MPC manuscript No.
(will be inserted by the editor)

MATRS – Heuristic methods for derivative-free bound-constrained mixed-integer optimization

Morteza Kimiaei · Arnold Neumaier

Received: date / Accepted: date

Abstract This paper introduces `MATRS`, a novel matrix adaptation trust-region strategy designed to solve noisy derivative-free mixed-integer optimization problems with simple bounds in low dimensions. `MATRS` operates through a repeated cycle of five phases: mutation, selection, recombination, trust-region, and mixed-integer, executed in this sequence. But if in the mutation phase a new best point (the point with the lowest inexact function value among all evaluated points so far) is found, the selection, recombination, and trust-region phases are skipped. Similarly, if the recombination phase finds a new best point, the trust-region phase is skipped. The mixed-integer phase is always performed. To search for a new best point, the mutation and recombination phases use extrapolation whereas the mixed-integer phase performs a mixed-integer line search along directions estimated to go into a valley. Numerical results on several collections of test problems show that `MATRS` is competitive with state-of-the-art derivative-free mixed-integer solvers.

Keywords Mixed-integer · derivative-free noisy optimization · heuristic optimization · randomized optimization · evolution strategy · trust-region · line search

Mathematics Subject Classification (2020) 90C1 · 90C30 · 90C56 · 90C15

M. Kimiaei
Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090, Wien, Austria
E-mail: kimiaeim83@univie.ac.at

A. Neumaier
Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090, Wien, Austria
E-mail: Arnold.Neumaier@univie.ac.at

1 Introduction

In this paper, we introduce MATRS, a *new mixed-integer matrix adaptation trust-region strategy*, for finding solutions of noisy derivative-free mixed-integer bound-constrained optimization problems

$$\begin{array}{l} \min f(x) \\ \text{s.t. } x \in \mathbf{X},\ x_i \in s_i\mathbb{Z},\ i \in I. \end{array} \tag{1}$$

Here

$$\mathbf{X} := \{x \in \mathbb{R}^n \mid \underline{x} \le x \le \overline{x}\} \ \text{ with } \underline{x}, \overline{x} \in \mathbb{R}^n \ (\underline{x} < \overline{x}) \tag{2}$$

is a **box**, I is a subset of $\{1, \cdots, n\}$, $s_i > 0$ is a resolution factor, and the real-valued function $f : \mathbf{X} \to \mathbb{R}$ is defined on the feasible set

$$C := \{x \in \mathbf{X} \mid x_i \in s_i\mathbb{Z}, \text{ for } i \in I\}. \tag{3}$$

We write x_I and x_K for the subvectors of x indexed by I and K, where I is the index set in (3) and $K := \{1, 2, \ldots, n\} \setminus I$.

Standard mixed-integer problems are covered for $s_i = 1$; other scaling factors define **granular variables** x_i (named so in Audet et al. [3]) whose values are fixed integral multiples of s_i. Granular variables were first handled in the SNOBFIT algorithm (Huyer & Neumaier [21]). They are needed, e.g., if variables are required to be represented in a fixed point format. Standard mixed-integer solvers can handle granular variables by a change of variables $x_i \leftarrow x_i/s_i$.

MATRS is intended for low-dimensional problems only; our numerical tests are restricted to dimensions ≤ 30. The reason is that solving with high reliability high-dimensional derivative-free problems with integer variables is numerically intractable. Our experience with derivative-free problems with only continuous variables (cf. Kimiaei et al. [27]) suggests that sensible heuristic results in high dimensions can be obtained only by using subspace techniques, which solve a sequence of low-dimensional subspace problems of the same kind. We intend to use MATRS as the solver for these subspace problems with a suitable subspace selection technique to be developed.

We assume that the function f is available only by a noisy oracle, returning an approximate function value $\tilde{f}(x)$ of $f(x)$. The **noise** $\tilde{f}(x) - f(x)$ is **deterministic** if calling the oracle repeatedly at the same point returns the same approximate function value, and **stochastic** otherwise. Sources of deterministic noise may be modelling, truncation, and/or discretization errors or rounding errors, and the sources of stochastic noise may be inaccurate measurements or stochastic simulation.

The point with the smallest computed function value among all function values of the points evaluated by MATRS so far is called the **best point** (the true

function value need not be smallest). We denote it by x^{best} and its function value by $\tilde{f}^{\text{best}} = \tilde{f}(x^{\text{best}})$. **Trial points** are points at which the objective function value is evaluated for an achievable improving x^{best}. They are used in line search, direct search, or trust-region techniques. **Trial directions** are directions along which a line search or direct search chooses trial points.

1.1 Related work

DFO techniques for mixed-integer problems are almost always extensions of techniques for continuous solvers that add features for handling integer variables.

Derivative-free optimization (DFO) algorithms have numerous applications in science, engineering, industry, and chemistry. The books of Audet & Hare [2] and Conn et al. [8] and the survey paper of Larson et al. [29] discuss DFO algorithms and their applications.

The survey by Ploskas & Sahinidis [37] investigated the behavior of integer and mixed-integer DFO solvers. `DFLINT` [31] is an integer DFO solver, while `BFO` [38], `DFLBOX` [30], `DFNDFL` [14], `NOMAD` [1], `MISO` [34], `SNOBFIT` [21], and `CMAES` [16] are mixed-integer DFO solvers.

The papers by Moré & Wild [33], Rios & Sahinidis [39], Kimiaei [23], and Kimiaei & Neumaier [26] survey the behavior of DFO algorithms with continuous variables. `DFN` [12], `DFOTR` [4], and `BCDFO` [15] are the three continuous DFO solvers.

Direct search methods evaluate f at trial points obtained from x^{best} by adding coordinate directions, directions from a fixed poll set, or random directions to find a reasonable reduction of the inexact function value. If such a reduction is found, the trial point is accepted as the new best point and the corresponding step size is expanded or remains unchanged. Otherwise, the trial points are discarded and the search is repeated with a reduced step size. `NOMAD` and `BFO` are the two well-known direct search solvers.

Line search methods use extrapolation to quickly leave a saddle point or maximizer in cases where the slope of the function at the current point is small, but no local minimizer is nearby. Extrapolation expands step sizes as long as reductions of inexact function values are found along a fixed random or coordinate direction. As in the direct search methods, the corresponding step size is reduced if no reduction of the inexact function value is found at the trial point. `DFLINT` uses an integer line search and `DFLBOX` uses integer and continuous line searches. `DFLINT` is an extended version of the integer line search of `DFLBOX` and `DFN` is an improved continuous line search method. `DFNDFL` uses `DFLINT` for integer searches and `DFN` for continuous searches.

Space-filling methods use sequences with good space filling properties for various purposes, such as

• the selection of initial points by global solvers (cf. Huyer & Neumaier [19]) with the goal of finding regions close to an approximate stationary point,

• the generation of well-distributed points on a unit simplex for evolutionary multi-objective optimization (cf. Blank et al. [6]),

• the selection of initial sampled points for the construction of quadratic models (cf. Huyer & Neumaier [21]),

• the generation of integer directions for line searches. `DFLINT` [31] calls `generate_dirs` to generate integer directions using the Halton sequences, a family of low-discrepancy sequences (cf. Dick & Pillichshammer [10] and Niederreiter [36]).

Model-based methods approximate the objective function values by quadratic model functions whose approximate gradient and Hessian matrix are obtained by interpolation or fitting. To avoid large steps, these models are constrained by trust regions. The constrained solutions of these models are chosen as the trust-region directions. The trial point is accepted as the new best point if the agreement between the objective function and the model function is good. In this case, the trust-region is extended or remains unchanged. Otherwise, the trial point is discarded and the trust-region is reduced to find small steps for finding a possible reduction of the inexact function value in the next attempt. `NOMAD` provides a model-based direct search method and `SNOBFIT`, `DFOTR` and `BCDFO` use the three various trust-region methods. `MISO` is another model-based solver that uses various types of radial basis functions, sampling techniques, and initial experimental design options.

Matrix adaptation evolution strategies (`MAES`) use a covariance matrix or an affine scaling matrix to define the newly sampled points (e.g., see for the continuous search the recent paper by Kimiaei & Neumaier [25] and for the mixed-integer search the paper by Hansen [16]). They alternate three different phases:

Mutation phase. In this phase, some mutation points are generated, each of which is the sum of the previous recombination point (defined below, initially an initial point) and the mutation direction, scaled by a fixed step size (initially given and then computed in the third phase). Each mutation direction is the product of the corresponding distribution direction and the affine scaling matrix, which is adaptively determined in a heuristic manner. The distribution directions are selected from a normal distribution.

Selection phase. The inexact function values of the mutation points are sorted in ascending order, and the distribution directions and mutation directions are sorted accordingly. Then a finite number of the sorted points with low

inexact function values and the sorted corresponding directions are selected for the next phase.

Recombination phase. The new recombination point is then the sum of the previous recombination point and the recombination mutation direction, scaled by the recombination step size. The fixed step size in the mutation phase is the recombination step size. The recombination mutation direction is a weighted average of the selected mutation directions. The recombination distribution direction, which is a weighted average of the selected distribution directions, is used directly to obtain a new affine scaling matrix and indirectly to calculate a new recombination step size.

DFO methods use descent conditions to reject trial points in regions close to a maximizer or saddle point, and control the size of steps by either step sizes (in a direct search and a line search) or radii (in a trust region) to avoid large steps. However, when noise is large, these DFO methods may get stuck, taking zero steps before identifying an approximate stationary point, because radii (in a trust region) and step sizes (in a direct search and a line search) are reduced whenever a reduction in the inexact function value cannot be found. But this occurs less with a `MAES` method because its step sizes are updated differently in a heuristic way. Hence, `MAES` solvers have more solved problems than the other DFO solvers when noise is large, although they may use more function evaluations than the other DFO methods when noise is small (cf. Kimiaei & Neumaier [25]). In this paper, a new variant of `MAES` is designed using line search and trust-region methods to evaluate mutation and recombination points in such a way that it avoids regions close to a maximizer or saddle point and not only increases the number of solved problems but also reduces the number of function evaluations.

Solving mixed-integer DFO problems is NP hard. In the worst case, all exponentially many integer-feasible combinations must be examined, as the black-box algorithm cannot leverage problem structure. Consequently, any complexity analysis, if feasible, would yield excessively pessimistic results. For this reason, we did not attempt to analyze the complexity of our new method.

1.2 An overview of our new solver

In this section, we summarize the main features of `MATRS`, a new solver based on mixed-integer matrix adaptation trust-region strategy. It is designed to find solutions of noisy derivative-free mixed-integer bound-constrained optimization problems of the form (1).

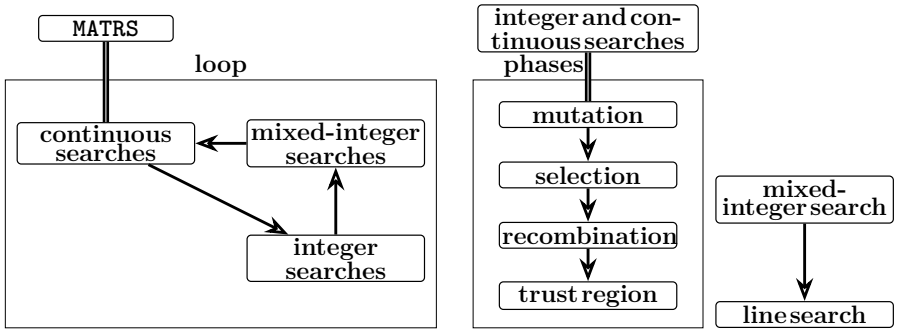


Fig. 1: A generic flowchart for MATRS (left) and how MATRS searches in a space of variables (middle and right).

MATRS alternately performs continuous searches in the space of all x_K, integer searches in the space of all x_I, and mixed-integer searches in the space of all $x_{I\cup K}$ in this order until an approximate stationary point of the problem (1) is found (cf. Fig. 1 and for all subroutines of MATRS and their relations, see Fig. 5, below).

Compared to MAES, MATRS preserves the selection phase, improves the mutation and recombination phases by line searches, and adds two new phases, trust-region and mixed-integer. To update x_{best}, MATRS uses line search strategies in the mutation, recombination, and mixed-integer phases and trust-region strategies in the trust-region phases (cf. Fig. 2).

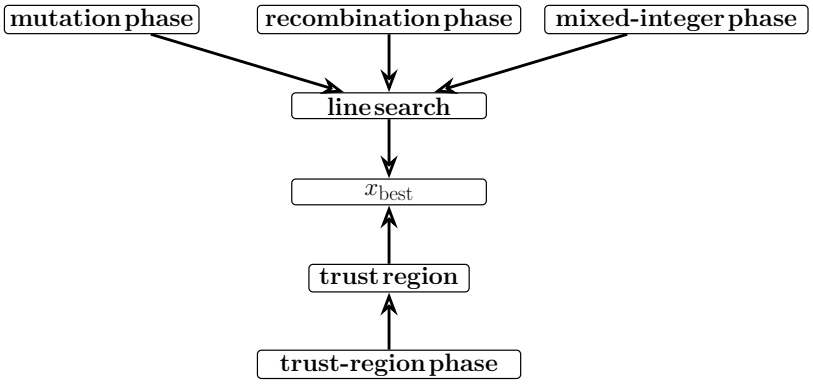


Fig. 2: Flowchart for how MATRS finds x_{best}.

Trial points are evaluated by line searches or trust regions along trial directions. We use three types of trial directions: mutation, recombination and trust-region directions.

In our line search strategies, if only two trial points are evaluated along a trial direction and its opposite direction, so that they cannot be a new best point, one of these two trial points with the lowest inexact function value is selected as either mutation point in the mutation phase or recombination point in the recombination phase, and so the line search ends without updating the best point. Otherwise, extrapolation evaluates at least two trial points either along a trial direction or its opposite direction, one of which with the lowest inexact function value is selected as a new best point.

In our trust-region strategies to find a new best point, a finite number of trial points along a trust-region direction are evaluated. A trust-region direction is computed by finding an approximate solution of a quadratic model, restricted to a trust region. If agreement between the model function and the objective function at a trial point is good, such a trial point is accepted as the new best point and the trust region is either expended or unchanged; otherwise, it is discarded and the trust region is reduced to generate small steps to increase the accuracy of the model function in the next attempts.

Trial points (mutation and recombination points) in the space of all x_I are called **integer trial points** and in the space of all x_K are called **continuous trial points**. Trial directions (mutation, recombination, and trust-region directions) in the space of all x_I are called **integer trial directions**. Trial directions in the space of all x_K are called **continuous trial directions**.

1.2.1 Improved mutation phase

Integer and continuous distribution directions are generated by a new technique with good space-filling properties, discussed in Section 2.

Our improved continuous mutation phase has the following new features:

• A continuous derivative-free line search strategy evaluates continuous trial points along a continuous trial direction or its opposite direction for possible selection as a new best point.

• Real initial step sizes of continuous line searches are found heuristically.

Our improved integer mutation phase has the following new features:

• An integer derivative-free line search strategy evaluates integer trial points along an integer trial direction or its opposite direction for possible selection as a new best point.

• Integer initial step sizes of integer line searches are found heuristically.

1.2.2 Improved recombination phase

Our improved continuous and integer recombination phases have the following new features:

• The weights used to compute the continuous and integer recombination mutation directions are scaled with a randomized scaling vector.

• A new continuous trust-region strategy evaluates continuous trial points for possible selection as a new best point.

• A new integer trust-region strategy evaluates integer trial points for possible selection as a new best point. Its subproblem is transformed in a new way into bound-constrained integer least squares problem.

• The product of the affine scaling matrix and its transpose is used inexpensively as an approximate symmetric Hessian matrix of the model function of the trust-region subproblem.

• Recombination step size is used as a good replacement for the initial trust-region radius.

1.2.3 New mixed-integer phase

Our new mixed-integer phase has the following new features:

• Two new combination directions are computed with the goal of going into or moving down a valley.

• A mixed-integer line search in the space of all x is attempted along exactly one of two combination directions to evaluate trial points, hoping to be one of them as a new best point.

• If the search in the space of all x is not possible, exactly one of integer line search in the space of all x_I and continuous line search in the space of all x_K along exactly one of combination directions is performed.

The integer search of `MATRS` is an improvement of the bound-constrained derivative-free integer optimization solver `IMATRS` discussed in the unpublished manuscript by Kimiaei & Neumaier [24]. `usequence` is not a component of `IMATRS`; rather, it is a novel algorithm here.

1.2.4 Reentrant implementation

We developed a reentrant implementation of the `MATRS` solver for use in software tuning.

A major use of mixed-integer programming software is for tuning algorithms – the task of finding the optimal values for continuous and discrete tuning parameters. There are various applications for tuning such as tuning strategies for quantum chemistry computations [41], auto-tuning solvers for partial differential equations [35], DAG scheduling for tuning distributed systems [42], and compiler tuning [18]. A **tuner** is a program that solves such mixed-integer problems using a mixed-integer solver as oracle for suggesting putative good parameter sets. A reentrant implementation means that the oracle takes a history of the previous feasible points and their function values and returns only a new feasible point for evaluation. Such a subroutine is called **reentrant** if the execution of a solver on a single processing system is stopped when a function value is requested and resumes with a new call once the function value is available. This provides maximal flexibility for its use in a tuner.

To achieve the reentrant behavior, we need to record the place where the algorithm should be continued. This is done using the variable `state`, used in the `MATRSstep` subroutine of the `MATRS` package [28], which encodes the possible places where a function value is required. This subroutine also contains a description of the meaning of each state.

2 Space-filling sequences

This section describes a new technique for generating short sequences with good space-filling properties. Space-filling methods generate sequences x_1, x_2,-$x_3, \ldots$ of points or directions such that for any k the first k points fill the box $[-1, 1]^n$ in a well-distributed way with large minimum distance.

`rand` and `randi` are two random generators in Matlab: rand generates a real random number from uniform distributed in the interval $[0, 1]$, while randi(n) generates uniformly distributed pseudorandom integers between 1 and n.

`haltonset` is a Matlab function to produce points from the Halton sequences. For a highly space-filling set of points, the Halton sequences use distinct prime bases in each dimension.

Our new randomized procedure `usequence` is a randomized procedure for selecting a space-filling set of points, for use in applications where no very long sequence of points is needed. Each but the first point in the sequence is obtained by picking from an initially random reservoir of points m points that have the largest minimum squared distance from the points already selected and replacing these points in the reservoir by a new random point.

Given a vector z with n nonnegative integral components, `usequence` generates sequences of well-distributed points, $x \in \mathbb{R}^n$ with components $x_i \in [-1, 1] \subseteq \mathbb{R}$ (if $z_i = 0$) and $x_i \in [-b; b] \subseteq \mathbb{Z}$ (if $z_i = b > 0$), that are not too close too each

other, no matter which initial segment of the sequence is considered (cf. Fig. 3). `usequence` can be made deterministic by fixing the seed of the random generator. This can be done by setting a parameter `det` = 1 (rather than the default value `det` = 0). Unlike the Halton sequences, alternate calls to our implementation of `usequence` produce unrelated sequences. Thus one must generate a complete space-filling sequence in a single call. This is sufficient for our application.

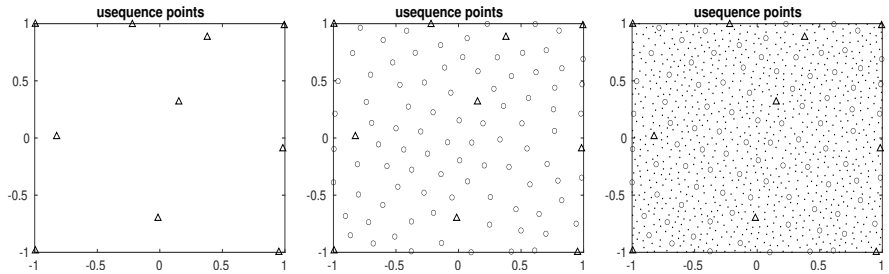


Fig. 3: Plots of 10 (△), 100 (∘), and 1000 (.) real points belonging $[-1,1]^2$ generated by `usequence`.

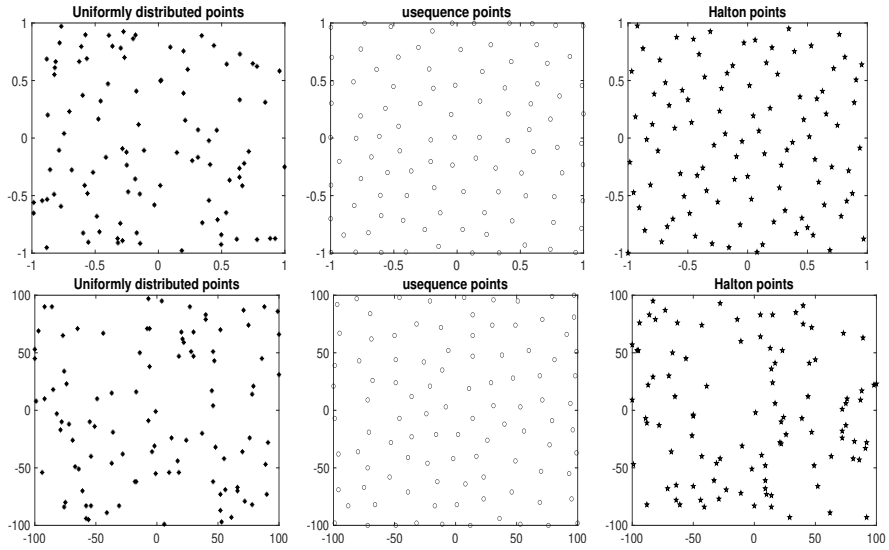


Fig. 4: Plots of points generated by random generators, by `usequence`, and the Halton sequences generated with `haltonset`. First row: The first 100 real points in $[-1,1]^2$. Second row: The first 100 integer points in $[-100:100]^2$.

Fig. 4 displays the distribution of 100 points in dimension $n = 2$ generated with the Halton sequence, with `usequence`, and with a uniform random generator. One can see that the distribution of points generated by `usequence` is markedly superior to the two other distributions. In particular, the Halton sequence suffers from irregularities when the number of points is not very large.

As a quantitative measure of quality, Table 1 tabulates the mean of the minimum distance of the generated point sets for dimensions $n = 3, 10, 15, 30, 100, 300$ and sequence lengths $N = 10, 30, 100, 300, 1000$. We can see that the mean of the minimum distance of points generated by `usequence` is always somewhat larger than those generated by random generators. The minimum distance of the first 10 points generated by the Halton sequences is far from optimal and persists in dimension $n > 10$ when N is much larger.

Algorithm 1 is pseudocode for `usequence`. We denote by $A_{:k}$ the kth column of the matrix A and by $A_{:,i:k}$ the matrix consisting of the columns between the ith and kth columns of A. In this algorithm, j is the index of the reservoir vector with the largest minimum distance from the vectors already chosen and p is the first point of the initial reservoir vector. Moreover, the variable `det` identifies the random generator settings, where 1 represents deterministic and 0 represents nondeterministic. `usequence` chooses a fixed choice `rdet` in line 1. Then, if the variable `det` is true, `usequence` saves the current generator setting in `rsaved` and uses a deterministic generator setting in line 3, which is used to generate all distributed points in lines 6, 7, 14, and 15. In lines 5-9, `usequence` generates an initial random matrix of reservoir points from which a sequence of well-distributed points is constructed. Here randi$([a_i, b_i], n, 1)$ generates integer random vectors whose n components are independent and uniformly distributed in $[a_i, b_i]$ (line 6) and rand$(n, 1)$ is a real random vector whose n components are independent, and uniformly distributed in $[0, 1]$ (line 7). `usequence` updates repeatedly the reservoir in line 18 by replacing the point chosen by a new random point generated in lines 13-17 and computes the minimum squared distance vector u in lines 19-25, where ones$(1, k)$ (line 15) is a $1 \times k$ vector whose components are one and zeros$(1, m)$ (line 21) is a $1 \times m$ vector whose components are zero. Then, in line 26, `usequence` chooses $R_{:j}$, which has the largest minimum squared distance from the reservoir points already selected. To keep randomness of `MATRS`, in line 28, `usequence` returns, for the deterministic case, the generator setting that was already saved in line 3.

		Halton	random generators		`usequence`	
N	n	deterministic	`mean`	`std`	`mean`	`std`
10	3	0.34392	0.35410	0.09586	0.46927	0.16219
10	10	1.01491	1.36889	0.17459	1.78888	0.18426
10	15	1.02151	2.01122	0.27223	2.32942	0.14122
10	30	1.02639	3.40481	0.29386	3.70212	0.43112
10	100	1.02858	7.03948	0.24714	7.24843	0.16617
10	300	1.02896	13.07775	0.22603	13.47416	0.09674
30	3	0.30951	0.14983	0.07240	0.21404	0.07694
30	10	1.01491	1.13884	0.09709	1.17640	0.22266
30	15	1.02151	1.65638	0.08624	1.88823	0.19552
30	30	1.02639	2.92813	0.12631	3.20092	0.18828
30	100	1.02858	6.69735	0.14894	7.12031	0.14249
30	300	1.02896	12.74704	0.20314	12.92384	0.30474
100	3	0.10818	0.06919	0.03235	0.07515	0.02897
100	10	1.01491	0.88073	0.09630	1.01720	0.06064
100	15	1.02151	1.26606	0.13121	1.57610	0.15016
100	30	1.02639	2.68192	0.17538	2.91472	0.13388
100	100	1.02858	6.31852	0.26944	6.61662	0.24021
100	300	1.02896	12.27942	0.20096	12.54167	0.11861
300	3	0.09519	0.03474	0.01385	0.04670	0.01642
300	10	0.95697	0.67310	0.05118	0.72516	0.13277
300	15	1.02151	1.06680	0.11719	1.26560	0.11805
300	30	1.02639	2.40627	0.08391	2.51586	0.19766
300	100	1.02858	6.17384	0.10614	6.27978	0.14626
300	300	1.02896	12.09904	0.20633	12.32334	0.09488
1000	3	0.03897	0.01378	0.00670	0.02063	0.00426
1000	10	0.83144	0.47928	0.05924	0.55778	0.03450
1000	15	1.02151	0.89846	0.07917	1.06050	0.08603
1000	30	1.02639	2.13785	0.10847	2.36120	0.05573
1000	100	1.02858	5.85372	0.09281	6.07526	0.08847
1000	300	1.02896	11.86711	0.14573	12.03653	0.12840

Table 1: Minimum distance of N real points in n dimensions generated by the Halton sequences (deterministic), generated with `haltonset`, and their means (`mean`) and standard deviations (`std`) of 10 runs of random generators and `usequence`.

Algorithm 1 `usequence`

goal: `usequence` generates a well-distributed sequence of points.

function $D = \texttt{usequence}(n, m, z, p, \texttt{det})$

input: n: dimension of the points
m: number of points
z: determine component types ($z_i > 0$: integer, $z_i = 0$: real)
p: first point of the sequence
`det`: generator settings (1: deterministic, 0: non-deterministic)
output: D: matrix whose columns are well-defined random points

assign `rdet`; ▷ a fixed choice
if `det` **then**
  `rsaved` = rand("state"); rand("state",`rdet`);
end if
for $i = 1, \ldots, n$ **do** ▷ generate random reservoir matrix
  if $z_i > 0$, $R_{i:} = \text{randi}([-z_i, z_i], 1, m) \in \mathbb{Z}^{1\times m}$;
  else, $R_{i:} = 2\,\text{rand}(1, m) - 1 \in \mathbb{R}^{1\times m}$;
  end if
end for
$R_{:1} = p$ and $j = 1$;
for $k = 1, \ldots, m$ **do**
  $D_{:k} = R_{:j}$; ▷ next point
  for $i = 1, \ldots, n$ **do** ▷ generate a new random vector
    if $z_i > 0$, $r_i = \text{randi}([-z_i, z_i], 1) \in \mathbb{Z}$;
    else, $r_i = 2\,\text{rand} - 1 \in \mathbb{R}$;
    end if
  end for
  $R_{:j} = r$; ▷ update reservoir
  `on` = ones$(1, k)$; $R' = r_{:,\texttt{on}}$; $D' = D_{:,1:k}$; ▷ forming matrices R' and D'
  $u = \sum_{\ell=1:k} (R'_{:\ell} - D'_{:\ell})^2$; $u_j = \min_{\ell=1:n} u$; ▷ minimum squared distance value
  `on` = k + zeros$(1, n)$; $D'' = D_{:,\texttt{on}}$; ▷ forming the matrix D''
  $s = \sum_{\ell=1:m} (R_{:\ell} - D''_{:\ell})^2$; ▷ distance between R and D''
  if $k = 1$, $u = s$;
  else, $u = \min(u, s)$; ▷ minimum squared distance vector u
  end if
  find $j = \max_{t=1:n} u_t$ and choose $R_{:j} \in \left\{R_{:l}\right\}_{l=1}^{k}$;
end for
if `det` **then** rand("state",`rsaved`); **end if**

3 The MATRS algorithm

MATRS is an improved matrix adaptation trust-region strategy. MATRS alternately performs cMATRS (a continuous MATRS) in the space of all x_K, iMATRS (an integer MATRS) in the space of all x_I, and miMATRS (a mixed-integer MATRS) the spaces of all x, x_K, and x_I in this order until an approximate stationary point of the problem (1) is found.

Fig. 5 is a flowchart for the ingredients of MATRS.

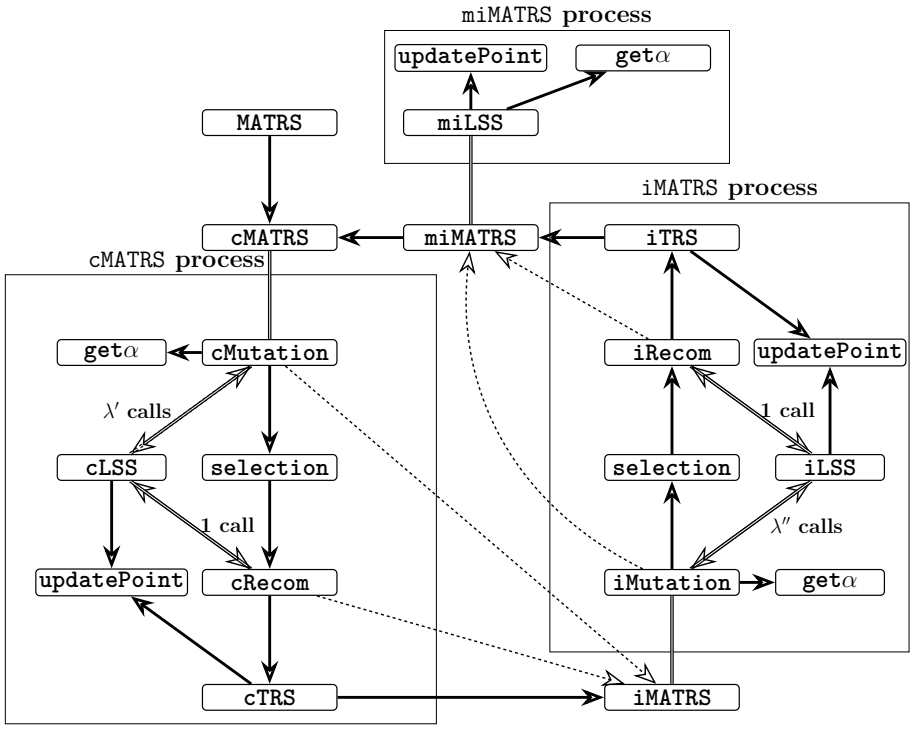


Fig. 5: Flowchart for MATRS. Dashed arrows indicate conditional jumps described in the main text.

MATRS improves its mutation and recombination phases by using line searches to evaluate trial points for possible selection as a new best point, as well as heuristic optimization techniques. Whenever trial points evaluated by line searches do not lead to a new best point, the trust-region strategy evaluates trial points for possible selection as a new best point. Regardless of whether or not trial points evaluated by the line search and trust-region strategies lead

to a new best point, the mixed-integer phase evaluates trial points for possible selection as a new best point.

Continuous searches: `cMATRS` calls `cMutation` (a continuous mutation phase) to generate a finite number λ' of continuous trial points by performing `cLSS` (a continuous line search strategy) along λ' continuous mutation directions. If at least one of these trial points leads to a new best point, `cMATRS` is reduced to `cMutation`, which is a continuous multi-line search. Otherwise, if none of λ' continuous trial points leads to a new best point, these trial points are accepted as continuous mutation points. In this case, `cMATRS` performs `selection` to sort inexact function values at λ' continuous mutation points in ascending order and selects some continuous mutation points, continuous distribution directions, and continuous mutation directions as selected information for `cRecom` (a continuous recombination phase). Then `cMATRS` performs `cRecom`, which computes a continuous recombination mutation direction or possibly its opposite directions along which `cLSS` is performed to evaluate at least one continuous trial point for possible selection as a new best point. If none of these trial points leads to a new best point, `cMATRS` calls `cTRS` (a continuous trust-region strategy) to evaluate a finite number of continuous trial points. `cTRS` uses only some trial points evaluated by `cLSS` in `cMutation` to construct the trust-region subproblem and the recombination step size to compute the initial trust-region radius. Then, it solves the trust-region subproblem whose solution is selected as the trust-region direction.

Integer searches: `iMATRS` calls `iMutation` (an integer mutation phase) to generate a finite number λ'' of integer trial points by performing `iLSS` (an integer line search strategy) along λ'' integer mutation directions. If at least one of these trial points leads to a new best point, `iMATRS` is reduced to `iMutation`, which is an integer multi-line search. Otherwise, if none of λ'' integer trial points can be a new best point, these trial points are accepted as integer mutation points. In this case, `iMATRS` performs `selection` to sort inexact function values at λ'' integer mutation points in ascending order and selects some integer mutation points, integer distribution directions, and integer mutation directions as selected information for `iRecom` (an integer recombination phase). Then `iMATRS` performs `iRecom`, which computes an integer recombination mutation direction or possibly its opposite directions along which `iLSS` is performed to evaluate at least one integer trial point for possible selection as a new best point. If none of these trial points leads to a new best point, `iMATRS` calls `iTRS` (an integer trust-region strategy) to evaluate a finite number of integer trial points. On the one hand, `iTRS` uses only some trial points evaluated by `iLSS` in `iMutation` to construct the trust-region subproblem, and on the other hand, it uses the recombination step size to calculate the initial trust-region radius.Then, it solves the trust-region subproblem whose solution is selected as the trust-region direction.

Reusing old values: Although `iMATRS` cannot guarantee finding different integer feasible points, it uses the history of the old evaluated points to know whether a new trial point is different from the old saved points or not. If this new trial point has not been evaluated before, then its function value is calculated; otherwise, its already known function value is reused.

Mixed-integer searches: `miMATRS` performs `miLSS` (a mixed-integer line search strategy) along combination directions or their opposite directions to evaluate at least one trial point for possible selection as a new best point. One of such directions goes into or move down a valley, at least in one of the spaces of all x, x_I, and x_K in this order. In fact, `miLSS` is a mixed-integer version of `iLSS` and `cLSS`. In some of its iterations, `miLSS` may reduce to `iLSS` in the space of all x_I or `cLSS` in the space of all x_K.

Requirements for line search and trust-region strategies. The largest allowed step sizes are computed by `get`α, which is used as input for `cLSS`, `iLSS`, and `miLSS`. `updatePoint` creates and updates the two different lists of evaluated points and their function values. The first list `XF` saves all evaluated points and their function values. This list is used to check whether or not trial points computed by line search and trust-region strategies are new trial points. The $n + 1$ current trial points and their function values of the first list are used to approximate the gradients of the model functions in the trust-region subproblems. The second list saves and updates at most m best evaluated points in X^{com} and their function values in F^{com} such that function values remain in ascending order in a cheap way. This second list is used to compute combination directions in `miMATRS`. Here $m > 1$ is a tuning parameter.

We wrote a reentrant implementation of the `MATRS` solver. It provides a simple interface for a tuner with `MATRS` as a point oracle. The solver is not passed a function handle but is called many times with one call to `MATRSstep` for each new function evaluation needed (see `driver_reentrant` in [28]). An internal action (a jump or a call) or an external action (an interrupt or a signal) causes the interruption as opposed to recursion. In addition, the reentrant version of `MATRS` uses a call to `MATRSinit` to initialize the solver environment and a call to `MATRSread` to return x^{best} and $\tilde{f}^{\text{best}}$ found by `MATRSstep`. In the Matlab code of `MATRS`, `MATRSinit` and `MATRSread` are part of `MATRSstep` because they must share their persistent variables.

`MATRS` can also be run as a stand-alone solver that evaluates the function thorough a subroutine passed by a function handle (see `driver_stand_alone` in [28]). The stand-alone version just alternately calls `MATRSstep` and the function evaluation routine.

Algorithm 2 is pseudocode for the `MATRS` solver. This solver takes the initial point x^0 as input, the column vector $\underline{x}$ of lower bounds, the column vector $\overline{x}$ of upper bounds, the index set K of continuous variables, the index set I of integer variables, the maximum number `nfmax` of inexact function evaluations,

and tuning parameters. To simplify our notation, we introduce the tuning parameters in Table 2 and do not treat them as input for all pseudocode. `MATRS` returns as output the last best point x^{best} and its inexact function value $\tilde{f}^{\mathrm{best}}$. Table 4 in Section 4 contains the default values for the tuning parameters of `MATRS`. The persistent variables of `MATRS` are `XF`, X^{com}, F^{com}, and `nf`.

Tuning parameters		
`cMATRS`	`iMATRS`	description
λ'	λ''	number of mutation points
μ'	μ''	number of selected mutation points
σ'_{init}	σ''_{init}	initial recombination step size
η'	η''	parameter for the trust-region condition
c'	c''	parameter for updating radii
ν'	ν''	parameter for updating step sizes in line searches
`sc`$'$	`sc`$''$	scaling value for combination directions
σ'_{min}	σ''_{min}	minimum value for recombination step sizes
σ'_{max}	σ''_{max}	maximum value for recombination step sizes
m'_{max}	m''_{max}	upper bound for the norm of the affine scaling matrix
Δ'_{min}	Δ''_{min}	minimum value for trust-region radii
Δ'_{max}	Δ''_{max}	maximum value for trust-region radii
κ'_{max}	κ''_{max}	parameter for updating distribution directions
$\mathbf{a}'_{\mathrm{init}}$	$\mathbf{a}''_{\mathrm{init}}$	list of initial mutation step sizes
M'_{init}	M''_{init}	initial affine scaling matrix
P'_{σ}	P''_{σ}	initial evolution path
α'_{init}	α''_{init}	initial recombination step size
n'_{scale}	n''_{scale}	number of times for scaling recombination mutation direction randomly, so that some trial points can be found
n'_{dd}	n''_{dd}	number of times to compute a distribution direction and a mutation direction so that some trial point can be found
—	$\overline{\Delta}$	upper bound for integer trust-region radii
—	`stuckmax`	maximum number of stucks before finding a new trial point
	`miMATRS`	description
	m	number of saved evaluated points for computing combination directions
	ν	parameter for updating line search step sizes
	α_{init}	initial step size for mixed-integer phase

Table 2: Tuning parameters of `cMATRS`, `iMATRS`, and `miMATRS`.

Details for `MATRS`. In lines 7-8 of `MATRS`, $\mathbf{a}'_{\text{init}}$, $\mathbf{a}''_{\text{init}}$, σ'_{init}, σ''_{init}, α_{init}, α'_{init}, α''_{init}, M'_{init}, and M''_{init} are tuning parameters defined in Table 2. In lines 8-9, P'_σ and P''_σ are the evolutions paths, which are used to compute the affine scaling matrices M' and M'', respectively (see `updateM` in Section 6.3.1) and the recombination step sizes σ' and σ'' (see `cRecom` in Subsection 6.3.2 and `iRecom` in Subsection 6.7). In lines 12 and 14, $x^{\text{best}}_{\texttt{cMATRS}}$, $\tilde{f}^{\text{best}}_{\texttt{cMATRS}}$, $x^{\text{best}}_{\texttt{iMATRS}}$, and $\tilde{f}^{\text{best}}_{\texttt{iMATRS}}$ are chosen and then used as input for `miMATRS` to compute combination directions (defined in Subsection 6.10). Let $n' := |K|$ and $n'' := |I|$. Following [5], the following parameters

$$
\begin{aligned}
&w_i^0 := \ln\left(\mu' + \frac{1}{2}\right) - \ln i, \quad w'_i := \frac{w_i^0}{\sum_{j=1}^{\mu} w_j^0} \quad \text{for } i = 1, \ldots, \mu', \\
&\mu'_w := \frac{1}{\sum_{j=1}^{\mu} w_j^2}, \quad c'_\sigma := \min\left(1.999, \frac{\mu'_w + 2}{n' + \mu'_w + 5}\right), \\
&e'_\sigma := \sqrt{n'}\Big(1 - 1/(4n') - 1/(21{n'}^2)\Big), \quad c'_1 := 2/\Big((n' + 1.3)^2 + \mu'_w\Big), \\
&c'_\mu := \min\left\{1 - c'_1, \frac{2(\mu'_w - 2 + 1)/\mu'_w}{(n' + 2)^2 + \mu'_w}\right\}, \\
&d'_\sigma := 1 + c'_\sigma + 2\max\left\{0, \sqrt{\frac{\mu'_w - 1}{n' + 1}} - 1\right\}
\end{aligned}
$$

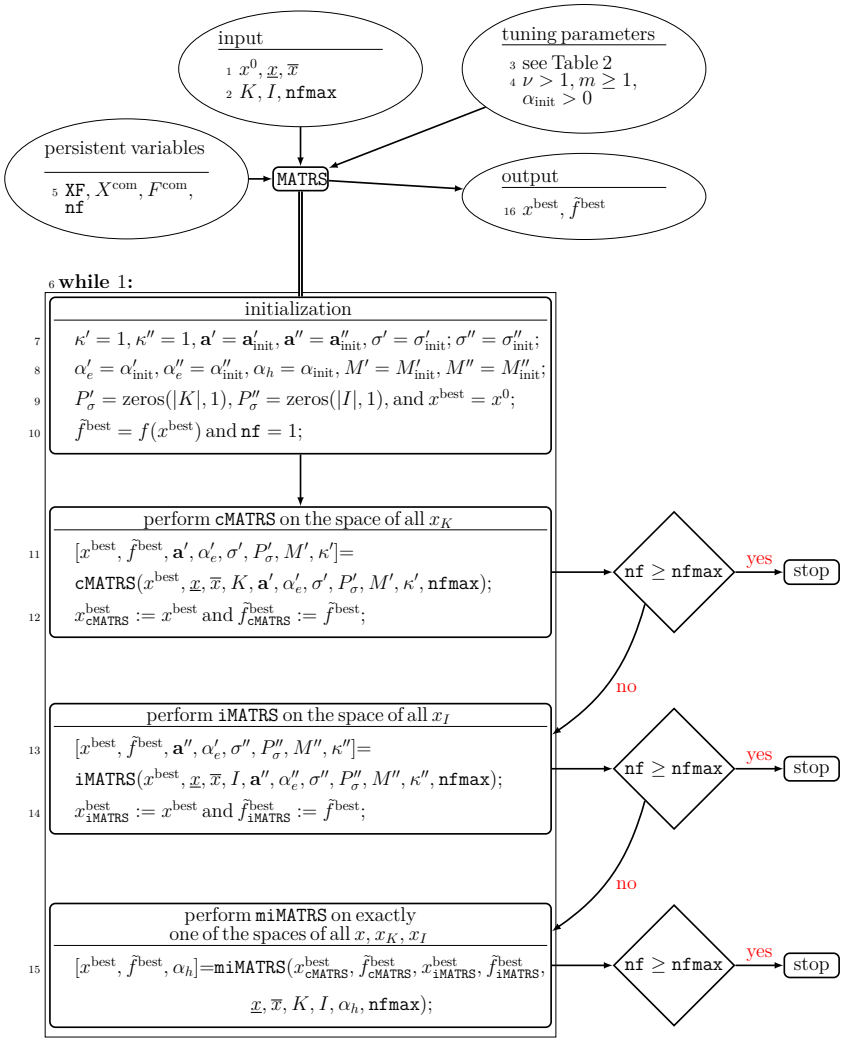
are used to compute the real step size σ' by `cRecom` and the affine scaling matrix M' by `updateM` in each iteration of `cMATRS`. Similar to `cMATRS`, `iMATRS` also computes these parameters, only all single quotation marks are substituted with double quotation marksin the parameter names. To simplify our notation, we do not treat these parameters as input for all pseudocode, except for `updateM`, `iRecom` and `cRecom`.

The function value at each evaluated point is computed as follows. First, `MATRS` takes a history of the previous feasible points and their function values and then computes a new feasible point and the tuner terminates `MATRS`, while returning the new evaluated feasible point and saving the values of parameters are needed for the next start of `MATRS`. Second, the function value at the new evaluated point is computed by any subroutine or simulation outside `MATRS`. Third, `MATRS` takes the new function value as input, updates the history of points and their function values, and goes back to the next place after the new evaluated point has already been evaluated.

3.1 A continuous mutation phase

The goal of `cMutation` is to generate continuous mutation directions and perform the continuous line search strategy `cLSS` along these directions or their

Algorithm 2 Pseudocode for `MATRS`



opposite directions to evaluate at least one continuous trial point for a possible selection as a new best point. This can be achieved by performing extrapolation with at least one more trial point to leave regions near the saddle point or maximizer.

`cMutation` has the following new features: (i) The use of the continuous distribution directions by `usequence` to be well-distributed, neither too close to each other; (ii) determining the initial real step sizes α^i_{init} $(i = 1, \ldots, \lambda')$ for `cLSS` based on the largest allowed real step sizes $\overline{\alpha}^i$ $(i = 1, \ldots, \lambda')$, the real

recombination step size σ', and the list $\mathbf{a}'$ of real mutation step sizes that are neither too small nor too large avoiding line search failure; (iii) updating $\mathbf{a}'$ in a new way that affects the determination of α_{init}^i $(i = 1, \ldots, \lambda')$; (iv) evaluating continuous trial points with `cLSS` for selection as either a new best point or continuous mutation points.

`cMutation` performs a continuous mutation phase with repeated calls to `get`α, `updatePoint`, and `cLSS`. Here `get`α finds the $\overline{\alpha}^i$ preserving feasibility.

`updatePoint` creates and updates the two lists of evaluated points and their function values with the goals of approximating the model gradients and computing the combination directions. If the inexact function value at the first trial point cannot be reduced, `cLSS` accepts the first trial point as the ith mutation point and ends. Otherwise, `cLSS` uses extrapolation along continuous mutation directions or their opposite directions to evaluate at least one more trial point in regions far from the saddle point or maximizer. Algorithms 3–5 in Subsections 6.1.1–6.1.3 of Appendix A discuss pseudocode for `get`α, `updatePoint`, and `cLSS`. Algorithm 6 in Subsections 6.1.4 of Appendix A gives pseudocode for `cMutation`.

3.2 Selection

The goal of the selection phase is to sort the points and directions obtained from the integer and continuous mutation phases such that points with low inexact functions values and the corresponding directions are selected for use in the recombination phase to compute the recombination distribution direction, the recombination mutation direction, the recombination point and step size, and the affine scaling matrix. Algorithm 7 in Subsection 6.2 of Appendix A gives pseudocode for `selection`.

3.3 A continuous recombination phase

The goal of `cRecom` is to compute the continuous recombination mutation direction and perform `cLSS` along this direction or its opposite direction to evaluate at least one trial point for possible selection as a new best point. This can be achieved by performing extrapolation to evaluate at least one more trial point in regions far from the saddle point or maximizer.

`cRecom` has the following new features: (i) The scale of the weights of the recombination distribution and mutation directions in a randomized way with the goal of reordering a fair sort in the selection phase due to noise; (ii) the determination of the initial real step size for `cLSS` that is neither too small nor too large to perform successful extrapolation.

`cRecom` computes the continuous recombination distribution direction p_{dd}, the continuous recombination mutation direction p_{rmd}, the initial real step size α_{init}, and the maximum allowed real step size $\overline{\alpha}$, which is computed by `get`α. Motivated by [25], it then performs `cLSS` along the continuous recombination mutation direction or its opposite direction to evaluate at least one trial point. If there is no decrease in the function value at the first trial point, this point is accepted as a new continuous recombination point. Otherwise, the first trial point is considered as the first trial point evaluated by extrapolation and then at least one more trial point is evaluated, a trial point with the lowest inexact function value evaluated by extrapolation is chosen as a new best point. `updateM` updates the affine scaling matrix $M \in \{M', M''\}$, which is used to compute the mutation directions in `cMutation`. Algorithms 8–9 in Subsections 6.3.1–6.3.2 of Appendix A discuss pseudocode for `updateM` and `cRecom`.

3.4 `cTRS`

When none of the trial points evaluated by `cLSS` performed by `cMutation` and `cRecom` is a new best point, `cTRS` is performed with the goal of evaluating a finite number of trial points for possible selection as a new best point. This goal can be achieved by avoiding large steps, which is one of the causes of the failure of `cLSS`, and generating trial points in regions far from the saddle point or maximizer that may not be searched by `cLSS`.

`cTRS` has the following new features: (i) The use of recombination step sizes σ' (neither too small nor too large) in the computation of the initial real trust-region radius Δ' to overcome the sensitivity of choosing this initial radius; (ii) the use of the product of the affine scaling matrix M' and its transpose as a cheap approximation to the Hessian matrix of the model function of the trust-region subproblem; (iii) avoiding getting stuck before finding an approximate stationary point by terminating `cTRS` in cases where the trust-region radius Δ' is below a given threshold $0 < \Delta'_{\mathrm{min}} < 1$.

Until $\Delta' > \Delta'_{\mathrm{min}}$, `cTRS` forms the trust-region subproblems and solves them to compute trust-region directions. Then, it evaluates trial points for a possible selection as a new best point. If the inexact function value at a trial point is reduced, the trust-region iteration is successful, such a trial point is accepted as the new best point, and Δ' is increased. Otherwise, it is unsuccessful and Δ' is reduced. Algorithm 10 in Subsection 6.4 of Appendix A discusses pseudocode for `cTRS`.

3.5 cMATRS

The goal of cMATRS is to update the best point x^{best}. This can be achieved by performing extrapolation or trust-region strategy to evaluate trial points in regions far from the saddle point or maximizer for possible selection as a new best point.

Whenever cMutation cannot update x^{best} by performing cLSS, the set of distribution directions selected from the normal distribution is changed to a new set whose well-distributed directions are generated by usequence and used in the next call to cMutation to update x^{best}. As long as at least one of trial points evaluated by cLSS can be selected as the new x^{best}, cMATRS skips selection, cRecom, and cTRS. In this case, cMATRS is actually reduced to cMutation, which is a continuous multi-line search due to λ' calls to cLSS. Otherwise, if none of the λ' trial points is chosen as the new x^{best}, such points are chosen as continuous mutation points. Then, selection, cRecom, and possibly cTRS in this order are performed to generate a finite number of trial points for possible selection as the new x^{best}. Algorithm 11 in Subsection 6.5 of Appendix A discusses pseudocode for cMATRS.

Here usequence generates a sequence of finite continuous random vectors in the K-dimensional unit cube plus the continuous combination direction p_K^{init} (computed by cMATRS in line 10) that for each leading subsequence, arbitrary vectors are not too close to each other. $p = p_K^{\text{init}}$ is chosen as input for usequence.

3.6 An integer mutation phase

The goal of iMutation is to generate integer mutation directions and perform the integer line search strategy iLSS along these directions or their opposite directions to evaluate at least one integer trial point for a possible selection as a new best point. This can be achieved by performing extrapolation with at least two integer trial points to leave regions near the saddle point or maximizer.

iMutation has the following new features: (i) The computation of the integer distribution directions by usequence to be well-distributed, neither too close to each other; (ii) determining the initial integer step sizes α_{init}^i $(i = 1, \ldots, \lambda'')$ for iLSS based on the largest allowed integer step sizes $\overline{\alpha}^i$ $(i = 1, \ldots, \lambda'')$, the integer recombination step size σ'', and the list $\mathbf{a}''$ of integer mutation step sizes that are neither too small nor too large avoiding line search failure; (iii) updating $\mathbf{a}''$ in a new way that affects the determination of α_{init}^i; (iv) evaluating integer trial points with iLSS for selection as either a new best point or integer mutation points.

`iMutation` computes integer distribution and integer mutation directions, integer mutation points, and the requirements (such as α_{init}^i, $\overline{\alpha}^i$, and $\mathbf{a}_i''$ for $i = 1, \ldots, \lambda''$) for λ'' calls to `iLSS`. `iMutation` has the same structure as `cMutation` and the same goal, but with differences in distribution directions and updating step sizes. A brief discussion of pseudocode for `iMutation` is given in Subsection 6.6 of Appendix A.

3.6.1 `iLSS`

The goal of `iLSS` is to use extrapolation to evaluate integer trial points in regions far from the saddle point or maximizer.

`iLSS` has the same structure as `cLSS` but it takes the two integer values α_{init}^i and $\overline{\alpha}^i$ for $i = 1, \ldots, \lambda''$ and generates integer step sizes. Whenever the function value is computed at each trial point, the function value and the corresponding point are restored to a list used to approximate the gradient in `iTRS` below, and check whether or not the evaluated point is a new point and has not yet been evaluated. This is only important in the integer case to have distinct points.

3.7 `iRecom`

The goal of `iRecom` is to look for a new best point by computing the integer recombination mutation direction and performing `iLSS` along this direction or its opposite direction to evaluate at least one trial point for a possible selection as a new best point. `iLSS` uses extrapolation to leave regions near the saddle point or maximizer.

`iRecom` has the following new features: (i) The scale of the weights of the integer recombination mutation direction in a randomized way with the goal of reordering a fair sort in the selection phase due to noise; (ii) the determination of the initial integer step size for `iLSS` that is neither too small nor too large to perform successful extrapolation.

`iRecom` has the same structure as `cRecom`, but differences that step sizes are rounded to integer and non-integer components of directions are rounded to integer. A brief discussion of pseudocode for `iRecom` is given in Subsection 6.7 of Appendix A.

3.8 iTRS

The goal of iTRS is to look for a new best point by avoiding large integer steps, which are one of the causes of the failure of iLSS, and evaluating trial points in regions far from the saddle point or maximizer that may not be searched by iLSS performed by iMutation and iRecom.

iTRS has the following new features: (i) The use of integer recombination step sizes σ'' (neither too small nor too large) as the initial trust-region radius in each call to iTRS; (ii) the use of the product of the affine scaling matrix M'' and its transpose as a cheap approximation to the symmetric Hessian matrix of the model function of the trust-region subproblem; (iii) the transformation of the trust-region subproblems into bound-constrained integer least squares problems.

As long as the integer trust-region radius is not below one, iTRS solves bound-constrained integer least squares problems to compute the integer trust-region directions, evaluates integer trial points, and checks whether or not decrease in the inexact function values at such trial point is found. A trial point whose function value is reduced is accepted as a new best point iTRS ends; otherwise, the integer trust-region radius is reduced. A brief discussion of pseudocode for iTRS is given in Subsection 6.8 of Appendix A.

3.9 iMATRS

The goal of iMATRS is to update the best point x^{best}. This can be achieved by performing extrapolation or iTRS to evaluate integer trial points in regions far from the saddle point or maximizer.

Whenever iMutation cannot update x^{best} by performing iLSS, the set of integer distribution directions is changed to a new set whose well-distributed directions are generated by usequence and used in the next call to iMutation to update x^{best}. As long as at least one of trial points evaluated by iLSS can be selected as the new x^{best}, iMATRS skips selection, iRecom, and iTRS. In this case, iMATRS is actually reduced to iMutation, which is an integer multi-line search due to λ'' calls to iLSS. Otherwise, if none of the λ'' trial points is chosen as the new x^{best}, such points are chosen as integer mutation points. Then, selection, iRecom, and possibly iTRS in this order are performed to generate a finite number of trial points for possible selection as a new best point. A brief discussion of pseudocode for iMATRS is given in Subsection 6.9 of Appendix A.

Here usequence generates a sequence of finite integer random vectors in the I-dimensional unit cube plus the integer combination direction p_I^{init} that for

each leading subsequence, arbitrary vectors are not too close to each other. This differs from the method of Liuzzi et al. [31], which uses Halton sequences to generate integer directions. $p = p_I^{\text{init}}$ is chosen as input for `usequence`.

3.10 A mixed-integer `MATRS`

The goal of `miMATRS` is to look for a significant decrease in the inexact function value by performing `miLSS` along combination directions or their opposite directions to evaluate trial points in regions far from the maximizer or saddle point.

`cMATRS` and `iMATRS` may generate many small improved steps. By accumulating these steps, a combination direction is computed, which starts at a point with a small inexact function value and reaching a point with smaller inexact function value. Then, the iterations of `MATRS` can enter a valley and move down to make further progress on the inexact function value.

After performing `cMATRS` and `iMATRS` by `MATRS`, regardless of whether or not x^{best} is updated, `miLSS` is performed along exactly one of the two new combination directions, or possibly their opposite directions, in the space of all x; otherwise, exactly in one of the spaces of all x_K and x_I in this order. Our experiments are shown that searching in the space of all x after searching in the space of all x_K and in the space of all x_I improves the efficiency and robustness of our algorithm. Algorithms 12–13 in Subsections 6.10.1–6.10.2 of Appendix A discuss pseudocode for `miLSS` and `miMATRS`.

4 Numerical results

In this section, we discuss numerical results, comparing `MATRS` with state-of-the-art mixed-integer solvers on problems with dimensions $n \leq 30$.

4.1 Codes compared

We compare our solver `MATRS` with the four mixed-integer solvers, `CMAES` of Hansen [16], `BFO` of Porcelli & Toint [38], `NOMAD` of Abramson et al. [1], and `MISO` of Müller [34], `DFLBOX` of Liuzzi et al. [30], the two continuous solvers `DFOTR` of Bandeira et al. [4], `BCDFO` of Gratton et al. [15], and the integer solver `DFLINT` of Liuzzi et al. [31] on test problems form the `BARON` collection of Sahinidis [40] for the dimensions $2 \leq n \leq 30$. The details of codes compared are summarized in Table 3.

solver	authors, code source, problem type, algorithm type
MATRS	the present solver https://github.com/GS1400/MATRS variable type: mixed-integer, constraint type: bound-constrained algorithm type: matrix adaptation trust-region strategy
CMAES	Hansen [16] http://www.cmap.polytechnique.fr/~nikolaus.hansen variable type: mixed-integer, constraint type: bound-constrained algorithm type: covariance matrix adaptation evolution strategy
BFO	Porcelli & Toint [38] https://github.com/m01marpor/BFO variable type: mixed-integer, constraint type: bound-constrained algorithm type: direct search method
NOMAD	Abramson et al. [1] https://www.gerad.ca/nomad/ variable type: mixed-integer, constraint type: all kinds of constraint algorithm type: model-based direct search method
MISO	Müller [34] obtained from Juliane Müller (personal communication) variable type: mixed-integer, constraint type: bound-constrained algorithm type: model-based, sampling, radial basis functions
DFLBOX	Liuzzi et al. [30] http://www.iasi.cnr.it/~liuzzi/DFL/ variable type: mixed-integer, constraint type: bound-constrained algorithm type: line search method
DFLint	Liuzzi et al. [31] http://www.iasi.cnr.it/~liuzzi/DFL/ variable type: integer, constraint type: bound-constrained algorithm type: line search method
DFOTR	Bandeira et al. [4] https://coral.ise.lehigh.edu/lnv/dfo-tr/ variable type: continuous, constraint type: unconstrained algorithm type: trust-region method
BCDFO	Gratton et al. [15] obtained from Anke Troeltzsch (personal communication) variable type: continuous, constraint type: bound-constrained algorithm type: trust-region method

Table 3: A list of DFO solvers. We selected MISO-CPTV of MISO from [34, Table 1] and renamed them MISO. Since DFOTR is an unconstrained solver, each evaluated point is projected into $\mathbf{X}$.

All solvers compared were used with the default parameters, except for NOMAD that uses the following option set

$$\texttt{opts} = \texttt{nomadset}(\text{`}\texttt{max\_eval}\text{'},\texttt{nfmax},\text{`max\_iterations'},\\ 2{*}\texttt{nfmax},\text{`model\_search'},\text{`1'})$$

and that DFLBOX uses alfa_stop $= -\infty$. Unfortunately, the source code of DFNDFL [14] is Python and we could not run it in Matlab. MATRS is used with the default values for its tuning parameters, given in Table 4.

`cMATRS`	$\lambda' = \max(6, \|K\|)$, $\mu' = 3 + \lceil \log(\|K\|) \rceil$, $\sigma'_{\rm init} = 1$, $\eta' = 10^{-20}$, $c' = \nu' = 4$, $\sigma'_{\rm min} = 10^{-10}$, $\mathtt{sc}' = 10$, $\Delta'_{\rm min} = 10^{-3}$, $m'_{\rm max} = 100$, $\Delta'_{\rm max} = 1$, $\kappa'_{\rm max} = 20$, $\sigma'_{\rm max} = 10^{10}$, $n'_{\rm scale} = 100$, $n'_{\rm dd} = 100$, $\alpha'_{\rm init} = 1$, $\mathbf{a}'_{\rm init} = \text{ones}(\|K\|, 1)$, $M'_{\rm init} = \text{eye}(\|K\|, \|K\|)$, $P'_\sigma = \text{zeros}(\|K\|, 1)$
`iMATRS`	$\lambda'' = \max(6, \|I\|)$, $\mu'' = 3 + \lceil \log(\|I\|) \rceil$, $\sigma''_{\rm init} = 1$, $\eta'' = 10^{-20}$, $c'' = \nu'' = 8$, $\sigma''_{\rm min} = 1$, $\mathtt{sc}'' = 5$, $\Delta''_{\rm min} = 10$, $m''_{\rm max} = 5$, $\Delta''_{\rm max} = 30$, $\kappa''_{\rm max} = 30$, $\sigma''_{\rm max} = 100$, $n''_{\rm scale} = 100$, $n''_{\rm dd} = 100$, $\alpha''_{\rm init} = 1$, $\mathbf{a}''_{\rm init} = \text{ones}(\|I\|, 1)$, $M''_{\rm init} = \text{eye}(\|I\|, \|I\|)$, $P''_\sigma = \text{zeros}(\|I\|, 1)$, $\overline{\Delta} = 3$
`miMATRS`	$m = 5$, $\nu = 4$, $\alpha_{\rm init} = 1$, $\lambda' = \max(3, \|K\|)$, $\mu' = 2 + \lceil \log(\|K\|) \rceil$, $\lambda'' = \max(3, \|I\|)$, $\mu'' = 2 + \lceil \log(\|I\|) \rceil$.

Table 4: The default values for tuning parameters of `MATRS`. Here the Matlab function eye(n, n) denotes an identity matrix. `MATRS` use the three parameters m, ν, and $\alpha_{\rm init}$ to solve mixed-integer problems. The values of λ', λ'', μ', and μ'' for these problems are less than those that are utilized for integer and continuous problems.

4.2 Test problems

To construct mixed-integer test problems, we followed [24] and modified three collections of test problems, namely the collections `global`, `bcp`, and `prince` from the `BARON` collection of Sahinidis [40] for the dimensions $2 \leq n \leq 30$, available at

`https://minlp.com/optimization-test-problems`.

Fig. 6 plots the number of problems with $n \leq d$. As in the paper [32, (16)] for discrete bound-constrained optimization problems, we define

$$\underline{x}_i := x_i^0 - 10, \quad \overline{x}_i := x_i^0 + 10, \quad \text{for } i = 1, 2, \cdots, n,$$

and generate the continuous bound-constrained optimization problem

$$\begin{aligned} &\min \Phi(x) \\ &\text{s.t. } \underline{x}_i \leq x_i \leq \overline{x}_i, \quad \text{for } i = 1, 2, \cdots, n. \end{aligned}$$

Here we denote by $x^0 \in \mathbb{R}^n$ the standard initial points of unconstrained test problems from all above collections and choose $x \in \mathbb{R}^n$. Then, we transform this problem into the bound-constrained mixed-integer optimization problem

$$\begin{aligned} &\min f(x) := \Phi\left(\begin{pmatrix} \underline{x}_I + 0.01(\overline{x}_I - \underline{x}_I) x_I \\ x_K \end{pmatrix}\right) \\ &\text{s.t. } 0 \leq x_i \leq 100, \quad \text{for } i \in I, \\ &\quad\ x_i \leq x_i \leq \overline{x}_i, \quad \text{for } i \in K. \end{aligned}$$

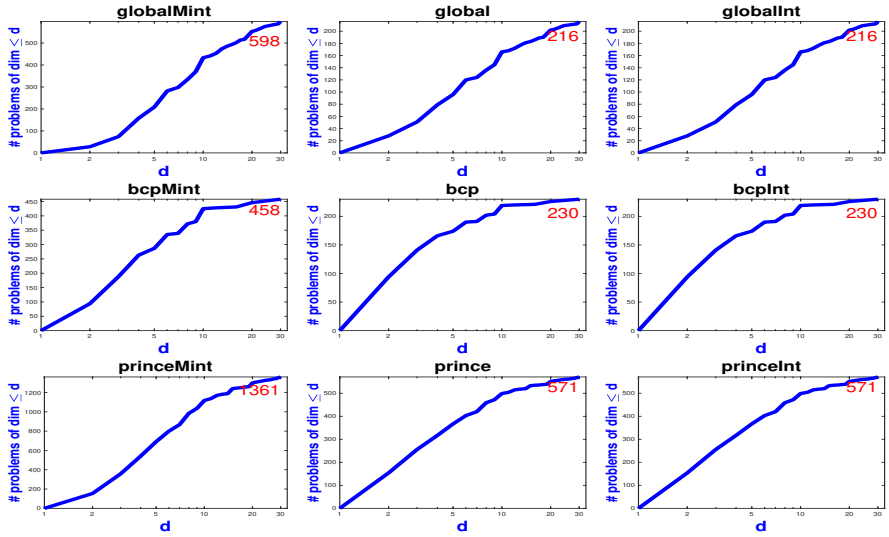


Fig. 6: The number of problems with $n \leq d$.

For integer problem collections, we selected $I := [1:n]$ and $K := \emptyset$. But for mixed-integer problem collections, we used the choices

$$I := \begin{cases} [2] & n = 2, \\ [2], [2,3] & n = 3, \\ [4], [3,4], [2,3,4] & n = 4, \\ [1:h-1],\ [1:h],\ [1:h+1],\ [1:h+2] & n = 2h+1 \geq 5, \\ [1:h-1],\ [1:h],\ [1:h+1] & n = 2h \geq 6 \end{cases}$$

and $K := [1:n] \setminus I$. These choices result in more mixed-integer problems than continuous and integer problems. Table 5 lists the number of continuous, integer, and mixed-integer problems in each problem class.

problem class	`global`	`bcp`	`prince`	$\sum$
original	370	326	987	1683
subset with dim $n \in [2:30]$	216	230	571	1017
integer problems	216	230	571	1017
mixed-integer problems	598	458	1361	2417

Table 5: Table with # of problems. With the three noise levels $\omega = 10^{-3}, 10^{-2}, 10^{-1}$, this gives a total of $3 \times 1017 = 3051$ continuous test problems, a total of $3 \times 1017 = 3051$ integer test problems, and a total of $3 \times 2417 = 7251$ mixed-integer test problems. Thus, a total of 13353 test problems is used in our comparison.

absolute uniform i.e.

The type of noise is absolute uniform, i.e., $\tilde{f} = f + (2 * \text{rand} - 1)\omega$ with rand $\sim \mathcal{N}(0, 1)$. For all test problems, the initial points are chosen as (i) $x_i^0 := 50$ for $i \in I$ as in [31, Section 4]; (ii) x_i^0 for $i \in K$ as given in [40].

4.3 Tools for efficiency and robustness

We denote by `nfmax` the maximum number `nf` of function evaluations and by `secmax` the maximum time in seconds (`sec`). The budget available for each solver is limited by allowing at most `secmax` := 360 seconds of run time and at most `nfmax` := $1200n$ function evaluations for a problem with n variables. We chosen `secmax` and `nfmax` so that the best solver can solve at least 75% of the selected problems for the three noise levels $\omega = 10^{-3}, 10^{-2}, 10^{-1}$. As can be seen from the first row of Table 6 below, for $\omega = 0.1$ all solvers solve 97% problems and the first ranked robust solver `MATRS` solves 92% problems. In this result, `MATRS` terminates in 0.08% of problems because `nfmax` is reached, while it does not terminate because `secmax` is reached. However, for integer problems, since it is difficult to find new integer feasible points, all solvers terminate due to reaching `secmax` at least once and increasing `secmax` does not change efficiency and robustness.

Let f_{init} denote the function value of the starting point (common to all solvers), f_{opt} denote the best point known to us, and f_s denote the best point found by the solver s. We say that the solver s solves a problem with dimension n if the target accuracy

$$q_f := (f_s - f_{\text{opt}})/(f_{\text{init}} - f_{\text{opt}}) \leq \epsilon = 10^{-4}$$

is satisfied. Otherwise, it cannot solve such a problem since either `nfmax` or `secmax` was reached. q_f identifies the convergence speed of the solver s to reach a minimum of the smooth true function f.

Denote by $\mathcal{S}$ the list of compared solvers and by $\mathcal{P}$ the list of problems. We say that the solver s is most *efficient* on a collection if it has the lowest relative cost of function evaluations. A good tool to evaluate the efficiency of the compared solvers is the performance profile of Dolan & Moré [11]. The performance profile of the solver s

$$\rho_s(\tau) := \frac{1}{|\mathcal{P}|}\left|\left\{p \in \mathcal{P} \;\middle|\; pfr_{s,p} \leq \tau\right\}\right| \tag{4}$$

counts the fraction of problems solved by the solver s such that the upper bound of the *performance ratio*

$$pfr_{s,p} := \frac{c_{s,p}}{\min(c_{\overline{s},p} \mid \overline{s} \in S)}$$

is τ. Here $c_{s,p}$ is the *cost measure* of the solver s to solve the problem p. The number `nf` of function evaluations and time in seconds `sec` are used as the two cost measures.

We say that the solver s is most *robust* on a collection if it has the highest number of solved problems. A good tool to evaluate the robustness of the compared solvers is the data profiles of MORÉ & WILD [33]. The data profile of the solver s

$$\delta_s(\kappa) := \frac{1}{|\mathcal{P}|}\Big|\Big\{p \in \mathcal{P} \;\Big|\; cr_{s,p} \le \kappa\Big\}\Big| \tag{5}$$

is the fraction of problems solved by the solver s with κ groups of $n_p + 1$ function evaluations such that κ is the upper bound of the *cost ratio* $cr_{s,p} := c_{s,p}/(n_p + 1)$. Here n_p denotes the dimension of the problem $p \in \mathcal{P}$.

For a given collection S of solvers, the strength of a solver $s \in S$ – relative to an ideal solver corresponding to the best solver for the problem $p \in \mathcal{P}$ – is measured for each given cost measure $c_{s,p}$ by the number $e_{s,p}$ given by

$$e_{s,p} := \begin{cases} 100\big(\min\limits_{\bar{s}\in S} c_{\bar{s},p}\big)/c_{s,p}, & \text{if the solver } s \text{ solves the problem } p, \\ 0, & \text{otherwise,} \end{cases}$$

called the *efficiency* of the solver s to solve the problem p with respect to $c_{s,p}$.

Comparisons are summarized in Tables 6–8 that indicate the efficiency and robustness of each solver. In these tables, efficiencies $e_{s,p}$ are given in percent. Larger $e_{s,p}$ with respect to the two cost measures $c_{s,p} \in \{\mathtt{nf}_{s,p}, \mathtt{sec}_{s,p}\}$ in these tables imply a better average behavior, while a zero efficiency indicates failure. All values are rounded (against zero) to integers. `nf` and `sec` are the two cost measures in these tables. In these tables not recording efficiencies, a sign
• n indicates that $\mathtt{nf} \ge \mathtt{nfmax} = 1200n$ *was reached.*
• t indicates that $\mathtt{sec} \ge \mathtt{secmax} = 360$ seconds *was reached.*
• f indicates that the solver s *failed* for other reasons, such as bugs or algorithmic terminations. In particular, `MISO` terminated because `secmax` was reached much more often than for the other solvers, even for problems with dimension 2. So changing `secmax` does not help `MISO`.

4.4 A comparison of DFO solvers

In this section, we illustrate the performance and data profiles of the DFO solvers provided in Table 3 to compare their robustness and efficiency.

In Subsection 4.4.1, we compare `MATRS`, `CMAES`, `NOMAD`, `DFOTR`, `BCDFO`, and `BFO` on the three continuous problem collections `global`, `bcp`, and `prince`.

In Subsection 4.4.2, we compare MATRS, CMAES, NOMAD, DFLINT, and BFO on the three integer problem collections globalInt, bcpInt, and princeInt.

For the mixed-integer collections, we show the dependence on different realization of the noise. Since the problems are contaminated by random noise, results are slightly different in different runs. Hence, we first run all proposed solvers once. Then, we run the four best solvers in terms of number of solved problems on the same problem 11 times and report results for the virtual solvers (<s>min, <s>med, <s>max), where s is a solver name. These virtual solvers are obtained by sorting results by decreasing number of solved problems and declaring the first, seventh, and eleventh results are the results of <s>min, <s>med, <s>max, respectively. These reported results show that the order of compared solvers in terms of efficiency and robustness remains unchanged. Hence, we execute each solver only once on the integer and continuous collections of test problems.

In Subsection 4.4.3, we compare the 12 virtual solvers MATRSmax, MATRSmed, MATRSmin, CMAESmax, CMAESmed, CMAESmin, NOMADmax, NOMADmed, NOMADmin, BFOmax, BFOmed, BFOmin, and the two solvers MISO and DFLINT on the three mixed-integer problem collections globalMint, bcpMint, and princeMint.

Before DFO methods can find an approximate stationary point, they may get stuck in the presence of strong noise. Since it difficult to find a reduction of $\tilde{f}$ in such cases, step sizes are decreased in line search techniques and trust-region radii are decreased in trust-region techniques, potentially resulting in zero steps. To overcome such problems and increase the efficiency and robustness, MATRS uses an alternative algorithmic approach by turning itself into multi-line searches with a specified set of directions, improved MAES algorithms, or improved trust-region algorithms for integer, continuous, and mixed-integer DFO problems. Additionally, MATRS uses a mixed-integer phase for mixed-integer DFO problems at the end of each iteration. There is no such alternative algorithmic approach for the other DFO solvers. Using such an alternative algorithmic approach, MATRS (i) leaves regions near a maximizer or a saddle point by performing continuous and integer line searches using extrapolation, (ii) avoids zero steps by performing integer and continuous improved MAES, (iii) increases the accuracy of the model functions and generates good trust-region directions by performing continuous and integer trust-region algorithms, (iv) finds a significant reduction of $\tilde{f}$ by performing mixed-integer line searches along directions pointing out the valley.

As can be seen from Figs. 7–9 and Tables 6–8 below, in contrast to other solvers, MATRS does excellent integer searches in addition to its continuous searches. More precisely, MATRS is more robust and slightly more efficient than the other solvers on the continuous and integer collections and more robust than the other solvers on the mixed-integer collections, while the most efficient solver, NOMAD, is slightly more efficient than MATRS (the second-ranked efficient solver) on the mixed-integer collections.

The efficiency and robustness of MATRS are marginally decreased by increased noise. With the exception of significant noise $\omega = 10^{-1}$ and for the mixed-integer collection princeMint, this is correct for NOMAD.

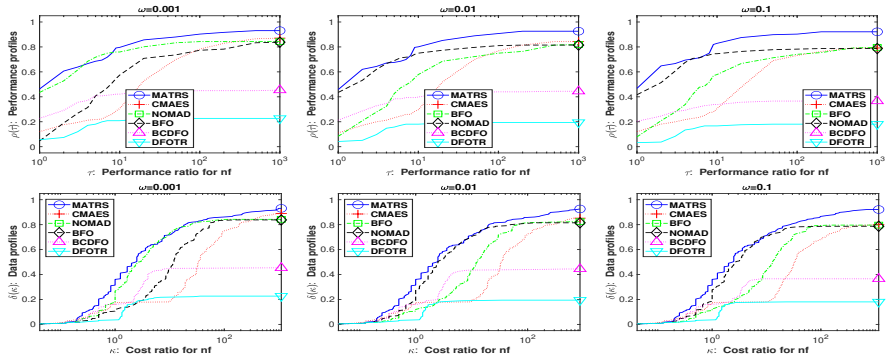
In our comparison, there are two randomized solvers: CMAES and MATRS. One other noteworthy finding from these results is that, when compared to the three virtual versions of CMAES and even to the determinist solver NOMAD, the efficiency and robustness of the three virtual versions of MATRS remain largely unchanged.

Continuous searches of CMAES performed substantially better than its integer searches. This demonstrates that integer searches cannot be conducted as effectively with their continuous searches. Although MATRS uses the same line search for both integer and continuous searches, it not only forms and solves the integer and continuous trust-region subproblems differently, but also updates the trust-region radii differently.

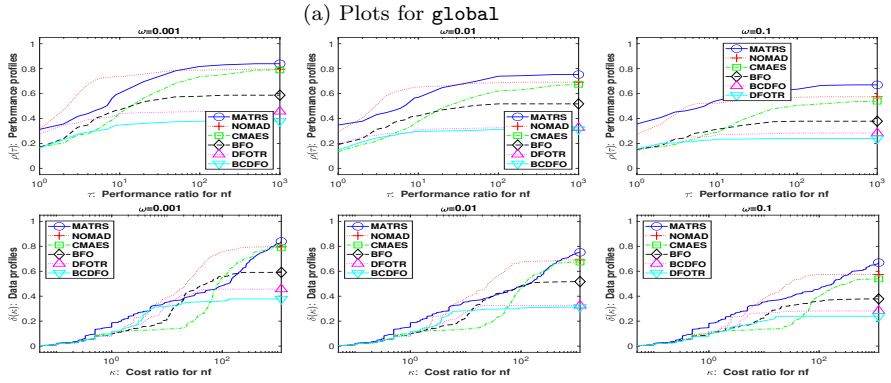
DFLINT uses Halton sequences to generate a set of good directions, then runs integer line searches along these directions. This explains why DFLINT outperforms the integer DFO solvers in terms of robustness and efficiency. As long as the best point can be updated, MATRS turns into integer or continuous multi-line searches, which perform along directions generated by usequence. This is the reason why MATRS can solve a large number of problems with a low number of function evaluations.

4.4.1 Results for global*,* bcp*,* prince

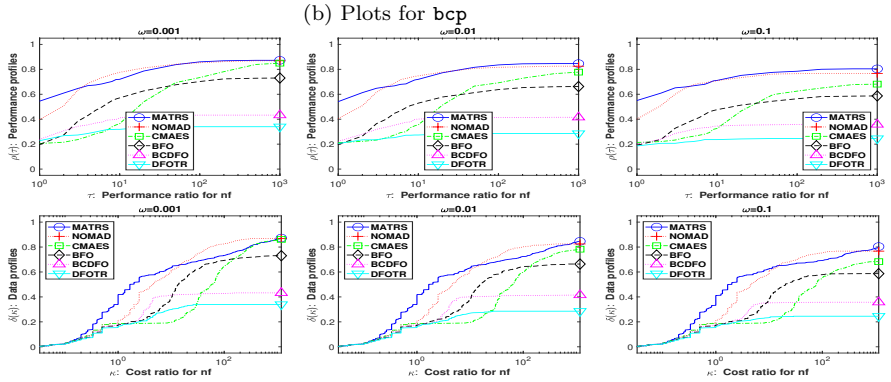
Fig. 7 and Table 6 show that for all noise levels:
• On all continuous problem collections, MATRS is the most robust solver.
• On bcp, NOMAD is the most efficient solver, while for global and prince, MATRS is the most efficient solver.



(a) Plots for `global`



(b) Plots for `bcp`



(c) Plots for `prince`

Fig. 7: Plots for (a) `global`, (b) `bcp`, (c) `prince` for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right). Performance profiles $\rho(\tau)$ (first row) are in dependence of a bound τ on the performance ratio (see (4)), while data profiles $\delta(\kappa)$ (second row) are in dependence of a bound κ on the cost ratio (see (5)). Problems solved by no solver are ignored.

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
215 of 216 global problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	201	15	0	0	61	65
CMAES	192	24	0	0	20	32
NOMAD	182	0	0	34	57	50
BFO	181	0	0	35	24	53
BCDFO	98	4	2	112	30	19
DFOTR	49	157	4	6	10	8

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
212 of 216 global problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	200	16	0	0	61	66
CMAES	184	32	0	0	19	31
BFO	178	0	0	38	26	52
NOMAD	176	0	0	40	56	46
BCDFO	96	10	5	105	29	19
DFOTR	42	166	3	5	8	8

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
209 of 216 global problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	199	17	0	0	63	68
CMAES	173	43	0	0	20	30
BFO	172	0	0	44	26	51
NOMAD	170	0	0	46	55	47
BCDFO	79	11	4	122	26	14
DFOTR	39	170	0	7	7	5

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
211 of 230 bcp problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	193	37	0	0	40	46
NOMAD	183	0	1	46	51	36
CMAES	182	48	0	0	25	42
BFO	136	5	0	89	28	48
DFOTR	105	90	1	34	34	19
BCDFO	87	65	1	77	24	11

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
193 of 230 bcp problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	173	57	0	0	42	45
NOMAD	159	0	2	69	45	36
CMAES	155	75	0	0	22	33
BFO	119	0	0	111	28	40
DFOTR	75	102	0	53	24	16
BCDFO	72	81	2	75	21	10

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
176 of 230 bcp problems solved

dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	154	76	0	0	42	46
NOMAD	132	0	0	98	38	34
CMAES	124	106	0	0	20	26
BFO	87	0	0	143	21	27
BCDFO	65	89	2	74	20	9
DFOTR	55	110	0	65	19	11

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
542 of 571 prince problems solved

dim∈[1,30]		# of anomalies			eff%	
noise level: $\omega = 10^{-3}$						
solver	solved	#n	#t	#f	nf	sec
MATRS	498	72	0	1	62	63
NOMAD	496	0	3	72	55	44
CMAES	492	75	0	4	26	31
BFO	417	4	0	150	33	48
BCDFO	247	117	5	202	31	11
DFOTR	194	311	12	54	27	14

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
529 of 571 prince problems solved

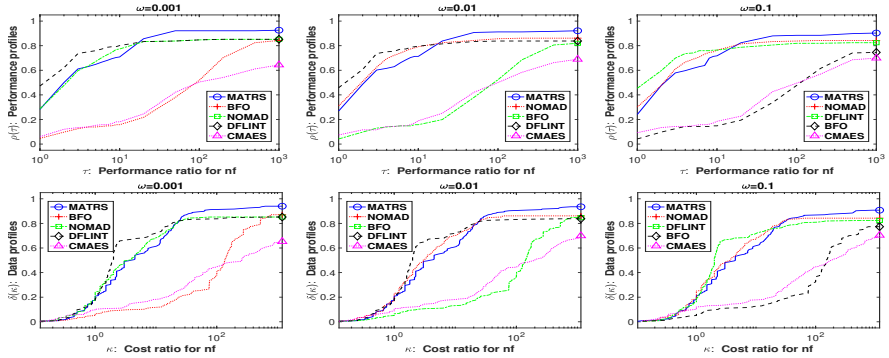
dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	483	87	0	1	62	61
NOMAD	471	0	2	98	54	43
CMAES	447	120	0	4	26	29
BFO	379	2	0	190	31	42
BCDFO	237	138	4	192	29	11
DFOTR	163	324	11	73	23	13

stopping test:
$q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
509 of 571 prince problems solved

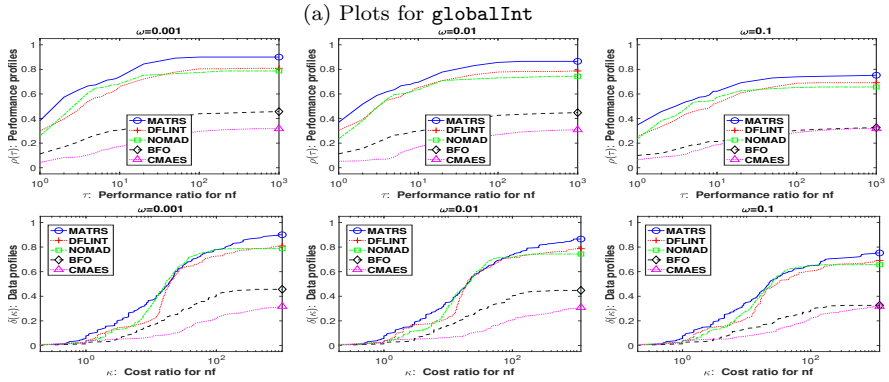
dim∈[1,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	459	112	0	0	62	61
NOMAD	438	0	0	133	53	42
CMAES	390	178	0	3	25	25
BFO	335	0	0	236	29	35
BCDFO	204	160	4	203	25	9
DFOTR	140	346	4	81	20	10

Table 6: Tabulated results for (first row) `global`, (second row) `bcp`, (third row) `prince` for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right).

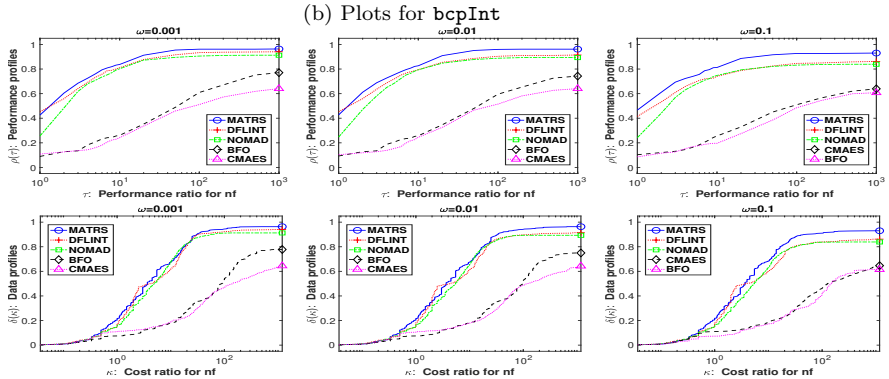
4.4.2 Results for `globalInt`, `bcpInt`, `princeInt`



(a) Plots for `globalInt`



(b) Plots for `bcpInt`



(c) Plots for `princeInt`

Fig. 8: Plots for (a) `globalInt`, (b) `bcpInt`, (c) `princeInt` for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right). Other details are as in Fig. 7.

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
215 of 216 globalInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	203	11	2	0	50	46
BFO	188	0	0	28	10	23
NOMAD	184	0	4	28	49	47
DFLINT	184	0	21	11	63	63
CMAES	141	75	0	0	12	22

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
214 of 216 globalInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	202	12	2	0	49	47
NOMAD	186	0	4	26	52	47
BFO	183	0	0	33	10	22
DFLINT	181	1	22	12	62	62
CMAES	151	65	0	0	12	22

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
214 of 216 globalInt problems solved

dim∈[2,30]		# of anomalies			e_* in %	
solver	solved	#n	#t	#f	nf	sec
MATRS	196	19	1	0	47	53
NOMAD	182	0	7	27	51	43
DFLINT	178	2	21	15	61	56
BFO	167	0	0	49	10	22
CMAES	152	64	0	0	13	25

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
226 of 230 bcpInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	207	20	3	0	57	51
DFLINT	186	30	3	11	44	55
NOMAD	181	0	1	48	47	38
BFO	105	1	0	124	19	27
CMAES	73	157	0	0	8	13

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
216 of 230 bcpInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	199	26	5	0	53	51
DFLINT	181	25	4	20	44	53
NOMAD	171	0	1	58	42	34
BFO	103	1	0	126	18	26
CMAES	71	159	0	0	8	12

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
203 of 230 bcpInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	173	50	7	0	47	45
DFLINT	159	39	3	29	37	45
NOMAD	151	0	5	74	38	33
BFO	75	1	0	154	14	19
CMAES	73	157	0	0	10	15

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-3}$
557 of 571 princeInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	550	13	7	1	61	63
DFLINT	537	11	6	17	59	61
NOMAD	521	0	1	49	51	41
BFO	444	0	0	127	15	31
CMAES	368	202	0	1	15	26

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-2}$
557 of 571 princeInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	550	12	8	1	61	64
DFLINT	522	20	6	23	59	55
NOMAD	510	0	3	58	49	38
BFO	428	0	0	143	15	31
CMAES	367	204	0	0	14	26

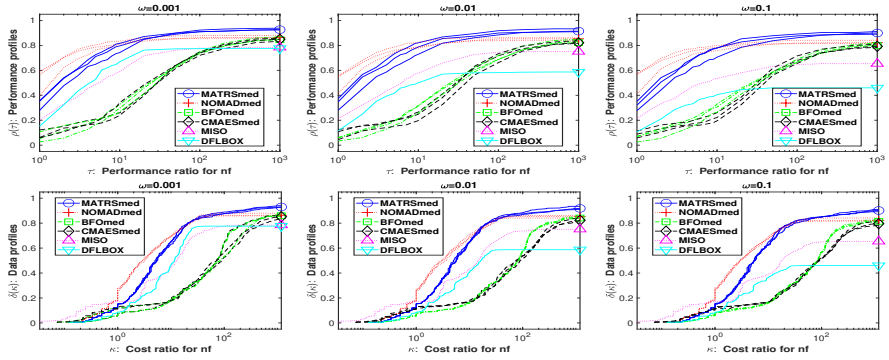
stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$
noise level: $\omega = 10^{-1}$
551 of 571 princeInt problems solved

dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRS	531	34	5	1	62	66
DFLINT	492	36	5	38	56	50
NOMAD	479	0	2	90	47	33
CMAES	369	202	0	0	15	27
BFO	351	0	0	220	13	26

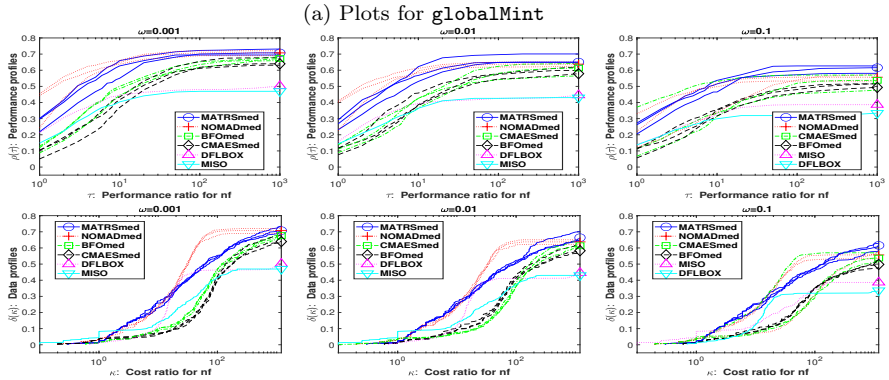
Table 7: Tabulated results for (first row) `globalInt`, (second row) `bcpInt`, (third row) `princeInt` for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right).

Fig. 8 and Table 7 show that for all noise levels:
- On all three integer collections, `MATRS` is the most robust solver.
- On `bcpInt` and `princeInt`, `MATRS` is the most efficient solver.
- On `globalInt`, `DFLINT` is the most efficient solver.

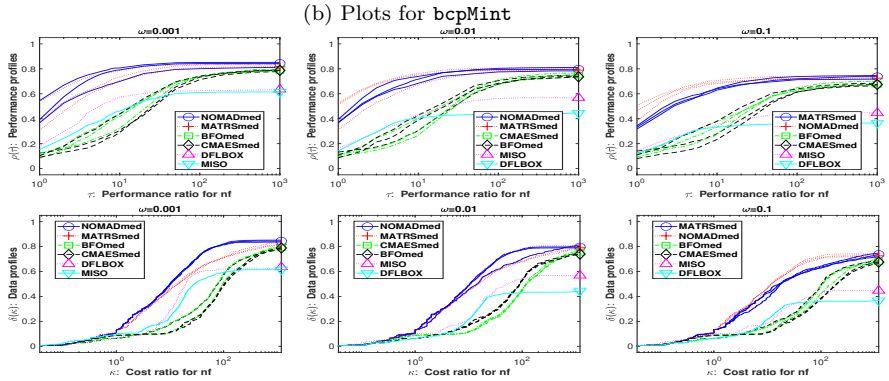
4.4.3 Results for `globalMint`, `bcpMint`, `princeMint`



(a) Plots for `globalMint`



(b) Plots for `bcpMint`



(c) Plots for `princeMint`

Fig. 9: Plots for (a) `globalMint`, (b) `bcpMint`, (c) `princeMint` for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right). Other details are as in Fig. 7.

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-3}$						
594 of 598 globalMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	562	32	4	0	44	39
MATRSmed	556	35	7	0	42	38
MATRSmin	553	41	4	0	44	39
NOMADmax	526	0	0	72	54	48
CMAESmax	520	78	0	0	10	21
NOMADmed	516	0	0	82	51	46
BFOmax	515	55	0	28	10	23
BFOmed	514	0	0	84	9	21
NOMADmin	513	0	0	85	53	43
BFOmin	510	0	0	88	9	22
CMAESmed	509	89	0	0	12	22
CMAESmin	500	98	0	0	10	21
MISO	469	0	129	0	32	15
DFLBOX	464	4	0	130	30	58

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-2}$						
589 of 598 globalMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	559	37	2	0	44	41
MATRSmed	549	46	3	0	43	42
MATRSmin	544	49	5	0	43	40
NOMADmax	515	0	0	83	50	45
NOMADmed	510	0	0	88	53	46
BFOmax	509	0	0	89	9	21
NOMADmin	503	0	0	95	48	45
CMAESmax	503	95	0	0	10	21
BFOmed	503	0	0	95	9	21
BFOmin	500	0	0	98	9	22
CMAESmed	492	106	0	0	10	20
CMAESmin	489	109	0	0	11	21
MISO	449	0	149	0	31	16
DFLBOX	351	7	0	240	25	46

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-1}$						
587 of 598 globalMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	545	48	5	0	40	41
MATRSmed	539	53	6	0	41	40
MATRSmin	533	58	7	0	42	42
NOMADmax	503	0	0	95	52	46
BFOmax	493	0	0	105	9	22
NOMADmed	492	0	0	106	51	48
NOMADmin	488	0	0	110	50	45
BFOmed	487	0	0	111	9	22
CMAESmax	486	112	0	0	10	21
BFOmin	483	0	0	115	9	21
CMAESmed	476	122	0	0	10	19
CMAESmin	470	128	0	0	9	20
MISO	391	0	207	0	31	15
DFLBOX	275	6	0	317	20	38

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-3}$						
446 of 458 bcpMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	336	122	0	0	33	26
NOMADmax	330	0	3	125	39	27
MATRSmed	324	133	1	0	31	25
NOMADmed	324	0	5	129	38	28
MATRSmin	318	140	0	0	32	27
NOMADmin	316	0	4	138	37	26
BFOmax	315	0	0	143	17	27
BFOmed	310	0	0	148	18	27
CMAESmax	310	148	0	0	14	26
BFOmin	303	0	0	155	16	25
CMAESmed	293	165	0	0	12	22
CMAESmin	289	169	0	0	13	26
DFLBOX	230	54	0	174	29	40
MISO	215	0	243	0	23	9

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-2}$						
427 of 458 bcpMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	321	136	1	0	32	25
MATRSmed	304	151	3	0	31	24
NOMADmax	298	0	2	158	36	25
MATRSmin	297	158	3	0	30	24
CMAESmax	294	164	0	0	12	26
NOMADmed	292	0	3	163	35	26
BFOmax	285	115	0	58	15	24
NOMADmin	285	0	3	170	34	24
CMAESmed	281	177	0	0	13	26
CMAESmin	274	184	0	0	11	24
BFOmed	267	0	0	191	15	24
BFOmin	261	0	0	197	15	23
DFLBOX	204	63	0	191	25	36
MISO	197	0	261	0	21	9

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-1}$						
405 of 458 bcpMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	289	164	5	0	29	24
MATRSmed	282	172	4	0	28	23
MATRSmin	266	187	5	0	27	23
CMAESmax	263	195	0	0	13	23
NOMADmax	261	0	0	197	31	24
NOMADmed	255	0	0	203	30	24
CMAESmed	246	212	0	0	12	22
NOMADmin	244	0	0	214	29	22
BFOmax	239	0	0	219	13	22
CMAESmin	236	222	0	0	11	21
BFOmed	228	0	0	230	13	20
BFOmin	220	0	0	238	12	18
MISO	177	0	281	0	19	8
DFLBOX	154	70	0	234	19	27

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-3}$						
1347 of 1361 princeMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
NOMADmax	1159	0	7	195	52	38
NOMADmed	1148	0	9	204	51	39
NOMADmin	1138	0	8	215	51	38
MATRSmax	1138	214	9	0	43	35
MATRSmed	1113	240	7	1	42	34
MATRSmin	1104	251	6	0	43	34
CMAESmax	1087	273	0	1	13	23
BFOmax	1081	0	0	280	16	26
BFOmed	1076	0	0	285	17	28
CMAESmed	1073	287	0	1	14	23
BFOmin	1068	0	0	293	16	27
CMAESmin	1061	298	0	2	13	22
DFLBOX	864	83	0	414	28	48
MISO	836	0	525	0	24	9

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-2}$						
1318 of 1361 princeMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	1104	247	9	1	43	35
NOMADmax	1097	0	3	261	49	37
NOMADmed	1084	0	2	275	49	39
MATRSmed	1075	275	10	1	42	34
NOMADmin	1074	0	2	285	49	38
MATRSmin	1066	285	8	2	42	34
CMAESmax	1045	315	0	1	13	22
BFOmax	1025	196	0	140	16	24
CMAESmed	1025	334	0	2	12	22
BFOmed	1007	0	0	354	15	25
CMAESmin	1004	355	0	2	14	22
BFOmin	1002	0	0	359	15	27
MISO	773	0	588	0	23	9
DFLBOX	606	89	0	666	22	35

stopping test: $q_f \leq 0.0001$, sec ≤ 360, nf $\leq 1200 * n$						
noise level: $\omega = 10^{-1}$						
1276 of 1361 princeMint problems solved						
dim∈[2,30]		# of anomalies			eff%	
solver	solved	#n	#t	#f	nf	sec
MATRSmax	1016	328	15	2	37	31
NOMADmax	1008	0	0	353	46	38
MATRSmed	1003	341	15	2	40	32
NOMADmed	987	0	0	374	46	38
MATRSmin	977	368	16	0	39	32
NOMADmin	976	0	0	385	44	36
CMAESmax	954	405	0	2	12	21
BFOmax	933	223	0	205	14	22
BFOmed	925	0	0	436	14	24
CMAESmed	919	441	0	1	11	18
BFOmin	912	0	0	449	14	23
CMAESmin	903	456	0	2	12	19
MISO	608	0	753	0	20	8
DFLBOX	499	116	0	746	18	30

Table 8: Tabulated results for (first row) globalMint, (second row) bcpMint, (third row) princeMint for dimensions $2 \leq n \leq 30$ and noise levels $\omega = 0.001$ (left), 0.01 (middle), 0.1 (right).

Fig. 9 and Table 8 show that:
• For all noise levels and on `globalMint` and `bcpMint`, `MATRSmed` is the most robust solver and `NOMADmed` is second-ranked.
• On `princeMint`, `MATRSmed` is for the largest noise level $\omega = 10^{-1}$ the most robust solver and `NOMADmed` is second-ranked, while `NOMADmed` is for the two noise levels $\omega = 10^{-3}, 10^{-2}$ the most robust solver and `MATRSmed` is second-ranked.
• For all noise levels and on all mixed-integer problems, `NOMADmed` is the most efficient solver and `MATRSmed` is second-ranked.

4.5 Discussion

Our numerical findings demonstrate that overall (and for large noise always), in terms of robustness, `MATRS` is the best solver, regardless of the kind of problem, the kind of variable, and the level of noise. Moreover, `MATRS` is a very efficient solver, second-ranked in efficiency only in the mixed-integer case, where `NOMAD` is more efficient.

Compared to the other solvers, `MATRS` retains its robustness and efficiency even with increased noise. To be efficient and robust for noisy problems, `MATRS` uses the three distinct algorithms (integer/continuous line searches, integer/continuous trust regions, and integer/continuous `MAES`). To enhance its efficiency, `MATRS` turns into line searches or trust regions. Trust regions improve the efficiency of `MATRS` by building quadratic model and avoiding large steps, whereas line searches use extrapolation to leave regions close to a saddle point or maximizer. To enhance its robustness, `MATRS` turns into `MAES`. Using `MAES`, `MATRS` avoids failure due to null steps and increases its robustness when line search step sizes and trust-region radii become tiny and the new best point cannot be found.

For the mixed-integer problems, `MATRS` (second-ranked efficient solver) usually requires more function evaluations than `NOMAD` (the most efficient solver for mixed-integer problems). As long as `cMutation` (`iMutation`) can update the best point, `MATRS` turns into a multi-line search, which may leads to a low number of function evaluations because of using extrapolation. Hence, `MATRS` can solve some problems with a low number of function evaluations. But, in some other particular problems, none of `cMutation` (`iMutation`) and `cRecom` (`iRecom`) may not update the best point in some iterations. In such cases, using the mutation points evaluated in `cMutation` (`iMutation`) for the construction of the trust-region subproblem, `cTRS` (`iTRS`) is tried to update the best point. Thus, although the number of function evaluations may increase as a result of performing `cMutation` (`iMutation`), `cRecom` (`iRecom`), and `cTRS` (`iTRS`), the number of solved problems may increase and that is the reason why `MATRS` is the most robust solver on the continuous, integer, and mixed-integer problems.

5 Conclusion

This paper describes a new matrix adaptation trust-region strategy for bound-constrained DFO problems with mixed-integer variables. This strategy finds the best points by integer and continuous line search methods in the mutation and recombination phases, by integer and continuous trust-region methods in the trust-region phase, and by mixed-integer line search method in the mixed-integer phase.

A new randomized space-filling method was proposed as a good replacement for the Halton sequences to generate well-distributed mutation points in the integer and continuous mutation phases.

Compared to other solvers, when the noise is increased the efficiency and robustness of `MATRS` are only marginally decreased in most problems. This is due to the combination of line searches, trust regions, and `MAES`. However, to be more robust than the other algorithms, `MATRS` may need to use all three of these methods in order to solve some particular problems. In these cases, `MATRS` requires a larger number of function evaluations.

6 A Appendix: Flowcharts of all subroutines

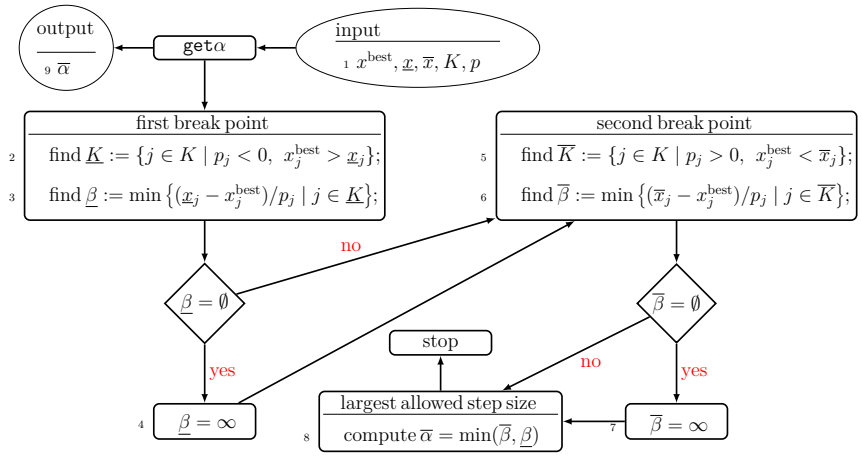
This section discusses flowcharts of several important subroutines of `MATRS` whose Matlab codes are available in the `MATRS` package [28].

6.1 A continuous mutation phase

6.1.1 `get`α

Algorithm 3 is pseudocode for `get`α. `get`α computes the two break points $\underline{\beta}$ and $\overline{\beta}$ in lines 3 and 6 and takes their minimum $\overline{\alpha}$ in line 8. Here p is a given trial direction. If `cMutation` calls `get`α, p is a mutation direction. Otherwise, if `cRecom` calls `get`α, p is a recombination mutation direction. In addition, if `miMATRS` calls `get`α, p is a combination direction. If the first break point does not exist, it is chosen to be infinity and the same applies to the second break point. After computing $\overline{\alpha}^i$, if $\overline{\alpha}^i = 0$ for the continuous variables and $\overline{\alpha}^i < 1$ for the integer variables, then there is no feasible trial point along $\pm p_{\mathrm{md}}$; hence at most n_{dd} times the corresponding distribution direction p_{dd} and then p_{md} are recomputed for a possible finding of some feasible trial points along new $\pm p_{\mathrm{md}}$. Here $n_{\mathrm{dd}} = n'_{\mathrm{dd}}$ for the continuous search and $n_{\mathrm{dd}} = n''_{\mathrm{dd}}$ for the

Algorithm 3 Pseudocode for `getα`



integer search, where n'_{dd} and n''_{dd} are the tuning parameters. To simplify `getα`, this improvement does not appear in Algorithm 3. In addition, in the Matlab code, `getα` also computes the initial step sizes for pseudocode of `cMutation`, `iMutation`, `cRecom`, `iRecom`, and `miMATRS`. Since there are different formulas for calculating the initial step sizes in the present paper, we calculate these step sizes after calculating $\overline{\alpha}$ by `getα` in all the mentioned pseudocode so that these pseudocode are easy to read.

6.1.2 `updatePoint`

Algorithm 4 is pseudocode for `updatePoint`. `updatePoint` creates and updates the two different lists of evaluated points and their function values.

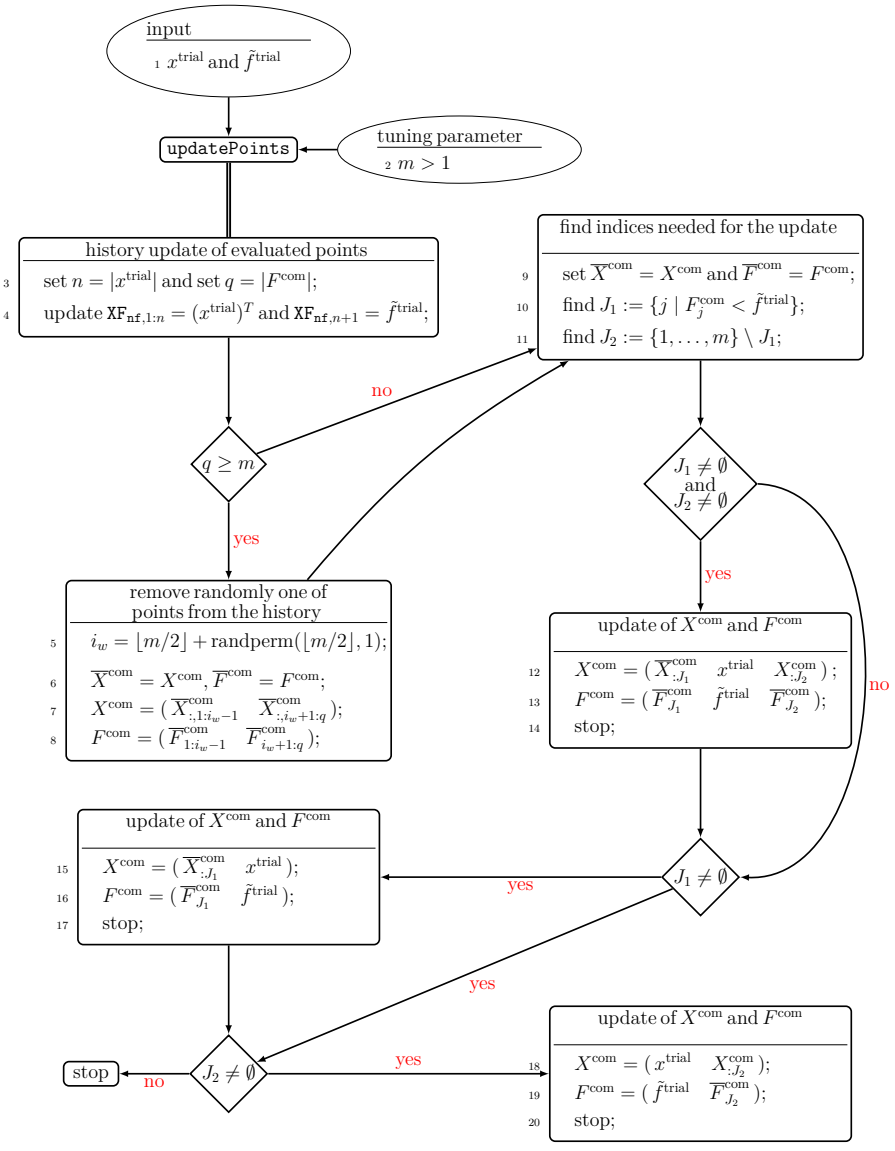
To approximate the gradients of the models in `cTRS` and `iTRS`, `updatePoint` saves any new trial point and its function value in the matrix `XF` (the first list) in line 4. To simplify pseudocode of `updatePoint` in the integer variable each trial point must be checked whether or not it has been evaluated already; this is done in the Matlab code of `updatePoint`.

To compute the combination directions in `miMATRS`, `updatePoint` saves and updates in the second list $(X)^{\mathrm{com}}, F^{\mathrm{com}}$ at most m best evaluated points in the matrix X^{com} and their function values in the vector F^{com} in ascending order. More precisely, `updatePoint` randomly selects an evaluated point whose place is between $m/2$ and m in line 5 by using the Matlab function randperm, removes this selected point and its inexact function value from X^{com} and F^{com} (the second list) in lines 7-8, and adds the new evaluated point and its inexact

function value to X^{com} and F^{com} in lines 12-13, 15-16, 18-19, so that the ascending order of inexact function values at these points is preserved.

To simplify all algorithms, we do not mention `XF`, X^{com}, and F^{com} as input and output of each subroutine, which computes the function value. Hence, `XF`, X^{com}, F^{com}, and `nf` are persistent variables.

Algorithm 4 Pseudocode for `updatePoint`

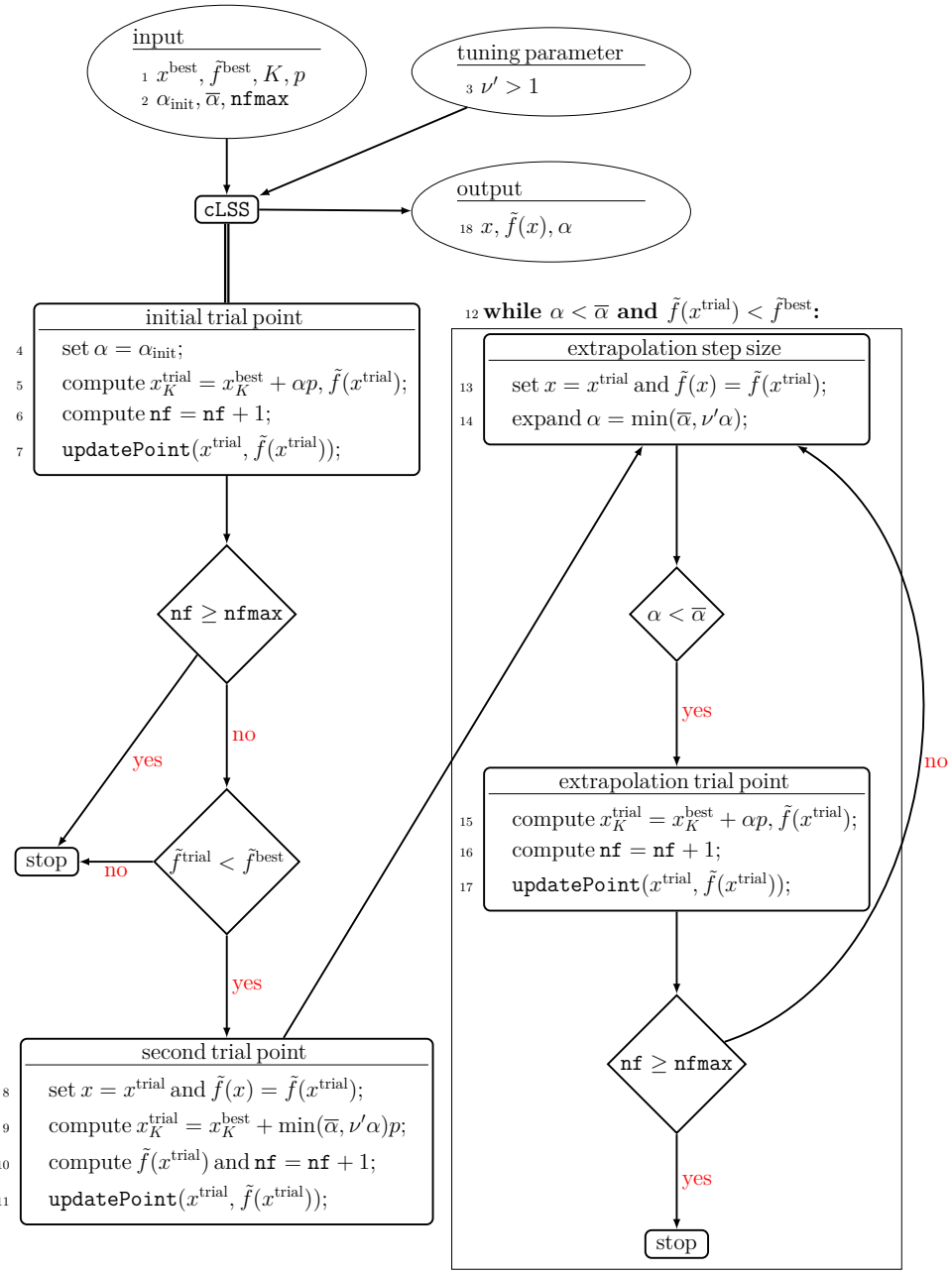


6.1.3 `cLSS`

Algorithm 5 is pseudocode for `cLSS`. In line 4 of `cLSS`, $\alpha = \alpha_{\text{init}}$ is chosen and the first continuous trial point x_K^{trial} and its inexact function value $\tilde{f}^{\text{trial}} := \tilde{f}(x^{\text{trial}})$ are calculated. The history of evaluated points is updated by calling `updatePoint` in line 7. If the descent condition $\tilde{f}^{\text{trial}} < \tilde{f}^{\text{best}}$ holds, the first trial point and its function value are saved in line 8 of `cLSS` and accepted as the first trial point of extrapolation. Then the new continuous trial point x_K^{trial} and its inexact function value $\tilde{f}^{\text{trial}}$ are calculated in lines 9-10 of `cLSS`. Otherwise, `cLSS` ends with the first trial point, which is accepted as either a mutation point in `cMutation` or a recombination point in `cRecom`. As long as the conditions $\alpha < \overline{\alpha}$ and $\tilde{f}^{\text{trial}} < \tilde{f}^{\text{best}}$ hold, an extrapolation step along the search direction p is continued by expanding the real step size in line 14 of `cLSS` and computing the new continuous trial point x_K^{trial} and its inexact function value $\tilde{f}^{\text{trial}}$ in line 15 of `cLSS`. Step sizes within `cLSS` are defined as in [31].

As in [26], after the extrapolation with at least two trial points is terminated, the trial point with the lowest inexact function values among all trial points evaluated by extrapolation is chosen as the new best point. To simplify the structure of `cLSS` this does not appear in pseudocode of `cLSS`, below. This is done in the Matlab code of `cLSS`. As described in lines 14 and 15 of `cMutation` below, the corresponding step size of the accepted trial point by extrapolation is stored in one of the components of $\mathbf{a}'$.

Algorithm 5 Pseudocode for `cLSS`



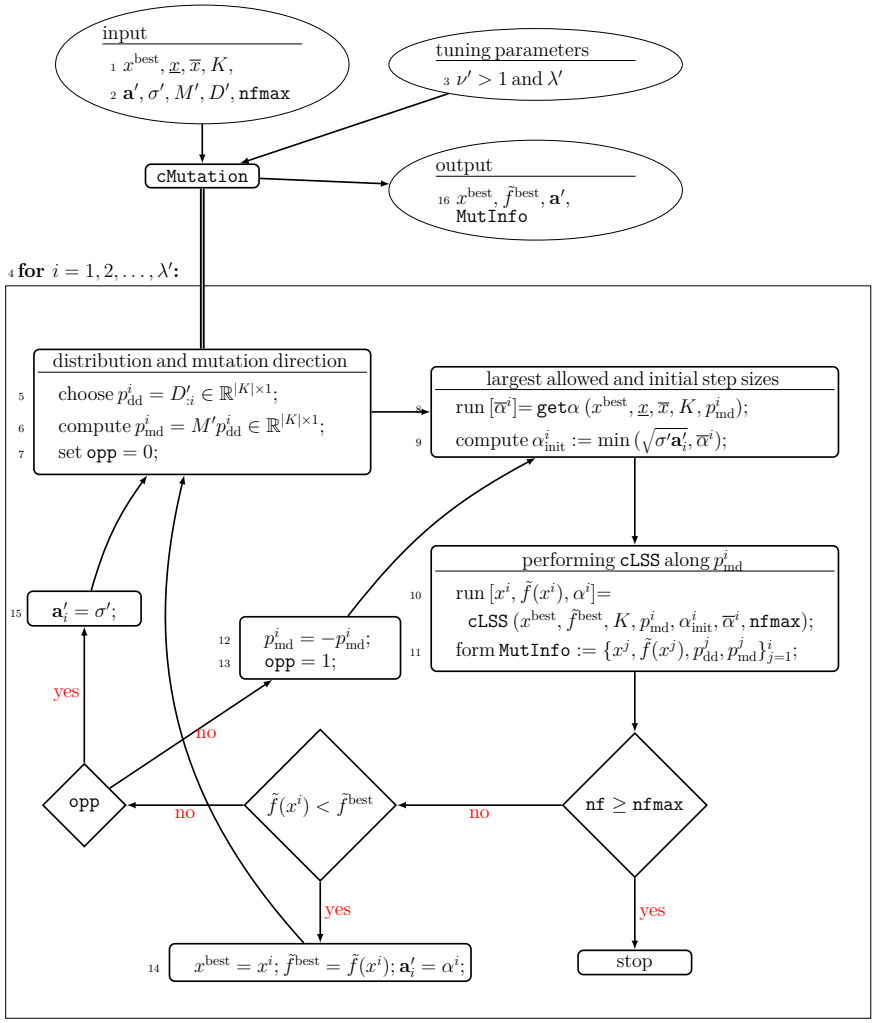
6.1.4 `cMutation`

Algorithm 6 is pseudocode for `cMutation`. The matrix $D'_{n\times\lambda'}$ denotes the set of distribution directions, each of which is chosen from the normal distribution $\mathcal{N}(0, I)$ with zero mean and variance I. In line 5 of `cMutation`, the ith distribution direction $p^i_{\rm dd} = D'_{:i} \in \mathbb{R}^{|K|\times 1}$ is chosen. Then, in line 6 of this algorithm, the ith mutation direction $p^i_{\rm md}$ is the product of the affine scaling matrix M' and $p^i_{\rm dd}$. Note that, in line 19 of `cRecom`, M' is updated by `updateM`. In line 7, the Boolean variable `opp` $= 0$ is evaluated, meaning that the opposite direction of $p^i_{\rm md}$ has not been tried yet. Before `cLSS` is executed, in line 8 of `cMutation`, the ith largest allowed real step size $\overline{\alpha}^i$ is computed by `get`α. In line 9 of `cMutation`, the ith initial real step size $\alpha^i_{\rm init}$ is chosen to be the minimum of $\overline{\alpha}^i$ and $\sqrt{\sigma'\mathbf{a}'_i}$, where σ' is the recombination step size, which is initially a positive tuning parameter and updated in lines 21-22 of `cRecom` below, and $\mathbf{a}'_i$ is the ith component of the mutation step size vector, which is updated in lines 14 and 15 of `cMutation`. The goal of this choice is to be neither too small nor too large to avoid line search failures. After computing $\alpha^i_{\rm init}$ and $\overline{\alpha}^i$, `cMutation` performs `cLSS` along the ith continuous distribution direction $p^i_{\rm md}$ to obtain the ith trial point $x^i_K = x^{\rm best}_K + \alpha^i p^i_{\rm md}$ in line 10, where α^i is found by `cLSS`. This trial point either is accepted (as either the continuous mutation point or the new best point) or rejected. If `nf` reaches `nfmax`, `cMutation` terminates. If `cLSS` cannot update $x^{\rm best}$ along $p^i_{\rm md}$, $p^i_{\rm md} = -p^i_{\rm md}$ is chosen in line 12 and `opp` $= 1$ is evaluated in line 13. After recomputing $\alpha^i_{\rm init}$ and $\overline{\alpha}^i$ in lines 8-9, respectively, `cMutation` performs `cLSS` along $p^i_{\rm md}$ and the list `MutInfo` is formed in line 11 of `cMutation`, which is used as input for `selection` in Section 6.2. Then, if `nf` reaches `nfmax`, `cMutation` terminates. Otherwise, if the descent condition $\tilde{f}(x^i) < \tilde{f}^{\rm best}$ holds, $x^{\rm best}$ and $\tilde{f}^{\rm best}$ are updated in line 14 of `cMutation`. In this case, x^i is one of the trial points evaluated by `cLSS` along $\pm p^i_{\rm md}$ with the lowest inexact function value $\tilde{f}(x^i)$ and the other trial points are rejected. If the first trial point cannot be the new best point, in this case, `cLSS` evaluates only one trial point and ends while accepting the first trial point as the ith mutation point.

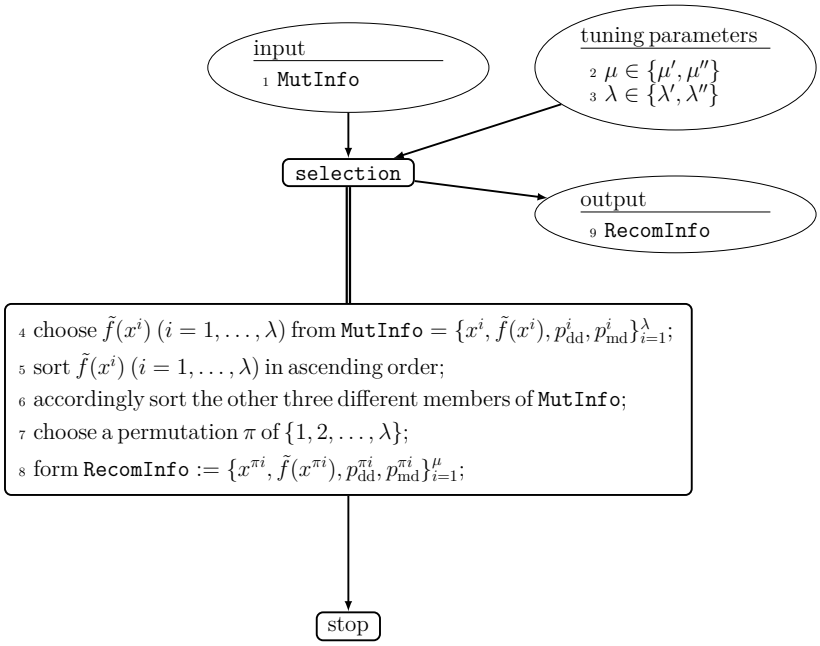
In lines 14 and 15 of `cMutation`, the ith component of the ith list $\mathbf{a}' \in \mathbb{R}^{\lambda'}$ of real mutation step sizes is updated, which is used to update the initial real step size $\alpha^i_{\rm init}$ for $i = 1, 2, \cdots, \lambda'$ in line 9 of `cMutation`. The vector $\mathbf{a}'$ is initially a tuning vector with real components ($\mathbf{a}'_i > 0$ for $i = 1, 2, \cdots, \lambda'$) and updated depending on whether or not decreases in the inexact function values are found at the trial points. After `cLSS` terminates to find the ith continuous mutation point, the corresponding step size of a point with the lowest inexact function value among all points evaluated in extrapolation is stored in $\mathbf{a}'_i$ in line 14 of `cMutation`. Otherwise, unlike [31] with $\mathbf{a}'_i = \mathbf{a}'_i/\nu'$ ($\nu' > 1$ is a given

tuning parameter), $\mathbf{a}'_i = \sigma'$ is stored in line 15 of `cMutation`. The reason for this new choice is that σ' does not become too small, avoiding getting stuck before an approximate stationary point is found. This is a new property of our algorithm, which is against line search failures.

Algorithm 6 Pseudocode for `cMutation`



Algorithm 7 Pseudocode for `selection`



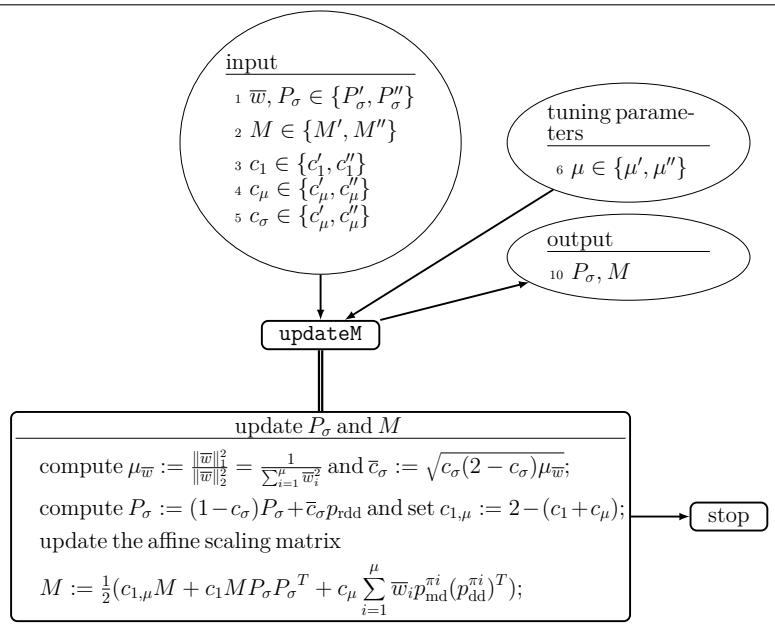
Algorithm 7 is pseudocode for `selection`. When `cMATRS` calls `selection`, $\lambda = \lambda'$ and $\mu = \mu'$ are chosen as the number of mutation points and the number of selected mutation points, respectively, but when `iMATRS` calls `selection`, $\lambda = \lambda''$ and $\mu = \mu''$ are chosen. `selection` takes the list `MutInfo` defined in line 4, where x^i $(i = 1, \ldots, \lambda)$ is the sequence of (integer and continuous) mutation points found. The sequence $\tilde{f}(x^i)$ $(i = 1, \ldots, \lambda)$ of inexact function values of the mutation points x^i $(i = 1, \ldots, \lambda)$ is sorted in ascending order

$$\tilde{f}(x^{\pi 1}) \leq \tilde{f}(x^{\pi 2}) \leq \ldots \leq \tilde{f}(x^{\pi \mu}) \leq \tilde{f}(x^{\pi(\mu+1)}) \leq \cdots \leq \tilde{f}(x^{\pi \lambda})$$

in line 5, where π is a permutation of $\{1, 2, \ldots, \lambda\}$. Then, accordingly the distribution directions $p^{\pi i}_{\mathrm{dd}}$ $(i = 1, \ldots, \lambda)$ and the mutation directions $p^{\pi i}_{\mathrm{md}}$ $(i = 1, \ldots, \lambda)$ are obtained in line 6 of `selection`. Finally, `selection` chooses π in line 7 and saves the best information in the list `RecomInfo` in line 8, which is used to compute new recombination points in the recombination phase, where μ is the number of selected points.

6.3.1 `updateM`

Algorithm 8 Pseudocode for `updateM`



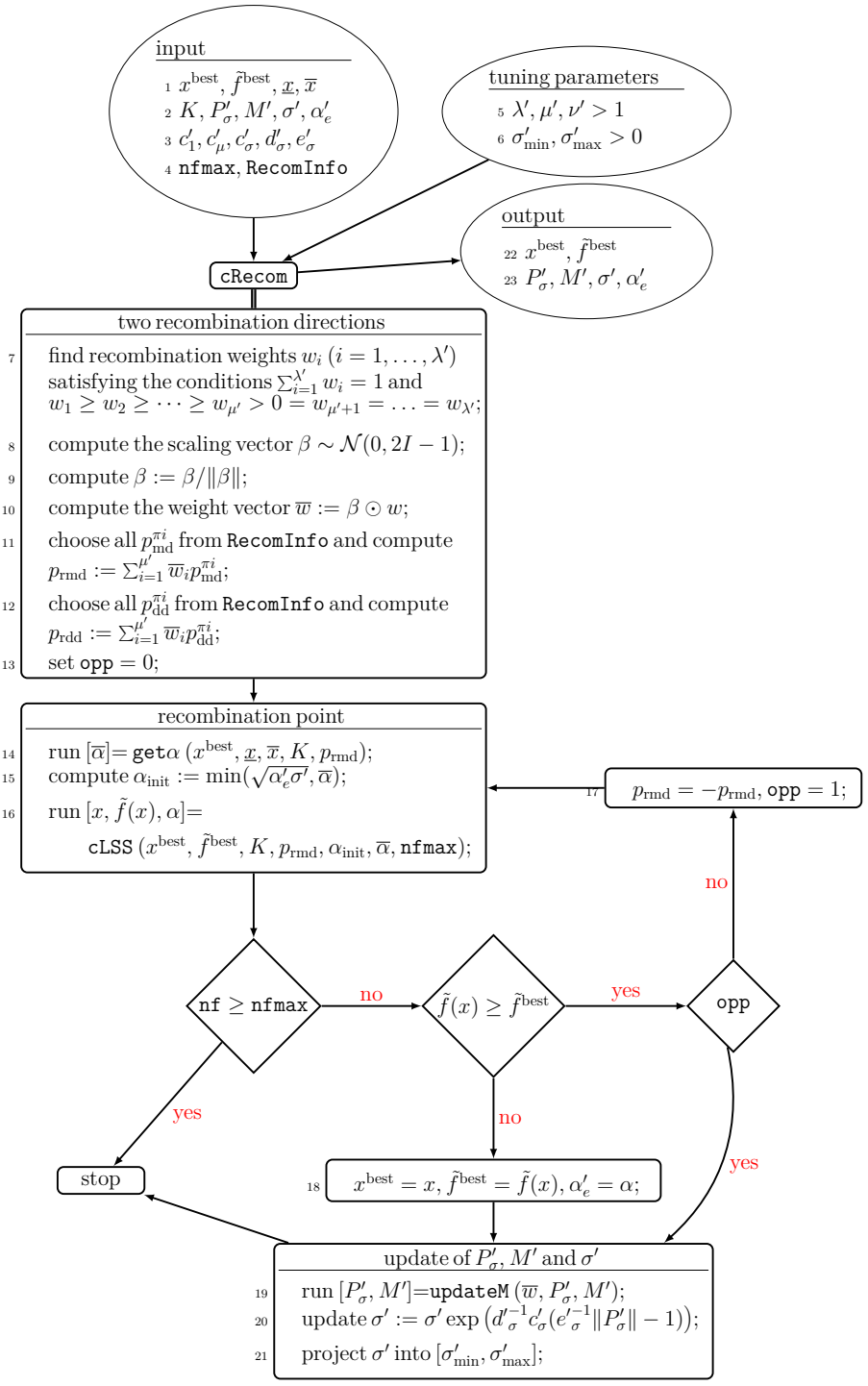
Algorithm 8 is pseudocode for `updateM`. As can be seen from line 6 of `updateM`, for an example, $\mu = \mu'$ is chosen when `cRecom` calls `updateM`, while $\mu = \mu''$ is chosen when `iRecom` calls `updateM`. This is the same for the other parameters defined in lines 1-6. In line 7 of `updateM`, the variance effective selection mass $\mu_{\overline{w}}$ and the normalization constant $\overline{c}_\sigma$ are computed, the second which is used in line 8 of `updateM` to update the evolution path P_σ and the recombination step size in lines 20 of `Recom` below. In line 9 of `updateM`, as in [5], the affine scaling matrix M is updated, where $0 < c_\mu \leq 1$ is a learning rate for updating M and $c_1 \leq 1 - c_\mu$ is a learning rate for the rank-one-update M (Section 3 discussed the numerical formulas for c_1, c_μ, and c_σ). In this rank-one-update: (i) The first term $c_{1,\mu}M$ includes the previous information and accumulates the information. (ii) The second term $c_1 M P_\sigma P_\sigma{}^T$ is the rank-one update, whose goal is to increase the probability of $p_{\mathrm{dd}}^{\pi i}$ $(i = 1, \ldots, \mu)$ for the next iteration, by maximizing the log-likelihood of $p_{\mathrm{dd}}^{\pi i}$ $(i = 1, \ldots, \mu)$. (iii) The third term $c_\mu \sum_{i=1}^{\mu} \overline{w}_i p_{\mathrm{md}}^{\pi i}(p_{\mathrm{dd}}^{\pi i})^T$ is the rank-μ update, whose goal is to take the mean of the estimated affine scaling matrices from all iterations.

The evolution path P_σ has two goals. Its first goal is to remedy losing the sign of $p_{\mathrm{dd}}^{\pi i}$ $(i = 1, \ldots, \mu)$ in the third term of M because $p_{\mathrm{dd}}^{\pi i}(p_{\mathrm{dd}}^{\pi i})^T = -p_{\mathrm{dd}}^{\pi i}(-p_{\mathrm{dd}}^{\pi i})^T$ and $p_{\mathrm{md}}^{\pi i} = M p_{\mathrm{dd}}^{\pi i}$. Its second goal is to update the recombination step size σ in line 20 of `cRecom`. Note that $p_{\mathrm{md}}^{\pi i}$ has been computed before in the mutation phase and here it only reuses, leading to $\mathcal{O}(n^2)$ operations due to the vector-matrix products. As a result, these three terms have different advantages and cause the affine scaling matrix behaves well in practice, compared to the rank-one update and rank-μ update.

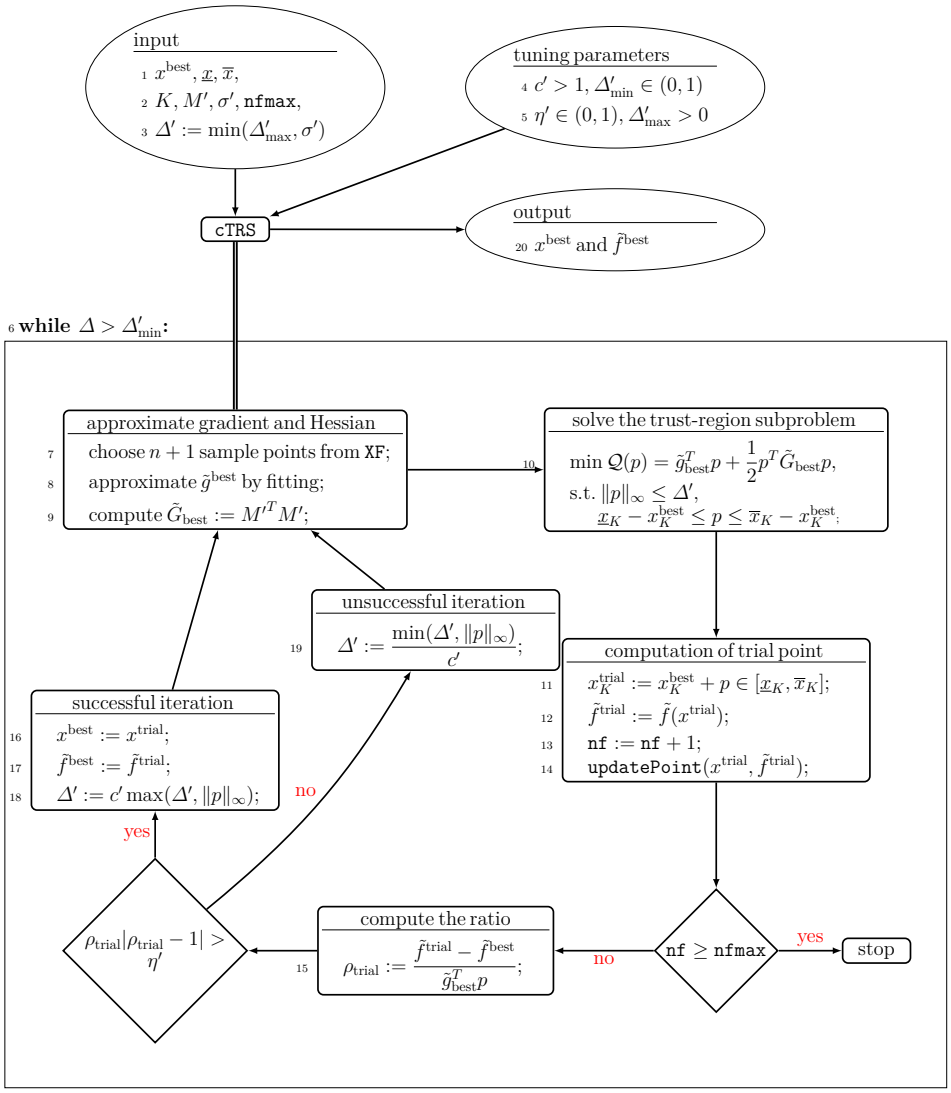
6.3.2 `cRecom`

Algorithm 9 is pseudocode for `cRecom`. `cRecom` computes the weights w_i $(i = 1, \ldots, \lambda')$ in line 7 and the scaling vector β in line 8, where $\mathcal{N}(0, 2I-1)$ is a normal distribution with zero mean and variance $2I-1$. Then it scales the weights w_i $(i = 1, \ldots, \lambda')$ by β in line 9 for reordering a possible fair sort during the selection phase when noise is high. This is a new feature of our algorithm. In line 10, $\odot$ denotes the componentwise product. In Section 3, we computed numerically w_i for $i = 1, \ldots, \lambda'$. Given `RecomInfo` $:= \{x^{\pi i}, \tilde{f}(x^{\pi i}), p_{\mathrm{dd}}^{\pi i}, p_{\mathrm{md}}^{\pi i}\}_{i=1}^{\mu}$, in lines 11-12 of `cRecom`, the continuous recombination mutation direction p_{rmd} and the continuous recombination distribution direction p_{rdd} are computed, which are used as input for `cLSS` in line 16 of `cRecom` and to update M' in line 9 of `updateM`, respectively. They are the weighted average of the μ' mutation directions and the weighted average of the μ' distribution directions, respectively. In line 13 of `cRecom`, `opp` $= 0$ is chosen, which means the opposite direction has not been tried yet. In line 16 of `cRecom`, to avoid the generation of too small step sizes and keep feasibility, the initial real step size α_{init} is updated only based on $\overline{\alpha}$, α'_e, and σ'. In line 15 of `cRecom`, `cLSS` is performed along $\pm p_{\mathrm{rmd}} \in \mathbb{R}^{|K|\times 1}$ to update x^{best} by extrapolation. `cRecom` terminates if `nf` reaches `nfmax`. If x^{best} cannot be updated, `cRecom` sets $p_{\mathrm{rmd}} = -p_{\mathrm{rmd}}$ in line 17, evaluates `opp` $= 1$, and computes α_{init} and $\overline{\alpha}$ in lines 14-15 and reruns `cLSS` along p_{rmd} to update x^{best}. Then, `cRecom` terminates if `nf` reaches `nfmax`. If $f(x) < \tilde{f}^{\mathrm{best}}$ holds, x, which is one of the trial points evaluated by extrapolation with the lowest inexact function value among all evaluated trial points, is accepted as the new best point in line 18 of `cRecom`. In lines 20-21 of `cRecom`, the real recombination step σ' is computed, where e'_σ is an approximate value of the expected value $\mathbf{E}(\|u\|)$ of the norm of the vector $u \sim \mathcal{N}(0, I)$, the constant $0 < \sigma'_{\max} < \infty$ is a maximum value for σ', $0 < \sigma'_{\min} < 1$ is a minimum value for σ', $c'_\sigma \leq 1$ is a learning rate for the cumulation for the step size, $d'_\sigma \approx 1$ is a damping parameter (cf. [17, Section 4]), see Section 3 the numerical formulas for d'_σ, c'_σ, and e'_σ. Moreover, as long as there is no feasible trial point along $\pm p_{\mathrm{rmd}}$, at most n'_{scale} times p_{rmd} rescales by $p_{\mathrm{rmd}} = \beta \odot p_{\mathrm{rmd}}$. Here n'_{scale} is a tuning parameter. This improvement can be found in the Matlab code of `cRecom`.

Algorithm 9 Pseudocode for `cRecom`



Algorithm 10 Pseudocode for cTRS



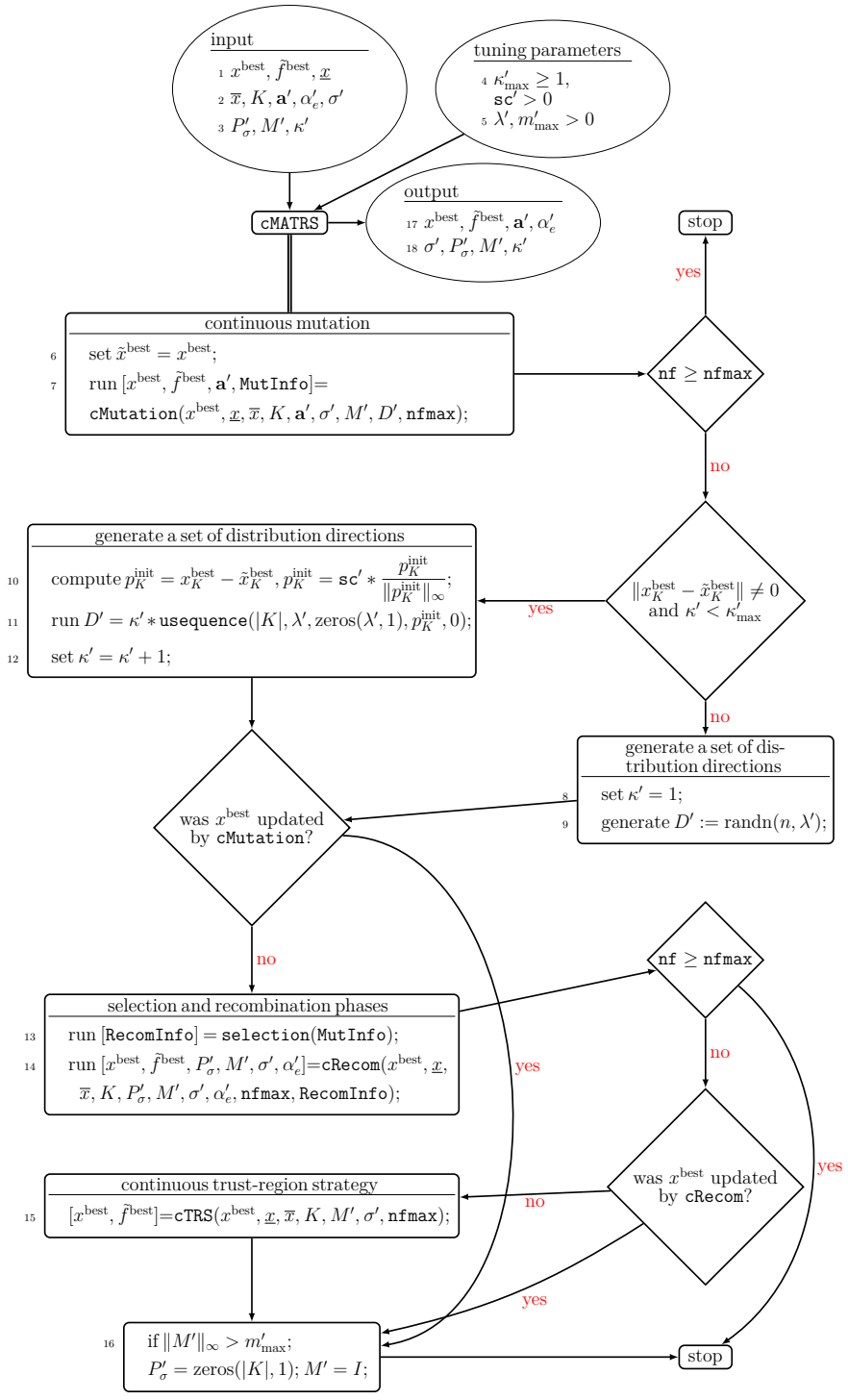
Algorithm 10 is pseudocode for `cTRS`. In line 3 of `cTRS`, $\Delta' > 0$ initially is chosen, where σ' is computed in line 21 of `cRecom` and $0 < \Delta'_{\max} < \infty$ is a tuning parameter. When the objective function value at each trial point of line search and trust-region strategies is computed, the function values and the corresponding points are saved in `XF`, whose $n+1$ current points and their function values are used to approximate $\tilde{g}_{\text{best}}$ in line 7 of `cTRS`. Then to form the trust-region subproblem, its approximate gradient vector $\tilde{g}_{\text{best}}$ is obtained by fitting in line 8 of `cTRS` as in HUYER & NEUMAIER [21] and its approximate symmetric Hessian matrix $\tilde{G}_{\text{best}} = {M'}^T M'$ is chosen in line 9 of `cTRS` as a new choice without additional cost. Afterwards, the trust-region subproblem defined in line 10 of `cTRS` is solved by `minq8` by HUYER & NEUMAIER [20]. After solving the trust-region subproblem, the continuous trial point x_K^{trial} and its inexact function value $\tilde{f}^{\text{trial}} := \tilde{f}(x^{\text{trial}})$ are computed in lines 11-12 of `cTRS` and the two different histories of points are updated in line 14 by `updatePoint`. If `nf` reaches `nfmax`, `cTRS` terminates. Given the tuning parameter $0 < \eta' < \frac{1}{4}$, if the sufficient descent condition $\rho_{\text{trial}}|\rho_{\text{trial}} - 1| > \eta'$ (suggested by KIMIAEI [22] for bound-constrained ill-conditioned problems) is satisfied, the current iteration of `cTRS` is called **successful** and x^{trial} is accepted as the new best point. Then, using the tuning parameter $c' > 1$, we expend Δ' in line 18 of `cTRS` by the traditional formula of CONN et al. [9]. Otherwise, the current iteration of `cTRS` is called **unsuccessful**. In this case, Δ' is reduced in line 19 of `cTRS`.

6.5 `cMATRS`

Algorithm 11 is pseudocode for `cMATRS`. The variable κ' is the counter for the number of times that the set D' of distribution directions must be generated by `usequence`. If `cMutation` cannot update x^{best} and κ' does not reach its upper bound $\kappa'_{\max} > 1$: (i) `cMATRS` computes the new combination direction p_K^{init} in line 10, where $\tilde{x}^{\text{best}}$ is the old best point and x^{best} is the current best point found and `sc`$'$ is a positive tuning parameter. For the next call to `cMutation`, there is a good chance to find decreases in the inexact function value by performing `cLSS` along at least one of $\pm p_K^{\text{init}}$, leaving points with large inexact function values and going into or moving down a valley. (ii) The set D' of random directions is regenerated in a new randomized way by `usequence` in line 11 of `cMATRS` with the goal of finding the new best point in a larger neighborhood of the old best point. Here κ' is updated in each iteration of `cMATRS` in lines 8 and 12. Otherwise, it sets $\kappa' = 1$ and distribution directions are selected from $\mathcal{N}(0, I)$ with zero mean and variance I.

In line 16 of `cMATRS`, if $\|M'\|_\infty$ is greater than a positive tuning parameter $m'_{\max}$, both the evolution path P'_σ and the affine scaling matrix M' are replaced

Algorithm 11 Pseudocode for `cMATRS`



by a zeros vector and an identity matrix, respectively, since large steps in `cMutation` and `cRecom` are one of the causes for line search failure.

6.6 An integer mutation phase

Pseudocode for `iMutation` is not introduced here because it shares the same structure as `cMutation`. Rather, here are some distinctions between `iMutation` and `cMutation`. They differ in how p^i_{dd} and p^i_{md} are computed and how $\overline{\alpha}^i$ and α^i_{init} are determined.

`iMutation` replaces lines 5 and 6 of `cMutation` by

$$p^i_{\mathrm{dd}} = D''_{:i} \in \mathbb{R}^{|I|\times 1}, \quad p^i_{\mathrm{md}} = \lceil M'' p^i_{\mathrm{dd}} \rfloor \in \mathbb{R}^{|I|\times 1}.$$

Here, integer distribution directions $p^i_{\mathrm{dd}} \in \mathbb{R}^{|I|\times 1}$ $(i = 1, \ldots, \lambda'')$ are chosen from a set D'' of integer directions, unlike `cMutation`, which selects continuous distribution directions from the normal distribution (D'' is initially a tuning matrix and it is updated in `iMATRS` below). Then integer mutation directions $p^i_{\mathrm{md}} \in \mathbb{R}^{|I|\times 1}$ are computed by rounding the product of the affine scaling matrix M'' and p^i_{dd} for $i = 1, \ldots, \lambda''$. Next, `iMutation` replaces lines 8 and 9 of `cMutation` by

$$[\overline{\alpha}^i] = \texttt{get}\alpha\ (x^{\mathrm{best}},\ \underline{x},\ \overline{x},\ I,\ p^i_{\mathrm{md}}),\ \overline{\alpha}^i := \max(1, \lfloor \overline{\alpha}^i \rfloor),$$

$$\alpha^i_{\mathrm{init}} := \min\left(\left\lfloor \sqrt{\sigma'' \mathbf{a}''_i} \right\rceil, \max(1, \lfloor \overline{\alpha}^i \rfloor)\right)$$

and line 10 of `cMutation` by

$$[x^i,\ \tilde{f}(x^i),\ \alpha^i] = \texttt{iLSS}\ (x^{\mathrm{best}},\ \tilde{f}^{\mathrm{best}},\ I,\ p^i_{\mathrm{md}},\ \alpha^i_{\mathrm{init}},\ \overline{\alpha}^i,\ \texttt{nfmax}).$$

`iLSS` and `cLSS` share the same structure although they differ in a few details. `cLSS` searches in the space of x_K, while `iLSS` searches in the space of x_I. In addition, their input (p^i_{md}, α^i_{init}, $\overline{\alpha}^i$) have been computed differently.

6.7 `iRecom`

Since `iRecom` has the same structure as `cRecom`, its pseudocode is not covered here. Instead, some differences between `cRecom` and `iRecom` are listed below. There are differences in the calculations of p_{rmd}, $\overline{\alpha}$, and α_{init} between them.

`iRecom` preserves line 11 of `cRecom`, but non-integer components of p_{rmd} (if any) are rounded to integer and replaces lines 14-16 of `cRecom`, respectively, by

$$[\overline{\alpha}]=\texttt{get}\alpha\ (x^{\text{best}},\ \underline{x},\ \overline{x},\ I,\ p_{\text{rmd}}),\ \overline{\alpha}:=\max(1,\lfloor\overline{\alpha}\rfloor),\ \alpha_{\text{init}}:=\min(\lfloor\sqrt{\alpha''_e\sigma''}\rceil,\overline{\alpha}),$$

$$[x,\ \tilde{f}(x),\ \alpha]=\texttt{iLSS}\ (x^{\text{best}},\ \tilde{f}^{\text{best}},\ I,\ p_{\text{rmd}},\ \alpha_{\text{init}},\ \overline{\alpha},\ \texttt{nfmax}).$$

Next, `iRecom` changes lines 19-21 of `iRecom` to

$$[P''_\sigma,\ M'']=\texttt{updateM}\ (\overline{w},\ P''_\sigma,\ M''),$$

$$\sigma'':=\left\lfloor\sigma''\exp\Big(\frac{c''_\sigma}{d''_\sigma}\Big(\frac{\|P''_\sigma\|}{e''_\sigma}-1\Big)\Big)\right\rceil\in[\sigma''_{\min},\sigma''_{\max}].$$

The affine scaling matrix M'' is computed by `updateM` without rounding its entries to integer. In the computation of integer mutation directions non-integer entries of these directions are rounded to integers. As in the continuous case, the real step size is computed, but rounded to integer. Moreover, if there is no feasible trial point along $\pm p_{\text{rmd}}$, at most n''_{scale} times $p_{\text{rmd}}:=\lfloor\beta\odot p_{\text{rmd}}\rceil$ is recomputed. Here n''_{scale} is a tuning parameter.

6.8 `iTRS`

Since `iTRS` and `cTRS` have the same structure, pseudocode for `iTRS` is not introduced here. For `iTRS` and `cTRS`, the trust-region subproblem is solved differently, and the trust-region radius is updated differently as well.

`iTRS` chooses the initial integer radius $\Delta''=\sigma''\in[\Delta''_{\min},\Delta''_{\max}]$. The two tuning parameters $\Delta''_{\max}>\Delta''_{\min}\geq 1$ control Δ'', so that it is neither too small or too large. Then, `iTRS` transforms the trust-region subproblem

$$\begin{aligned}&\min\ \mathcal{Q}(p)=\tilde{g}^T_{\text{best}}p+\frac{1}{2}p^T\tilde{G}_{\text{best}}p\\&\text{s.t.}\ \ \|p\|\leq\Delta'',\ \ p\ \text{integral},\\&\qquad x^{\text{best}}_I+p\in[\underline{x}_I,\overline{x}_I]\end{aligned}$$

into the bound-constrained integer least squares problem

$$\begin{aligned}&\min\ \tfrac{1}{2}\|M''p-r\|_2^2\\&\text{s.t.}\ \ \|p\|\leq\Delta'',\ \ p\ \text{integral},\\&\qquad x^{\text{best}}_I+p\in[\underline{x}_I,\overline{x}_I]\end{aligned}$$

by choosing $r:=-M''^{-T}\tilde{g}_{\text{best}}$ and setting

$$\tilde{g}^T_{\text{best}}p+\frac{1}{2}p^T\tilde{G}_{\text{best}}p=\frac{1}{2}\|M''p-r\|_2^2-\frac{1}{2}\|r\|^2.\tag{6}$$

This least squares problem is solved by a variant of Schnorr–Euchner search [7,13]. If the approximate solution of such a problem is zero, it is replaced by

$$p = x^{\text{best}} - x_1, \quad p = \Delta'' \left\lfloor p/\|p\| \right\rceil,$$

where x_1 is the first sample point used for computing $\tilde{g}_{\text{best}}$. In (6), the approximate gradient vector $\tilde{g}_{\text{best}}$ is obtained by fitting as in HUYER & NEUMAIER [21] and the approximate symmetric Hessian matrix $\tilde{G}_{\text{best}} = M''^T M''$ is chosen without additional cost. The objective function value at each trial point of line search and trust region is computed and saved in a list whose $n+1$ current points and their function values are used to approximate $\tilde{g}_{\text{best}}$.

Given the integer tuning parameters $\overline{\Delta} > 1$ and $1 < c'' < \infty$, `iTRS` updates the trust-region radius for unsuccessful iterations by

$$\Delta'' = \begin{cases} \lfloor \min(\|p\|_\infty, \Delta'')/c'' \rfloor & \text{if } \Delta'' \le \overline{\Delta}, \\ \Delta'' - 1 & \text{otherwise.} \end{cases}$$

Here the traditional formula $\Delta'' = \min(\|p\|_\infty, \Delta'')/c''$ of CONN et al. [9] is used to quickly reduce the trust-region radius to take advantage of small steps and increase the accuracy of the model function. The new choice $\Delta'' = \Delta'' - 1$ has the same goal as the traditional formula, but it reduces slowly the trust-region radius, which is not now large, and therefore `iTRS` may try many trial feasible points to update the best point before the radius becomes one. With the same goal as unsuccessful iterations, `iTRS` updates the trust-region radius for successful iterations by

$$\Delta'' = \begin{cases} \lfloor c'' \min(\|p\|_\infty, \Delta'') \rfloor & \text{if } \Delta'' \ge \overline{\Delta}, \\ \Delta'' + 1 & \text{otherwise.} \end{cases}$$

If an integer trust-region method cannot find new different feasible trial points, it terminates without updating the best point. Hence, `iTRS` carries out attempts to find feasible trial points that differs from the evaluated points in one of the following two ways: (i) Different sample points can be chosen randomly from the list of previous evaluated points to differently approximate the gradient of the trust-region subproblem. (ii) Different trust-region radii are generated in a new randomized way.

We first randomly select $n+1$ points from the list of stored evaluated points to compute g_{best}. If a new integer feasible point cannot be found by this change, to update the trust-region radius we randomly use at most `stuckmax` one of the two formulas

$$\Delta'' = \lfloor \Delta''/\zeta \rceil \quad \text{with } \zeta = \text{randi}([1, \Delta''_{\min}], 1)$$

and

$$\Delta'' = |\Delta'' + \text{sign}(\text{rand} - 0.5)\zeta| \quad \text{with } \zeta = \text{randi}([1, \Delta''_{\min}], 1)$$

until $\Delta'' \ge 1$. Here `stuckmax` ≥ 1 is a tuning parameter, $\Delta''_{\min} > 1$ is an integer tuning parameter, and rand and randi are as in Section 2.

Pseudocode for iMATRS is not discussed here because it shares the same structure as cMATRS. iMATRS and cMATRS differ in how they generate the set of distribution directions, solve trust-region subproblems, and update trust-region radii and line search step sizes.

iMATRS replaces line 7 of cMATRS by

$$[x^{\text{best}},\, \tilde{f}^{\text{best}},\, \mathbf{a}'',\texttt{MutInfo}] = \texttt{iMutation}(x^{\text{best}},\, \underline{x},\, \overline{x},\, I,\, \mathbf{a}'',\, \sigma'',\, M'',\, D'',\, \texttt{nfmax})$$

and then line 8 of cMATRS by $\kappa'' = 1$. It calls igeneratorD to compute the set D'' of integer distribution directions instead of lines 9 of cMATRS. For further information, refer to the Matlab code of igeneratorD for D'', which is one of three sets: a set of permuted coordinate directions, a set generated by usequence, and a combination of them. If iMutation cannot update x^{best} and κ'' does not reach its upper bound $\kappa''_{\max} > 1$, iMATRS replaces lines 10-12 of cMATRS by

$$p_I^{\text{init}} = x_I^{\text{best}} - \tilde{x}_I^{\text{best}}, \quad p_I^{\text{init}} = \texttt{sc}'' * \left\lceil \frac{p_I^{\text{init}}}{\|p_I^{\text{init}}\|_\infty} \right\rceil,$$

$$D'' = \texttt{usequence}(|I|, \lambda'', \kappa'' \,\text{ones}(\lambda'', 1), p_I^{\text{init}}, 0), \quad \kappa'' = \kappa'' + 1.$$

iMATRS computes the new combination direction p_I^{init}, where $\tilde{x}^{\text{best}}$ is the old best point and x^{best} is the current best point found and $\texttt{sc}''$ is a positive tuning parameter. For the next call to iMutation, there is a good chance to find decreases in the inexact function value by performing iLSS along at least one of $\pm p_I^{\text{init}}$, leaving points with large inexact function values and going into or moving down a valley. Next, iMATRS replaces line 14 of cMATRS by

$$[x^{\text{best}},\, \tilde{f}^{\text{best}},\, P_\sigma'',\, M'',\, \sigma'',\, \alpha_e''] = \texttt{iRecom}(x^{\text{best}},\, \tilde{f}^{\text{best}},\, \underline{x},\, \overline{x},\, I,\, P_\sigma'',\, M'',\, \sigma'',\, \alpha_e'',\, \texttt{nfmax},\, \texttt{RecomInfo})$$

and line 15 of cMATRS by

$$[x^{\text{best}},\, \tilde{f}^{\text{best}}] = \texttt{iTRS}(x^{\text{best}},\, \underline{x},\, \overline{x},\, I,\, M'',\, \sigma'',\, \texttt{nfmax}).$$

Large steps in iMutation and iRecom are one of the causes for line search failure. To avoid these, iMATRS changes line 16 of cMATRS by replacing the evolution path P_σ'' by a zeros vector and the affine scaling matrix M'' by an identity matrix if $\|M''\|_\infty$ is greater than a positive tuning parameter $m''_{\max}$.

6.10 A mixed-integer MATRS

6.10.1 miLSS

Algorithm 12 is pseudocode for miLSS. In line 5 of miLSS, the initial real and integer step sizes are chosen and in lines 6-8 of miLSS the first trial point in the space of all x and its function value is computed. Afterwards, if the second trial point and its function value can be computed in lines 10-13, then extrapolation at least in one of spaces of all x, x_K, and x_I is performed. After computing each trial point, if nf reaches nfmax, miLSS ends. If $\alpha' < \overline{\alpha}'$ and $\alpha'' < \overline{\alpha}''$ hold, miLSS does not reduce to cLSS or iLSS. Otherwise, if $\alpha' < \overline{\alpha}'$ holds, miLSS is converted to cLSS, and if $\alpha'' < \overline{\alpha}''$ holds, miLSS is converted to iLSS. After computing each trial point and its function value in lines 9, 14, 21, updatePoint is performed to update the two histories of points.

Extrapolation evaluates at least one more trial point, one of which with the lowest inexact function value is accepted as the new best point. To simplify miLSS, this case does not appear in its pseudocode.

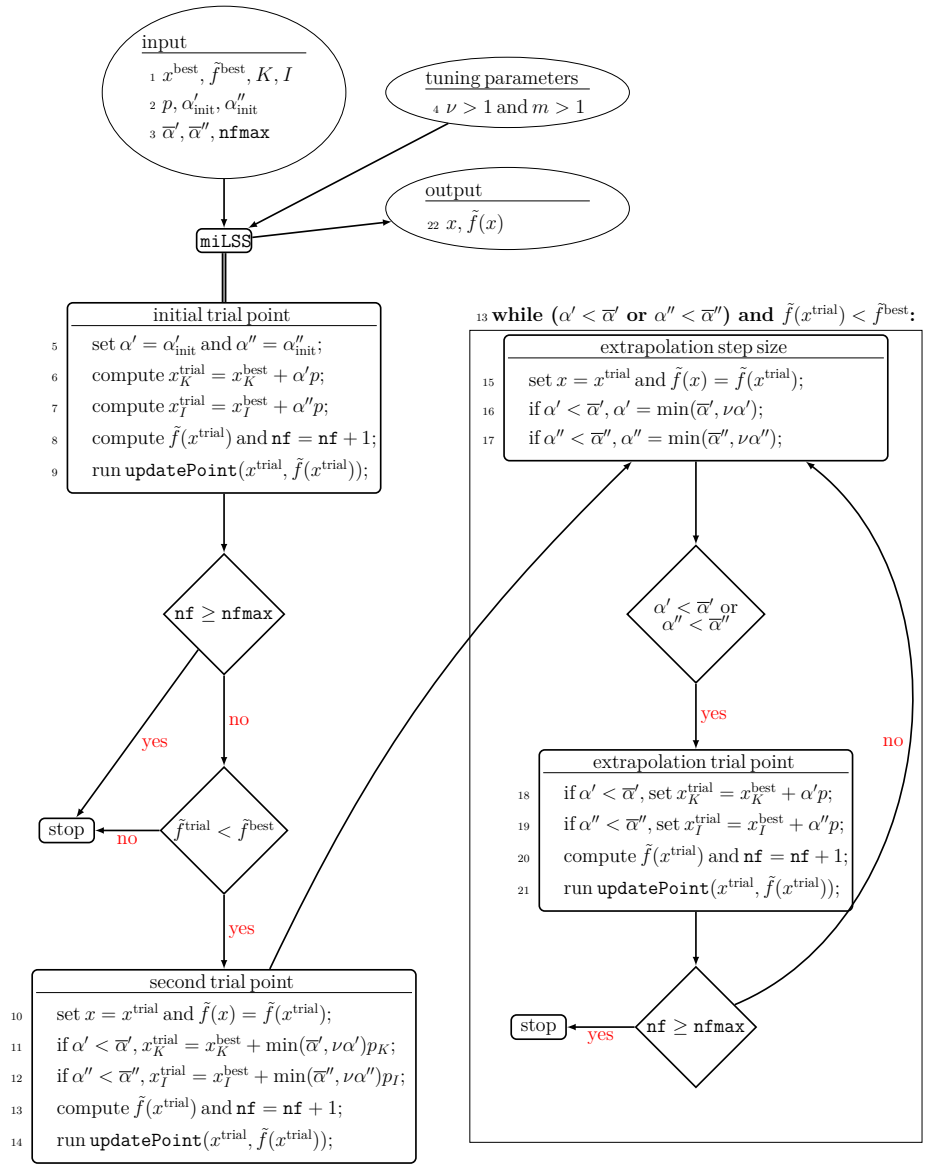
6.10.2 miMATRS

Algorithm 13 is pseudocode for miMATRS. To compute the first and (possibly) second combination directions, in line 9 of miMATRS, the scaling vector sc is computed by

$$
\begin{aligned}
&dX_{:i} = X_{:i} - x^{\mathrm{best}}, \quad \text{for } i = 1, \ldots, m, \quad \mathtt{sc} = |\sup(dX)|,\\
&\mathtt{sc}_j = 1, \quad \text{for } j \in J = \{j \mid \mathtt{sc}_j = 0\},\\
&\mathtt{sc}_j = \max(1, \lceil 1/\mathtt{sc}_j \rceil), \quad \text{for } j \in I, \quad \mathtt{sc}_j = \min(1, 1/\mathtt{sc}_j), \quad \text{for } j \in K.
\end{aligned}
$$

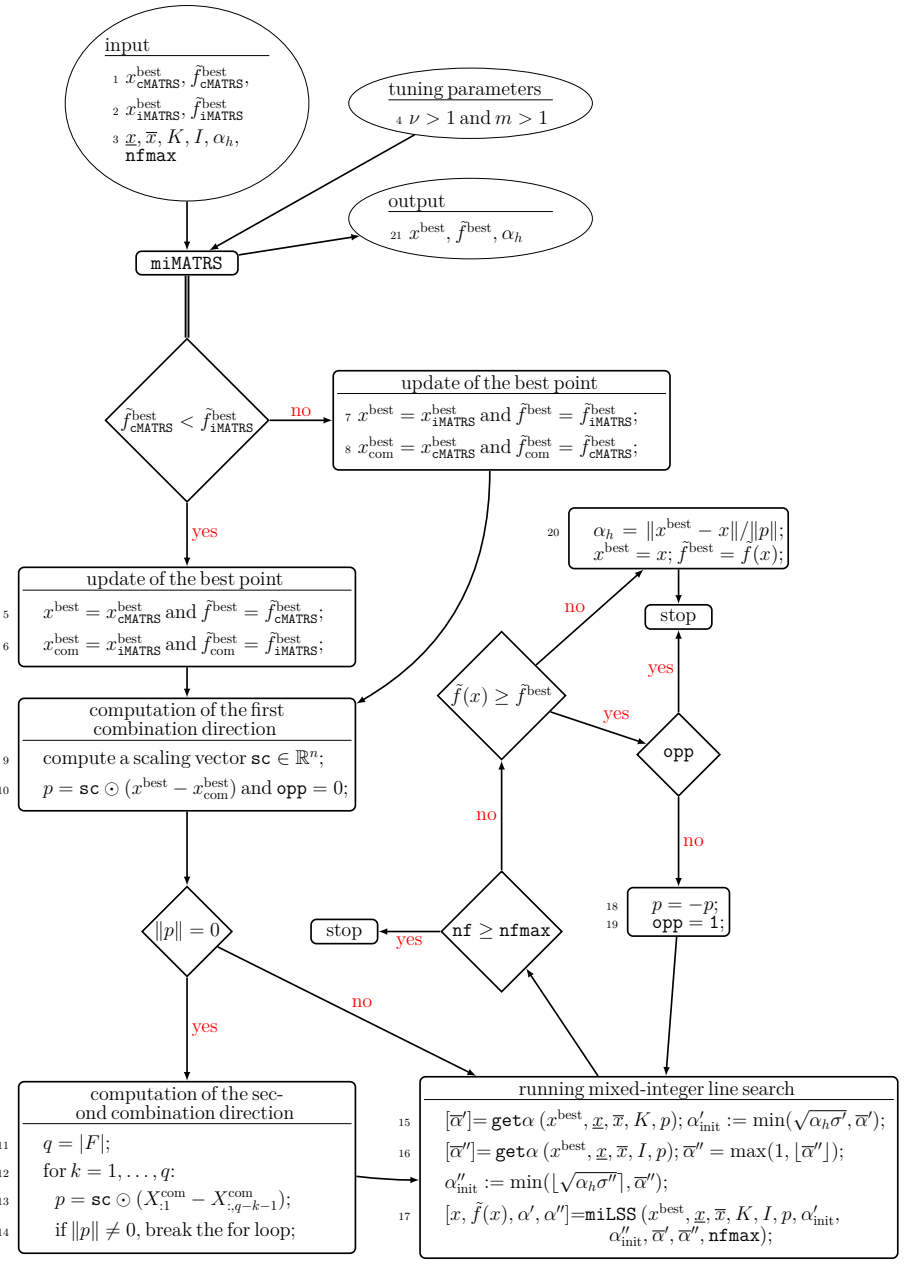
Here the tuning parameter m is the number of points used for the computation of combination directions. Then, in line 10, the product of sc and the difference p of the two best points found by cMATRS and iMATRS is computed and chosen as the first combination direction. In this case, at least one of cMATRS and iMATRS can update x^{best}. The Boolean variable opp = 0 is evaluated, which means miLSS has not been performed along the first combination direction yet. If $\|p\| = 0$, it means that both cMATRS and iMATRS could not update x^{best}. In this case, in line 13 of miMATRS the second combination direction is computed, which is the product of sc and the difference p of the best point and the worst point saved in XF. Then, miMATRS performs miLSS along this direction or its opposite direction in line 17 after the initial real and integer step sizes α'_{init} and α''_{init} and the largest real and integer allowed step size $\overline{\alpha}'$ and $\overline{\alpha}''$ are found as in lines 15-16 of cRecom and iRecom, respectively, but with the difference that the heuristic step size α_h is used instead of α'_e and α''_e. The reason for this

Algorithm 12 Pseudocode for `miLSS`



difference is that in practice finding α'_e and α''_e is difficult since extrapolation may be at least in one of the spaces of all x, x_I, and x_K.

Algorithm 13 Pseudocode for `miMATRS`



Acknowledgment The first author acknowledges financial support of the Austrian Science Foundation under Project No. P 34317.

References

1. M. A. Abramson, C. Audet, G. Couture, J. E. Dennis, Jr., S. Le Digabel, and C. Tribes. The NOMAD project. Software available at `https://www.gerad.ca/nomad/`.
2. C. Audet and W. Hare. *Derivative-Free and Blackbox Optimization*. Springer International Publishing (2017).
3. C. Audet, S. Le Digabel, and C. Tribes. The Mesh Adaptive Direct Search Algorithm for Granular and Discrete Variables. *SIAM J. Optim* **29** (2019), 1164–1189.
4. A. S. Bandeira, K. Scheinberg, and L. N. Vicente. Computation of sparse low degree interpolating polynomials and their application to derivative-free optimization. *Math. Program.* **134** (2012), 223–257.
5. H. G. Beyer. Design principles for matrix adaptation evolution strategies (2020). In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO'20). ACM, New York, 682–700. `https://doi.org/10.1145/3377929.3389870`.
6. J. Blank, K. Deb, Y. Dhebar, S. Bandaru, and H. Seada. Generating well-spaced points on a unit simplex for evolutionary many-objective optimization. *IEEE Trans. Evol.* **25** (2021), 48–60.
7. X. W. Chang, X. Yang, and T. Zhou. MLAMBDA: A modified LAMBDA method for integer least-squares estimation. *J. Geod.* **79** (2005), 552–565.
8. A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics (2009).
9. A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust region methods*. Society for Industrial and Applied Mathematics (2000).
10. J. Dick and F. Pillichshammer. *Digital nets and sequences: discrepancy theory and quasi–Monte Carlo integration*. Cambridge University Press (2010).
11. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.* **91** (2002), 201–213.
12. G. Fasano, G. Liuzzi, S. Lucidi, and F. Rinaldi. A linesearch-based derivative-free approach for nonsmooth constrained optimization. *SIAM J. Optim* **24** (2014), 959–992.
13. A. Ghasemmehdi and E. Agrell. Faster recursions in sphere decoding. *IEEE Trans. Inf. Theory* **57** (2011), 3530–3536.
14. T. Giovannelli, G. Liuzzi, S. Lucidi, and F. Rinaldi. Derivative-free methods for mixed-integer nonsmooth constrained optimization. *Comput. Optim. Appl.* **82** (2022), 293–327.
15. S. Gratton, Ph. L. Toint, and A. Tröltzsch. An active-set trust-region method for derivative-free nonlinear bound-constrained optimization. *Optim. Methods Softw.* **26** (October 2011), 873–894.
16. N. Hansen A CMA-ES for Mixed-Integer Nonlinear Optimization. [Research Report] RR-7751, INRIA. 2011. inria-00629689.
17. N. Hansen. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772* (2016).
18. K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), 62–72.
19. W. Huyer and A. Neumaier. Global optimization by multilevel coordinate search. *J. Glob. Optim.* **14** (1999), 331–355.
20. W. Huyer and A. Neumaier. MINQ8: general definite and bound constrained indefinite quadratic programming. *Comput. Optim. Appl.* **69** (2018), 351–381.

21. W. Huyer and A. Neumaier. SNOBFIT – stable noisy optimization by branch and fit. *ACM. Trans. Math. Softw.* **35** (2008), 1–25.
22. M. Kimiaei. An active set trust-region method for bound-constrained optimization. *Bull. Iran. Math. Soc.* **48** (2022), 1721–1745.
23. M. Kimiaei. A developed randomized algorithm with noise level tuning for large-scale noisy unconstrained DFO problems, Manuscript (2023). Available at `https://optimization-online.org/?p=16687`.
24. M. Kimiaei and A. Neumaier. Efficient composite heuristics for integer bound constrained noisy optimization, Manuscript (2022). Available at `https://optimization-online.org/2022/07/8998/`.
25. M. Kimiaei and A. Neumaier. Effective matrix adaptation strategy for noisy derivative-free optimization. *Math. Program. Comput.* (2024). `https://doi.org/10.1007/s12532-024-00261-z`.
26. M. Kimiaei and A. Neumaier. Efficient unconstrained black box optimization. *Math. Program. Comput.* (2022), 365–414.
27. M. Kimiaei, A. Neumaier, P. Faramarzi. New subspace method for unconstrained derivative-free optimization. *ACM. Trans. Math. Softw.* **49** (2023), 1–28.
28. Morteza Kimiaei. (2024). GS1400/MATRS: MATRSv2.0 (MATRSv2.0). Zenodo. `https://doi.org/10.5281/zenodo.10692874`
29. J. Larson, M. Menickelly, and S. M. Wild. Derivative-free optimization methods. *Acta Numer.* **28** (2019), 287–404.
30. G. Liuzzi, S. Lucidi, and F. Rinaldi. Derivative-free methods for bound constrained mixed-integer optimization. *Comput. Optim. Appl.* **53** (2011), 505–526.
31. G. Liuzzi, S. Lucidi, and F. Rinaldi. An algorithmic framework based on primitive directions and nonmonotone line searches for black-box optimization problems with integer variables. *Math. Program. Comput.* **12** (2020), 673–702.
32. G. Liuzzi, S. Lucidi, and F. Rinaldi. TESTINT - a collection of 240 inequality constrained plus 61 bound constrained test problems for black-box integer programming. DFL – derivative-free library, http://www.iasi.cnr.it/ liuzzi/dfl/ (2022).
33. J. J. Moré and S. M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM J. Optim.* **20** (2009), 172–191.
34. J. Müller. MISO: mixed-integer surrogate optimization framework. *Optim. Eng.* **17** (2015), 177–203.
35. T. Muranushi. Paraiso: an automated tuning framework for explicit solvers of partial differential equations newblock *Computational Science & Discovery.* **5** (2012), 015003.
36. H. Niederreiter. *Random number generation and quasi-Monte Carlo methods.* SIAM (1992).
37. N. Ploskas and N. V. Sahinidis. Review and comparison of algorithms and software for mixed-integer derivative-free optimization. *J. Glob. Optim.* **82** (2021), 433–462.
38. M. Porcelli and Ph. L. Toint. Exploiting problem structure in derivative free optimization. *ACM. Trans. Math. Softw.* **48** (2022), 1–25.
39. L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *J. Global. Optim.* **56** (2012), 1247–1293.
40. N. V. Sahinidis. *BARON 21.1.13: Global Optimization of Mixed-Integer Nonlinear Programs,* User's Manual (2017).
41. L. Seshagiri, M. S. Wu, M. Sosonkina, and Z. Zhang. Exploring tuning strategies for quantum chemistry computations. *Software Automatic Tuning: From Concepts to State-of-the-Art Results.* (2010), 193–208.
42. W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical DAG scheduling for hybrid distributed systems *2015 IEEE International Parallel and Distributed Processing Symposium.* (2015), 156–165.