Benchmarking large-scale continuous optimizers: the bbob-largescale testbed, a COCO software guide and beyond

Konstantinos Varelas, Ouassim Ait El Hara, Dimo Brockhoff, Nikolaus Hansen, Duc Manh Nguyen, Tea Tušar, Anne Auger

▶ To cite this version:

Konstantinos Varelas, Ouassim Ait El Hara, Dimo Brockhoff, Nikolaus Hansen, Duc Manh Nguyen, et al.. Benchmarking large-scale continuous optimizers: the bbob-largescale testbed, a COCO software guide and beyond. Applied Soft Computing, 2020, 97 (A), pp.106737. 10.1016/j.asoc.2020.106737 . hal-02960253

HAL Id: hal-02960253
https://inria.hal.science/hal-02960253

Submitted on 7 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Benchmarking large-scale continuous optimizers: the `bbob-largescale` testbed, a COCO software guide and beyond

Konstantinos Varelas[a,d,*], Ouassim Ait El Hara[a], Dimo Brockhoff[a], Nikolaus Hansen[a], Duc Manh Nguyen[b], Tea Tušar[c], Anne Auger[a]

[a] *Inria, RandOpt team, CMAP, Ecole Polytechnique, IP Paris, France*
[b] *Hanoi National University of Education, Vietnam*
[c] *Jožef Stefan Institute, Slovenia*
[d] *Thales LAS France SAS*

Abstract

Benchmarking of optimization solvers is an important and compulsory task for performance assessment that in turn can help in improving the design of algorithms. It is a repetitive and tedious task. Yet, this task has been greatly automatized in the past ten years with the development of the Comparing Continuous Optimizers platform (COCO).

In this context, this paper presents a new testbed, called `bbob-largescale`, that contains functions ranging from dimension 20 to 640, compatible with and extending the well-known single-objective noiseless `bbob` test suite to larger dimensions. The test suite contains 24 single-objective functions in continuous domain, built to model well-known difficulties in continuous optimization and to test the scaling behavior of algorithms. To reduce the computational demand of the orthogonal search space transformations that appear in the `bbob` test suite, while retaining some desired properties, we use permuted block diagonal orthogonal matrices. The paper discusses implementation technicalities and presents a guide for using the test suite within the COCO platform and for interpreting the postprocessed output. The source code of the new test suite is available on GitHub as part of the open source COCO benchmarking platform.

*Corresponding author

Keywords: large-scale optimization, black-box optimization, benchmarking

1. Introduction

Benchmarking is an important task in optimization that every algorithm designer has to do to validate a new algorithm. It can also assist the designer by pointing out weaknesses that have been overlooked in the first conception phase of the algorithm. The choice of the test functions is crucial as performance is often aggregated over sets of functions and a bias towards certain properties can lead to a misrepresentation of the "real" performance of an algorithm.

Optimization problems with more than one hundred variables are common in many domains. We therefore naturally need benchmarking suites to test algorithms in these dimensions and to investigate their scalability.

This paper introduces a new benchmarking test suite with the following objectives.

- We extend the widely used Black-Box Optimization Benchmarking suite [1], `bbob`, to larger dimensions. The `bbob` suite is part of the Comparing Continuous Optimizers benchmarking platform [2, 3], COCO, a general tool for benchmarking continuous solvers. The suite has been widely used for the performance comparison of various types of solvers (deterministic, stochastic, evolutionary, gradient-free, gradient-based, etc.), see e.g. [4, 5, 6, 7, 8].
- We allow to investigate the scaling of algorithms up to dimension 640 in a quantitative way, based on the standardized experimental setup of COCO. A unique feature of our proposal is that the presented suite is an extension of the well-established COCO platform with its corresponding advantages: it offers a thought-out, standardized experimental setup, facilitates the automated processing of results (see introduction of Section 3), uses the number of function evaluations for the quantitative assessment of the performance and of the scaling with dimension on the highest possible measurement scale (see Section 3.2), and allows to easily collect and

compare algorithm performance data from different sources (see introduction of Section 5 and third paragraph of Appendix A.2). In addition, the new suite naturally extends the dimensionality of the original `bbob` problems where overlapping dimensions allow to verify that the two suites are compatible (see the introduction of Section 4).

While some results of specific solvers are included as examples, our contribution is not a benchmarking study in itself and does not provide an empirical analysis of benchmarking data. Our contribution only intends to serve as a tool for conducting future benchmarking studies in dimension up to 640, to which we will refer, in the context of this paper, as "large-scale".

In this context, the main contribution of this paper is to introduce thoroughly the novel `bbob-largescale` test suite based on the `bbob` suite [1] and based on the idea of permuted orthogonal block-diagonal matrices [9].

The `bbob-largescale` test suite is implemented within the COCO platform [2, 3]. We discuss in detail the adjustments needed and decisions taken to arrive at the final test suite. These adjustments are necessary to be backwards compatible with the `bbob` test suite and to avoid artificial biases towards certain algorithms or algorithm settings (like optima too close to the origin because of normalization factors).

Additionally, we illustrate how to use the new test suite in the context of the COCO platform to be able to benchmark a novel algorithm. In the appendix we provide a software user guide, show the plots that are automatically producible with COCO and outline which scientific information we can gather from them.

The paper is organized as follows: Section 2 discusses available test suites for large-scale optimization and their relation to the proposed `bbob-largescale` one. Section 3 details the terminology and philosophy underlying the COCO platform in which we implement the proposed suite. The actual `bbob-largescale` suite is then presented in Section 4 with a detailed definition of each `bbob-largescale` function in Subsection 4.7. In Section 5 we provide practical information related to the implementation of the suite and summarize results

obtained from different CMA-ES variants. Last, Appendix A showcases how
60 the new test suite can be used in COCO and Appendix B gives examples of scientific conclusions that can be obtained from running numerical experiments with the suite.

2. Related work

In this section we introduce the bbob test suite and discuss related work in
65 large-scale benchmarking.

2.1. The BBOB test suite

The testbed we will introduce later is based on the Black-Box Optimization Benchmarking test suite (bbob, [1]) of the COCO platform [2, 3], introduced in 2009. The bbob test suite was constructed with the idea to provide

70
- functions that represent well-known difficulties in continuous optimization, namely non separability, multimodality, ill-conditioning and landscape ruggedness;
- transformations to make functions look less regular, because we do not expect that many real world problems can be expressed in simple and
75 closed mathematical formulas;
- function pairs and groups that allow to test specific properties of an algorithm (for instance, "does the algorithm exploit separability?");
- a wide range of challenging test problems to reduce the risk of overfitting and to challenge algorithms as much as possible.

In comparison to other well-known test function suites (for example the CUTEr/CUTEst suite [10] [11]), the bbob functions are mostly non-convex and non-smooth. The bbob test suite is structured into five function groups, namely separable functions, functions with low or moderate conditioning, unimodal functions with high conditioning, multimodal functions with adequate global

structure, and multimodal functions with weak global structure. Since the notion of separability can be formulated mathematically in various ways, we hereby adopt the following definition: a function $f : \mathbb{R}^n \to \mathbb{R}$ is called separable if it can be expressed as:

$$f(x) = \sum_{i=1}^{n} f_i(x_i)$$

for some functions $f_i : \mathbb{R} \to \mathbb{R}$, that is, if it is additively decomposable into the sum of univariate functions of single coordinates.

Each `bbob` function group contains 5 functions except the second one that contains four functions. This balance between the number of functions per group is important to keep in mind when interpreting aggregated performance results.

An additional important aspect of the `bbob` functions is their scalability: every function has an analytic expression and is defined for an arbitrary dimension. This suggests that the `bbob` test suite could be used to test "large"-scale algorithms. Yet there is a practical limitation of the original `bbob` test suite that precludes its usage for dimensions larger than a few hundreds of variables: many of the `bbob` functions involve matrix multiplications with dense matrices to make them non-separable. More precisely, these `bbob` functions are constructed in an onion-like fashion as:

$$f(x) = F_1 \circ F_2 \circ \ldots \circ F_k \circ f_{\text{raw}} \circ T_1 \circ T_2 \circ \ldots T_l(x)$$

where f_{raw} is the underlying raw objective function, for example the ellipsoid function $f_{\text{elli}}(x) = \sum_{i=1}^{n} 10^{6\frac{i-1}{n-1}} x_i^2$, the F_i are objective space transformations of the form $F_i : \mathbb{R} \to \mathbb{R}$, and the T_i are search space transformations of the form $T_i : \mathbb{R}^n \to \mathbb{R}^n$. Examples of such search space transformations are simple translations and search space *rotations* $T_R : x \mapsto Rx$ with R being an orthogonal matrix in $\mathbb{R}^n \times \mathbb{R}^n$.

Orthogonal matrices, that we also refer to as rotation matrices, are at the core of the constructions of many benchmark functions. They allow to have a simple writing of the functions while not favoring a specific representation of the

95 problem (the representation given by the original coordinate system): we can start from a separable function that is typically easy to write and to comprehend and we rotate it to get a non-separable function [12]. This way, we keep the simplicity of the writing of separable functions but take out the separability bias. This construction is scalable. Yet, if a dense orthogonal matrix is used, 100 the matrix vector product calculation is quadratic in the problem dimension and the computation becomes too prohibitive when having, say, more than a few hundred variables and hundreds of problem instances.

For this reason, the idea to replace orthogonal matrices by *sparse orthogonal* matrices has been introduced in [9] to build benchmark functions in large 105 dimensions. Each dense orthogonal matrix is thereby replaced by a permuted block matrix P_1BP_2 with only a linear (in the dimension) number of non-zero coefficients where P_1 and P_2 are permutation matrices and B is a block-diagonal matrix. The reason for using such so-called *permuted orthogonal block-diagonal matrices* in the context of large-scale optimization benchmarking is two-fold: 110 on the one hand, the computation time for the test functions becomes linear in the problem dimension instead of quadratic, resulting in reasonable computation times, on the other hand, we also reckon that real-world problems in large dimensions typically have less than quadratically many degrees of freedom and a test problem construction via sparse orthogonal matrices will automatically 115 keep the number of variable dependencies lower than quadratic.

2.2. Large-scale benchmarking

A few test suites for benchmarking numerical optimizers have been around for some time. In the context of large-scale optimization, most notably developed by the "classical" optimization community, are the COPS 3.0 problems 120 [13] and the general CUTEr/CUTEst problems [10] [11].

The Constrained Optimization Problem Set (COPS) 3.0 test suite contains 22 large-scale problems with 398 to 19240 variables, some of which can be used in arbitrary dimension while others are only defined for very specific dimensions. Despite the suite's name, three of the COPS problems are unconstrained.

The CUTEr/CUTEst library, on the other hand, contains many more problems (more than 1000), with 378 of them being unconstrained. Of those, 184 problems are available in any dimension and can thus be used to benchmark large-scale optimization algorithms in principle. From these 184 scalable unconstrained problems, finally only 73 of them are not constant, linear, quadratic, or of a sum of squares type.

In the evolutionary computation community, large-scale competitions have been organized at the CEC conference from which three large-scale test suites evolved over time:

- The CEC 2008 suite [14] with 7 functions: shifted Sphere, shifted Schwefel's Problem 2.21, shifted Rosenbrock, shifted Rastrigin, shifted Griewank, shifted Ackley and FastFractal "DoubleDip", tested in three different dimensions.
- The CEC 2010 suite [15] with 20 functions in total and 6 *underlying* functions: Sphere, rotated Ellipsoid, Schwefel's Problem 1.2, Rosenbrock, rotated Rastrigin, and rotated Ackley. These basic functions are combined with no/partial/full rotations to create the 20 functions overall. The competition was setup with the single dimension 1000.
- The CEC 2013 suite [16], based on the CEC 2010 suite, with additional `bbob` transformations, nonuniform subcomponent sizes, imbalance in the contribution of subcomponents and functions with overlapping subcomponents. The competition was setup with the single dimension 1000.

The CEC competitions are setup with a single or small number of different dimensions (although the problems are, in principle, scalable) and the performance assessment is prescribed for a few given budgets and also for three given targets in the CEC 2010 case. This setup does not allow to reliably measure scaling behavior with dimension—one of the most important characteristics a benchmarking experiment for large-scale algorithms should investigate. A different setup was followed in [17], where test functions from the CEC 2010 test

suite were used with adjusted dimensions and budgets.

The benchmark suite introduced in [18] consists of a subset of the test functions introduced in the CEC competitions together with additional test functions. In particular, the functions from the CEC 2008 competition without the FastFractal "DoubleDip" function, 5 (shifted) functions, namely the Schwefel's Problem 2.22, the Schwefel's Problem 1.2, the extended Schaffer function, the Bohachevsky and the Schaffer function, as well as hybrid composition functions built from them formed a testbed of 19 problems in total [19]. In contrast to CEC, the performance was assessed for 5 different dimensions between 50 and 1000, for a given budget and with independent restarts. The performance criterion was the distance between the best achieved and the optimal function value.

Similar to the COPS and CUTEr/CUTEst problems, also for the CEC problems, no effort was spent on investigating whether target difficulties are comparable over problems and dimensions, however, this similarity is necessary to aggregate performances properly over different problems and to investigate the scaling behavior with the problem dimension.

None of the mentioned test suites is furthermore implemented to allow for an *automated benchmarking*, during which the performance data are recorded automatically, to relieve the user from the burden of implementing this tedious task. We address the automated benchmarking issue and the above mentioned shortcomings of the currently available test suites for large-scale (nonlinear or black-box) optimization benchmarking by proposing the `bbob-largescale` suite and by providing its implementation via the COCO platform.

3. Automated Benchmarking with the Comparing Continuous Optimizers Platform

The COCO platform [2, 3] has been designed to simplify and standardize the tedious tasks of benchmarking black-box algorithms in continuous domain. It provides several test suites (for example the unconstrained single-objective `bbob`

and `bbob-noisy` suites and the bi-objective `bbob-biobj` suite), interfaces several languages (C/C++, Java, Matlab/Octave, Python, R) and supports Linux,
185 Mac, and Windows operating systems. Provided example experiment scripts showcase how to connect basic algorithms to the supported test suites. During an experiment, performance data in terms of runtimes to reach predetermined target function values for each problem instance are automatically collected and written to files. Those data files can then be read in with COCO's postprocessing
190 module (written in Python) that displays performance in graphical and tabular form in both pdf and html format. A great advantage of the standardized COCO data format is that data from a few hundred algorithm variants can by now be compared easily with its postprocessing.

In order to introduce the new `bbob-largescale` test suite in the next section,
195 we will first discuss the basic COCO terminology and philosophy, especially regarding the ideas of problem instances, recorded runtimes, and function target values.

Throughout the paper, we consider single-objective, unconstrained minimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x),$$

where n is the problem dimension. The objective is to find, as quickly as possible, one or several solutions x in the search space $\mathbb{R}^n$ with *small* value(s) of
200 $f(x) \in \mathbb{R}$. We generally measure the *time* of an optimization run as the number of calls to (queries of) the objective function f.

More precisely, the term objective **function** f refers to a parametrized mapping $\mathbb{R}^n \to \mathbb{R}$, where n is not a priori specified, i.e. the search space is scalable. The parametrization allows the definition of different *instances* of f, by applying
205 transformations in the search or objective space, e.g. rotations or translations.

A **problem** is an instance of an objective function on which the optimization algorithm under consideration is run. Aiming to assess the performance of the algorithm, we further attach target f-values to the problem.

The measure that is used to evaluate the algorithm's performance is the

210 **runtime**, or **run-length**, defined as the conducted *number of evaluations*, also referred to as number of *function* evaluations, to reach a given target on a given problem for the first time. These targets are determined by a set of fixed target precisions added to the optimal f-value.

Collecting such problems constitutes the **test-** or **benchmark-suite**.

215 *3.1. Functions, Instances and Problems*

Each function in a COCO suite is defined and *parametrized* by the (input) dimension, $n \in \mathbb{N}_+$, its identifier $i \in \mathbb{N}_+$, and the instance number, $j \in \mathbb{N}_+$, that is:

$$f_i^j \equiv f[n,i,j] : \mathbb{R}^n \to \mathbb{R} \quad x \mapsto f_i^j(x) = f[n,i,j](x).$$

In the previous context, a fixed triple $[n,i,j] \equiv [n,f_i,j]$ corresponds to the optimization problem presented to the optimization algorithm. Diversifying n or j varies the search space dimension or the instance respectively of the same objective function $i \equiv f_i$.

220 Specific instances are deterministically defined as specific sets of transformations applied to the objective function. The instance number j is in practice the integer that is used for seeding the pseudo-random generation of the transformations.

One advantage of problem instances in a test suite is that experiments of 225 algorithms on slightly varying instances of the same underlying function allows to naturally compare stochastic with deterministic algorithms. The recorded runtimes over the instances of a function can be interpreted (for both stochastic and deterministic algorithms) in the same way as runtimes from multiple runs on the same instance of a stochastic algorithm.

230 *3.2. Runtime and Target Values*

In order to measure the runtime (number of function evaluations) of an algorithm on a problem, we prescribe a **target f-value**, t [20]. In a single run, if the target value t of a problem (f_i,n,j,t) is reached or surpassed, the

problem is solved.[1][2] Recorded runtimes are the only means of evaluating the algorithm performance. Runtimes can be quantitatively interpreted on a ratio scale and allow to measure scaling with the dimension. They are undetermined if the problem is not solved in a single run—however lower bounded by the total number of f-evaluations of this run. Since larger budgets increase the probability of reaching the targets, they are generally preferable. Reasonable termination conditions are not to be disregarded, though, and restarts should be conducted in case [21].

4. The `bbob-largescale` Test Suite

The `bbob-largescale` test suite provides 24 functions in six dimensions (20, 40, 80, 160, 320 and 640) within the COCO framework. All 24 functions are, in principle, scalable to an arbitrary dimension. The suite is derived from the existing single-objective, unconstrained `bbob` test suite with modifications that allow the user to benchmark algorithms on higher-dimensional problems efficiently. As the experimental setup for the `bbob` suite specifies dimensions 2, 3, 5, 10, 20, and optionally also dimension 40, a natural extension was to use dimension 40 or 80 as the smallest dimension in the new suite. However, in order to facilitate comparison and verification across both test suites, we decide to guarantee one overlapping dimension, namely 20. Hence, the `bbob-largescale` suite starts with dimension 20 and provides, following the tried-and-tested set-

[1] Note that we use the term *problem* in two meanings: the tuple (f_i, n, j) is the concrete objective function, an algorithm $\mathcal{A}$ has access to while in combination with a target t, we are interested in the runtime $\mathrm{RT}(f_i, n, j, t)$ of $\mathcal{A}$ to hit the target t (which might fail). Each problem (f_i, n, j) gives raise to a collection of dependent problems (f_i, n, j, t). Viewed as random variables, $\mathrm{RT}(f_i, n, j, t)$ given (f_i, n, j) are not independent for different values of t.

[2] Target values are directly linked to a problem, leaving the burden to properly define the targets with the designer of the benchmark suite. The alternative is to present final f-values as results, leaving the (rather unsurmountable) burden to interpret these values to the reader. Fortunately, there is an automatized generic way to generate target values from observed runtimes, the so-called run-length based target values [20].

ting for the bbob testbed, six different dimensions, increasing by a factor of two up to dimension 640, where the last dimension is again optional. Based on the current implementation of the functions, it is however straightforward to adapt the suite implementation to any set of dimensions, in particular to even larger dimensions. We explain in this section how the bbob-largescale test suite is built.

4.1. *The single-objective* bbob *functions*

The bbob test suite relies on the use of so-called *raw* functions from which 24 bbob functions are generated. A series of transformations on these raw functions, such as linear transformations (e.g., translation, rotation, scaling) and/or non-linear transformations (e.g., $T_{\text{osz}}, T_{\text{asy}}$) is applied to obtain the actual bbob test functions. For example, the test function $f_{13}(\mathbf{x})$ (Sharp Ridge function) with (vector) variable $\mathbf{x}$ is derived from a raw function defined as follows:

$$f_{\text{raw}}^{\text{Sharp Ridge}}(\mathbf{z}) = z_1^2 + 100\sqrt{\sum_{i=2}^{n} z_i^2}.$$

Then one applies a sequence of transformations: a translation by using the vector $\mathbf{x}^{\text{opt}}$; then a rotational transformation $\mathbf{R}$; then a scaling transformation $\mathbf{\Lambda}^{10}$; then another rotational transformation $\mathbf{Q}$ to get the relationship $\mathbf{z} = \mathbf{Q}\mathbf{\Lambda}^{10}\mathbf{R}(\mathbf{x} - \mathbf{x}^{\text{opt}})$; and finally a translation in objective space by using $\mathbf{f}_{\text{opt}}$ to obtain the final function in the testbed:

$$f_{13}(\mathbf{x}) = f_{\text{raw}}^{\text{Sharp Ridge}}(\mathbf{z}) + \mathbf{f}_{\text{opt}}.$$

There are two main reasons behind the use of transformations here:

(i) provide non-trivial problems that cannot be solved by simply exploiting some of their properties (separability, optimum at fixed position, ...) and

(ii) allow to generate different instances, ideally of similar difficulty, of the same problem by using different (pseudo-)random transformations.

Rotational transformations are used to avoid separability and thus coordinate system dependence in the test functions. The rotational transformations

consist in applying an orthogonal matrix to the search space: $x \mapsto z = \mathbf{R}x$, where $\mathbf{R}$ is the orthogonal matrix. While the other transformations used in the `bbob`
270 test suite could be naturally extended to the large-scale setting due to their linear complexity, rotational transformations have quadratic time and space complexities. Thus, we need to reduce the complexity of these transformations in order for them to be usable, in practice, in the large-scale setting.

4.2. Extension to large-scale setting

275 Our objective is to construct a large-scale test suite where the cost of a function call is acceptable in higher dimensions while preserving the main characteristics of the original functions in the `bbob` test suite. To this end, we replace the dense orthogonal matrices of the rotational transformations with orthogonal transformations that have linear complexity in the problem dimension:
280 *permuted orthogonal block-diagonal matrices* [9].

Specifically, the matrix of a rotational transformation $\mathbf{R}$ is represented as:

$$\mathbf{R} = P_{\text{left}} B P_{\text{right}}.$$

Here, P_{left} and P_{right} are two permutation matrices[3] and B is a block-diagonal matrix of the form:

$$B = \begin{pmatrix} B_1 & 0 & \dots & 0 \\ 0 & B_2 & \dots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & B_{n_b} \end{pmatrix},$$

where n_b is the number of blocks and $B_i, 1 \le i \le n_b$ are square matrices of sizes $s_i \times s_i$ satisfying $s_i \ge 1$ and $\sum_{i=1}^{n_b} s_i = n$. If we choose the matrices $B_i, 1 \le i \le n_b$ such that they are all orthogonal, the resulting matrix B is also an orthogonal matrix.

[3] A *permutation matrix* is a square binary matrix that has exactly one entry of 1 in each row and each column and 0s elsewhere.

285 This representation allows the rotational transformation $\mathbf{R}$ to satisfy three desired properties:

1. Have (almost) linear cost (due to the block structure of B).
2. Introduce non-separability.
3. Preserve the eigenvalues and therefore the condition number of the original
290 function when it is convex quadratic (since $\mathbf{R}$ is orthogonal).

4.3. Generating the orthogonal block matrix B

The block-matrices $B_i, i = 1, 2, ..., n_b$ are distributed with the Haar measure, the unique measure that is invariant under group multiplication in the set of orthogonal matrices of the same size [22]. Columns and rows of these matrices are
295 uniformly distributed on the unit hypersphere surface. To create these matrices, we first generate square matrices of sizes s_i $(i = 1, 2, \ldots, n_b)$ whose entries are i.i.d. standard normally distributed and then apply the Gram-Schmidt process to orthogonalize these matrices [22].

The parameters of this procedure include:

300
- the dimension of the problem n,
- the block sizes $s_1, \ldots, s_{n_b}$, where n_b is the number of blocks. In the `bbob-largescale` test suite, we set $s_i = s := \min(n, 40)$ for all $i = 1, 2, ..., n_b$ (except for the last block which can be smaller)[4] and thus $n_b = \lceil n/s \rceil$.

4.4. Generating the permutation matrices P

305 In order to generate the permutation matrices P_{left} and P_{right}, we start from the identity matrix and apply, successively, a set of n_s so-called *truncated uniform swaps*—independently for both matrices. Each row/column chosen in a random order is swapped with a row/column chosen uniformly from the set of rows/columns within a fixed range r_s.

[4] This setting allows to have the problems in dimensions 20 and 40 overlap between the `bbob` test suite and its large-scale extension since in these dimensions, the block sizes coincide with the problem dimensions.

Let i be the index of the first variable/row/column to be swapped, then the index of the second swap variable obeys

$$j \sim U(\{l_b(i), l_b(i)+1, \dots, u_b(i)\} \backslash \{i\}),$$

where $U(S)$ is the uniform distribution over the set S and $l_b(i) = \max(1, i - r_s)$ and $u_b(i) = \min(n, i + r_s)$ with r_s a parameter of the approach. If $r_s \leq (n-1)/2$, the average distance between the first and the second swap variable ranges from $(\sqrt{2}-1)r_s + 1/2$ (in the case of an asymmetric choice for j, i.e. when i is chosen closer to 1 or n than r_s) to $r_s/2 + 1/2$ (in the case of a symmetric choice for j). It is maximal when the first swap variable is at least r_s away from both extremes or is one of them.

Algorithm 1 describes the process of generating a permutation using a series of truncated uniform swaps with the following parameters:

- n, the number of variables,
- n_s, the number of swaps.
- r_s, the swap range.

The order of rows/columns which are chosen as first swap variables is defined by a permutation π, drawn uniformly at random.

Algorithm 1 is applied independently to permute the rows/columns of the matrices P_{left} and P_{right}. In the proposed test suite, we further set $n_s = n$ and $r_s = \lfloor n/3 \rfloor$. Some numerical results in [9] show that with such parameters, the proportion of variables that are moved from their original position when applying Algorithm 1 is approximately 100% for all dimensions 20, 40, 80, 160, 320, and 640 of the `bbob-largescale` test suite.

4.5. Implementation of the permuted orthogonal block-diagonal transformations

Now, we describe how these changes to the rotational transformations are implemented with the realizations of $P_{\text{left}} B P_{\text{right}}$. We illustrate this through an

Algorithm 1 Truncated Uniform Permutations

Require: problem dimension n, number of swaps n_s, swap range r_s.

Ensure: returns a vector $\mathbf{p} \in \mathbb{N}^n$, defining a permutation.

1: $\mathbf{p} \leftarrow (1, \dots, n)$

2: Generate a permutation π uniformly at random

3: **for** $1 \leq k \leq n_s$ **do**

4: $\quad i \leftarrow \pi(k)$, i.e., $\mathbf{p}_{\pi(k)}$ is the first swap variable

5: $\quad l_b \leftarrow \max(1, i - r_s)$

6: $\quad u_b \leftarrow \min(n, i + r_s)$

7: $\quad S \leftarrow \{l_b, l_b + 1, \dots, u_b\} \backslash \{i\}$

8: $\quad$ Sample j uniformly at random in S

9: $\quad$ Swap $\mathbf{p}_i$ and $\mathbf{p}_j$

10: **end for**

11: **return** p

example on the Ellipsoidal function (rotated) $f_{10}(\mathbf{x})$ (see the table in the next section), which is defined by

$$f_{10}(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} 10^{6\frac{i-1}{n-1}} z_i^2 + \mathbf{f}_{\mathrm{opt}}, \text{with } \mathbf{z} = T_{\mathrm{osz}}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}})), \mathbf{R} = P_1 B P_2,$$

as follows:

(i) First, we obtain the three matrices needed for the transformation, B, P_1, P_2, as follows:

```
coco_compute_blockrotation(B, seed1, n, s, n_b);
coco_compute_truncated_uniform_swap_permutation(P1, seed2, n, n_s, r_s);
coco_compute_truncated_uniform_swap_permutation(P2, seed3, n, n_s, r_s);
```

335

Then, wherever in the `bbob` test suite, we use the following

```
problem = transform_vars_affine(problem, R, b, n);
```

to make a rotational transformation, in the `bbob-largescale` test suite we

340 replace it with the three transformations

```
problem = transform_vars_permutation(problem, P2, n);
problem = transform_vars_blockrotation(problem, B, n, s, n_b);
problem = transform_vars_permutation(problem, P1, n);
```

Here, n is again the problem dimension, s the size of the blocks in B, n_b the number of blocks, n_s the number of swaps, and r_s the swap range as presented previously.

4.6. Adjustments of the functions for scalability performance assessments

Apart from the important modification of the applied rotational transformations described above, which aims at reducing the computational cost of evaluating the function values, further adjustments of the test suite's function definitions are made in order to compare the performance of algorithms with increasing dimensions in a correct way.

The goal of these adjustments is twofold. First, the intrinsic difficulty of the test functions should be independent of the dimension. Second, the range of target values should be defined compatible with how the performance is assessed within the COCO framework. Since this is achieved by recording the same target precision values over all problems (fixed within a given range), the function values are rescaled for each function to avoid that target precisions become too easy to reach when the dimension increases. Without this adjustment, even very simple algorithms such as the pure random search may be able to solve a relevant proportion of some test problems, leading to misinterpretations of algorithm performances.

In particular, we made the following three changes to the raw functions in the `bbob` test suite.

- All functions are normalized by dimension. Except for the six functions Schwefel, Schaffer, Weierstrass, Gallagher, Griewank-Rosenbrock and Katsuura, which are already normalized with dimension, the functions are normalized by the parameter $\gamma(n) = \min(1, 40/n)$ to make their target values comparable, in difficulty, over a wide range of dimensions without losing backwards compatibility.
- The Discus, Bent Cigar and Sharp Ridge functions are generalized such that they have a constant proportion of $\lceil n/40 \rceil$ distinct axes that remain

consistent with the `bbob` test suite.

- For the two Rosenbrock functions and the related Griewank-Rosenbrock function, a different scaling is used than in the original `bbob` functions: instead of using the factor $\max(1, \frac{\sqrt{n}}{8})$ with n being the problem dimension, we scale the rotated search vector by the factor $\max(1, \frac{\sqrt{s}}{8})$, where $s = \min(n, 40)$ is the block size in the matrix B. Because $\sqrt{40} < 8$, this corresponds to no scaling. An additional constant is added to the z vector to reduce, with high probability, the risk to move important parts of the test function's characteristics out of the domain of interest. Without these adjustments, the original functions become significantly easier in higher dimensions due to the optimum being too close to the origin. For more details, we refer the interested reader to the discussion on the corresponding GitHub issue [23].

For a better understanding of the properties of these functions and for the definitions of the used transformations and abbreviations, we refer the reader to the original `bbob` function documentation [24].

4.7. Functions in the Suite

Tables 1, 2, and 3 below present the definition of all 24 functions of the `bbob-largescale` test suite in detail.

Table 1: Function descriptions of the separable and moderately conditioned function groups of the `bbob-largescale` test suite.

Group 1: Separable functions

Formulation	*Transformations*
Sphere Function $f_1(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} z_i^2 + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = \mathbf{x} - \mathbf{x}^{\text{opt}}$
Ellipsoidal Function $f_2(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} 10^{6\frac{i-1}{n-1}} z_i^2 + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = T_{\text{osz}}\left(\mathbf{x} - \mathbf{x}^{\text{opt}}\right)$
Rastrigin Function $f_3(\mathbf{x}) = \gamma(n) \times \left(10n - 10\sum_{i=1}^{n} \cos(2\pi z_i) + \|z\|^2\right) + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = \mathbf{\Lambda}^{10} T_{\text{asy}}^{0.2}\left(T_{\text{osz}}\left(\mathbf{x} - \mathbf{x}^{\text{opt}}\right)\right)$
Bueche-Rastrigin Function $f_4(\mathbf{x}) = \gamma(n) \times \left(10n - 10\sum_{i=1}^{n} \cos(2\pi z_i) + \|z\|^2\right) + 100 f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}}$	$z_i = s_i T_{\text{osz}}\left(x_i - x_i^{\text{opt}}\right)$ for $i = 1, \ldots, n$; $s_i = \begin{cases} 10 \times 10^{\frac{1}{2}\frac{i-1}{n-1}} & \text{if } z_i > 0 \text{ and } i \text{ odd} \\ 10^{\frac{1}{2}\frac{i-1}{n-1}} & \text{otherwise} \end{cases}$ for $i = 1, \ldots, n$
Linear Slope Function $f_5(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} (5\|s_i\| - s_i z_i) + \mathbf{f}_{\text{opt}}$	$z_i = \begin{cases} x_i & \text{if } x_i^{\text{opt}} x_i < 5^2 \\ x_i^{\text{opt}} & \text{otherwise} \end{cases}$ for $i = 1, \ldots, n$, $s_i = \text{sign}\left(x_i^{\text{opt}}\right) 10^{\frac{i-1}{n-1}}$ for $i = 1, \ldots, n$, $\mathbf{x}^{\text{opt}} = \mathbf{z}^{\text{opt}} = 5 \times \mathbf{1}_{-}^{+}$

Group 2: Functions with low or moderate conditioning

Formulation	*Transformations*
Attractive Sector Function $f_6(\mathbf{x}) = T_{\text{osz}}\left(\gamma(n) \times \sum_{i=1}^{n} (s_i z_i)^2\right)^{0.9} + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = \mathbf{Q}\mathbf{\Lambda}^{10}\mathbf{R}(\mathbf{x} - \mathbf{x}^{\text{opt}})$ with $\mathbf{R} = P_{11}B_1P_{12}, \mathbf{Q} = P_{21}B_2P_{22}$, $s_i = \begin{cases} 10^2 & \text{if } z_i \times x_i^{\text{opt}} > 0 \\ 1 & \text{otherwise} \end{cases}$ for $i = 1, \ldots, n$
Step Ellipsoidal Function $f_7(\mathbf{x}) = \gamma(n) \times 0.1 \max\left(\|\hat{z}_1\|/10^4, \sum_{i=1}^{n} 10^{2\frac{i-1}{n-1}} z_i^2\right) + f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}}$	$\hat{\mathbf{z}} = \mathbf{\Lambda}^{10}\mathbf{R}(\mathbf{x} - \mathbf{x}^{\text{opt}})$ with $\mathbf{R} = P_{11}B_1P_{12}$, $\tilde{z}_i = \begin{cases} \lfloor 0.5 + \hat{z}_i \rfloor & \text{if } \|\hat{z}_i\| > 0.5 \\ \lfloor 0.5 + 10\hat{z}_i \rfloor / 10 & \text{otherwise} \end{cases}$ for $i = 1, \ldots, n$, $\mathbf{z} = \mathbf{Q}\tilde{\mathbf{z}}$ with $\mathbf{Q} = P_{21}B_2P_{22}$
Rosenbrock Function, original $f_8(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n-1} \left(100\left(z_i^2 - z_{i+1}\right)^2 + (z_i - 1)^2\right) + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = \max\left(1, \frac{\sqrt{s}}{8}\right)(\mathbf{x} - \mathbf{x}^{\text{opt}}) + \mathbf{1}$, $\mathbf{x}^{\text{opt}} \in [-3, 3]^n$
Rosenbrock Function, rotated $f_9(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n-1} \left(100\left(z_i^2 - z_{i+1}\right)^2 + (z_i - 1)^2\right) + \mathbf{f}_{\text{opt}}$	$\mathbf{z} = \max\left(1, \frac{\sqrt{s}}{8}\right)\mathbf{R}(\mathbf{x} - \mathbf{x}^{\text{opt}}) + \mathbf{1}$ with $\mathbf{R} = P_1BP_2$, $\mathbf{x}^{\text{opt}} \in [-3, 3]^n$

Table 2: Function descriptions of the ill-conditioned and adequately structured multimodal function groups of the `bbob-largescale` test suite.

Group 3: Ill-conditioned functions

Formulation	*Transformations*
Ellipsoidal Function $f_{10}(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} 10^{6\frac{i-1}{n-1}} z_i^2 + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = T_{\mathrm{osz}}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_1 B P_2$
Discus Function $f_{11}(\mathbf{x}) = \gamma(n) \times \left(10^6 \sum_{i=1}^{\lceil n/40 \rceil} z_i^2 + \sum_{i=\lceil n/40 \rceil+1}^{n} z_i^2\right) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = T_{\mathrm{osz}}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_1 B P_2$
Bent Cigar Function $f_{12}(\mathbf{x}) = \gamma(n) \times \left(\sum_{i=1}^{\lceil n/40 \rceil} z_i^2 + 10^6 \sum_{i=\lceil n/40 \rceil+1}^{n} z_i^2\right) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{R} T_{\mathrm{asy}}^{0.5}(\mathbf{R}((\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_1 B P_2$
Sharp Ridge Function $f_{13}(\mathbf{x}) = \gamma(n) \times \left(\sum_{i=1}^{\lceil n/40 \rceil} z_i^2 + 100\sqrt{\sum_{i=\lceil n/40 \rceil+1}^{n} z_i^2}\right) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{Q}\mathbf{\Lambda}^{10}\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}})$ with $\mathbf{R} = P_{11} B_1 P_{12}, \mathbf{Q} = P_{21} B_2 P_{22}$
Different Powers Function $f_{14}(\mathbf{x}) = \gamma(n) \times \sum_{i=1}^{n} \lvert z_i \rvert^{\left(2+4\times\frac{i-1}{n-1}\right)} + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}})$ with $\mathbf{R} = P_1 B P_2$

Group 4: Multi-modal functions with adequate global structure

Formulation	*Transformations*
Rastrigin Function $f_{15}(\mathbf{x}) = \gamma(n) \times \left(10n - 10\sum_{i=1}^{n} \cos(2\pi z_i) + \lVert \mathbf{z} \rVert^2\right) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{R}\mathbf{\Lambda}^{10}\mathbf{Q} T_{\mathrm{asy}}^{0.2}\left(T_{\mathrm{osz}}\left(\mathbf{R}\left(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}\right)\right)\right)$ with $\mathbf{R} = P_{11} B_1 P_{12}, \mathbf{Q} = P_{21} B_2 P_{22}$
Weierstrass Function $f_{16}(\mathbf{x}) = 10\Big(\frac{1}{n}\sum_{i=1}^{n}\sum_{k=0}^{11} \frac{1}{2^k} \cos\left(2\pi 3^k (z_i + 1/2)\right) - f_0\Big)^3 + \frac{10}{n} f_{pen}(\mathbf{x}) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{R}\mathbf{\Lambda}^{1/100}\mathbf{Q} T_{\mathrm{osz}}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_{11} B_1 P_{12}, \mathbf{Q} = P_{21} B_2 P_{22}$, $f_0 = \sum_{k=0}^{11} \frac{1}{2^k} \cos(\pi 3^k)$
Schaffers F7 Function $f_{17}(\mathbf{x}) = \left(\frac{1}{n-1}\sum_{i=1}^{n-1}\left(\sqrt{s_i} + \sqrt{s_i}\sin^2\left(50(s_i)^{1/5}\right)\right)\right)^2 + 10 f_{pen}(\mathbf{x}) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{\Lambda}^{10}\mathbf{Q} T_{\mathrm{asy}}^{0.5}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_{11} B_1 P_{12}, \mathbf{Q} = P_{21} B_2 P_{22}$ $s_i = \sqrt{z_i^2 + z_{i+1}^2}, i = 1, \ldots, n-1$
Schaffers F7 Function, moderately ill-conditioned $f_{18}(\mathbf{x}) = \left(\frac{1}{n-1}\sum_{i=1}^{n-1}\left(\sqrt{s_i} + \sqrt{s_i}\sin^2\left(50(s_i)^{1/5}\right)\right)\right)^2 + 10 f_{pen}(\mathbf{x}) + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \mathbf{\Lambda}^{1000}\mathbf{Q} T_{\mathrm{asy}}^{0.5}(\mathbf{R}(\mathbf{x} - \mathbf{x}^{\mathrm{opt}}))$ with $\mathbf{R} = P_{11} B_1 P_{12}, \mathbf{Q} = P_{21} B_2 P_{22}$, $s_i = \sqrt{z_i^2 + z_{i+1}^2}, i = 1, \ldots, n-1$
Composite Griewank-Rosenbrock Function F8F2 $f_{19}(\mathbf{x}) = \frac{10}{n-1}\sum_{i=1}^{n-1}\left(\frac{s_i}{4000} - \cos(s_i)\right) + 10 + \mathbf{f}_{\mathrm{opt}}$	$\mathbf{z} = \max\left(1, \frac{\sqrt{s}}{8}\right)\mathbf{R}\mathbf{x} + \frac{\mathbf{1}}{2}$ with $\mathbf{R} = P_1 B P_2$, $s_i = 100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2$, for $i = 1, \ldots, n-1$, $\mathbf{z}^{\mathrm{opt}} = \mathbf{1}$

Table 3: Function descriptions of the ill-conditioned and adequately structured multimodal function groups of the `bbob-largescale` test suite.

Group 5: Multi-modal functions with weak global structure

Formulation	*Transformations*
Schwefel Function $f_{20}(\mathbf{x}) = -\frac{1}{100n}\sum_{i=1}^{n} z_i \sin\left(\sqrt{\lvert z_i\rvert}\right) + 4.189828872724339 + 100 f_{pen}(\mathbf{z}/100) + \mathbf{f}_{\text{opt}}$	$\hat{\mathbf{x}} = 2 \times \mathbf{1}_-^+ \otimes \mathbf{x}$, $\hat{z}_1 = \hat{x}_1$, $\hat{z}_{i+1} = \hat{x}_{i+1} + 0.25\left(\hat{x}_i - 2\lvert x_i^{\text{opt}}\rvert\right)$, for $i = 1, \ldots, n-1$, $\mathbf{z} = 100\left(\mathbf{\Lambda}^{10}\left(\hat{\mathbf{z}} - 2\lvert\mathbf{x}^{\text{opt}}\rvert\right) + 2\lvert\mathbf{x}^{\text{opt}}\rvert\right)$, $\mathbf{x}^{\text{opt}} = 4.2096874633/2\mathbf{1}_-^+$
Gallagher's Gaussian 101-me Peaks Function $f_{21}(\mathbf{x}) = T_{\text{osz}}\left(10 - \max_{i=1}^{101}\left(w_i \exp\left(-\frac{1}{2n}(\mathbf{z}-\mathbf{y}_i)^T \mathbf{B}^T\mathbf{C_i}\mathbf{B}(\mathbf{z}-\mathbf{y}_i)\right)\right)\right)^2 + f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}}$	$w_i = \begin{cases} 1.1 + 8 \times \frac{i-2}{99} & \text{for } 2 \le i \le 101 \\ 10 & \text{for } i = 1 \end{cases}$ $\mathbf{B}$ is a block-diagonal matrix without permutations of the variables. $\mathbf{C_i} = \Lambda^{\alpha_i}/\alpha_i^{1/4}$, where Λ^{α_i} is defined as usual, but with randomly permuted diagonal elements. For $i = 2, \ldots, 101$, α_i is drawn uniformly from the set $\left\{1000^{2\frac{j}{99}}, j = 0, \ldots, 99\right\}$ without replacement, and $\alpha_i = 1000$ for $i = 1$. The local optima $\mathbf{y}_i$ are uniformly drawn from the domain $[-5,5]^n$ for $i = 2, \ldots, 101$ and $\mathbf{y}_1 \in [-4,4]^n$. The global optimum is at $\mathbf{x}^{\text{opt}} = \mathbf{y}_1$.
Gallagher's Gaussian 21-hi Peaks Function $f_{22}(\mathbf{x}) = T_{\text{osz}}\left(10 - \max_{i=1}^{21}\left(w_i \exp\left(-\frac{1}{2n}(\mathbf{z}-\mathbf{y}_i)^T \mathbf{B}^T\mathbf{C_i}\mathbf{B}(\mathbf{z}-\mathbf{y}_i)\right)\right)\right)^2 + f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}}$	$w_i = \begin{cases} 1.1 + 8 \times \frac{i-2}{19} & \text{for } 2 \le i \le 21 \\ 10 & \text{for } i = 1 \end{cases}$ $\mathbf{B}$ is a block-diagonal matrix without permutations of the variables. $\mathbf{C_i} = \Lambda^{\alpha_i}/\alpha_i^{1/4}$, where Λ^{α_i} is defined as usual, but with randomly permuted diagonal elements. For $i = 2, \ldots, 21$, α_i is drawn uniformly from the set $\left\{1000^{2\frac{j}{19}}, j = 0, \ldots, 19\right\}$ without replacement, and $\alpha_i = 1000^2$ for $i = 1$. The local optima $\mathbf{y}_i$ are uniformly drawn from the domain $[-4.9, 4.9]^n$ for $i = 2, \ldots, 21$ and $\mathbf{y}_1 \in [-3.92, 3.92]^n$. The global optimum is at $\mathbf{x}^{\text{opt}} = \mathbf{y}_1$.
Katsuura Function $f_{23}(\mathbf{x}) = f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}} + \left(\frac{10}{n^2}\prod_{i=1}^{n}\left(1 + i\sum_{j=1}^{32}\frac{\lvert 2^j z_i - [2^j z_i]\rvert}{2^j}\right)^{10/n^{1.2}} - \frac{10}{n^2}\right)$	$\mathbf{z} = \mathbf{Q}\mathbf{\Lambda}^{100}\mathbf{R}(\mathbf{x} - \mathbf{x}^{\text{opt}})$ with $\mathbf{R} = P_{11}B_1P_{12}, \mathbf{Q} = P_{21}B_2P_{22}$
Lunacek bi-Rastrigin Function $f_{24}(\mathbf{x}) = \gamma(n) \times \left(\min\left(\sum_{i=1}^{n}(\hat{x}_i - \mu_0)^2, n + s\sum_{i=1}^{n}(\hat{x}_i - \mu_1)^2\right) + 10\left(n - \sum_{i=1}^{n}\cos(2\pi z_i)\right)\right) + 10^4 f_{pen}(\mathbf{x}) + \mathbf{f}_{\text{opt}}$	$\hat{\mathbf{x}} = 2\text{sign}(\mathbf{x}^{\text{opt}}) \otimes \mathbf{x}$, $\mathbf{x}^{\text{opt}} = 0.5\mu_0\mathbf{1}_-^+$ $\mathbf{z} = \mathbf{Q}\mathbf{\Lambda}^{100}\mathbf{R}(\hat{\mathbf{x}} - \mu_0\mathbf{1})$ with $\mathbf{R} = P_{11}B_1P_{12}$, $\mathbf{Q} = P_{21}B_2P_{22}$, $\mu_0 = 2.5, \mu_1 = -\sqrt{\frac{\mu_0^2 - 1}{s}}$, $s = 1 - \frac{1}{2\sqrt{n+20} - 8.2}$

5. Implementation of the large-scale testbed and repository for datasets

The `bbob-largescale` suite is implemented within the COCO open source project and the code is available in the repository github.com/numbbo/coco. Its test problems are implemented in C based on the COCO problem structure `coco_problem_s`. One main purpose of the COCO platform is to attract researchers from various domains of continuous optimization to assess and compare the performance of their algorithms in a generic black-box setting. Any researcher can provide datasets of benchmarked solvers, which are collected in a publicly available repository and are directly available for comparison with any other solver. Historically, this collection of datasets has been performed through the Black-Box Optimization Benchmarking (BBOB) workshop series. For the `bbob-largescale` test suite, 11 data sets are already available online. In Appendix A we provide a detailed guide on using the COCO platform and in particular the `bbob-largescale` suite, as well as accessing and post processing the datasets collected in the past.

The `bbob-largescale` suite has been used in [25] to analyze the search performance of large-scale CMA-ES [26] variants. However, no details about the used test problems were provided. The study presented in [25] is an example of how the proposed suite allows the differentiation among algorithms. Among other results, it shows that the VkD-CMA-ES (k Vectors and Diagonal Covariance Matrix Evolution Strategy, [27]) overall outperforms the limited memory CMA-ES (LM-CMA-ES) [28] and the RmES (Rank-m Evolution Strategy, [29]) in small dimensions, while LM-CMA-ES shows higher success rates in larger dimensions and for higher budgets. The study also confirms the advantage of an increased population size on the group of multimodal functions with global structure, increasing the success rate by at least a factor of 2 on this function group. It concludes that the L-BFGS algorithm [30] outperforms the large-scale CMA-ES variants for a restricted budget range, after which the best CMA-ES variant has higher success rates. However, over all functions, the cumulative runtime distributions of L-BFGS and the best CMA-ES variant differ by a fac-

tor smaller than 4 in high dimensions, see for example Figure B.2 in Appendix B.[5]

6. Conclusions

In this paper, we proposed a new benchmarking test suite[6] for black-box optimization up to dimension 640 and based on the existing `bbob` test suite of the COCO platform. In contrast to the `bbob` suite, the new `bbob-largescale` suite has linear computational complexity in the dimension which is achieved by replacing orthogonal matrices with permuted orthogonal block-diagonal matrices, previously proposed in [9]. While the new functions are fully backwards comparable with the functions from the `bbob` test suite, additional adjustments were made (i) to have uniform target values that are comparable in difficulty over a wide range of dimensions, (ii) to have a constant proportion of distinct axes that remain consistent with the `bbob` test suite for the Discus, Bent Cigar and Sharp Ridge functions, and (iii) to not make the Rosenbrock functions significantly easier in higher dimensions due to diminishing distances between the optimum and the search space origin when the dimension increases.

Our new suite is a natural extension of the well-established `bbob` suite. By building on the COCO framework with a standardized and established performance assessment procedure, any future benchmarking results can be seamlessly compared with results previously obtained by other researchers. For the new `bbob-largescale` suite, 11 data sets are already online available to compared with. We showcase in the following appendices how automated benchmarking experiments on the `bbob-largescale` test suite can be performed and give examples where the graphical output reveals deficiencies of current large-scale

[5] The same post processed data with [25] are used in the guide of Appendix B, as output example of COCO, where it is clarified how the platform allows the algorithm differentiation and which scientific information we can obtain from the benchmarking procedure.

[6] The source code is available at `https://github.com/numbbo/coco/blob/master/code-experiments/src/suite_largescale.c` as part of the COCO platform.

optimization algorithms.

Acknowledgments

This work was supported by the grant ANR-12-MONU-0009 (NumBBO) of the French National Research Agency. This work was further supported by a public grant as part of the Investissement d'avenir project, reference ANR-11-LABX-0056-LMH, LabEx LMH, in a joint call with Gaspard Monge Program for optimization, operations research and their interactions with data sciences. The PhD thesis of Konstantinos Varelas is funded by the French MoD DGA/MRIS and Thales Land & Air Systems. Tea Tušar acknowledges financial support from the Slovenian Research Agency (research project No. Z2-8177 and research program No. P2-0209) and the European Commission's Horizon 2020 research and innovation program (grant agreement No. 692286).

[1] N. Hansen, S. Finck, R. Ros, A. Auger, Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions, Research Report RR-6829, INRIA (2009).
URL `https://hal.inria.fr/inria-00362633`

[2] N. Hansen, A. Auger, O. Mersmann, T. Tusar, D. Brockhoff, COCO: A platform for comparing continuous optimizers in a black-box setting, arXiv preprint arXiv:1603.08785.

[3] N. Hansen, D. Brockhoff, O. Mersmann, T. Tusar, D. Tusar, O. A. ElHara, P. R. Sampaio, A. Atamna, K. Varelas, U. Batu, D. M. Nguyen, F. Matzner, A. Auger, COmparing Continuous Optimizers: numbbo/COCO on Github, Zenodo (Mar. 2019). `doi:10.5281/zenodo.2594848`.
URL `https://github.com/numbbo/coco`

[4] N. Hansen, A. Auger, R. Ros, S. Finck, P. Pošík, Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009, in: Genetic and Evolutionary Computation Conference (Companion), ACM, New York, NY, USA, 2010, pp. 1689–1696.

[5] A. Auger, N. Hansen, M. Schoenauer, Benchmarking of continuous black
475 box optimization algorithms, Evolutionary Computation 20 (2012) 481.

[6] P. Pošík, V. Klemš, JADE, an adaptive differential evolution algorithm, benchmarked on the BBOB noiseless testbed, in: Genetic and Evolutionary Computation Conference (Companion), ACM, New York, NY, USA, 2012, pp. 197–204.

480 [7] R. Tanabe, A. Fukunaga, Tuning differential evolution for cheap, medium, and expensive computational budgets, in: IEEE Congress on Evolutionary Computation (CEC), IEEE, 2015, pp. 2018–2025.

[8] K. Varelas, M.-A. Dahito, Benchmarking multivariate solvers of scipy on the noiseless testbed, in: Genetic and Evolutionary Computation Confer-
485 ence (Companion), 2019, pp. 1946–1954.

[9] O. Ait Elhara, A. Auger, N. Hansen, Permuted orthogonal block-diagonal transformation matrices for large scale optimization benchmarking, in: Genetic and Evolutionary Computation Conference, ACM, New York, NY, USA, 2016, pp. 189–196.

490 [10] I. Bongartz, A. R. Conn, N. Gould, P. L. Toint, CUTE: Constrained and unconstrained testing environment, ACM Transactions on Mathematical Software (TOMS) 21 (1) (1995) 123–160.

[11] N. I. Gould, D. Orban, P. L. Toint, CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimiza-
495 tion, Computational Optimization and Applications 60 (3) (2015) 545–557.

[12] R. Salomon, Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. a survey of some theoretical and practical aspects of genetic algorithms, BioSystems 39 (3) (1996) 263–278.

[13] A. S. Bondarenko, D. M. Bortz, J. J. Moré, Cops: Large-scale nonlinearly
500 constrained optimization problems, Tech. rep., Argonne National Lab., IL (US) (2000).

[14] K. Tang, X. Yao, P. Suganthan, C. MacNish, Y. Chen, C. Chen, Z. Yang, Benchmark functions for the CEC'2008 special session and competition on large scale global optimization, Tech. rep., University of Science and
505 Technology of China (2007).

[15] K. Tang, X. Li, P. Suganthan, Z. Yang, T. Weise, Benchmark functions for the CEC'2010 special session and competition on large-scale global optimization, Tech. rep., University of Science and Technology of China (2009).

[16] X. Li, K. Tang, M. N. Omidvar, Z. Yang, K. Qin, Benchmark functions
510 for the CEC'2013 special session and competition on large scale global optimization, Tech. rep., Evolutionary Computation and Machine Learning Group, RMIT University, Australia (2013).

[17] A. Kabán, J. Bootkrajang, R. J. Durrant, Toward large-scale continuous eda: A random matrix theory perspective, Evolutionary computation 24 (2)
515 (2016) 255–291.

[18] M. Lozano, D. Molina, F. Herrera, Editorial scalability of evolutionary algorithms and other metaheuristics for large-scale continuous optimization problems, Soft Computing 15 (2011) 2085–2087. doi:10.1007/s00500-010-0639-2.

520 [19] F. Herrera, M. Lozano, D. Molina, M. With, Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems (2010).

[20] N. Hansen, A. Auger, D. Brockhoff, D. Tusar, T. Tusar, COCO: Performance assessment, arXiv e-prints, arXiv:1605.03560 (2016).
525 URL https://hal.inria.fr/hal-01315318

[21] N. Hansen, T. Tusar, O. Mersmann, A. Auger, D. Brockhoff, COCO: The experimental procedure, arXiv e-prints, arXiv:1603.08776 (2016).
URL https://hal.inria.fr/hal-01294167

[22] F. Mezzadri, How to generate random matrices from the classical compact groups, arXiv preprint math-ph/0609050.

[23] COCO GitHub issue #1733, `https://github.com/numbbo/coco/issues/1733` (2018).

[24] R. R. Steffen Finck, Nikolaus Hansen, A. Auger, Real-parameter black-box optimization benchmarking 2010: Presentation of the noiseless functions, Tech. Rep. 2009/20, Research Center PPE, Fachhochschule Vorarlberg, Austria, errata in 2019 (2009).
URL `https://coco.gforge.inria.fr/downloads/download16.00/bbobdocfunctions.pdf`

[25] K. Varelas, A. Auger, D. Brockhoff, N. Hansen, O. A. ElHara, Y. Semet, R. Kassab, F. Barbaresco, A comparative study of large-scale variants of CMA-ES, in: International Conference on Parallel Problem Solving from Nature, Springer, 2018, pp. 3–15.

[26] N. Hansen, A. Ostermeier, Completely derandomized self-adaptation in evolution strategies, Evolutionary Computation 9 (2) (2001) 159–195.

[27] Y. Akimoto, N. Hansen, Online model selection for restricted covariance matrix adaptation, in: International Conference on Parallel Problem Solving from Nature, Springer, 2016, pp. 3–13.

[28] I. Loshchilov, LM-CMA: An alternative to L-BFGS for large-scale black box optimization, Evolutionary Computation 25 (1) (2017) 143–171.

[29] Z. Li, Q. Zhang, A simple yet efficient evolution strategy for large-scale black-box optimization, IEEE Transactions on Evolutionary Computation 22 (5) (2017) 637–646.

[30] D. C. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (3) (1989) 503–528.

555 [31] numbbo/coco: Comparing continuous optimizers. Getting started, https://github.com/numbbo/coco#getting-started- (continuously updating).

[32] J. Moré, S. Wild, Benchmarking Derivative-Free Optimization Algorithms, SIAM J. Optimization 20 (1) (2009) 172–191, preprint available as Mathematics and Computer Science Division, Argonne National Laboratory, 560 Preprint ANL/MCS-P1471-1207, May 2008.

Appendix A. A guide for benchmarking with COCO

The code basis of COCO consists of two parts:

The experiments part. It defines the test suites, allows to conduct the experi-
565 ments and provides the output data to be postprocessed. The code is written in C and wrapped in other languages (currently C/C++, Java, Matlab/Octave and Python), providing an easy-to-use interface. Apart from the currently implemented test suites, COCO allows the definition and integration of new test problems, as well as other functionalities, e.g. data logging options.

570 *The Postprocessing.* It processes the output data from the experimental part, provides the option of processing data from previously archived datasets, and generates various figures and tables presenting aggregated runtime results.

Appendix A.1. Launching experiments

For the installation steps, we refer to the Getting Started guide of COCO
575 [31]. After installation, launching an experiment slightly differs for each language. The `example_experiment` file is modified so that the solver to be benchmarked is connected to COCO and other parameters of the experiment are set. In Python, for which the more recent `example_experiment2.py` file is available, the following additions and modifications compared to the default choices are
580 left to the user:

(i) The necessary imports and the definition of the desired optimizer to be benchmarked:

```python
import scipy.optimize
fmin = scipy.optimize.fmin_l_bfgs_b
```

585 (ii) The selection of the test suite and the maximum budget of function evaluations:

```python
suite_name = "bbob-largescale"
budget_multiplier = 1e4  # times dimension, increase to 10, 100, ...
```

The maximum number of function evaluations on each problem equals to

590 the budget multiplier times the problem dimension. It is highly advisable to run the first experiments with a much smaller budget multiplier, for example 2, 5, or 10.

(iii) The user can optionally filter the suite and perform the experiment on a subset of the suite problems. For example, one can exclude the largest

595 dimension 640 and select specific problem instances:

```python
suite_filter_options = ( "dimensions: 20,40,80,160,320 " +
                         "instance_indices: 1-5 ")
```

(iv) In Python, an automatized way for a parallel execution of the experiment is provided: running the experiment in batches generates a partition of the

600 set of problems of the *filtered, as described above,* suite, and the experiment can be performed in parallel for every batch. The execution time of the experiment can be restrictive, e.g., with a large maximum budget or when high-dimensional problems are benchmarked. Setting:

```python
batches = 1
current_batch = 10
```

605

conducts the experiment only on the first out of ten batches.

(v) Finally, the minimizer has to be added in the restarts loop, where the user can set its specific options, e.g., termination conditions. Stopping information can also be recorded:

```python
while evalsleft() > 0 and not problem.final_target_hit:
    irestart += 1
    if fmin is scipy.optimize.fmin_l_bfgs_b:
        output = fmin(problem, propose_x0(), approx_grad=True,
                      maxfun=evalsleft())
        stoppings[problem.index].append(output[2]['task'])
```

Many of the options for the experimental setting can also be directly set when the code is called from a system shell, like:

```
python example_experiment2.py budget_multiplier=1e4 batch=1/10 suite_name=
    bbob-largescale
```

With the execution of the experiment for the first time, a root folder called `exdata` is created. A new subfolder in `exdata` is created with each launched experiment and, in Python, its name by default contains the solver name, the module from which the solver was imported, the maximum budget and the test suite name. This subfolder contains all the logged data of the specific experiment to be later read by the postprocessing. In case of parallel execution, several subfolders are created, one per each batch, also with the batch number contained in their names. In this case, a folder containing all these subfolders must later be passed to the postprocessing.

The Python experiment prints a timing summary like the following

```
*** Full experiment done in 0h10:37 ***
Timing summary:
  dimension  median seconds/evaluations
  -------------------------------------
     20       8.2e-06
     40       1.0e-05
     80       1.6e-05
    160       2.7e-05
    320       5.2e-05
    640       1.0e-04
  -------------------------------------
```

here taken on a 2019 Macbook Pro with `budget_multiplier=10` and minimal overhead from the solver. Hence, an experiment over all functions, instances and dimensions with `budget_multiplier=10000` and parallelized over 20 CPUs will

take about 10h for the computations of the function evaluations (not accounting for internal solver time). This time requirement is likely to be small compared to the time requirements of the solver.

Practical hint: It is highly recommended to start the experiments with small budgets, before increasing them gradually. Benchmarking data with different budgets can only be postprocessed as data from separate experiments and cannot be merged. However, the idea is to quickly get completed (and independent) data sets for inspection in order to i) track unexpected results indicating a bug in the code early, ii) successively get reliable estimates for the execution time of longer experiments, and iii) be able to inspect chance variations by directly comparing the generated data sets. In addition, the experiment on the `bbob-largescale` test suite can be easily run in parallel batches.

Appendix A.2. Postprocessing

This part of the code, written entirely in Python, aggregates the runtime data to generate various figures and tables in html format and include them into LaTeX documents. Both single algorithm results or comparison results of several algorithms are available. Several ways to aggregate the data are used, and each figure is described in the next section.

Initially, the `cocopp` Python package is installed. Then, executing from a Python shell

```
>>> import cocopp
>>> cocopp.main('[-o OUTPUTFOLDER] YOURDATAFOLDER [MORE_DATAFOLDERS]')
```

or from a system shell:

```
python -m cocopp [-o OUTPUTFOLDER] YOURDATAFOLDER [MORE_DATAFOLDERS]
```

will postprocess the logged data contained in any subfolder of the folder arguments. This allows to collect the data from several batches under root folders, e.g. `YOURDATAFOLDER`. Each one of them corresponds to data from **one** solver. Single-algorithm evaluation results are created in case where only `YOURDATAFOLDER` is given as argument and comparison data when `MORE_DATAFOLDERS` are present. By default, if the `OUTPUTFOLDER` is not specified, the postprocessed

results are stored in a new folder called `ppdata`, and they can be explored from
665 the `ppdata/index.html` file.

Archived data from over 200 algorithms are also provided by COCO for post-processing, 11 of them on the `bbob-largescale` suite, allowing a comparison of a wide range of solvers benchmarked in the past. For example,

```
1 >>> cocopp.archives.bbob_largescale('bfgs')
```

670 lists all available data sets with `'bfgs'` in their name,

```
1 >>> cocopp.main('bbob-largescale/.*bfgs')
```

generates comparison data for all data sets of the list, and

```
1 >>> cocopp.main('bfgs!')
```

postprocesses the first data set with `'bfgs'` in its name (though not necessarily
675 from the `bbob-largescale` suite).

Archived and local data can be mixed for postprocessing, e.g.

```
1 >>> cocopp.main('YOURDATAFOLDER bbob-largescale/2019/LBFGS')
```

The given substring must match a unique data set of the archive. Otherwise, all data sets that match are listed, but none is postprocessed. To display algorithms
680 in the background, the `genericsettings.background` variable can be set as:

```
1 >>> cocopp.genericsettings.background = {None: ['DataFolder1', 'DataFolder2',
      ...]}
```

before running the postprocessing where `None` invokes the default background color and linestyle `cocopp.genericsettings.background_default_style`.

685 For the creation of a single document with the postprocessed results, COCO provides several LaTeX templates that compile the generated tables and figures. For this,

(i) the template with the associated style files must be copied to the directory where the output folder `ppdata` is and

690 (ii) the template can be (optionally) edited, in particular the algorithm name(s).

Appendix B. The Different COCO Graphs: How to Read Them and What Can Be Learned From Them

In this section, we present various graphs and tables generated by the COCO Postprocessing (version 2.3.3) and we explain how they quantify the performance
695 comparison and how they can be interpreted. The shown data compare large-scale variants of the Covariance Matrix Adaptation Evolution Strategy [26] and of L-BFGS [30] on the `bbob-largescale` test suite [25].

Appendix B.1. Runtime distribution graphs (ECDF)

With COCO a benchmarking experiment is recorded as a set of number
700 of function evaluations, also called runtimes, to reach (or surpass) some given target function values on each function and in each dimension. It is natural to display the empirical distribution of these recorded runtimes in empirical cumulative distribution functions (ECDF), denoted as *runtime distributions* in the following. Runtime distributions for a single target value are also known as
705 data profiles [32]. The COCO runtime distribution plots differ in three ways from standard data profiles: (i) the target values do not depend on the shown data; (ii) results for *multiple* targets are aggregated in a single distribution graph; (iii) otherwise undefined runtimes of *unsuccessful* trials are generated by simulated restarts.

710 In general, a runtime distribution or data profile shows the **success rate** on the y-axis, i.e., the proportion of problems solved (in the sense of Section 3), for any given budget on the x-axis (measured in number of function evaluations divided by dimension, #f-evals/dimension). Considering the y-axis as independent, we read for any given fraction of problems (sorted by their runtime)
715 their maximal runtime on the x-axis. As an example, Figure B.1 shows such distributions for six algorithms.

The runtime distribution does not correspond to a single trial: aggregation is over runs with independent restarts and on several instances of a function (Figure B.1 left) or groups of functions (Figure B.1 right). An important remark

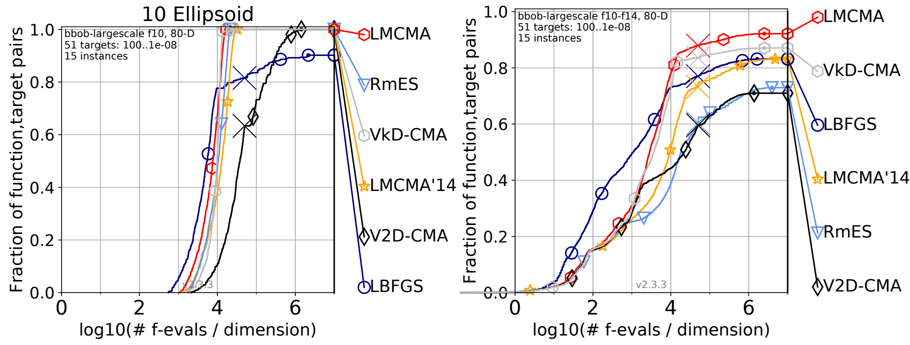

Figure B.1: Bootstrapped runtime distributions for 51 targets in $10^{[-8..2]}$ for a single function (left) and for the group of functions **f10–f14** (right) in dimension 80. **f10–f14** is the group of unimodal functions with high conditioning in the `bbob-largescale` suite.

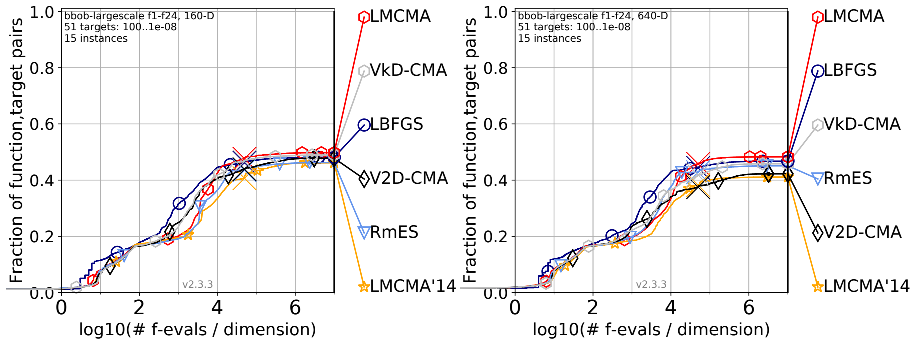

Figure B.2: Bootstrapped runtime distributions of a variety of large-scale solvers, taken from [25]. Shown are 51 targets in $10^{[-8..2]}$ for all functions of the `bbob-largescale` suite in dimension 160 (left) and 640 (right).

720 here is that domination of one algorithm over another in the distribution graph does not necessarily mean that the former is faster on every single problem, due to the fact that the displayed runtimes are sorted by length and hence differently for each algorithm and the information about the underlying function is lost in the graphs.

725 If the success ratio on any given problem is smaller than one but greater than zero, the runtime of unsuccessful trials is determined via simulated restarts from the recorded data of all trials on the very same problem (bootstrapped) thereby mimicking the truly restarted algorithm [20].

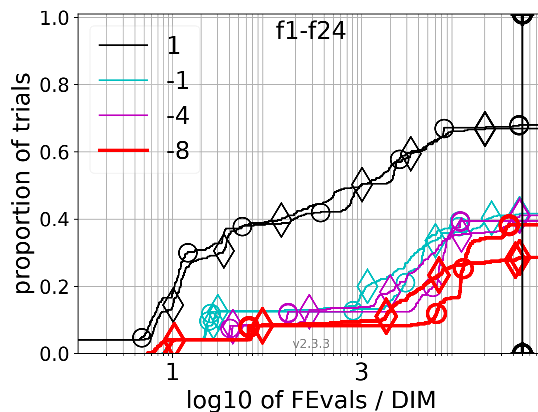

Figure B.3: Runtime distributions for all functions in dimension 320 to reach a target value $\Delta f + f_{opt}$ with $\Delta f = 10^k$, where k is given in the legend, for LMCMA (∘) and VkD-CMA (◊).

Runtime distributions allow a quantified comparison between solvers: a horizontal shift of the graph corresponds to a runtime difference with the respective factor. In the figure for the Ellipsoid function, for example, this comparison would be: Limited memory CMA-ES (LMCMA) [28] is $10^{0.2}$ times faster than Rank-m Evolution Strategy (RmES) [29]. They also can expose possible defects of an algorithm: the same figure shows that L-BFGS does not reach the more difficult target values, suggesting that the finite difference approximation of the gradients deteriorates the performance on the ill-conditioned, non separable Ellipsoid function.

A runtime distribution may contain only runtimes to reach a single target value, instead of several ones. In the case of single-solver or two-solvers data, the Postprocessing generates runtime distribution graphs for selected targets and dimensions, where aggregation is over groups of functions (Figures B.3 and B.4 left). This way, information for easier problems (larger target values) and more difficult ones for the specific function group is now displayed.

Apart from runtime distributions, other quantities are also considered. In the case of single-algorithm data, the Postprocessing provides distribution graphs of the best achieved target value for given budgets of function evaluations (Figure B.4 right). In the case of two solvers, runtime **ratio** distributions of the

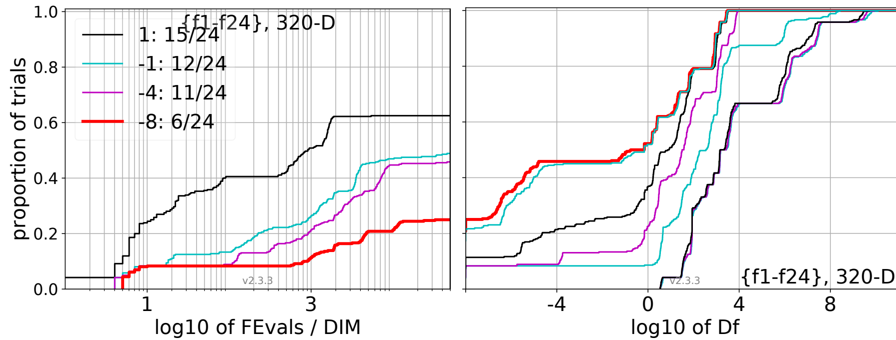

Figure B.4: Left: ECDF of the number of function evaluations of LBFGS divided by search space dimension, to fall below $f_{\text{opt}}+10^k$, where k is the first value in the legend. Right: ECDF of the best achieved target value Δf (shown as Df in the axis label) for budgets of 0.5D, 1.2D, 3D, 10D, 100D, ... function evaluations (from right to left cycling cyan-magenta-black ...) and for the total budget (red).

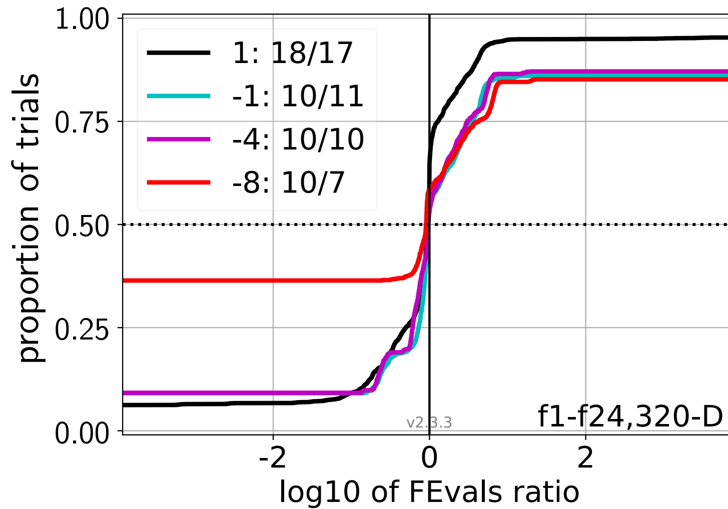

Figure B.5: ECDF of runtime ratios of LMCMA divided by VkD-CMA for all functions in dimension 320 to reach target values 10^k with k given in the legend; all trial pairs for each function. Pairs where both trials failed are disregarded, pairs where one trial failed are visible in the limits being > 0 or < 1. The legend also indicates, after the colon, the number of functions that were solved in at least one trial (LMCMA first).

solvers for selected targets are generated (Figure B.5).

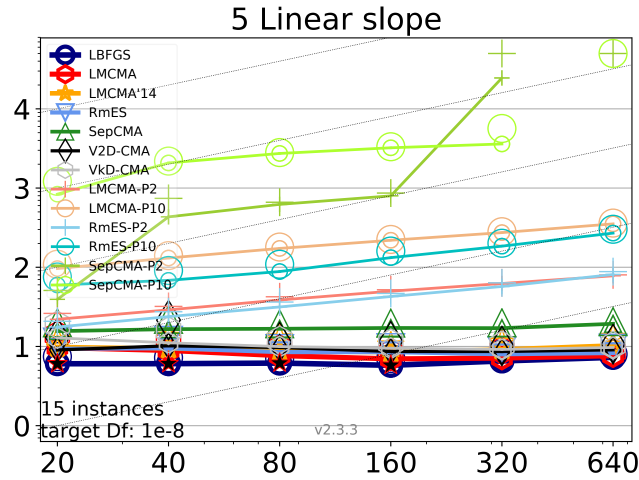

Figure B.6: Expected running time (ERT in number of f-evaluations as $\log_{10}$ value), divided by dimension for target function value 10^{-8}. Black stars indicate a statistically better result compared to all other algorithms with $p < 0.01$ and Bonferroni correction by the number of dimensions (six).

Appendix B.2. Scaling graphs

750 In contrast to runtime distributions that display the ECDF of runtimes for different targets (and potentially different functions), a scaling graph like in Figure B.6 displays the expected (estimated) runtime values (ERT) for a particular function and target value against dimension. As the name indicates, these plots illustrate the scalability of solvers with dimension.

755 Specifically, the scaling graphs show the expected runtimes to reach a certain target function value which are computed as the sum of all function evaluations of the unsuccessful trials, plus the sum of runtimes until the target is hit of successful trials, both divided by the number of successful trials [20].[7]

The ERT values in #f-evals/dimension are plotted versus dimension in a 760 log-log plot, thus a constant graph corresponds to linear scaling. Slanted grid lines indicate quadratic scaling.

Figure B.6 shows the scaling of CMA-ES variants and L-BFGS on the linear

[7]If all trials are successful this is the average runtime.

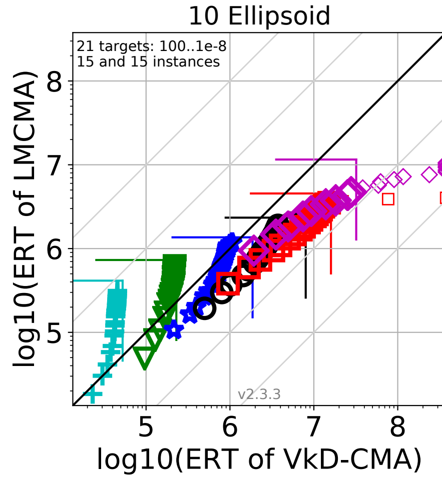

Figure B.7: Expected running time (ERT in $\log_{10}$ of number of function evaluations) of LMCMA (y-axis) versus VkD-CMA (x-axis) for 21 target values between 10^2 and 10^{-8} in each dimension on the Ellipsoid function. Colored markers represent dimension 20:+, 40:▽, 80:⋆, 160:○, 320:□, 640:◇. The rectangle indicates the maximal budget. Small markers indicate that values are computed from simulated restarts (due to some trials being unsuccessful) and markers on the figure edge indicate that the target was never reached by the respective algorithm.

slope function. It is linear for most solvers, except for those with a population size larger than the default (solvers with suffices P2 and P10). Specifically for the separable CMA-ES with larger population sizes, the graphs reveal a performance defect in particular in larger dimension due to the step size adaptation mechanism, as verified after supplementary experiments, see also [25].

Appendix B.3. Scatter plots

In the case of comparison of two solvers, the COCO postprocessing generates *scatter plots* of the algorithms' ERT values for several targets for every function of the suite, see Figure B.7 for an example. The graph is in log-log scale and the first solver corresponds to the y-axis. Each color represents a different dimension.

Scatter plots maintain information for single problems separately (after av-

775 eraging over instances), since for every function and for every target the average runtime is displayed, allowing a comparison between easier and more difficult problems.

Figure B.7 illustrates that on the Ellipsoid function only in dimensions smaller than 80 VkD-CMA (k Vectors and Diagonal Covariance Matrix Evo-780 lution Strategy, [27]) outperforms LMCMA on the difficult target values, by a factor increasing with the target value precision. The picture changes for dimensions larger than 80, where VkD-CMA has worse ERT values on *every* problem. In dimension 160 VkD-CMA is about 2–4 times slower than LMCMA for all targets. In dimensions 320 and 640, VkD-CMA does not reach the most 785 difficult targets anymore.

Appendix B.4. Runtime (ERT) tables

Tables with the expected runtime to reach several target function values are also produced, for every function and dimension. Similarly to the scatter plots, they maintain information on single problems separately, but for a smaller set 790 of target values. They are produced for data of any number of solvers. As an example, a part of the tables comparing LMCMA and VkD-CMA that contains information only for two test functions in dimension 160 is given in Table B.4. In braces, the half difference between 10 and 90 percentiles of runtimes is shown as dispersion measure. The last column gives the number of successful trials 795 to reach the most difficult target $\Delta f + 10^{-8}$. If this target is never reached, the median of conducted function evaluations is given in italics. Finally, a star indicates statistically significantly better results (according to the rank sum test) of a solver when compared to every other algorithm of the table, with $p = 0.05$ or $p = 10^{-k}$ where k is given after the star, and with Bonferroni correction with 800 the number of functions (24).

Table B.4: Excerpt of runtime (ERT) tables generated from COCO here in dimension 160

Δf_{opt}	1e1	1e0	1e-1	1e-2	1e-3	1e-5	1e-7	#succ
f13								
LMCMA	**6418**(333)$^{\star 3}$	**1.8e5**(1e5)	9.7e6(1e7)	**1.1e8**(1e8)	∞	∞	∞ *8e6*	0/15
VkD-CMA	7454(904)	1.9e5(4e5)	**8.8e6**(9e6)	∞	∞	∞	∞ *8e6*	0/15
f14								
LMCMA	**1302**(117)	**3100**(263)$^{\star 3}$	**4439**(268)$^{\star 3}$	**6595**(324)$^{\star 4}$	**1.3e4**(679)$^{\star 4}$	**1.1e5**(6752)$^{\star 4}$	**1.6e6**(1e5)	14/15
VkD-CMA	1457(188)	3607(318)	5261(526)	8789(482)	1.9e4(2054)	2.0e5(4e4)	3.6e6(3e6)	0/15