BFO, a trainable derivative-free Brute Force Optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables

M. Porcelli and Ph. L. Toint

Report naXys-06-2015 2 July 2015

University of Namur
8, rempart de la Vierge, B5000 Namur (Belgium)

http://www.naxys.be

BFO, a trainable derivative-free Brute Force Optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables

M. Porcelli* and Ph. L. Toint†

2 July 2015

Abstract

A direct-search derivative-free Matlab optimizer for bound-constrained problems is described, whose remarkable features are its ability to handle a mix of continuous and discrete variables, a versatile interface as well as a novel self-training option. Its performance compares favourably with that of NOMAD, a state-of-the art package. It is also applicable to multilevel equilibrium- or constrained-type problems. Its easy-to-use interface provides a number of user-oriented features, such as checkpointing and restart, variable scaling and early termination tools.

Keywords: derivative-free optimization, direct-search methods, mixed integer optimization, bound constraints, trainable algorithms.

Mathematics Subject Classification: 65K05, 90C56, 90C90.

1 Introduction

The efficient solution of optimization problems arising in real applications increasingly calls for the development of efficient and easy-to-use implementations of derivative-free algorithms. In applicative contexts such as engineering design [10, 20], medical image registration [26] and design of algorithms [6] (amongst many others), optimization problems are often defined by functions computed by costly simulation. A single simulation performed to evaluate the costly function may, for instance, require the solution of large systems of partial differential equations or a even a costly measurement campaign, and hence, may take from a few minutes to many hours or days depending on the particular application. Functions have therefore to be treated as expensive black-boxes and due to the high computational cost involved, it is important to use optimization algorithms that produce reasonably good solutions with a limited number of function evaluations. Moreover, optimization variables can be of different nature: continuous (e.g. geometrical parameters), integer (e.g. on/off element of a structure) or more generally categorical variables, which are discrete variables which identify an element of an unordered set (e.g. colors, shapes, or materials). It is also fairly common to have restrictions on the expected

*Università di Bologna, Dipartimento di Matematica, Piazza di Porta S. Donato 5, 40127 Bologna, Italy Email: margherita.porcelli@unibo.it

†Namur Center for Complex Systems (NAXYS), University of Namur, 61, rue de Bruxelles, B-5000 Namur, Belgium. Email: philippe.toint@unamur.ac.be

size of each variable which can be formulated as bound constraints. In some situations, the presence of bound constraints can prevent the computation of solutions which have no physical meaning. Furthermore, it is often helpful to introduce reasonable bounds on the variables when there is a good guess of the domain where solutions are expected. Finally, some applications also call for the solution of multilevel problems of the min-max type.

In order to model problems encompassing that complexity, we start by considering the following bound-constrained mixed variables nonlinear programming problem

$$\min_{x \in \Omega} f(x) \tag{1.1}$$

where $f : \Omega \to \mathbb{R}$ is a possibly nonsmooth (or even non continuous) function and the domain Ω is partitioned into continuous and discrete variables Ω^c and Ω^d of dimension n_c and n_d, respectively. Both domains are bound constrained, i.e. $\Omega^c = [l^c, u^c]$, where $l^c, u^c \in \mathbb{R}^{n_c} \cup \{\pm\infty\}$, $l^c \leq u^c$ and $\Omega^d = [l^d, u^d]$, where $l^d, u^d \in \mathbb{Z}^{n_d} \cup \{\pm\infty\}$, $l^d \leq u^d$. Let l, u be n-dimensional vectors such that $l^T = ({l^c}^T, {l^d}^T)$ and $u^T = ({u^c}^T, {u^d}^T)$, with $n = n_c + n_d$. Let $\mathcal{C} = \{i \in \{1, \dots n\} \mid x_i \text{ is continuous } \}$ and $\mathcal{D} = \{1, \dots, n\} \backslash \mathcal{C}$ be index sets of the continuous and the discrete variables ($n_c = |\mathcal{C}|, n_d = |\mathcal{D}|$). We assume that the evaluation of the objective function is time-consuming, and that no derivative is available. Moreover, non convexity assumption is made.

The very general nature of the problem effectively limits the choice among algorithm's classes for its solution to that of "direct-search methods", that is methods based on the (somewhat brute force) exploration of the variables' domain based on local sampling (for instance using pre-specified geometric patterns) and restricted to comparing objective function values (without interpolation or other type of modeling). A good introduction to direct-search methods may be found in Chapter 7 of the book by Conn, Scheinberg and Vicente [9]. These methods have a long history in the optimization literature (see Nelder and Mead [25], Hookes and Jeeves [18], Box and Wilson [7]) and have proved to be very popular among users, mostly because of their ease of use and robustness. A few direct-search methods can be applied to problem (1.1) in continuous variables although they involve some elements of objective function modeling: this the case of NOMAD by Abramson et al. [1, 2], SID-PSM by Custódio and Vicente [12, 13] and NMLSR/NMDFU by Grippo and Rinaldi [17]. On the other hand, to our knowledge, existing literature for solving (1.1) where the variables are mixed is not very extensive. Papers by Audet and Dennis [5] and by Lucidi, Piccialli and Sciandrone [23] as well the paper by Abramson et al. [1] (describing NOMAD) consider the mixed case and [21, 22] deals with problems whose variables are purely integer.

We also consider the multilevel problem

$$\min_{x_1 \in \Omega_1} \max_{x_2 \in \Omega_2} \dots \min_{x_m \in \Omega_m} f(x_1, \dots, x_m) \tag{1.2}$$

where x_1 to x_m form a partition of the variables in m "levels", for which specific feasible sets $\Omega_1, \dots \Omega_m$ are specified and where the objective function may be minimized or maximized (at the user's choice) at each level. Furthermore, we allow that each of the set Ω_ℓ ($1 < \ell \leq m$) may depend on the values of the variables in $x_1, \dots, x_{\ell-1}$. The authors are not aware of any software package aimed at solving this class of problems.

The purpose of the present paper is to present a new algorithm of the direct-search class for finding local solutions of problems (1.1) or (1.2) which handles mixed variables and also has the novel feature to be trainable by users to (typically application dependent) families of problems for improved efficiency. We discuss a Matlab code associated with this algorithm.

The paper is organized as follows. Section 2 presents the algorithm itself in the context of the optimization problem (1.1). Section 3 is dedicated to the numerical validation of BFO and

discusses how the algorithm can be used to optimize algorithmic parameters in itself or other numerical methods (Subsection 3.2). It also discusses a numerical comparison with NOMAD (Subsection 3.3). Section 4 describes how the parameter tuning feature of BFO can be used to make BFO itself trainable by users. Section 5 then considers how the algorithm can be adapted to problems of the form (1.2) while Section 6 describes additional code features. Finally, some conclusions and perspectives are outlined in Section 8.

Notation

Let I_q denote the Identity matrix of dimension $q \times q$. For any $v \in \mathbb{R}^q$ and $\mathcal{K} \subset \{1, \ldots, q\}$, we write $v_\mathcal{K}$ for the subvector of v having components $v_i, i \in \mathcal{K}$. Further, if $V = (V_{ij}) \in \mathbb{R}^{q \times q}$ we denote either by $V_{\mathcal{K}\mathcal{L}}$ or by $(V)_{\mathcal{K}\mathcal{L}}$ the submatrix of V with elements $V_{ij}, i \in \mathcal{K}, j \in \mathcal{L}$, let $\mathcal{N}$ represent the whole index set $\{1, \ldots, q\}$ and let $V_{\mathcal{N}j}$ be the jth column of V. Given the matrices $B \in \mathbb{R}^{m \times p}$ and $C \in \mathbb{R}^{m \times q}$, let $[B\ C] \in \mathbb{R}^{m \times (p+q)}$ denote the matrix concatenation and let $W_{p,q}$ denote a $p \times q$ matrix of uniformly distributed random numbers.

2 The BFO algorithm

We start by outlining the general features of the new BFO[1] algorithm for solving the optimization problem (1.1). BFO generates a sequence of feasible iterates whose objective function value is decreasing. Its underlying structure is that of a pattern search algorithm: at any given iteration, the objective function is evaluated at a finite number of points on a mesh in the neighbourhood of the current iterate, in an attempt to find a new point with a lower objective function value, which then becomes the next iterate. It is hoped that the sequence of such iterates approaches a local minimizer of problem (1.1).

More specifically, each iteration of the algorithm is initiated with the current iterate $\bar{x}$ and the current function value $\bar{f} = f(\bar{x})$ as well as with an enumerable set D of search directions for both continuous and discrete variables. These directions implicitly define the current mesh as the set of all points in $\Omega^c \times \Omega^d$ which can be reached from $\bar{x}$ by a move along one of the search directions in D with a fixed stepsize. Note that stepsizes and directions are fixed by the problem definition for discrete variables (for instance, the direction must be a coordinate vector and the step size has to be integer if the variable is integer), but both can be varied for continuous variables in the course of the numerical solution.

At the first iteration of BFO, an initial solution $\bar{x}$ and an initial set D of search directions is given, as well as a vector of stepizes $\delta = \delta^0 \in \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d}$. The algorithm then proceeds by exploring the lattice specified by these elements until no further improvement is possible. The continuous stepsizes $\delta_\mathcal{C}$ are then decreased, the search directions for continuous variables are possibly updated and the process repeated.

Exploration of a given mesh is conducted in one or two phases. In the first phase (called the POLL STEP and detailed in Section 2.1), a search is performed by computing forward and backward steps along all the current search directions from the current iterate. If this first phase does not succeed in improving the current point $\bar{x}$ and $n_d \neq 0$, a second phase may be entered. In this phase, a further search is performed by exploring the subproblems defined by fixing successively each of the discrete variables to a value neighbouring (by a move with the proper stepsize) that present in $\bar{x}$. This second phase is performed by recursively calling the algorithm itself for the solution of each such subproblem. This phase is called the RECURSIVE STEP and is detailed in Section 2.2.

[1] For Brute Force Optimizer.

These (possible) two phases are followed by the TERMINATION STEP (see Section 2.3). If a point with a better objective value than $\bar{f}$ is found in either phase, then the iteration is declared *successful*, otherwise the iteration is declared *unsuccessful*. In the successful case, the better point becomes the current iterate, and the next iteration is initiated with a coarser mesh in the continuous variables (EXPANSION SUBSTEP). In the unsuccessful case, either the grid may be further refined so that the next iteration is initiated at the same current solution, but with a finer mesh on the continuous variables (REFINEMENT SUBSTEP), or a check is performed to declare convergence on the finest grid (CHECK-CONV SUBSTEP).

A general outline of BFO is given in Algorithm 2.1 and its steps are described in detail in the following subsections.

Algorithm 2.1: An outline of the BFO algorithm for solving (1.1)

Initialization. Let $\bar{x} \in \Omega^c \times \Omega^d$ and $\bar{f} = f(\bar{x})$, the initial stepsize $\delta = \delta^0 \in \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d}$ and the initial set of moving directions D. Set the initial best value $x^{best} = \bar{x}$, $f^{best} = \bar{f}$.

Until convergence

1. POLL STEP. Perform a search loop on the variables, moving forward and backward along the directions in D with stepsize δ. If a poll point x^p constructed in this way is found such that $f(x^p)$ significantly improves f^{best}, then set $x^{best} = x^p$ and $f^{best} = f(x^p)$.
   If $f^{best} < \bar{f}$ or $n_d = 0$, go to the TERMINATION STEP.
2. RECURSIVE STEP. If requested, apply the BFO algorithm to solve the subproblems defined by fixing each of the discrete variables to a value differing from that in $\bar{x}$ by plus or minus the corresponding stepsize. If a point x^r is found such that $f(x^r) < f^{best}$, then set $x^{best} = x^r$ and $f^{best} = f(x^r)$.
3. TERMINATION STEP. If $f^{best} < \bar{f}$ [successful iteration], update $\bar{x} = x^{best}$ and $\bar{f} = f^{best}$, increase $\delta_{\mathcal{C}}$, update the set D for continuous variables and go to the POLL STEP (EXPANSION SUBSTEP).
   Else [unsuccessful iteration], check for convergence (CHECK-CONV SUBSTEP).
   If convergence is not declared, decrease $\delta_{\mathcal{C}}$, choose a new set D for for continuous variables and go to the POLL STEP (REFINEMENT SUBSTEP).

Note that it is also possible to include an additional "surrogate search step" before the TERMINATION STEP of this algorithm, for instance by exploiting any available approximation of the objective function constructed either from the available function values (see the "BFGS-finish" option in Section 6) or from additional specific evaluations.

2.1 The POLL STEP

The POLL STEP is a standard feature of direct-search algorithms: given the current iterate and function value $(\bar{x}, \bar{f})$, the objective function is evaluated at forward and backward mesh points in search of a better pair (x^{best}, f^{best}) such that $f^{best} < \bar{f}$. The algorithm therefore performs

a loop on both continuous and discrete variables, moving along the current direction as follows. Consider the i-th variable and assume first that $i \in \mathcal{C}$. Let j_c be the index in the ordered set of continuous variables, $Q \in \mathbb{R}^{n_c \times n_c}$ be a matrix whose columns form an orthonormal basis in Ω^c and let δ_i be the current stepsize. The continuous components of the forward poll point are then given by

$$x_{\mathcal{C}}^{fwd} = \bar{x}_{\mathcal{C}} + \alpha_f Q_{\mathcal{C}j_c}$$

where

$$\alpha_f = \min_{i \in \mathcal{C}} \alpha_i, \text{ with } \alpha_i = \begin{cases} \min\{u_i - \bar{x}_i, \delta_i\}/Q_{ij_c} & \text{if } Q_{ij_c} > 0, \\ \min\{l_i - \bar{x}_i, -\delta_i\}/Q_{ij_c} & \text{if } Q_{ij_c} < 0, \\ \infty & \text{else.} \end{cases}$$

If $i \in \mathcal{D}$, then the i-th component of the forward poll point is simply given by

$$x_i^{fwd} = \min\{\bar{x}_i + \delta_i, u_i\}.$$

The backward poll point associated with the i-th variable is competed analogously, setting

$$x_{\mathcal{C}}^{bwd} = \bar{x}_{\mathcal{C}} - \alpha_b Q_{\mathcal{C}j_c}$$

where

$$\alpha_b = \min_{i \in \mathcal{C}} \alpha_i, \text{ with } \alpha_i = \begin{cases} \min\{\bar{x}_i - l_i, \delta_i\}/Q_{ij_c} & \text{if } Q_{ij_c} > 0, \\ \min\{\bar{x}_i - u_i, -\delta_i\}/Q_{ij_c} & \text{if } Q_{ij_c} < 0, \\ \infty & \text{else,} \end{cases}$$

if $i \in \mathcal{C}$, and $x_i^{bwd} = \max\{\bar{x}_i - \delta_i, l_i\}$ if $i \in \mathcal{D}$. The current best function value f^{best} is then updated if $f_i^{fwd} = f(x^{fwd}) < f^{best}$ or $f_i^{bwd} = f(x^{bwd}) < f^{best}$.

The loop on the variables is terminated as soon as significant decrease is found, i.e. if

$$\bar{f} - f_i^{fwd} \geq \eta \Delta_f \quad \text{or} \quad \bar{f} - f_i^{bwd} \geq \eta \Delta_f \tag{2.3}$$

where $\eta \in (0, 1)$ is a parameter and Δ_f is initialized to plus infinity during the first loop, and updated after each complete loop using

$$\Delta_f = \bar{f} - f^{best}. \tag{2.4}$$

In order to avoid unnecessary computation, BFO also keeps track of 'fixed variables', that is variables whose lower and upper bounds are equal. Such variables are simply skipped in the poll-step loop.

Whenever the loop over all continuous non-fixed variables is completed, the POLL STEP is also exploited to compute an estimate of the projected gradient size for the continuous variables (whether this gradient exists or not) using the formula

$$\epsilon_{cr} = \| \max(l_{\mathcal{C}}, \min(x_{\mathcal{C}} - Q^T g_{dif}, u_{\mathcal{C}})) - x_{\mathcal{C}} \|, \tag{2.5}$$

where, for $i \in \mathcal{C}$,

$$(g_{dif})_i = \frac{f_i^{fwd} - f_i^{bwd}}{\|x_i^{fwd} - x_i^{bwd}\|}.$$

The value of ϵ_{cr} is not used by the algorithm but is supplied as an informational output to the user.

2.2 The RECURSIVE STEP

BFO allows the user to require further exploration of subspaces defined by fixing discrete variables. As is common in many techniques for exploring fixed subsets of integer variables, we model this exploration by a tree, where a child subset (or, in our case, subspace) is obtained from its father subset by fixing an additional variable, the root subset being that with no fixed variables at all. BFO allows the user to choose between the "depth-first" and "breadth-first" strategies to recursively explore the subspace tree. In the latter, all subspaces corresponding to potentially interesting values of the discrete variables are explored before grid refinement. In contrast, grid refinement is performed as soon as possible (before exploring other possible subspaces for the same mesh size) when depth-first search is chosen.

More specifically, let j be the index of a discrete variable and assume that a recursive subspace exploration is entered starting from x^s which is either x^{fwd} or x^{bwd}. Let also F be the index of the discrete variables which are already fixed at $\bar{x}$. BFO is then called to solve the subproblem

$$\min_{l \le x \le u} f(x) \quad \text{subject to} \quad x_i \text{ fixed for } i \in \{j\} \cup F,$$

where we use the 'fixed variable' feature mentioned in the previous paragraph.

In our implementation, the index $j \in \mathcal{D}$ is selected in increasing order starting from 1, but this choice is admittedly arbitrary. If breadth-first search is chosen, the recursion is called every time the POLL STEP did not produce an improved f^{best} (and discrete variables are present) and the mesh size of the child subproblem is inherited from the father calling problem. On the other hand, if depth-first search is employed, the recursion starts if both the POLL STEP is unfruitful and the search in the current subspace has terminated with a CHECK-CONV SUBSTEP. In this case, the mesh size of the subproblem is reset to the user-defined initial one. In both strategies, the recursive call ends when convergence is declared in the CHECK-CONV SUBSTEP.

2.3 The TERMINATION STEP

After the POLL STEP and (possibly) the RECURSIVE STEP, the algorithm evaluates the set

$$\mathcal{S} = \{i \in \mathcal{C},\ x_i^{best} - l_i \le \delta_i \text{ or } u_i - x_i^{best} \le \delta_i\},$$

of nearly saturated bounds at x_{best}, sets $n_s = |\mathcal{S}|$ and determines the matrix N of normals of these nearly saturated constraints. Then, if the iteration is successful, i.e. $f^{best} < \bar{f}$, it performs the EXPANSION SUBSTEP, itself consisting of three parts. Firstly, the mesh size for continuous variables is increased by a constant factor by setting

$$\delta_{\mathcal{C}} \leftarrow \min\Big[(u-l)_{\mathcal{C}},\ \min(\alpha\delta_{\mathcal{C}}, \gamma\delta^0_{\mathcal{C}})\Big], \tag{2.6}$$

where $\alpha \ge 1$ and $\gamma > 0$ are grid expansion factor and maximum grid expansion factor, respectively. Secondly, given an integer parameter `inertia` and the "progress direction" defined by

$$\Delta^{avg} = \sum_{j=1}^{\texttt{inertia}} \Delta x^{acc}_{\mathcal{C}j} \tag{2.7}$$

where $\Delta x^{acc}_{\mathcal{C}j}$ are the directions of descent $x_{best} - \bar{x}$ over the last `inertia` iterations, a new set of orthonormal directions $Q^{new} \in \mathbb{R}^{n_c \times n_c}$ is computed from the QR factorization

$$[N \ \ \Delta^{avg} \ \ W_{n_c, n_c - n_s - 1}] = Q^{new} R$$

for some upper-triangular matrix R. This change of basis has the effect of redefining the continuous variables, of projecting the progress direction onto the nullspace of the nearly saturated bounds and of ensuring that the normals of the nearly saturated constraints belong to the new basis. Thirdly, the new current iterate is redefined by $\bar{x} = x^{best}, \bar{f} = f^{best}$ and a new iteration started.

If, by contrast, the iteration is unsuccessful, the algorithm enters the CHECK-CONV SUBSTEP and checks for termination in the sense that convergence is (preliminarily) declared if

$$|(\delta_{\mathcal{C}})_i| \leq \epsilon, \quad i \in \mathcal{C}, \tag{2.8}$$

where $\epsilon > 0$ is a mesh-size threshold. In this case, further attempts to reduce the objective function are performed by a user-specified number of poll steps, each using a new randomly drawn orthonormal basis Q^{new} obtained from the QR factorization

$$[N \; W_{n_c, n_c - n_s}] = Q^{new} R. \tag{2.9}$$

If condition (2.8) is met every time, final convergence of BFO is declared and x^{best} is returned to the user as the best approximation solution found.

If this convergence test fails, the REFINEMENT SUBSTEP is then entered, where, given a grid shrinking factor $\beta \in (0, 1)$, the grid for the continuous variables is refined by setting

$$\delta_{\mathcal{C}} \leftarrow \max\{\epsilon/2, \; \beta\delta_{\mathcal{C}}\} \tag{2.10}$$

and Δ_f in (2.4) is reduce by the factor β. Analogously to the procedure used in the EXPANSION SUBSTEP, the new basis Q^{new} is defined from the QR factorization (2.9) and a new iteration is started.

3 Numerical experiments

This section is devoted to the numerical validation of the Matlab implementation of BFO. This issue is carried out through two series of experiments performed on a given set of benchmark problems: the first regards the fine-tuning of the BFO parameters and the second compares the practical behavior of BFO with that of a competitor solver chosen to represent the state-of-the-art of derivative-free solvers.

The comparative computational analysis is carried out by using performance and data profiles proposed in [24] for benchmarking derivative-free optimization algorithms. Following [24], we compare different solvers and algorithmic versions using this convergence test

$$f(x_0) - f(x) \geq (1 - \tau)(f(x_0) - f_*) \tag{3.11}$$

where x_0 is the starting point for the problem, x is the solution returned by a solver, f_* is computed for each problem as the smallest value of f obtained by any solver within a given number μ_f of function evaluations, $\tau \in [0, 1]$ is a tolerance that represents the the percentage decrease from the starting value $f(x_0)$. In practice (3.11) measures the function value reduction $f(x_0) - f(x)$ achieved by x relative to the best possible reduction $f(x_0) - f_*$.

Let $\texttt{fe}_{P,S}$ denote the total number of function evaluations needed for the solver S to solve problem P, that is to satisfy (3.11) for a given tolerance τ, and let $\texttt{fe}_P$ be the total number of function evaluations employed by the best solver to solve problem P.

We consider the classical *performance profile* function π_S defined as

$$\pi_S(t) = \frac{\text{number of problems s.t. } \texttt{fe}_{P,S} \leq t\,\texttt{fe}_P}{\text{number of problems}}, \quad t \geq 1,$$

that is the probability for solver S that a performance ratio $\texttt{fe}_{P,S}/\texttt{fe}_P$ is within a factor t of the best possible ratio.

We also consider a further measure of performances, the *data profile* function, that computes the percentage of problems that can be solved (for a given tolerance τ) within a certain number of function evaluations ν (the "budget"). The data profile function is defined as

$$\delta_S(\nu) = \frac{\text{number of problems s.t. } \texttt{fe}_{P,S} \leq \nu\,(n_P + 1)}{\text{number of problems}}, \quad \nu > 0,$$

where n_P is the number of variables in problem P. With the scaling $n_P + 1$, $\delta_S(\nu)$ can be interpreted as the percentage of problems that can be solved with the equivalent of ν simplex gradient estimates.

We note that the performance profile $\pi_S(t)$ measures how well the solver S performs relative to the other competitive solvers on a given set of problems, while the data profile $\delta_S(\nu)$ for a given solver S is independent of other solvers.

In our experiments we allowed a maximum number of 10000 function evaluations and considered two levels of accuracy $\tau = 10^{-i}$, $i = 4, 8$. In order guarantee the satisfaction of the condition (3.11) for these values of τ, we used tight tolerances in the solver converging tests. Finally, performance and data profiles are plotted in the following sections selecting $t \in [1, 5]$ and $\nu \in [0, 2500]$, respectively. Experiments were carried out using Matlab R2012a on Intel Core 2 Duo U7006 @1.2GHx2, 1.5 GB RAM.

3.1 The benchmark problems

Numerical results are given for problems from the CUTEst test collection [16]. The test examples we consider are constructed using the CUTEst interactive select tool in order to locate the subset of bound constrained problems and picking the problems with $n \leq 12$ and those that could be modified to reduced their dimension below 12 without loosing their meaning [2].

The resulting testing set consists of the 55 problems listed in Table 3.1 with their name, dimension n, number of free variables n_{fr}, fixed variables n_{fix} lower bounded variables n_l, upper bounded variables n_u and number of variables with both lower and upper bounds n_{lu}.

We consider two set of problems: the first, denoted as `Set-cont`, contains problems where variables are continuous (original problem formulation) and the second, `Set-mix`, which consists of problems with mixed-integer variables. `Set-mix` is built modifying the CUTEst problems imposing that some variables can only assume integer values. In particular, we imposed that all variables with even indexes are integers and rounded accordingly the corresponding bounds, i.e. $x_{2i}, l_{2i}, u_{2i} \in \mathbb{Z}$ for all i [3].

The new Matlab interfaces provided in [16] were used to test solvers on CUTEst problems.

3.2 The BFO parameters self-tuning

The BFO algorithm depends on a set of algorithmic parameters whose value may influence its numerical performance. Ideally, one would like to set these parameters to those values which gives the best numerical performance of BFO and set them as the default ones. In practice, a

[2] We exclude MINSURFO, the TORSION* and the OBSTCLA* family because arising from the discretization of 2D problems, the JNLBRNG* family because reducing the dimensions yield to only fixed variables. Moreover, we randomly chose only 4 problems within the PALMER* family.

[3] The only exceptions are problems HATFLDB, MAXLIKA, KOABHELB for which we considered variables 'icic', 'ccicici', 'cci', respectively, to obtain problems with meaningful bounds ('c'and 'i' stand for continuous and integer variables).

Name	n	n_{fr}	n_l	n_u	n_{lu}	n_{fix}	Name	n	n_{fr}	n_l	n_u	n_{lu}	n_{fix}
ALLINIT	4	1	1	1		1	KOEBHELB	4	1	2			
BDEXP	10		10				LINVERSE	9	4	5			
BIGGSB1	10	1			9		LOGROS	2		2			
CAMEL6	2				2		MAXLIKA	8				8	
CHARDIS0	10				10		MCCORMCK	10				10	
CHEBYQAD	4				4		MDHOLE	2	1	1			
CVXBQP1	10				10		NCVXBQP1	10				10	
EG1	3	1			2		NCVXBQP2	10				10	
EXPLIN	12				12		NCVXBQP3	10				10	
EXPLIN2	12				12		NONSCOMP	10				10	
EXPQUAD	12	6			6		OSLBQP	8		3		5	
HADAMALS	4				2	2	PALMER1A	6	4	2			
HARKERP2	10		10				PALMER2B	4	2	2			
HART6	6				6		PALMER3E	8	7	1			
HATFLDA	4		4				PALMER4A	6	4	1			
HATFLDB	4		3		1		PALMER4	4	1	3			
HATFLDC	9	1			8		PENTDI	5		5			
HIMMELP1	2				2		POWELLBC	6			6		
HS110	10				10		PROBPENL	10				10	
HS1	2	1	1				PSPDOC	4	3		1		
HS25	3				3		QUDLIN	12				12	
HS2	2	1	1				S368	8				8	
HS38	4				4		SIMBQP	2	1				1
HS3	2	1	1				SINEALI	4				4	
HS3MOD	2	1	1				SPECAN	9				9	
HS45	5				5		WEEDS	3		2		1	
HS4	2		2				YFIT	3	2	1			
HS5	2				2								

Table 3.1: The benchmark problem set.

default parameter configuration can be computed by approximating the parameters which gives the best performance of BFO on a set of test problems that is chosen to be representative enough to "ensure" good performance of the solver on further problems.

We focus on the 7 BFO parameters reported in Table 3.2 together with their description and type: 5 parameters are continuous c, one is integer i and one (`stype`) is categorical d. For `stype`, we associated to the discrete tree search strategy labels $\{\text{BF}, \text{DF}, \text{none}\}$ the integer values $\{0, 1, 2\}$, respectively, and treated it as an integer parameter in the range $[0, 2]$. Note that this parameter is not necessary if a testing problem has only continuous variables. In addition, optimization with respect to the seed of the random number generator (labeled as `rseed`) is considered.

$p_\#$	Parameter	Type	Description
p_1	α	c	The grid expansion factor (see (2.6))
p_2	β	c	The grid shrinking factor (see (2.10))
p_3	γ	c	The maximum grid expansion factor (see (2.6))
p_4	δ	c	The initial stepsize vector (see Section 2)
p_5	η	c	The sufficient decrease fraction in the poll step (see (2.3))
p_6	`inertia`	i	The inertia for continuous step accumulation (see (2.7))
p_7	`stype`	d	The discrete tree search strategy $\{\text{BF}, \text{DF}, \text{none}\}$ (see Section 2.2)

Table 3.2: Table of BFO parameters.

The aim of this section is to estimate the best BFO parameters configuration with respect to the benchmark problem set. To address this issue we consider two parameter optimization

problems formulated as a bound-constrained black-box optimization problem of the form (1.1) where the variables are the BFO parameters, and we use the BFO itself to solve it.

The first formulation, first proposed in [6] and used later in [3, 4], is based on defining the number of objective function evaluations as measure of (negative) performance of the algorithm and use a derivative-free solver to minimize it. This technique is implemented as follows. Let $\mathcal{T}$ be the set of test problems described in Section 3.1 and let $p = (p_1, \ldots, p_7)$ be the BFO parameters listed in Table 3.2. We choose reasonable default values p_0 for the parameters and associated lower/upper bounds l_p and u_p. Then, we use BFO to solve the "Average Objective" (AO) problem

$$\min_{l_p \le p \le u_p} \ \phi_{BFO}(p) \tag{3.12}$$

where $\phi_{BFO}(p)$ counts the total number of evaluations of f to solve all the problems in $\mathcal{T}$ with parameters p.

The second formulation relies on robust optimization which provides a tool for protecting against strong local variation of performance by looking for a safe worst-case scenario [11]. We follow this approach by allowing perturbations of each continuous algorithmic parameter by at most 5% around each tested value and defining the local box

$$\mathcal{B} = \prod_{i=1}^{5} [0.95\, p_i, 1.05\, p_i] \times \prod_{i=6}^{7} [p_i, p_i]\,,$$

and use BFO to solve the problem "Robust Objective" (RO) problem

$$\min_{l_p \le p \le u_p} \ \max_{p \in \mathcal{B}} \ \phi_{BFO}(p) \tag{3.13}$$

where ϕ_{BFO}, l_p, u_p are defined as above.

In the experiments, we set the starting parameter p_0 and the bounds l_p, u_p as given in Table 3.3 and the initial scaling $\delta_p = (u_p - l_p)/10$ for continuous parameters and $\delta_p = 1$ for the integer ones.

	α	β	γ	δ	η	`inertia`	`stype`	`rseed`
p_0	2	0.5	5	1	10^{-3}	10	0	0
l_p	1	0.01	1	0.25	10^{-5}	5	0	0
u_p	2	0.95	10	10	0.5	30	2	100

Table 3.3: Parameter setting for the BFO self-tuning.

Moreover, we used the value $\epsilon = 10^{-13}$ in the convergence test (2.8) in the solution of a single problem in $\mathcal{T}$ required to evaluate ϕ_{BFO}. On the other hand, in the same test, we set $\epsilon = 10^{-2}$ in the solution of the outer minimization problem in both (3.12) and (3.13), and $\epsilon = 10^{-1}$ in the solution of the inner minimization problem (3.13). Finally, we set an upper bound of 100 parameter configuration trials.

Due to the strongly non-monotone performance of the code as a function of the random seed, three tuning options were considered : optimize the algorithmic parameters first and the random seed second (PS strategy), optimize the random seed first and the algorithmic parameters second (SP strategy), or optimize the random seed and the algorithmic parameters together (T strategy). Moreover, the optimization was conducted separately on the set of continuous and the set of mixed-integer problems. The results (in terms of percentage of performance improvement) are reported in Table 3.4. As a consequence, the PS tuning option was chosen

for the continuous problems and the SP strategy for the mixed-integer ones[4]. This choice is confirmed while comparing performance profiles for the different options (not presented here).

training strategy	tuning option	continuous problems		mixed-integer problems	
		gain for each step	total gain	gain for each step	total gain
AO	PS	23% + 5%	28%	13% + 1%	14%
	SP	8% + 0%	8%	1% + 14%	15%
	T	23%	23%	13%	13%
RO	PS	21% + 9%	30%	19% + 0%	19%
	SP	8% + 20%	28%	1% + 18%	19%
	T	23%	23%	16%	16%

Table 3.4: Improvement in performance according to tuning options.

In Table 3.5 we give the estimated parameters p_{AO} and p_{RO} computed using the AO formulation (3.12) and the RO formulation (3.13), respectively. Values of p_{AO} and p_{RO} slightly differ and both suggest to use the depth-first strategy (`stype` = `1`) in the RECURSIVE STEP. Figure 3.1 shows the corresponding performance profiles which reveals that: using p_{RO} yields the most efficient version of BFO on `Set-cont`; the performance of BFO with p_{RO} and p_{AO} is comparable for $t \approx 1$ on `Set-mix` and both outperform BFO with p_0; the robustness of the three versions of BFO is comparable (see percentage values in brackets).

		α	β	γ	δ	η	`inertia`	`stype`	`rseed`
continuous problems	p_{AO}	1.6366	0.3426	4.4507	8.5744	0.1142	13	-	64
	p_{RO}	1.4248	0.1997	2.3599	1.0368	0.4528	11	-	53
mixed-integer problems	p_{AO}	2	0.0448	5	1	0.1190	10	1	91
	p_{RO}	2	0.3135	5	3.6030	10^{-5}	10	1	91

Table 3.5: Estimated optimal BFO parameters.

We conclude this section by the (important in our view) observation that the same approach can be used to optimize performance of other numerical algorithms, using BFO to solve the associated AO or RO problems.

3.3 Comparison with NOMAD

As a competitor solver, we considered the state-of-the-art solver for derivative-free mixed variable nonlinear optimization NOMAD release 3.6.2 [2, 19]. NOMAD (Nonsmooth Optimization by Mesh Adaptive Direct Search) belongs to the class of direct-search methods and is based on the the recent development of Mesh Adaptive Direct Search methods [1]. NOMAD is in fact an hybrid method that enhances the efficiency of MADS methods by combining direct-search strategies with different types of surrogate models in a mesh adaptive direct search filter method.

We now report on the numerical comparison between BFO and NOMAD evaluated in both "direct-search" mode and "model-based" mode (denoted as NOMAD-DS and NOMAD-MB, respectively). In our experiments we set $\mu_f = 10000$ and considered two levels of accuracy $\tau = 10^{-i}$, $i = 4, 8$. In both BFO and NOMAD, the internal stopping criterion is based on driving the mesh size below a tolerance ϵ that we set as $\epsilon = 10^{-13}$. All other NOMAD parameters have been set as the default ones [5].

[4]This choice will be used in all the experiments presented in this paper in the subsequent sections.

[5]When NOMAD is tested in the "direct-search" mode, we disabled the options `MODEL_SEARCH` and `MODEL_EVAL_SORT` in order not to use model based strategies.

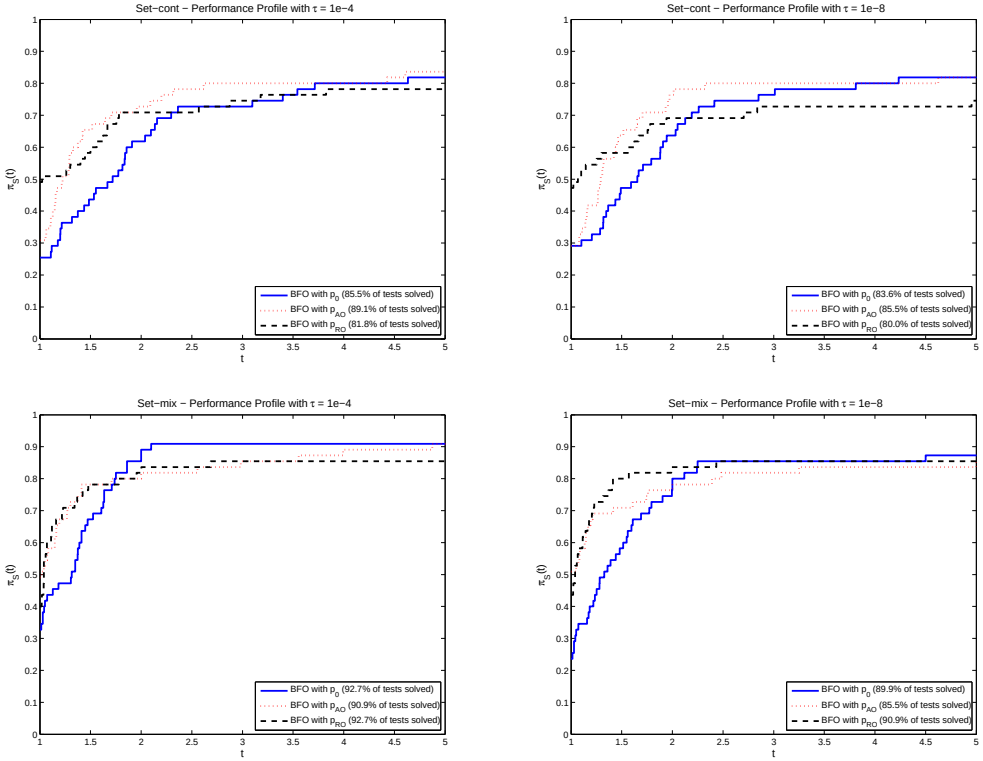

Figure 3.1: Performance profiles of BFO with different algorithmic parameters on `Set-cont` (top) and on `Set-mix` (bottom). Cutoff $\tau = 10^{-4}$ (left) and $\tau = 10^{-8}$ (right).

In Figures 3.2 and 3.3, we plot the comparison between the three versions of BFO, i.e. with parameters p_0, p_{AO} and p_{RO} of Tables 3.3 and 3.5, and the two versions of NOMAD.

Figure 3.2 shows that BFO with p_{RO} is the most efficient in the 40% of the tests on `Set-cont` while in the solution of `Set-mix` BFO with p_{AO} is the most efficient for $\tau = 10^{-4}$; for $\tau = 10^{-8}$ the performance of the compared solvers is similar for $t \approx 1$. It is also clear in Figure 3.3, that the tuned versions of BFO are very competitive with NOMAD-MB on `Set-mix` since the plotted curves are very close for $t \geq 1.5$ while, unsurprisingly, the NOMAD model based approach is more efficient than BFO on the smooth continuous problems in `Set-cont`, especially for $\tau = 10^{-8}$.

Remarkably, BFO is on average more robust than NOMAD in that it solves around 10-15% more problems then the competitor (see the percentage values in brackets).

Finally, we report the corresponding data profiles in Figures 3.4 and 3.5. From these profiles, it is clear that when the computational budget is small, say 100 simplex gradient evaluations, the behaviour of the competing solvers is comparable. On the other hand, for both accuracy levels τ, as the computational budget increases, BFO solves a larger number of problems than NOMAD (both versions) and the difference increases with the number of simplex gradients ν.

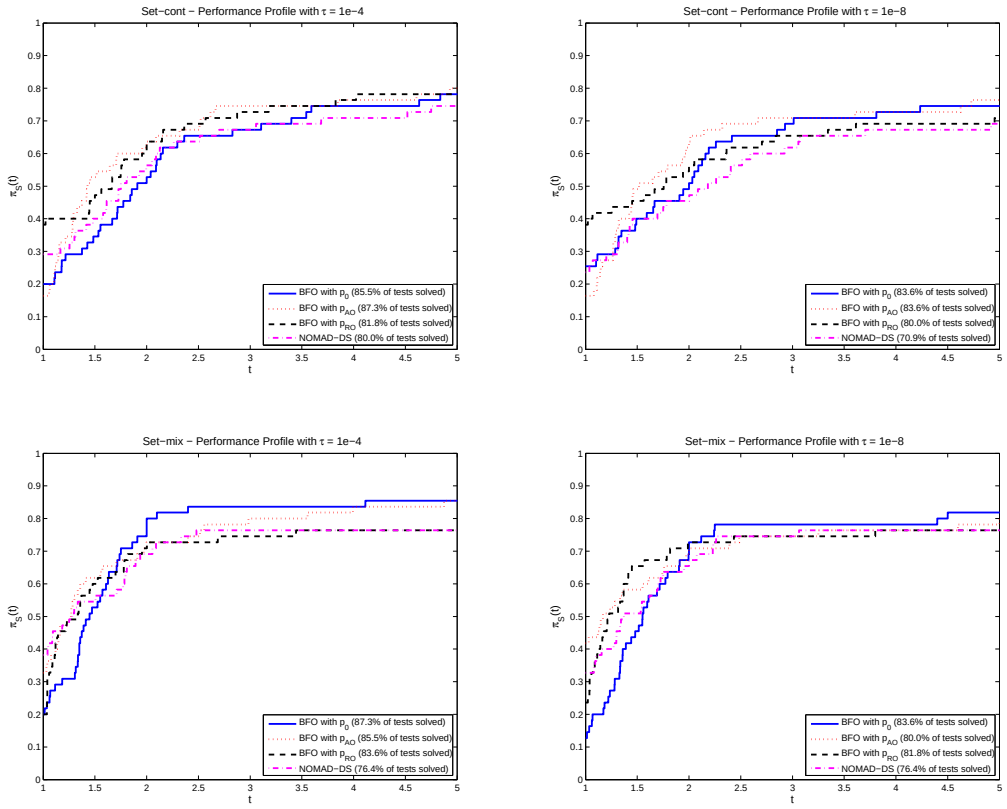

Figure 3.2: Performance profiles of BFO against NOMAD-DS on `Set-cont` (top) and on `Set-mix` (bottom). Cutoff $\tau = 10^{-4}$ (left) and $\tau = 10^{-8}$ (right).

4 BFO as a trainable algorithm

If the performance of the BFO algorithm can be optimized with itself, the obvious next step is to provide this facility within the code, allowing a user to specify a set of test problems (we used the CUTEst test problems above) and optimizing performance on this class. As a result, we obtain what we call a "trainable algorithm": given a set of test problems, a trainable algorithm can be trained for (hopefully) improved performance on this set and subsequently applied to further problems (of the same type) using the internal configuration (algorithmic parameters) resulting from this training.

In BFO, this facility is implemented by allowing the user to specify three possible "training modes". The first correspond to the training phase only and solves problem AO or RO on the user-supplied set of training problems. The second mode first perform this training and then immediately uses the resulting optimized algorithmic parameters to solve one or more new problems. The third mode first reads previously trained parameters from a file and then uses them for the solution of a new problem. Various options may be specified, allowing the user to choose between AO and RO, specifying the name of the file where trained parameters are saved and restricting the training to certain meaningful sets of parameters. We refer to the description of the BFO input parameters for more detail. Note that BFO being a descent method implies

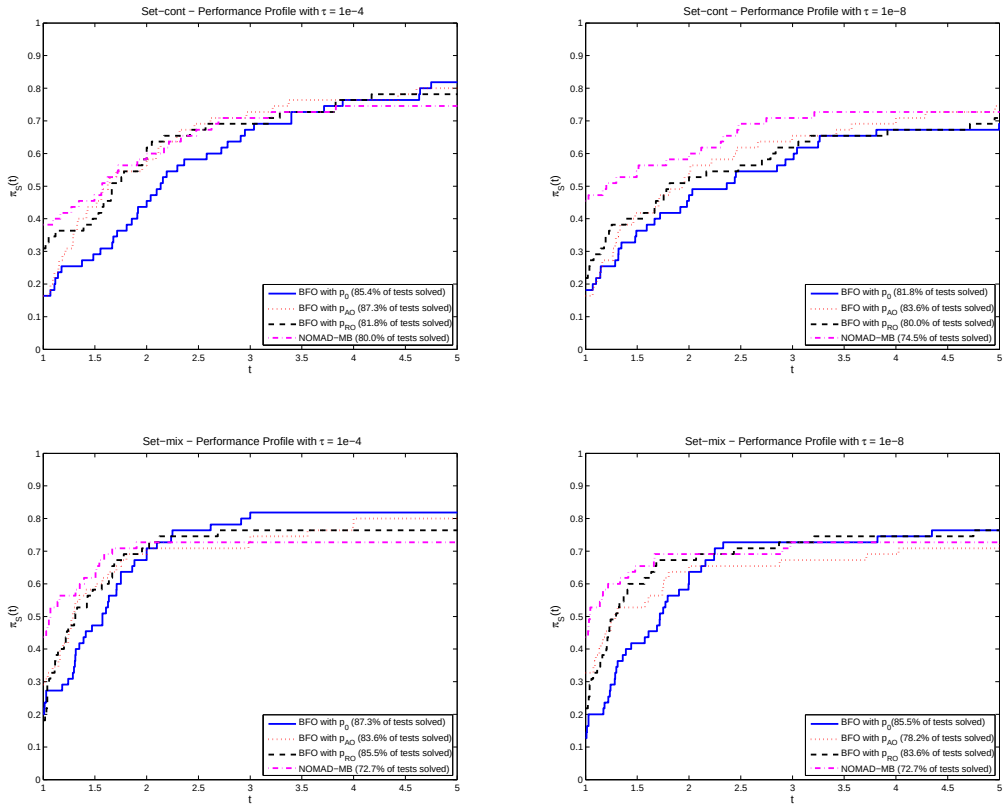

Figure 3.3: Performance profiles of BFO against NOMAD-MB on `Set-cont` (top) and on `Set-mix` (bottom). Cutoff $\tau = 10^{-4}$ (left) and $\tau = 10^{-8}$ (right).

that performance is improved at every training iteration, and therefore that accurate solution of the training problem AO or RO is generally unnecessary (and potentially leading to overfitting).

We now illustrate the potential benefits and pitfalls of training by considering three specific class of minimization problems.

4.1 Nonlinear least squares

The first is a class of nonlinear least-squares problems with bounds[6] where one seeks to fit a nonlinear model of a vibrating beam (hence our name of VBEAM for this problem class) to data by minimizing

$$f(x_1, x_2, x_3) = \sum_{j=0}^{16} \left[x_3 \tan\left(x_1 \left(1 - \frac{j}{16}\right) + x_2 \frac{j}{16} \right) - y_j \right]^2$$

[6]Derived from the YFIT problem in CUTEst and corresponding to fitting data to Doppler measures of a vibrating beam. The original problem was proposed by L. Watson (VPI).

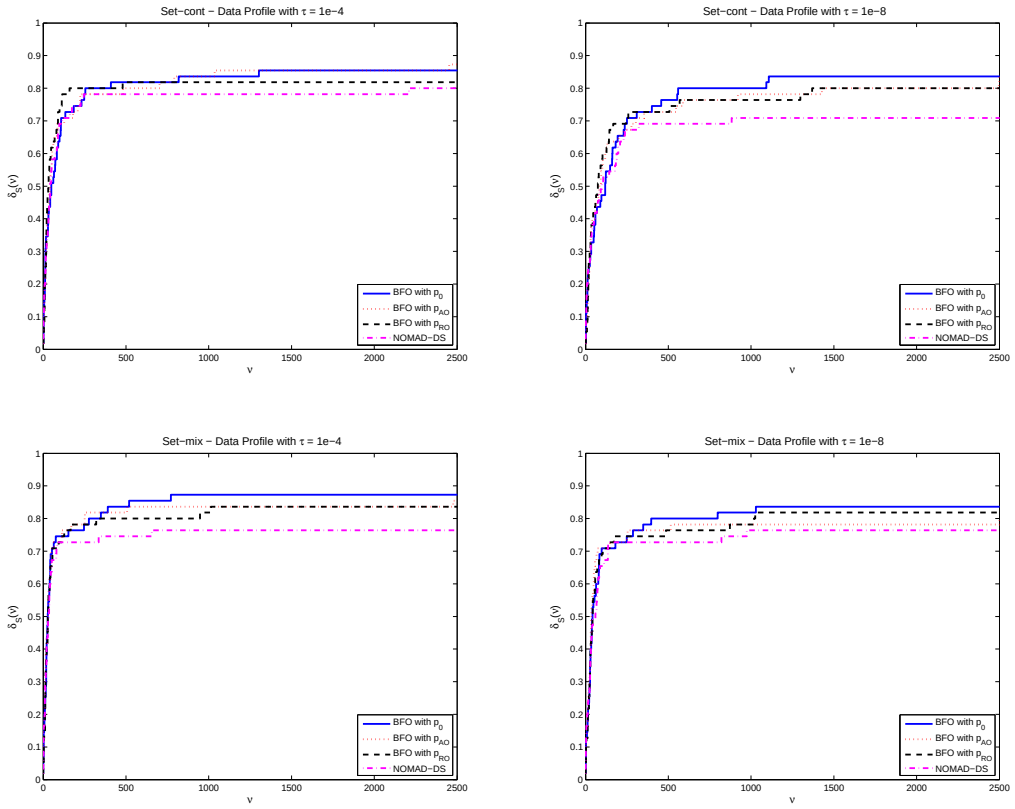

Figure 3.4: Data profiles of BFO against NOMAD-DS on `Set-cont` (top) and on `Set-mix` (bottom). Cutoff $\tau = 10^{-4}$ (left) and $\tau = 10^{-8}$ (right).

where $x_3 \geq 0$. We fixed values for the three variables[7] and generated several classes of 10 problems, where

$$y_j = x_3 \tan\left(x_1\left(1 - \frac{j}{16}\right) + x_2\,\frac{j}{16}\right)(1+\eta_j) \quad (j = 0, \ldots 16),$$

with $\eta_0 = 0$ and η_j $(j > 0)$ being a realisation of a Gaussian noise with zero mean and a prescribed value of the standard deviation σ, each class of test problems corresponding to a different value of σ. For each such class, we trained BFO using both average and robust training strategies on the 10 generated test problems, and then applied the trained BFO on 20 additional control problems generated with the same parameters and standard deviation, in order to measure effectiveness of the training on this control set.

We report some results obtained in the setting in Tables A.6 and A.7 in appendix. In these tables, ϵ_1 is the accuracy requirement of the outer minimization in the training problem formulation (see (3.12) and (3.13)), ϵ_2 is the accuracy requirement in the inner maximization of (3.13), σ is the standard deviation described in the preceding paragraph, "neval" is the total number of evaluation of problems in the testing set, "t-set" is the gain/loss (in percentage) in

[7] $x_1^* = 0.21$, $x_2^* = -0.35$ and $x_3^* = 1$, $x_3^* = 10$ or $x_3^* = 100$.

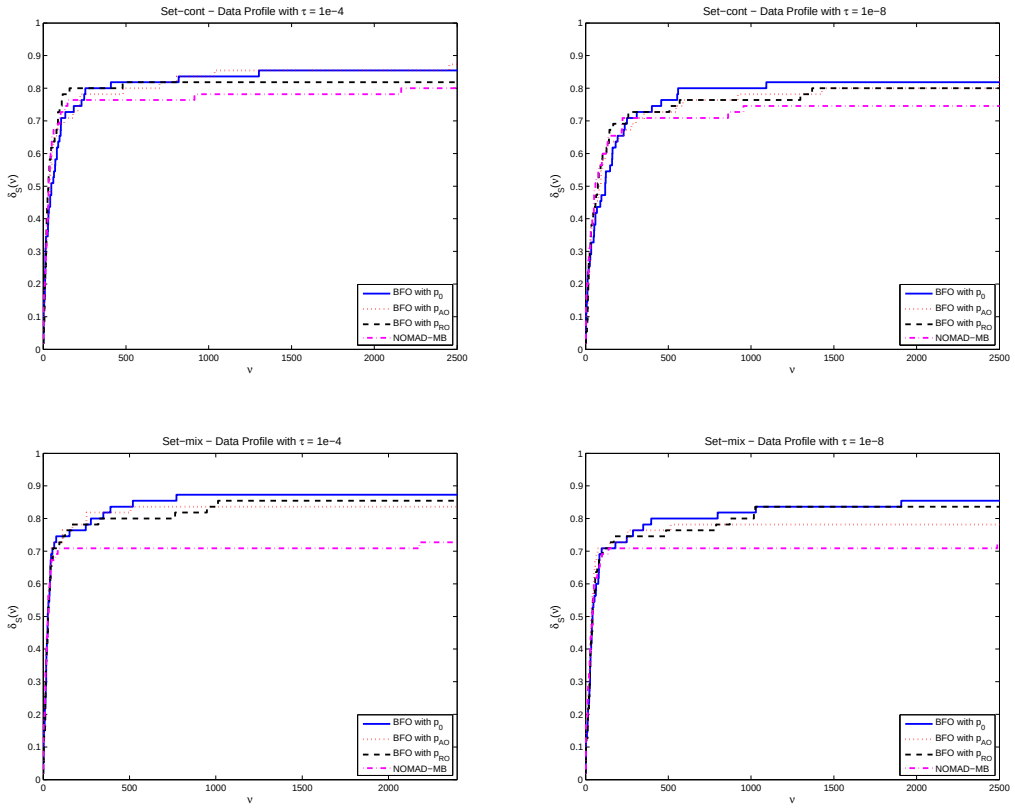

Figure 3.5: Data profiles of BFO against NOMAD-MB on `Set-cont` (top) and on `Set-mix` (bottom). Cutoff $\tau = 10^{-4}$ (left) and $\tau = 10^{-8}$ (right).

the number of problem function evaluations achieved by the training phase on the 10 training problems of each class and "c-set" is the gain/loss of problem function evaluations obtained on the control set consisting of the 20 additional problems of each class which were not included in the training set. We note that

We now attempt some tentative conclusions on this specific class of problems:

1. The gains in performance obtained by training using either strategy, although clearly not guaranteed, may be substantial, especially for narrowly defined problems classes (small values of σ). Reported values approach 60% in some cases, but further experimentation not reported here indicate that even higher gains may sometimes be possible. On average, gains appear to be of the order of 25%.

2. Asking more accuracy for the outer minimization in training ($\epsilon_1 = 0.1$) is typically[8] more costly.

[8]The nonconvexity of the training criteria nevertheless shows in that different approximate minimizers can be found for successive training runs with different accuracy requirements. Similarly, the geometry of problems may vary with σ. As a consequence monotonicity of cost and/or performance with respect to increased requirements or decreasing noise is not always preserved.

3. Unsurprisingly, a very moderate accuracy requirement for this outer minimization often seems perfectly sufficient. Higher accuracy levels may increase the cost of training without any guarantee of improvement in performance. This observation appears to hold for both training strategies, with the possible exception of the worse conditioned problems ($x_3 = 100$) with larger standard deviation ($\sigma \geq 0.5$) when the RO strategy is used.

4. Again unsurprisingly, the RO training strategy is nearly always substantially more costly that the AO one (by a factor often exceeding one order of magnitude). It may however produce benefits in terms of training capacity for problems with relatively high values of σ, that is problems where deviations within the class are proportionally larger.

5. Increasing the accuracy in the inner maximization (ϵ_2) in the RO strategy again appears to bring few benefits, with the same marginal exception of the worse conditioned problems with larger standard deviation.

4.2 Regularized cubics

The second class of problems (RCUBIC) on which training was experimented is a class of unconstrained problems featuring an regularized cubic objective function of the form

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= -\mu[x_1(1+\nu_1) + x_2(1+\nu_2) + x_3(1+\nu_3) + x_4(1+\nu_4)] \\ &\quad -x_1^2 - 2x_2^2 - 3x_3^2 - 4x_4^2 + 10(1+\nu_5)[x_1^2 + x_2^2 + x_3^2 + x_4^2]^{3/2} \end{aligned}$$

where μ is a parameter in $\{0.1, 1, 10\}$ and $\{\nu_i\}_{i=1}^5$ are realizations of a Gaussian random noise with zero mean and prescribed values of the standard deviation σ. As for the VBEAM class, 10 training problems were generated for each value of μ and σ. BFO was then trained on this set for different accuracy levels using both the AO and RO strategies and the resulting algorithmic parameters were then used to solve 20 control problems generated with the same σ. The results of these experiments are reported in Tables B.8 and B.9 in appendix.

These tables suggest the following comments.

1. The typical gains in performance are smaller on this problem class than those obtained for the VBEAM class, but they remain non-negligible (from 10 to 30% except for the case $\sigma = 1$).

2. For both strategies, the evolution of the gains with increasing value of the noise standard deviation σ is somewhat unpredictable, the best results being often obtained for intermediate values of this problem parameter. However, the largest standard deviation typically leads to worse performance.

3. As for the VBEAM class, moderate training accuracy seems most often sufficient to extract good performance, both for the AO and RO strategies.

4. Again the RO strategy is considerably more costly, in this case for results comparable on the whole to those obtained for AO.

Summarizing, the experiments reported in this section indicate that training has a non-negligible cost but may yield significant efficiency gains when the class of problems considered is sufficiently well-defined. Moreover, it appears that the RO training strategy, albeit clearly designed for increased robustness, does not (in these two experiments at least) deliver very convincingly on this ground in view of its very significantly higher training cost. However it definitely should be remembered that gains are not guaranteed, and therefore that the encouraging but

tentative conclusion (especially for the AO strategy) must be taken with due caution. If the user wishes to solve a large number of problems within a well-defined class of interest, some experimentation with training is probably advisable.

5 Tackling multilevel problems

We now turn to the description of how BFO has been adapted to solve problems of the form (1.2), provided each optimization subproblem (at different levels) are well-defined. The fact that forward and backward steps are used by the algorithm makes it easy to restrict minimization (or maximization) to specific subspace. In particular, variable selection is trivial as it is sufficient to fix one or more variables to define a proper subspace. Indeed, assume for simplicity that there are two levels involving vectors of variables x_1 and x_2, respectively. In order to evaluate the objective function for the outer (level 1) optimization problem, BFO recursively calls itself to optimize the objective on x_2 while keeping the variables in x_1 fixed.

Which variable is assigned to which level is specified by the user using an optional argument (it is then required that every variable is assigned a level, and that every level is assigned at least one variable). Another argument allows the specification of the choice of minimization or maximization at each level.

As we indicated in the introduction, the variables at each level may be constrained by their own set of bounds, and these bounds may themselves depend on the value of the variables at levels of lower index. This is achieved by calling a user-supplied function defining these "variable bounds" in a reasonably flexible format. This feature has the marginal effect that it allows considering constrained problems of the form

$$\min_x f(x_1, x_2) \quad \text{subject to} \quad g(x_1) \leq x_2 \leq h(x_1)$$

by reformulating them as two levels problem

$$\min_{x_1} \min_{x_2} f(x_1, x_2)$$

where the "variable bounds" on x_2 are defined by $g(x_1) \leq x_2 \leq h(x_1)$. Since the "variable bounds" definition may itself call BFO, it is also possible to tackle problems of the form

$$\min_{x_1 \in \mathbb{R}^n} f(x_1, x_2) \quad \text{subject to} \quad \min_{x_1 \in \mathbb{R}^n} g(x_1) \leq x_2 \leq \min_{x_1 \in \mathbb{R}^n} h(x_1)$$

where x_1 and/or $x_2 \in \mathbb{R}$ may contain a mix of discrete or continuous variables and where additional (fixed) bounds may be imposed on x_1 and x_2. An example where these features are used is provided in the file `test_equilibrium.m`, where the code is used to solve a non-smooth mixed-integer leader-follower equilibrium problem where a producer of two goods optimizes the values and prices of these goods, given production costs depending on price and limits on prices depending on values, and given a demand for the two goods. The demand is determined by a class of consumers who, in turn, optimize at their level the perceived values of the goods they buy under a budget constraint. The first good can only be bought in integer quantities while the second good is bulk and can be bought in real quantities.

The multilevel facility coupled with the definition of the variable bounds technique just illustrated therefore shows considerable flexibility, but it must be kept in mind that recursive optimization over two or more levels may be expensive in terms of objective function evaluations, even after training on a specific class of multilevel problems.

6 Additional BFO features

The BFO code also provides a few additional facilities for the user, which we briefly describe.

termination on objective function target: In some applications, the cost of complete optimization is simply to high, especially in the context of derivative-free methods where asserting convergence may itself be a reasonably costly algorithmic phase. Many users are therefore more interested in obtaining a decent decrease/increase of the objective function from a starting value than in pursuing optimization to its conclusion. BFO allows the user to terminate optimization before convergence in two different ways. The first is for the user to return an infinite or NaN value, in which case termination occurs immediately. The second is to specify a "target value" for the objective function, and the algorithm then terminates as soon as this target is attained.

cheap unsuccessful objective evaluation: Many relevant optimization problem occur in the form where the objective function consists of a sum of positive terms. In these cases, it is possible to save significant computational effort by determining, in the course of the evaluation of the objective function itself, it the accumulated value for a given number of terms already results in a value too large for the evaluation to be considered successful by the algorithm. BFO provides the necessary interface to allow stopping evaluation by ignoring further terms.

We note that algorithm training as described above is by nature a problem where this feature can be applied. Indeed its use results in a 10% saving in evaluations when training BFO with the AO strategy, compared to the naive version ignoring this structure. When combined with the use of objective function targets (just described), the computational cost of RO training strategy typically decreases by an order of magnitude.

variable-dependent scaling: It may happen that variables in problem have different scalings, resulting in ill-conditioning and termination difficulties if this property is ignored. BFO allows the user to explicitly specify variable scalings in order to avoid this type of detrimental numerical behaviour. This is achieved by specifying a scaling vector (xscale) whose values are used to (statically) scale the variables before problem solution is attempted.

user-specified discrete lattice: By default, BFO considers discrete variables as integer, but also provides the possibility for the user to specify a discrete lattice on which optimization must be carried on. This is done by passing to BFO matrix whose columns corresponding to discrete variables contain a basis of this lattice. Optimization on the i-th (discrete) variable is then interpreted as optimization along fixed multiples of the i-th column of the given matrix.

checkpointing and restart: Because optimization with costly objective function may be time consuming, it is useful for an algorithm to provide checkpointing and restart facilities. This is the case in BFO, where the user may specify the checkpointing frequency and the name of the checkpointing file(s).

BFGS finish: When convergence is approached on smooth problems, the grid is refined following iterations where no improvement can be obtained in the polling loop. However, a complete polling loop provides enough function evaluations to allow for a central difference estimation of the gradient at the current iterate. This in turn can be exploited at successive iterates of this type, the associated differences in (estimated) gradient being used to build a BFGS [8, 14, 15, 27] variable-metric approximation of the (assumed) second derivatives,

at least in the presence of positive curvature. A quasi-Newton step may then be computed. This facility is provided as an option in BFO, and often results in significantly higher accuracy of the solution for a moderate increase in function evaluations.

a CUTEst interface: In order to facilitate comparison with other packages, an interface to the CUTEst testing environment [16] is also provided.

7 The BFO code and examples of use

The Matlab code for BFO consists of the single Matlab m-file `bfo.m`. A reasonably thorough test program (`test_bfo.m`) is also provided, as well as a small set of problem examples and testing cases. Substantial effort went into the documentation of the code, in order to make it understandable and as easy to use as possible. In particular, the various arguments and options for the main function, `bfo.m`, are explained at length in its header.

To illustrate the use and flexibility of BFO, we now give a few examples of use, without describing the different keywords in their details. However, we believe that they are sufficiently self-explanatory for the reader to grasp the overall picture and be convinced that using the code is easy.

1. Minimize the ever-famous Rosenbrock "banana valley" function of $x = (x_1, x_2)$ from the starting point $(-1.2, 1)$ with

   ```
   [ x, fx ] = bfo( @banana, [-1.2 1] )
   ```

   where

   ```
   function fx = banana( x )
   fx  = 100 * ( x(2) - x(1)^2 )^2 + (1 - x(1) )^2;
   ```

2. Minimize the banana function subject to $x_1 \geq 0$ and $x_2 \leq 2$:

   ```
   [ x, fx ] =  bfo( @banana,[-1.2,1],'xlower',[0,-Inf],'xupper',[Inf,2] )
   ```

3. Minimize the banana function by limiting grid accuracy and maximum number of objective function evaluations:

   ```
   [ x, fx ] =  bfo( @banana, [-1.2 1], 'epsilon', 1e-2, 'maxeval', 50 )
   ```

4. Minimize the banana function, assuming that x_1 is fixed to -1.2:

   ```
   [ x, fx ] =  bfo( @banana, [-1.2, 1], 'xtype', 'fc' )
   ```

5. Minimize the banana function, assuming that x_1 can only take integer values:

   ```
   [ x, fx ] =  bfo( @banana, [-1, 1], 'xtype', 'ic' )
   ```

6. Minimize the banana function, assuming that x_1 and x_2 can only move along unit multiples of the (1, 1) and (-1, 1) vectors, respectively:

   ```
   [ x, fx ] =  bfo( @banana, [-1, 1], 'xtype', 'ii',...
                     'lattice-basis', [ 1 -1; 1 1 ] )
   ```

7. Maximize the negative of the banana function:

```
[ x, fx ] = bfo( @negbanana, [-1.2 1], 'max-or-min', 'max' )
```

8. Minimize the banana function without any printout:

```
[ x, fx ] =  bfo( @banana, [-1.2, 1], 'verbosity', 'none' )
```

9. Minimize the banana function with checkpointing every 10 evaluations in the file 'bfo.restart'

```
[ x, fx ] = bfo( @banana, [-1.2, 1],                              ...
                 'save-freq', 10, 'restart-file', 'bfo.restart' )
```

10. Restart the minimization of the banana function after a saved checkpointing run using the file 'bfo.restart':

```
[ x, fx ] = bfo( @banana, [-1.2, 1], x0,                          ...
                 'restart', 'use', 'restart-file', 'bfo.restart' )
```

11. Train BFO on the 'fruit training set' and save the resulting algorithmic parameters in the file 'fruity':

```
[ x, ... ,trained_parameters ] =                                    ...
  bfo( 'training-mode', 'train', 'trained-parameters, 'fruity',     ...
       'training-problems' ,   {@banana,     @apple,     @kiwi},    ...
       'training-problem-data',{@banana_data,@apple_data,@kiwi_data} )
```

12. Train BFO on the 'fruit training set' and use the resultant trained algorithm to solve the `orange` problem:

```
[ x, ... ,trained_parameters ] = bfo( @orange, x0,                  ...
       'training-mode',        'train-and-solve',                   ...
       'training-problems',    {@banana,     @apple,     @kiwi},    ...
       'training-problem-data',{@banana_data,@apple_data,@kiwi_data} )
```

13. Solve the `orange` problem after having trained BFO on the 'fruit training set' and having saved the resulting algorithmic parameters in the file 'fruity' (for instance by previously using the call indicated in Example 11 above):

```
[ x, ... ,trained_parameters ] = bfo( @orange, x0,                ...
       'training-mode', 'solve', 'trained-bfo-parameters' ,'fruity')
```

or

```
[ xbest, ... ,trained_parameters ] =                        ...
       bfo( @orange, x0, 'trained-bfo-parameters', 'fruity')
```

14. Train BFO on three problems from CUTEst and save the resulting algorithmic parameters in the file 'cutest.parms':

```
[ x, ... ,trained_parameters ] =                                ...
       bfo( 'training-mode','train',                            ...
            'trained-parameters, 'cutest.parms',                ...
            'training-problems', {'HS4', 'YFIT', 'KOWOSB'}, ...
            'training-problems-library', 'cutest' )
```

15. Solve the problem of computing the unconstrained min-max of the function `fruit_bowl`(x) defined as the sum of `apple`(x_1, x_2) and `negbanana`(x_3, x_4), where the min is taken on x_1 and x_2 and the max on x_3 and x_4:

```
[ x, fx ] = bfo( @fruit_bowl, [ -1.2 1 -1.2 1 ],                    ...
                 'xlevel', [ 1 1 2 2 ], 'max-or-min', ['min';'max'] )
```

16. Solve (very inefficiently) the problem of minimizing the banana function subject to the constraints $0 \leq x_1 \leq 2$ and $x_1 \leq x_2$:

```
[ x, fx ] = bfo( @banana, [ 0, 0 ], 'xlevel', [ 1 2 ],         ...
                 'max-or-min', ['min';'min'],                   ...
                 'xlower', [ 0, -Inf ], 'xupper', [ 2, Inf ], ...
                 'variable-bounds', 'banana_vb' )
```

where the user has provided

```
function [ xlow, xupp ] = banana_vb( x, level, xlevel, xlower, xupper )
xlow(1) = 0; xlow(2) = x(1); xupp = xupper;
```

8 Conclusion

We have presented BFO, a versatile and robust Brute Force Optimizer for small-scale bound-constrained problems based on a simple direct-search strategy, whose distinguishing features are its ability to handle a mix of continuous and discrete variables and its innovative self-training capacity. The fact that it can also handle multilevel problems, although possibly at higher cost, is also a plus. BFO is written in Matlab. On CUTEst problems, its performance compares favourably with that of NOMAD, a state-of-the-art direct-search package, most notably in terms of reliability.

As its name suggests and despite a favourable but limited comparison with NOMAD, BFO has no pretense of superior efficiency, especially for multilevel problems. It is hoped that it will nevertheless turn out to be useful because of its versatility, trainable nature and robustness. In particular, it application for optimizing algorithmic parameters in various numerical methods, in optimization and beyond, is of definite interest.

Acknowledgments

The work of the first author was supported by *National Group of Computing Science (GNCS-INdAM)* of Italy. This author wishes to thank Francesco Rinaldi for his support and for valuable discussions on derivative-free methods. Both authors also thank Dominique Orban for helpful suggestions.

References

[1] M. A. Abramson, C. Audet, J. W. Chrissis, and J. G. Walston, *Mesh adaptive direct search algorithms for mixed variable optimization*, Optimization Letters, 3 (2009), pp. 35–47.

[2] M. A. Abramson, C. Audet, G. Couture, J. J. Dennis, and S. Le Digabel, *The NOMAD project*, Software available at http://www.gerad.ca/nomad.

[3] C. Audet, C.-K. Dang, and D. Orban, *Algorithmic parameter optimization of the DFO method with the OPAL framework*, in Software Automatic Tuning, K. Naono, K. Teranishi, J. Cavazos, and R. Suda, eds., Springer New York, 2010, pp. 255–274.

[4] C. Audet, K.-C. Dang, and D. Orban, *Optimization of algorithms with OPAL*, Mathematical Programming Computation, (2014), pp. 1–22.

[5] C. Audet and J. Dennis, *Pattern search algorithms for mixed variable programming*, SIAM Journal on Optimization, 11 (2001), pp. 573–594.

[6] C. Audet and D. Orban, *Finding optimal algorithmic parameters using derivativefree optimization*, SIAM Journal on Optimization, 17 (2006), pp. 642–664.

[7] G. E. P. Box and K. B. Wilson, *On the experimental attainment of optimum conditions*, Journal of the Royal Statistical Society. Series B (Methodological), 13 (1951), pp. pp. 1–45.

[8] C. G. Broyden, *The convergence of a class of double-rank minimization algorithms 1. general considerations*, IMA Journal of Applied Mathematics, 6 (1970), pp. 76–90.

[9] A. Conn, K. Scheinberg, and L. Vicente, *Introduction to Derivative-Free Optimization*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[10] A. R. Conn, K. Scheinberg, and P. L. Toint, *A derivative free optimization algorithm in practice*, in Proceedings of 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, vol. 48, 1998.

[11] A. R. Conn and L. N. Vicente, *Bilevel derivative-free optimization and its application to robust optimization*, Optimization Methods and Software, 27 (2012), pp. 561–577.

[12] A. Custódio, H. Rocha, and L. Vicente, *Incorporating minimum Frobenius norm models in direct search*, Computational Optimization and Applications, 46 (2010), pp. 265–278.

[13] A. Custódio and L. Vicente, *Using sampling and simplex derivatives in pattern search methods*, SIAM Journal on Optimization, 18 (2007), pp. 537–555.

[14] R. Fletcher, *A new approach to variable metric algorithms*, The Computer Journal, 13 (1970), pp. 317–322.

[15] D. Goldfarb, *A family of variable-metric methods derived by variational means*, Mathematics of computation, 24 (1970), pp. 23–26.

[16] N. I. Gould, D. Orban, and P. L. Toint, *CUTEst: a Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization*, Computational Optimization and Applications, 60 (2015), pp. 545–557.

[17] L. Grippo and F. Rinaldi, *A class of derivative-free nonmonotone optimization algorithms employing coordinate rotations and gradient approximations*, Computational Optimization and Applications, 60 (2015), pp. 1–33.

[18] R. Hooke and T. Jeeves, *"Direct search" solution of numerical and statistical problems*, Journal of the ACM (JACM), 8 (1961), pp. 212–229.

[19] S. Le Digabel, *Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm*, ACM Transactions on Mathematical Software, 37 (2011), pp. 44:1–44:15.

[20] M. Leonetti, P. Kormushev, and S. Sagratella, *Combining local and global direct derivative-free optimization for reinforcement learning*, Cybernetics and Information Technologies, 12 (2012), pp. 53–65.

[21] G. Liuzzi, S. Lucidi, and F. Rinaldi, *Derivative-free methods for bound constrained mixed-integer optimization*, Computational Optimization and Applications, 53 (2012), pp. 505–526.

[22] ——, *Derivative-free methods for mixed-integer constrained optimization problems*, Journal of Optimization Theory and Applications, (2014), pp. 1–33.

[23] S. Lucidi, V. Piccialli, and M. Sciandrone, *An algorithm model for mixed variable programming*, SIAM Journal on Optimization, 15 (2005), pp. 1057–1084.

[24] J. Moré and S. Wild, *Benchmarking derivative-free optimization algorithms*, SIAM Journal on Optimization, 20 (2009), pp. 172–191.

[25] J. A. Nelder and R. Mead, *A simplex method for function minimization*, The computer journal, 7 (1965), pp. 308–313.

[26] R. Oeuvray and M. Bierlaire, *A new derivative-free algorithm for the medical image registration problem*, International Journal of Modelling and Simulation, 27 (2007), p. 115.

[27] D. F. Shanno, *Conditioning of quasi-Newton methods for function minimization*, Mathematics of computation, 24 (1970), pp. 647–656.

A Results for the training on the VBEAM problems

		$\epsilon_1 = 0.1$			$\epsilon_1 = 0.01$			$\epsilon_1 = 0.001$		
x_3^*	σ	neval	t-set	c-set	neval	t-set	c-set	neval	t-set	c-set
1.0	0.05	78016	57%	58%	84676	57%	58%	91376	57%	58%
	0.10	74664	57%	57%	92144	57%	57%	99099	57%	57%
	0.50	76770	55%	51%	90714	55%	51%	97697	55%	51%
	1.00	118002	25%	18%	127859	25%	18%	137598	25%	18%
10.0	0.05	115103	25%	24%	117627	30%	25%	153660	31%	26%
	0.10	91736	30%	27%	101330	30%	27%	116878	31%	27%
	0.50	140689	29%	29%	150873	29%	29%	161938	29%	29%
	1.00	371770	74%	-3%	391895	74%	-3%	469877	75%	-3%
100.0	0.05	1528163	40%	17%	2259418	46%	35%	2409211	46%	35%
	0.10	949534	74%	-63%	990102	74%	-63%	1028300	74%	-63%
	0.50	1058333	44%	37%	1311953	44%	20%	1420676	44%	20%
	1.00	2246316	53%	44%	2282538	42%	28%	2428030	42%	28%

Table A.6: Training performance for the VBEAM problems (average strategy).

			$\epsilon_2 = 0.1$			$\epsilon_2 = 0.01$			$\epsilon_2 = 0.001$		
x_3^*	σ	ϵ_1	neval	t-set	c-set	neval	t-set	c-set	neval	t-set	c-set
1.0	0.05	0.100	628646	34%	32%	1785918	26%	13%	1524726	24%	19%
		0.010	696865	29%	25%	2228106	32%	14%	1861051	24%	19%
		0.001	801217	30%	22%	2388132	31%	21%	2526828	27%	22%
	0.10	0.100	613487	39%	34%	1206625	26%	22%	2082234	30%	35%
		0.010	594036	28%	14%	1602390	30%	29%	1786235	34%	32%
		0.001	832147	28%	14%	2227721	33%	29%	1814774	34%	36%
	0.50	0.100	752798	34%	18%	829479	26%	0%	1291995	25%	11%
		0.010	826141	26%	15%	1569947	32%	15%	1915998	27%	9%
		0.001	893174	27%	3%	1576704	32%	15%	2209387	28%	10%
	1.00	0.100	915320	35%	23%	1428637	34%	26%	1903385	31%	15%
		0.010	1007303	35%	27%	1316097	32%	21%	3275141	33%	16%
		0.001	1057979	36%	28%	1415476	33%	23%	3454439	33%	15%
10.0	0.05	0.100	550136	31%	26%	2231802	29%	25%	2154878	25%	19%
		0.010	742751	33%	27%	2384730	31%	25%	3565739	30%	26%
		0.001	711696	33%	27%	2384978	31%	25%	3425180	29%	29%
	0.10	0.100	356406	34%	27%	1825640	33%	22%	483576	34%	27%
		0.010	415057	34%	27%	1482960	29%	16%	542227	34%	27%
		0.001	437556	34%	27%	1576560	29%	16%	564726	34%	27%
	0.50	0.100	581248	49%	26%	960202	64%	14%	1568987	71%	25%
		0.010	630377	49%	26%	971803	64%	14%	1648454	71%	25%
		0.001	718977	49%	26%	1015169	64%	14%	1828169	71%	25%
	1.00	0.100	1875988	72%	-5%	2012749	69%	0%	4731855	64%	5%
		0.010	2495629	70%	7%	2416306	69%	0%	5870814	65%	3%
		0.001	2687561	70%	7%	2391194	69%	0%	6639303	64%	6%
100.0	0.05	0.100	22800147	49%	53%	23756246	46%	51%	46830361	52%	57%
		0.010	13796099	49%	50%	54424183	53%	55%	41559646	45%	54%
		0.001	16016353	49%	50%	54424183	53%	55%	46977030	45%	54%
	0.10	0.100	12074874	51%	18%	24934693	53%	24%	28315648	50%	20%
		0.010	18603805	51%	20%	26371545	53%	24%	22841398	43%	21%
		0.001	17117883	46%	21%	27787326	53%	24%	27516073	43%	21%
	0.50	0.100	5825325	41%	32%	14433847	34%	44%	15522908	42%	44%
		0.010	5929728	41%	32%	17728327	34%	44%	18043836	42%	44%
		0.001	5943597	41%	32%	19799363	35%	47%	20980871	42%	44%
	1.00	0.100	15984178	39%	43%	50144474	46%	46%	49081955	54%	23%
		0.010	24300473	43%	43%	69466970	49%	13%	57829751	56%	56%
		0.001	26649456	43%	43%	72037240	49%	13%	61968367	56%	56%

Table A.7: Training performance for the VBEAM problems (robust strategy).

B Results for the training on the RCUBIC problems

		$\epsilon_1 = 0.1$			$\epsilon_1 = 0.01$			$\epsilon_1 = 0.001$		
μ	σ	neval	t-set	c-set	neval	t-set	c-set	neval	t-set	c-set
0.1	0.05	171390	21%	18%	215379	21%	18%	236948	21%	18%
	0.10	147549	27%	32%	163969	27%	32%	179003	27%	32%
	0.50	210625	27%	22%	231710	28%	21%	353271	30%	21%
	1.00	1316810	2%	2%	1669344	2%	1%	2420867	2%	1%
1.0	0.05	157465	29%	19%	251780	31%	24%	364525	35%	22%
	0.10	145578	27%	21%	164747	27%	21%	209898	27%	18%
	0.50	167160	33%	20%	213654	33%	20%	275630	34%	19%
	1.00	884294	7%	4%	1117489	7%	4%	1325105	8%	5%
10.0	0.05	170696	24%	17%	190527	24%	17%	210716	24%	17%
	0.10	250660	27%	21%	197321	21%	17%	217332	21%	17%
	0.50	252651	17%	19%	292688	17%	19%	313567	17%	19%
	1.00	365754	9%	6%	577169	11%	7%	621727	11%	7%

Table B.8: Training performance for the RCUBIC problems (average strategy).

			$\epsilon_2 = 0.1$			$\epsilon_2 = 0.01$			$\epsilon_2 = 0.001$		
μ	σ	ϵ_1	neval	t-set	c-set	neval	t-set	c-set	neval	t-set	c-set
0.1	0.05	0.100	904337	25%	25%	2666451	20%	18%	1780697	17%	20%
		0.010	955052	25%	25%	2816439	20%	18%	1865210	17%	19%
		0.001	1051446	26%	27%	2969297	20%	18%	2656816	17%	22%
	0.10	0.100	408303	25%	28%	1899920	24%	34%	1975824	22%	27%
		0.010	426958	25%	28%	1910765	22%	27%	3459456	24%	33%
		0.001	486780	25%	28%	2770792	25%	25%	2713023	23%	28%
	0.50	0.100	929811	22%	16%	1038464	22%	20%	1262599	24%	21%
		0.010	776602	22%	20%	1098088	22%	20%	1594679	23%	20%
		0.001	775473	22%	20%	1774483	24%	23%	1740226	23%	20%
	1.00	0.100	5435143	2%	1%	22978138	1%	2%	42813655	1%	1%
		0.010	6475795	2%	1%	25992859	1%	2%	31156577	1%	-1%
		0.001	9047623	2%	1%	43715316	1%	2%	34462404	1%	0%
1.0	0.05	0.100	935440	19%	14%	1395242	23%	19%	1613559	19%	21%
		0.010	1306734	20%	18%	1532900	21%	16%	3032890	22%	20%
		0.001	1382806	20%	18%	1631930	21%	16%	2916307	22%	16%
	0.10	0.100	1036208	20%	21%	723048	20%	20%	2423883	19%	19%
		0.010	1262466	22%	26%	1257674	20%	23%	3574962	21%	20%
		0.001	1485215	23%	21%	1420361	21%	17%	4467090	22%	22%
	0.50	0.100	878268	23%	20%	980388	23%	20%	4097011	22%	20%
		0.010	990193	27%	25%	1141757	26%	21%	2641673	21%	16%
		0.001	1180931	27%	21%	1269361	26%	21%	2810517	21%	16%
	1.00	0.100	4589589	4%	1%	6923078	4%	1%	11537196	4%	2%
		0.010	5064799	4%	1%	8512039	4%	1%	24376570	4%	4%
		0.001	6214429	4%	2%	8925949	4%	1%	21217312	4%	2%
10.0	0.05	0.100	797424	26%	12%	2328438	24%	17%	1578391	25%	17%
		0.010	818112	26%	12%	2358203	27%	20%	1794730	25%	17%
		0.001	951865	26%	12%	2498016	27%	20%	1992326	25%	17%
	0.10	0.100	701318	20%	11%	2566099	20%	18%	2978151	24%	21%
		0.010	1065867	18%	15%	3025223	20%	18%	3711438	23%	18%
		0.001	1183465	18%	15%	3166889	21%	17%	4214931	23%	14%
	0.50	0.100	1383221	19%	21%	1529375	18%	17%	1671648	18%	19%
		0.010	1355317	19%	16%	2017567	19%	23%	2307945	18%	20%
		0.001	1481907	19%	16%	2136717	19%	23%	4720503	19%	20%
	1.00	0.100	2161854	5%	5%	7751704	6%	6%	4880611	7%	4%
		0.010	2254753	5%	5%	10487409	7%	4%	9767706	8%	6%
		0.001	2234031	5%	5%	10908540	7%	4%	10192074	8%	6%

Table B.9: Training performance for the RCUBIC problems (robust strategy).