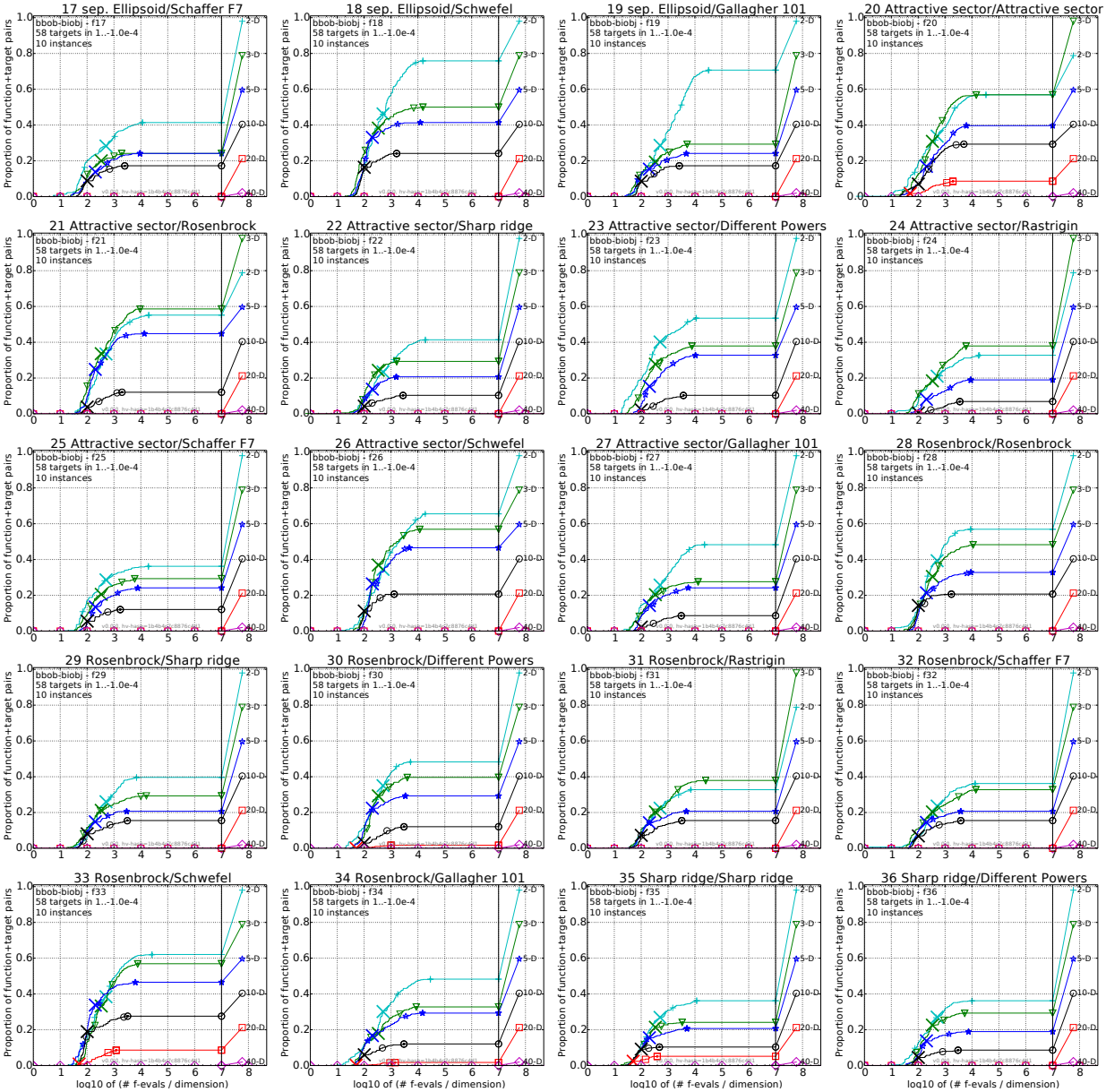

Figure 5: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of objective function evaluations, divided by dimension (FEvals/DIM) for the targets as given in Fig. 4 for functions f_{17} to f_{36} and all dimensions.

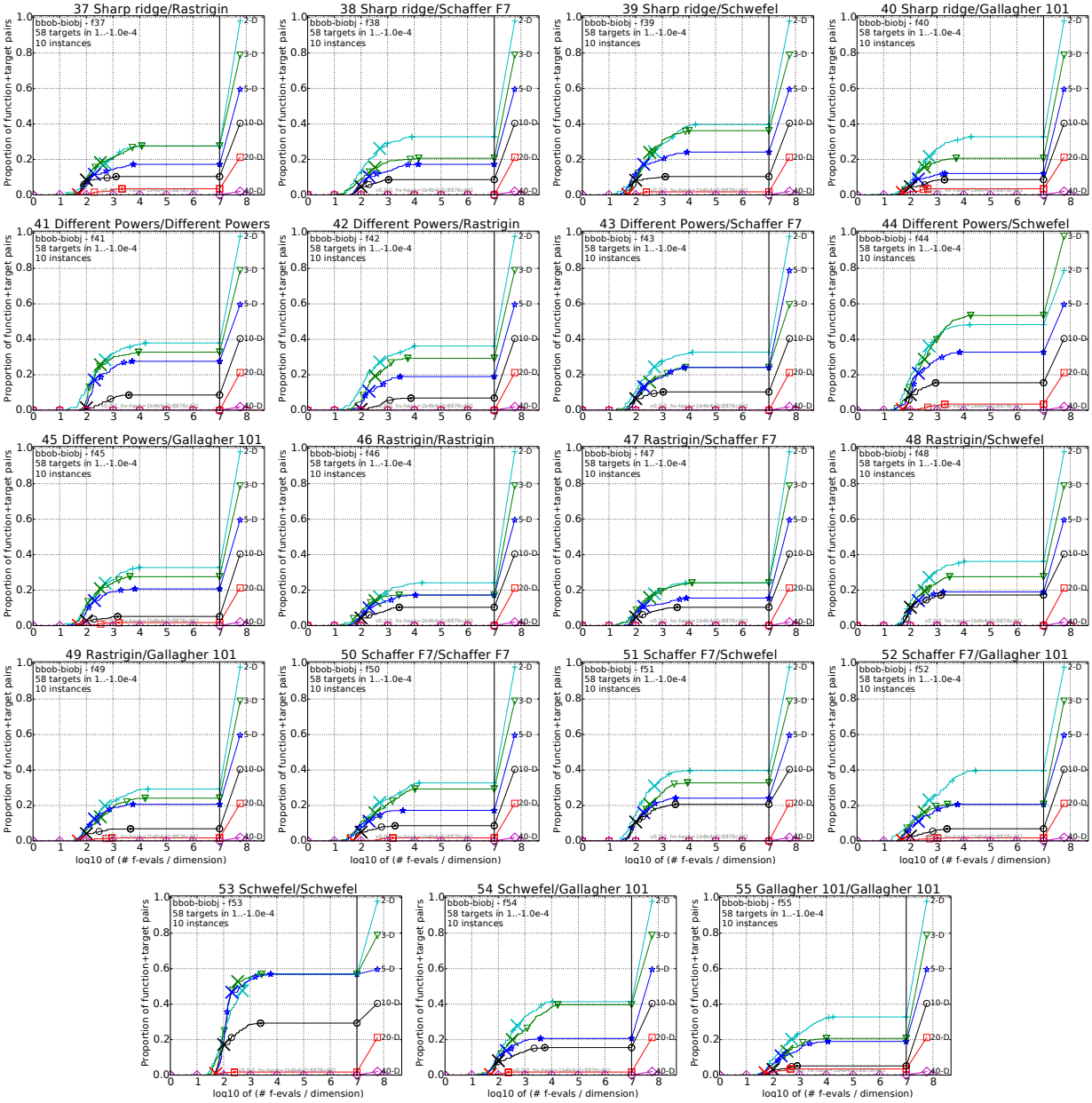

Figure 6: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of objective function evaluations, divided by dimension (FEvals/DIM) for the targets as given in Fig. 4 for functions f_{37} to f_{55} and all dimensions.

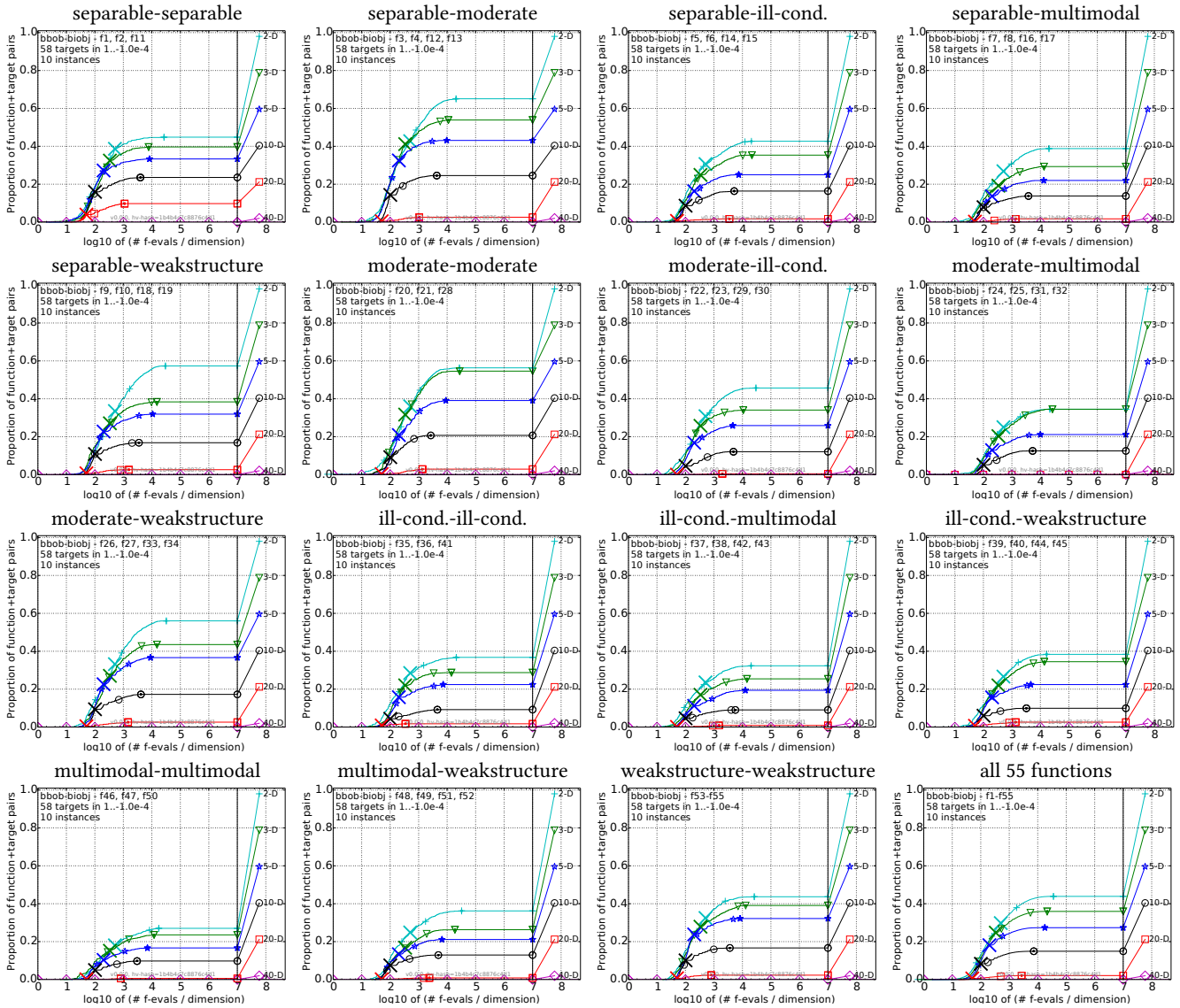

Figure 7: Empirical cumulative distribution of simulated (bootstrapped) runtimes, measured in number of objective function evaluations, divided by dimension (FEvals/DIM) for the 58 targets $\{-10^{-4}, -10^{-4.2}, -10^{-4.4}, -10^{-4.6}, -10^{-4.8}, -10^{-5}, 0, 10^{-5}, 10^{-4.9}, 10^{-4.8}, \ldots, 10^{-0.1}, 10^{0}\}$ for all function groups and all dimensions. The aggregation over all 55 functions is shown in the last plot.

Δf	1e+0	1e-1	1e-2	1e-3	1e-4	1e-5	#succ
$\mathbf{f_1}$	1	75	584	3660	22505	1.4e5	10/10
	209(40)	10(8)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_2}$	5.0	105	601	3715	52410	3.1e5	10/10
	46(15)	6.5(4)	17(23)	∞	∞	∞ *1000*	0/10
$\mathbf{f_3}$	3.0	115	665	5170	44865	3.7e5	10/10
	102(22)	10(8)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_4}$	2.0	109	571	2669	34961	2.8e5	10/10
	131(55)	6.3(2)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_5}$	2.0	120	1277	21889	1.3e5	7.4e5	10/10
	107(33)	12(7)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_6}$	3.0	73	688	3976	48662	4.3e5	10/10
	93(37)	19(10)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_7}$	2.0	2763	1.2e5	3.5e6	2.7e7	∞	0/10
	134(18)	1.7(1)	∞	∞	∞	∞	0/10
$\mathbf{f_8}$	3.0	2167	1.6e5	2.1e6	1.8e7	∞	0/10
	91(45)	4.6(3)	∞	∞	∞	∞	0/10
$\mathbf{f_9}$	4.0	96	521	1986	22026	6.5e5	10/10
	55(12)	7.1(3)	19(18)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{10}}$	4.0	323	9839	52107	1.0e5	4.6e5	10/10
	66(31)	31(20)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{11}}$	5.0	56	436	9189	1.0e5	1.1e6	10/10
	25(18)	6.4(5)	3.3(2)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{12}}$	5.0	44	641	3991	21105	1.0e5	10/10
	46(37)	11(14)	1.4(2)	0.42(0.2)	∞	∞ *1000*	0/10
$\mathbf{f_{13}}$	7.0	60	560	5582	30727	86619	10/10
	32(10)	6.0(3)	1.5(1)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{14}}$	5.0	247	1713	12801	1.0e5	4.8e5	10/10
	66(53)	19(15)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{15}}$	6.0	74	677	6559	43240	2.2e5	10/10
	84(94)	24(16)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{16}}$	3.0	2810	2.0e5	4.2e6	9.7e6	1.8e7	3/10
	104(39)	0.77(1)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{17}}$	29	2935	48624	2.4e6	1.2e7	1.9e7	3/10
	15(27)	1.6(3)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{18}}$	2.0	56	557	6912	64945	1.6e6	8/10
	129(58)	8.2(2)	2.8(2)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{19}}$	9.0	1292	6164	88464	2.3e5	7.2e5	9/10
	25(12)	2.2(1)	∞	∞	∞	∞ *1000*	0/10

Δf	1e+0	1e-1	1e-2	1e-3	1e-4	1e-5	#succ
$\mathbf{f_{20}}$	4.0	70	1162	5724	34382	2.5e5	10/10
	112(60)	24(38)	8.4(8)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{21}}$	5.0	86	3249	9924	36306	2.0e5	10/10
	75(23)	17(16)	0.95(0.5)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{22}}$	3.0	97	1168	12608	1.1e5	1.1e6	9/10
	125(53)	23(33)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{23}}$	1	59	618	4410	42195	3.1e5	10/10
	582(442)	50(37)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{24}}$	5.0	2347	1.8e5	4.1e6	∞	∞	0/10
	127(74)	4.2(4)	∞	∞	∞	∞	0/10
$\mathbf{f_{25}}$	9.0	3143	1.2e5	2.5e6	1.8e7	1.8e7	3/10
	52(29)	1.0(1)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{26}}$	7.0	59	2182	13673	5.4e5	2.2e6	8/10
	71(25)	15(5)	2.2(1)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{27}}$	6.0	2631	21971	44576	78111	6.5e5	10/10
	66(33)	1.2(0.9)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{28}}$	2.0	20	145	1230	55474	6.7e5	9/10
	175(62)	53(30)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{29}}$	3.0	114	1413	13660	1.7e5	4.8e5	10/10
	86(32)	42(25)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{30}}$	1	33	366	2294	54786	2.1e5	10/10
	320(129)	25(15)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{31}}$	3.0	2166	50028	3.2e6	2.7e7	∞	0/10
	105(41)	2.2(2)	∞	∞	∞	∞	0/10
$\mathbf{f_{32}}$	3.0	2131	1.1e5	2.4e6	1.8e7	5.7e7	2/10
	100(39)	2.2(3)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{33}}$	6.0	627	1214	3730	46518	3.4e6	8/10
	46(24)	0.99(1.0)	0.99(0.8)	∞	∞	∞ *1000*	0/10
$\mathbf{f_{34}}$	9.0	1376	15575	54195	81900	3.4e5	10/10
	31(12)	3.5(3)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{35}}$	1	141	2268	51388	4.3e5	2.1e6	8/10
	229(106)	23(29)	∞	∞	∞	∞ *1000*	0/20
$\mathbf{f_{36}}$	2.0	204	4640	41237	1.2e5	5.2e5	10/10
	171(114)	49(72)	∞	∞	∞	∞ *1000*	0/20
$\mathbf{f_{37}}$	2.0	3956	1.5e5	2.5e6	2.7e7	5.6e7	1/10
	157(30)	∞	∞	∞	∞	∞ *1000*	0/10

Δf	1e+0	1e-1	1e-2	1e-3	1e-4	1e-5	#succ
$\mathbf{f_{38}}$	4.0	4366	2.7e5	5.4e6	1.3e7	5.8e7	2/10
	69(15)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{39}}$	2.0	215	1160	47472	87155	2.0e6	8/10
	112(30)	14(27)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{40}}$	2.0	998	34442	2.0e5	3.6e5	1.8e6	8/10
	147(48)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{41}}$	2.0	48	708	6343	48028	3.7e5	10/10
	240(78)	38(30)	∞	∞	∞	∞ *1000*	0/20
$\mathbf{f_{42}}$	2.0	2525	3.4e5	4.3e6	5.6e7	∞	0/10
	254(104)	3.9(3)	∞	∞	∞	∞	0/10
$\mathbf{f_{43}}$	4.0	2468	1.5e5	3.8e6	1.3e7	2.8e7	2/10
	106(67)	3.8(6)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{44}}$	6.0	86	513	1853	63019	1.1e6	10/10
	70(54)	10(5)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{45}}$	4.0	816	5730	53653	91090	3.5e5	10/10
	82(22)	5.9(10)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{46}}$	4.0	11925	4.9e5	5.6e7	∞	∞	0/10
	92(35)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{47}}$	3.0	4172	2.7e5	4.4e6	5.8e7	∞	0/10
	102(27)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{48}}$	9.0	2160	1.1e5	2.1e6	2.5e7	2.7e7	2/10
	39(15)	1.1(0.7)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{49}}$	9.0	3737	2.0e5	1.3e6	1.2e7	5.6e7	1/10
	35(19)	1.3(1)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{50}}$	6.0	3913	2.2e5	2.6e6	5.7e7	5.8e7	1/10
	44(22)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{51}}$	8.0	1251	32465	2.5e6	1.3e7	5.8e7	1/10
	32(13)	1.6(1)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{52}}$	2.0	3027	1.4e5	2.1e6	7.7e6	5.7e7	1/10
	135(26)	3.2(3)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{53}}$	2.0	13	42	1443	17113	2.1e6	2/10
	158(53)	38(10)	16(4)	2.2(3)	∞	∞ *1000*	0/10
$\mathbf{f_{54}}$	98	1514	12856	45502	56608	3.2e5	10/10
	3.4(3)	3.2(3)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{55}}$	3.0	1415	18086	51245	7.2e5	1.3e6	9/10
	74(26)	7.1(6)	∞	∞	∞	∞ *1000*	0/10

Table 3: Average running time (aRT **in number of function evaluations) divided by the** aRT **of the best algorithm from BBOB 2016 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 $\Delta I_{\mathrm{HV}}^{\mathrm{COCO}}$-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 2016, with $p = 0.05$ or $p = 10^{-k}$ when the number $k > 1$ is following the $\downarrow$ symbol, with Bonferroni correction by the number of functions.**

Data produced with COCO v0.0.0, hv-hash=ff0e71e8cd978373

Δf	1e+0	1e-1	1e-2	1e-3	1e-4	1e-5	#succ
$\mathbf{f_1}$	1	157	1468	8244	81769	6.2e5	10/10
	2437(2021)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_2}$	5.0	163	2170	21153	2.5e5	4.5e6	10/10
	648(707)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_3}$	1	216	2384	11104	1.0e5	1.4e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_4}$	1	172	1682	10200	2.2e5	4.3e6	9/10
	4986(11110)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_5}$	1	190	3207	32860	2.2e5	1.9e6	10/10
	1343(782)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_6}$	3.0	236	2134	16568	1.2e5	8.4e5	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_7}$	1	24981	6.8e5	3.8e7	∞	∞	0/10
	5018(5546)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_8}$	4.0	16448	2.4e6	2.8e7	∞	∞	0/10
	1238(631)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_9}$	1	144	1468	5944	43100	7.4e5	10/10
	3128(2525)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{10}}$	1	6124	2.0e5	2.4e5	5.3e5	2.9e6	10/10
	1897(1087)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{11}}$	3.0	246	3177	1.1e5	4.7e6	1.9e8	1/10
	226(39)	41(47)	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{12}}$	11	131	1483	11435	1.6e5	6.6e5	10/10
	441(552)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{13}}$	7.0	139	1087	17899	2.2e6	1.4e7	9/10
	713(610)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{14}}$	3.0	349	6102	63789	6.7e5	8.4e6	10/10
	3353(5050)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{15}}$	6.0	210	2099	25094	3.7e5	3.6e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{16}}$	6.0	32242	1.0e6	1.9e7	2.3e8	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{17}}$	1	33890	9.3e5	2.7e7	2.3e8	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{18}}$	6.0	141	1507	17541	4.2e5	7.5e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{19}}$	7.0	18889	2.4e5	5.1e6	1.4e7	2.1e7	6/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{20}}$	3.0	136	1870	8641	46568	6.8e5	10/10
	3248(2609)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{21}}$	5.0	331	1753	9785	1.1e6	6.5e6	8/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{22}}$	1	313	4469	23797	2.0e5	3.1e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{23}}$	3.0	332	2660	16975	1.2e5	1.6e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{24}}$	1	61504	8.1e5	6.5e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{25}}$	1	34160	9.2e5	1.2e7	1.1e8	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{26}}$	3.0	136	1117	4425	1.2e5	4.3e6	9/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{27}}$	1	60235	1.3e6	6.4e6	9.3e6	1.1e7	7/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{28}}$	1	62	539	6898	5.0e5	1.4e7	8/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{29}}$	1	359	3841	30739	2.8e5	1.0e7	8/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{30}}$	1	220	1777	10156	1.4e5	6.9e6	9/10
	10061(8080)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{31}}$	3.0	27722	8.6e5	3.0e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{32}}$	6.0	32985	1.1e6	2.6e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{33}}$	1	33	278	3348	3.7e5	1.5e7	6/10
	3283(2579)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{34}}$	3.0	45588	1.9e5	2.3e5	2.6e6	1.1e7	8/10
	3353(3872)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{35}}$	1	338	4782	41429	3.2e5	3.0e6	10/10
	1836(1529)	∞	∞	∞	∞	∞ *1000*	0/20
$\mathbf{f_{36}}$	1	454	4824	48159	3.2e5	2.9e6	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/2
$\mathbf{f_{37}}$	1	10528	5.2e5	3.5e7	∞	∞	0/10
	9955(9090)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{38}}$	3.0	36705	3.9e6	2.2e8	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{39}}$	1	457	3876	25958	1.7e5	2.2e6	10/10
	5020(2775)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{40}}$	1	26134	2.1e5	2.6e6	6.1e6	1.4e7	7/10
	1386(1041)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{41}}$	2.0	245	2389	15818	1.2e5	8.2e5	10/10
	∞	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{42}}$	1	34578	2.9e6	2.3e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{43}}$	2.0	55957	2.3e6	3.9e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{44}}$	3.0	168	1485	6584	46647	1.0e6	10/10
	3346(6481)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{45}}$	1	49797	2.4e6	5.2e6	9.0e6	1.0e7	7/10
	9994(9090)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{46}}$	1	55141	1.0e6	1.1e8	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{47}}$	2.0	54618	2.2e6	5.2e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{48}}$	1	22301	3.5e5	3.3e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{49}}$	2.0	1.6e5	1.1e7	7.1e7	∞	∞	0/10
	5050(6818)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{50}}$	3.0	22169	1.3e6	2.0e7	∞	∞	0/10
	3338(3114)	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{51}}$	1	8574	9.9e5	2.2e7	∞	∞	0/10
	∞	∞	∞	∞	∞	∞	0/10
$\mathbf{f_{52}}$	1	82166	4.9e6	2.4e7	1.1e8	2.3e8	1/10
	4970(4040)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{53}}$	1	28	159	1952	14849	9.2e6	8/10
	2499(2279)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{54}}$	7.0	6049	1.6e5	2.5e6	2.7e6	7.3e6	8/10
	464(510)	∞	∞	∞	∞	∞ *1000*	0/10
$\mathbf{f_{55}}$	1	8.4e5	8.8e6	8.8e6	1.6e7	1.9e7	7/10
	1907(1558)	∞	∞	∞	∞	∞ *1000*	0/10

Table 4: Average running time (aRT **in number of function evaluations) divided by the** aRT **of the best algorithm from BBOB 2016 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 $\Delta I_{\mathrm{HV}}^{\mathrm{COCO}}$-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 2016, with $p = 0.05$ or $p = 10^{-k}$ when the number $k > 1$ is following the $\downarrow$ symbol, with Bonferroni correction by the number of functions.**

Data produced with COCO v0.0.0, hv-hash=ff0e71e8cd978373

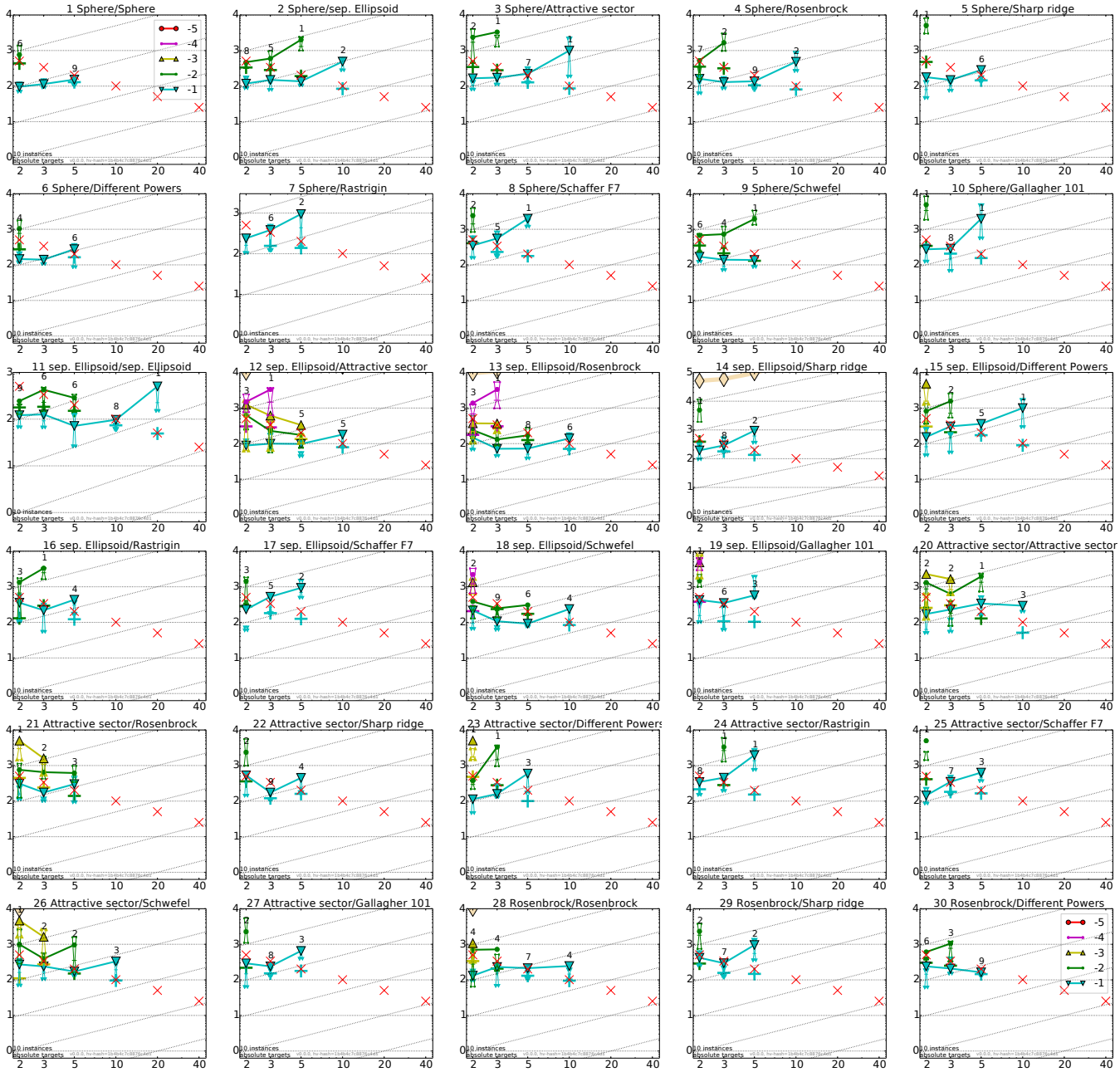

Figure 8: Scaling of runtime with dimension to reach certain target values $\Delta I_{\mathrm{HV}}^{\mathrm{COCO}}$. Lines: average runtime (aRT); Cross (+): median runtime of successful runs to reach the most difficult target that was reached at least once (but not always); Cross (×): maximum number 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 $\Delta I_{\mathrm{HV}}^{\mathrm{COCO}} = 10^k$ 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 2016 for the most difficult target. Horizontal lines mean linear scaling, slanted grid lines depict quadratic scaling.

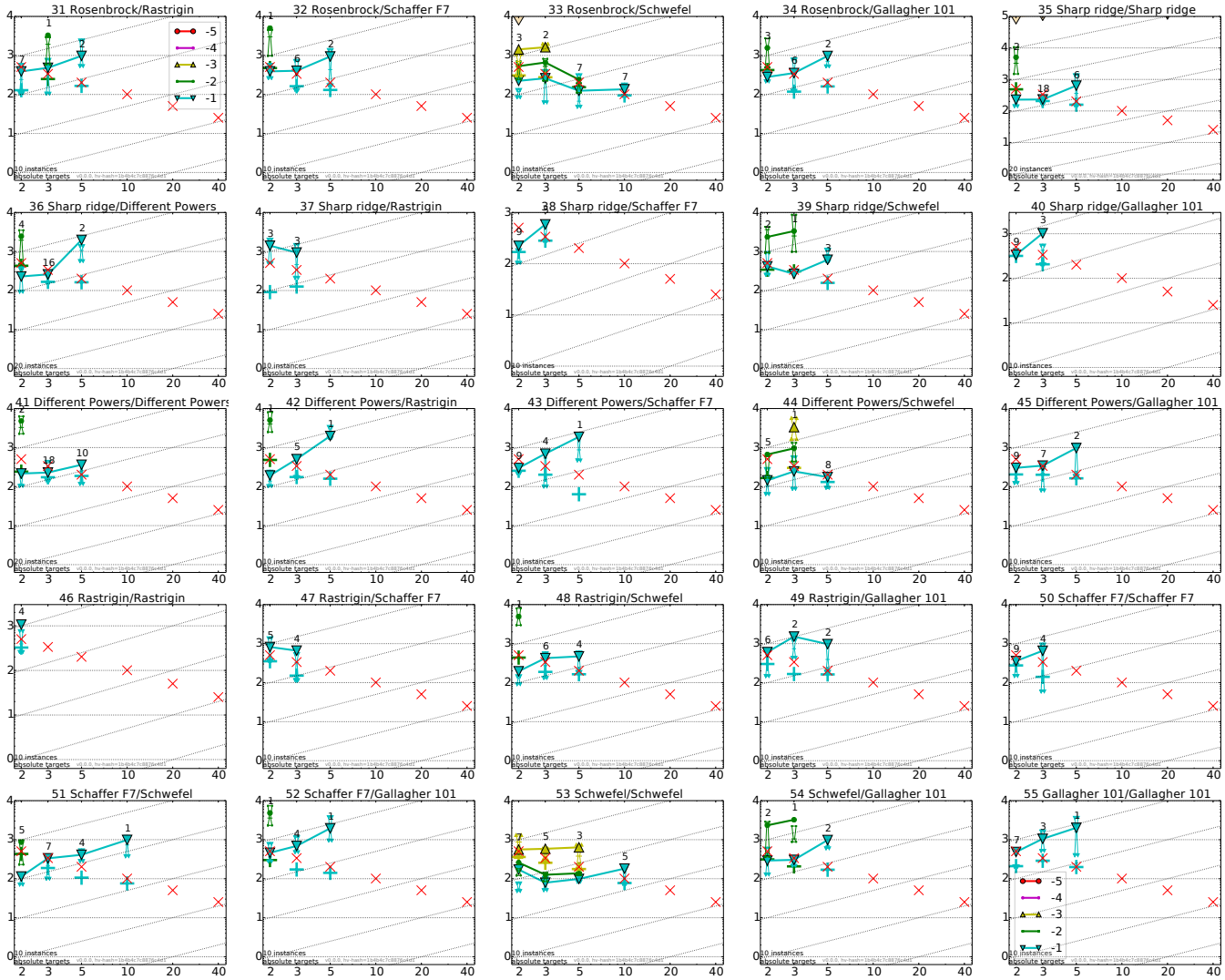

Figure 9: Runtime versus dimension as described in Fig. 8, here for functions f_{31} to f_{55}.