Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers

Coralia Cartis * Jan Fiala † Benjamin Marteau ‡ Lindon Roberts §

29th May 2019

Abstract

We present two software packages for derivative-free optimization (DFO): DFO-LS for nonlinear least-squares problems and Py-BOBYQA for general objectives, both with optional bound constraints. Inspired by the Gauss-Newton method, DFO-LS constructs simplified linear regression models for the residuals and allows flexible initialization for expensive problems, whereby it can begin making progress after as few as two objective evaluations. Numerical results show DFO-LS can gain reasonable progress on some medium-scale problems with fewer objective evaluations than is needed for one gradient evaluation. DFO-LS has improved robustness to noise, allowing sample averaging, regression-based model construction, and multiple restart strategies with an auto-detection mechanism. Our extensive numerical experimentation shows that restarting the solver when stagnation is detected is a cheap and effective mechanism for achieving robustness, with superior performance over sampling and regression techniques. The package Py-BOBYQA is a Python implementation of BOBYQA (Powell, 2009), with novel features such as the implementation of robustness to noise strategies. Our numerical experiments show that Py-BOBYQA is comparable to or better than existing general DFO solvers for noisy problems. In our comparisons, we introduce an adaptive accuracy measure for data profiles of noisy functions, striking a balance between measuring the true and the noisy objective improvement.

Keywords: derivative-free optimization, least-squares, trust region methods, stochastic optimization, mathematical software, performance evaluation.

Computing Classification Scheme: Mathematics of computing ∼ Nonconvex optimization

1 Introduction

The ability to solve optimization problems in the absence of derivative information — known as derivative-free optimization (DFO) — is an important goal for optimization software. The need for DFO software particularly arises when function evaluations are expensive (so finite differencing is too costly), or when evaluations are noisy (so the accurate evaluation of derivatives is impossible). A state-of-the-art category of DFO algorithms are the so-called 'model-based' methods. These methods are similar to classical trust-region methods, which require the iterative minimization of local models for the objective over a trust-region ball, except the local models are constructed by interpolation instead of using derivative information. Model-based DFO solvers are known to capture curvature in the objective well [13], and have good practical performance [26].

In this paper, we focus on improving the flexibility and robustness of model-based DFO solvers for two regimes:

Expensive: objectives are expensive to evaluate, and may be noiseless. Here, the goal is to make reasonable progress, not necessarily reaching high accuracy in the solution, using very few evaluations; and,

*Mathematical Institute, University of Oxford, Radcliffe Observatory Quarter, Woodstock Road, Oxford, OX2 6GG, United Kingdom (cartis@maths.ox.ac.uk).

†Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, United Kingdom (jan.fiala@nag.co.uk).

‡Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, United Kingdom (benjamin.marteau@nag.co.uk).

§Mathematical Institute, University of Oxford, Radcliffe Observatory Quarter, Woodstock Road, Oxford, OX2 6GG, United Kingdom (robertsl@maths.ox.ac.uk). This work was supported by the EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in collaboration with the Numerical Algorithms Group Ltd. In compliance with EPSRC's open access initiative, the data in this paper is available from http://dx.doi.org/10.5287/bodleian:KzboJ5E2V.

Noisy: objectives which contain noise, but may be cheap(er) to evaluate. We aim to improve the robustness of the solver — maximizing the amount of progress the solver can make, and hence, the number of problems that can be solved despite the difficulties associated with inaccurate local models and objective evaluations.

Clearly, the two regimes may overlap, in which case we still aim and show that we can make reasonable progress in our proposed algorithms. We are particularly interested in solving unconstrained (or possibly bound-constrained) nonlinear least-squares problems, but also consider general nonlinear objectives.

Regarding the 'expensive' regime, model-based DFO solvers typically require at least $n + 1$ objective evaluations (for an n-dimensional problem) before they can begin the main iteration; this evaluation cost represents the cost of setting up the first local model, from scratch, while subsequent iterations commonly only update the interpolation set and the local model at a much lower evaluation cost[1]. However, in the 'expensive' regime, this start-up cost may be prohibitive, and the user may wish to see decreases in the objective much sooner. Direct search DFO solvers, such as BFO by Porcelli and Toint [28], can make progress with very few objective evaluations, but this flexibility is not generally found in model-based DFO methods.

For the 'noisy' regime, model-based DFO solvers can generally make some progress on a problem, but often stagnate at incorrect solutions, without even using the full computational budget provided by the user; see Figure 1, for instance. Two main methods have been suggested for robustly handling noisy objectives in a model-based DFO context. Sample averaging is the most common approach for handling noisy evaluations; see [15, 16, 36, 8]. For theoretical convergence guarantees to hold, one must compute $\mathcal{O}(\Delta_k^{-4})$ samples of the objective at each point (e.g. [8]), where Δ_k is the trust region radius at iteration k. However, this requirement rapidly becomes infeasible, so $\mathcal{O}(\Delta_k^{-1})$ samples is a more sensible choice in practice [8]. The other main approach is to build regression models (i.e. having more interpolation points than degrees of freedom in the model) rather than interpolation models [10, 5, 8]. We note in particular the STORM algorithm from Chen, Menickelly and Scheinberg [8], which uses $\mathcal{O}(\Delta_k^{-1})$ interpolation points at each iteration k, and determines whether a step gives sufficient objective decrease by averaging over $\mathcal{O}(\Delta_k^{-1})$ samples. In both cases, there is a tradeoff between robustness of the solver and performance in early phases, where the latter is very slow as sampling and regression require a large amount of problem information to accumulate before starting to generate substantial objective improvement.

An alternative approach for the 'noisy' regime is used in SNOWPAC by Augustin and Marzouk [3]. This solver extends a previous model-based DFO code for constrained nonlinear programs by the same authors, NOWPAC [2], by constructing a Gaussian Process surrogate model for the noisy objective, from previously-seen objective values and standard errors. This approach avoids the performance loss in early phases, however it requires the user to provide standard error estimates for each objective evaluation, and introduces potentially expensive surrogate model construction steps, especially when using a large set of observations. Here, we are particularly interested in nonlinear least-squares problems, where we build local models for each residual separately. In this context especially, building surrogate models may prohibitively expensive.

Algorithm development and software contributions In this paper we introduce two new model-based DFO packages in Python, one for nonlinear least-square problems and one for general minimization problems, both with optional bound constraints.

The package for nonlinear least-squares with optional bound constraints, called DFO-LS (Derivative-Free Optimization for Least-Squares), builds on our previous code for nonlinear least-squares, DFO-GN [7], in that it continues to use linear local models for each residual function (rather than quadratic), which reduces the computational cost of the interpolation step. DFO-LS has a wide variety of additional default and optional features, that can be used on their own or in combination, with defaults selected based on extensive testing. These features, apart from averaging and regression sampling, are novel for model-based DFO solvers. The most notable of these features are:

Reduced Initialization Cost: The ability to begin the main iteration after an initial setup cost for the algorithm of as few as 2 objective evaluations (as opposed to at least $n+1$ for an n-dimensional problem in other model-based solvers); thus after this low setup cost, subsequent evaluations are selected based on minimizing a model for the objective, and represent a genuine attempt to progress towards the solution. The mechanism is described in Section 2.1 and its testing in Section 5.1;

[1] In practice, solvers can accept any objective decrease from the initial evaluations [31, 1], and use the best initial point as the first iterate. In our code we do this, but also include a mechanism to begin the main iteration after few initial evaluations, where new iterates are specifically generated to progress towards the solution from the best point so far, rather than just sample the search space.

Multiple Default Parameter Choices: The modification of some algorithm parameters (such as trust-region parameters, termination criteria) to more appropriate values if the objective function is noisy. This is described in Section 3.1 and its numerical impact discussed in Section 6;

Sample Averaging & Regression: The optional use of sample averaging (allowing an extensive range of sampling methodologies) and/or regression-based model construction. These are described in Section 3.3 and their testing in Sections 5.2 and 5.3; and,

Multiple Restarts: The use of multiple restarts to allow greater exploration of the search space for noisy objectives. Although this feature is novel in the model-based DFO setting, similar techniques have been commonly used in numerical analysis, such as multiple restarts of nonlinear conjugate gradient methods [27, Chapter 5] and GMRES [14, Chapter 6], as well as for robustness improvement of the Nelder-Mead algorithm [21]. There are also connections to multistart methods, where several runs of an algorithm are initialized from different starting points, a technique common in global optimization [23, 20], and also used for local derivative-free optimization (e.g. [12] or [1, Example 3.5]). In contrast, our multiple restarts technique only ever uses a single initialization point for a run, and employs the final iterate of a run as the starting point for the restarted run. In our results, we find that multiple restarts greatly enhance the performance of DFO-LS for noisy problems, yielding superior performance even compared to DFO-LS with a high level of sample averaging. In particular, we note that the multiple restarts approach avoids the early loss of performance typical of sample averaging and regression, does not require extra user input common to surrogate model approaches, and is cheap to implement. The restarts mechanism is described in Section 3.2 and its testing in Section 5.4.

The 'reduced initialization cost' feature is designed for the 'expensive' regime; the others are designed for the 'noisy' regime. We additionally demonstrate that these regimes are not mutually exclusive: using a reduced initialization cost works similarly well for noisy problems (as for noiseless problems), and multiple restarts can sometimes improve performance, including escaping local minima, for noiseless problems.

The second package, for general unconstrained minimization with optional bound constraints, is called Py-BOBYQA, as it is a Python implementation of Powell's Bound Optimization BY Quadratic Approximation (BOBYQA) [31]. Py-BOBYQA constructs quadratic local models for the objective, which are based on fully- or under-determined interpolation sets. Some of the above features of DFO-LS are not closely tied to the least-squares problem structure, and so are included in the Py-BOBYQA package. In particular, Py-BOBYQA implements *multiple default parameter choices*, *sample averaging*, and *multiple restarts*. The discussion of the underlying Py-BOBYQA framework and these new algorithmic features, followed by numerical results, are given in Section 7.

Testing Framework Contribution We also propose an improvement to the measurement standards of solver performance for noisy problems. As detailed in [26], data profiles are useful measures for comparing DFO solvers on a standard given test set, which measure the number of objective evaluations required to reach an objective value below a problem- and accuracy-dependent threshold. We assume that a collection of deterministic test problems is used — such as Moré & Wild or CUTEst — and that noisy variants of each problem are created by perturbing the objective or residual functions by multiplicative or additive stochastic noise. In this context, one can check decrease using either the value of the true (noiseless) objective, or the actual (noisy) objective seen by the solver; these two approaches are used, for instance, in [8] and [5] respectively. In this paper, we show that these two measures produce similar results until a problem- and noise-specific accuracy level is reached; beyond this cut-off level, measured performance is better when the 'noisy objective' is used due, most commonly, to successful sampling (rather than optimization). As a result, we propose showing profile results using an adaptive accuracy level; namely, at the desired accuracy level whenever the latter is larger than the per-problem accuracy cut-off, and at the cut-off accuracy level, otherwise. We illustrate that this approach is a fairer approach for testing which focuses on genuine objective reductions rather than 'lucky' sampling errors. We note other testing framework exist, as outlined in [4], such as performance profiles [17] and 'operational zones' [35]. Our framework is described in Section 4.1.

Comparisons to Related Software In our numerical results, we compare DFO-LS to DFO-GN [7] and DFBOLS [38], also designed for nonlinear least-squares problems[2]. We find that using different default parameters for noisy problems, coupled with multiple restarts, makes DFO-LS have substantially improved robustness to noise over both DFO-GN and DFBOLS, without the early loss of performance associated with

[2]There is only one other nonlinear least squares DFO solver that we are aware of, namely, POUNDERS [37]. We have already compared it against DFO-GN and DFBOLS in [7].

sample averaging and regression models. We also find that using a reduced initialization cost for medium-scale problems ($n \approx 100$ dimensions) allows DFO-LS to make reasonable progress on some problems with fewer than n objective evaluations, but with a slight performance penalty for medium-sized budgets.

As mentioned above, the general-objective solver Py-BOBYQA is based on the original package by Powell [31, 39]. In our testing, we compare Py-BOBYQA with the original BOBYQA, together with (S)NOWPAC [2, 3], and our own implementation[3] of STORM [8]. In our testing for noisy problems, we find that the different default parameters and multiple restarts in Py-BOBYQA means it substantially outperforms BOBYQA. It achieves a similar or better level of robustness than SNOWPAC and STORM, but with a mechanism which is cheap to implement and does not penalize performance in early phases.

Software Availability The two Python packages in this paper, DFO-LS and Py-BOBYQA, are available on Github[4]. They are released under the GNU General Public License.

Paper Structure The roadmap of the paper is as follows:

1. We first describe the DFO-LS package for nonlinear least-squares minimization, including the underlying algorithmic framework in Section 2, key details of the implementation and a full description of the new software features in Section 3. The description of the reduced initialization cost mechanism is in Section 2.1, as it relates to the basic model construction procedure, multiple default parameter choices are described in Section 3.1, multiple restarts are described in Section 3.2, and sample averaging & regression are described in Section 3.3;

2. Section 4 outlines the framework for our numerical testing, including the use of problem-dependent accuracy levels to isolate algorithm performance from genuine progress rather than sampling errors in Section 4.1;

3. Section 5 contains our numerical studies of DFO-LS. Here, we analyse the performance of the new features, and demonstrate how the default options for DFO-LS were selected. We consider the reduced initialization mechanism (Section 5.1), sample averaging & regression (Sections 5.2 and 5.3), and multiple restarts (Section 5.4);

4. The last component devoted to DFO-LS is Section 6, where we compare its performance against other derivative-free nonlinear least-squares solvers;

5. Finally, in Section 7, we introduce the Py-BOBYQA package for general minimization with optional bound constraints, discuss how the new features from DFO-LS (multiple default parameter choices, sample averaging and multiple restarts) are incorporated into the implementation, and demonstrate Py-BOBYQA's performance against other derivative-free solvers.

We summarize our results and conclude in Section 8.

2 General Algorithmic Framework

The DFO-LS software is designed to solve the nonlinear least-squares problem[5]

$$\min_{\mathbf{x}\in\mathbb{R}^n} \; f(\mathbf{x}) := \|\mathbf{r}(\mathbf{x})\|^2 = \sum_{i=1}^{m} r_i(\mathbf{x})^2, \tag{2.1}$$

where $\mathbf{r}(\mathbf{x}) := [r_1(\mathbf{x}) \; \cdots \; r_m(\mathbf{x})]^\top$ is a continuously differentiable function from $\mathbb{R}^n$ to $\mathbb{R}^m$, but its Jacobian matrix of first derivatives is unavailable. Both the case when $m \geq n$ (least-squares) and $m \leq n$ (inverse problems) are allowed. Lastly, we use $\|\cdot\|$ for the 2-norm of vectors and matrices (i.e. Euclidean norm and largest singular value respectively) unless otherwise specified, and for $\mathbf{x} \in \mathbb{R}^n$ and $\Delta > 0$, we define $B(\mathbf{x}, \Delta) := \{\mathbf{y} \in \mathbb{R}^n : \|\mathbf{y} - \mathbf{x}\| \leq \Delta\}$.

[3] There are several versions of the STORM algorithm given for different noise settings. We use the version of STORM designed for unbiased noise, which builds regression models from independent samples at every iteration, because it showed better performance than other variants.

[4] See `https://github.com/numericalalgorithmsgroup/dfols` (DOI:10.5281/zenodo.2630426) and `https://github.com/numericalalgorithmsgroup/pybobyqa` (DOI:10.5281/zenodo.2630437) respectively. Versions 1.0.1 of both packages were used for all the testing below.

[5] Note that in line with the implementation of DFO-LS, we do not have a constant 1/2 factor in (2.1).

2.1 Regression Interpolation Models

DFO-LS constructs a linear model for $\mathbf{r}(\mathbf{x})$ in a neighbourhood of the current iterate $\mathbf{x}_k$ at every iteration. To achieve this in a derivative-free way, we maintain a set of $p+1$ points $Y_k = \{\mathbf{x}_k, \mathbf{y}_1, \ldots, \mathbf{y}_p\} \subset \mathbb{R}^n$, where we let $\mathbf{y}_0 := \mathbf{x}_k$ for notational convenience. The usual regime has $p \geq n$, but if needed, we also allow $p < n$ in early iterations in order to reduce the initial evaluation cost of DFO-LS; both constructions are described here. The default option in DFO-LS is to initialize with a full set of $n+1$ points (so $p = n$).

When $p \geq n$, we build a model

$$\mathbf{r}(\mathbf{x}_k + \mathbf{s}) \approx \mathbf{m}_k(\mathbf{s}) := \mathbf{r}_k + J_k\mathbf{s}, \tag{2.2}$$

by solving the regression problem

$$\min_{\mathbf{r}_k, J_k} \sum_{t=0}^{p} \|\mathbf{m}_k(\mathbf{y}_t - \mathbf{x}_k) - \mathbf{r}(\mathbf{y}_t)\|^2. \tag{2.3}$$

This corresponds to finding the least-squares solutions to the overdetermined linear systems

$$W_k \begin{bmatrix} r_{k,i} \\ \mathbf{j}_{k,i} \end{bmatrix} := \begin{bmatrix} 1 & (\mathbf{y}_0 - \mathbf{x}_k)^\top \\ \vdots & \vdots \\ 1 & (\mathbf{y}_p - \mathbf{x}_k)^\top \end{bmatrix} \begin{bmatrix} r_{k,i} \\ \mathbf{j}_{k,i} \end{bmatrix} = \begin{bmatrix} r_i(\mathbf{y}_0) \\ \vdots \\ r_i(\mathbf{y}_p) \end{bmatrix}, \tag{2.4}$$

for all $i = 1, \ldots, m$, where $r_{k,i}$ and $\mathbf{j}_{k,i}^\top$ are the i-th entry of $\mathbf{r}_k$ and row of J_k respectively. The matrix W_k has full column rank whenever the set $\{\mathbf{y}_1 - \mathbf{x}_k, \ldots, \mathbf{y}_p - \mathbf{x}_k\}$ spans $\mathbb{R}^n$; we ensure this in DFO-LS by calling procedures to improve the geometry of Y_k (in a specific sense discussed below). However, as the algorithm progresses, the points $\mathbf{y}_t$ get progressively closer to $\mathbf{x}_k$, so W_k becomes ill-conditioned. To avoid this issue, we precondition (2.4) by scaling the second through last columns of W_k by α_k^{-1}, where $\alpha_k := \max_{t=1,\ldots,p} \|\mathbf{y}_t - \mathbf{x}_k\|$.

Once we have built the vector model $\mathbf{m}_k$ (2.2), we construct a quadratic model m_k for the full objective $f(\mathbf{x})$ in the obvious way, by defining

$$f(\mathbf{x}_k + \mathbf{s}) \approx m_k(\mathbf{s}) := \|\mathbf{m}_k(\mathbf{s})\|^2 = \|\mathbf{r}_k\|^2 + \mathbf{g}_k^\top \mathbf{s} + \frac{1}{2}\mathbf{s}^\top H_k \mathbf{s}, \tag{2.5}$$

where $\mathbf{g}_k := 2J_k^\top \mathbf{r}_k$ and $H_k := 2J_k^\top J_k$.

This approach is similar to the DFO-GN algorithm, but our slightly different formulation of the interpolation problem (2.3) is designed to allow improved robustness for noisy problems, and reduce the initialization cost of the algorithm.

Remark 2.1. An alternative interpolation framework which we considered, inspired by a comment in [11, Chapter 4], designed to balance accuracy of interpolation against large changes in the model between iterations, was to replace (2.3) with

$$\min_{\mathbf{r}_k, J_k} \|J_k - J_{k-1}\|_F^2 + \lambda_k \sum_{t=0}^{p} \|\mathbf{m}_k(\mathbf{y}_t - \mathbf{x}_k) - \mathbf{r}(\mathbf{y}_t)\|^2, \tag{2.6}$$

where $\lambda_k > 0$ is an algorithm parameter. This idea of allowing inexact interpolation was motivated by the case of noisy objective evaluation. However, our extensive testing showed that the best results for this framework, even for noisy objectives, required setting λ_k very large (at least 10^{10}), which means that we are essentially solving (2.3).

Reduced Initialization Cost for Expensive Objectives The interpolation problem (2.3) requires $p \geq n$, so that the system (2.4) is square or overdetermined. This means that before the first model can be constructed, we must evaluate the objective at $p+1$ points — this is common in model-based DFO algorithms. Although these evaluations may be parallelized, a user may not have the ability to do this, and the cost of these evaluations may be prohibitive. In such settings, DFO-LS can proceed with a reduced initialization cost, constructing the model (2.2) using as few as 2 objective evaluations. Since this mechanism is designed for the regime where objective evaluations are expensive, we assume that this mechanism is not used for runs of the algorithm with overdetermined/regression models; that is, interpolation models are used, so $p = n$ and (2.4) is a square system.

Suppose we have evaluated the objective at $p+1$ affinely-independent points $\{\mathbf{y}_0, \ldots, \mathbf{y}_p\}$ with $\mathbf{y}_0 := \mathbf{x}_k$, where we now assume $1 \leq p < n$. We construct $\mathbf{m}_k$ by solving the same interpolation system (2.4), which

is now underdetermined, and for which we select the minimal (Euclidean) norm solution — linear model construction in this way was analyzed in [32]. The resulting $\mathbf{r}_k$ and J_k are solutions to[6]

$$\min_{\mathbf{r}_k, J_k} \|\mathbf{r}_k\|^2 + \alpha_k \|J_k\|_F^2 \qquad \text{s.t.} \qquad \mathbf{m}_k(\mathbf{y}_t - \mathbf{x}_k) = \mathbf{r}(\mathbf{y}_t), \quad \forall t = 0, \ldots, p, \tag{2.7}$$

where α_k, defined above, is the column scaling used to precondition (2.4).

However, the construction (2.7) is not ideal, because, as proven in Lemma 2.2 below, the resulting J_k is not full rank, so the models $\mathbf{m}_k$ and m_k are not full-dimensional; that is, there are directions along which these are constant, regardless of the objective.

Lemma 2.2. *Suppose* $\mathbf{m}_k$ (2.2) *is constructed using* (2.7) *with* $p < n$*, and where* $\{\mathbf{y}_0, \ldots, \mathbf{y}_p\}$ *are affinely independent. Then* J_k *has column rank at most* p*.*

Proof. The solution of (2.7) is the minimal norm solution for system (2.4). Using $\mathbf{y}_0 = \mathbf{x}_k$, we write W_k in (2.4) as

$$W_k = \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{e} & L_k \end{bmatrix}, \qquad \text{where} \qquad L_k := \begin{bmatrix} (\mathbf{y}_1 - \mathbf{x}_k) & \cdots & (\mathbf{y}_p - \mathbf{x}_k) \end{bmatrix}^\top \in \mathbb{R}^{p \times n}, \tag{2.8}$$

and $\mathbf{e} \in \mathbb{R}^p$ is the vector of ones. Since the interpolation points are affinely independent, $L_k^\top$ has full column rank p, so we have the QR factorization $L_k^\top = \widehat{Q}\widehat{R}$, where $\widehat{Q} \in \mathbb{R}^{n \times p}$ has columns which are an orthonormal basis for $\text{col}(L_k^\top)$ and $\widehat{R} \in \mathbb{R}^{p \times p}$ is invertible and upper triangular. Then the minimal-norm solution to (2.4) is given by

$$\begin{bmatrix} r_{k,i} \\ \mathbf{j}_{k,i} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & \widehat{Q} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{e} & \widehat{R}^\top \end{bmatrix}^{-1} \begin{bmatrix} r_i(\mathbf{y}_0) \\ \vdots \\ r_i(\mathbf{y}_p) \end{bmatrix}. \tag{2.9}$$

That is, $\mathbf{j}_{k,i} \in \text{col}(\widehat{Q})$, so J_k can be written as $J_k = \widehat{J}_k \widehat{Q}^\top$ for some $\widehat{J}_k \in \mathbb{R}^{m \times p}$, and thus J_k has column rank at most p. □

Thus by using this model, we will not in general be able to find a solution. A simple way to address this issue would be to, at each iteration, replace one point with the new iterate $\mathbf{x}_{k+1}$ using standard methods, then add another point to the interpolation set, chosen to increase the dimension of the model (until a full-dimensional model is achieved). However, this would require two objective evaluations per iteration, which may be wasteful when evaluations are expensive.

Making the Jacobian have full rank in the expensive regime Instead, after calculating the rank-deficient J_k, DFO-LS makes it full rank — and hence makes the model full-dimensional — by increasing its $n - p$ smallest singular values $\sigma_{p+1} = \cdots = \sigma_n = 0$ to the level of the smallest nonzero singular value $\sigma_p > 0$; this requires the calculation of the SVD of J_k. To handle the case when J_k has rank strictly less then p, we also floor all singular values at a small positive value (default 10^{-6}), to ensure the model is always full-dimensional; this means that some interpolation conditions may not be satisfied, but only during this initialization phase — once a full set of $n + 1$ interpolation points is available, we no longer perturb the singular values of J_k. Since J_k has $\min(m, n)$ singular values, if $m < n$ then this SVD-based approach still produces a model which is constant in some directions, which is not desirable. Thus the SVD-based mechanism is only used in DFO-LS when $m \geq n$.

Alternative mechanism for expanding the search space DFO-LS has another optional mechanism for increasing the dimension of the model, instead of perturbing the singular values of J_k. In this approach, after finding the trust region step $\mathbf{s}_k$, we replace the new candidate point $\mathbf{x}_k + \mathbf{s}_k$ with the perturbed point $\mathbf{x}_k + \mathbf{s}_k + \mathbf{d}_k$, where $\mathbf{d}_k$ is a random direction orthogonal to our current set of search directions (with length a constant multiple of Δ_k). This mechanism is the default in DFO-LS for inverse problems with $m < n$.

We compare these two approaches in Section 5.1, and conclude that the SVD-based variant has similar performance to the random direction extension for small budgets, but better performance for longer budgets. Thus the SVD approach is chosen as the default in DFO-LS when an initialization with less than n interpolation points is used, provided $m \geq n$ (and the alternative mechanism is used only when $m < n$).

2.2 Core Algorithmic Framework

Trust Region Framework The general algorithmic framework of DFO-LS is that of trust region methods [9]. In these methods we maintain a radius parameter $\Delta_k > 0$, and say that we expect m_k (2.5) to be a good approximation for f in $B(\mathbf{x}_k, \Delta_k)$, the so-called 'trust region'.

[6] Note that because in this phase of the algorithm we never remove points from Y_k, provided $\{\mathbf{y}_t - \mathbf{x}_k : t = 1, \ldots, p\}$ is linearly independent, (2.7) is equivalent to minimizing the change in the model, $\min_{\mathbf{r}_k, J_k} \|\mathbf{r}_k - \mathbf{r}_{k-1}\|^2 + \alpha_k \|J_k - J_{k-1}\|_F^2$.

At each step in the algorithm, we construct m_k and calculate a step by solving the trust region subproblem

$$\mathbf{s}_k \approx \arg\min_{\|\mathbf{s}\| \leq \Delta_k} m_k(\mathbf{s}). \tag{2.10}$$

Efficient algorithms exist for solving (2.10) approximately (e.g. [9]); we use the approach from [31], based on the conjugate gradient method. Having calculated a step $\mathbf{s}_k$, we evaluate $f(\mathbf{x}_k + \mathbf{s}_k)$. If this step produces a sufficient decrease in the objective, in the sense that

$$r_k := \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{s}_k)}, \tag{2.11}$$

is sufficiently large, then we accept the step (i.e. set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$) and increase Δ_k. If the step does not produce sufficient decrease, then we reject the step (i.e. $\mathbf{x}_{k+1} = \mathbf{x}_k$) and decrease Δ_k. In our derivative-free setting, we then need to update Y_k to ensure it includes the (possibly new) point $\mathbf{x}_{k+1}$, and where necessary, move points in Y_k to improve its geometry.

Geometric Considerations When developing model-based DFO methods, it is well-known (e.g. [34]) that one needs to take steps to keep the geometry of Y_k 'good', and prevent degeneracy.

For the linear regression models in DFO-LS, the notion of 'good' was defined by Conn, Scheinberg and Vicente [10]. First, we define the regression Lagrange polynomials of Y_k, as the linear functions $\{\Lambda_0(\mathbf{y}), \ldots, \Lambda_p(\mathbf{y})\}$ given by

$$\Lambda_t(\mathbf{y}) := c_t + \mathbf{g}_t^\top(\mathbf{y} - \mathbf{x}_k), \quad \text{where } c_t \text{ and } \mathbf{g}_t \text{ solve} \quad \min_{c_t, \mathbf{g}_t} \sum_{s=0}^{p} \left(\Lambda_t(\mathbf{y}_s) - \delta_{s,t}\right)^2. \tag{2.12}$$

These polynomials exist and are unique whenever W_k (2.4) has full column rank [10]. Given these Lagrange polynomials, the measure of the quality of Y_k is given by the following definition.

Definition 2.3 (Λ-poised, regression sense)**.** For $B \subset \mathbb{R}^n$ and $\Lambda > 0$, the set Y_k with $|Y_k| = p+1$ is Λ-poised in B in the regression sense if $p \geq n$ and

$$\max_{t=0,\ldots,p} \max_{\mathbf{y} \in B} |\Lambda_t(\mathbf{y})| \leq \Lambda, \tag{2.13}$$

for all $\mathbf{y} \in B$, where $\{\Lambda_0(\mathbf{y}), \ldots, \Lambda_p(\mathbf{y})\}$ are the regression Lagrange polynomials for Y_k.

A similar definition holds for exact (i.e. non-regression) interpolation when $p = n$; see [7] for details. As in that case, a small value of Λ indicates that the geometry of Y_k is 'good'. The steps we use in DFO-LS to improve the Λ-poisedness of Y_k are outlined in Section 3.1, and the details of how Λ-poisedness leads to good regression models are given in Appendix A.

No geometry-improving steps are allowed in the early iterations of the expensive regime, while $p < n$.

2.3 DFO-LS Algorithm

A full statement of the DFO-LS algorithm is given in Algorithm 1. The overall structure of DFO-LS builds upon that of DFO-GN [7], with a key difference being the flexible model construction, which allows us to have a reduced initialization cost when evaluations are expensive (when (2.4) is underdetermined), and to implement regression models when the problem is noisy (when (2.4) is overdetermined). For the latter regime, the most efficient contribution of DFO-LS is the multiple restarts feature described in Section 3.2. Other key features of DFO-LS are described in Section 3.1, and optional features for noisy problems (such as regression and sampling) are described in Section 3.3.

We note that DFO-LS uses a standard trust region framework, but maintains two different measures of trust region radius: the usual Δ_k as described in Section 2.2, and a lower bound $\rho_k \leq \Delta_k$. Originally a feature from Powell [29], this is used to ensure that we do not decrease Δ_k too much until we are confident that the geometry of Y_k is sufficiently good; i.e. that unsuccessful steps (where $r_k < \eta_1$) are not because of a poor quality model, but because the nature of the objective near $\mathbf{x}_k$ requires a small trust region in order to make good progress.

A summary of the convergence guarantees of DFO-LS is given in Appendix A.

3 New Algorithmic Features

In this section, we describe the general features of DFO-LS and several new features for handling noisy objectives.

Algorithm 1 DFO-LS: Derivative-Free Optimization for Least-Squares.

Input: Starting point $\mathbf{x}_0 \in \mathbb{R}^n$, initial trust region radius $\Delta_0 > 0$ and integers p_{init} and p, the sizes of the initial and final interpolation sets, respectively, where $1 \leq p_{init} \leq p$ and $p \geq n$.

Parameters: maximum trust region radius $\Delta_{max} \geq \Delta_0$, minimum trust region radius $0 < \rho_{end} < \Delta_0$, trust region radius scalings $0 < \gamma_{dec} < 1 < \gamma_{inc} \leq \overline{\gamma}_{inc}$ and $0 < \alpha_1 < \alpha_2 < 1$, acceptance thresholds $0 < \eta_1 \leq \eta_2 < 1$, safety reduction factor $0 < \omega_S < 1$, safety step threshold $0 < \gamma_S < 1$, and Boolean flag `NOISY` for the presence of noise in the objective.

1: Build an initial interpolation set $Y_0 \subset B(\mathbf{x}_0, \Delta_0)$ of size $p_{init} + 1$, with $\mathbf{x}_0 \in Y_0$. Set $\rho_0 = \Delta_0$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: **if** `NOISY` **and** *all values* $\{f(\mathbf{y}) : \mathbf{y} \in Y_k\}$ *are within noise level of* $f(\mathbf{x}_k)$ **then**
4: Call restart (set $\Delta_{k+1} = \rho_{k+1} = \Delta_0$ and build Y_{k+1} as per Section 3.2) and **goto** line 6.
5: **end if**
6: Given $\mathbf{x}_k$ and Y_k, construct the model $\mathbf{m}_k(\mathbf{s})$ (2.2) by solving the interpolation problem (2.7) if $|Y_k| < p + 1$, otherwise (2.4).
7: Form the full model m_k (2.5), and approximately solve the trust region subproblem (2.10) to get a step $\mathbf{s}_k$.
8: **if** $\|\mathbf{s}_k\| < \gamma_S \rho_k$ **then**
9: **if** $|Y_k| < p + 1$ **then**
10: Safety Phase (Growing): Form $Y_{k+1} = Y_k \cup \{\mathbf{x}_k + \mathbf{s}\}$ for some $\mathbf{s}$ orthogonal to $\{\mathbf{y} - \mathbf{x}_k : \mathbf{y} \in Y_k\}$ with $\|\mathbf{s}\| = \Delta_k$.
11: Set $(\rho_{k+1}, \Delta_{k+1}) = (\rho_k, \Delta_k)$.
12: **else**
13: Safety Phase: Set $\mathbf{x}_{k+1} = \mathbf{x}_k$ and $\Delta_{k+1} = \max(\rho_k, \omega_S \Delta_k)$, and form Y_{k+1} by improving the geometry of Y_k.
14: If $\Delta_{k+1} = \rho_k$, set $(\rho_{k+1}, \Delta_{k+1}) = (\alpha_1 \rho_k, \alpha_2 \rho_k)$, otherwise set $\rho_{k+1} = \rho_k$.
15: If $\rho_{k+1} \leq \rho_{end}$: call restart if `NOISY`, else **terminate**.
16: **end if**
17: **goto** line 6.
18: **end if**
19: Evaluate $\mathbf{r}(\mathbf{x}_k + \mathbf{s}_k)$ and calculate ratio r_k (2.11).
20: Accept/reject step and update trust region radius: set

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}_k + \mathbf{s}_k, & r_k \geq \eta_1, \\ \mathbf{x}_k, & r_k < \eta_1, \end{cases} \quad \text{and} \quad \Delta_{k+1} = \begin{cases} \min(\max(\gamma_{inc}\Delta_k, \overline{\gamma}_{inc}\|\mathbf{s}_k\|), \Delta_{max}), & r_k \geq \eta_2, \\ \max(\gamma_{dec}\Delta_k, \|\mathbf{s}_k\|, \rho_k), & \eta_1 \leq r_k < \eta_2, \\ \max(\min(\gamma_{dec}\Delta_k, \|\mathbf{s}_k\|), \rho_k), & r_k < \eta_1. \end{cases} \tag{2.14}$$

21: **if** $|Y_k| < p + 1$ **then**
22: Growing Phase: Form $Y_{k+1} = Y_k \cup \{\mathbf{x}_k + \mathbf{s}_k\}$ and set $\rho_{k+1} = \rho_k$.
23: **else if** $r_k \geq \eta_1$ **then**
24: Successful Phase: Form $Y_{k+1} = Y_k \cup \{\mathbf{x}_{k+1}\} \setminus \{\mathbf{y}\}$ for some $\mathbf{y} \in Y_k$ and set $\rho_{k+1} = \rho_k$.
25: If objective decrease is too slow: call restart if `NOISY`, else **terminate**.
26: **else if** `NOISY` **and** *restart auto-detected* **then**
27: Restart Auto-Detection: Call restart.
28: **else if** *geometry of* Y_k *is not good* **then**
29: Model Improvement Phase: Improve the geometry of Y_{k+1} and set $\rho_{k+1} = \rho_k$.
30: **else**
31: Unsuccessful Phase: Set $Y_{k+1} = Y_k$, and if $\Delta_{k+1} = \rho_k$, set $(\rho_{k+1}, \Delta_{k+1}) = (\alpha_1 \rho_k, \alpha_2 \rho_k)$, otherwise set $\rho_{k+1} = \rho_k$.
32: If $\rho_{k+1} \leq \rho_{end}$: call restart if `NOISY`, else **terminate**.
33: **end if**
34: **end for**

3.1 General Features of DFO-LS Implementation

We describe how some of the steps in Algorithm 1 are performed in practice. The majority of these general features are inherited from DFO-GN [7], but DFO-LS includes variable scaling, two new termination criteria, different default trust region parameters for noisy problems, and a slightly different approach for determining the initial set Y_0.

Geometry-Improving Steps The goal of the geometry-improving steps in Algorithm 1 is to enhance the quality of the model m_k; specifically, we wish to make Y_k Λ-poised in $B(\mathbf{x}_k, \Delta_k)$, so m_k is a fully linear model for f in the trust region. However, as mentioned above, guaranteeing the Λ-poisedness of Y_k if $|Y_k| > n+1$ is not straightforward. If $|Y_k| = n+1$, then we can achieve Λ-poised via the iteration [11, Algorithm 6.3]

1. Select the point $\mathbf{y}_t \in Y_k$ ($\mathbf{y}_t \neq \mathbf{x}_k$) for which $\max_{\mathbf{y} \in B(\mathbf{x}_k, \Delta_k)} |\Lambda_t(\mathbf{y})|$ is maximized;
2. Replace $\mathbf{y}_t$ in Y_k with $\mathbf{y}^+$, where
$$\mathbf{y}^+ = \underset{\mathbf{y} \in B(\mathbf{x}_k, \Delta_k)}{\arg\max} |\Lambda_t(\mathbf{y})|, \tag{3.1}$$
and repeat until Y_k is Λ-poised.

As in DFO-GN, in practice we perform a simplified geometry-improving phase: we simply choose $\mathbf{y}_t \in Y_k$ to be the point furthest from $\mathbf{x}_k$, and replace it with $\mathbf{y}^+$ as defined by (3.1). We do not repeat this process; only one point is moved per call of the geometry-improving phase.

Similarly, we use a simplified test to determine if the geometry of Y_k needs improving at all. In theory, we need to check if Y_k is Λ-poised. Instead, we say that the geometry of Y_k needs improving if $\max_t \|\mathbf{y}_t - \mathbf{x}_k\| > \epsilon$, for some threshold ϵ, usually a constant multiple of Δ_k or ρ_k.

We note that in DFO-LS not all interpolation points will be inside the trust region $B(\mathbf{x}_k, \Delta_k)$. This is not required for Λ-poisedness (Definition 2.3), but it is generally desirable that interpolation points remain close to $\mathbf{x}_k$. We ensure this in two ways: firstly, by using $\max_t \|\mathbf{y}_t - \mathbf{x}_k\| > \epsilon$ (with ϵ a multiple of Δ_k or ρ_k) as the test for calling geometry-improvement. Secondly, when selecting an interpolation point to replace (line 24 of Algorithm 1), we remove points which are far from $\mathbf{x}_k$ and/or which would give the best improvement to the Λ-poisedness of the interpolation set. More details on this process are given in [7, Section 4.2].

Model Updating In Algorithm 1, we only add $\mathbf{x}_k + \mathbf{s}_k$ to Y_k in successful steps. However in practice, like in DFO-GN, we always incorporate new information when it becomes available, and so we update $Y_{k+1} = Y_k \cup \{\mathbf{x}_k + \mathbf{s}_k\} \setminus \{\mathbf{y}_t\}$ for some $\mathbf{y}_t \neq \mathbf{x}_{k+1}$ at every iteration, successful or otherwise. Similarly, we always choose to centre our trust region at the best value found so far, so we ensure $\mathbf{x}_k = \arg\min_{\mathbf{y}_t \in Y_k} f(\mathbf{y}_t)$ at every iteration — this optimal point (so far) can come from a trust region step, or even from a geometry-improving phase.

Given a point to add to Y_k (to form Y_{k+1}), we use the method from DFO-GN for determining which point it should replace. This method uses a criterion which chooses to remove points which are far from $\mathbf{x}_k$ and for which the replacement would most improve the geometry of Y_{k+1}.

Inclusion of Bound Constraints and Variable Scaling The implementation of DFO-LS solves problems with optional bound constraints. That is, it solves (2.1) subject to $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$. The only changes to Algorithm 1 required for this are in the calculation of the trust region step (2.10) and geometry-improving step (3.1), which now also have bound constraints.

For the calculation of a trust region step subject to bound constraints, we use the routine `TRSBOX` from BOBYQA [31], as modified by Zhang et al. in DFBOLS [38]. Geometry-improving steps with bound constraints are calculated using [7, Algorithm 3].

Because bound constraints can often provide information about the natural scaling of a problem, DFO-LS allows the optional internal scaling of variables based on the bound constraints, to reduce the likelihood of ill-conditioning. If this is used, we internally shift and scale the inputs so that the new feasible region is $\mathbf{x} \in [0,1]^n$.

Termination Criteria There are four ways in which DFO-LS can terminate. The first three are inherited from DFO-GN:

- Small objective value: since we have the lower bound $f \geq 0$ always, we allow termination when $f(\mathbf{x}_k) \leq \max\{\epsilon_{abs}, \epsilon_{rel} f(\mathbf{x}_0)\}$, for user-specified parameters ϵ_{abs} and ϵ_{rel}. Having this feature is especially useful for DFO solvers, when often just achieving some desired decrease in the objective is the goal, rather than solving to full optimality (e.g. when function evaluations are expensive);

- Small trust region: we know that $\rho_k \to 0$ as $k \to \infty$ [7, Lemma 3.11], so we terminate when $\rho_k \leq \rho_{end}$; and
- Computational budget: we terminate after a given number of evaluations of the objective.

The last two of these are designed to cause termination after a sufficient number of unsuccessful steps. The first criteria is triggered by successful steps, but is likely to be triggered only for zero-residual problems.

To ensure a timely termination based on successful steps, we introduce an extra criterion, similar to the "$f_i^{*\prime}$ test" of Larson and Wild [22]. We define a successful iteration as 'slow' if the last K successful iterations have produced an average reduction in $\log(f(\mathbf{x}_k))$ below a given threshold. That is, if $\{k_i : i \in \mathbb{N}\}$ are the successful iterations, then iteration k_i is 'slow' if

$$\frac{\log(f(\mathbf{x}_{k_{(i-K)}})) - \log(f(\mathbf{x}_{k_i}))}{K} < \epsilon, \tag{3.2}$$

for some value $\epsilon > 0$. Note that since we are only considering successful iterations, $f(\mathbf{x}_{k_i})$ will be the best objective value found up to iteration k_i. Our termination condition is then: quit after successful iteration k_i if $\{k_{i-N+1}, \ldots, k_{i-1}, k_i\}$ were all 'slow', for some $N \in \mathbb{N}$.

Lastly, DFO-LS also includes an optional noise-aware termination condition. Specifically, we terminate if all function values $f(\mathbf{y}_t)$ are within some user-provided 'noise level' of $f(\mathbf{x}_k)$. That is, for all $t = 1, \ldots, p$, either

$$|f(\mathbf{y}_t) - f(\mathbf{x}_k)| \leq \text{const} \cdot \frac{\epsilon}{\sqrt{N_t}} \qquad \text{or} \qquad \left|\frac{f(\mathbf{y}_t)}{f(\mathbf{x}_k)}\right| \leq \text{const} \cdot \frac{\epsilon}{\sqrt{N_t}}, \tag{3.3}$$

where N_t is the number of samples used to estimate the value $f(\mathbf{y}_t)$; see Section 3.2 for details. Which of these criteria is used depends on whether the user has specified ϵ as an additive or multiplicative noise level in the evaluation of f, and the value of 'const' is also user-provided (default is 1).

Default Parameters for Noisy Problems One of the main problem types that DFO-LS is designed to solve is where objective evaluations are noisy. In this situation, the set of default parameters — which are designed for smooth objectives — are not necessarily good choices.

The most notable examples of this are the parameters which govern decreases of Δ_k and ρ_k, namely γ_{dec}, α_1 and α_2 (default values 0.5, 0.1 and 0.5 respectively). When we have noisy evaluations, it is common to get unsuccessful iterations even when a step $\mathbf{s}_k$ is useful, because the noise in the objective evaluation leads to inaccuracies in the calculated r_k (2.11). This then leads to unnecessary reductions in the trust region radius, causing the algorithm to progress more slowly, and potentially terminate too early.

In DFO-LS, we allow the user to specify if their objective evaluation is noisy, and consequently modify the default values for several algorithm parameters. Note that the user can choose to override any parameter value by specifying it directly, even if the default has been modified. For example, the 'noisy problem' default values of γ_{dec}, α_1 and α_2 are 0.98, 0.9 and 0.95 respectively.

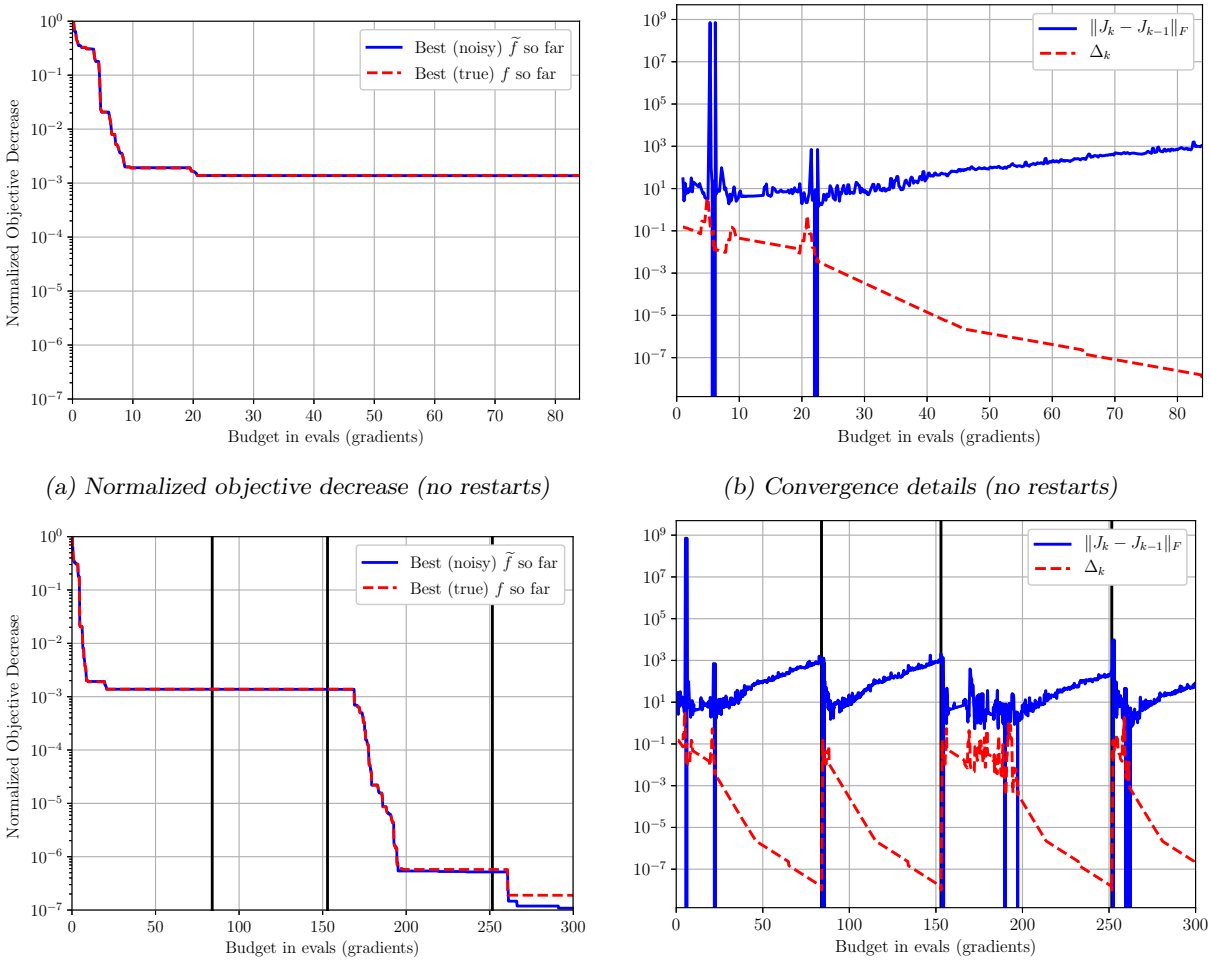
Other Differences There are other small differences between the implementation of DFO-LS and Algorithm 1, which are inherited from DFO-GN; a list of these may be found in [7, Section 4.4].

We also change the default method for constructing the initial set Y_0. In DFO-GN, like DFBOLS [38] and BOBYQA [31], the initial set is typically taken to be $\mathbf{x}_0 \pm \Delta_0 \mathbf{e}_t$ for coordinate vectors $\mathbf{e}_t$ (adjusted when for bound constraints and for more than $2n + 1$ interpolation points). In DFO-LS, the default mechanism is to use $\mathbf{x}_0 \pm \Delta_0 \mathbf{q}_t$ for random orthonormal vectors $\mathbf{q}_t$ (again, adjusted in the case of bound constraints or $p > 2n$).

3.2 Multiple Restarts for Noisy Objectives

To improve robustness to noisy objectives, DFO-LS uses a multiple restarts mechanism. As motivation, note that, in DFO trust-region methods, Δ_k tends to zero as a measure of convergence; see for example, [7, Lemma 3.11] and Theorem A.7 for the deterministic case. However, when the function is noisy, as Δ_k gets small, the interpolation points get very close together and the corresponding objective values are all within noise level. As a result, Δ_k no longer reflects convergence and the solver can stagnate in a suboptimal region.

An illustrative example. We may see this effect by considering a test problem. In Figure 1, we compare two runs of DFO-LS — with and without multiple restarts — for the Osborne 1 test problem [26, Problem 36], where we have added unbiased multiplicative Gaussian noise with $\sigma = 10^{-2}$. After making some initial progress, the run without restarts has many unsuccessful steps, and Δ_k shrinks as the solver attempts to



(a) Normalized objective decrease (no restarts)

(b) Convergence details (no restarts)

(c) Normalized objective decrease (with restarts)

(d) Convergence details (with restarts)

Figure 1. Normalized objective decrease achieved by DFO-LS, measured in both the noisy and true objective, and convergence information for test problem 'Osborne 1' with $(n, m) = (5, 33)$ and unbiased multiplicative Gaussian noise of size $\sigma = 10^{-2}$. The vertical black lines indicate when restarts occurred. Restart type was 'soft (moving $\mathbf{x}_k$)', and the budget was $300(n + 1)$ evaluations; the (a) and (b) runs terminated early on small trust-region radius.

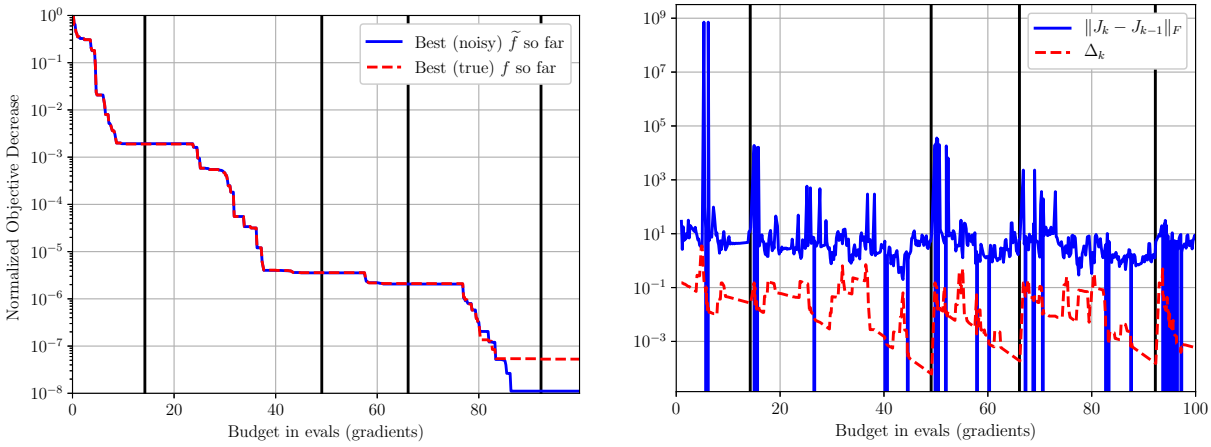
find a descent direction. When this happens, the interpolated Jacobian J_k begins to change substantially at each iteration. This indicates that the noise in the interpolation problem is dominating the true descent information.

When we introduce restarts, the stagnation can eventually be overcome. When a restart occurs (and we increase Δ_k to its original level), the changes in J_k reduce quickly, and so the interpolation is more likely to capture genuine information about changes in the objective. As a result, the solver is able to progress, and ultimately finds a much higher accuracy solution. □

In DFO-LS, a restart may be triggered by all the termination criteria, except for small objective and maximum computational budget. At its simplest, a restart involves increasing Δ_k to a much larger value, and possibly moving some of the points in Y_k. There are two main types of restart which DFO-LS can perform:

Hard restart: Reset the trust region radius to $\Delta_k = \rho_k = \Delta_0$, and rebuild Y_k in the new (larger) trust region $B(\mathbf{x}_k, \Delta_k)$ from scratch using the same mechanism as how Y_0 was originally constructed in the case $p = n$ (see Section 3.1); and,

Soft restart (moving $\mathbf{x}_k$): Reset the trust region radius to $\Delta_k = \rho_k = \Delta_0$, and save the current best point $\mathbf{x}_k$ separately. Then, move $\mathbf{x}_k$ to a geometry-improving point in the new trust region $B(\mathbf{x}_k, \Delta_k)$, shifting the trust region to this new point. Finally, move the $N - 1 < p$ points in Y_k which were closest to the old value of $\mathbf{x}_k$ to geometry-improving points in the new (larger & shifted) trust region $B(\mathbf{x}_k, \Delta_k)$ as per (3.1). The iteration then continues from whichever of these N new points has the least objective value, which may be worse than the value from the end of the previous iteration. The final solution



(a) Normalized objective decrease *(b) Convergence details*

Figure 2. Normalized objective decrease achieved by DFO-LS, measured in both the noisy and true objective, and convergence information as per Figure 1, but allowing auto-detection of restarts. The budget was $100(n+1)$ objective evaluations. The difference between the 'noisy' and 'true' objective reduction measures is discussed in Section 4.1.

returned by the solver takes the optimal value seen so far, including the saved endpoints from previous restarts.

The soft restart approach with $N = \min(3, p)$ is the default approach in DFO-LS.

We see that soft restarts require the objective to be evaluated $N < p$ times, whereas hard restarts require a full p objective evaluations (as we have changed all interpolation points except $\mathbf{x}_k$). We also note that the soft restart mechanism is intrinsically linked to the model-based DFO framework, and there is not a clear derivative-based equivalent of this procedure.

In DFO-LS, when restarts are used, we add an extra termination criterion: we terminate if the last M consecutive restarts have not achieved any objective reduction (default $M = 10$).

Auto-detection of restarts As discussed above, we saw in Figure 1 that the need for a restart can be determined by a series of unsuccessful iterations, coupled with large changes in J_k, with this change increasing rapidly with k. By contrast, selecting an a priori value of ρ_{end} which provides a timely trigger for restarts is not straightforward. Thus, DFO-LS uses previous iteration information to auto-detect when a restart is needed. A restart is triggered if, in the last N iterations:

- The trust region radius Δ_k has never been increased, and it has been decreased on at least twice as many iterations as it has been kept constant; and,
- The slope and correlation coefficient of a linear fit through the points $\{(k, \log \|J_k - J_{k-1}\|_F)\}$ exceed given thresholds; that is, $\|J_k - J_{k-1}\|_F$ is consistently increasing at a given exponential rate[7].

An illustrative example, revisited. In Figure 2, we see the same results as in Figure 1, but with auto-detection of when to restart. Because of the auto-detection, restarts are triggered much earlier (avoiding the iterations during which no progress was made), and we achieve accuracy $\tau = 10^{-6}$ after approximately $15(n + 1)$ evaluations, rather than approximately $80(n + 1)$ evaluations without auto-detection. □

Alternative Restart Mechanism DFO-LS has another approach available for performing soft restarts.

Soft restart (fixed $\mathbf{x}_k$): Reset the trust region radius to $\Delta_k = \rho_k = \Delta_0$, and move the $N < p$ points $\mathbf{y}_t \neq \mathbf{x}_k$ in Y_k which are closest to $\mathbf{x}_k$ to geometry-improving points in the new trust region $B(\mathbf{x}_k, \Delta_k)$ as per (3.1). The iteration then continues from whichever of these N new points has the least objective value.

As we will see in Section 5, the soft restart (fixed $\mathbf{x}_k$), with the default value $N = \min(3, p)$, performs noticeably worse than the 'moving $\mathbf{x}_k$' version of soft restarts.

[7] This condition is similar to the noise detection mechanism from [2], which we became aware of after the condition had been chosen.

3.3 Optional Features for Noisy Objectives

Aside from multiple restarts, DFO-LS also implements the two most common approaches for handling noisy objectives in model-based DFO: sample averaging and regression models. In Section 5, we show numerically that using multiple restarts gives better performance than averaging and regression. Therefore, the latter features are not used in DFO-LS by default.

3.3.1 Sample Averaging

Sample averaging replaces evaluations of the noisy objective $f(\mathbf{x})$ with an average of N samples. In the least-squares case, we replace an evaluation of the noisy objective $\overline{\mathbf{r}}$ with the sample average

$$\overline{\mathbf{r}}_N(\mathbf{x}) := \frac{1}{N}\sum_{i=1}^{N} \mathbf{r}(\mathbf{x}, \xi_i), \tag{3.4}$$

where $\xi_1, \ldots, \xi_N$ are different realizations of the random variable ξ defining the noise.

For theoretical convergence guarantees to hold, one must choose $N = \mathcal{O}(\Delta_k^{-4})$ (e.g. [8]). However, as Δ_k can easily be of size 10^{-2} or 10^{-3}, this amount of averaging rapidly becomes impractical, so $N = \mathcal{O}(\Delta_k^{-1})$, for instance, is a more sensible choice in practice (e.g. Algorithm TR-SAA in [8]).

In the implementation of DFO-LS, the use of sample averaging is governed by a user-specified function which allows for a wide range of sample averaging techniques:

$$N = \text{nsamples}(\rho_k, \Delta_k, k, n_{\text{restarts}}), \tag{3.5}$$

where $n_{\text{restarts}} \in \{0, 1, 2, \ldots\}$ is the number of restarts that the solver has performed (see Section 3.2 for details) and $k \in \{0, 1, 2, \ldots\}$ is the iteration number since the most recent restart. The default option for nsamples gives $N \equiv 1$ always; i.e. no sample averaging.

3.3.2 Regression Models for Noisy Objectives

Building regression models rather than interpolation models requires having more interpolation points than degrees of freedom in the model [10, 5, 8]. Our formulation of the DFO-LS model construction problem (2.3) allows regression models, when $p > n$. It remains to consider how to evolve the set Y_k at each iteration.

DFO-LS has three mechanisms for moving multiple points on successful iterations, which can be used alongside regression models:

Nothing: Replace one point in Y_k with $\mathbf{x}_{k+1}$, and nothing else;

Geometry-based: Replace one point in Y_k with $\mathbf{x}_{k+1}$, and then one-by-one[8] move the N points in Y_k which are furthest from Y_{k+1} to geometry-improving locations in $B(\mathbf{x}_{k+1}, \Delta_{k+1})$, given by (3.1); and

Momentum-based: Replace one point in Y_k with $\mathbf{x}_{k+1}$, and then move the N points in Y_k which are furthest from Y_{k+1} to $\mathbf{x}_{k+1} + \Delta_{k+1}\mathbf{d}$, where $\mathbf{d}$ is a random unit vector with[9] $\mathbf{d}^\top \mathbf{s}_k > 0$.

The last two mechanisms above, that replace multiple points, try to mimic/match the situation arising in sample averaging, where moving one point affects c function values, where c is the sampling rate.

The first mechanism ('nothing') is the default in DFO-LS, when regression models are used; they are compared with each other, and against sample averaging, in Section 5.3.

4 Testing Framework

We outline the framework we used for testing and comparing DFO-LS to other solvers, which is similar to Moré and Wild [26] and to [7], but designed to capture the two regimes of interest - expensive and/or noisy. We also introduce a new approach for measuring solver performance for noisy problems.

[8] That is, (3.1) is calculated sequentially, after the previous point has been moved (and Lagrange polynomials recalculated).

[9] When using bound constraints, if $\Delta_{k+1}\mathbf{d}$ gives a point outside the bounds, we instead take $\alpha\mathbf{d}$ for some $\alpha \in (0, \Delta_{k+1})$ such that the bounds are satisfied. If this requires $\alpha < 10^{-3}$, we replace $\mathbf{d}$ with $-\mathbf{d}$, sacrificing the requirement that $\mathbf{d}^\top \mathbf{s}_k > 0$.

4.1 Testing Methodology

When running each solver, we choose the maximum allowed budget in units of simplex gradients (i.e. in multiples of $n+1$) to provide fair comparisons across problems of different dimensions. The measure of solver performance is the number of evaluations required to achieve a specified reduction in the objective.

Suppose we have an underlying smooth objective f, but only see evaluations of the noisy objective $\widetilde{f} \approx f$, where

$$\widetilde{f}(\mathbf{x}) := \alpha f(\mathbf{x}) + \beta + \sigma(\mathbf{x})\epsilon, \tag{4.1}$$

for constants $\alpha > 0$ and $\beta \in \mathbb{R}$, and where $\sigma(\mathbf{x})$ is the standard deviation of the noise. The stochastic noise ϵ has zero mean and unit variance, and so $|\epsilon| \sim \mathcal{O}(1)$. Under this assumption — which holds for all the noise models we consider in Section 4.2 — minimizing $\mathbb{E}[\widetilde{f}(\mathbf{x})]$ yields a minimizer of the true objective f. Then, for a solver $\mathcal{S}$, problem p, and accuracy level $\tau \in (0,1)$, we define the number of evaluations required to solve the (smooth or noisy) problem as:

$$\left\{ \begin{array}{c} N_p(\mathcal{S};\tau) \\ \widetilde{N}_p(\mathcal{S};\tau) \end{array} \right\} := \#\text{ objective evaluations taken to find a point } \mathbf{x} \text{ satisfying } \left\{ \begin{array}{lcl} f(\mathbf{x}) & \leq & f(\mathbf{x}^*) + \tau(f(\mathbf{x}_0) - f(\mathbf{x}^*))] \\ \widetilde{f}(\mathbf{x}) & \leq & \mathbb{E}[\widetilde{f}(\mathbf{x}^*) + \tau(\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*))] \end{array} \right\}, \tag{4.2}$$

where $\mathbf{x}^*$ is an estimate of the true minimizer of the smooth objective f[10]. If the required reduction (4.2) was never achieved in the maximum allowed budget, we define $N_p(\mathcal{S};\tau)$ or $\widetilde{N}_p(\mathcal{S};\tau) = \infty$.

Following [26], we compare different solvers using data profiles[11]. These measure the proportion of test problems for which the required budget $N_p(\mathcal{S};\tau)$ or $\widetilde{N}_p(\mathcal{S};\tau)$ is less than a given value (in units of simplex gradients), namely $\alpha(n_p+1)$, where n_p is the dimension of problem p. The data profiles are defined, for the solver $\mathcal{S}$ and set of test problems $\mathcal{P}$, to be the curves

$$\left\{ \begin{array}{c} d_{\mathcal{S}}(\alpha) \\ \widetilde{d}_{\mathcal{S}}(\alpha) \end{array} \right\} := \frac{1}{|\mathcal{P}|} \cdot \left| \left\{ p \in \mathcal{P} : \left\{ \begin{array}{c} N_p(\mathcal{S};\tau_p) \\ \widetilde{N}_p(\mathcal{S};\tau_p) \end{array} \right\} \leq \alpha(n_p+1) \right\} \right|, \qquad \alpha \geq 0, \tag{4.3}$$

where n_p is the dimension of problem p. $N_p(\mathcal{S};\tau)$ measures genuine progress towards the minimizer, excluding any objective reductions from sampling errors, but $\widetilde{N}_p(\mathcal{S};\tau)$ measures progress using information actually available to the solver. The two different performance measures (4.2) have each been used previously (e.g. $N_p(\mathcal{S};\tau)$ in [8] and $\widetilde{N}_p(\mathcal{S};\tau)$, in [5]), but we are not aware of work where the two measures have been compared and combined. We propose to do so, in the accuracy level we choose.

We use throughout a problem-specific accuracy level τ_p rather than a constant value for all problems, which we set based on the accuracy which is reasonable for a solver to attain given the noise level in the problem. By adapting τ_p to each problem, we can measure progress using $\widetilde{N}_p$ but gain the benefit of N_p, thus capturing genuine progress in the objective.

Note that throughout, for noisy problems, we ran each solver on 10 instances of each problem (i.e. independent realizations of the random noise in each objective evaluations). For data profiles, we treat each instance as a separate problem to be solved (e.g. for the (MW) set, we plot the proportion of 530 problem instances solved in a given budget). We also use 10 instances for each run of DFO-LS and Py-BOBYQA for noiseless objectives, because the generation of the initial set Y_0 uses random orthogonal directions.

Choice of adaptive accuracy level τ_p Given (4.2), we would expect the two measures N_p and $\widetilde{N}_p$ to be similar, provided that our desired objective reduction was much larger than the noise level. If N_p and $\widetilde{N}_p$ were similar, we could conclude that reductions in the observable $\widetilde{f}$ correspond to genuine objective reductions, and not sampling error. Specifically, suppose a solver has reached a point $\mathbf{x}_k$, which corresponds to an accuracy of (exactly) τ based on N_p and of $\widetilde{\tau}$ based on $\widetilde{N}_p$,

$$f(\mathbf{x}_k) = f(\mathbf{x}^*) + \tau(f(\mathbf{x}_0) - f(\mathbf{x}^*)) \quad \text{and} \quad \widetilde{f}(\mathbf{x}_k) = \mathbb{E}[\widetilde{f}(\mathbf{x}^*) + \widetilde{\tau}(\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*))]. \tag{4.4}$$

Letting ϵ_k be the realization of ϵ for the given $\widetilde{f}(\mathbf{x}_k)$, we combine the expressions in (4.4) using (4.1), to get

$$\underbrace{\widetilde{\tau}}_{\text{noisy progress}} = \tau + \frac{\sigma(\mathbf{x}_k)}{\alpha(f(\mathbf{x}_0) - f(\mathbf{x}^*))}\epsilon_k = \underbrace{\tau}_{\text{true progress}} + \underbrace{\frac{\sigma(\mathbf{x}_k)}{\mathbb{E}\left[\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*)\right]}\epsilon_k}_{\text{sampling error}}, \tag{4.5}$$

[10] For our two sets of test problems (Moré & Wild and CUTEst), values of $f(\mathbf{x}^*)$ are given in [7].

[11] We also generated performance profiles [17] for all results in this paper. However, we found they do not add much additional information or change the view on solvers' performance, and so are omitted here. For interested readers, performance profiles for the key figures in this paper are available from the authors upon request.

where we note that $\alpha(f(\mathbf{x}_0) - f(\mathbf{x}^*)) = \mathbb{E}\left[\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*)\right]$ follows from the choice of noise model (4.1) and the fact that $\mathbb{E}[\sigma(\mathbf{x})\epsilon] = \sigma(\mathbf{x})\mathbb{E}[\epsilon] = 0$ (as we assumed ϵ has mean zero). Since $|\epsilon_k| \sim \mathcal{O}(1)$, we can expect $\widetilde{\tau}$ and τ to be similar in size whenever

$$\tau \text{ and/or } \widetilde{\tau} \gg \frac{\sigma(\mathbf{x}_k)}{\mathbb{E}\left[\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*)\right]}. \tag{4.6}$$

Effectively, (4.6) provides a limit on the accuracy we can reasonably expect a solver to achieve, given the noise level in a particular problem. In our numerical results, we approximate (4.6) in a way that is independent of $\mathbf{x}_k$, and say that the best accuracy we can expect a solver to achieve is $\tau_{crit}(p)$, where

$$\tau_{crit}(p) := 10^{\lceil \log_{10} \widehat{\tau}(p) \rceil}, \qquad \text{where} \qquad \widehat{\tau}(p) := \frac{\sigma(\mathbf{x}^*)}{\mathbb{E}\left[\widetilde{f}(\mathbf{x}_0) - \widetilde{f}(\mathbf{x}^*)\right]}. \tag{4.7}$$

Finally, to construct our data profiles, we choose a desired level of accuracy τ, usually 10^{-5}, and set our problem-specific tolerance to be either τ if we can expect the solver to reach this accuracy, otherwise we choose $\tau_{crit}(p)$. Thus, in our data profiles (4.3), we use the problem-specific accuracy level

$$\tau_p := \min(\tau_{max}, \max(\tau_{crit}(p), \tau)), \tag{4.8}$$

where $\tau_{max} := 10^{-1}$ is an upper bound on τ_p. We note that for noiseless problems we have $\tau_{crit}(p) = 0$, so $\tau_p = \tau$ is problem-independent.

Impact of τ_p We now illustrate that using the per-problem accuracy level τ_p for measuring noisy progress $\widetilde{N}_p$ allows us to compare performance based on genuine objective decreases, not sampling errors. We do this by verifying that the performance measures N_p and $\widetilde{N}_p$, and data profiles $d_{\mathcal{S}}$ and $\widetilde{d}_{\mathcal{S}}$, give similar results.

First, we consider the problem and noise model used in Figures 1 and 2, where $\tau_{crit}(p) = 10^{-7}$. In two runs[12] of the same problem, shown in Figures 1c and 2a, we see that the decreases achieved under both measures are essentially identical until they reach accuracy level very close to $\tau_{crit}(p)$.

Next, in Figure 3, we show a set of data profiles which we will discuss[13] in Section 6, but we compare the same solvers using both the 'noisy $\widetilde{f}$' and 'true f' measures of objective reduction, and consider either a fixed $\tau = 10^{-5}$ for all problems, or using the per-problem value τ_p (4.8). We see that when a constant value of τ_p is used, the two data profiles look very different, with the noisy $\widetilde{d}_{\mathcal{S}}$ profile showing more problems solved than the true $d_{\mathcal{S}}$ profile. When we switch to the per-problem threshold (4.8), the two profiles look much more consistent both in shape and magnitude. Most importantly, conclusions about relative solver performance in both low-budget and long-budget regimes based on the $\widetilde{d}_{\mathcal{S}}$ profile would be consistent with those based on the true $d_{\mathcal{S}}$ measure.

Thus, for the remainder of the paper, we present our numerical results using the problem-adjusted τ_p with the noisy data profile $\widetilde{d}_{\mathcal{S}}$ (e.g. Figure 3c).

4.2 Test Problems and Solver Settings

We test DFO-LS on the two collections used in [7]:

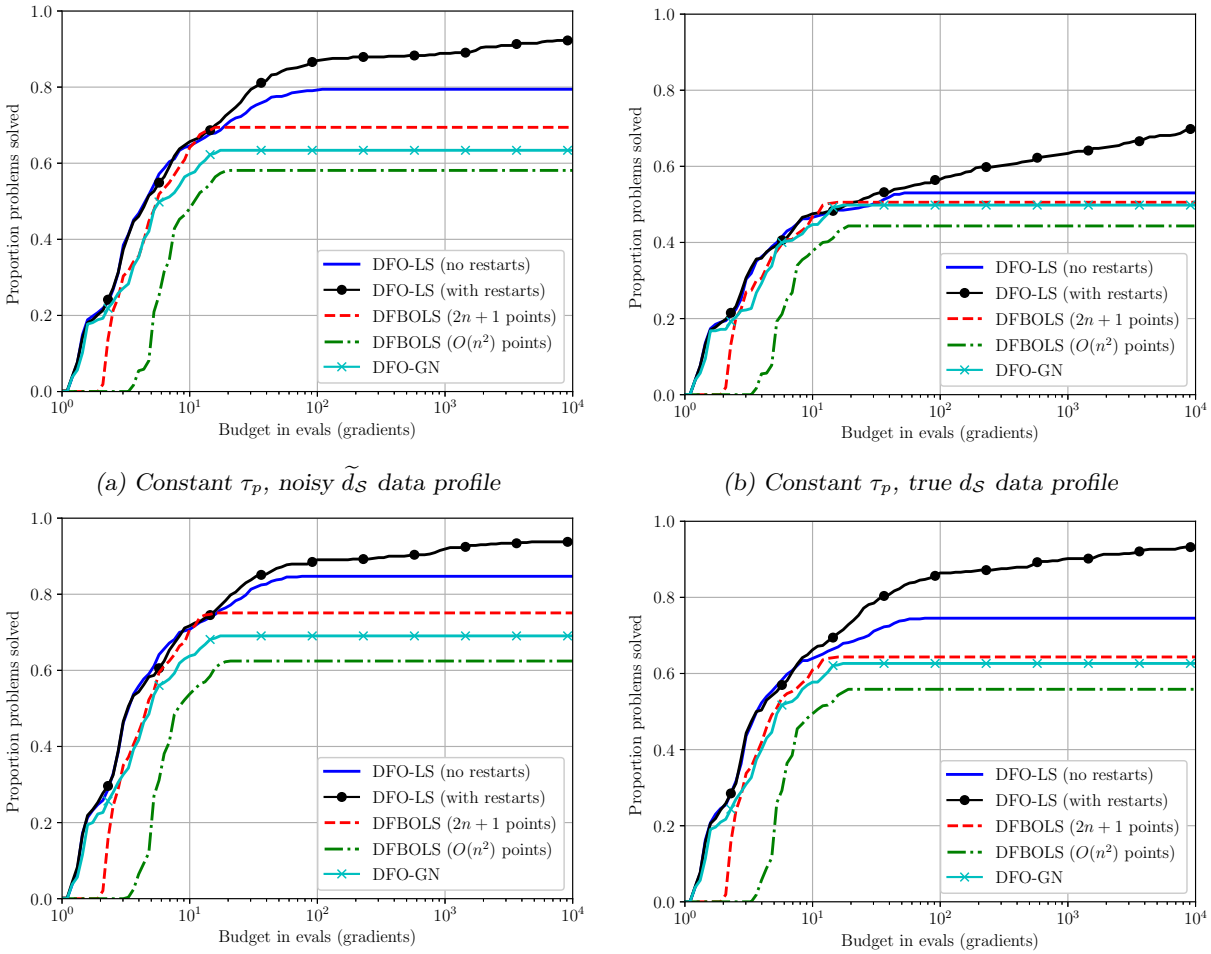
(MW) The set of 53 nonlinear least-squares from Moré and Wild [26]. The problems are low-dimensional, with $2 \leq n \leq 12$ and $n \leq m \leq 65$, so this collection is used as the main test set for the 'noisy' regime (see noise models below);

(CR) The set of 60 nonlinear least-squares from [7], available via the CUTEst package [18]. The problems are medium-sized, with $25 \leq n \leq 120$ and $n \leq m \leq 400$, so this collection is used as the main test set for the 'expensive' regime.

Full details of both collections may be found in [7].

[12] Although the same random seed was used in both runs for reproducibility, since they have a different sequence of restarts, the set of iterates is ultimately different.

[13] This figure compares DFO-LS to other solvers for objectives with additive Gaussian noise; see Figure 10b. The same conclusions may be drawn by using the other plots in this paper.



(a) Constant τ_p, noisy $\widetilde{d}_{\mathcal{S}}$ data profile

(b) Constant τ_p, true $d_{\mathcal{S}}$ data profile

(c) Per-problem τ_p, noisy $\widetilde{d}_{\mathcal{S}}$ data profile

(d) Per-problem τ_p, true $d_{\mathcal{S}}$ data profile

Figure 3. A comparison of the results in Figure 10 with additive Gaussian noise, using the data profiles $d_{\mathcal{S}}$ and $\widetilde{d}_{\mathcal{S}}$ (4.3), and either choosing $\tau_p = 10^{-5}$ for all problems, or applying the per-problem threshold (4.8).

Noise Models For results robustness, we allow the evaluation of $\mathbf{r}(\mathbf{x})$ to include several types of stochastic noise. In the following sections, we show results for the following noise models, where $\widetilde{f}(\mathbf{x}) = \sum_{i=1}^{m} \widetilde{r}_i(\mathbf{x})^2$,

- Smooth (noiseless) evaluations: $\widetilde{r}_i(\mathbf{x}) = r_i(\mathbf{x})$;
- Multiplicative Gaussian noise: $\widetilde{r}_i(\mathbf{x}) = (1+\epsilon)r_i(\mathbf{x})$, where $\epsilon \sim N(0, \sigma^2)$;
- Additive Gaussian noise: $\widetilde{r}_i(\mathbf{x}) = r_i(\mathbf{x}) + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$; and
- Additive χ^2 noise: $\widetilde{r}_i(\mathbf{x}) = \sqrt{r_i(\mathbf{x})^2 + \epsilon^2}$, where $\epsilon \sim N(0, \sigma^2)$.

In each case, ϵ is drawn i.i.d. for each $\mathbf{x}$ and each $i = 1, \ldots, m$; and $\sigma = 10^{-2}$. For noisy problems, our goal is to minimize $\widetilde{f}(\mathbf{x})$ in expectation — note that $\mathbb{E}[\widetilde{f}(\mathbf{x})]$ is an affine transformation of $f(\mathbf{x})$ for these noise models, so they have the same minimizer(s).

Solver Settings In the below, we compare DFO-LS v1.0.1 against DFO-GN v0.2 [7] and DFBOLS [38]. For DFBOLS, we show results using with $2n+1$ and $(n+1)(n+2)/2$ interpolation points. For all solvers, we choose trust region settings $\Delta_0 = 0.1 \max(\|\mathbf{x}_0\|_\infty, 1)$ and $\rho_{end} = 10^{-8}$, and the default values for all other parameters (unless otherwise specified).

We used a maximum budget of $10^4(n+1)$ evaluations for the Moré and Wild problems (MW). Particularly for noisy problems, we are interested in a regime where objective evaluations are cheap, and we are concerned with the robustness of each solver — how many problems can it solve, if budget were not an issue. Since this budget is much larger than is often used for testing (e.g. [38, 7]), we show data profiles with a log-scale for budget, so we can easily compare solvers both for large budgets (to check robustness) and for realistically

small budgets[14]. For the CUTEst problems (CR), we used a much smaller budget of $50(n+1)$ evaluations, to represent the other regime, where objectives are expensive to evaluate.

5 Numerical Studies of New DFO-LS Features

We test the new features of DFO-LS and showcase the successful ones which are chosen as defaults, with the remaining features available as options. Section 6 then compares DFO-LS with its default settings against state-of-the-art DFO least-squares solvers.

5.1 Reduced Initialization Cost

In Figure 4, we consider the CUTEst problems (CR) with noiseless evaluations. We compare the basic implementation of DFO-LS against DFO-LS with a reduced initialization cost of 2, $n/4$ and $n/2$ function evaluations (growing the direction space via both mechanisms described in Section 2.1 above: modifying J_k using its SVD, and perturbing the trust region step). After this reduced initialization cost, the algorithm begins the main iteration, and starts to progress towards the solution. Using our small budget of $50(n+1)$ evaluations, we show data profiles in two settings[15]: for a small budget ($5(n+1)$ evaluations) with low accuracy $\tau = 10^{-1}$, and the full budget with high accuracy $\tau = 10^{-5}$.

In the short budget, low accuracy plots, we see the benefit of a reduced initialization cost — we are able to solve a notable fraction of the problems to low accuracy with very few evaluations; less than the number required to perform a single gradient evaluation. We also see the tradeoff of this benefit, which is a lower performance at small budgets (1–3 gradients). However, the long budget, high accuracy plots in Figure 4 show that we do not lose robustness regardless of the initialization cost, as the small-budget performance loss does not perpetuate to longer budgets. The difference between initializing with 2, $n/4$ and $n/2$ points is not substantial. Comparing the two mechanisms for increasing the model dimensionality, we find that the SVD approach performs similarly to the perturbed trust region approach for small budgets, but better matches DFO-LS with a full initialization set. We note that all our test problems are least-squares or nonlinear systems, so $m \geq n$ in all cases.

5.2 Sample Averaging

To demonstrate the impact of using sample averaging, Figure 5 shows data profiles with averaging strategies $N \in \{1, 2, 5, 10, 30, \max(1, \lfloor \Delta_k^{-1} \rfloor)\}$. Each of the N samples for a given $\mathbf{x}_k$ are counted towards the maximum computational budget of $10^4(n+1)$ values.

Unsurprisingly, we see that using a larger number of samples can improve the robustness of DFO-LS. Of course, to achieve this robustness, a proportionally larger number of evaluations are required, so for small-to-medium budgets (in serial) we lose in performance. This does not take into account the benefits of parallelization that may be available when sample averaging is used. We also notice that using $N = \mathcal{O}(\Delta_k^{-1})$ can provide a compromise — it still makes progress for small budgets, but manages to achieve a reasonable level of robustness overall.

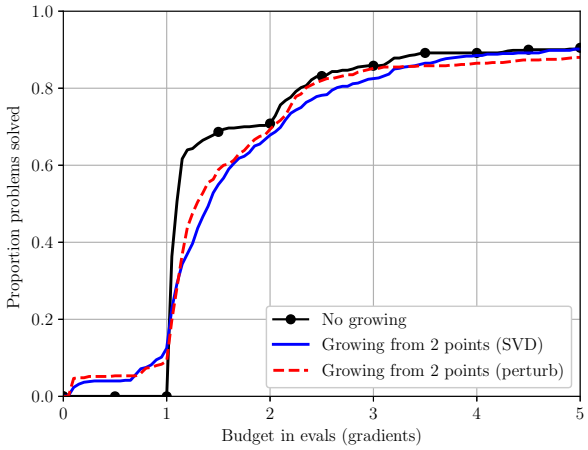
5.3 Regression Models

In practice, we find that the geometry-based moves perform similarly to or slightly better than momentum-based ones, so we do not show results for the latter mechanism. Figure 6 compares the remaining two techniques with varying numbers of regression points ($|Y_k| = c(n+1)$ for $c \in \{5, 30\}$) against interpolation models ($|Y_k| = n+1$). We see that using a larger sample set improves the robustness of DFO-LS, particularly for additive Gaussian noise, and this improvement (for $|Y_k| = c(n+1)$) is generally comparable to, or slightly worse than, the use of sample averaging (with c samples at each point). The geometry-based mechanism for moving multiple points makes the algorithm progress more slowly, as indicated by the performance profiles, while at times providing a slight improvement over the 'basic' approach (moving one point per iteration).

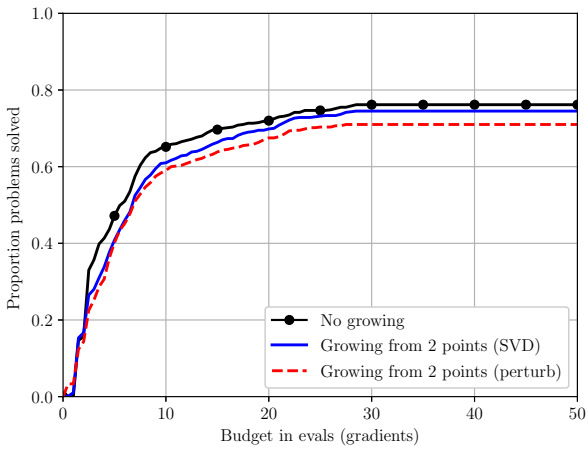
In [6, Appendix C], we argue, briefly and in a simplified framework, that regression and sample averaging generate similar model error; thus, since sample averaging produces a better estimate of objective decrease for fixed noise level, we expect that overall, regression will be slightly less robust compared to sample averaging when considering large computational budgets.

[14] Note that, unlike performance profiles, reading a small-budget result from long-budget data profiles would match exactly the result of running the solvers with a smaller budget in the first place.

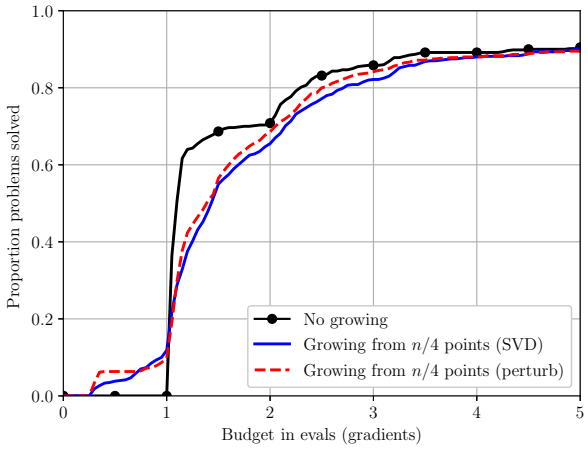
[15] We omit the data profiles for the full budget with accuracy $\tau = 10^{-1}$, and for small budget ($5(n+1)$ evaluations) and $\tau = 10^{-5}$, for brevity reasons and because they do not add additional qualitative information. These profiles are nevertheless available from the authors upon request.



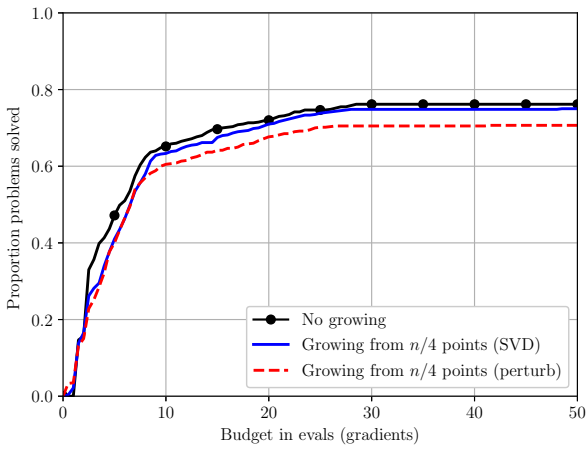
(a) Short budget, 2 starting points, $\tau = 10^{-1}$



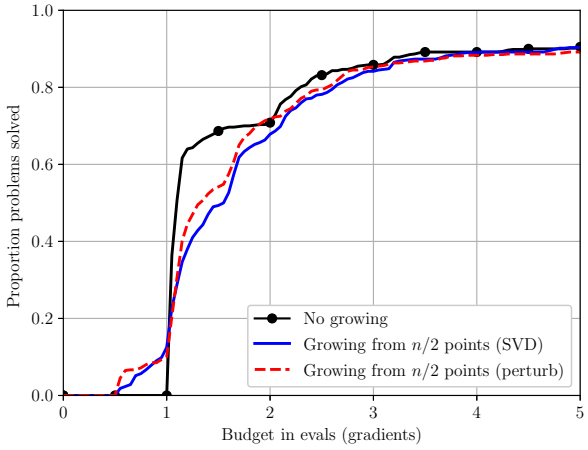
(b) Long budget, 2 starting points, $\tau = 10^{-5}$



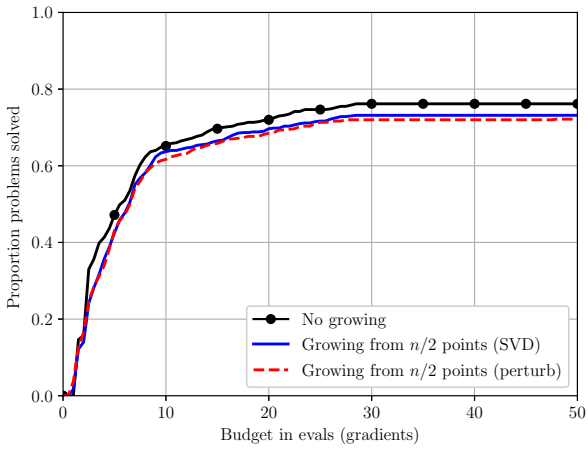
(c) Short budget, $n/4$ starting points, $\tau = 10^{-1}$



(d) Long budget, $n/4$ starting points, $\tau = 10^{-5}$

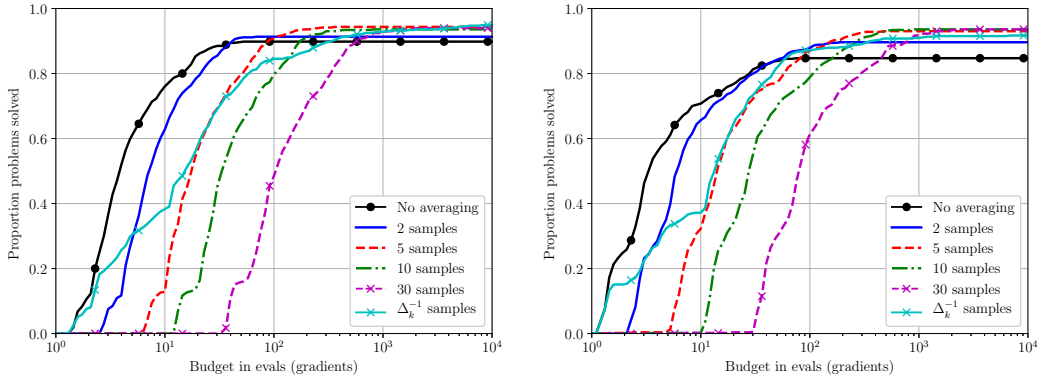


(e) Short budget, $n/2$ starting points, $\tau = 10^{-1}$

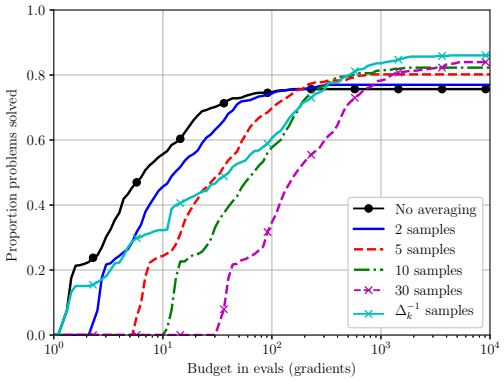


(f) Long budget, $n/2$ starting points, $\tau = 10^{-5}$

Figure 4. Data profiles showing the impact of the reduced initialization cost of DFO-LS (using $n + 1$ interpolation points) against using the full initial set, for smooth objectives. Results an average of 10 runs in each case. The problem collection is (CR).

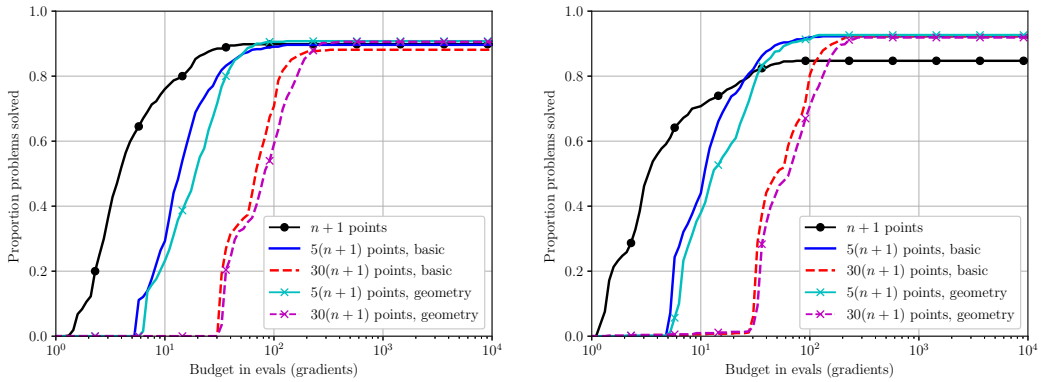


(a) Multiplicative Gaussian noise *(b) Additive Gaussian noise*

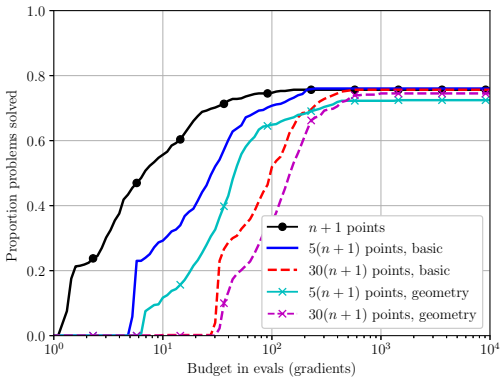


(c) Additive χ^2 noise

Figure 5. Comparison of different sample averaging methods for DFO-LS (using $n+1$ interpolation points). We are using noisy objective evaluations with $\sigma = 10^{-2}$, high accuracy $\tau = 10^{-5}$, and an average of 10 runs for each solver. The problem collection is (MW).

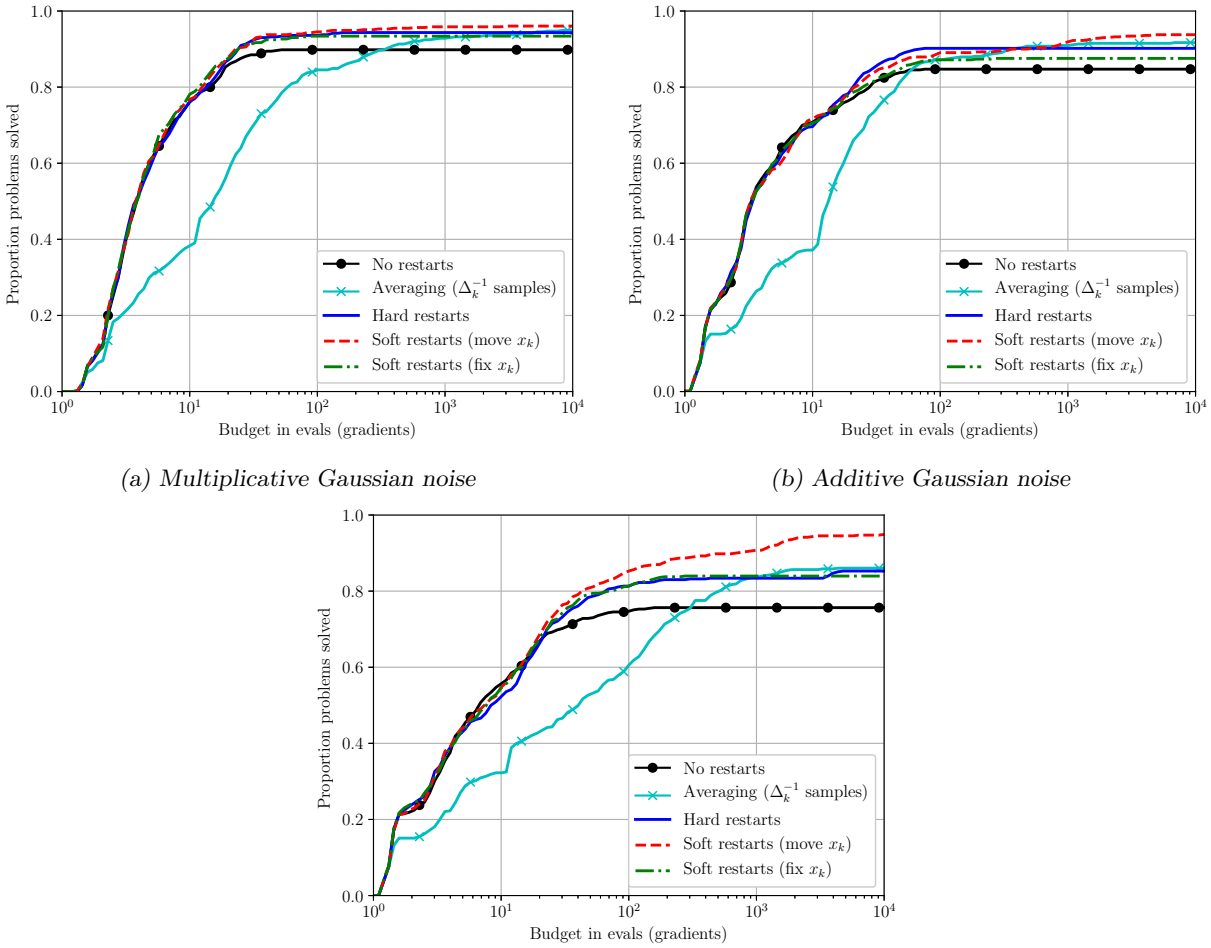


(a) Multiplicative Gaussian noise *(b) Additive Gaussian noise*



(c) Additive χ^2 noise

Figure 6. Impact of regression for DFO-LS. We are using noisy objective evaluations with $\sigma = 10^{-2}$, high accuracy $\tau = 10^{-5}$, and an average of 10 runs for each solver. The problem collection is (MW).



(a) Multiplicative Gaussian noise

(b) Additive Gaussian noise

(c) Additive χ^2 noise

Figure 7. Impact of multiple restarts for DFO-LS (using $n+1$ interpolation points). We are using noisy objective evaluations with $\sigma = 10^{-2}$, high accuracy $\tau = 10^{-5}$, and an average of 10 runs for each solver. The problem collection is (MW).

5.4 Multiple Restarts

Figure 7 compares the different restart methods against sample averaging (30 samples at every point). All runs use auto-detection of restarts and the optional noise-based termination criterion (3.3).

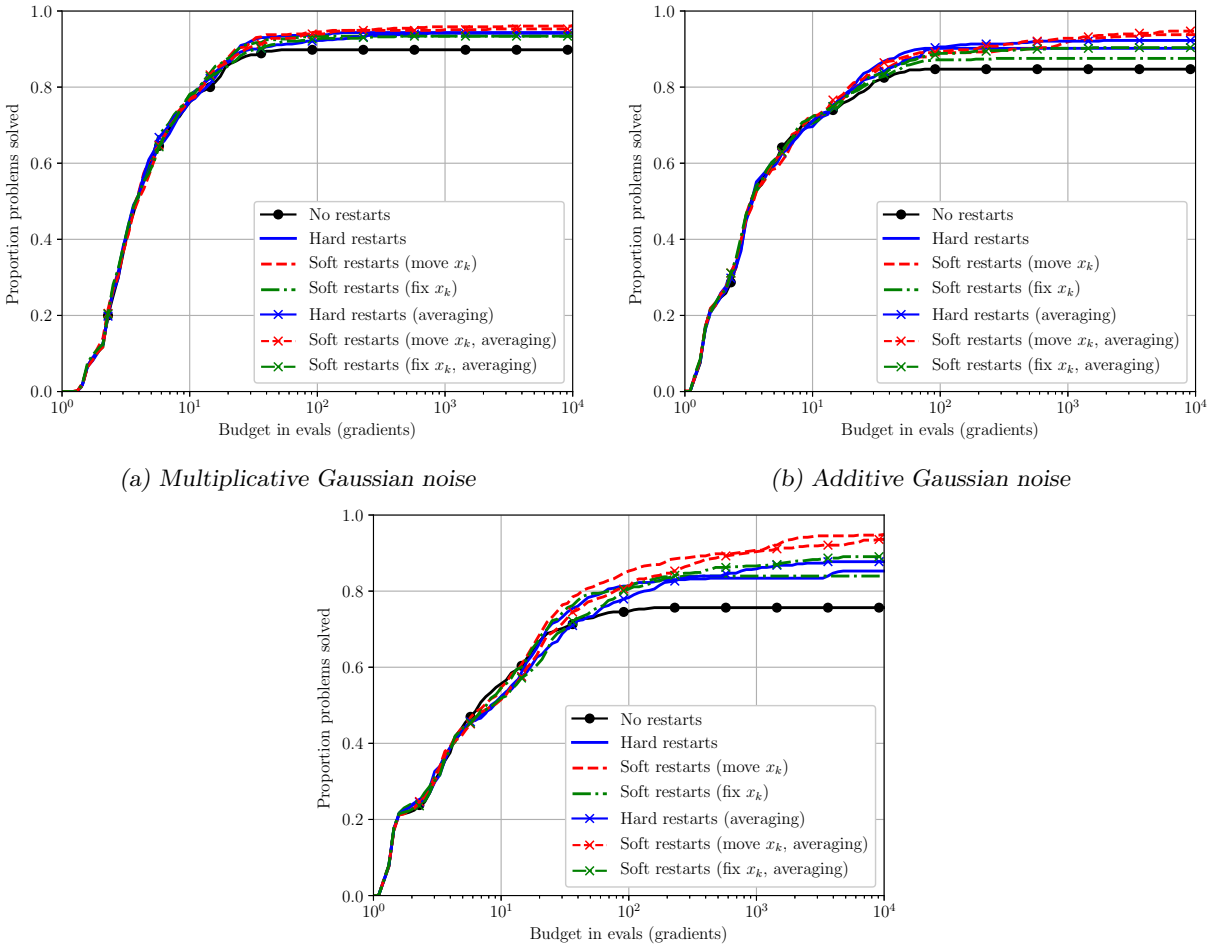
We see that soft restarts (moving $\mathbf{x}_k$) is the most successful restarts mechanism, followed by hard restarts, then soft restarts (fixing $\mathbf{x}_k$), and that all these mechanisms are better than DFO-LS without any noise-based features. Compared to the case of using averaging with Δ_k^{-1} samples at every point, soft restarts (moving $\mathbf{x}_k$) achieve a similar or better level of robustness with many fewer objective evaluations — this is most clearly observed in the case of additive χ^2 noise.

The improvements in robustness from using multiple restarts are obvious at the end of the full budget of 10^4 simplex gradients, but there are still benefits to be found at much smaller budgets (e.g. $\mathcal{O}(100)$ simplex gradients). As a result of these benefits, the soft restarts (moving $\mathbf{x}_k$) mechanism is activated by default in DFO-LS for noisy problems.

Next, we consider the impact of using increased levels of sample averaging with every restart. The reason for this is that after every restart, we hope to be closer to the desired solution, so an increased amount of averaging may help distinguish points near to this optimum. To achieve this, in (3.5) we use

$$N = \text{nsamples}(\rho_k, \Delta_k, k, n_{\text{restarts}}) = \min\{n_{\text{restarts}} + 1, 30\}. \tag{5.1}$$

Figure 8 shows that augmenting multiple restarts with sample averaging improves the robustness of hard and soft restarts (fixing $\mathbf{x}_k$), but not for the default mechanism (soft restarts moving $\mathbf{x}_k$). Ultimately, using soft restarts (moving $\mathbf{x}_k$) is better than the other two restart mechanisms, with or without sample averaging. Hence, we do not use any sample averaging in DFO-LS by default.



(a) Multiplicative Gaussian noise

(b) Additive Gaussian noise

(c) Additive χ^2 noise

Figure 8. Comparison of multiple restarts with and without sample averaging for DFO-LS (using $n+1$ interpolation points). We are using noisy objective evaluations with $\sigma = 10^{-2}$, high accuracy $\tau = 10^{-5}$, and an average of 10 runs for each solver. The problem collection is (MW).

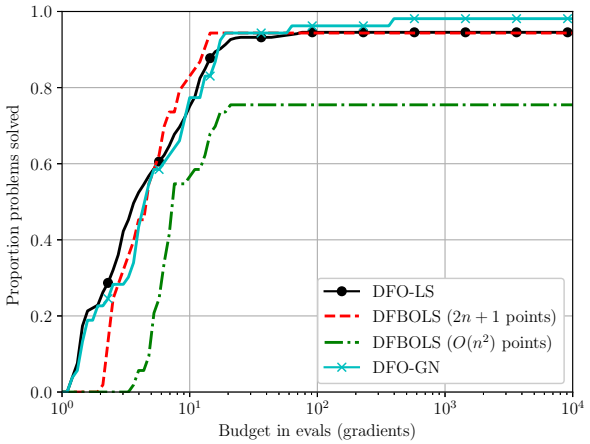
6 Benchmark Comparisons of DFO-LS

This section compares the performance of DFO-LS with other, state-of-the-art derivative-free solvers for nonlinear least-squares problems, namely, DFO-GN [7], and DFBOLS [38] with $2n+1$ and $(n+1)(n+2)/2$ points. DFO-LS uses $p+1 = n+1$ interpolation points and the default values for all other parameters, unless otherwise specified. We use the computational budget, and initial and final trust region radii as in Section 4.1, with accuracy level $\tau = 10^{-5}$.

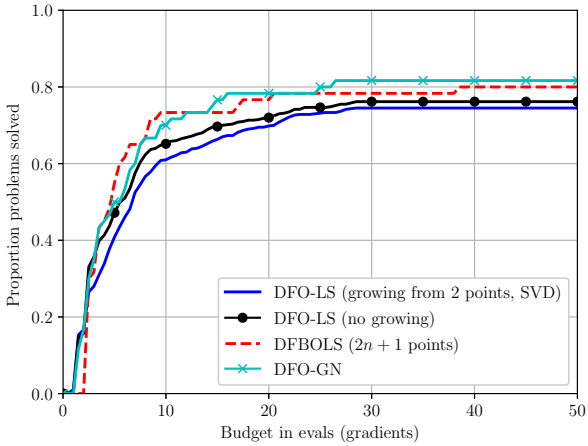
Figure 9 shows results for smooth (noiseless) objective functions for both the (MW) and (CR) test sets. Since DFO-LS uses randomized initial points, we show an average result over 10 runs. For the (CR) set, we do not show DFBOLS with $(n+1)(n+2)/2$ points, as in most cases the initialization cost will use almost all of the available budget. DFO-LS performs similarly to DFO-GN and DFBOLS, which is to be expected given the similarity of these algorithms. Note that for (CR), initializing DFO-LS with only 2 evaluations yields only slightly worse performance, but gains the benefit of decreasing the objective at a very low cost (as shown in Section 5.1).

Similarly, for noisy functions (from the (MW) set), Figure 10 shows DFO-LS with and without restarts versus the same solvers as above. It is in this scenario that the flexibility of DFO-LS becomes evident — its ability to adjust the default algorithm parameters in the presence of noisy evaluations allows it to solve a larger proportion of problems than both DFBOLS and DFO-GN, and this robustness is further improved, by a significant margin, by the use of multiple restarts.

Expensive & Noisy Problems Next, we illustrate that the two regimes — 'expensive' and 'noisy' — are not mutually exclusive. In Figure 11, we run DFO-LS, DFBOLS and DFO-GN on the (CR) problem set with additive Gaussian noise. The DFO-LS runs use the default settings for noisy problems (i.e. slower trust

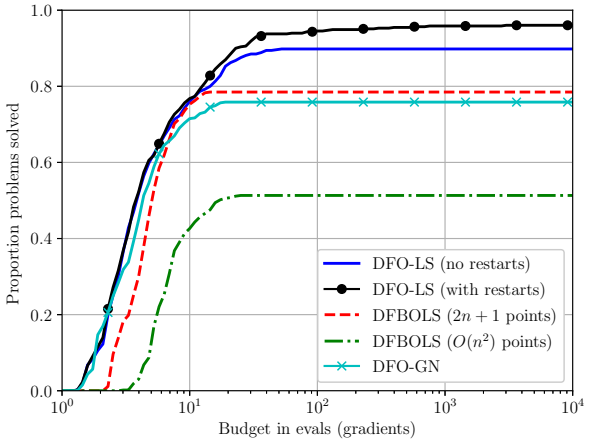


(a) Problem collection (MW)

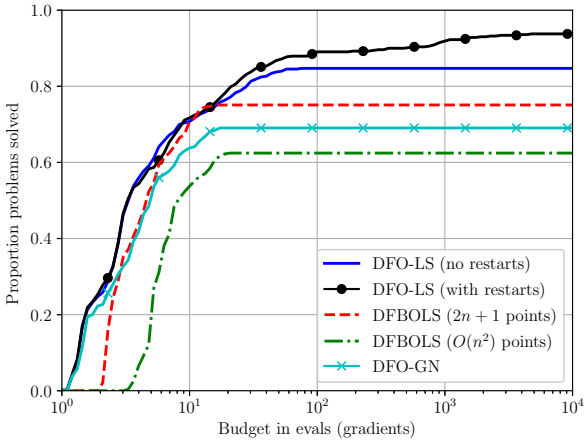


(b) Problem collection (CR)

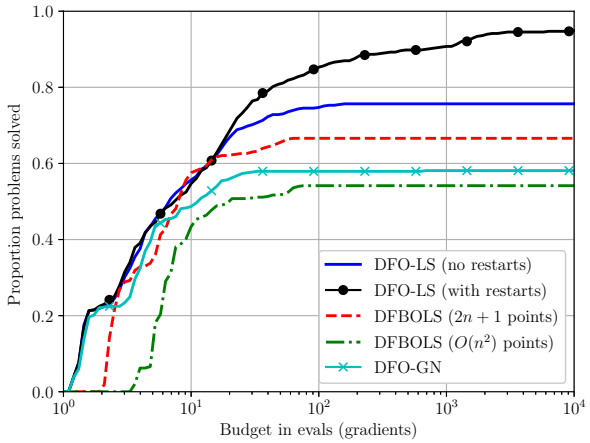
Figure 9. Comparison of the basic implementation of DFO-LS (using $n+1$ interpolation points) with DFBOLS and DFO-GN for smooth objective evaluations and high accuracy $\tau = 10^{-5}$. For DFBOLS, $2n+1$ and $\mathcal{O}(n^2) = (n+1)(n+2)/2$ are the number of interpolation points. For DFO-LS, results are an average of 10 runs. In (b), we show results using the full initialization cost of $n+1$ evaluations, and a reduced cost of 2 evaluations (using the SVD method).



(a) Multiplicative Gaussian noise

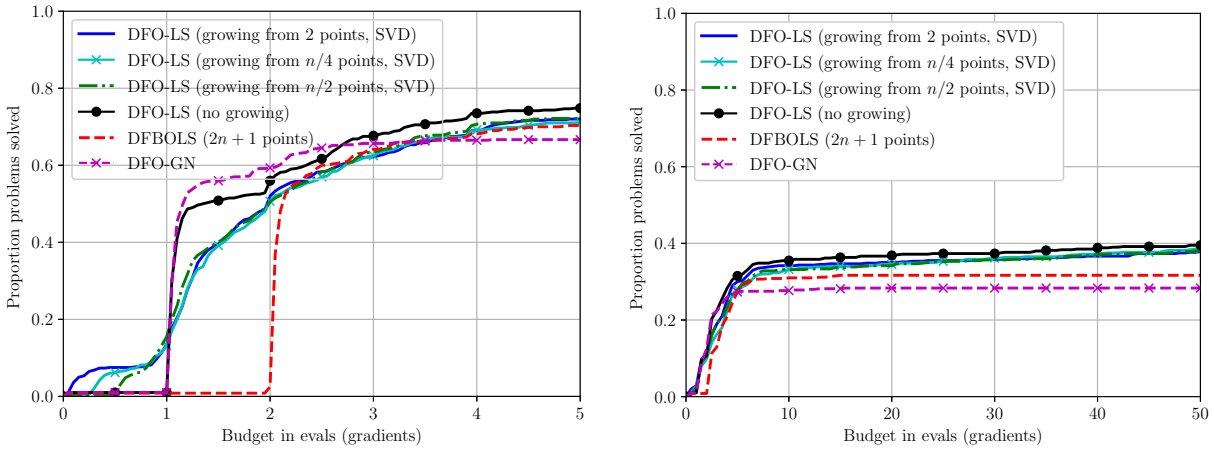


(b) Additive Gaussian noise



(c) Additive χ^2 noise

Figure 10. Comparison of the basic implementation of DFO-LS (using $n+1$ interpolation points) with DFBOLS and DFO-GN for noisy objective evaluations with $\sigma = 10^{-2}$ and high accuracy $\tau = 10^{-5}$. For DFBOLS, $2n+1$ and $\mathcal{O}(n^2) = (n+1)(n+2)/2$ are the number of interpolation points. Results shown are an average of 10 runs for each solver. The problem collection is (MW).



(a) Short budget, $\tau = 10^{-1}$

(b) Long budget, $\tau = 10^{-5}$

Figure 11. Comparison of the reduced initialization cost of DFO-LS (using $n+1$ interpolation points, SVD method) against using the full initial set, DFBOLS and DFO-GN, for objectives with additive Gaussian noise, $\sigma = 10^{-2}$. For DFBOLS, $2n+1$ is the number of interpolation points. Results are an average of 10 runs in each case. The problem collection is (CR).

region decrease parameters, multiple restarts). The results are very similar to the smooth case (see Figures 4 and 9b): the reduced initialization cost allows progress to be made within $n+1$ objective evaluations for some problems, at the cost of reduced small-budget performance, but achieves similar overall robustness, with performance for long budgets at high accuracy levels.

Multiple Restarts for Noiseless Problems Although the multiple restarts feature is designed for noisy problems, it can also be useful for smooth objectives. In Figure 12a, we show Figure 9a, but including results for DFO-LS with soft (moving $\mathbf{x}_k$) and hard restarts[16]. Both restart mechanisms provide a slight improvement — for most problems, the restarts give similar performance (although using the full computational budget allowed), but in some cases they are beneficial.

The first possible benefit of multiple restarts is being able to escape local minima. In Figure 12b, we show the objective value $f(\mathbf{x}_k)$ for one run of DFO-LS with soft restarts for problem 14 in (MW); the vertical lines show where restarts occurred. This problem has two local minima, with $f(\mathbf{x}^*) \approx 48.98$ and $f(\mathbf{x}^*) = 0$ [25]. We see that when the first restart occurs, we have found the local minimum with higher objective value — this is when DFO-LS would usually terminate. However, if we allow DFO-LS to perform three soft restarts, it manages to find the other local minimum (which is also the global minimum).

The other possible benefit is a faster convergence rate. In Figure 12c, we consider problem 18 in (MW), and we again show $f(\mathbf{x}_k)$ for DFO-LS without restarts, and with hard restarts. For this problem, DFO-LS with the default settings terminates on the 'slow progress' termination criterion (see Section 3.1; the solid circle in the plot), but we show how DFO-LS without restarts continues to make progress when this criterion is disabled. We also show DFO-LS with hard restarts; in this case, we keep the 'slow progress' termination criterion, and this triggers a restart. We can see that eventually, the run with multiple restarts finds better objective values, and seems to be converging at a faster asymptotic rate than DFO-LS without restarts.

7 Py-BOBYQA: DFO for General Objective Problems

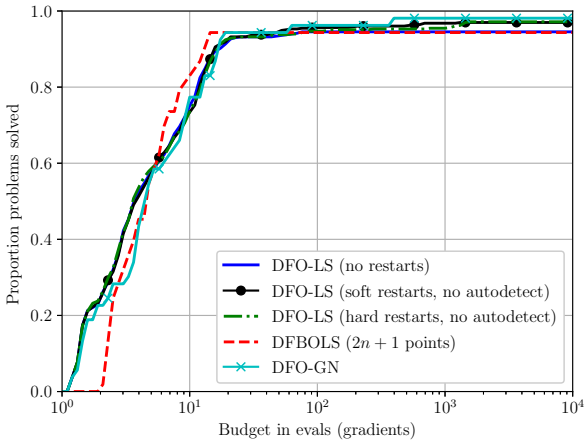
In this section we consider the case of general objective problems; that is,

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), \tag{7.1}$$

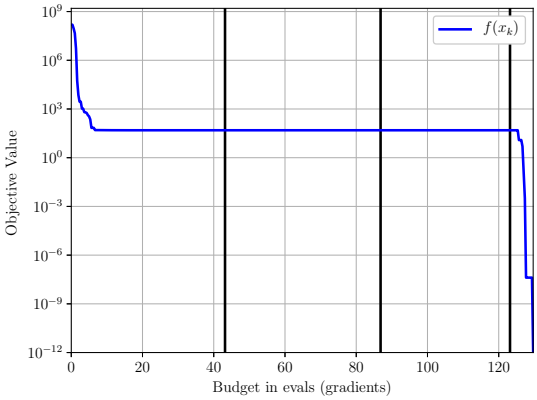
for some sufficiently smooth f with unknown structure. We call our solver Py-BOBYQA, as it is a Python-based solver which is very similar to Powell's (Fortran) BOBYQA [31].

The overall algorithmic structure of (Py-)BOBYQA is the same as Algorithm 1: we construct an interpolation-based model for f, calculate a step to minimize this model inside a trust region, and perform one of several phases (safety, successful, model-improving, unsuccessful) depending on the outcome. The

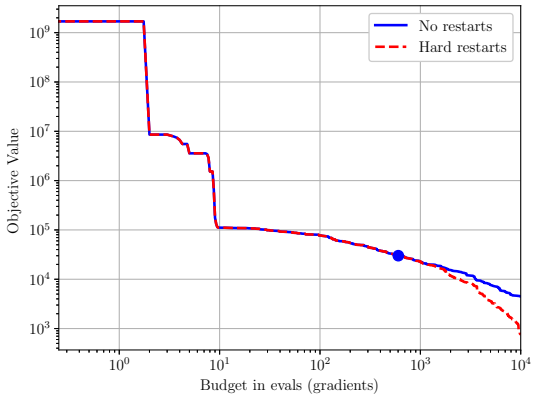
[16] For noiseless problems, we do not use the autodetection of restarts feature from Section 3.2.



(a) Data Profile, $\tau = 10^{-5}$



(b) Objective reduction, problem 14 (soft restarts)



(c) Objective reduction, problem 18

Figure 12. Illustration of the impacts of multiple restarts for noiseless problems. Figure (a) is the same as Figure 9a, but also showing DFO-LS with soft and hard restarts, without use of autodetection (problem collection (MW)). Figure (b) shows the objective value $f(\mathbf{x}_k)$ using DFO-LS with soft restarts (moving $\mathbf{x}_k$), for (MW) problem 14; the vertical lines indicate where restarts occurred. Figure (c) shows the objective value using DFO-LS with and without hard restarts, for (MW) problem 18. The dot indicates where the default 'slow decrease' termination criterion is triggered; the rest of the results for the 'no restarts' case are found by disabling this criterion.

most important difference is that the model $m_k(\mathbf{s}) \approx f(\mathbf{x}_k+\mathbf{s})$ is built by directly interpolating $f(\mathbf{y}_t)$ for $\mathbf{y}_t \in Y_k$ and is typically quadratic. Specifically, for an interpolation set of size $|Y_k| \in \{n+1, \ldots, (n+1)(n+2)/2\}$, we construct

$$m_k(\mathbf{s}) = c_k + \mathbf{g}_k^\top \mathbf{s} + \frac{1}{2}\mathbf{s}^\top H_k \mathbf{s}, \tag{7.2}$$

satisfying the interpolation (not regression) conditions

$$m_k(\mathbf{y}_t - \mathbf{x}_k) = f(\mathbf{y}_t), \quad \text{for all } \mathbf{y}_t \in Y_k. \tag{7.3}$$

If $|Y_k| < (n+1)(n+2)/2$, the solution to (7.3) is non-unique; following [31] we use the remaining degrees of freedom by choosing $H_k = 0$ if $|Y_k| = n+1$, and solving

$$\min_{c_k, \mathbf{g}_k, H_k} \|H_k - H_{k-1}\|_F^2 \quad \text{subject to (7.3)}, \tag{7.4}$$

otherwise. The value of $|Y_k|$ is a user-specified input, which defaults to $2n+1$ for smooth problems and $(n+1)(n+2)/2$ for noisy problems.

Simplifications from original BOBYQA For the purposes of a simplified code, and to be more closely aligned with DFO-LS, we simplify the model construction process in Py-BOBYQA as compared to its original implementation in [31]. Specifically, in [30], it was noted that changing a single interpolation point yielded a low-rank update to the linear system corresponding to (7.4). This, together with a well-chosen system for

building Y_0, meant that the linear system for (7.4) was never solved directly; instead, a factorization of the corresponding matrix inverse was maintained at all iterations, and updated using the Sherman-Morrison-Woodbury formula. By contrast, in Py-BOBYQA, as in DFO-LS, we use random directions to build Y_0, and construct the model by solving the linear system resulting from (7.4) at every iteration.

Improvements from original BOBYQA The goal of implementing Py-BOBYQA was to endow it with some of the key features from DFO-LS in order to improve its robustness to noise. Given the extra complexity of managing quadratic rather than linear models, we transferred the features from DFO-LS which did not require a large redesign of the model construction routines. Specifically, Py-BOBYQA contains the following new features:

- The user can specify $|Y_k| = n + 1$, compared to $|Y_k| \geq n + 2$ as required by BOBYQA;
- Larger range of termination conditions, as per Section 3.1. The changes are that the 'small objective value' threshold is just $f(\mathbf{x}_k) \leq \epsilon_{abs}$, as we no longer have $f \geq 0$ guaranteed, and for the same reason the slow decrease condition (3.2) in Py-BOBYQA uses $f(\mathbf{x}_{k_{(i-K)}}) - f(\mathbf{x}_{k_i})$ rather than log-decrease;
- Flexible choice of algorithm parameters, including setting different default values for noisy problems, as per Section 3.1;
- Sample averaging using (3.5), as per Section 3.2; and
- Multiple restarts as per Section 3.2 (both soft and hard restarts). However, the automatic detection of restarts uses linear fits for both $\{(k, \log \|\mathbf{g}_k - \mathbf{g}_{k-1}\|)\}$ and $\{(k, \log \|H_k - H_{k-1}\|_F)\}$ instead of $\{(k, \log \|J_k - J_{k-1}\|_F)\}$ in DFO-LS.

Numerical Results for Smooth Problems Figure 13 compares the basic implementation of Py-BOBYQA v1.0.1 (no sample averaging or restarts) with the original BOBYQA [31] and NOWPAC [2] for smooth problems. We use the (MW) problem set, and a third collection of test problems:

(CFMR) The set of 60 nonlinear least-squares problems (CR), and the 30 general-objective problems from CUTEst listed in Appendix B. These extra problems are also medium-sized, with $50 \leq n \leq 110$.

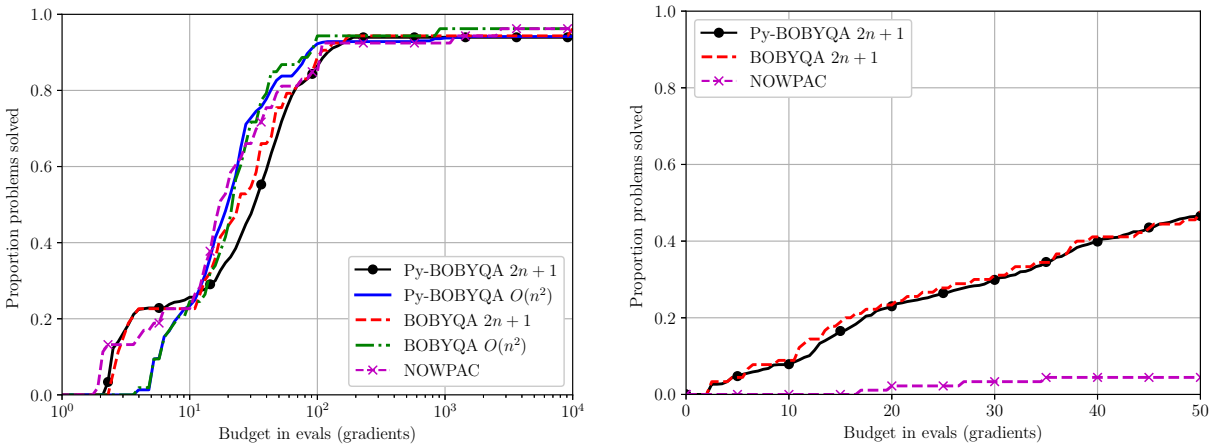
We also use the same budget and trust region radii settings as in Section 6 (as described in Section 4.2), and our budget is $50(n + 1)$ evaluations for (CFMR), like for the (CR) test problems.

For (Py-)BOBYQA applied to (MW), we show results for the default choice $|Y_k| = 2n + 1$, as well as the maximum value $|Y_k| = (n + 1)(n + 2)/2$, which is Py-BOBYQA's default choice for noisy problems. We do not show the $|Y_k| = (n + 1)(n + 2)/2$ results for (CFMR), because the small budget and high dimension means that almost all of the budget would be used by the initialization phase. We see that Py-BOBYQA has comparable performance with BOBYQA and NOWPAC for smooth problems, which we expect given the similarity of the algorithms. Due to the size of the (CFMR) problems, we allowed Py-BOBYQA and NOWPAC to run for a maximum of 12 hours per problem.

Numerical Results for Noisy Problems In Figure 14, we compare Py-BOBYQA with BOBYQA, STORM for unbiased noise [8][17] and SNOWPAC [3]. Here, in line with the rest of the paper, we show results for the (MW) set only, and use Py-BOBYQA's noisy default of $|Y_k| = (n + 1)(n + 2)/2$ for both Py-BOBYQA and BOBYQA. We also show Py-BOBYQA with $|Y_k| = 2n+1$ (the default for smooth problems), for comparison purposes. As the slowest solver to run, we allowed SNOWPAC to run for a maximum of 12 hours per problem.

Since SNOWPAC uses points from the full history of observations of the objective to construct a Gaussian Process surrogate model, its performance can slow down rapidly as the computational budget is increased; as a result, we only update the surrogate model every $5n$ iterations. Similar to our results for DFO-LS, we see that using multiple restarts gives a substantial improvement in the robustness of Py-BOBYQA, and it performs substantially better than BOBYQA. We also see the benefit of using a larger value of $|Y_k|$ for Py-BOBYQA, for noisy problems; as observed in [33], choosing a smaller $|Y_k|$ is useful for small budgets, but it ultimately leads to a reasonable reduction in robustness. Hence, we use the maximal value $|Y_k| = (n + 1)(n + 2)/2$ as the default choice in Py-BOBYQA for noisy problems, but $2n + 1$ for smooth problems, where the robustness issue does not appear (see discussion of Figure 13 in "Numerical Results for Smooth Problems" above). In our experiments, Py-BOBYQA can solve more problems than STORM within

[17] As mentioned in the introduction, there are several variants of STORM proposed in [8]. We chose this version because it showed better performance than other variants.



(a) Problem collection (MW) *(b) Problem collection (CFMR)*

Figure 13. Comparison of the basic implementation of Py-BOBYQA with the original Fortran BOBYQA and NOWPAC for smooth objective evaluations and high accuracy $\tau = 10^{-5}$. For (Py-)BOBYQA, $2n+1$ and $\mathcal{O}(n^2) = (n+1)(n+2)/2$ are the number of interpolation points. For Py-BOBYQA, results are an average of 10 runs.

the computational budget, and can solve many problems much more efficiently. We note that STORM relies on constructing models which are entirely independent at each iteration, so it takes many more evaluations to begin seeing the desired objective reductions. Compared to SNOWPAC, which uses both objective values and noise standard errors from each evaluation, Py-BOBYQA performs either comparably or better, with the difference most noticeable for multiplicative noise. The multiple restarts approach in Py-BOBYQA has the advantage of not requiring extra user input, and being cheap to implement compared to constructing a surrogate model.

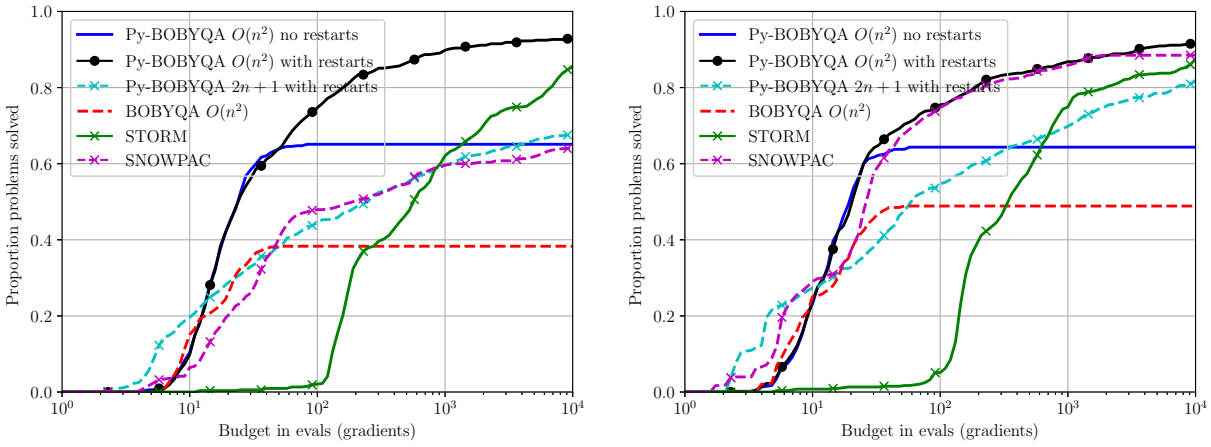
To illustrate the relative cost of multiple restarts compared to building surrogate models, Figure 15 shows the runtime[18] for Py-BOBYQA and SNOWPAC for two different noisy problems from (MW), using the large budget of $10^4(n+1)$ objective evaluations and additive Gaussian noise. For each problem, we imposed a timeout on each solver after 12 hours, and mark when each solver achieved the particular objective reduction $\tau_{crit}(p)$; see (4.8). For both problems, the runtime of Py-BOBYQA grows linearly with the number of objective evaluations, after the initial setup cost of $\mathcal{O}(n^2)$ evaluations. However SNOWPAC's runtime starts to grow much more quickly for large budgets. In SNOWPAC, the number of points used to build the surrogate model — which drives the cost of surrogate model construction — depends on the number of evaluated points in the entire run which are sufficiently close to $\mathbf{x}_k$. In many cases, this means the rapid increase in runtime occurs in the asymptotic regime, when a good solution has already been found (i.e. accuracy $\tau_{crit}(p)$ has been achieved), and SNOWPAC is trying to improve the quality of the solution using a more accurate surrogate. This occurs in problem 1, for instance, where $\tau_{crit}(p) = 10^{-2}$, and SNOWPAC achieves this accuracy well before the runtime starts to grow quickly. However, problem 53 is an example where the increase in runtime comes before this high accuracy regime: we have $\tau_{crit}(p) = 10^{-13}$, and SNOWPAC terminates (from the timeout) without achieving accuracy 10^{-4}. By comparison, on this problem, Py-BOBYQA terminates on maximum budget after reaching the much higher accuracy level $\tau = 10^{-11}$ before terminating (on budget). Overall, the use of a surrogate model is beneficial for achieving robustness to noise, but may result in reduced performance in order to realise this benefit.

Multiple Restarts for Noiseless Problems Similar to DFO-LS (see Section 6), we conclude by illustrating that there may also be some benefit in using multiple restarts when running Py-BOBYQA on smooth problems[19]. As before, since the first run of Py-BOBYQA with restarts is the same as the full solver run without restarts, there is no performance loss from using multiple restarts (although more of the computational budget is used). In Figure 16a, we compare Py-BOBYQA without restarts against soft (moving $\mathbf{x}_k$) and hard restarts for the (MW) collection. As expected, at this accuracy level, multiple restarts either gives the same or slightly better robustness than no restarts — the improvement is larger when using $(n+1)(n+2)/2$ interpolation points.

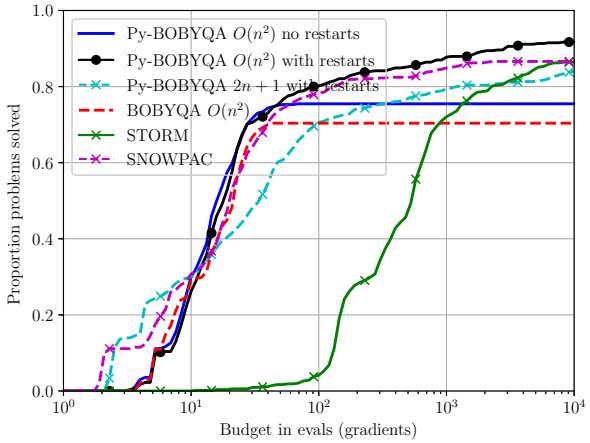
However, as for DFO-LS, we find that multiple restarts may help Py-BOBYQA to escape local minima.

[18] CPU time, measured on a Lenovo ThinkCentre M900 (with one 64-bit Intel i5 processor, 8GB of RAM).

[19] Unlike Section 6, we do not consider reduced initialization cost for noisy problems, as Py-BOBYQA does not have this feature.

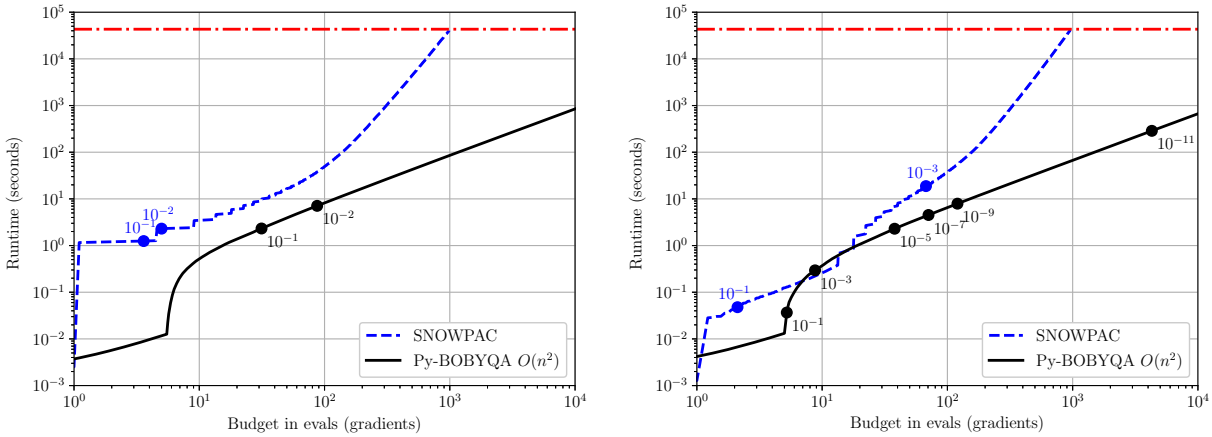


(a) Multiplicative Gaussian noise *(b) Additive Gaussian noise*



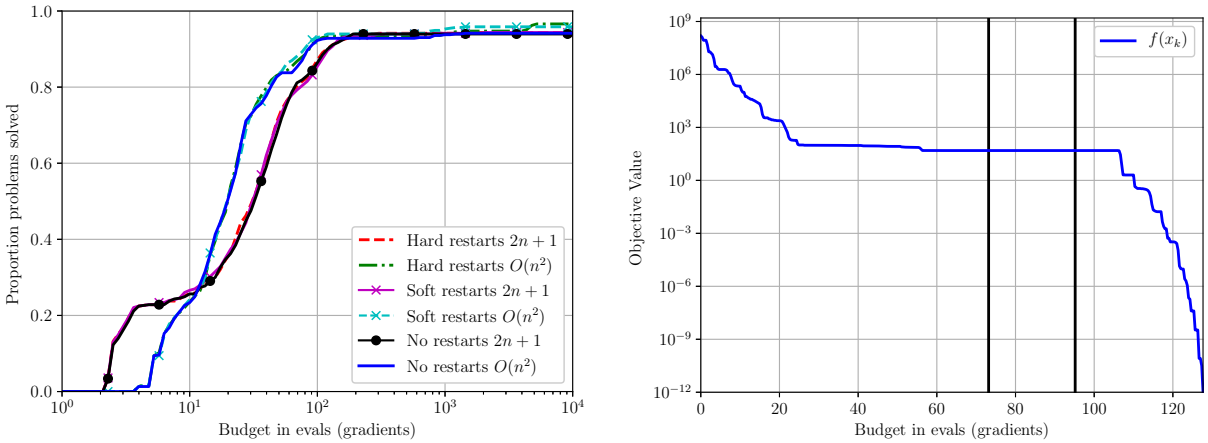
(c) Additive χ^2 noise

Figure 14. Comparison of the basic implementation of Py-BOBYQA with the original Fortran BOBYQA, STORM and SNOWPAC for noisy objective evaluations with $\sigma = 10^{-2}$ and high accuracy $\tau = 10^{-5}$. For (Py-)BOBYQA, $2n+1$ and $\mathcal{O}(n^2) = (n+1)(n+2)/2$ are the number of interpolation points. Results shown are an average of 10 runs for each solver. The problem collection is (MW).



(a) Problem 1, $\tau_{crit}(p) = 10^{-2}$ *(b) Problem 53, $\tau_{crit}(p) = 10^{-13}$*

Figure 15. Comparison of average runtimes — up to a maximum of 12 hours (horizontal dot-dash line) — for Py-BOBYQA (with $(n+1)(n+2)/2$ interpolation points and multiple restarts) and SNOWPAC, for two problems from (MW). The marked points are average budget/runtime when each solver achieved the labelled objective reduction τ. Both problems had additive Gaussian noise with $\sigma = 10^{-2}$. Results shown are an average of 10 runs for each solver.



(a) Data Profile, $\tau = 10^{-5}$

(b) Objective reduction, problem 14 (soft restarts)

Figure 16. Illustration of the impacts of multiple restarts for Py-BOBYQA on noiseless problems. Figure (a) is the same as Figure 13a, but only showing Py-BOBYQA without restarts, and with soft (moving $\mathbf{x}_k$) and hard restarts, without use of autodetection (problem collection (MW)). Figure (b) shows the objective value $f(\mathbf{x}_k)$ using Py-BOBYQA with soft restarts and $2n+1$ interpolation points, for (MW) problem 14; the vertical lines indicate where restarts occurred.

In Figure 16b, we show the objective value $f(\mathbf{x}_k)$ for one run of Py-BOBYQA with soft restarts and $2n+1$ interpolation points for problem 14 in (MW) — this is the same as Figure 12b for DFO-LS. As before, we see that the first run of Py-BOBYQA finds the local minimum $f(\mathbf{x}^*) \approx 48.98$, but after two restarts, it manages to escape and find the global minimum $f(\mathbf{x}^*) = 0$.

8 Conclusion

We have presented two model-based DFO routines: DFO-LS for nonlinear least-squares problems, and Py-BOBYQA for general objective problems, both with optional bound constraints. Both routines perform comparably to or better than state-of-the-art solvers on noisy problems with large, inexpensive budgets. This is due to their ability to select different, more appropriate, algorithm parameters for noisy problems, and their use of multiple restarts. Compared to other techniques for improving robustness to noise, such as sample averaging, regression models, and surrogate models, multiple restarts are cheap to implement and do not cause a deterioration in performance in the early phase of the algorithm. However, both codes also allow the user to employ a wide family of sample averaging strategies, and DFO-LS additionally allows the use of regression models. Although multiple restarts are designed for noisy problems, they do not disadvantage performance on smooth problems and can sometimes even improve it, such as when allowing the algorithm to escape local minima.

In addition, DFO-LS has the ability to use a reduced initialization cost of as few as 2 objective evaluations, after which the main iteration can begin, rather than at least $n+1$ as in many model-based DFO codes (for an n-dimensional problem). This is a useful feature when objective evaluations are expensive, and can be used for noisy and noiseless objectives alike. By reducing the initialization cost in this way, reasonable progress can be made on several problems even with fewer than n objective evaluations (i.e. less than the cost of evaluating the gradient of the objective at a single point). This improvement has a tradeoff in performance for medium-sized budgets, but achieves the same long-term performance as having a full initialization cost.

Throughout, we have shown results for noisy problems using a problem- and noise-adjusted accuracy level. This adjustment is chosen so that the progress defined by decreases in the noisy and underlying smooth objective produce similar results. Therefore, this approach may be a useful way of benchmarking solvers for noisy problems, by focusing on a regime where progress as measured in the noisy objective (which is seen by the solver/user) corresponds to genuine optimization steps, and not luck in sampling errors.

8.1 Acknowledgements

This work was supported by the EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in collaboration with the Numerical Algorithms Group Ltd. We would like to thank Michael Ferris, Nick Gould, Raphael Hauser, Katya Scheinberg and Amy Willis for useful discussions

regarding the DFO-LS algorithm, measuring solver performance, and comparing averaging and regression models. We also acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility[20] in carrying out this work.

References

[1] C. Audet and W. Hare, *Derivative-Free and Blackbox Optimization*, Springer Series in Operations Research and Financial Engineering, Springer, Cham, Switzerland, 2017.

[2] F. Augustin and Y. M. Marzouk, *NOWPAC: A provably convergent derivative-free nonlinear optimizer with path-augmented constraints*, arXiv Prepr. arXiv:1403.1931, (2014).

[3] ——, *A trust-region method for derivative-free nonlinear constrained stochastic optimization*, arXiv Prepr. arXiv:1703.04156, (2017).

[4] V. Beiranvand, W. Hare, and Y. Lucet, *Best practices for comparing optimization algorithms*, Optim. Eng., 18 (2017), pp. 815–848.

[5] S. C. Billups, J. W. Larson, and P. Graf, *Derivative-Free Optimization of Expensive Functions with Computational Error Using Weighted Regression*, SIAM J. Optim., 23 (2013), pp. 27–53.

[6] C. Cartis, J. Fiala, B. Marteau, and L. Roberts, *Improving the flexibility and robustness of model-based derivative-free optimization solvers*, tech. rep., Technical Report, University of Oxford, Mathematical Institute, 2018. Available on Optimization Online.

[7] C. Cartis and L. Roberts, *A derivative-free Gauss-Newton method*, tech. rep., University of Oxford, Mathematical Institute, 2017. Available on Optimization Online.

[8] R. Chen, M. Menickelly, and K. Scheinberg, *Stochastic optimization using a trust-region method and random models*, arXiv Prepr. arXiv:1504.04231, (2016).

[9] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust-Region Methods*, MPS-SIAM Series on Optimization, MPS/SIAM, Philadelphia, 2000.

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Geometry of sample sets in derivative-free optimization: Polynomial regression and underdetermined interpolation*, IMA J. Numer. Anal., 28 (2008), pp. 721–748.

[11] ——, *Introduction to Derivative-Free Optimization*, vol. 8 of MPS-SIAM Series on Optimization, MPS/SIAM, Philadelphia, 2009.

[12] A. L. Custódio, J. F. A. Madeira, A. I. F. Vaz, and L. N. Vicente, *Direct multisearch for multiobjective optimization*, SIAM J. Optim., 21 (2011), pp. 1109–1140.

[13] A. L. Custódio, K. Scheinberg, and L. N. Vicente, *Methodologies and Software for Derivative-free Optimization*, in Adv. Trends Optim. with Eng. Appl., T. Terlaky, M. F. Anjos, and S. Ahmed, eds., MOS-SIAM Book Series on Optimization, SIAM, Philadelphia, 2017.

[14] J. W. Demmel, *Applied Numerical Linear Algebra*, Other Titles in Applied Mathematics, SIAM, Philadelphia, 1997.

[15] G. Deng and M. C. Ferris, *Adaptation of the UOBYQA algorithm for noisy functions*, in Proc. 2006 Winter Simul. Conf., L. F. Peronne, F. P. Weiland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds., 2006, pp. 312–319.

[16] ——, *Variable-number sample-path optimization*, Math. Program., 117 (2009), pp. 81–109.

[17] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Math. Program., 91 (2002), pp. 201–213.

[18] N. I. M. Gould, D. Orban, and P. L. Toint, *CUTEst: a Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization*, Comput. Optim. Appl., 60 (2015), pp. 545–557.

[20] `http://dx.doi.org/10.5281/zenodo.22558`

[19] G. N. Grapiglia, J. Yuan, and Y.-x. Yuan, *A derivative-free trust-region algorithm for composite nonsmooth optimization*, Comput. Appl. Math., 35 (2016), pp. 475–499.

[20] W. Hare, J. Loeppky, and S. Xie, *Methods to compare expensive stochastic optimization algorithms with random restarts*, J. Glob. Optim., 72 (2018), pp. 781–801.

[21] C. T. Kelley, *Detection and Remediation of Stagnation in the Nelder–Mead Algorithm Using a Sufficient Decrease Condition*, SIAM J. Optim., 10 (1999), pp. 43–55.

[22] J. W. Larson and S. M. Wild, *Non-intrusive termination of noisy optimization*, Optim. Methods Softw., 28 (2013), pp. 993–1011.

[23] M. Locatelli and F. Schoen, *Global Optimization: Theory, Algorithms, and Applications*, MOS-SIAM Series on Optimization, SIAM, Philadelphia, 2013.

[24] L. Lukšan, C. Matonoha, and J. Vlček, *Modified CUTE Problems for Sparse Unconstrained Optimization*, tech. rep., Academy of Sciences of the Czech Republic, 2010.

[25] J. J. Moré, B. S. Garbow, and K. E. Hillstrom, *Testing Unconstrained Optimization Software*, ACM Trans. Math. Softw., 7 (1981), pp. 17–41.

[26] J. J. Moré and S. M. Wild, *Benchmarking Derivative-Free Optimization Algorithms*, SIAM J. Optim., 20 (2009), pp. 172–191.

[27] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer Series in Operations Research and Financial Engineering, Springer, New York, 2nd ed., 2006.

[28] M. Porcelli and P. L. Toint, *BFO, A Trainable Derivative-free Brute Force Optimizer for Nonlinear Bound-constrained Optimization and Equilibrium Computations*, ACM Trans. Math. Softw., 44 (2017), pp. 6:1–6:25.

[29] M. J. D. Powell, *On trust region methods for unconstrained minimization without derivatives*, Math. Program., 97 (2003), pp. 605–623.

[30] ——, *Least Frobenius norm updating of quadratic models that satisfy interpolation conditions*, Math. Program., 100 (2004), pp. 183–215.

[31] ——, *The BOBYQA algorithm for bound constrained optimization without derivatives*, Tech. Rep. DAMTP 2009/NA06, University of Cambridge, 2009.

[32] R. G. Regis, *The calculus of simplex gradients*, Optim. Lett., 9 (2015), pp. 845–865.

[33] L. Roberts, *Derivative-free optimisation for data fitting*, Tech. Rep. InFoMM CDT Report, University of Oxford, 2016. `http://people.maths.ox.ac.uk/robertsl/docs/DFO_MiniprojectReport_updateNov18.pdf`.

[34] K. Scheinberg and P. L. Toint, *Self-Correcting Geometry in Model-Based Algorithms for Derivative-Free Unconstrained Optimization*, SIAM J. Optim., 20 (2010), pp. 3512–3532.

[35] Y. D. Sergeyev, D. E. Kvasov, and M. S. Mukhametzhanov, *Operational zones for comparing metaheuristic and deterministic one-dimensional global optimization algorithms*, Mathematics and Computers in Simulation, 141 (2017), pp. 96–109.

[36] S. Shashaani, F. S. Hashemi, and R. Pasupathy, *ASTRO-DF: A class of adaptive sampling trust-region algorithms for derivative-free stochastic optimization*, arXiv Prepr. arXiv:1610.06506, (2016).

[37] S. M. Wild, *POUNDERS in TAO: Solving Derivative-Free Nonlinear Least-Squares Problems with POUNDERS*, in Adv. Trends Optim. with Eng. Appl., SIAM, Philadelphia, PA, 2017, ch. 40, pp. 529–539.

[38] H. Zhang, A. R. Conn, and K. Scheinberg, *A Derivative-Free Algorithm for Least-Squares Minimization*, SIAM J. Optim., 20 (2010), pp. 3555–3576.

[39] Z. Zhang, *Software by Professor M. J. D. Powell*. `http://mat.uc.pt/~zhang/software.html`, 2017.

A Convergence Guarantees for DFO-LS

In this section, we provide details of the convergence theory for the DFO-LS algorithm. These results largely follow the arguments in [7, Section 3].

A.1 Accuracy of Regression Models

In Section 2.2, we introduced Λ-poisedness as the key measure of the quality of the geometry of Y_k. The Λ-poisedness of Y_k guarantees accuracy of the regression models $\mathbf{m}_k$ (2.2) and m_k (2.5), in the following sense [11, 19]:

Definition A.1 (Fully linear, scalar model)**.** A model $m_k \in C^1$ for a scalar function $f \in C^1$ is fully linear in $B(\mathbf{x}_k, \Delta_k)$ if there exist positive constants κ_{ef} and κ_{eg}, independent of $\mathbf{x}_k$ and Δ_k such that

$$|m_k(\mathbf{s}) - f(\mathbf{x}_k + \mathbf{s})| \leq \kappa_{ef}\Delta_k^2, \tag{A.1}$$

$$\|\nabla m_k(\mathbf{s}) - \nabla f(\mathbf{x}_k + \mathbf{s})\| \leq \kappa_{eg}\Delta_k, \tag{A.2}$$

for all $\|\mathbf{s}\| \leq \Delta_k$.

Definition A.2 (Fully linear, vector model)**.** A model $\mathbf{m}_k \in C^1$ for a vector function $\mathbf{r} \in C^1$ is fully linear in $B(\mathbf{x}_k, \Delta_k)$ if there exist positive constants κ_{ef}^r and κ_{eg}^r, independent of $\mathbf{x}_k$ and Δ_k such that

$$\|\mathbf{m}_k(\mathbf{s}) - \mathbf{r}(\mathbf{x}_k + \mathbf{s})\| \leq \kappa_{ef}^r\Delta_k^2, \tag{A.3}$$

$$\|J^m(\mathbf{s}) - J(\mathbf{x}_k + \mathbf{s})\| \leq \kappa_{eg}^r\Delta_k, \tag{A.4}$$

for all $\|\mathbf{s}\| \leq \Delta_k$, where J^m and J are the Jacobians of $\mathbf{m}_k$ and $\mathbf{r}$ respectively.

To establish the connection between Λ-poisedness of Y_k and full linearity of our regression interpolation models $\mathbf{m}_k$ and m_k, we require extra assumptions on the smoothness of the objective.

Assumption A.3. The function $\mathbf{r}$ is C^1 and its Jacobian $J(\mathbf{x})$ is Lipschitz continuous in $\mathcal{B}$, the convex hull of $\cup_k B(\mathbf{x}_k, \Delta_{max})$, with constant L_J. We also assume that $\mathbf{r}(\mathbf{x})$ and $J(\mathbf{x})$ are uniformly bounded in the same region; i.e. $\|\mathbf{r}(\mathbf{x})\| \leq r_{max}$ and $\|J(\mathbf{x})\| \leq J_{max}$ for all $\mathbf{x} \in \mathcal{B}$.

If Assumption A.3 holds, then ∇f is Lipschitz continuous in $\mathcal{B}$ with constant $L_{\nabla f} := r_{max}L_J + J_{max}^2$ [7, Lemma 3.3]. The main result, analogous to [7, Lemma 3.4], is the following.

Lemma A.4. *Suppose Assumption A.3 holds, and Y_k with $|Y_k| = p + 1$ is Λ-poised in $B(\mathbf{x}_k, \Delta_k)$ in the regression sense. Then $\mathbf{m}_k$ and m_k are fully linear models in $B(\mathbf{x}_k, \Delta_k)$ for $\mathbf{r}$ and f respectively, with constants*

$$\kappa_{ef}^r = 2\kappa_{eg}^r, \tag{A.5}$$

$$\kappa_{eg}^r = \frac{1}{2}L_J\left(\sqrt{p}\,C + 2\right), \tag{A.6}$$

$$\kappa_{ef} = \kappa_{eg} + L_{\nabla f}/2 + \left(2r_{max} + \kappa_{eg}^r\Delta_{max}\right)\kappa_{eg}^r + (\kappa_{eg}^r\Delta_{max} + J_{max})^2, \tag{A.7}$$

$$\kappa_{eg} = L_{\nabla f} + 2J_{max}\kappa_{eg}^r\Delta_{max} + 2\kappa_{eg}^r r_{max} + 2(\kappa_{eg}^r\Delta_{max} + J_{max})^2, \tag{A.8}$$

where $C = \mathcal{O}(\Lambda)$.

Proof. This result extends the proof[21] of [11, Theorem 2.13] and [32, Proposition 7] to vector-valued functions, and their composition in a least-squares objective, in the style of [7, Lemma 3.4].

Using $\mathbf{y}_0 = \mathbf{x}_k$, we may write

$$W_k := \begin{bmatrix} 1 & (\mathbf{y}_0 - \mathbf{x}_k)^\top \\ \vdots & \vdots \\ 1 & (\mathbf{y}_p - \mathbf{x}_k)^\top \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{e} & L_k \end{bmatrix}, \tag{A.9}$$

where $\mathbf{e} \in \mathbb{R}^p$ is the vector of ones and $L_k \in \mathbb{R}^{p\times n}$ has rows $(\mathbf{y}_t - \mathbf{x}_k)^\top$ for $t = 1, \ldots, p$.

We will also use scaled versions of these matrices:

$$\hat{W}_k := \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{e} & \hat{L}_k \end{bmatrix}, \quad \text{where} \quad \hat{L}_k := \begin{bmatrix} (\mathbf{y}_1 - \mathbf{x}_k)^\top/\Delta_k \\ \vdots \\ (\mathbf{y}_p - \mathbf{x}_k)^\top/\Delta_k \end{bmatrix} = L/\Delta_k. \tag{A.10}$$

[21] This argument is not in the original text, but given in the errata (`http://www.mat.uc.pt/~lnv/idfo/`).

Equivalently, we have

$$\hat{W}_k = W_k D_k, \quad \text{where} \quad D_k := \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & \frac{1}{\Delta_k} I_{n\times n} \end{bmatrix}. \tag{A.11}$$

To begin, we recall that if $J(\mathbf{x})$ is continuous with Lipschitz constant L_J, then [27, Appendix A]

$$\|\mathbf{r}(\mathbf{y}) - \mathbf{r}(\mathbf{x}_k) - J(\mathbf{x}_k)(\mathbf{y} - \mathbf{x}_k)\| \leq \frac{1}{2} L_J \|\mathbf{y} - \mathbf{x}_k\|^2. \tag{A.12}$$

Our overdetermined interpolation system (2.4) can be rewritten in matrix form as

$$W_k \begin{bmatrix} \mathbf{r}_k^\top \\ J_k^\top \end{bmatrix} = \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ \mathbf{r}(\mathbf{y}_1)^\top \\ \vdots \\ \mathbf{r}(\mathbf{y}_p)^\top \end{bmatrix}, \quad \text{and so} \quad \begin{bmatrix} \mathbf{r}_k^\top \\ J_k^\top \end{bmatrix} = W_k^\dagger \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ \mathbf{r}(\mathbf{y}_1)^\top \\ \vdots \\ \mathbf{r}(\mathbf{y}_p)^\top \end{bmatrix}. \tag{A.13}$$

Separately, we compute

$$W_k \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ J(\mathbf{x}_k)^\top \end{bmatrix} - \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ \mathbf{r}(\mathbf{y}_1)^\top \\ \vdots \\ \mathbf{r}(\mathbf{y}_p)^\top \end{bmatrix} = \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ [\mathbf{r}(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{y}_1 - \mathbf{x}_k)]^\top \\ \vdots \\ [\mathbf{r}(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{y}_p - \mathbf{x}_k)]^\top \end{bmatrix} - \begin{bmatrix} \mathbf{r}(\mathbf{x}_k)^\top \\ \mathbf{r}(\mathbf{y}_1)^\top \\ \vdots \\ \mathbf{r}(\mathbf{y}_p)^\top \end{bmatrix} =: E. \tag{A.14}$$

Combining (A.13) and (A.14), we get

$$\begin{bmatrix} [\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k]^\top \\ [J(\mathbf{x}_k) - J_k]^\top \end{bmatrix} = W_k^\dagger E, \tag{A.15}$$

and so from (A.11), since D_k is invertible, we have $\hat{W}_k^\dagger = D_k^{-1} W_k^\dagger$ and hence conclude

$$\begin{bmatrix} [\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k]^\top \\ \Delta_k [J(\mathbf{x}_k) - J_k]^\top \end{bmatrix} = D_k^{-1} \begin{bmatrix} [\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k]^\top \\ [J(\mathbf{x}_k) - J_k]^\top \end{bmatrix} = \hat{W}_k^\dagger E. \tag{A.16}$$

Since the first row of $E \in \mathbb{R}^{(p+1)\times m}$ is zero, and the norms of all other rows are bounded by (A.12), we have

$$\|E\| \leq \|E\|_F = \left(0 + \sum_{t=1}^{p} \|\mathbf{r}(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{y}_t - \mathbf{x}_k) - \mathbf{r}(\mathbf{y}_t)\|^2 \right)^{1/2} \leq \frac{1}{2} L_J \sqrt{p} \Delta_k^2. \tag{A.17}$$

This gives us the error bounds

$$\|\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k\| \leq \left\| \begin{bmatrix} [\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k]^\top \\ \Delta_k [J(\mathbf{x}_k) - J_k]^\top \end{bmatrix} \right\| \leq \frac{1}{2} L_J \sqrt{p} \|\hat{W}_k^\dagger\| \Delta_k^2, \tag{A.18}$$

$$\|J(\mathbf{x}_k) - J_k\| \leq \Delta_k^{-1} \left\| \begin{bmatrix} [\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k]^\top \\ \Delta_k [J(\mathbf{x}_k) - J_k]^\top \end{bmatrix} \right\| \leq \frac{1}{2} L_J \sqrt{p} \|\hat{W}_k^\dagger\| \Delta_k. \tag{A.19}$$

Thus we conclude that for any $\mathbf{y} \in B(\mathbf{x}_k, \Delta_k)$

$$\|J_k - J(\mathbf{y})\| \leq \|J_k - J(\mathbf{x}_k)\| + \|J(\mathbf{y}) - J(\mathbf{x}_k)\| \leq L_J \left(1 + \frac{1}{2} \sqrt{p} \|\hat{W}_k^\dagger\| \right) \Delta_k. \tag{A.20}$$

For convenience, define $\kappa_{eg}^r := L_J \left(1 + \frac{1}{2} \sqrt{p} \|\hat{W}_k^\dagger\| \right)$. Next, we compute

$$\begin{aligned} \|\mathbf{m}_k(\mathbf{y} - \mathbf{x}_k) - \mathbf{r}(\mathbf{y})\| &= \|\mathbf{r}(\mathbf{y}) - \mathbf{r}_k - J_k(\mathbf{y} - \mathbf{x}_k)\|, & \text{(A.21)} \\ &\leq \|\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k\| + \|\mathbf{r}(\mathbf{y}) - \mathbf{r}(\mathbf{x}_k) - J(\mathbf{x}_k)(\mathbf{y} - \mathbf{x}_k)\| \\ &\qquad + \|J(\mathbf{x}_k) - J_k\| \cdot \|\mathbf{y} - \mathbf{x}_k\|, & \text{(A.22)} \\ &\leq \left(\frac{1}{2} L_J \sqrt{p} \|\hat{W}_k^\dagger\| + \frac{L_J}{2} + \kappa_{eg}^r \right) \Delta_k^2, & \text{(A.23)} \end{aligned}$$

where we use (A.20) and (A.12). Hence $\mathbf{m}_k$ is a fully linear model for $\mathbf{r}$ with constants κ_{eg}^r defined above and $\kappa_{ef}^r = 2\kappa_{eg}^r$.

Since $\mathbf{m}_k$ is fully linear, (A.20) gives us $\|J_k\| \leq \|J(\mathbf{x}_k) - J_k\| + \|J(\mathbf{x}_k)\| \leq \kappa_{eg}^r \Delta_{max} + J_{max}$, so $\|J_k\|$ is uniformly bounded for all k. Since $H_k = 2J_k^\top J_k$, we get that $\|H_k\| = 2\|J_k\|^2$ is uniformly bounded for all k. Similarly, using (A.18), we have the bounds $\|\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k\| \leq \kappa_{eg}^r \Delta_k^2$ and $\|\mathbf{r}_k\| \leq r_{max} + \kappa_{eg}^r \Delta_{max}^2$.

To prove full linearity of m_k, we begin by computing

$$\begin{aligned}
\|\nabla m_k(\mathbf{y} - \mathbf{x}_k) - \nabla f(\mathbf{y})\| &= \|\nabla f(\mathbf{y}) - 2J_k^\top \mathbf{r}_k - 2J_k^\top J_k(\mathbf{y} - \mathbf{x}_k)\|, && \text{(A.24)}\\
&\leq \|\nabla f(\mathbf{y}) - \nabla f(\mathbf{x}_k)\| + \|2J(\mathbf{x}_k)^\top (\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k)\| \\
&\qquad + \|2(J(\mathbf{x}_k) - J_k)^\top \mathbf{r}_k\| + \|2J_k^\top J_k\| \cdot \|\mathbf{y} - \mathbf{x}_k\|, && \text{(A.25)}\\
&\leq L_{\nabla f} \Delta_k + 2J_{max} \kappa_{eg}^r \Delta_k^2 \\
&\qquad + 2\kappa_{eg}^r r_{max} \Delta_k + 2\left(\kappa_{eg}^r \Delta_{max} + J_{max}\right)^2 \Delta_k, && \text{(A.26)}\\
&\leq \kappa_{eg} \Delta_k, && \text{(A.27)}
\end{aligned}$$

where $\kappa_{eg} := L_{\nabla f} + 2J_{max}\kappa_{eg}^r \Delta_{max} + 2\kappa_{eg}^r r_{max} + 2(\kappa_{eg}^r \Delta_{max} + J_{max})^2$, as required. Then, we use (A.12) and the above to get

$$\begin{aligned}
|m_k(\mathbf{y} - \mathbf{x}_k) - f(\mathbf{y})| &= \left| f(\mathbf{y}) - \|\mathbf{r}_k\|^2 - \mathbf{g}_k^\top (\mathbf{y} - \mathbf{x}_k) - \frac{1}{2}(\mathbf{y} - \mathbf{x}_k)^\top H_k (\mathbf{y} - \mathbf{x}_k) \right|, && \text{(A.28)}\\
&\leq \left| f(\mathbf{y}) - f(\mathbf{x}_k) - \nabla f(\mathbf{x}_k)^\top (\mathbf{y} - \mathbf{x}_k) \right| + \|\mathbf{r}(\mathbf{x}_k) - \mathbf{r}_k\| \left(\|\mathbf{r}(\mathbf{x}_k)\| + \|\mathbf{r}_k\| \right) \\
&\qquad + \left\| \nabla f(\mathbf{x}_k) - \mathbf{g}_k - \frac{1}{2} H_k (\mathbf{y} - \mathbf{x}_k) \right\| \cdot \|\mathbf{y} - \mathbf{x}_k\|, && \text{(A.29)}\\
&\leq \frac{1}{2} L_{\nabla f} \Delta_k^2 + \kappa_{eg}^r \Delta_k^2 \left(2r_{max} + \kappa_{eg}^r \Delta_{max} \right) \\
&\qquad + \left[\|\nabla f(\mathbf{x}_k) - \nabla m_k(\mathbf{x}_k)\| + \frac{1}{2}\|H_k\| \cdot \|\mathbf{y} - \mathbf{x}_k\| \right] \cdot \Delta_k, && \text{(A.30)}\\
&\leq \frac{1}{2} L_{\nabla f} \Delta_k^2 + \kappa_{eg}^r \left(2r_{max} + \kappa_{eg}^r \Delta_{max} \right) \Delta_k^2 \\
&\qquad + \left[\kappa_{eg} \Delta_k + (\kappa_{eg}^r \Delta_{max} + J_{max})^2 \Delta_k \right] \Delta_k, && \text{(A.31)}\\
&\leq \kappa_{ef} \Delta_k^2, && \text{(A.32)}
\end{aligned}$$

where $\kappa_{ef} := \kappa_{eg} + L_{\nabla f}/2 + \kappa_{eg}^r \left(2r_{max} + \kappa_{eg}^r \Delta_{max} \right) + (\kappa_{eg}^r \Delta_{max} + J_{max})^2$. Lastly, we have $C := \|\hat{W}_k^\dagger\| \leq \sqrt{p+1}\,\Lambda = \mathcal{O}(\Lambda)$ from [10, Theorem 2.9]. □

For our convergence theory to hold, we need to be more specific about the geometry of Y_k being 'good' in Algorithm 1; for the purposes of convergence we take 'good' to mean 'Y_k is Λ-poised'. Note that in the $p = n$ case of exact interpolation, there are algorithms for changing Y_k to make it Λ-poised. For the regression case of $p > n$, it suffices to make a subset of $n + 1$ points in Y_k Λ-poised — see [11, Chapter 6] for a discussion of these issues. The case of reduced initialization ($p < n$) is addressed at the end of next section.

A.2 Global Convergence and Complexity

To ensure global convergence of DFO-LS, we need to add one more phase in Algorithm 1. This phase, known as the 'criticality phase', is called when the interpolated model constructed in line 6 has $\|\mathbf{g}_k\| \leq \epsilon_C$. In this situation, our model gradient is small, so we impose two requirements: shrink Δ_k to be of the same magnitude as $\|\mathbf{g}_k\|$ (specifically, we achieve $\Delta_k \leq \mu\|\mathbf{g}_k\|$), and ensure m_k is fully linear. Details of this phase can be found in [7, Appendix B]. A version of DFO-LS including the criticality phase is given in Algorithm 2. We consider this version of DFO-LS only for case of noiseless objectives (so `NOISY=FALSE`), and where we do not use the reduced initialization cost (i.e. $p_{init} = p$).

To guarantee convergence of our algorithm, we want our approximate solution to the trust region subproblem (2.10) to provide a reasonable decrease in m_k, and so we require the following minimal assumption.

Assumption A.5. The calculated step $\mathbf{s}_k$ (2.10) in line 7 of Algorithm 1 satisfies the 'Cauchy decrease' condition

$$m_k(\mathbf{0}) - m_k(\mathbf{s}_k) \geq c_1 \|\mathbf{g}_k\| \min\left(\Delta_k, \frac{\|\mathbf{g}_k\|}{\max(\|H_k\|, 1)} \right), \qquad \text{(A.33)}$$

for some $c_1 \in [1/2, 1]$ independent of k.

Algorithm 2 DFO-LS with criticality phase.

Input: Starting point $\mathbf{x}_0 \in \mathbb{R}^n$, initial trust region radius $\Delta_0^{init} > 0$ and interpolation set size $p \geq n$.

Parameters from Algorithm 1 are the same, except $p_{init} = p$ and `NOISY=FALSE`. Additional parameters are criticality threshold $\epsilon_C > 0$, criticality scaling $\mu > 0$ and poisedness constant $\Lambda \geq 1$. We also require $\gamma_S < 2c_1/(1+\sqrt{1+2c_1})$, with c_1 from Assumption A.5.

1: Build an initial interpolation set $Y_0 \subset B(\mathbf{x}_0, \Delta_0^{init})$ of size $p_{init}+1 = p+1$, with $\mathbf{x}_0 \in Y_0$. Set $\rho_0^{init} = \Delta_0^{init}$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Given $\mathbf{x}_k$ and Y_k, solve the interpolation problem (2.4) and form m_k^{init} (2.5).
4: **if** $\|\mathbf{g}_k^{init}\| \leq \epsilon_C$ **then**
5: Criticality Phase: using [7, Algorithm 2], modify Y_k and find $\Delta_k \leq \Delta_k^{init}$ such that Y_k is Λ-poised in $B(\mathbf{x}_k, \Delta_k)$ and $\Delta_k \leq \mu\|\mathbf{g}_k\|$, where $\mathbf{g}_k$ is the gradient of the new m_k. Set $\rho_k = \min(\rho_k^{init}, \Delta_k)$.
6: **else**
7: Set $m_k = m_k^{init}$, $\Delta_k = \Delta_k^{init}$ and $\rho_k = \rho_k^{init}$.
8: **end if**
9: Follow lines 7 to 33 of Algorithm 1 to determine $\mathbf{x}_{k+1}$, Δ_{k+1}^{init} and ρ_{k+1}^{init}, updating Y_k as needed. All references to 'improving the geometry of Y_k' must be changed to 'make Y_k Λ-poised in $B(\mathbf{x}_{k+1}, \Delta_{k+1}^{init})$'. Similarly, checking 'geometry of Y_k is good' must be changed to 'Y_k is Λ-poised in $B(\mathbf{x}_k, \Delta_k)$'. We do not terminate if $\rho_k \leq \rho_{end}$, or if objective decrease is too slow.
10: **end for**

This assumption is easy to achieve, for instance by one iteration of steepest descent with exact linesearch (achieving $c_1 = 1/2$). Lastly, we require one more assumption, which is very common for trust region methods.

Assumption A.6. We assume that $\|H_k\| \leq \kappa_H$ for all k, for some $\kappa_H \geq 1$.

We can now state the convergence result for DFO-LS; aside from the details in Section A.1 the details of the proof are identical to [7].

Theorem A.7. *Suppose Assumptions A.5, A.3 and A.6 hold. Then Algorithm 2 produces iterates $\mathbf{x}_k$ such that $\lim_{k\to\infty} \Delta_k = 0$ and $\lim_{k\to\infty} \|\nabla f(\mathbf{x}_k)\| = 0$.*

Again following the details from [7], we can also bound the number of iterations and objective evaluations required to achieve $\|\nabla f(\mathbf{x}_k)\| \leq \epsilon$.

Theorem A.8. *Suppose Assumptions A.5, A.3 and A.6 hold, and that the criticality threshold $\epsilon_C \geq c_3\epsilon$ for some constant $c_3 > 0$. Then the number of iterations i_ϵ (i.e. the number of times a model m_k (2.5) is built) needed by Algorithm 2 until $\|\nabla f(\mathbf{x}_{i_\epsilon+1})\| < \epsilon$ is at most*

$$\left\lfloor \frac{4f(\mathbf{x}_0)}{\eta_1 c_1} \left(1 + \frac{\log \overline{\gamma}_{inc}}{|\log \alpha_3|}\right) \max\left(\kappa_H c_4^{-2}\epsilon^{-2}, c_4^{-1}c_5^{-1}\epsilon^{-2}, c_4^{-1}\Delta_0^{-1}\epsilon^{-1}\right) + \frac{4}{|\log \alpha_3|} \max\left(0, \log\left(\Delta_0 c_5^{-1}\epsilon^{-1}\right)\right) \right\rfloor \tag{A.34}$$

where $c_4 := \min\left(c_3, (1+\kappa_{eg}\mu)^{-1}\right)$, where μ is the criticality phase threshold on $\|\mathbf{g}_k\|$, and

$$c_5 := \min\left(\frac{\omega_C}{\kappa_{eg}+1/\mu}, \frac{\alpha_1 c_4}{\kappa_H}, \alpha_1\left(\kappa_{eg} + \frac{2\kappa_{ef}}{c_1(1-\eta_2)}\right)^{-1}\right). \tag{A.35}$$

For succinctness, we can look at the complexity bounds to leading order in ϵ.

Corollary A.9. *Suppose the assumptions of Theorem A.8 hold. Then for $\epsilon \in (0, 1]$, the number of iterations i_ϵ needed by Algorithm 2 until $\|\nabla f(\mathbf{x}_{i_\epsilon+1})\| < \epsilon$ is at most $\mathcal{O}(\kappa_H \kappa_d^2 \epsilon^{-2})$, and the number of objective evaluations until i_ϵ is at most $\mathcal{O}(\kappa_H \kappa_d^2 p \epsilon^{-2})$, where $\kappa_d := \max(\kappa_{ef}, \kappa_{eg}) = \mathcal{O}(pL_J^2)$.*

If the reduced initialization phase with $p < n$ is appended at the start of Algorithm 2, Theorem A.7 continues to hold, and the complexity bounds in Theorem A.8 and Corollary A.9 for the resulting algorithm increase by n iterations and function evaluations. This is due to the growing set of directions until full-dimensionality that is being generated in the early phase, when no points get removed; the geometry of this set is automatically adjusted by the algorithm, if needed.

B General Objective Test Problems

#	Problem	n	$f(\mathbf{x}_0)$	$f(\mathbf{x}^*)$	Parameters
1	ARWHEAD	100	297	0	$N = 100$
2	BDEXP*	100	26.52572	0	$N = 100$
3	BOX	100	0	-11.24044	$N = 100$
4	BOXPOWER	100	866.2462	0	$N = 100$
5	BROYDN7D	100	350.9842	40.12284	$N/2 = 50$
6	CHARDIS1	98	830.9353	0	$NP1 = 50$
7	COSINE	100	86.88067	-99	$N = 100$
8	CURLY10	100	-6.237221×10^{-3}	-1.003163×10^{4}	$N = 100$
9	CURLY20	100	-1.296535×10^{-2}	-1.003163×10^{4}	$N = 100$
10	DIXMAANA	90	856	1	$M = 30$
11	DIXMAANF	90	1.225292×10^{3}	1	$M = 30$
12	DIXMAANP	90	2.128648×10^{3}	1	$M = 30$
13	ENGVAL1	100	5841	109.0881	$N = 100$
14	FMINSRF2	64	23.461408	1	$P = 8$
15	FMINSURF	64	32.84031	1	$P = 8$
16	NCB20	110	202.002	179.7358	$N = 100$
17	NCB20B	100	200	196.6801	$N = 100$
18	NONCVXU2	100	2.639748×10^{6}	231.8274	$N = 100$
19	NONCVXUN	100	2.727010×10^{6}	231.6808	$N = 100$
20	NONDQUAR	100	106	0	$N = 100$
21	ODC	100	0	-1.098018×10^{-2}	$(NX, NY) = (10, 10)$
22	PENALTY3	100	9.801798×10^{7}	0.001	$N/2 = 50$
23	POWER	100	2.550250×10^{7}	0	$N = 100$
24	RAYBENDL	62	98.03445	96.25168	$NKNOTS = 32$
25	SCHMVETT	100	-280.2864	-294	$N = 100$
26	SINEALI*	100	-0.8414710	-9.900962×10^{3}	$N = 100$
27	SINQUAD	100	0.6561	-4.005585×10^{3}	$N = 100$
28	TOINTGOR	50	5.073786×10^{3}	1.373905×10^{3}	—
29	TOINTGSS	100	892	10.10204	$N = 100$
30	TOINTPSP	50	1.827709×10^{3}	225.5604	—

*Table 1. Details of medium-scale general objective test problems from the CUTEst test set, including the value of $f(\mathbf{x}^*)$ used in (4.2) for each problem. Some problems are variable-dimensional; the relevant parameters yielding the given n are provided. Problems marked * have bound constraints. The value of n shown excludes fixed variables. Some of the problems were taken from [24].*