manuscript No.
(will be inserted by the editor)

Effective matrix adaptation strategy for noisy derivative-free optimization

Morteza Kimiaei[1] · **Arnold Neumaier**

the date of receipt and acceptance should be inserted later

Abstract In this paper, we construct and implement two effective versions of the matrix adaptation evaluation strategy (`MAES`) for noisy derivative-free optimization problems, a fast version `MADFOF` and a limited memory version `MADFOL`. `MADFOF` uses Cholesky factorization of the covariance matrix to compute the search direction in a low-cost way compared to the traditional `MAES`, while `MADFOL` computes the search directions without storing the covariance matrix. Unlike the various `MAES` solvers, `MADFOF` and `MADFOL` use a new stochastic non-monotone line search condition to detect whether or not a reduction of the inexact function value has been found, and to generate candidate points with different heuristic step sizes. Like derivative-free line search algorithms for the noiseless case, `MADFOF` and `MADFOL` use extrapolation steps in an attempt to speed up the solution process. If this attempt does not reduce the inexact function value, up to five heuristically constructed points are tried, and the new point with the lowest inexact function value is accepted as the new point. A comparison with state-of-the-art solvers show that `MADFOF` and `MADFOL` are highly competitive in the presence of strong noise, having the lowest relative cost of function evaluations and the highest number of solved problems.

Keywords Noisy derivative-free optimization · evaluation strategy · heuristic optimization · stochastic optimization

2020 AMS Subject Classification: primary primary 90C15; 90C30; 90C56.

1. The author acknowledges the financial support ....

M. Kimiaei
Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria
E-mail: kimiaeim83@univie.ac.at
`WWW:http://www.mat.univie.ac.at/~kimiaei/`

A. Neumaier
Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria
E-mail: Arnold.Neumaier@univie.ac.at
`WWW:http://www.mat.univie.ac.at/~neum/`

1 Introduction

In this paper we address the problem of minimizing the noisy derivative-free optimization (NDFO) problem

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t.}\ \ x \in \mathbb{R}^n \end{aligned} \tag{1}$$

without constraints, assuming (throughout the paper) that the smooth real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ is available through a noisy oracle that takes $x \in \mathbb{R}^n$ and gives an approximated function value $\tilde{f}(x)$ of $f(x)$. The noise may be *deterministic* or *stochastic*. Sources of deterministic noise may be modelling, truncation, and/or discretization errors, and sources of stochastic noise may be rounding errors, simulation noise, or inaccurate measurements.

The algorithms do not assume any knowledge of the structure of the objective function, the true gradient or its Lipschitz constant, and the statistical properties of the noise. Some assumptions on these are, however, made in the analysis of the algorithms.

In finite precision arithmetic, the goal is to find an *ε-approximate stationary point* of the unconstrained NDFO problem, i.e., a point whose unknown exact gradient is below a given threshold $\varepsilon > 0$. To find out which solver can find ε-approximate stationary points of the unconstrained NDFO problem faster and cheaper than the state-of-the-art solvers in terms of the relative cost of function evaluations, we use three different profiles: Performance profile (Dolan and Moré [13]), data profile (Moré and Wild [37]), and Morales profile (Morales and Nocedal [36]). Accordingly, we say that a solver is *efficient* if it has *lowest cost of function evaluations* and *robust* if it has *highest number of solved problems*. Therefore, we say that a solver is *competitive* if it is efficient and robust.

There are many solvers that can be used to solve the unconstrained NDFO problem, such as derivative-free line search based solvers (cf. Larson et al. [32, Section 2.3.4]), derivative-free trust region based solvers ([32, Section 2.4]), direct search solvers (see e.g., [32, Section 2.1]), matrix adaptation evaluation strategies (see e.g., Auger and Hansen [5], Loshchilov et al. [34], Beyer [7], Beyer and Sendhoff [8]). Two books with some historical references for DFO are Audet and Hare [4] and Conn et al. [11]. For the behaviour of these solvers, for the noiseless case see Rios and Sahinidis [42] and Kimiaei and Neumaier [28] and for the noisy case see Kimiaei [26]. Other useful references for noisy DFO are Berahas et al. [6], Chen [10], Elster and Neumaier [14], Gratton et al. [16,17], Gratton et al. [18], Huyer and Neumaier [23], Lucidi and Sciandrone [35], Moré and Wild [38], Powell [40,41], Shi et al. [43], and Wild et al. [46],

In the presence of strong noise, derivative-free line search and trust region solvers using approximate gradients with finite difference methods and Hessians with quasi-Newton methods cannot preserve their efficiency and robustness, even when enriched by techniques (e.g., Moré and Wild [38] and Shi et al. [43]) that estimate noise. These solvers require many function evaluations for the estimation gradients and some function evaluations for the estimated noise, sometimes even many function evaluations if the estimated noise is not detected in the first attempt. If these line search and trust region solvers cannot find a reduction in the inexact function value when the noise is high, the trust region radii and the line search step sizes become too small, while the generated points may be far from an ε-approximate stationary point of the unconstrained NDFO problem, resulting in a failure. These solvers are suitable in the presence of rounding errors.

One way to improve the line search and trust region methods is to replace the inexact function value at the old point with a non-monotone term in the line search condition and the trust region ratio, respectively, with the goal of finding a point with lowest inexact function value when noise is present, the function is flat, or the valley is narrow, e.g., see Ahookhosh and Amini [1], Amini et al. [2], Birgin et al. [9], Diniz-Ehrhardt [12], Grippo and Rinaldi [20], Grippo et al. [19], Kimiaei [25], Kimiaei and Neumaier [29], Kimiaei and Rahpeymaii [31], Kimiaei et al. [27], Lucidi and Sciandrone [35], and Toint [44].

As with derivative-free line search and trust region solvers, the step sizes of matrix adaptation evaluation strategy solvers may become too small in the presence of strong noise, resulting in candidate points that are not as close as possible to an ε-approximate stationary point of the unconstrained NDFO problem, and the algorithm fails. Moreover, these solvers do not have a descent condition to check whether the inexact function value is reduced or not. Therefore, in both the noiseless and noisy cases, these solvers may accept a point that is far from an ε-approximate stationary point. Therefore, they must use a descent condition such as the line search condition to accept a point not far from an ε-approximate stationary point of the unconstrained NDFO problem, provided that such a line search condition is enriched by estimating noise or a non-monotone term.

The model-based solvers (cf. [32, Section 2.2]) are only effective for problems in low dimensions in the presence of noise, but cannot handle problems in medium to high dimensions because a large number of sample points is needed to construct fully quadratic models. Kimiaei [26] handled some of these model-based solvers in random subspaces to handle problems in medium and high dimensions and showed that these solvers are robust but not very efficient for problems in medium dimensions.

Efficiency and robustness of direct search solvers depend on the dimension, which is reduced by increasing the dimension, e.g., see Kimiaei [26], Torczon [45], and Wright [47]. These solvers are not faster than the derivative-free line search solvers because they ignore extrapolation along fixed directions; see [26, 28].

The goal of this paper is to develop a new solver that combines many new techniques to be efficient and robust in the presence of noise compared to the state-of-the-art derivative-free solvers. In practice, our solver addresses the drawbacks of derivative-free line search and evaluation strategy solvers while preserving their advantages. Four important components of our solver are *a new stochastic non-monotone formula*

inserted into the line search condition, *heuristic step sizes* that depend on the largest absolute old point over the absolute direction in the component, *subspace directions* based on the old and current weighted average covariance search directions, and a *heuristic technique* to hopefully find a point with lowest inexact function value when no reduction in the inexact function value can be found.

1.1 Derivative-free line search solvers

Derivative-free line search solvers are among the fast and robust solvers for noisy derivative-free optimization problems, e.g., `VRBBO` by Kimiaei and Neumaier [28], `VRBBON` by Kimiaei [26], and `SDBOX` by Lucidi and Sciandrone [35]; for other methods, e.g., see [32, Section 2.3.4]. `FMINUNC` by Matlab Optimization Toolbox and `SSDFO` by Kimiaei et al. [30] are two other line search-based solvers which are effective for the noiseless case, ultimately in the presence of rounding errors.

Derivative-free line search solvers can be divided into two classes depending on whether the gradient is estimated or not. `VRBBO`, `VRBBON`, and `SDBOX` do not use gradient estimation in the line search condition, although `VRBBO` and `VRBBON` use gradient estimation to generate some heuristic techniques.

`FMINUNC` and `SSDFO` estimate the gradient using the finite difference method and use the approximate gradient in the directional derivative of the approximate Wolfe line search conditions. Further classification is based on whether these solvers are randomized or not. `VRBBO` and `VRBBON` are randomized, but `FMINUNC`, `SSDFO`, and `SDBOX` are deterministic. As shown in [26],
• `FMINUNC` is numerically very poor at small to large noise because the approximate gradient is not accurate due to noise, leading to poor quasi-Newton directions;
• `VRBBON`, `VRBBO`, and `SDBOX` are robust line search solvers at low to high noise for problems in low to high dimensions and are effective for problems in medium to high dimensions.

The efficiency of derivative-free line search with the finite difference technique for the gradient approximation strongly depends on whether noise can be estimated in an efficient way or not. Moré and Wild [38] try to find an interval for the step size t without giving a guarantee, satisfying

$$|\Delta(t)| \geq \tau_1 \omega_f \quad \text{and} \quad |\tilde{f}(x \pm tp) - \tilde{f}(x)| \leq \tau_2 \max\{|\tilde{f}(x)|, |\tilde{f}(x \pm tp)|\}, \tag{2}$$

for a direction $p \in \mathbb{R}^n$ with $\|p\| = 1$, where $\tau_1 \gg 1$, $\tau_2 \in (0, 1)$, and ω_f is the noise level, hopefully prevent the production of too small/large t. Denote by $\tilde{g}_i(x)$ the ith component of the gradient at x estimated by the finite difference method and by h_i the ith finite difference step size. Shi et al. [43] gave an adaptive estimation of the constants L_i for the forward finite difference formula

$$\tilde{g}_i(x) = \frac{\tilde{f}(x + h_i e_i) - \tilde{f}(x)}{h_i} \quad \text{with } h_i = \sqrt[4]{8}\sqrt{\frac{\omega_f}{L_i}} \tag{3}$$

by a second-order difference $\Delta(t) := \tilde{f}(x + tp) - 2\tilde{f}(x) + \tilde{f}(x - tp)$ for a direction $p \in \mathbb{R}^n$ with $\|p\| = 1$, leading to $L \approx \Delta(t)/t^2$. For this estimation, [43, Procedure I]

has been proposed. This procedure involves two steps. In the first step, t_i is calculated for $i = 1, 2, \cdots, n$ in the same way as [38]. If (2) is forced by such a t_i, then $L_i = \max\{0.1, |\Delta(t_i)|/t_i^2\}$ is computed which is used in (3). Otherwise, $L_i = 0.1$ is chosen. Then, the vector $\mathrm{L} = (L_1, \cdots, L_n)$ is obtained and the second-order differencing interval is calculated to $t = \|\mathrm{L}\|/\sqrt{n}$, which is used in the directional derivative in the approximate Wolfe line search. In the second step, in each iteration of the L-BFGS algorithm [33], if the line search finds a step size smaller than half, L is estimated as in the first step. Although these strategies are suitable in the presence of rounding errors , they are not suitable in the presence of high noise and sometimes require more additional function evaluations to estimate noise. Using these methods to estimate the noise in `VRBBON`, `VRBBO`, and `SDBOX` is not recommended because these solvers do not use the approximate directional derivative in the line search condition. If one can estimate the noise, even if the noise is very large, then the line search based solvers `VRBBON`, `VRBBO`, and `SDBOX` can work more effectively since they do not use the approximate directional derivative in the line search condition.

Let us describe the main ingredient – *extrapolation* – of these line search solvers. As long as the inexact function values are reduced, extrapolation increases the step sizes and computes the new trial points and their inexact function values along a fixed direction. By extrapolation, these line search solvers can actually obtain an ε-approximate stationary point of the unconstrained NDFO problem quickly in the noiseless case, but in the noisy case if a stochastic non-monotone term can be added to the line search condition. As in [43, Shi et al.], applying these estimator noise methods to the L-BFGS algorithm is useful, but not in the presence of high noise, since they approximate the gradient by the finite difference method and use the approximate directional derivative in the line search condition.

1.2 Evaluation strategy

Evaluation strategy is an algorithm that performs repeated interaction of variation through three phases (mutation, selection, and recombination) (cf. [3]). *Mutation* is a perturbation with zero mean and *selection* means to select some individuals (candidate solutions) with the increasing sorted inexact function values to make them the parents of the next generation (iteration). *Recombination* means to choose a new mean for the distribution. The evaluation strategy goes back to the principle of biological evolution. In summary, in each generation the first and second phases produce new individuals by changing the current parental individuals, possibly in a random way. The third phase then generates the parents of the next generation.

`CMAES` (*covariance matrix adaptation evolution strategy*) is a well-known numerical randomized derivative-free optimization methods used for solving nonlinear or non-convex continuous optimization problems with possibly noisy objective functions. The search distribution of `CMAES` is a *multivariate normal distribution* $\mathcal{N}(0, C)$ with zero mean and covariance matrix C, which specifies the pairwise dependencies between the variables in the distribution. Indeed, `CMAES` updates the covariance matrix and is particularly useful when the objective function is ill-conditioned. There are different ways to update C, e.g., Auger and Hansen [5] (the `CMAES` solver), Loshchilov

et al. [34] (the `LMMAES` solver), Beyer [7] (the `fMAES` solver), Beyer and Sendhoff [8] (the `BiPopMAES` solver). Inspired by the approximation of the inverse Hessian matrix in the quasi-Newton methods, a second order model of the objective function by an adaptation of the covariance matrix can be constructed. No derivative is needed, only the ascending rank order of the inexact function values of the candidate solutions is used to learn the sample distribution. Thus, unlike most traditional optimization methods, the nature of the underlying objective function requires fewer assumptions. Therefore, these solvers are preferable to the quasi-Newton methods in terms of the smaller number of function evaluations used. However, these solvers do not care whether the inexact function value is reduced or not when updating the step sizes. Therefore, they may accept points that are far from an ε-approximate stationary point of the unconstrained NDFO problem.

We design several ways to improve covariance matrix adaptation evaluation strategy solvers that have not been used before:
• A stochastic non-monotone formula applied in the line search condition can be used to check whether or not a reduction of the inexact function value is found, and then a variant of extrapolation can be used to quickly obtain an ε-approximate stationary point of the unconstrained NDFO problem, such as the derivative-free line search that does not use the approximate directional derivative.
• The adjustment of step sizes is essential from a heuristic point of view, since step sizes can become too small, leading to the null step (a scaled direction whose norm is zero, used in recombination) before finding an ε-approximate stationary point of the unconstrained NDFO problem, leading to slow convergence or even failure.
• Adjustments of directions are necessary because the effect of selection is significantly reduced (leading to the incorrect ascending rank order of the inexact function values of the candidate solutions) due to strong noise and the algorithm cannot generate effective directions for recombination. In this case, the directions can be constructed in a subspace spanned by the previous directions and the current direction, since the subspace can contain the information that was not missed by the accumulating noise.
• The heuristic generation of different step sizes can enrich selection and lead to the generation of candidate points with a meaningful reduction in function values. In fact, basic selection does not seem to be effective, affecting recombination, because a fixed step size (fixed standard deviation) is used to compute all candidate points.

1.2.1 `fMAES` and `LMMAES`

This subsection gives the details of the solvers `fMAES` and `LMMAES` mentioned above.

Unlike `CMAES` with the $\mathcal{O}(n^3)$ operation needed to compute the covariance matrix C^t, `fMAES` uses Cholesky factorization to obtain the matrix $M^t = \sqrt{C^t}$ with the $\mathcal{O}(n^2)$ operation. Both `fMAES` and `LMMAES` generate a population of new search points

$$x_i^t \sim y^t + \sigma_t \mathcal{N}(0, M^t) \sim \mathcal{N}(y^t, \sigma_t^2 M^t) \quad \text{for } i = 1, 2, \cdots, \lambda, \tag{4}$$

which is a perturbation of the current solution vector y^t. Here $\lambda \geq 2$ is the *sample size* and σ_t is the *scaling factor for the mutation phase*, $\mathcal{N}(0, M^t)$, at the iteration t.

Moreover, $\sim$ stands for the same distribution in left and right side and $M^t \in \mathbb{R}^{n\times n}$ is a matrix of the search distribution which is initially the identity matrix $M^0 = I$ or zero matrix $M^0 = O$. Candidate solutions are sorted as

$$\{x^t_{i:\lambda}\}^{\lambda}_{i=1} = \{x^t_i\}^{\lambda}_{i=1} \text{ with } \tilde{f}(x^t_{1:\lambda}) \leq \tilde{f}(x^t_{2:\lambda}) \leq \tilde{f}(x^t_{\mu:\lambda}) \leq \tilde{f}(x^t_{\mu+1:\lambda}) \leq \cdots \leq \tilde{f}(x^t_{\lambda:\lambda}),$$

where μ denotes the *parent number* or the *number of selected search points in the populations*. The weights w_i of recombination satisfy

$$\sum_j w_j = \sum_{i=1}^{\lambda} w_i \quad \text{and} \quad w_1 \geq w_2 \geq \cdots \geq w_\mu > 0 \geq w_{\mu+1} \geq w_\lambda.$$

Using these weights, the best value of *variance effective selection mass*

$$\mu_w := \frac{\|w\|_1^2}{\|w\|_2^2} = \frac{1}{\sum_{i=1}^{\mu} w_i^2} \in [1, \mu]$$

is approximately $\lambda/4$ (cf. [21]). It is used in updating the *evaluation path*

$$P^0_\sigma = 0, \quad P^{t+1}_\sigma = (1-c_\sigma)P^t_\sigma + \bar{c}_\sigma \sum_{i=1}^{\mu} w_i \mathbf{y}^{t+1}_{i:\lambda} \quad \text{with } \bar{c}_\sigma := \sqrt{c_\sigma(2-c_\sigma)\mu_w}$$

with the *selected steps*

$$\mathbf{y}^{t+1}_{i:\lambda} := \frac{x^{t+1}_{i:\lambda} - y^t}{\sigma_t} \quad \text{for } i = 1, \cdots, \mu, \tag{5}$$

whose goal is to update the step size

$$\sigma_{t+1} = \sigma_t \exp\left(\frac{c_\sigma}{d_\sigma}\Big(\frac{\|P^{t+1}_\sigma\|}{\mathbf{E}\|\mathcal{N}(0,I)\|} - 1\Big)\right).$$

Here $c_\sigma \leq 1$ is the *learning rate* for the cumulation for the step size and $d_\sigma \approx 1$ is damping parameter and $\mathbf{E}$ denotes the expectation value (cf. [21, Section 4]). `fMAES` updates the new matrix

$$M^{t+1} = (1 - \frac{c_1}{2} - \frac{c_\mu}{2})M^t + \frac{c_1}{2} d^t_\sigma P^{t+1}_\sigma (P^{t+1}_\sigma)^T + \frac{c_\mu}{2} \sum_{i=1}^{\mu} w_i d^{t+1}_{i:\lambda} (\mathbf{y}^{t+1}_{i:\lambda})^T \tag{6}$$

and computes the *covariance search direction*

$$d^t_\sigma = M^t P^t_\sigma. \tag{7}$$

Here $0 < c_\mu \leq 1$ is the *learning rate for updating the covariance matrix* and $c_1 \leq 1 - c_\mu$ is the *learning rate for the rank-one update of the covariance matrix*. The formula (6) is a combination of the *rank-one update*

$$M^{t+1} = (1-c_1)M^t + c_1 d^t_\sigma P^{t+1}_\sigma (P^{t+1}_\sigma)^T \tag{8}$$

and the *rank-μ update*

$$M^{t+1} = (1 - c_\mu)M^t + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{t+1} (\mathbf{y}_{i:\lambda}^{t+1})^T. \tag{9}$$

The weighted selection used in the third term of (6) and second term of (9) lead to a better covariance matrix. In (9), when $c_\mu = 0$, learning is not taken and when $c_\mu = 1$ prior information is not retained.

Maximum likelihood estimator (`MLE`) of the covariance matrix differs slightly from the unbiased estimator; e.g., the empirical covariance matrix (mean of the actual realized sample) which is an unbiased estimator of the original covariance matrix (the true mean value). `MLE` maximizes a likelihood function such that the observed data are most likely under the assumed statistical model. In the rank-μ update (9), the second term is a result of maximizing a log-likelihood. Hence, the rank-one update for the covariance matrix inserts the maximum likelihood term into the old estimate of the covariance matrix. Its goal is to increase the probability of candidate solutions and search steps such that the likelihood of previously candidate solutions is maximized. The rank-μ update for the covariance matrix is the mean of the estimated covariance matrices from all iterations, which is a reliable estimate for the selected steps whose weights are equal.

`LMMAES` [34] handles `fMAES` for problems in high dimensions by computing $d_\sigma^t = M^t P_\sigma^t$ like limited memory quasi-Newton directions Liu and Nocedal [33], not requiring to restore M^t. On the other hand, the evaluation path P_σ^t, used to identify the sign of selected steps (5), is lost in calculating M^{t+1} in (6), which leads to a better M^{t+1} when μ_w is small. To remedy this problem, `LMMAES` uses the *conjugate evaluation path*

$$P_m^0 = 0, \quad P_m^{t+1} = (1 - c_m)P_m^t + \overline{c}_m \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{t+1} \quad \text{with } \overline{c}_m := \sqrt{c_m(2 - c_m)\mu_w}.$$

Here $c_m \leq 1$ is the *learning rate* for the cumulation for the rank-one update. In fact, `LMMAES` uses P_σ^t only for updating σ_t. By setting $c_\mu = 0$ and $M^0 = I$ in (6), using $M^1 = (1 - \frac{c_1}{2})I + \frac{c_1}{2} P_m^1 (P_m^1)^T$ and P_m instead of P_σ^t, `LMMAES` computes in the first iteration

$$d_{i:\lambda}^1 = M^1 \mathbf{y}_{i:\lambda}^1 = \mathbf{y}_{i:\lambda}^1 (1 - \frac{c_1}{2}) + \frac{c_1}{2} P_m^1 \Big((P_m^1)^T \mathbf{y}_{i:\lambda}^1 \Big).$$

Then, using

$$M^i = (1 - \frac{c_1}{2})I + \frac{c_1}{2} P_m^i (P_m^i)^T \quad \text{for } i = 1, 2, \cdots, t-1, \tag{10}$$

`LMMAES` does not require saving M^t to compute $d_{i:\lambda}^t$ since $(P_m^i)^T \mathbf{y}_i^t$ is scaler for $i = 1, 2, \cdots, t-1$. Hence `LMMAES` recursively uses, for $i = 1, 2, \cdots, t-1$, the form

$$d_{i:\lambda}^t = M^1 M^2 \cdots M^{t-1} \mathbf{y}_{i:\lambda}^t \tag{11}$$

with $\mathcal{O}(nm_{\max})$ operation and without saving $M^1, M^2, \cdots, M^{t-1}$. Here $t \ll n$ and is computed by $t = \min(m_{\max}, n)$, where $m_{\max}$ is the maximum number of P_m^i for $i = 1, 2, \cdots, t-1$. Consequently, `LMMAES` takes advantage of the rank-one update and tends to increase the probability of sampling from $\mathcal{N}(0, M^{t+1})$.

[34, Algorithm 1] discusses the similarity and difference of various `CMAES` methods and includes the implementation of `LMMAES` and `fMAES`.

As apparent from [34, Algorithm 1], `fMAES` and `LMMAES` are identical except for their treatment of the covariance matrix. Anticipating their later refinement to our new algorithms `MADFOF` and `MADFOL` we describe them in a single algorithm `MAESB` with two options `MAESF` (the fast version) and `MAESL` (the limited memory version), defined in the variable `ver`.

Flowchart (a)-(c) of Figure 1 shows a simple structure of `MAESF` and `MAESL`. Let us describe how `MAESF` and `MAESL` work:
• Tuning parameters are described in lines 1-4 of Algorithm 1.
• Then some necessary information for `mutation-basic` (basic mutation), `selection-basic` (basic selection), and `recombination-basic` (basic recombination) in lines 5–9 of Algorithm 1 are initialized.
• Then the main loop is started in lines 10–21 of Algorithm 1. This loop repeatedly performs three phases `mutation-basic`, `selection-basic`, and `recombination-basic` until an ε-approximate stationary point of the unconstrained NDFO problem is not found and the maximum number `nfmax` of function evaluations is not reached.

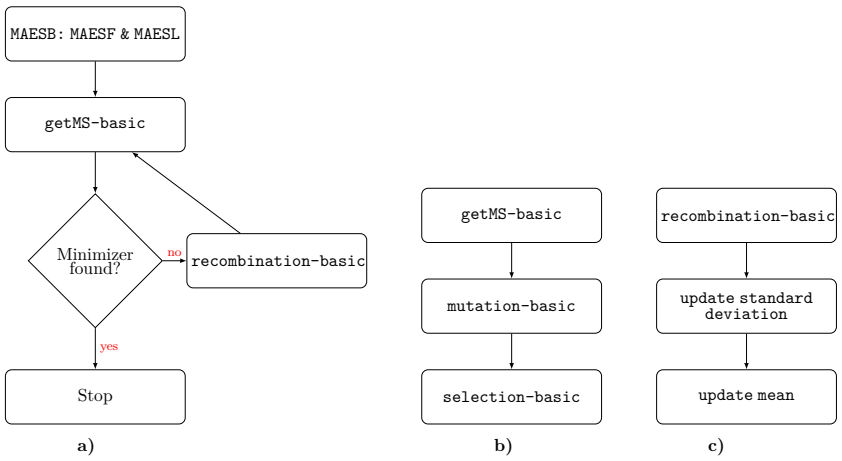


Fig. 1: Flowchart for (a) `MAESF` and `MAESL`, (b) `getMS-basic`, (c) `recombination-basic`.

`mutation-basic` and `selection-basic` are performed by calling `getMS-basic`, while `recombination-basic` uses `updateInfo` to update the information required to obtain the new standard deviation of distribution and then computes the new mean of distribution.

`mutation-basic` suffers the fixed standard deviation (step size) for generating candidate points, which affect the ability of both `selection-basic` and `recombination-basic`.

`selection-basic` computes the weighted average d_w of the μ-covariance search directions and `recombination-basic` uses d_w to compute the new mean of the distribution. Due to high noise, sorting the inexact function values at the candidate points may be not a successful process. Therefore, the ordering of the weights of d_w may not be correct and `selection-basic` may not be effective in the noisy case.

The shortcomings of both `mutation-basic` and `selection-basic` impair the ability of `recombination-basic`. On the other hand, `recombination-basic` suffers from the fact that there is no a descent condition, like the line search condition, to check whether a reduction of the inexact function value is found or not.

Algorithm 1 `MAESB` (`MAESF` and `MAESL`), *basic matrix adaptation strategy for unconstrained NDFO*

Tuning parameters

Given the tuning parameter `ver` (`MAESF`: fast, `MAESL`: limited), `MAESB` accordingly chooses:
$\mu > 0$ (number of sample points),
$w \in \mathbb{R}^{\mu}$ (recombination weights),
$\mu_w > 0$ (the variance effective selection mass for the mean),
$P_{\sigma}^{0} \in \mathbb{R}^{n}$ (initial evolution path),
$0 < c_{\sigma} \leq 1$ (learning rate for the cumulation for the step size control),
$d_{\sigma}, e_{\sigma} \in \mathbb{R}$ (parameters for updating σ_t),
if `ver` is `MAESFL` **then**
  $0 < c_{\mu} \leq 1$ (learning rate for updating the μ-rank update),
  $c_1 \leq 1 - c_{\mu}$ (learning rate for updating the one-rank update),
else
  $m_{\max} > 0$ (maximum columns in the matrix $P_m \in \mathbb{R}^{n \times m_{\max}}$),
  $c_d = (\, c_{d,1} \quad \cdots \quad c_{d,m_{\max}} \,)$ (learning rate for updating the search distribution)
  whose components satisfy $0 < c_{d,i} \leq 1$ for $i = 1, 2, \cdots, m_{\max}$.
  $c_m = (\, c_{m,1} \quad \cdots \quad c_{m,m_{\max}} \,)$ (learning rate for updating the one-rank update)
  whose components satisfy $c_{m,i} \leq 1$ for $i = 1, 2, \cdots, m_{\max}$.
end if

Initialization

for $i = 1, \cdots, m_{\max}$ **do**
  $\overline{c}_{m,i} = \sqrt{c_{m,i}(2 - c_{m,i})\mu_w}$; ▷ normalization constant
  $(P_m^0)_{:i} = O_{1 \times n}$; ▷ the ith column of the initial evaluation path matrix
end for ▷ $O_{1 \times n}$ is a zero vector and $A_{:i}$ denotes the ith column of the matrix A
initialize:
`nf` $= 0$ (number of function evaluations);
$\overline{c}_{\sigma} = \sqrt{c_{\sigma}(2 - c_{\sigma})\mu_w}$ (normalization constant);
$y^0 \in \mathbb{R}^n$ (initial mean value of the search distribution);
$\sigma_0 > 0$ (initial overall standard deviation/initial step size);

Main loop

10: **for** $t = 1, 2, \cdots$ **do**

Phases I and II (`getMS-basic` – `mutation-basic` and `selection-basic`):
computing the μ search distribution
computing the μ candidate points
computing the weighted average of the μ search distribution
computing the weighted average of the μ covariance search directions

11: perform $[d_w, z_w, \mathtt{zd}_w, \mathtt{nf}] = \mathtt{getMS\text{-}basic}(c_d, P_m^{t-1}, M^{t-1}, \mu, t, m_{\max}, y^{t-1}, \sigma_{t-1}, w, \mathtt{ver}, \mathtt{nf}, \mathtt{nfmax})$;
12: **if** $\mathtt{nf} \geq \mathtt{nfmax}$, **then**, `MAESB` terminates; **end if** ▷ stopping test

Phase III (`recombination-basic`):
updating information
computing new standard deviation and mean of distribution)

13: perform $[M^t, P_m^t, P_\sigma^t] = \mathtt{updateInfo}(M^{t-1}, P_m^{t-1}, c_m, \overline{c}_m, c_1, c_\mu, \mu, z_w, \mathtt{zd}_w, c_\sigma, \overline{c}_\sigma, d_\sigma, e_\sigma, P_\sigma^{t-1})$;
14: **if** `ver` is `MAESL` **then** ▷ updating standard deviation of distribution σ_t
15: compute $\sigma_t = \sigma_{t-1} \exp(\frac{1}{2} c_\sigma (\|P_\sigma^{t-1}\|^2/n - 1))$; ▷ limited memory version
16: **else**
17: compute $\sigma_t = \sigma_{t-1} \exp((c_\sigma/d_\sigma)(\|P_\sigma^{t-1}\|/e_\sigma - 1))$; ▷ fast version
18: **end if**
19: compute $y^{t+1} = y^t + \sigma_t d_w$; $\tilde{f}^{t+1} = \tilde{f}(y^{t+1})$; $\mathtt{nf} = \mathtt{nf} + 1$; ▷ updating mean
20: **if** $\mathtt{nf} \geq \mathtt{nfmax}$, **then**, `MAESB` terminates; **end if** ▷ stopping test
21: **end for**

End of the main loop

continued

1.2.2 `getMS-basic`, basic mutation and selection

`getMS-basic` is derived from mutation (`M`) and selection (`S`), respectively. It performs the basic mutation and the basic selection. It uses the tuning parameters $m_{\max}$ (maximum columns of the evaluation path matrix P_m), w (weights for selection), `ver` (type of algorithm), c_d (learning rate for updating the search distribution), and μ (number of sample points).

`getMS-basic` computes the search distribution in line 22 and then computes the covariance search direction (fast version in line 24) and (limited memory version in line 25). Next, it computes the ith candidate point and its inexact function value in line 28. Then, it checks whether the maximum number `nfmax` of function evaluations is reached or not. This process is performed μ times. Then the basic mutation ends and the basic selection starts sorting the μ inexact function values at the candidate points in ascending order and accordingly the μ search distribution $z_1, \cdots, z_\mu$,

and μ covariance search directions $d_1, \cdots, d_\mu$. Then, the weighted average of the μ-covariance search directions and the μ search distribution are calculated in lines 32 and 33, respectively. The rank-μ update term, the third term in (6) independent of its factor $c_\mu/2$, is computed in line 34 for the fast version.

As mentioned earlier, all candidate points (computed in line 28) use the fixed step size σ_t, which does not seem to be numerically good. Therefore, it is important to use unfixed step sizes in order to have an effective choice that affects recombination.

---

function $[d_w,\, z_w,\, \mathtt{zd}_w,\, \mathtt{nf}] = \mathtt{getMS\text{-}basic}(c_d,\, P_m^t,\, M^t,\, \mu,\, t,\, m_{\max},\, y^t,\, \sigma_t,\, w,\, \mathtt{ver},\, \mathtt{nf},\, \mathtt{nfmax})$

---

goal: `getMS-basic` performs mutation and basic selection

---

mutation: search distribution, candidate points, and covariance direction

22: **for** $i = 1, \cdots, \mu$ **do**, compute $z_i^t \in \mathcal{N}(0, I)$; ▷ search distribution
23: **switch** `ver`
24: **case** `MAESF`, set $d_i^t = M^t z_i^t$; ▷ fast version
25: **case** `MAESL`, set $d_i^t = z_i^t$ and $\bar{t} = \min(t, m_{\max})$; ▷ limited version
26: **for** $j = 1, \cdots, \bar{t}$, **then** $d_i^t = (1 - c_{d,j}) d_i^t + c_{d,j} (P_m^t)_{:j} ((P_m^t)_{:j})^T d$; **end for**
27: **end switch** ▷ $c_{d,j}$ denotes the jth component of c_d
28: compute $x_i^t = y^t + \sigma_t d_i^t$, $\tilde{f}_i^t = \tilde{f}(x_i^t)$, and $\mathtt{nf} = \mathtt{nf} + 1$; ▷ μ candidate points
29: **if** $\mathtt{nf} \geq \mathtt{nfmax}$, `getMS-basic` terminates; **end if** ▷ stopping test
30: **end for**

selection:
sorting inexact function values in an ascending order
computing weighted average of directions

31: sort $(\, \tilde{f}_1^t \quad \cdots \quad \tilde{f}_\mu^t \,)$ as an ascending order in the form $(\, \tilde{f}_{1:\mu}^t, \cdots, \tilde{f}_{\mu:\mu}^t \,)$ and accordingly $(\, d_1^t \quad \cdots \quad d_\mu^t \,)$ in the form $(\, d_{1:\mu}^t \quad \cdots \quad d_{\mu:\mu}^t \,)$, and $(\, z_1^t \quad \cdots \quad z_\mu^t \,)$ in the form $(\, z_{1:\mu}^t \quad \cdots \quad z_{\mu:\mu}^t \,)$;
32: compute $d_w = \sum_{i=1}^\mu w_i d_{i:\mu}^t$; ▷ μ-covariance search direction
33: compute $z_w = \sum_{i=1}^\mu w_i z_{i:\mu}^t$; ▷ μ search distribution
34: **if** `ver` is `MAESF`, **then** compute $\mathtt{zd}_w = \sum_{i=1}^\mu w_i d_{i:\mu}^t z_{i:\mu}^t$; **end if** ▷ fast version

---

1.2.3 `updateInfo`*, updating information*

`updateInfo` updates M^{t-1}, P_m^{t-1}, P_σ^{t-1}, which are used to update the step size σ_t and compute the covariance search direction. P_σ^{t-1} is updated in the same way in the both fast version (`ver` = `MAESF`) and limited memory (`ver` = `MAESL`) version but M^{t-1} is updated only in the fast version and P_m^{t-1} only in the limited memory version.

---

function $[M^t,\, P_m^t,\, P_\sigma^t] = \mathtt{updateInfo}(M^{t-1},\, P_m^{t-1},\, c_m,\, \bar{c}_m,\, c_1,\, c_\mu,\, \mu,\, z_w,\, \mathtt{zd}_w,$

c_σ, $\overline{c}_\sigma$, d_σ, e_σ, P_σ^{t-1})

---

goal: `updateInfo` updates M^{t-1}, P_m^{t-1}, P_σ^{t-1}

---

updating M^{t-1}, P_m^{t-1}, P_σ^{t-1}

compute $P_\sigma^t = (1 - c_\sigma)P_\sigma^{t-1} + \overline{c}_\sigma z_w$;
switch `ver`
case `MAESF` ▷ updating M^{t-1} for fast version
  set $c_{1,\mu} := 1 - \frac{1}{2}(c_1 + c_\mu)$;
  update $M^t = c_{1,\mu}M^{t-1} + \frac{1}{2}c_1 M^t P_\sigma^t (P_\sigma^t)^T + \frac{1}{2}c_\mu \mathtt{zd}_w$;
case `MAESL` ▷ updating P_m^{t-1} for limited version
  for $k = 1, \cdots, \mu$, update $(P_m^t)_{:k} = (1 - (c_m)_k)(P_m^{t-1})_{:k} + (\overline{c}_m)_k z_w$; **end for**
end switch

1.2.4 Improving `MAESF` and `MAESL`

To improve the basic mutation using the fixed step size σ_t, two improved versions of `MAESF` and `MAESL` can heuristically construct unfixed σ_t, hopefully leading to candidate points with better inexact function values.

To improve the basic selection, which may use an incorrect ascending order in the presence of strong noise, two improved versions of `MAESF` and `MAESL` can construct the subspace direction based on the current weighted average of the μ-covariance search directions and recursively its previous directions, increasing the chance of finding a good direction under such conditions.

`MAESF` and `MAESL` accept points regardless of whether the inexact function value is reduced or not; however, they have slightly better behaviour in the presence of strong noise, as shown in [26]. But they are not effective in the presence of low and medium noise. Applying the estimator-noise methods discussed in Section 1.1 to `MAESF` and `MAESL` is not necessarily recommended, since they do not require the approximate gradient provided by the finite difference method and the number of function evaluations is increased. Unlike the various versions of the `CMAES` solvers, derivative-free line search solvers accept points with the lowest inexact function values, which is useful if noise is not large. Otherwise, the step sizes of these methods become too small because they cannot find a decrease in the inexact function value, or too large because they increase the step sizes by extrapolation, causing a point to appear incorrectly as having the lowest inexact function value.

To improve the basic recombination, it is important to design two improved versions of `MAESF` and `MAESL` that retain their efficiency and robustness in the presence of strong noise, perhaps by enriching the line search condition with a stochastic non-monotone term. In `MAESF`, `MAESL` and the other various versions of `CMAES`, no attempt was made to use a descent condition to check whether or not a reduction in the inexact function value is found. If a stochastic non-monotone line search condition is used, two improved versions of `MAESF` and `MAESL` can perform extrapolation to

accelerate the reaching of an ε-approximate stationary point of the unconstrained NDFO problem. As explained earlier, if the step sizes are too small, there is a risk that the norm of the directions (7) and (11) become too small, leading to zero steps. In this case, `MAESF` and `MAESL` cannot approach an ε-approximate stationary point of the unconstrained NDFO problem. Therefore, adjusting the step sizes and directions in the presence of noise may lead to better results. All of the discussed `CMAES` have no plan to overcome these shortcomings.

1.3 An overview of our method

This paper designs and implements an efficient solver, called *matrix adaptation evaluation strategy* (`MADFO`) for unconstrained NDFO problems. `MADFO` provides two efficient and robust versions of `MAESF` (`FMAES` [7]) and `MAESL` (`LMMAES` [34]) with the new variants that are discussed below, called `MADFOF` and `MADFOL`, respectively.

A particular point with lowest inexact inexact function value is called the *better point*. If `MADFOF` and `MADFOL` cannot check whether a reduction of the inexact function value is found or not, then its better point can be considered as a spurious apparent better point. This case may occur in some instances. So if noise is strong, the algorithm will slowly approach an ε-approximate stationary point of the unconstrained NDFO problem.

`MADFOF` and `MADFOL` make some improvements over `MAESF` and `MAESL` to be competitive compared to the state-of-the-art derivative-free optimization solvers:
• (Improved mutation). Unfixed step sizes are generated heuristically in the mutation phase, resulting in candidate points with better inexact function values that affect the generation of the weighted average of the μ-covariance search directions and the μ distribution search, leading to an effective recombination phase.
• (Improved selection). Subspace directions based on the previous/current weighted average of the μ-covariance search directions are generated to enrich the selection phase in cases where the sorting of inexact function values at the μ-candidate points may be not correct due to high noise and such weighted average directions are not effective.
• (Improved recombination). After calculating the new step size in the recombination phase, it is checked whether the new step size is too small or not. If it is too small, it should be recomputed heuristically because the weighted average of the μ-covariance search direction is scaled by this new step size and may lead to zero step, resulting in a very slow convergence speed or even failure. Moreover, a new stochastic non-monotone line search condition is used to know whether a reduction of the inexact function value is found or not. In this way, the chance of finding the better points is significantly increased, although false apparent points can still be accepted. Then `MADFOF` and `MADFOL` can perform extrapolation to quickly find an ε-approximate stationary point of the unconstrained NDFO problem. If extrapolation is not possible, `MADFOF` and `MADFOL` generate at most five heuristic points, one of which with lowest inexact function value is accepted as the new point, hopefully close to an ε-approximate stationary point of the unconstrained NDFO problem.

Like the derivative-free line search solvers, MADFOF and MADFOL do not require the approximate gradient and does not use the approximate directional derivative in the line search condition, but adds a new stochastic non-monotone term to the line search condition and hence uses extrapolation to quickly find an ε-approximate stationary point of the unconstrained NDFO problem. If no better point can be found by extrapolation, MADFOF and MADFOL accept a point from at most five points by a new heuristic procedure unlike derivative-free line search solvers. Unlike MAESF and MAESL and other versions of CMAES, MADFOF and MADFOL enrich three phases (mutation, selection, recombination) as described above.

We compare MADFOF and MADFOL with the state-of-the-art solvers with recommendations on noise levels, and show which solvers are competitive. MADFOF and MADFOL are implemented in Matlab whose source codes are obtainable from

https://github.com/GS1400/MADFO.

2 Our new algorithm

This section introduces MADFOF and MADFOL and describes how they work. MADFOF and MADFOL use only the previous procedure updateInfo to update information needed for updating step sizes and directions. They construct new subalgorithms:
• getRMS (performing an improved mutation phase and an improved selection phase),
• subspaceDir (constructing subspace directions),
• getStepSize (updating and reconstructing σ_t heuristically),
• stochasticNM (computing stochastic non-monotone term),
• extStepDone (performing extrapolation along d_w),
• extStepTri (attempting to perform extrapolation along one of $\pm d_w$),
• heuristicPoint (finding at most five heuristic points, one of which is accepted).
heuristicPoint calls subspaceStep for finding step sizes and randsubPoint for generating a random subspace point inside a triangle whose vertices are the previous better points. Flowcharts (a)-(i) shows a simple structure of MADFOF and MADFOL and their functions.

We describe how to work MADFOF and MADFOL:
• It uses some tuning parameters discussed in lines 1-4, and initializes some necessary information needed for recombination, mutation, and selection in lines 5-9.
• The main loop then includes lines 10-23. Three phases (mutation, selection, and recombination) are performed until no ε-approximate stationary point of the unconstrained NDFO problem is found.

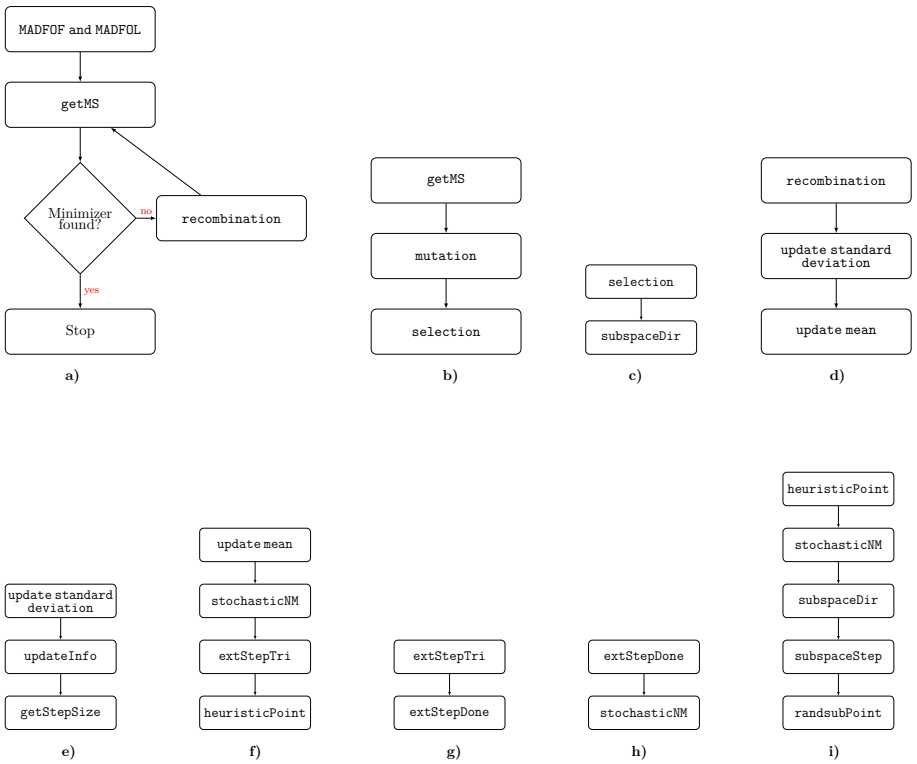


Fig. 2: Flowchart for (a) `MADFOF` and `MADFOL`, (b) `getMS`, (c) selection, (d) recombination, (e) update standard deviation, (f) update mean, (g) `extStepTri`, (h) `extStepDone`, and (i) `heuristicPoint`.

Algorithm 2 `MADFOF` and `MADFOL`, *fast and limited memory matrix adaptation evaluation strategy for unconstrained NDFO*

1: Given the tuning parameter `ver` (`MAESF`: fast, `MAESL`: limited), it is chosen:
$\mu > 0$ (number of sample points), $0 < q < 1$ (parameters for updating σ_t),
$\mu_w > 0$ (the variance effective selection mass for the mean),
$P_\sigma^0 \in \mathbb{R}^n$ (initial evolution path), $0 < \gamma < 1$ (parameter for decrease in $\tilde{f}$)
$c_\sigma > 0$ (learning rate for the cumulation for controlling σ_t), $w \in \mathbb{R}^\mu$ (weights),
$\sigma_{\max} > \sigma_{\min}$ (upper bound for σ_t), $\sigma_{\min}, \underline{\sigma} \in (0, 1)$,
$\varepsilon_a, \overline{\alpha} > 0$ (parameters for adjusting d_w), $\overline{\sigma} > 1$ (parameters for adjusting σ_t)
$\gamma_e > 1$ (parameter for expanding step size in extrapolation)
2: **if** `ver` is `MAESFL` **then**
$0 < c_\mu \leq 1$ (learning rate for updating the μ-rank update),
$c_1 \leq 1 - c_\mu$ (learning rate for updating the one-rank update),
3: **else**
$m_{\max} > 0$ (maximum columns in the matrix $P_m \in \mathbb{R}^{n \times m_{\max}}$),
$c_d = (\, c_{d,1} \quad \cdots \quad c_{d,m_{\max}} \,)$ (learning rate for updating the search distribution)
whose components satisfy $0 < c_{d,i} \leq 1$ for $i = 1, 2, \cdots, m_{\max}$.
$c_m = (\, c_{m,1} \quad \cdots \quad c_{m,m_{\max}} \,)$ (learning rate for updating the one-rank update)
whose components satisfy $c_{m,i} \leq 1$ for $i = 1, 2, \cdots, m_{\max}$.
4: **end if**

Initialization

for $i = 1, \cdots, m_{\max}$ **do**
  $\overline{c}_{m,i} = \sqrt{c_{m,i}(2 - c_{m,i})\mu_w}$; ▷ normalization constant
  $(P_m^0)_{:i} = O_{1\times n}$; ▷ the ith column of the initial evaluation path matrix
end for▷ $O_{1\times n}$ is a zero vector and $A_{:i}$ denotes the ith column of the matrix A
initialize:
`nf` $= 0$ (number of function evaluations); `ext` $= 0$; (Boolean variable for a decrease in $\tilde{f}$)
$\overline{c}_\sigma = \sqrt{c_\sigma(2 - c_\sigma)\mu_w}$ (normalization constant);
$y^0 \in \mathbb{R}^n$ (initial mean value of the search distribution);
$\sigma_0 > 0$ (initial overall standard deviation/initial step size);

Main loop

for $t = 1, 2, \cdots$ **do**
  phases I-II (`getMS` – improved mutation and improved selection):
  run $[d_w, d_w^o, z_w, \mathtt{zd}_w, X_\mu^t, F_\mu^t, \mathtt{nf}] = \mathtt{getMS}(c_d, P_m^t, M^t, \mu, t, m_{\max}, y^t, \sigma_t, d_w^o, \varepsilon_a, \overline{\alpha}, q, w, \mathtt{ver}, \mathtt{nf}, \mathtt{nfmax})$;
  if `nf` $\geq$ `nfmax`, **then** `MADFOF` and `MADFOL` terminate; **end if** ▷ stopping test
  phase III (improved recombination):
  updating the information (M^t, P_m^t, P_σ^t) by `updateInfo`
  updating and adjusting the variance σ_t of distribution by `getStepSize`
  computing the subspace direction by `subspaceDir`
  computing the trial point and its inexact function value
  computing the non-monotone term by `stochasticNM`
  performing extrapolation along one of $\pm d_w$ by `extStepTri` if possible
  accepting a heuristic point by `heuristicPoint` if no extrapolation was done
  perform $[M^t, P_m^t, P_\sigma^t] = \mathtt{updateInfo}(M^{t-1}, P_m^{t-1}, c_m, \overline{c}_m, c_1, c_\mu, \mu, z_w, \mathtt{zd}_w, c_\sigma, \overline{c}_\sigma, d_\sigma, e_\sigma, P_\sigma^{t-1})$;
  perform $\sigma_t = \mathtt{getStepSize}(\mathtt{ext}, y^t, d_w, c_\sigma, e_\sigma, d_\sigma, P_\sigma^t, \sigma_{\min}, \sigma_{\max}, \underline{\sigma}, \sigma_t, \overline{\sigma})$;
  compute $y_{\text{trial}} = y^t + \sigma_t d_w$, $\tilde{f}_{\text{trial}} = \tilde{f}(y_{\text{trial}})$, and `nf` $=$ `nf` $+ 1$;
  perform $f_{\text{nm}}^t = \mathtt{stochasticNM}(\mathtt{mem}, F^t, \tilde{f}^t, \tilde{f}_{\text{trial}})$;
  if `nf` $\geq$ `nfmax`, **then** `MADFOF` and `MADFOL` terminate; **end if** ▷ stopping test
  run $[\mathtt{ext}, y^{t+1}, \tilde{f}^{t+1}, \mathtt{nf}] = \mathtt{extStepTri}(y^t, F^t, f_{\text{nm}}^t, \tilde{f}_{\text{trial}}, \sigma_t, d_w, \gamma, \mathtt{mem}, \gamma_e, \mathtt{nf}, \mathtt{nfmax})$;
  if `nf` $\geq$ `nfmax`, **then** `MADFOF` and `MADFOL` terminate; **end if** ▷ stopping test
  run $[y^{t+1}, \tilde{f}^{t+1}, \mathtt{nf}] = \mathtt{heuristicPoint}(\mathtt{ext}, X^t, y^t, f_{\text{nm}}^t, \tilde{f}_{\text{trial}}, \mu, \mathtt{nf}, \mathtt{nfmax})$;
  if `nf` $\geq$ `nfmax`, **then** `MADFOF` and `MADFOL` terminate; **end if** ▷ stopping test
end for

End of the main loop

2.1 `subspaceDir`, subspace direction

`subspaceDir` constructs subspace directions in `getMS` (line 52) and `heuristicPoint` (line 120). The goal is to generate subspace direction spanned by the current direction and old direction. In fact, the old directions are usefull when noise is accumulated by increasing iterations and the current direction may be not effective.

In `getMS`, sorting the inexact function value at candidate points in ascending order may fail due to strong noise. In this case, the use of subspace directions can be useful, since they contain some information about the previous directions in cases where noise has not yet accumulated. But `heuristicPoint` searches along subspace directions spanned by directions moving toward the previous better points and escaping the worst point, there is a good chance to find a new better point.

Initially, `subspaceDir` computes the step size $\texttt{alp}_{\max}$ heuristically in line 26 and then its scaled version `sc` in line 27. Moreover, $\overline{\alpha} > 1$, $0 < \varepsilon_a < 1$ and $0 < \varepsilon_b < 1$ are three tuning parameters while rand $\in (0, 1)$ is a uniformly random value and t is a counter for the number of iterations of `MAESF` and `MAESL`.

---

function $d = \texttt{subspaceDir}(t,\ d,\ d_{\text{old}},\ \varepsilon_a,\ \varepsilon_b,\ \overline{\alpha})$

---

goal: `subspaceDir` constructs the subspace direction spanned based on the current direction d and the previous direction d_{old}

23: **if** $\|d_{\text{old}}\| \neq 0$ **then**
24: compute the vector $\mathbf{a} = |d|//|d_{\text{old}}|$; ▷ // denotes componentwise division
25: **if** $\mathbf{a} \neq \emptyset$ **then**
26: compute $\mathbf{a}_{\max} = \max_i \{\mathbf{a}_i \mid \mathbf{a}_i < \overline{\alpha}\}$;
27: compute $\texttt{sc} = \text{rand} * \dfrac{\varepsilon_a}{(1+t)^{\varepsilon_b}} \mathbf{a}_{\max}$ for scaling d_{old};
28: construct the subspace direction $d = \text{span}(d, d_{\text{old}}) = d + \texttt{sc} * d_{\text{old}}$;
29: **end if**
30: **end if**

In line 24 we remove from the vector $\mathbf{a}$ the NaN or infinity component (if any). In line 26, we use the tuning parameter $\overline{\alpha}$ to ignore the large component of $\mathbf{a}$ that may cause `sc` to be a large step size. In this case, the second component d_{old} of the subspace dominates the first component d of the subspace, the subspace direction is the old direction and cannot be useful since it ignores the new direction. On the other hand, the factor of $\mathbf{a}_{\max}$ (line 27) is slowly reduced and when the iterations are near an ε-approximate stationary point, this factor becomes too small and the subspace direction uses more of the first component of the subspace.

2.2 `getMS`, an improved version of `getMS-basic`

As mentioned in Section 1.3, we replace the traditional mutation discussed in - `getMS-basic` with its improved version, whose goal is to heuristically generate different step sizes (lines 37-44). This choice (line 42) not only improves the basic mutation, but also yields the μ candidate points with a better inexact function value, which accordingly sorts the covariance search directions and the covariance search directions, yielding the two effective directions, the weighted mean of the μ covariance search directions (line 49) and the weighted mean of the μ search distribution (line 50). In fact, the improvement in mutation also improves selection, resulting in an improved version `getMS` of `getMS-basic`. Another improvement in selection is to construct subspace directions (line 52) based on the weighted average of the μ-covariance search directions, which is useful when the inexact function values of the μ-candidate points appear to be missorted due to strong noise. In such a case, the subspace direction can use some previous weighted average of the μ-covariance search directions where not so much noise has yet accumulated.

`getMS` uses the tuning parameters μ (number of sample points), $\overline{\alpha} > 1$, $0 < \varepsilon_a < 1$ and $0 < \varepsilon_b < 1$ (parameters for subspace direction), $0 < q < 1$ (parameter for adjusting the step size), $\texttt{ver} \in \{\texttt{MAESF}, \texttt{MAESL}\}$ (type of algorithm), c_d (learning rate for updating the search distribution).

---

function $[d_w,\, d_w^o,\, z_w,\, \texttt{zd}_w,\, X_\mu^t,\, F_\mu^t,\, \texttt{nf}] = \texttt{getMS}(c_d,\, P_m^t,\, M^t,\, \mu,\, t,\, m_{\max},\, y^t,\, \sigma_t,\, d_w^o,\, \varepsilon_a,\, \overline{\alpha},\, q,\, w,\, \texttt{ver},\, \texttt{nf},\, \texttt{nfmax})$

---

goal: `getMS` performs an improved mutation and selection

---

improved mutation:
search distribution, heuristic unfixed step sizes, candidate points,
weighted covariance direction and search distribution

31: **for** $i = 1, \cdots, \mu$ **do**, compute $z_i^t \in \mathcal{N}(0, I)$; ▷ search distribution
32: **switch** `ver`
33: **case** `MAESF`, set $d_i^t = M^t z_i^t$; ▷ fast version
34: **case** `MAESL`, set $d_i^t = z_i^t$ and $\bar{t} = \min(t, m_{\max})$; ▷ limited memory version
35: **for** $j = 1, \cdots, \bar{t}$, **then** $d_i^t = (1 - c_{d,j}) d_i^t + c_{d,j} (P_m^t)_{:j} ((P_m^t)_{:j})^T d$; **end for**
36: **end switch** ▷ $c_{d,j}$ denotes the jth component of c_d
37: **if** $\|d^t\| \neq 0$ **then** ▷ generate unfixed σ_t for computing candidate points
38: **if** $\|y^t\| \neq 0$ **then** compute $\mathbf{a} = |y^t| // |d^t|$;
39: **else**, compute $\mathbf{a} = 1 // |d^t|$; ▷ // denotes componentwise division
40: **end if**
41: **if** $\mathbf{a} \neq \emptyset$ **then**
42: compute $\mathbf{a}_{\min} = \min_i \{\mathbf{a}_i \mid \mathbf{a}_i < 2\sigma_t\}$, $\sigma_t = \max(\sigma_t, (\sigma_t \mathbf{a}_{\min})^{1/q})$;
43: **end if**
44: **end if**
45: compute $x_i^t = y^t + \sigma_t d_i^t$, $\tilde{f}_i^t = \tilde{f}(x_i^t)$, and $\texttt{nf} = \texttt{nf} + 1$; ▷ μ candidate points
46: **if** $\texttt{nf} \geq \texttt{nfmax}$, `getMS` terminates; **end if** ▷ stopping test

47: **end for**

improved selection: sorting inexact function values in an ascending order computing weighted average of directions

48: sort $(\tilde{f}_1^t \quad \cdots \quad \tilde{f}_\mu^t)$ as an ascending order in the form $F_\mu^t = (\tilde{f}_{1:\mu}^t, \cdots, \tilde{f}_{\mu:\mu}^t)$ and accordingly $(d_1^t \quad \cdots \quad d_\mu^t)$ in the form $(d_{1:\mu}^t \quad \cdots \quad d_{\mu:\mu}^t)$, $(z_1^t \quad \cdots \quad z_\mu^t)$ in the form $(z_{1:\mu}^t \quad \cdots \quad z_{\mu:\mu}^t)$, and $(x_1^t \quad \cdots \quad x_\mu^t)$ in the form $X_\mu^t = (x_{1:\mu}^t, \cdots, x_{\mu:\mu}^t)$;

49: compute $d_w = \sum_{i=1}^{\mu} w_i d_{i:\mu}^t$; ▷ weighted covariance search direction

50: compute $z_w = \sum_{i=1}^{\mu} w_i z_{i:\mu}^t$; ▷ weighted search distribution

51: **if** `ver` is `MAESF`, **then** compute $\texttt{zd}_w = \sum_{i=1}^{\mu} w_i d_{i:\mu}^t z_{i:\mu}^t$; **end if** ▷ fast version

52: **if** $t > 1$ **then**, run $d_w =$ `subspaceDir`$(t, d_w, d_w^o, \varepsilon_a, \overline{\alpha})$; **end if**

53: set $d_w^o = d_w$; ▷ save d_w since it is an input of `subspaceDir` for the next call

---

The rank-μ update term $\texttt{zd}_w$, the third term in (6) regardless of its factor $c_\mu/2$, is computed in line 51 for the fast version. It is used by `updateInfo` to compute M^t.

When the vector $\mathbf{a}$ is computed in lines 38-39, we remove from this vector the NaN or infinity component (if any). In line 42, the components of the vector $\mathbf{a}$ that are greater than $2\sigma_t$ are removed and $\mathbf{a}_{\min}$ is calculated. The goal is to produce step sizes that are not too large such that candidate points could be far from an ε-approximate stationary point. Finally, in line 42, the step size σ_t is heuristically recomputed to be slightly larger than the current step size, which is an input of `getMS`. Indeed, the goal is to produce an unspecified step size that is not smaller than the old step size and not large. This improvement enriches both mutation and selection, resulting in the weighted average of the covariance search direction computed in line 52 being effective in the presence of noise.

2.3 `getStepSize` – updating and adjusting step sizes

If the norm of the current point y^t is not zero and σ_{t-1} is too small, `getStepSize` reconstructs σ_t heuristically in line 63, regardless of the too small σ_{t-1}. Otherwise, σ_t is computed in line 67. c_σ, e_σ, d_σ are computed once before the main algorithm starts and used to update the step size σ_t. Moreover, $\overline{\sigma} > 1$ and $0 < \underline{\sigma} < 1$ are tuning parameters for adjusting the heuristic step size σ_h in line 63 while $0 < \sigma_{\min} < 1$ and $\sigma_{\max} > 1$ are minimum and maximum value for the step size σ_t and P_σ^{t-1} is the $(t-1)$th evaluation path. The Boolean variable `heu` is used to know whether step size needs to be reconstructed heuristically or not.

---

function $\sigma_t =$ `getStepSize`$(y^t, d_w, c_\sigma, e_\sigma, d_\sigma, P_\sigma^{t-1}, \sigma_{\min}, \sigma_{\max}, \underline{\sigma}, \sigma_{t-1}, \sigma_{\exp}, \overline{\sigma})$

---

goal: `getStepSize` compute σ_t if the old σ_{t-1} is not too small; otherwise it adjusts σ_t heuristically

---

54: set `heu` $= 0$ and `tt` $= (c_\sigma/d_\sigma)(\|P_\sigma^{t-1}\|/e_\sigma - 1)$;
55: **if** not `ext` and `tt` > 0 **then** `tt` $= -$`tt`; **end if**
56: compute $\sigma_{\exp} = \exp($`tt`$)$;
57: **if** $\sigma_{t-1} \leq \sigma_{\min}$ **then** ▷ old step size is too small
58: **if** $\|y^t\| \neq 0$ **then**
59: compute $\mathbf{a} = |y^t|//|d_w|$; ▷ // denotes componentwise division
60: remove from $\mathbf{a}$ NaN or infinity component (if any);
61: **if** $\mathbf{a} \neq \emptyset$ **then** ▷ heuristic step size is made
62: set `heu` $= 1$ and reconstruct $\sigma_h = \underline{\sigma} \max_i \{\mathbf{a}_i \mid \mathbf{a}_i \leq \overline{\sigma}\}$;
63: restrict $\sigma_t = \min(\sigma_{\max}, \sigma_h \sigma_{\exp})$;
64: **end if**
65: **end if**
66: **end if**
67: **if** `heu` is false, **then** compute $\sigma_t = \min(\sigma_{\max}, \sigma_{t-1}\sigma_{\exp})$; **end if**

Since decreasing the step size σ_t is preferable to increasing it if no decrease in $\tilde{f}$ is found, we replace `tt` (computed in line 56) with $-$`tt` if no decrease in $\tilde{f}$ was found in the previous iteration of the main algorithm (`ext` is false). In line 62, the tuning parameter $\overline{\sigma}$ is used as an upper bound on the components of the vector $\mathbf{a}$ to avoid generating a large step size that may lead to point far from an ε-approximate stationary point, which can lead to slow convergence or even failure. The heuristic step size σ_h is calculated in line 62 and σ_t is then recalculated in line 63. In practice, the heuristic step size can be computed if the old step size σ_{t-1} is too small and the vector $\mathbf{a}$ has at least one nonzero real component that is less than or equal to $\overline{\sigma}$. Otherwise, the traditional formula is used to update σ_t in line 67.

2.4 `stochasticNM` – stochastic non-monotone term

The goal of `stochasticNM` is to help `extStepTri` to distinguish better points from spurious seemingly points. `stochasticNM` generates a sequence of inexact function values which are slightly stronger than the inexact function values at the previous points in stochastic and heuristic ways.

`mem` denotes the number of points whose inexact function values are used to construct the non-monotone term. `stochasticNM` first selects a sample of inexact function values at `mem` points. Then, it computes the minimum, maximum, and median of these inexact function values in line 69, and accordingly computes two adaptive parameters η_1 and η_2 in lines 70 and 71. Then, the parameter η is calculated heuristically in lines 72-77, depending on η_1 and η_2, where rand $\in (0, 1)$. Finally, our stochastic non-monotone term $f_{\rm nm}$ is computed as a convex combination of two pairs $(f_{\rm median}, f_{\max})$ or $(f_{\min}, f_{\rm median})$ with respect to the status of the inexact function value $f_{\rm trial}$ at the trial point $x_{\rm trial}$ compared to three values $f_{\min}$, $f_{\rm median}$, and $f_{\max}$:

• If $\tilde{f}_{\rm trial} \geq f_{\max}$, $f_{\rm nm}$ (line 78) is stronger than $f_{\rm mean}$ and slightly weaker than $f_{\max}$.

• Otherwise, if $\tilde{f}_{\rm trial} \geq f_{\rm median}$, then $f_{\rm nm}$ (line 79) is slightly stronger than $f_{\rm median}$ and weaker than $f_{\max}$.

• Otherwise, if $\tilde{f}_{\text{trial}} \geq f_{\min}$, then f_{nm} (line 80) is stronger than $f_{\min}$ and slightly weaker than f_{median}.
• Otherwise, f_{nm} (line 81) is slightly stronger than $f_{\min}$ and weaker than f_{median}.

From Figure 3, we conclude that f_{nm} generates values that are near the exact and noisy function values.

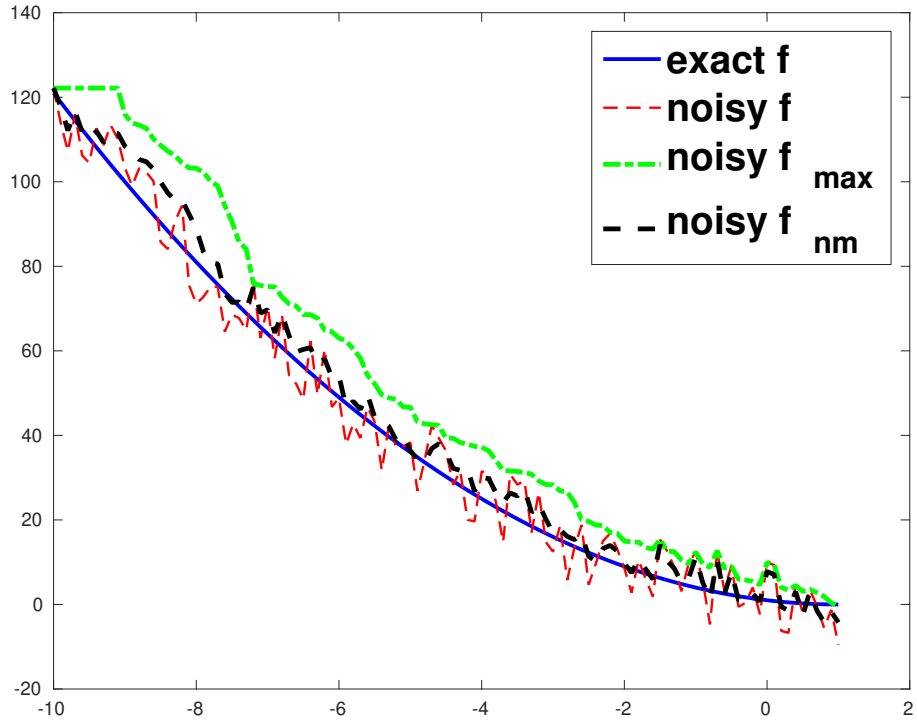


Fig. 3: The plot of exact/noisy function values, the noisy traditional non-monotone term ($f_{\max}$), and the new noisy non-monotone term f_{nm} of the one-dimensional objective function $f(\alpha) = (\alpha - 1)^2$ for $\alpha \in [-10, 1]$. The absolute uniform noise with noise level 10 was used.

function $f_{\text{nm}} = \texttt{stochasticNM}(\texttt{mem}, F^t, \tilde{f}^t, \tilde{f}_{\text{trial}})$

goal: `stochasticNM` computes a stochastic non-monotone term

choose randomly a subset I of $\{1, 2, \cdots, n\}$ with `mem` members and set $F^t = F^t_I$;
compute $f_{\max} = \max(F^t)$, $f_{\min} = \min(\tilde{f}^t, \min(F^t))$, and $f_{\text{median}} = \text{median}(F^t)$;
compute $\eta_1 = (f_{\text{median}} - f_{\min})/(f_{\max} - f_{\min})$;
compute $\eta_2 = (f_{\max} - f_{\text{median}})/(f_{\max} - f_{\min})$;
if $\eta_1 \neq 0$ and $\eta_2 \neq 0$ **then** set $\eta = \min(\eta_1, \eta_2)$;
else if η_1 is nonzero **then** set $\eta = \eta_1$;
else if η_2 is nonzero **then** set $\eta = \eta_2$;
else, set $\eta = \text{rand}$;
end if
choose $\eta = \eta/(\text{rand} + 2)$; ▷ reducing η
if $\tilde{f}_{\text{trial}} \geq f_{\max}$ **then** compute $f_{\text{nm}} = (1 - \eta)f_{\max} + \eta f_{\text{median}}$;
else if $\tilde{f}_{\text{trial}} \geq f_{\text{median}}$ **then** compute $f_{\text{nm}} = (1 - \eta)f_{\text{median}} + \eta f_{\max}$;
else if $\tilde{f}_{\text{trial}} \geq f_{\min}$ **then** compute $f_{\text{nm}} = (1 - \eta)f_{\text{median}} + \eta f_{\min}$;
else, compute $f_{\text{nm}} = (1 - \eta)f_{\min} + \eta f_{\text{median}}$;
end if

2.5 `extStepTri` and `extStepDone` – extrapolation

As mentioned earlier, the goal of extrapolation is to reach an ε-approximate stationary point of the unconstrained NDFO problem quickly. This is acceptable in the noiseless case or in the presence of rounding errors. But in the presence of high noise, extrapolation may produce points that are far from an ε-approximate stationary point of the unconstrained NDFO problem because it cannot detect whether or not a reduction in the inexact function value has been found. To overcome this problem, `stochasticNM` is added to the line search condition so that the extrapolation hopefully accepts points with lowest inexact function values.

`extStepTri` calls `extStepDone` (line 85 or line 91) to perform extrapolation along one of $\pm d_w$, while f_{nm} is computed by `stochasticNM`. The tuning parameters $0 < \gamma < 1$ (parameter for the line search condition), $\gamma_e > 1$ (parameter for expanding the step size) and `mem` (memory for non-monotone term) are used. The Boolean variable `ext` is used to know whether `extStepDone` is performed by `extStepTri` to find reduction of $\tilde{f}$ or not.

function $[\texttt{ext}, y^{t+1}, \tilde{f}^{t+1}, \texttt{nf}] = \texttt{extStepTri}(y^t, F^t, f^t_{\rm nm}, \tilde{f}_{\rm trial}, \sigma_t, d_w, \gamma, \texttt{mem}, \gamma_e, \texttt{nf}, \texttt{nfmax})$

goal: `extStepTri` performs hopefully extrapolation along one of $\pm d_w$

83: $\texttt{ext} = 0$; ▷ Boolean variable for checking reduction of $\tilde{f}$
84: **if** $f^t_{\rm nm} > \tilde{f}_{\rm trial} + \gamma\sigma_t^2$ **then** ▷ reduction of $\tilde{f}$ is found
85: $[y^{t+1}, \tilde{f}^{t+1}, \texttt{nf}] = \texttt{extStepDone}(y^t, F^t, f^t_{\rm nm}, \tilde{f}_{\rm trial}, \sigma_t, d_w, \gamma, \texttt{mem}, \gamma_e, \texttt{nf}, \texttt{nfmax})$;
86: set $\texttt{ext} = 1$;
87: **end if**
88: **if** not `ext` **then**
89: set $d_w = -d_w$ and compute $y_{\rm trial} = y^t + \sigma^t d_w$ and $f_{\rm trial} = f(y_{\rm trial})$;
90: **if** $f^t_{\rm nm} > \tilde{f}_{\rm trial} + \gamma\sigma_t^2$ **then** ▷ reduction of $\tilde{f}$ is found
91: $[y^{t+1}, \tilde{f}^{t+1}, \texttt{nf}] = \texttt{extStepDone}(y^t, F^t, f^t_{\rm nm}, \tilde{f}_{\rm trial}, \sigma_t, d_w, \gamma, \texttt{mem}, \gamma_e, \texttt{nf}, \texttt{nfmax})$;
92: set $\texttt{ext} = 1$;
93: **end if**
94: **end if**

function $[y^{t+1}, \tilde{f}^{t+1}, \texttt{nf}] = \texttt{extStepDone}(y^t, F^t, f^t_{\rm nm}, \tilde{f}_{\rm trial}, \sigma_t, d_w, \gamma, \texttt{mem}, \gamma_e, \texttt{nf}, \texttt{nfmax})$

goal: `extStepDone` performs extrapolation along d_w

95: set $\overline{F} = \tilde{f}_{\rm trial}$ and $\alpha = \sigma_t$;
96: **while** 1 **do** ▷ perform extrapolation along d_w
97: expand $\sigma_t = \gamma_e\sigma_t$ and set $\alpha = (\,\alpha \quad \sigma_t\,)$;
98: compute $y_{\rm trial} = y^t + \sigma_t d_w$, $\tilde{f}_{\rm trial} = \tilde{f}(y_{\rm trial})$, and $\texttt{nf} = \texttt{nf} + 1$;
99: **if** $\texttt{nf} \geq \texttt{nfmax}$, **then return**; **end if** ▷ stopping test
100: set $\overline{F} = (\,\overline{F} \quad \tilde{f}_{\rm trial}\,)$ and $F^t = (\,F^t \quad \overline{F}\,)$;
101: perform $f_{\rm nm} = \texttt{stochasticNM}(\texttt{mem}, F^t, \tilde{f}^t, \tilde{f}_{\rm trial})$; ▷ non-monotone trem
102: **if** $f_{\rm nm} \leq \tilde{f}_{\rm trial} + \gamma\sigma_t^2$ **then**, **break**; ▷ extrapolation stops
103: **end if**
104: **end while**
105: find $i_b = \text{argmin}(\overline{F})$ and set $\sigma_t = \alpha_{i_b}$, $y^{t+1} = y^t + \sigma_t d_w$, $\tilde{f}^{t+1} = \tilde{f}(y^{t+1}) = \overline{F}_{i_b}$.

The condition used in lines 84 and 90 is our new non-monotone line search condition that does not include a directional derivative; instead, the term $\gamma\sigma_t^2$ is used as a forcing function. During the extrapolation (lines 96-104), many better points can be found. Here, a point with the lowest inexact function value is accepted as the new point (line 105). As can be seen from lines 100-101, we use the vector F^t of function values (as an input) and the vector $\overline{F}$ of function values stored during the extrapolation to calculate the non-monotone term.

`MADFOF` and `MADFOL` accept in each iteration a point not far from an ε-approximate stationary point of the unconstrained NDFO problem by `heuristicPoint` when `extStepTri` cannot accept a new better point due to strong noise.

In the first through third iterations, `heuristicPoint` stores three points. In subsequent iterations, point with largest inexact function value is then replaced by a new better point; lines 118-122. If the condition

$$\tilde{f}(y_{\text{trial}}) < f_{\text{nm}}^t \tag{12}$$

holds, the new point is accepted. Otherwise, at most five points are heuristically generated, one of which is accepted as the new point. Let us describe how to select these points. For all trial points, `stochasticNM` computes f_{nm}^t (line 144) which is used in (12) (line 124).

Given $i, j, k \leq t$, `heuristicPoint` sorts the stored three points

$$y_1 = y_{\text{best}} = y^i, \quad y_2 = y^j, \quad y_3 = y_{\text{worst}} = y^k,$$

in the ascending order of their inexact function values ($\tilde{f}(y_1) < \tilde{f}(y_2) < \tilde{f}(y_3)$):
Case 1. The centre $y_{2,3}$ of y_2 and y_3 is calculated. Then the direction $d_1 = y_1 - y_{2,3}$ and the new trial point $y_{\text{trial}} = y_{2,3} + \alpha_1 d_1$ are computed. Here the step size

$$\alpha_1 = \texttt{subspaceStep}(y_{2,3},\, t,\, d_1,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha})$$

is computed, where $0 < \varepsilon_a < 1$, $0 < \varepsilon_b < 1$, and $1 < \overline{\alpha} < \infty$ are tuning parameters.

---

function $\alpha = \texttt{subspaceStep}(x,\, t,\, d,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha})$

---

goal: `subspaceStep` find step sizes greater than one

106: **if** $\|x\| \neq 0$ **then**, compute $\mathbf{a} = |x|//|d|$;
107: **else**, compute $\mathbf{a} = 1//|d|$; ▷ // denotes componentwise division
108: **end if**
109: remove NaN or infinity components (if any) from $\mathbf{a}$;
110: **if** $\mathbf{a} \neq \emptyset$ **then**, ▷ the step size α is computed heuristically
111: compute $\alpha = \max\left(1 + \text{rand},\, \dfrac{\varepsilon_a * \text{rand}}{(1+t)^{\varepsilon_b}} \max_i \{\mathbf{a}_i \mid \mathbf{a}_i \leq \overline{\alpha}\}\right)$;
112: **else**, set $\alpha = (1 + \text{rand})$;
113: **end if**

In line 111, $\overline{\alpha}$ is used as an upper bound on the components of the vector $\mathbf{a}$ to remove components that are large, while the factor $(\varepsilon_a * \text{rand})/(1+t)^{\varepsilon_b}$ is used to slowly decrease the step size α, which must be greater than one. The guarantee of finding such a step size is that there is at least one nonzero real component of the vector $\mathbf{a}$ less than or equal to $\overline{\alpha}$; otherwise α is calculated as a random value greater than or equal to one in line 112.

The idea is to escape the midpoint $y_{2,3}$ of y_2 and y_3 and to search along d_1, because

$$\tilde{f}(y_1) < \min(\tilde{f}(y_2), \tilde{f}(y_3))$$

and generate y_{trial} around y_1 hoping to find a new better point. If the condition (12) holds, y_{trial} is accepted as the new point y^{t+1} in the $(t+1)$th iteration. Here y^t is the old accepted point in the tth iteration. This case is shown in Figure 4(a). Otherwise, the next case is tried.

Case 2. The centre $y_{1,2}$ of y_1 and y_2, the direction $d_2 = y_{1,2} - y_{2,3}$, the subspace direction

$$d_2^s = \text{span}(d_2, d_1) = \texttt{subspaceDir}(t,\, d_2,\, d_1,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha})$$

and the new trial point $y_{\text{trial}} = y_{2,3} + \alpha_2 d_2^s$ are computed. Here the step size α_2 is computed by calling

$$\alpha_2 = \texttt{subspaceStep}(y_{2,3},\, t,\, d_2^s,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha}).$$

In fact, the idea is to escape the midpoint $y_{2,3}$ of y_2 and y_3 and to search along the subspace direction d_2^s spanned by the directions d_1 and d_2 hoping to find a new better point. If the condition (12) holds, x_{trial} is accepted as the new point y^{t+1}. This case is shown in Figure 4(b). Otherwise, the next case is tried.

Case 3. The centre $y_{1,3}$ of y_1 and y_3, the direction $d_3 = y_{1,3} - y_{2,3}$, the subspace direction

$$d_3^s = \text{span}(d_3, d_1) = \texttt{subspaceDir}(t,\, d_3,\, d_1,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha})$$

and the new trial point $y_{\text{trial}} = y_{2,3} + \alpha_3 d_3^s$ are computed. Here the step size

$$\alpha_3 = \texttt{subspaceStep}(y_{2,3},\, t,\, d_3^s,\, \varepsilon_a,\, \varepsilon_b,\, \overline{\alpha})$$

is computed heuristically. The idea is to escape the midpoint $y_{2,3}$ of y_2 and y_3 and to search along the subspace direction d_3^s spanned by the directions d_1 and d_3 hoping to find a new better point. If the condition (12) holds, y_{trial} is accepted as the new point y^{t+1}. This case is shown in Figure 4(c). Otherwise, the next case is tried.

Case 4. A random subspace trial point within a triangle whose vertices are

$$\{y_1, y_{1,2}, y_{1,3}\}$$

is tried by `randsubPoint` (shown in Figure 4(d)) in the hope of finding a reduction of the inexact function value. These points are sorted in the columns of the matrix $\overline{X}$ so that their inexact function values were sorted in ascending order.

---

function $y_{\text{trial}} = \texttt{randsubPoint}(\overline{X})$

---

goal: `randsubPoint` find step sizes greater than one for `subspaceDir`

114: compute $\alpha = (\ \alpha_1 \quad \alpha_2 \quad \alpha_3\ ) \in \mathcal{N}(0, I)$;
115: scale $\alpha = \alpha / \|\alpha\|$;
116: compute $y_{\text{trial}} = \sum_{i=1}^{3} \alpha_i \overline{X}_{:i}$;

Case 5. A random subspace trial point within a triangle whose vertices are

$$\{y_{1,3}, y_{1,2}, y_{2,3}\}$$

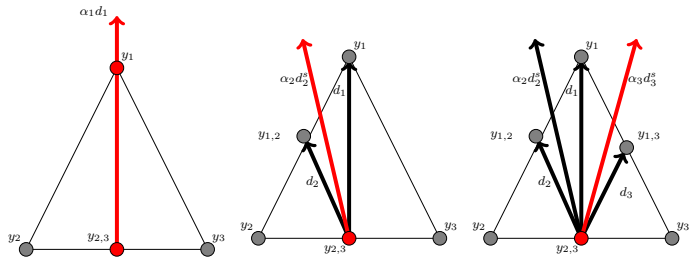
is tried by `randsubPoint` (shown in Figure 4(e)).

If five points generated by `heuristicPoint` in all cases cannot find a reduction in the inexact function value, a point with lowest inexact function value among five points is accepted as the new point

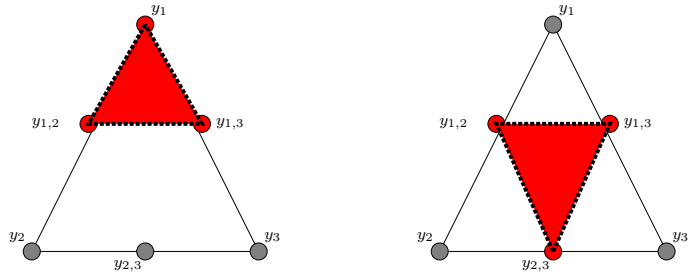
$$y^{t+1} = y_{\text{trial}} = y_{i'}, \quad i' := \underset{i=1:5}{\operatorname{argmin}}\{\tilde{f}(y_i)\},$$

which has a good chance of being not far from an ε-approximate stationary point of the unconstrained NDFO problem.

`heuristicPoint` uses the tuning parameters `mem` (memory for non-monotone term), $\overline{\alpha} > 1$, $0 < \varepsilon_a < 1$, $0 < \varepsilon_b < 1$ (parameters for heuristic step size) , $\overline{\beta}$ (parameter for step size of subspace direction). As defined earlier, the Boolean variable `ext` is used to know whether `extStepDone` is performed by `extStepTri` to find reduction of $\tilde{f}$ or not. The Boolean variable `dec` is identified whether the inexact function value at one of the five heuristic points is decreased or not.



a) Case 1 **b)** Case 2 **c)** Case 3



d) Case 4 e) Case 5

Fig. 4: $y_1 = y_{\text{best}}$, y_2, and $y_3 = y_{\text{worst}}$ are three previous points whose inexact function values sorted in the increasing order. Denote by $y_{1,2}$ the middle of points y_1 and y_2, by $y_{1,3}$ the middle of points y_1 and y_3, by $y_{2,3}$ the middle of points y_2 and y_3.

function $[y^{t+1},\ \tilde{f}^{t+1},\ \overline{X},\ \overline{F},\ \mathtt{nf}] = \mathtt{heuristicPoint}(\mathtt{ext},\ \overline{X},\ \overline{F},\ y^t,\ \tilde{f}^t,\ y_{\rm trial},\ \tilde{f}_{\rm trial},\ \mathtt{mem},\ \overline{\alpha},\ \varepsilon_a,\ \varepsilon_b,\ \overline{\beta},\ \mathtt{nf},\ \mathtt{nfmax})$

goal: `heuristicPoint` constructs at most five heuristic points, one of which is accepted as a new point

117: set $\mathtt{dec} = 0$; ▷ Boolean variable for finding decrease in $\tilde{f}$
118: **if** $t+1 \le 3$ **then**, set $\overline{X}_{:t+1} = y_{\rm trial}$ and $\overline{F}_{:t+1} = \tilde{f}_{\rm trial}$
119: **else**
120: find the index $i_w = \underset{i=1:3}{\rm argmax}\{\overline{F}_{:i}\}$, replace $\overline{X}_{:i_w} = y_{\rm trial}$ and $\overline{F}_{:i_w} = \tilde{f}_{\rm trial}$;
121: sort $\overline{F}$ in the increasing order and accordingly $\overline{X}$;
122: **end if**
123: **if** not `ext` **then**
124: **if** $\tilde{f}_{\rm trial} < f^t_{\rm nm}$ **then** ▷ no decrease in the inexact function value along $\pm d_w$
125: set $y^{t+1} = y_{\rm trial}$ and $\tilde{f}^{t+1} = \tilde{f}(y^{t+1})$;
126: **else**
127: set $x_1 = \overline{X}_{:1}$, $x_2 = \overline{X}_{:2}$, and $x_3 = \overline{X}_{:3}$;
128: compute $x_{1,2} = \frac{1}{2}(x_1 + x_2)$, $x_{1,3} = \frac{1}{2}(x_1 + x_3)$, and $x_{2,3} = \frac{1}{2}(x_2 + x_3)$;
129: **for** $j = 1, 2, 3, 4, 5$ **do**
130: **switch** j ▷ compute the search direction
131: **case** 1, compute $d_j = x_1 - x_{2,3}$;
132: **case** 2, compute $d_j = x_{1,2} - x_{2,3}$;
133: **case** 3, compute $d_j = x_{1,3} - x_{2,3}$;
134: **case** 4, compute $\overline{X} = (\ x_1 \quad x_{1,2} \quad x_{1,3}\ )$;
135: **case** 5, compute $\overline{X} = (\ x_{2,3} \quad x_{1,2} \quad x_{1,3}\ )$;
136: **end switch**
137: **if** $j \in \{2, 3\}$, call $d_j = \mathtt{subspaceDir}(t,\ d_j,\ d_{j-1},\ \varepsilon_a,\ \varepsilon_b,\ \overline{\beta})$; **end if**
138: **if** $j \in \{1, 2, 3\}$ **then**
139: call $\alpha_j = \mathtt{subspaceStep}(x_j,\ t,\ d_j,\ \varepsilon_a,\ \varepsilon_b,\ \overline{\alpha})$;
140: compute $y^j_{\rm trial} = x_j + \alpha_j d_j$;
141: **else**, perform $[y^j_{\rm trial}] = \mathtt{randsubPoint}(\overline{X})$;
142: **end if**
143: compute $\tilde{f}^j_{\rm trial} = \tilde{f}(y^j_{\rm trial})$ and set $\mathtt{nf} = \mathtt{nf} + 1$;
144: compute $f^t_{\rm nm} = \mathtt{stochasticNM}(\mathtt{mem},\ \overline{F},\ \tilde{f}^t,\ \tilde{f}^j_{\rm trial})$;
145: **if** $\mathtt{nf} \ge \mathtt{nfmax}$ **then** ▷ stopping test
146: **if** $\tilde{f}^j_{\rm trial} < f^t_{\rm nm}$, set $y^{t+1} = y^j_{\rm trial}$ and $\tilde{f}^{t+1} = \tilde{f}(y^j_{\rm trial})$; **end if**
147: **return**
148: **end if**
149: **if** $\tilde{f}^j_{\rm trial} < f^t_{\rm nm}$, set $\mathtt{dec} = 1$; **break**; **end if** ▷ decrease in $\tilde{f}$ found
150: **end for**
151: **if** not `dec`, set $j' = \underset{j=1:5}{\rm argmin}\{\tilde{f}^j_{\rm trial}\}$, $\tilde{f}^{t+1} = \tilde{f}^{j'}_{\rm trial}$, $y^{t+1} = y^{j'}_{\rm trial}$; **end if**
152: **end if**
153: **end if**

3 Numerical results

We compare `MADFOF` and `MADFOL` with the state-of-the-art solvers on the unconstrained `CUTEst` test problems from the collection of Gould et al. [15].

The convergence speed of the solver s to reach a minimum of the smooth true function f is identified by measuring the quotients

$$q_s := (f_s - f_{\rm opt})/(f_0 - f_{\rm opt}) \quad \text{for } s \in \mathcal{S}; \tag{13}$$

not available in real applications, where $\mathcal{S}$ denotes the list of compared solvers, f_s denotes the best function value found by the solver so, f_0 denotes the function value at the starting point (common for all solvers), and $f_{\rm opt}$ denotes the function value at the best known point (in most cases a global minimizer or at least a better local minimizer) found by running a sequence of gradient-based and local/global gradient free solvers; see Appendix B in [28].

we consider a problem *solved* by the solver s if $q_s \leq \varepsilon$ and neither the maximum number `nfmax` of function evaluations nor the maximum allowed time `secmax` in seconds is satisfied, and *unsolved* otherwise. ε, `secmax` and `nfmax` are chosen so that the best solver can solve at least half of the problems, unless due to high noise increasing `secmax` and `nfmax` cannot change the efficiency and robustness. The following choices were found valuable:

$$\texttt{secmax} = 360, \quad \texttt{nfmax} = 2000n + 5000, \quad \varepsilon \in \{10^{-4}, 10^{-2}\} \quad \text{if } 1 \leq n \leq 20.$$

We use absolute/relative uniform and Gaussian noises with the noise level $\omega = 10^{-k}$ for $k = -1, \cdots, 5$. Hence, for small scale problems $1 < n \leq 20$, we generate totally the 9912 test problems since $\varepsilon \in \{10^{-4}, 10^{-2}\}$ is chosen and 177 test problems exist.

Following [26,28], the starting point $y^0 := 0$ is chosen and shifted by

$$\xi_i := (-1)^{i-1} \frac{2}{2+i}, \text{ for all } i = 1, \ldots, n.$$

The reason for this choice is that there are some toy problems in the `CUTEst` library with a simple solution whose solution can be easily guessed by the solver. Indeed, we choose the initial point by $y^0 := \xi$ and the initial inexact function value $\tilde{f}_0 := \tilde{f}(y^0)$, while we compute the other inexact function values by $\tilde{f}_\ell := \tilde{f}(y^\ell + \xi)$ for all $\ell \geq 0$.

The details of all compared solvers are as follows:
• `VRBBO` is a randomized algorithm by Kimiaei and Neumaier [28]; it can be downloaded from

`https://www.mat.univie.ac.at/~neum/software/VRBBO/`.

• `VRBBON` is a randomized algorithm by Kimiaei [26]; it can be downloaded from

`https://www.mat.univie.ac.at/~kimiaei/software/VRBBON`.

• `SDBOX` – a derivative-free algorithm for bound constrained optimization problems discussed in [35], downloaded from

http://www.iasi.cnr.it/~liuzzi/DFL/index.php/list3.

• NELDER by Kelley [24], obtained from

https://ctk.math.ncsu.edu/matlab_darts.html

and NMSMAX by Higham [22], obtained from

http://www.ma.man.ac.uk/~higham/mctoolbox/

are two versions of Nelder–Mead simplex method for direct search optimization algorithms.

• UOBYQA and NEWUOA, obtained from

https://www.pdfo.net/docs.html,

are model-based solvers by Powell [40,41]. Tuning parameters are default.

• BFO, available at

https://github.com/m01marpor/BFO,

is a trainable stochastic derivative-free solver for mixed integer bound-constrained optimization by Porcelli and Toint [39]. Tuning parameters are default.

• BCDFO, obtained from Anke Troeltzsch (personal communication), is a deterministic model-based trust-region algorithm for derivative-free bound-constrained minimization by Gratton et al. [18]. Tuning parameters are default.

• LMMAES (MAESL) by Loshchilov et al. [34] and fMAES (MAESF) by Beyer [7], obtained from

https://homepages.fhv.at/hgb/downloads.html,

are two effective covariance matrix adaptation evolution strategies.

Following [7,34], we choose $w_i^0 = \ln(\mu+\frac{1}{2}) - \ln i$ and $w_i := \frac{w_i^0}{\sum_{j=1}^{\mu} w_j^0}$ for $i = 1, \cdots, \mu$, $m_{\max} = \lambda = 4 + \lfloor 3 \ln n \rfloor$, $\mu = \left\lfloor \frac{\lambda}{2} \right\rfloor$, $\mu_w := \frac{1}{\sum_{j=1}^{\mu} w_j^2}$, $c_s = \min\left(1.999, \frac{\mu_w + 2}{n + \mu_w + 5}\right)$, $\overline{c}_s = \sqrt{c_s(2-c_s)\mu_w}$, $e_s = \sqrt{n}(1 - 1/(4n) - 1/(21n^2))$, $c_1 = 2/((n+1.3)^2 + \mu_w)$,

$$c_\mu = \min\left\{1 - c_1, \frac{2\left(\mu_w - 2 + \frac{1}{\mu_w}\right)}{(n+2)^2 + \mu_w}\right\}, \quad d_s = 1 + c_s + 2\max\left\{0, \sqrt{\frac{\mu_w - 1}{n+1}} - 1\right\}.$$

Moreover, for $i = 1, \cdots, m_{\max}$, we choose

$$s_i = 1, \quad cd_i = \frac{1.5^{i-1}}{n}, \quad c_{m,i} = \min\left(1.999, \frac{\lambda}{n4^{i-1}}\right), \quad \overline{c}_{m,i} = \sqrt{c_{m,i}(2 - c_{m,i})\mu_w}.$$

In contrast to [34], we added the upper bound 1.999 on both c_s and cp since both $\overline{c}_s$ and $\overline{cp}$ should be real value and vector, respectively. For LMMAES, we made this

modification because in the original code these values were complex sometimes. Other tuning parameters for MADFO are $\gamma = 10^{-12}$, $m_{\max} = 10$, $\sigma_0 = 1$, $\gamma_e = 2$, $\sigma_{\min} = 10^{-12}$, $\sigma_{\max} = 10^4$, $\overline{\sigma} = \overline{\alpha} = 10^{10}$, $\overline{\sigma} = 0.99$, $\varepsilon_a = 0.01$, $\varepsilon_b = 0.85$, $q = 5$.

To identify which solver is *robust* and *efficient*, we use the data profile of Moré and Wild [37] and the performance profile of Dolan and Moré [13]. We denote the list of compared solvers by $\mathcal{S}$ and the list of problems by $\mathcal{P}$. The data profile of the solver s, the fraction of problems that the solver s can solve with κ groups of $n_p + 1$ function evaluations, is

$$\delta_s(\kappa) := \frac{1}{|\mathcal{P}|}\left|\left\{p \in \mathcal{P} \;\middle|\; cr_{p,s} := \frac{c_{p,s}}{n_p + 1} \leq \kappa\right\}\right|. \tag{14}$$

Here n_p is the dimension of the problem p, $c_{p,s}$ is the *cost measure* of the solver s to solve the problem p and $cr_{p,s}$ is the *cost ratio* of the solver s to solve the problem p. The performance profile of the solver s

$$\rho_s(\tau) := \frac{1}{|\mathcal{P}|}\left|\left\{p \in \mathcal{P} \;\middle|\; pr_{p,s} := \frac{c_{p,s}}{\min(c_{p,\overline{s}} \mid \overline{s} \in S)} \leq \tau\right\}\right|. \tag{15}$$

is the fraction of problems that the performance ratio $pr_{p,s}$ is at most τ. In particular, the fraction of problems that the solver s wins compared to the other solvers is $\rho_s(1)$ and the fraction of problems for sufficiently large τ (or κ) that the solver s can solve is $\rho_s(\tau)$ (or $\delta_s(\kappa)$).

The data and performance profiles are based on the problem scales, but not on the noise levels. The other two plots are based on the noise levels. These four plots are used to identify the behaviour of the compared solvers with respect to problem scales and noise levels. Additionally, we plot the number of problems solved and the efficiency versus the noise level to show the behavior of the compared solvers with respect to the noise levels.

We summarize our results in Table 1 and Figures 5 and 6. For more results with respect to the type of noise, see Section 4 (Figures 7-10). As mentioned earlier, MADFOF and MADFOL are the improved versions of fMAES and LMMAES. It is shown that our new techniques added to MADFOF and MADFOL make them more robust compared to fMAES, LMMAES, and other model-based, line search-based, and direct search solvers.

From Table 1, we conclude that MADFOF is more robust than others, while fMAES and MADFOL are the second and third robust solvers, respectively. In fact, MADFOF can solve 368 more problems than fMAES, while MADFOL can solve 2041 more problems than LMMAES. Also, MADFOF can solve 1357 more problems than UOBYQA and 1615 more problems than VRBBON.

The first row of Figure 5 contains two Box plots in terms of the number of function evaluations, the second row contains performance profiles, while the third row contains data profiles in terms of the number of function evaluations, and the fourth row of it includes Pie charts in terms of the number of solved problems. In all rows, the left graph is for $\varepsilon = 10^{-2}$ and the right graph is for $\varepsilon = 10^{-4}$. Also, Figure 6 is Morales profile in terms of the number of function evaluations of the two most robust solvers, whose goal is to show which has the lowest function evaluation cost.

Figure 5 reflexes the same results as the Table 1, except that performance profiles show that NEWUOA and UOBYQA are the first and second efficient solvers (lowest relative cost of function evaluations). From Figure 6, we conclude that fMAES is slightly efficient than MADFOF in some problems.

Table 1: The number of solved problems by all compared solvers for $\varepsilon \in \{10^{-4}, 10^{-2}\}$ and $\omega = 10^{-i}$ with $i = -1, \cdots, 5$.

	absolute/relative uniform and Gaussian noises									
	solvers									
	MADFOF	fMAES	MADFOL	UOBYQA	VRBBON	VRBBO	NMSMAX	NEWUOA	BFO	LMMAES
ε	npr: number of solved problems									
10^{-4}	3056	2832	2526	2334	2144	1911	1932	1874	1583	1519
10^{-2}	3858	3714	3750	3223	3135	2935	3044	2972	2737	2716
$\sum$	6914	6546	6276	5557	5299	4986	4976	4846	4320	4235
npr (%)	69.7538	66.0411	63.3171	56.0633	53.4604	50.3027	50.2017	48.8902	43.5835	42.7259
	absolute uniform noise									
	solvers									
	MADFOF	fMAES	MADFOL	UOBYQA	VRBBON	NMSMAX	VRBBO	NEWUOA	BFO	LMMAES
ε	npr: number of solved problems									
10^{-4}	822	728	663	604	592	483	494	464	380	390
10^{-2}	1050	1009	1011	890	883	835	807	824	753	724
$\sum$	1872	1737	1674	1494	1475	1318	1301	1288	1133	1114
npr (%)	75.5447	70.0969	67.5544	60.2906	59.5238	53.1881	52.5020	51.9774	45.7224	44.9556
	absolute Gaussian noise									
	solvers									
	MADFOF	fMAES	MADFOL	UOBYQA	VRBBON	NMSMAX	NEWUOA	VRBBO	LMMAES	BFO
ε	npr: number of solved problems									
10^{-4}	769	703	628	575	498	469	465	445	386	347
10^{-2}	1018	976	998	861	806	814	769	811	701	704
$\sum$	1787	1679	1626	1436	1304	1283	1234	1256	1087	1051
npr (%)	72.1146	67.7563	65.6174	57.9500	52.6231	51.7756	49.7982	50.6860	43.8660	42.4132
	relative uniform noise									
	solvers									
	MADFOF	fMAES	MADFOL	UOBYQA	VRBBON	NMSMAX	NEWUOA	VRBBO	BFO	LMMAES
ε	npr: number of solved problems									
10^{-4}	757	719	649	598	584	511	506	498	445	367
10^{-2}	929	882	901	763	765	715	696	694	367	445
$\sum$	1686	1601	1550	1361	1349	1226	1202	1192	812	812
npr (%)	68.0387	64.6086	62.5504	54.9233	54.4391	49.4754	48.5069	48.1033	32.7684	32.7684
	relative Gaussian noise									
	solvers									
	MADFOF	fMAES	MADFOL	UOBYQA	VRBBON	NMSMAX	NEWUOA	VRBBO	BFO	LMMAES
ε	npr: number of solved problems									
10^{-4}	708	682	586	557	470	469	454	459	411	376
10^{-2}	861	847	840	709	681	680	665	641	609	616
$\sum$	1569	1529	1426	1266	1151	1149	1119	1100	1020	992
npr (%)	63.3171	61.7030	57.5464	51.0896	46.4487	46.3680	45.1574	44.3906	41.1622	40.0323

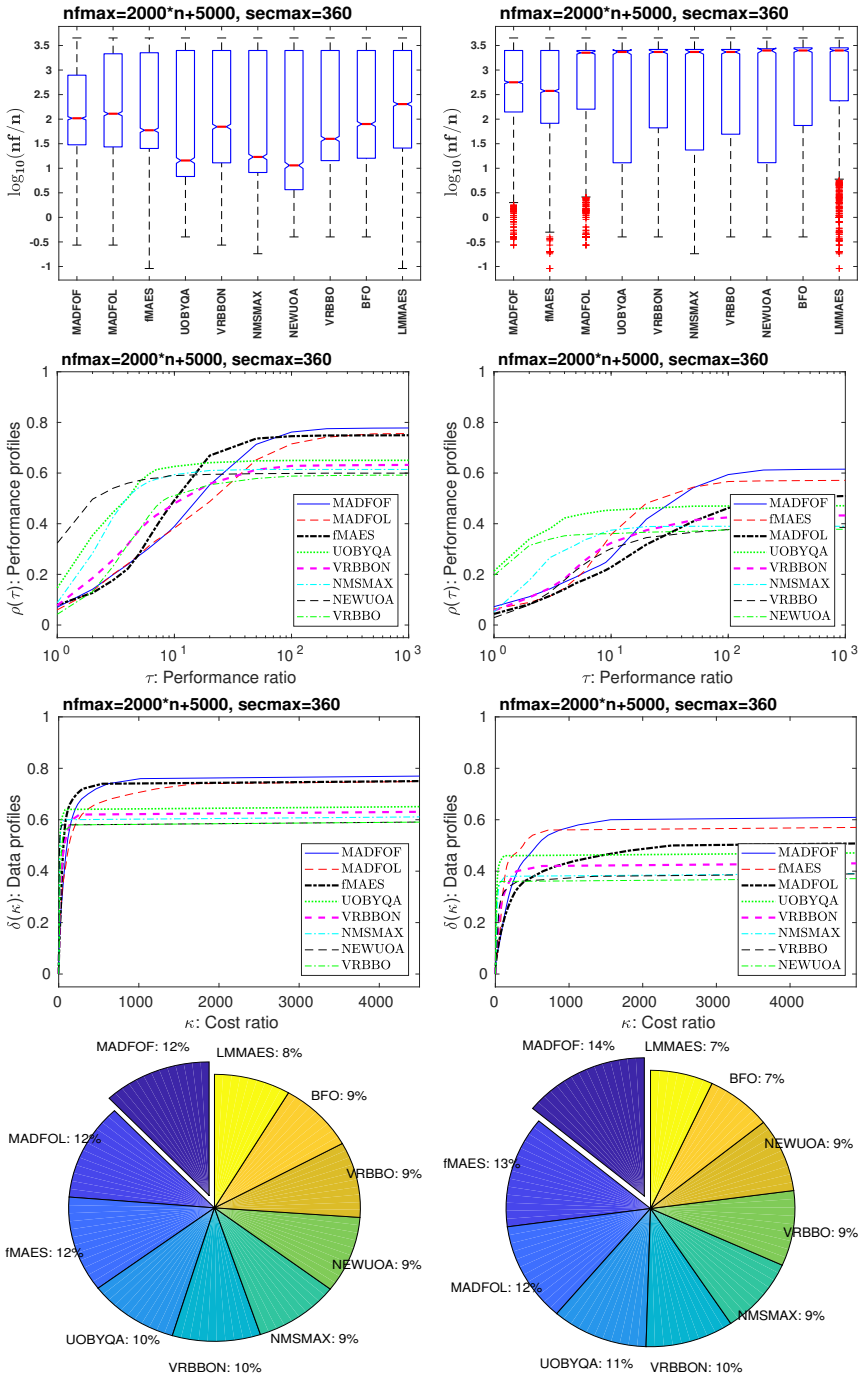


Fig. 5: For the absolute/relative uniform and Gaussian noises with noise levels $\omega = 10^{-k}$ for $k = -1, \cdots, 5$ and small dimensions $1 < n \leq 20$. Left side $\epsilon = 10^{-2}$ and right side $\epsilon = 10^{-4}$. Box plots in terms of the number of solved problems, data profile $\delta(\kappa)$ in dependence of a bound κ on the cost ratio, see (14), performance profile $\rho(\tau)$ in dependence of a bound τ on the performance ratio, see (15), and Pie charts in terms of the number of solved problems. Problems solved by no solver are ignored.

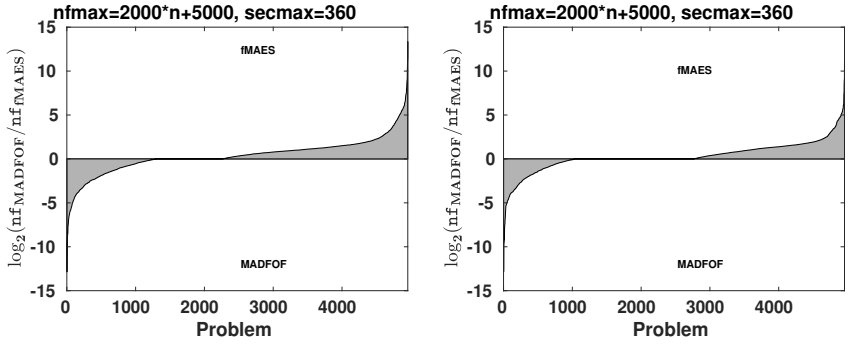


Fig. 6: For the absolute/relative uniform and Gaussian noises with noise levels $\omega = 10^{-k}$ for $k = -1, \cdots, 5$ and small dimensions $1 < n \leq 20$. Left side $\epsilon = 10^{-2}$ and right side $\epsilon = 10^{-4}$. Box plots in terms of the number of solved problems, data profile $\delta(\kappa)$ in dependence of a bound κ on the cost ratio, see (14), performance profile $\rho(\tau)$ in dependence of a bound τ on the performance ratio, see (15), and Pie charts in terms of the number of solved problems. Problems solved by no solver are ignored.

4 Tools for MADFO

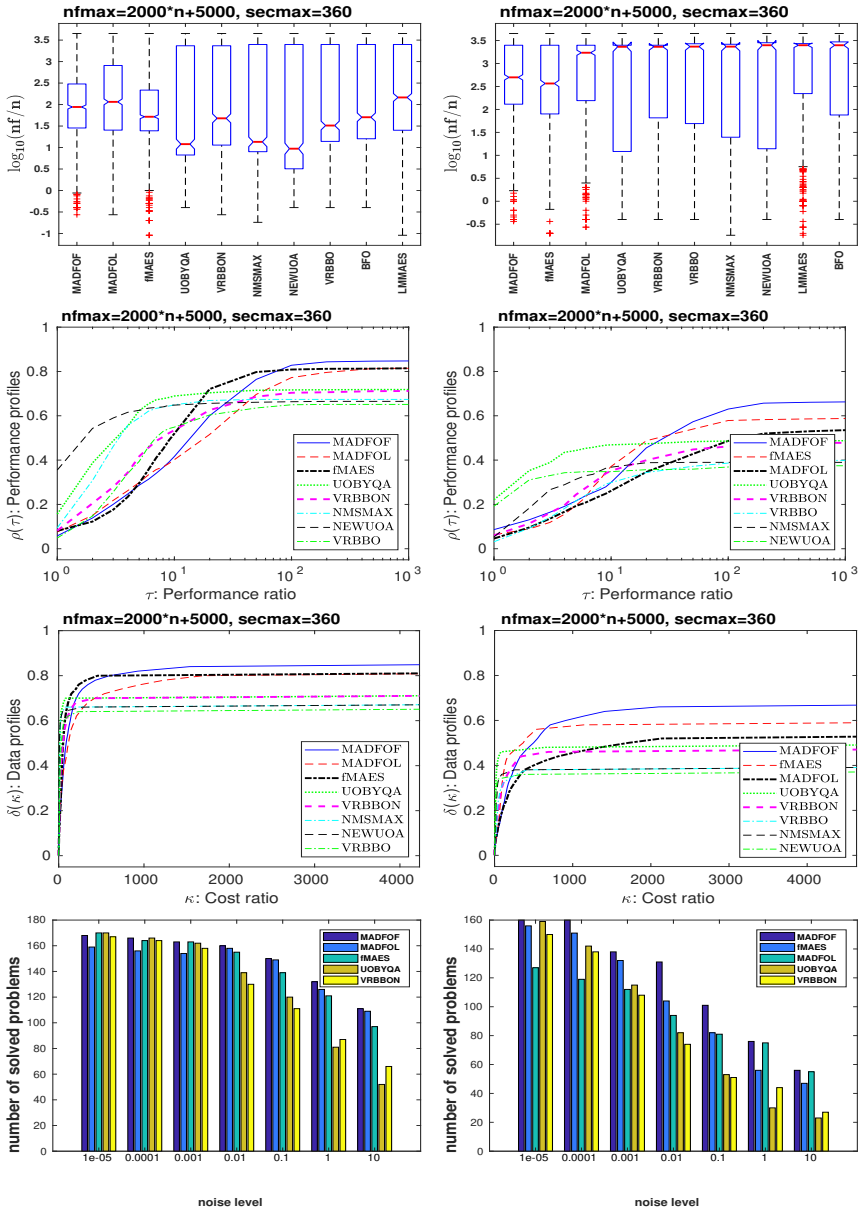


Fig. 7: For the absolute uniform noise. Other details as Figure 5.

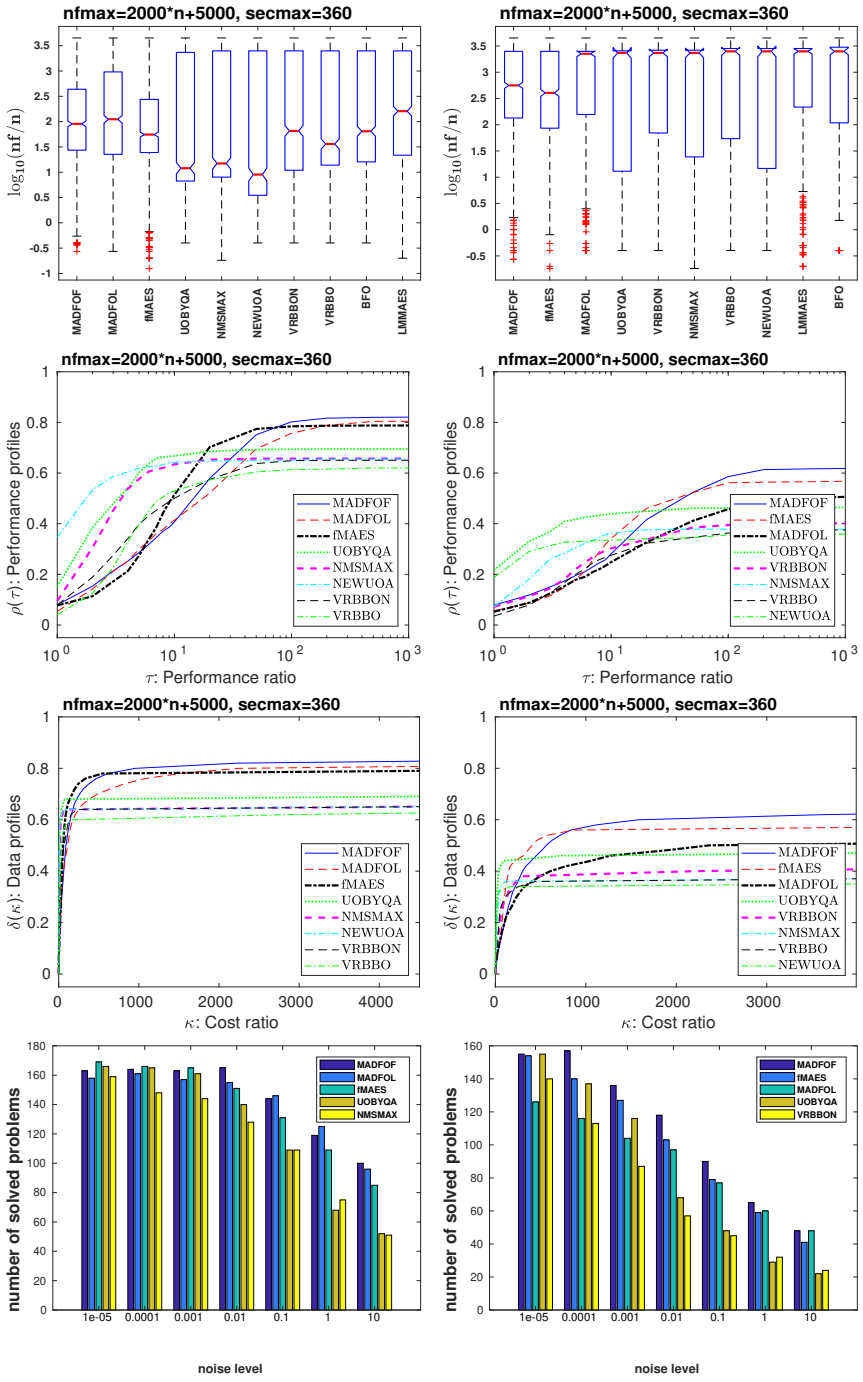


Fig. 8: For the absolute Gaussian noise. Other details as Figure 5.

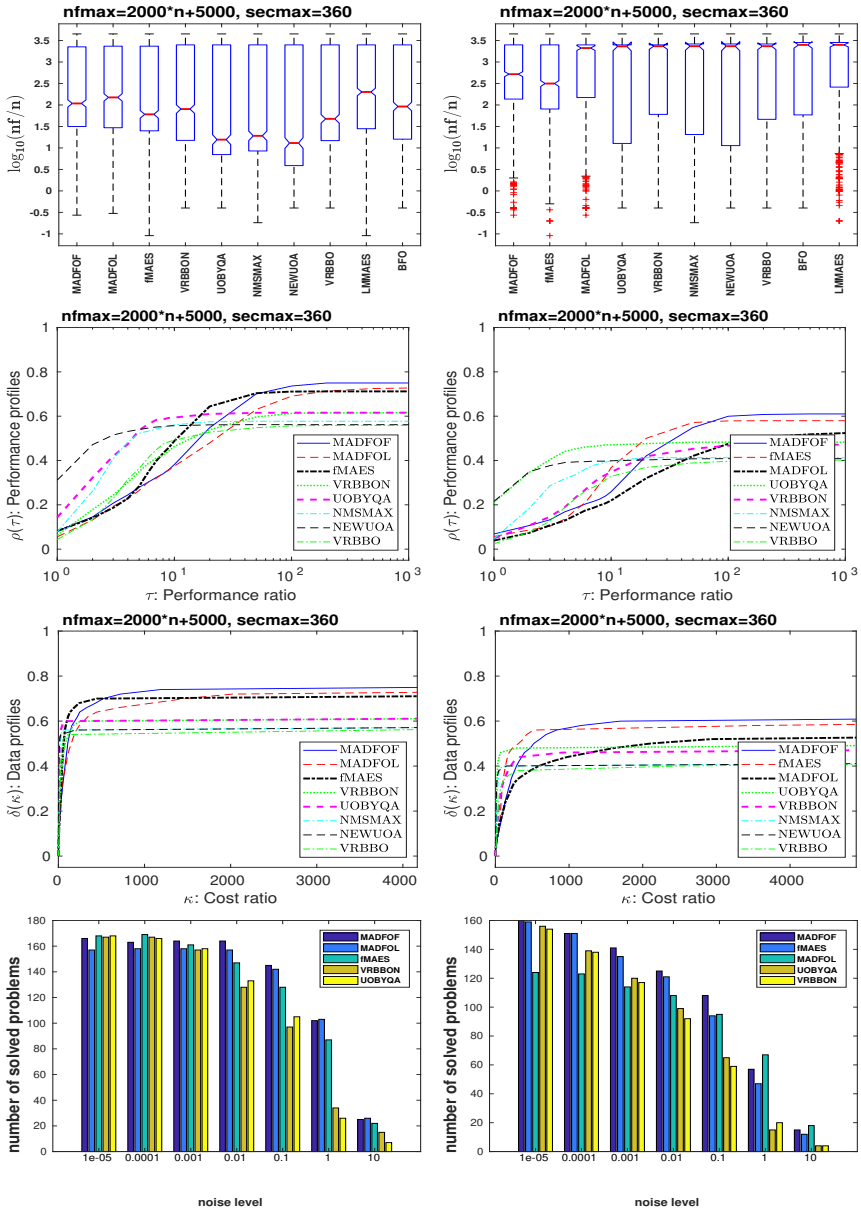


Fig. 9: For the relative uniform noise. Other details as Figure 5.

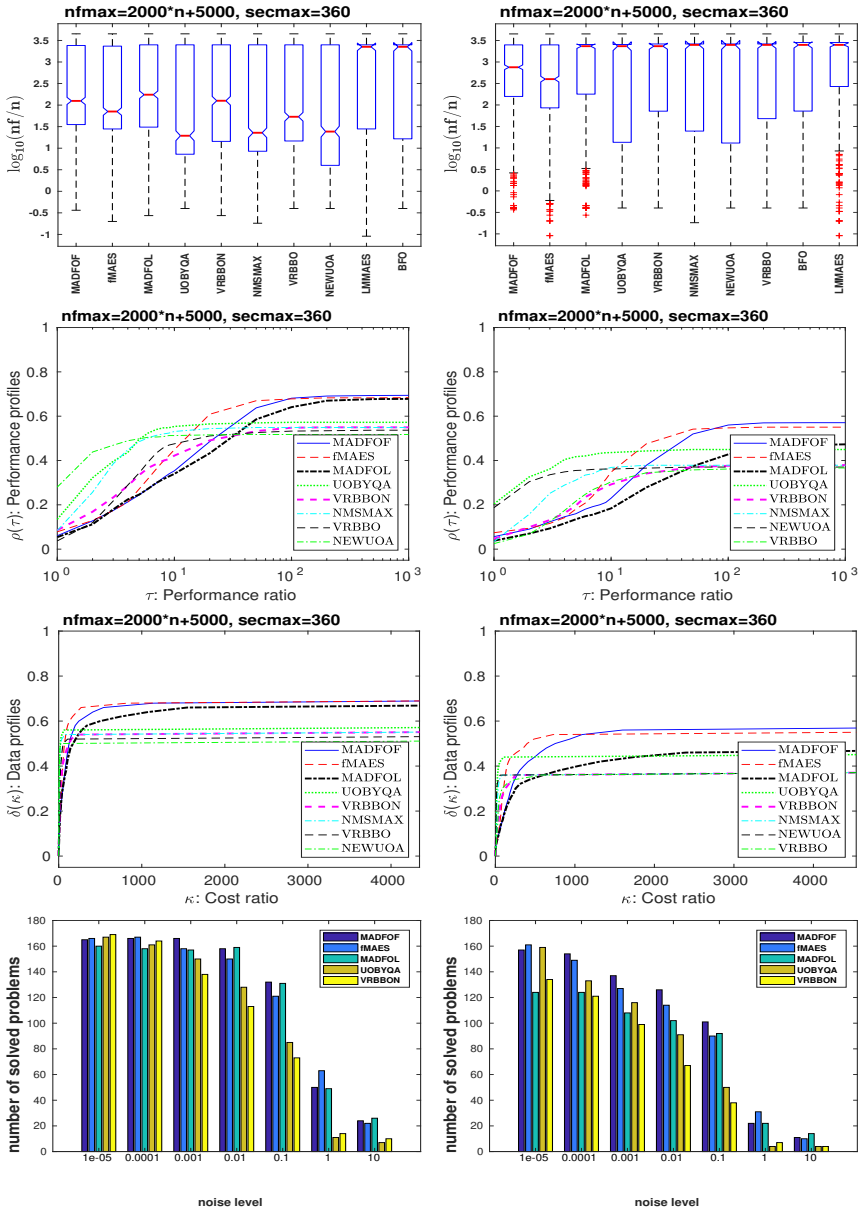


Fig. 10: For the relative Gaussian noise. Other details as Figure 5.

References

1. M. Ahookhosh and K. Amini. An efficient nonmonotone trust-region method for unconstrained optimization. *Numer. Algorithms* **59** (September 2011), 523–540.
2. K. Amini, H. Esmaeili, and M. Kimiaei. A nonmonotone trust-region-approach with nonmonotone adaptive radius for solving nonlinear systems. *IJNAO* **6** (February 2016).
3. S. Araki and S. Nishizaki. CALL-BY-NAME EVALUATION OF RPC AND RMI CALCULI. In *Theory and Practice of Computation*. WORLD SCIENTIFIC (September 2014).
4. C. Audet and W. Hare. *Derivative-Free and Blackbox Optimization*. Springer International Publishing (2017).
5. A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In *2005 IEEE Congress on Evolutionary Computation*. IEEE (2005).
6. A. S. Berahas, R. H. Byrd, and J. Nocedal. Derivative-free optimization of noisy functions via quasi-newton methods. *SIAM J. Optim.* **29** (January 2019), 965–993.
7. H. G. Beyer. Design principles for matrix adaptation evolution strategies (2020).
8. H. G. Beyer and B. Sendhoff. Simplify your covariance matrix adaptation evolution strategy. *IEEE Trans. Evol. Comput.* **21** (October 2017), 746–759.
9. E. G. Birgin, J. M. Martínez, and M. Raydan. Nonmonotone spectral projected gradient methods on convex sets. *SIAM J. Optim.* **10** (1999), 1196–1211.
10. R. Chen. *Stochastic Derivative-Free Optimization of Noisy Functions*. PhD thesis, Lehigh University (2015). Theses and Dissertations. 2548.
11. A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics (January 2009).
12. M.A. Diniz-Ehrhardt, J.M. Martínez, and M. Raydan. A derivative-free nonmonotone line-search technique for unconstrained optimization. *J. Comput. Appl. Math.* **219** (October 2008), 383–397.
13. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.* **91** (January 2002), 201–213.
14. C. Elster and A. Neumaier. A grid algorithm for bound constrained optimization of noisy functions. *IMA J. Numer. Anal.* **15** (1995), 585–608.
15. N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Comput. Optim. Appl.* **60** (2015), 545–557.
16. S. Gratton, C. W. Royer, L. N. Vicente, and Z. Zhang. Direct search based on probabilistic descent. *SIAM J. Optim* **25** (January 2015), 1515–1541.
17. S. Gratton, C. W. Royer, L. N. Vicente, and Z. Zhang. Complexity and global rates of trust-region methods based on probabilistic models. *IMA J. Numer. Anal.* **38** (August 2017), 1579–1597.
18. 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.
19. L. Grippo, F. Lampariello, and S. Lucidi. A nonmonotone line search technique for newton's method. *SIAM journal on Numerical Analysis* **23** (1986), 707–716.
20. L. Grippo and F. Rinaldi. A class of derivative-free nonmonotone optimization algorithms employing coordinate rotations and gradient approximations. *Comput. Optim. Appl.* **60** (June 2014), 1–33.
21. N. Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772* (2016).
22. N. J. Higham. Optimization by direct search in matrix computations. *SIAM J. Matrix Anal. Appl.* **14** (April 1993), 317–333.
23. W. Huyer and A. Neumaier. SNOBFIT – stable noisy optimization by branch and fit. *ACM. Trans. Math. Softw.* **35** (July 2008), 1–25.
24. C. T. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics (January 1999).

25. M. Kimiaei. A new class of nonmonotone adaptive trust-region methods for nonlinear equations with box constraints. *Calcolo* **54** (October 2016), 769–812.
26. M. Kimiaei. Line search in noisy unconstrained black box optimization. `http://www.optimization-online.org/DB_HTML/2020/09/8007.html` (Sep 2020).
27. M. Kimiaei, H. Esmaeili, and F. Rahpeymaii. A trust-region method using extended nonmonotone technique for unconstrained optimization. *Iranian Journal of Mathematical Sciences and Informatics* **16** (2021), 15–33.
28. M. Kimiaei and A. Neumaier. Efficient unconstrained black box optimization. *Mathematical Programming Computation* (February 2022).
29. M. Kimiaei and A. Neumaier. A new limited memory method for unconstrained nonlinear least squares. *Soft Computing* **26** (2022), 465–490.
30. M. Kimiaei, A. Neumaier, and P. Faramarzi. New subspace method for unconstrained black box optimization.
31. M. Kimiaei and F. Rahpeymaii. A new nonmonotone line-search trust-region approach for nonlinear systems. *TOP* **27** (January 2019), 199–232.
32. J. Larson, M. Menickelly, and S. M. Wild. Derivative-free optimization methods. *Acta Numer.* **28** (May 2019), 287–404.
33. D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.* **45** (August 1989), 503–528.
34. I. Loshchilov, T. Glasmachers, and H. G. Beyer. Large scale black-box optimization by limited-memory matrix adaptation. *IEEE Trans. Evol. Comput.* **23** (April 2019), 353–358.
35. S. Lucidi and M. Sciandrone. A derivative-free algorithm for bound constrained optimization. *Comput. Optim. Appl.* **21** (2002), 119–142.
36. J. L. Morales and J. Nocedal. Remark on "algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound constrained optimization". *ACM Transactions on Mathematical Software (TOMS)* **38** (2011), 1–4.
37. J. J. Moré and S. M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM J. Optim.* **20** (January 2009), 172–191.
38. J. J. Moré and S. M. Wild. Estimating derivatives of noisy simulations. *ACM Trans. Math. Softw.* **38** (April 2012), 1–21.
39. M. Porcelli and P. Toint. Global and local information in structured derivative free optimization with BFO. *arXiv: Optimization and Control* (2020).
40. M. J. D. Powell. UOBYQA: unconstrained optimization by quadratic approximation. *Math. Program.* **92** (May 2002), 555–582.
41. M. J. D. Powell. Developments of NEWUOA for minimization without derivatives. *IMA. J. Numer. Anal.* **28** (February 2008), 649–664.
42. L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *J. Global. Optim.* **56** (July 2012), 1247–1293.
43. H. J. M. Shi, Y. Xie, M. Q. Xuan, and J. Nocedal. Adaptive finite-difference interval estimation for noisy derivative-free optimization. *arXiv preprint arXiv:2110.06380* (2021).
44. Ph. L. Toint. An assessment of nonmonotone linesearch techniques for unconstrained optimization. *SIAM J. Sci. Comput.* **17** (1996), 725–739.
45. V. J. Torczon. *Multidirectional search: A direct search algorithm for parallel machines.* PhD thesis, Diss., Rice University (1989).
46. S. M. Wild, R. G. Regis, and C. A. Shoemaker. ORBIT: Optimization by radial basis function interpolation in trust-regions. *SIAM J. Sci. Comput.* **30** (January 2008), 3197–3219.
47. M. H Wright. Direct search methods: Once scorned, now respectable. *Pitman Research Notes in Math. Series* (1996), 191–208.