Derivative-Free Optimisation for Data Fitting

InFoMM
Industrially Focused
Mathematical Modelling

Lindon Roberts
University of Oxford
Supervisors: Coralia Cartis, Jan Fiala, Benjamin Marteau

A technical report for
InFoMM CDT Mini-Project 1
in partnership with
Numerical Algorithms Group (NAG)
Trinity 2016

Contents

1 Introduction

Fitting mathematical models to observational data is a crucial part of solving quantitative problems in many industries and application areas. One of the most common techniques for data fitting is least-squares minimisation. Thus data fitting gives rise to a class of optimisation problems with specific mathematical structure. We can use this structure to develop specialised optimisation algorithms which are more efficient than algorithms for more general problems.

A common feature of many standard optimisation algorithms is the necessity of having gradients (or higher derivatives) of the objective and constraints. In iterative methods, derivatives tell us about the local structure of the problem near our current iterate, which can inform how we choose the next iterate. There are several ways that optimisation software may gather derivative information, such as

- Requiring an input function which calculates derivatives, which the user must code themselves;
- Estimating derivatives with finite differences; and
- Algorithmic differentiation, where the chain rule applied to the underlying mathematical operations within source code builds up derivative information automatically.

However, there are situations where derivative information is not available. Common examples are when third-party proprietary code is used (so the exact calculations and source code are unavailable), and where the objective calculation is very expensive to calculate, or subject to noise (making finite difference approximations either intractable or inaccurate).

Derivative-free optimisation (DFO) algorithms are designed to handle such situations. These algorithms avoid any calculation of derivatives, explicitly or implicitly, and as such have some fundamental differences to standard optimisation algorithms. DFO algorithms are suited to real-world problems in algorithm parameter optimisation, energy, weather forecasting, and engineering design, among others. Some applications of DFO can be found in [3, Chapter 1].

In this report, we will look at derivative-free methods to solve least-squares optimisation problems. The Numerical Algorithms Group (NAG) is interested in incorporating such techniques into their software library, which will be useful for their customers with data fitting problems, particularly where model evaluation is expensive or noisy. Such problems arise, for instance, in model calibration for finance, energy and climate.

The key outcomes of this project are:

1. The implementation in the NAG software library of a derivative-free algorithm for least-squares problems (*update Nov 2018*: see Section 8.1 for further information);
2. A comparison of existing derivative-free algorithms and their performance on least-squares problems;

3. An analysis of areas where state-of-the-art DFO algorithms could be improved.

The outline of this report is as follows: in Section 2, we introduce derivative-free optimisation for general problems, with a focus on model-based methods. To do this, we will first consider the case where we have derivative information, which will highlight exactly how the lack of derivatives force us to change approach. Then in Section 3, I will introduce least-squares problems, their mathematical structure, and common derivative-based solution methods. In Section 4, we will see how general-purpose DFO techniques combine with the least-squares problem structure to produce a DFO algorithm specifically designed for least-squares problems. Having introduced a number of different algorithms, in Sections 5 and 6, we will examine a framework for performing numerical experiments on DFO algorithms for least-squares problems, and then see how the different algorithms perform. Lastly, as this work will continue into a 3-year DPhil project, in Section 7 we will consider different ways the current algorithms could be extended, paying particular attention to large scale and noisy problems.

2 Methods for General Derivative-Free Optimisation

In this section, we consider how to construct derivative-free optimisation algorithms. Much of the material in this section is drawn from [14, Chapter 9] and [3]. As we will see in Section 2.1, many common optimisation techniques rely on a Taylor series-type approximation of the objective at each step. However, when derivatives are not available, we cannot construct a Taylor expansion; we must use other techniques. The methods we will consider are model-based variants of trust region algorithms. However, for completeness, we will briefly consider some common alternative classes of DFO algorithms. All methods in this section apply to general problems, not just least-squares minimisation. We will consider the general unconstrained minimisation problem

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), \tag{2.1}$$

where f has sufficient smoothness for the relevant context.

2.1 Classical Model-Based Methods

To motivate the model-based methods for derivative-free optimisation in Section 2.2, we will first discuss general model-based optimisation algorithms when derivatives are available.

The basis for many methods for solving (2.1) is a Taylor series approximation of the objective f around our current iterate $\mathbf{x}_k$:

$$f(\mathbf{x}_k + \mathbf{s}) \approx m_k(\mathbf{s}) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T\mathbf{s} + \frac{1}{2}\mathbf{s}^T\nabla^2 f(\mathbf{x}_k)\mathbf{s}. \tag{2.2}$$

Here we think of the second-order Taylor series as a model for the value of the objective in a neighbourhood of $\mathbf{x}_k$.

In Newton's method, we try to find $\mathbf{s}$ minimising $m_k(\mathbf{s})$ by setting $\nabla m_k = 0$. This gives us the update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$, where $\mathbf{s}_k$ solves the linear system

$$\nabla^2 f(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k), \tag{2.3}$$

and $\alpha_k > 0$ is some step size. The value of α_k is chosen to satisfy certain conditions which allow the algorithm to converge. A common choice, used in the NAG Library implementation [8, 11], for instance, are the *Wolfe conditions*:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) + \gamma \alpha_k \nabla f(\mathbf{x}_k)^T \mathbf{s}_k \tag{2.4}$$

$$|\nabla f(\mathbf{x}_{k+1})^T \mathbf{s}_k| \leq \eta |\nabla f(\mathbf{x}_k)^T \mathbf{s}_k|, \tag{2.5}$$

where $0 < \gamma < \eta < 1$ are parameters. If we are close to a local minimum, then $\nabla^2 f(\mathbf{x}_k)$ will be positive definite and $\alpha_k = 1$ satisfies (2.4) and (2.5). Thus we are truly minimising m_k, and Newton's method converges quadratically to that local minimum [14, Theorem 3.5]. However, in general, we cannot guarantee that $\nabla^2 f(\mathbf{x}_k)$ is positive definite, or even invertible. That is, $\mathbf{s}$ satisfying (2.3) may not exist, or not be a descent direction of f.

To address this, we impose a constraint on our minimisation of m_k, based on the understanding that (2.2) is only accurate for small $\mathbf{s}$. Our update becomes $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$, where

$$\mathbf{s}_k = \underset{\mathbf{s} \in \mathbb{R}^n}{\arg\min}\, m_k(\mathbf{s}) \qquad \text{subject to} \qquad \|\mathbf{s}\| \leq \Delta_k, \tag{2.6}$$

where $\|\cdot\|$ is the Euclidean norm (although other norms are sometimes used). The feasible set $\|\mathbf{s}\| \leq \Delta_k$ is known as the *trust region*, with radius Δ_k, and corresponds to the region where we trust the model (2.2) to be a good approximation to $f(\mathbf{x}_k + \mathbf{s})$.

If we solve the trust region subproblem (2.6) sufficiently well – and there are specialised algorithms that achieve this – and make sure to update Δ_k at each iteration based on how well the approximation (2.2) holds for the chosen step $\mathbf{s}_k$, we can get a globally convergent algorithm. The only requirements on f are smoothness conditions and that $\nabla^2 f(\mathbf{x}_k)$ are uniformly bounded for all k [14, Theorem 4.5].

2.2 Derivative-Free Model-Based Methods

As mentioned above, the main derivative-free methods we will consider in this report are model-based. In essence, these methods maintain a set of points $Y_k \subset \mathbb{R}^n$, typically of size $\mathcal{O}(n)$ or $\mathcal{O}(n^2)$, at which f has been evaluated. We then use these values to construct an interpolating polynomial m_k, designed to approximate f in a neighbourhood of the current iterate $\mathbf{x}_k \in Y_k$. In the trust region method described in Section 2.1, this replaces the Taylor approximation (2.2) – the other features of the algorithm remain the same.

Previously, we built the model (2.2), which required first and second derivatives of f at $\mathbf{x}_k$. In the derivative-free context, we instead construct the quadratic model

$$f(\mathbf{x}_k + \mathbf{s}) \approx m_k(\mathbf{s}) = f(\mathbf{x}_k) + \mathbf{g}_k^T \mathbf{s} + \frac{1}{2}\mathbf{s}^T H_k \mathbf{s}, \tag{2.7}$$

and find $\mathbf{g}_k$ and H_k from the interpolation conditions

$$m_k(\mathbf{y} - \mathbf{x}_k) = f(\mathbf{y}), \qquad \forall \mathbf{y} \in Y_k. \tag{2.8}$$

Note that (2.8) is automatically satisfied for $\mathbf{y} = \mathbf{x}_k$ because we have already set the constant term in (2.7) to be $f(\mathbf{x}_k)$. We can use this derivative-free model m_k to solve the trust region subproblem (2.6). This is the basic framework of the model-based algorithms we will consider. Before we do, there are some extra complications from (2.7) we must address.

Firstly, we need to choose an interpolation set Y_k such that (2.8) is able to be satisfied. This is possible as long as the set of points is not degenerate; i.e. does not lie in a quadratic manifold. Also, if Y_k has fewer than $(n+1)(n+2)/2$ points, then the model (2.7) is underdetermined. To get a unique solution, we need to impose extra conditions on the interpolation. A common approach is to use minimum Frobenius norm updating; that is, solve the quadratic program

$$\begin{aligned} \min_{\mathbf{g}_k, H_k} \quad & \|H_k - H_{k-1}\|_F^2, && \text{(2.9a)} \\ \text{subject to} \quad & H_k \text{ symmetric}, && \text{(2.9b)} \\ & m_k(\mathbf{y} - \mathbf{x}_k) = f(\mathbf{y}), \qquad \forall \mathbf{y} \in Y_k. && \text{(2.9c)} \end{aligned}$$

As detailed in [18, 19, 20], this problem can be efficiently solved via a single linear system of size $|Y_k| + n + 1$. In the first instance, we usually take $H_{-1} = 0$.

Although it is not difficult to find a set Y_k for which (2.9c) can be satisfied, we want our interpolating model to be stable to small perturbations in the sample set (i.e. the interpolation problem is well-conditioned), and to provide a good local approximation to the objective. To achieve these goals, we need to control the geometry of Y_k in a more specific way.

Following [3, Chapter 5], we call a Y_k 'poised' if there exists an interpolating model satisfying (2.8). A suitable measure of 'poisedness' is given by the following.

Definition 2.1. Let $\Lambda > 0$, $B \subset \mathbb{R}^n$, and $\mathcal{P}_n^2$ be the space of quadratic polynomials on $\mathbb{R}^n$, with dimension $q = (n+1)(n+2)/2$. A poised set $Y_k = \{\mathbf{y}_1, \ldots, \mathbf{y}_p\} \subset \mathbb{R}^n$, where $n + 1 \leq p \leq q$ is Λ-poised in B (in the minimum Frobenius norm sense) if

$$\max_{1 \leq i \leq p} \max_{\mathbf{x} \in B} |\ell_i(\mathbf{x})| \leq \Lambda, \tag{2.10}$$

where $\{\ell_1, \ldots, \ell_p\}$ is the minimum Frobenius norm Lagrange polynomial basis for Y_k in $\mathcal{P}_n^2$.

The minimum Frobenius norm Lagrange polynomial basis is defined by $\ell_i(\mathbf{y}_j) = \delta_{i,j}$ for all $i, j = 1, \ldots, p$, and minimising $\|\nabla^2 \ell_i\|_F$ over all such quadratics. Generally

speaking, small Λ corresponds to good geometry in the set Y_k. If some point $\mathbf{y} \in Y_k$ is also in B, then $\Lambda \geq 1$. In our algorithms, we want our set Y_k to be Λ-poised in a neighbourhood of $\mathbf{x}_k$. This gives us error bounds on the approximation $m_k(\mathbf{s}) \approx f(\mathbf{x}_k + \mathbf{s})$. Specifically, we have the following result [22, Lemma 3.1].

Theorem 2.2. *Let $B = \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}_k\| \leq \Delta\} \subset \mathbb{R}^n$ be the ball of radius Δ about $\mathbf{x}_k$. Suppose $Y_k \subset B$ is Λ-poised in the ball B, and $n + 1 \leq |Y_k| \leq (n + 1)(n + 2)/2$. Suppose also that f is continuously differentiable and ∇f is Lipschitz continuous with constant L in a neighbourhood of B. Then for any $\|\mathbf{s}\| \leq \Delta$ we have the bounds*

$$\|\nabla f(\mathbf{x}_k + \mathbf{s}) - \nabla m_k(\mathbf{s})\| \leq c_1(\|\nabla^2 m_k\| + L)\Delta, \qquad \textit{and} \tag{2.11}$$

$$|f(\mathbf{x}_k + \mathbf{s}) - m_k(\mathbf{s})| \leq c_2(\|\nabla^2 m_k\| + L)\Delta^2, \tag{2.12}$$

where $c_1, c_2 > 0$ are constants depending only on Y_k.

2.2.1 Algorithm DFO-basic

The simplest algorithm we consider is called DFO-basic, based on the algorithm described in [2]. A full specification is given in Algorithm 1.

The most important feature of this algorithm is that it begins with an interpolation set of size $2n+1$, then allows the set to change size, up to a maximum of $(n+1)(n+2)/2$ points. However, it does not do any checks about the geometry of the interpolation set. Each iteration is then a standard trust region iteration, where it uses built-in MATLAB routines to recalculate the whole interpolated model and solve the trust region subproblem exactly.

2.2.2 Algorithm BOBYQA

The algorithm BOBYQA [20] is intended to solve the bound-constrained problem

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}), \qquad \text{subject to } \mathbf{x}^- \leq \mathbf{x} \leq \mathbf{x}^+. \tag{2.13}$$

However, in this report we only consider unconstrained problems, so we simplify below for the case where $\mathbf{x}^{\pm} = (\pm\infty, \ldots, \pm\infty)$.

BOBYQA differs from DFO-basic in that it uses a fixed size interpolation set. The interpolation set is allowed to be of size p, where $n+2 \leq p \leq (n+1)(n+2)/2$. When we have a new point, we swap it with another point currently in the interpolation set. It does, however, control the geometry of the interpolation set by ensuring Λ-poisedness near $\mathbf{x}_k$. A more detailed description of BOBYQA is given in Algorithm 2, although for a complete description the reader is referred to [20].

Note that each geometry-improving step maintains $\mathbf{x}_k$, but may replace the other points. It also updates the interpolant to make sure that (2.8) still holds.

It is worth noting here that [7] concludes that, for smooth functions f, a derivative-free method which does not control the geometry still shows 'quite satisfactory' performance. That is, although the geometry-improving steps are important for proving convergence, they are perhaps not essential for practical purposes.

Algorithm 1 DFO-basic

Input: Objective $f : \mathbb{R}^n \to \mathbb{R}$, starting point $\mathbf{x}_0 \in \mathbb{R}^n$ and trust region radius $\Delta > 0$.

Generate an interpolation set Y of size $2n+1$ given by $\mathbf{x}_0$ and $\mathbf{x}_0 \pm \Delta \mathbf{e}_i$ for $i = 1, \ldots, n$.
Evaluate f at each point in Y.
for $k = 0, 1, 2, \ldots$ **do**
    Interpolate a quadratic model (2.7) using the data $\{(\mathbf{y}, f(\mathbf{y}) : \mathbf{y} \in Y\}$, by solving (2.9).
    Solve the trust region subproblem to get a step $\mathbf{s}_k$.
    Evaluate f at the trial point $\mathbf{x}_k + \mathbf{s}_k$.
    If $f(\mathbf{x}_k + \mathbf{s}_k) < f(\mathbf{x}_k)$, then set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ [trial step accepted], otherwise set $\mathbf{x}_{k+1} = \mathbf{x}_k$ [trial step rejected].
    Adjust the trust region radius Δ.
    if $|Y| < (n+1)(n+2)/2$ **then**
        Add $\mathbf{x}_k + \mathbf{s}_k$ to Y.
    else if trial step accepted **then**
        Replace the point in Y furthest from $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ with $\mathbf{x}_{k+1}$.
    else
        Replace the point in Y furthest from $\mathbf{x}_{k+1} = \mathbf{x}_k$ with $\mathbf{x}_k + \mathbf{s}_k$, if it is closer to $\mathbf{x}_{k+1}$.
    end if
    if $\Delta < 10^{-3}$ **and** $\|\mathbf{s}_k\| < 10^{-1}$ **then**
        Remove from Y all points $\mathbf{y}$ such that $\|\mathbf{y} - \mathbf{x}_{k+1}\| \geq 100\Delta$.
    end if
end for

2.3 Coordinate and Pattern Search Methods

In coordinate and pattern search methods, we aim to explore the space near our current iterate $\mathbf{x}_k$ in a systematic way. For coordinate search, we always take a step along a particular coordinate direction: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{e}_k$ for some canonical basis vector $\mathbf{e}_k$. We choose α_k via a line search. Not all choices of $\mathbf{e}_k$ allow convergence. One method which does converge comes from the sequence:

$$(\mathbf{e}_k)_{k=1}^{\infty} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{n-1}, \mathbf{e}_n, \mathbf{e}_{n-1}, \ldots, \mathbf{e}_2, \mathbf{e}_1, \mathbf{e}_2, \ldots). \tag{2.14}$$

Pattern search is a more general variant of coordinate search. Instead of fixing a single direction in every iteration and performing a line search to find α_k, we fix α_k and choose our step direction from a set of possible directions. Formally, suppose we have a set of possible directions $\mathcal{S}_k \subset \mathbb{R}^n$. Our tentative step direction $\mathbf{s}_k \in \mathcal{S}_k$ is chosen to make $f(\mathbf{x}_k + \alpha_k \mathbf{s}_k)$ small (although not necessarily minimal) in $\mathcal{S}_k$. We take $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$ as long as it ensures sufficient objective reduction, otherwise we reduce α_k and start again. Using

$$\mathcal{S}_k = \{\mathbf{e}_1, \ldots, \mathbf{e}_n, -\mathbf{e}_1, \ldots, -\mathbf{e}_n\}, \tag{2.15}$$

is one choice which allows a pattern search algorithm to converge.

Algorithm 2 BOBYQA

Input: Objective $f : \mathbb{R}^n \to \mathbb{R}$, starting point $\mathbf{x}_0 \in \mathbb{R}^n$, trust region radius $\Delta > 0$ and interpolation set size p satisfying $n + 2 \leq p \leq (n + 1)(n + 2)/2$.

Generate an interpolation set Y of size p in a similar way to DFO-basic.
Evaluate f at each point in Y and build an interpolant by solving (2.9) with $H_{-1} = 0$.
for $k = 0, 1, 2, \ldots$ **do**
    Solve the trust region subproblem to get a step $\mathbf{s}_k$.
    if $\|\mathbf{s}_k\| < \Delta/2$ **then**
        Adjust Δ and improve the poisedness of Y.
        Restart the loop with $\mathbf{x}_{k+1}$ being the point in Y with minimum objective value.
    end if
    Choose which point $\mathbf{y} \in Y$ to replace with the new point, to get optimal geometry.
    If rounding errors have accumulated sufficiently, improve the poisedness of Y and restart the loop.
    Evaluate f at new point $\mathbf{x}_k + \mathbf{s}_k$.
    Adjust the trust region radius Δ.
    if $f(\mathbf{x}_k + \mathbf{s}_k) < f(\mathbf{x}_k)$ **then**
        Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$.
    else
        Set $\mathbf{x}_{k+1} = \mathbf{x}_k$.
    end if
    Replace $\mathbf{y}$ with $\mathbf{x}_k + \mathbf{s}_k$ and update the interpolant by solving (2.9).
    If the actual reduction was not large, adjust Δ and improve the poisedness of Y.
end for

2.4 Simplex Methods

In simplex methods, we maintain a set of $n+1$ points in $\mathbb{R}^n$ which form a simplex. The most popular variant is the Nelder-Mead method [13], although there are problems for which it is known not to converge. In Nelder-Mead, at every iteration we modify the simplex via one of a set of geometric transforms, typically replacing the point with the largest objective value with another on the line between that point and the centroid of the simplex. If none of these options provide an objective decrease, we can alternatively perform a 'shrinkage' step, where we reduce the size of the simplex, moving all the vertices towards the one with smallest objective value.

3 Least-Squares Minimisation

In this section, we introduce least-squares problems and some classical (derivative-based) solution methods. The content in this section is primarily drawn from [14, Chapter 10].

Least-squares problems have the form:

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) := \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|^2 = \frac{1}{2}\sum_{i=1}^{m} r_i(\mathbf{x})^2, \tag{3.1}$$

where $r_i : \mathbb{R}^n \to \mathbb{R}$ for $1 \leq i \leq m$, the vector-valued function $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^m$ is defined by $\mathbf{r}(\mathbf{x}) := (r_1(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T$, and $\|\cdot\|$ is the Euclidean norm. There are circumstances where we may want to add constraints on the input $\mathbf{x}$, but for now we consider the unconstrained version.

Typically for data fitting problems we have more observations than values we wish to calibrate; i.e. $m \geq n$. A common situation is when the residuals r_i are the difference between the model output for some input data and an empirical observation, so $r_i(\mathbf{x}) = \text{model}(\mathbf{d}_i; \mathbf{x}) - y_i$ for observations $(\mathbf{d}_i, y_i)$.

If we define $J(\mathbf{x})$ to be the $m \times n$ Jacobian matrix with i-th row $\nabla r_i(\mathbf{x})^T$, we have

$$\nabla f(\mathbf{x}) = J(\mathbf{x})^T \mathbf{r}(\mathbf{x}), \qquad \text{and} \tag{3.2}$$

$$\nabla^2 f(\mathbf{x}) = J(\mathbf{x})^T J(\mathbf{x}) + \sum_{i=1}^{m} r_i(\mathbf{x}) \nabla^2 r_i(\mathbf{x}). \tag{3.3}$$

An important feature of least-squares problems is that we can approximate

$$\nabla^2 f(\mathbf{x}) \approx J(\mathbf{x})^T J(\mathbf{x}), \tag{3.4}$$

and so can estimate second derivatives using only first derivative information. This approximation works well for problems which have small or approximately linear residuals near the solution.

3.1 Gauss-Newton Method

The Gauss-Newton method is a simple algorithm for solving nonlinear least-squares problems, which is very similar to Newton's method. To get this method, we modify

(2.3) with the approximation (3.4), and instead solve the linear system

$$[J(\mathbf{x}_k)^T J(\mathbf{x}_k)]\mathbf{s}_k = -\nabla f(\mathbf{x}_k). \tag{3.5}$$

Again, our iteration is $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$ for some $\alpha_k > 0$ satisfying (2.4) and (2.5). The Gauss-Newton method converges to a stationary point if the Jacobians $J(\mathbf{x})$ have singular values uniformly bounded away from zero [14, Theorem 10.1]. If (3.4) holds closely, such as when each r_i is linear or close to zero, a Gauss-Newton step is exactly a Newton step, so we get quadratic convergence. By contrast, when the approximation error is large, we expect convergence to be approximately linear.

An important observation here is that Newton's method corresponds to approximating the full objective f with a second-order Taylor expansion around $\mathbf{x}_k$ (2.2). The Gauss-Newton method can be derived by linearising each residual r_i separately,

$$r_i(\mathbf{x}_k + \mathbf{s}) \approx r_i(\mathbf{x}_k) + \nabla r_i(\mathbf{x}_k)^T \mathbf{s} \qquad i = 1, \ldots, m, \tag{3.6}$$

which corresponds to the approximate second-order expansion

$$f(\mathbf{x}_k + \mathbf{s}) \approx \frac{1}{2}\|\mathbf{r}(\mathbf{x}_k) + J(\mathbf{x}_k)\mathbf{s}\|^2. \tag{3.7}$$

If each r_i is linear, we can write $\mathbf{r}(\mathbf{x}) = J\mathbf{x} - \mathbf{y}$, and (3.5) corresponds to the normal equations for linear least squares problems with $\mathbf{x}_k = \mathbf{0}$.

A version of the Gauss-Newton method, which uses finite differences to estimate gradients, is implemented in the NAG library routine E04FC. This routine is based on [8], and uses $\gamma = 10^{-4}$ in (2.4). The other parameter η in (2.5) is an input to the routine, with default value $\eta = 0.5$.

3.2 Levenberg-Marquardt Method

In cases where the Jacobian is not full rank, the approximation (3.4) does not produce a positive definite matrix, and Gauss-Newton may fail to produce a descent direction. As in (2.6), we can move (3.7) into a trust region framework, and solve at each iteration

$$\mathbf{s}_k = \underset{\mathbf{s}\in\mathbb{R}^n}{\arg\min}\ \frac{1}{2}\|\mathbf{r}(\mathbf{x}_k) + J(\mathbf{x}_k)\mathbf{s}\|^2, \qquad \text{subject to } \|\mathbf{s}\| \le \Delta_k. \tag{3.8}$$

We then either take the step $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$, or adjust the trust region radius Δ_k. The KKT conditions for this problem say that either the solution corresponds to the full Gauss-Newton step (3.5), or the solution to the regularised problem

$$[J(\mathbf{x}_k)^T J(\mathbf{x}_k) + \lambda I]\mathbf{s}_k = -\nabla f(\mathbf{x}_k), \tag{3.9}$$

for some $\lambda > 0$. The resulting trust region algorithm is called the Levenberg-Marquardt method, and converges at a similar rate to the Gauss-Newton method.

4 Derivative-Free Methods for Least-Squares Problems

In this section, we combine the ideas from Sections 2 and 3 to form a derivative-free algorithm specifically tailored to least-squares problems. Here we follow the method from [22], which is an adaptation of BOBYQA. The resulting algorithm, called DFBOLS, also handles bound constraints, but again we ignore these for this report. That is, we are solving (3.1).

The key idea of DFBOLS is to replace a single model for the full objective f with different models for each residual r_i, in the style of (3.6). This involves larger memory requirements (storing m different models) and processing requirements (the interpolation problem (2.9) must be solved m times whenever Y is updated). However, the benefit is we can create a model for the full objective f which exploits its least-squares structure.

Formally, for each $i = 1, \dots, m$, we have a model

$$r_i(\mathbf{x}_k + \mathbf{s}) \approx m_k^{(i)}(\mathbf{s}) = r_i(\mathbf{x}_k) + (\mathbf{g}_k^{(i)})^T \mathbf{s} + \frac{1}{2}\mathbf{s}^T H_k^{(i)} \mathbf{s}, \tag{4.1}$$

satisfying the interpolation conditions

$$m_k^{(i)}(\mathbf{y} - \mathbf{x}_k) = r_i(\mathbf{y}), \qquad \forall \mathbf{y} \in Y_k. \tag{4.2}$$

We write $\mathbf{m}_k = (m_k^{(1)}, \dots, m_k^{(m)})^T$ as the vector of models, and we have the Jacobian $J_k = J(\mathbf{x}_k) = (\mathbf{g}_k^{(1)}, \dots, \mathbf{g}_k^{(m)})^T$. Approximating the full objective f by taking a second-order expansion of $\frac{1}{2}\|\mathbf{m}_k(\mathbf{s})\|^2$, we get

$$f(\mathbf{x}_k + \mathbf{s}) \approx m_k^f(\mathbf{s}) = \frac{1}{2}\|\mathbf{r}(\mathbf{x}_k)\|^2 + (\mathbf{g}_k^f)^T \mathbf{s} + \frac{1}{2}\mathbf{s}^T H_k^f \mathbf{s}, \tag{4.3}$$

where

$$\mathbf{g}_k^f = J_k^T \mathbf{r}(\mathbf{x}_k), \tag{4.4}$$

and

$$H_k^f = J_k^T J_k + \begin{cases} 0 & \text{if } \|\mathbf{g}_k^f\| \geq \kappa_1, \\ \kappa_3 \|\mathbf{r}(\mathbf{x}_k)\| I & \text{if } \|\mathbf{g}_k^f\| < \kappa_1 \text{ and } \frac{1}{2}\|\mathbf{r}(\mathbf{x}_k)\|^2 < \kappa_2 \|\mathbf{g}_k^f\|, \\ \sum_{i=1}^m r_i(\mathbf{x}_k) H_k^{(i)} & \text{otherwise.} \end{cases} \tag{4.5}$$

In [22], the suggested values of the constants κ_1, κ_2 and κ_3 are 1, 1 and 0.01 respectively. The three possibilities for H_k^f correspond to:

- Gauss-Newton when we are far from a stationary point;
- Levenberg-Marquardt when we are close to a stationary point with small residual; and
- Full Newton when we are close to a stationary point with large residual.

When we are close to a stationary point, the Levenberg-Marquardt step regularises the Hessian to try and maintain positive definiteness. If we suspect our problem has nonzero residual at its minimum, however, we typically lose superlinear convergence of these methods, so we move to a full Newton Hessian.

A description of DFBOLS is given in Algorithm 3. Aside from the formation of the full model (4.3), it is the same as BOBYQA.

Algorithm 3 DFBOLS

Input: Residuals $r_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, m$, starting point $\mathbf{x}_0 \in \mathbb{R}^n$, trust region radius $\Delta > 0$ and interpolation set size p satisfying $n + 2 \leq p \leq (n + 1)(n + 2)/2$.

Generate an interpolation set Y of size p in a similar way to DFO-basic.
Evaluate each r_i at each point in Y and build m interpolating models (4.1) by solving (2.9) with $H_{-1}^{(i)} = 0$ for all i.
for $k = 0, 1, 2, \ldots$ **do**
    Form the model (4.3) for the full objective f.
    Solve the trust region subproblem with objective (4.3) to get a step $\mathbf{s}_k$.
    if $\|\mathbf{s}_k\| < \Delta/2$ **then**
        Adjust Δ and improve the poisedness of Y.
        Restart the loop with $\mathbf{x}_{k+1}$ being the point in Y with minimum objective value.
    end if
    Choose which point $\mathbf{y} \in Y$ to replace with the new point, to get optimal geometry.
    If rounding errors have accumulated sufficiently, improve the poisedness of Y and restart the loop.
    Evaluate f at new point $\mathbf{x}_k + \mathbf{s}_k$.
    Adjust the trust region radius Δ.
    if $f(\mathbf{x}_k + \mathbf{s}_k) < f(\mathbf{x}_k)$ **then**
        Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$.
    else
        Set $\mathbf{x}_{k+1} = \mathbf{x}_k$.
    end if
    Replace $\mathbf{y}$ with $\mathbf{x}_k + \mathbf{s}_k$ and update the m interpolating models by solving (2.9).
    If the actual reduction was not large, adjust Δ and improve the poisedness of Y.
end for

This completes the description of the different derivative-free algorithms. We now consider how they perform in practice.

5 Description of Testing Methodology

In this section, we consider the various aspects required to set up the numerical experiments to compare different DFO algorithms. This includes the set of problems and solvers tested, plus a description of how to compare derivative-free optimisation algorithms. As in [22], the testing methodology from [10] is used.

5.1 Solvers Compared

We will compare the performance of a number of different algorithms:

- Gauss-Newton (see Section 3.1) using finite differences, from the NAG library routine E04FC. The line search parameter η was varied over $\eta \in \{0.01, 0.1, 0.5, 0.9\}$, but the default is $\eta = 0.5$;
- MATLAB's built-in function `fminsearch`, an implementation of Nelder-Mead (see Section 2.4);
- DFO-basic (see Section 2.2.1);
- BOBYQA (see Section 2.2.2) with interpolation set sizes $n + 2$, $2n + 1$ and $(n + 1)(n + 2)/2$, from the NAG library routine E04JC;
- DFBOLS (see Section 4) with interpolation set sizes $n + 2$, $2n + 1$ and $(n + 1)(n + 2)/2$;
- MATLAB's built-in function `fminunc`, an implementation of BFGS which uses finite differences;
- BFO, a state-of-the-art pattern search algorithm (see Section 2.3) from [17]; and
- SID-PSM, a state-of-the-art pattern search algorithm from [5, 4].

The NAG library version used was Mark 24.4. DFO-basic, BOBYQA and DFBOLS all require a starting trust region radius. For BOBYQA, NAG recommends using 10% of the largest expected change in any coordinate between the starting value $\mathbf{x}_0$ and the solution [12]. To this end, the value $0.1 \max(\|\mathbf{x}_0\|_\infty, 1)$ was used for all three solvers. Additionally, the Gauss-Newton routine requires an estimate of the distance between $\mathbf{x}_0$ and the solution; for this, $\|\mathbf{x}_0\|_2$ was used. In DFBOLS, the constants κ_1, κ_2 and κ_3 in (4.5) were given the recommended values 1, 1 and 0.01 respectively.

Within DFBOLS, we have three versions:

1. The original code from [22], which only allows up to $2n + 1$ interpolation points;
2. A new implementation designed to integrate with the NAG library, which allows the full $\mathcal{O}(n^2)$ interpolation points;
3. The new implementation with a slightly different trust region radius update in the case of a good actual reduction (taken from BOBYQA, rather than [22]).

In Section 6, these versions are labelled 'DFBOLS', 'E04JC-LS' (NAG's BOBYQA routine is named E04JC; this version is adapted to least-squares) and 'E04JC-LS BBQTR' (for BOBYQA Trust Region) respectively. They are all implementations of the DFBOLS algorithm, as specified in Algorithm 3, and are extensions of the original BOBYQA algorithm. However, the version from [22] has extra small modifications that are not included in the NAG implementations (due to licensing restrictions). The two NAG implementations, E04JC-LS and E04JC-LS BBQTR differ in the trust

region radius update formula. When the ratio of actual reduction to predicted model reduction is at least 0.7 (i.e. a good trust region step), the two updates are

$$\Delta_{k+1} = \max(0.5\Delta_k, 2\|\mathbf{d}_k\|), \tag{5.1}$$

$$\Delta_{k+1} = \max(\Delta_k, 2\|\mathbf{d}_k\|), \tag{5.2}$$

where (5.1) is used in BOBYQA and E04JC-LS BBQTR, and (5.2) is used in E04JC-LS.

The BFO solver has the option to train several of its search parameters. Both untrained parameters and parameters trained on the full test set (with noiseless objective) were considered. In SID-PSM, the default mesh size was set to be the same as the starting trust region radius for the model-based methods, i.e. $0.1\max(\|\mathbf{x}_0\|_\infty, 1)$.

Of the full set of solvers, all except Gauss-Newton and `fminunc` are derivative free, but only DFBOLS and Gauss-Newton are adapted specifically to least-squares problems.

The primary termination condition of DFO-basic, BOBYQA and DFBOLS is when the trust region radius reaches some lower threshold. This indicates that the set of interpolation points is in a very small region of space, which should be close to the stationary point. For our testing, we are interested in the progress that each algorithm can make in a fixed number of function evaluations. Therefore, the lower threshold on the trust region radius was in all cases set to be much smaller than machine precision. This means that if the software did terminate due to reaching the lower trust region radius bound, we should still be very close to the solution, and shouldn't expect the algorithm to show substantial improvement even with more function evaluations. Similarly, for all other solvers, all available termination options were set to force computational budget to be the main constraint on the algorithm's termination.

5.2 Problem Set

The problem set used is a standard one for testing DFO, and comes from [10]. It was also used in [22]. Both papers use a test suite of 53 least-squares problems, most of which are from the test set in [9]. These problems have $2 \leq n \leq 12$ and $2 \leq m \leq 65$, and cover linear (full-rank and rank-deficient) and nonlinear residual functions. The problems also have both zero and nonzero residual solutions, which is important for testing the approximation (3.4).

As mentioned earlier, one important situation where derivative free algorithms are useful is when the objective evaluation is noisy. In this test set, residual evaluation can optionally include either deterministic or stochastic noise of size $\sigma > 0$. The deterministic noise is incorporated by evaluating $\tilde{r}_i(\mathbf{x}) = \sqrt{1 + \sigma\epsilon(\mathbf{x})}\, r_i(\mathbf{x})$, where

$$\epsilon(\mathbf{x}) := \pi(\mathbf{x})(4\pi(\mathbf{x})^2 - 3). \tag{5.3}$$

This is a composition of the high-frequency noise term

$$\pi(\mathbf{x}) := 0.9\sin(100\|\mathbf{x}\|_1)\cos(100\|\mathbf{x}\|_\infty) + 0.1\cos(\|\mathbf{x}\|_2), \tag{5.4}$$

with a third-order Chebyshev polynomial. Stochastic noise is incorporated by evaluating $\tilde{r}_i(\mathbf{x}) = [1 + \sigma\epsilon]r_i(\mathbf{x})$, where ϵ is a uniformly distributed random variable on $(-1, 1)$, independently drawn for each i and $\mathbf{x}$.

5.3 Measuring Convergence

Typically when testing the convergence of optimisation algorithms, we calculate the gradient of the objective at each iteration (and stop when it is small). However, in the derivative free case, this option is not available to us. Therefore we need some other kind of convergence test. Again, we follow the approach from [10],

We have used settings to force the primary termination criteria to be some computational budget, say N_f function evaluations. Suppose when we ran a solver $\mathcal{S}$, we calculated objective values $(f_1^{\mathcal{S}}, \ldots, f_{N_f}^{\mathcal{S}})$. Since our goal is to compare the results of different solvers, define the optimum value for a problem to be the smallest objective value found by any solver:

$$f^* = \min_{\mathcal{S}} \min_{1 \le i \le N_f} f_i^{\mathcal{S}}. \tag{5.5}$$

We say that a problem p is 'solved' by $\mathcal{S}$ within a budget of $N_p(\mathcal{S}; \tau)$ function evaluations if it is smallest value N such that

$$f_N^{\mathcal{S}} \le f^* + \tau(f(\mathbf{x}_0) - f^*), \tag{5.6}$$

holds for a given threshold $\tau > 0$. If a problem is not solved in the sense of (5.6) within the full N_f function evaluations allowed, we set $N_p(\mathcal{S}; \tau) = \infty$.

In this case, the set of solvers is the set of all methods listed in Section 5.1 over all input parameter value (e.g. all interpolation set sizes). Also, to allow for the greater difficulty of problems in higher dimensions, we measure the computational budget in units of gradients; i.e. units of $n + 1$ function evaluations. We denote our collection of problems p by $\mathcal{P}$.

For the purposes of displaying the numerical results of our testing, we use 'data profiles', which plot the proportion of problems solved within a given computational budget of α gradients. Formally, for a given solver $\mathcal{S}$ and τ, we plot the curve:

$$d_{\mathcal{S},\tau}(\alpha) := \frac{|\{p \in \mathcal{P} : N_p(\mathcal{S}; \tau) \le \alpha(n_p + 1)\}|}{|\mathcal{P}|}, \qquad 1 \le \alpha \le N_g, \tag{5.7}$$

where n_p is the dimension of problem p and N_g is the maximum computational budget (in gradients, so $N_f = N_g(n_p + 1)$).

We also include bar graphs showing the proportion of problems for which a given solver achieved the optimal value f^*, to within machine precision.

6 Numerical Results

In this section, we consider the performance of the different optimisation solvers, using the methodology from Section 5.

6.1 Comparison of Existing Solvers

First, we compare DFBOLS to other DFO and least-squares algorithms.

In the first set of results, Figure 1, we achieve similar results to those presented in [22], but considering a broader range of routines. Specifically, we compare DFBOLS to Gauss-Newton, BOBYQA, DFO-basic and Nelder-Mead. There are several important features of these results worth mentioning. Firstly, Gauss-Newton does not perform as well as DFBOLS. This is expected for noisy problems, since it uses finite differences. However, it performs worse even for noiseless problems, which is surprising. The main reason for this is that we are constraining the computational budget to measure results, which means we are unlikely to be reaching the beneficial asymptotic regime of Gauss-Newton. We are also limiting the desired accuracy in the solution, which further dilutes the benefit from the asymptotic convergence rate of Gauss-Newton.

Secondly, DFBOLS performs better than both BOBYQA and DFO-basic. The improvement here, particularly against BOBYQA, comes down to the exploitation of the least-squares problem structure. However both solvers are much more robust to noise than Gauss-Newton, indicative of the benefits of derivative-free methods. The methods BOBYQA and DFO-basic perform similarly. However, BOBYQA performs overall better, particularly when higher accuracy is required. This is likely due to the geometry-improving steps, ensuring the interpolation set continues to produce high quality quadratic models, and the work in the code to avoid significant rounding errors. Note that the choice of $\mathcal{O}(n^2)$ interpolation points for BOBYQA matches the long-term behaviour of DFO-basic, which typically grows the interpolation set from $2n+1$ to $\mathcal{O}(n^2)$ points, where it is maintained. Similarly, the larger interpolation set is important for a fair comparison to DFBOLS, since even with $2n+1$ interpolation points DFBOLS includes a significant amount of curvature information.

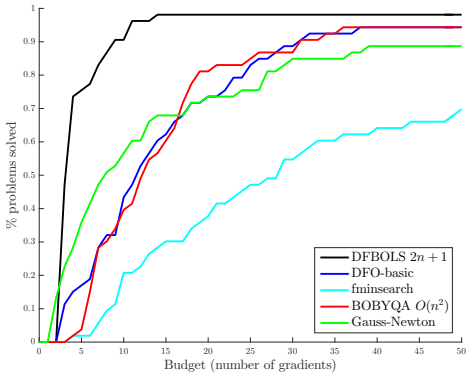
Throughout, it is of particular interest to note that DFBOLS is able to make substantial progress with very small budgets. It is here where the benefit compared to the other solvers is most noticeable.

Lastly, the performance of Nelder-Mead is substantially worse than the other general-purpose DFO solvers (BOBYQA and DFO-basic). Although it is well-known and popular among practicioners, these results do demonstrate that the development of model-based methods have noticeably improved the performance of DFO algorithms.

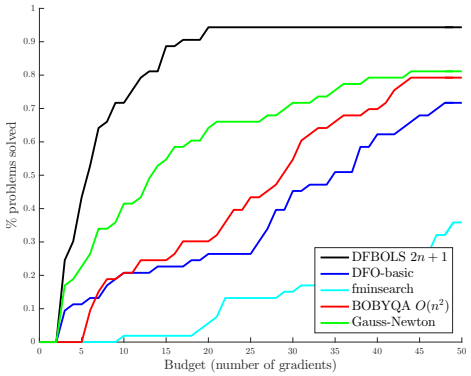
6.2 Performance of New Solver

As we have seen, the DFBOLS algorithm has strong numerical performance. This demonstrates that it would be a worthwhile inclusion in the NAG library (hence the implementation of E04JC-LS). Here we examine the performance of this algorithm.

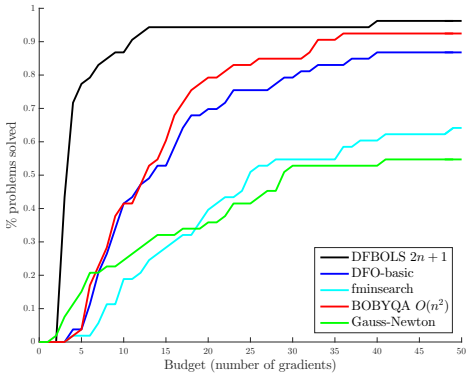
In Figure 2, we compare DFBOLS with the E04JC-LS code implemented on NAG's BOBYQA codebase. Both versions of trust region update formulae (5.1) and (5.2) are included. We see that both versions of E04JC-LS perform similarly to DFBOLS, particularly when low accuracy is required. However, when higher accuracy is required, DFBOLS performs better. As mentioned in Section 5.1, there are some small differences between DFBOLS and E04JC-LS, and these end up having



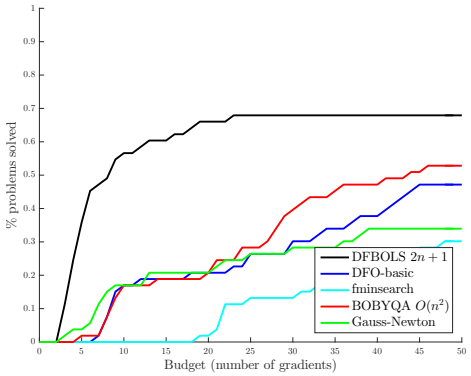
(a) Noiseless objective, $\tau = 10^{-3}$



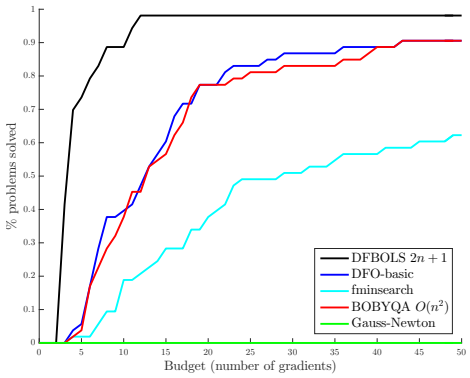
(b) Noiseless objective, $\tau = 10^{-7}$



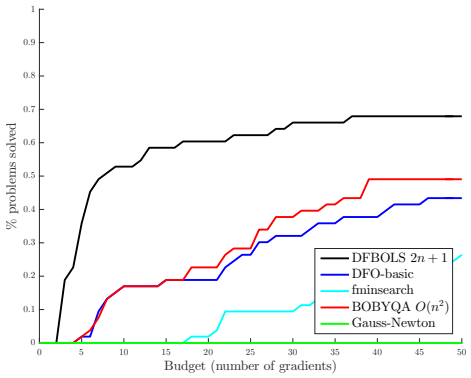
(c) Deterministic noise, $\tau = 10^{-3}$



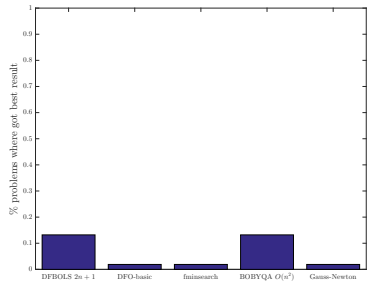
(d) Deterministic noise, $\tau = 10^{-7}$



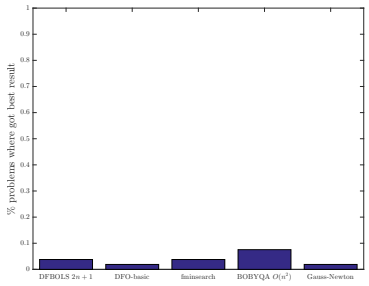
(e) Stochastic noise, $\tau = 10^{-3}$



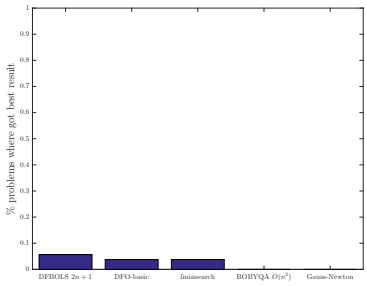
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective



(h) Deterministic noise



(i) Stochastic noise

Figure 1: Comparison of solvers: DFO-basic, BOBYQA with $(n+1)(n+2)/2$ interpolation points, DFBOLS with $2n+1$ interpolation points, Gauss Newton (line-search parameter $\eta = 0.5$), and Nelder-Mead. Noise level is $\sigma = 10^{-3}$.

a moderate impact on the solvers' performances. There is no clear preference between the two trust region update formulae (5.1) and (5.2) – the update (5.2) is better for noiseless problems, and (5.1) is better for noisy problems.

Although E04JC-LS does not perform quite as well as DFBOLS, in Figure 3, we show the same solvers as Figure 1 but replacing DFBOLS with E04JC-LS (with trust region update (5.2)). The same conclusions as drawn above continue to apply here. Specifically, E04JC-LS performs better than all other algorithms (even Gauss-Newton for noiseless problems). This holds for noiseless and noisy function evaluations, and with low or high accuracy requirements. Again, the difference between E04JC-LS and the other solvers is most noticeable for small computational budgets (e.g. fewer than 5 gradients), but it is also apparent more generally – E04JC-LS solves more problems, and solves them more quickly.

Now consider Figure 4. Here, we just consider the BOBYQA algorithm with different interpolation set sizes – $n+2$, $2n+1$ and $(n+1)(n+2)/2$. Throughout, it is preferable to have $2n+1$ points rather than $n+2$ (the minimum allowed). This tells us that having some curvature information provides substantial benefit, even if it means a greater start-up cost, and the interpolation set moves towards the solution more slowly (as more points have to be shifted). Again, there is a benefit to moving towards a fully quadratic model with $\mathcal{O}(n^2)$ interpolation points. However because of the substantial increase in the start-up cost, this benefit is not seen for small computational budgets. The benefit of having a larger interpolation set is more noticeable when high accuracy is desired, and when function evaluations are noiseless. This is an important observation – by adding noise, we lose some of the benefit of larger interpolation sets. We will return to the issue of noisy problems in Section 7.

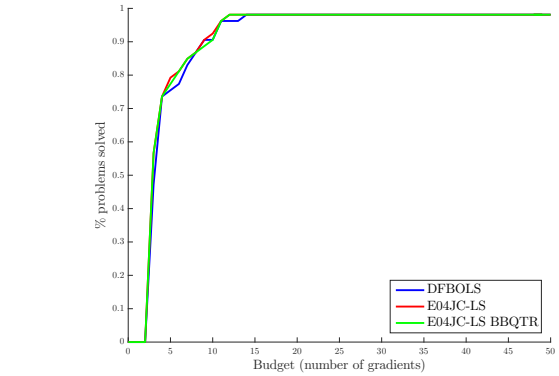
In Figure 5, we perform the same analysis but for E04JC-LS. The contrast with Figure 4 is stark – the impact of changing interpolation set size from $n+2$ to $2n+1$ to $(n+1)(n+2)/2$ is minimal. Regardless of the amount of noise or the desired accuracy, very little difference is observed. The main difference is the start-up cost, which is unavoidable with the current algorithm, where the objective is evaluated at all points in an initial interpolation set before starting the trust region iterations.

Lastly, we consider Figure 6. This compares E04JC-LS and BOBYQA with the general-purpose solvers `fminunc` (using finite differences), BFO and SID-PSM (both pattern-search DFO algorithms). None of these algorithms perform as well as E04JC-LS, which is unsurprising since none are adapted for least-squares problems; a fairer comparison is against BOBYQA.

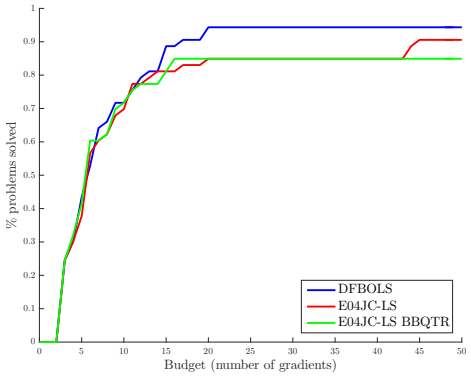
As we saw with Gauss-Newton, `fminunc` does well (comparable with BOBYQA) for noiseless problems, but significantly worse as soon as noise is introduced.

BFO tends not to perform as well as the other solvers. This is likely because it is a very general-purpose algorithm, which can handle constraints and even discrete variables. Surprisingly, the benefit from training the search parameters – and using the same problems for training as testing – is small.

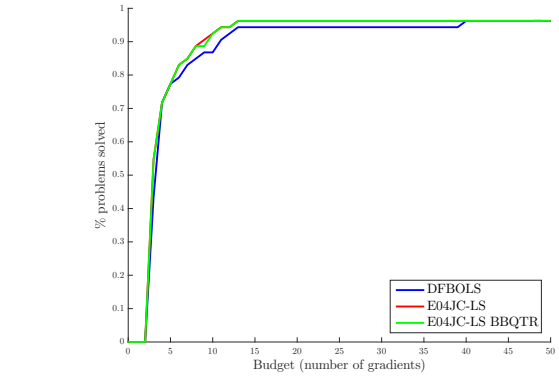
SID-PSM, however, does perform consistently well. Its performance is mostly below BOBYQA's, but it is often able to make some progress very quickly, within 1-2 gradients. This is an indication that it uses a good choice of initial search directions, which may be worth incorporating into model-based methods.



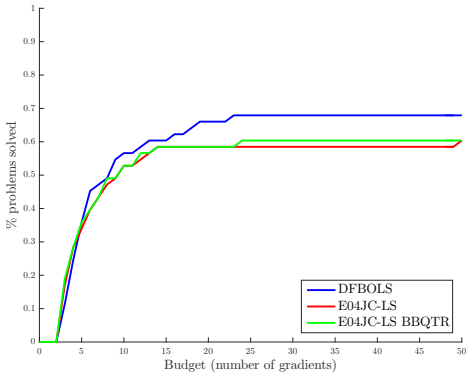
(a) Noiseless objective, $\tau = 10^{-3}$



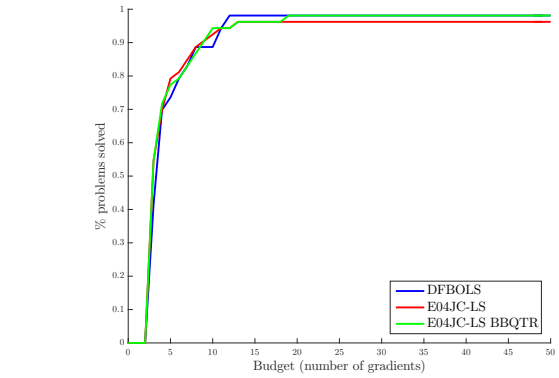
(b) Noiseless objective, $\tau = 10^{-7}$



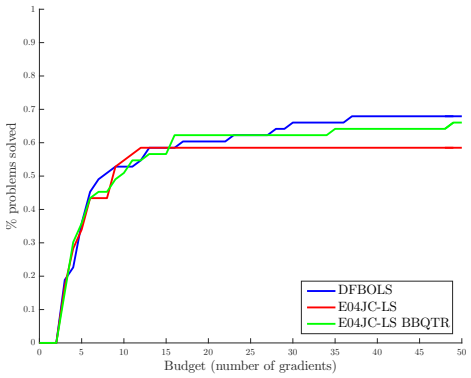
(c) Deterministic noise, $\tau = 10^{-3}$



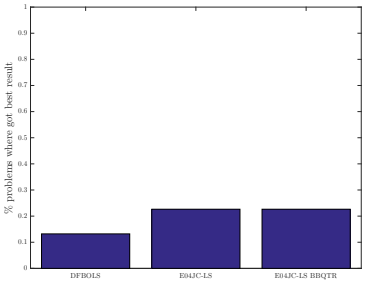
(d) Deterministic noise, $\tau = 10^{-7}$



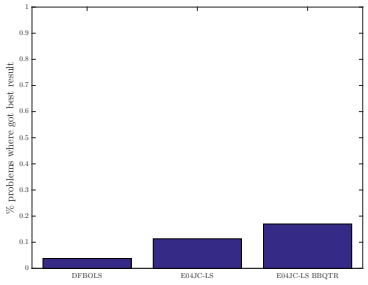
(e) Stochastic noise, $\tau = 10^{-3}$



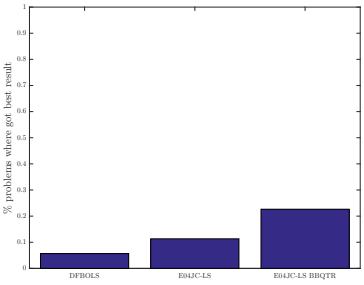
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective

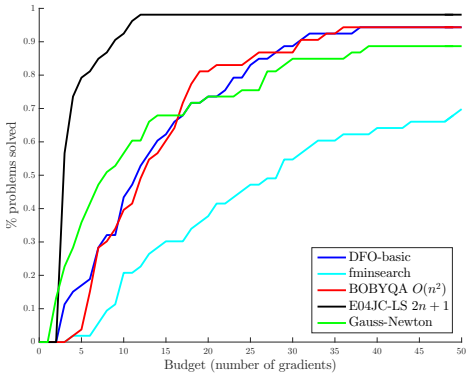


(h) Deterministic noise

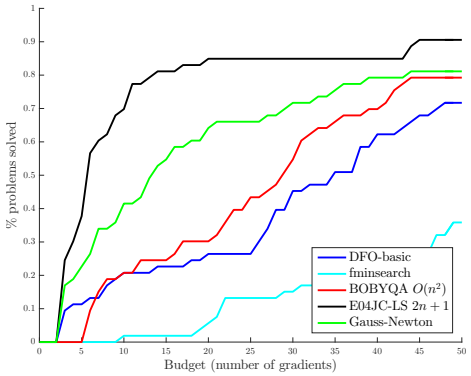


(i) Stochastic noise

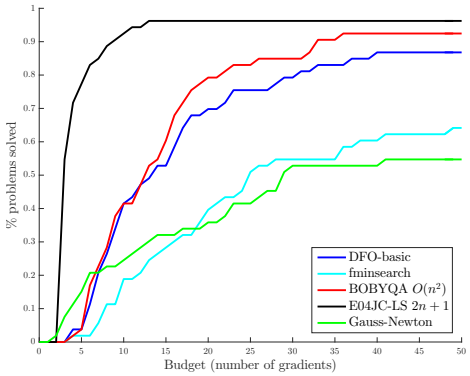
Figure 2: Comparison of different versions of DFBOLS, all with $2n + 1$ interpolation points. Versions are Zhang's code, and NAG's code with two trust region updates. Noise level is $\sigma = 10^{-3}$.



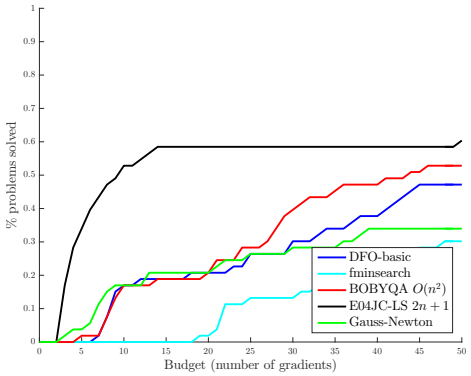
(a) Noiseless objective, $\tau = 10^{-3}$



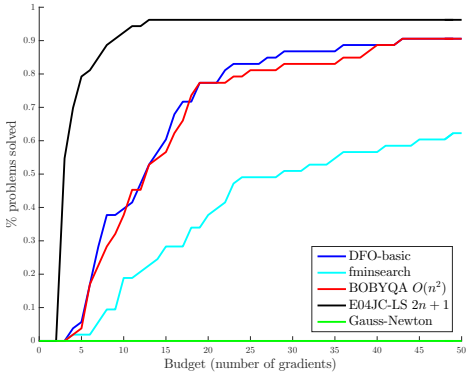
(b) Noiseless objective, $\tau = 10^{-7}$



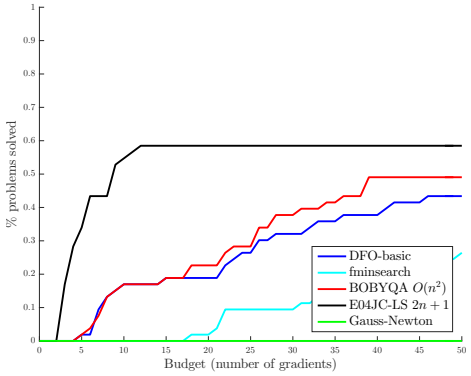
(c) Deterministic noise, $\tau = 10^{-3}$



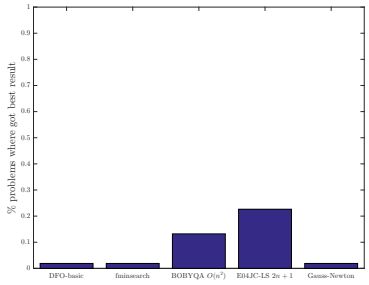
(d) Deterministic noise, $\tau = 10^{-7}$



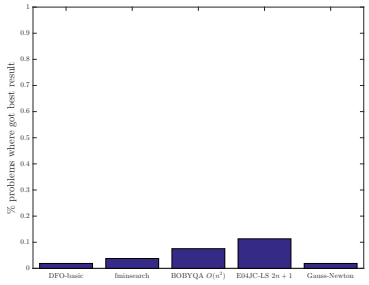
(e) Stochastic noise, $\tau = 10^{-3}$



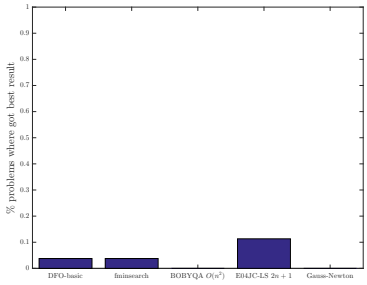
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective

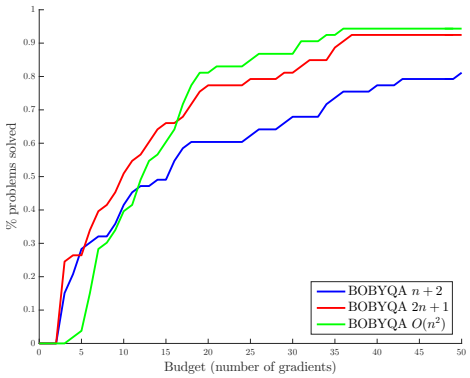


(h) Deterministic noise

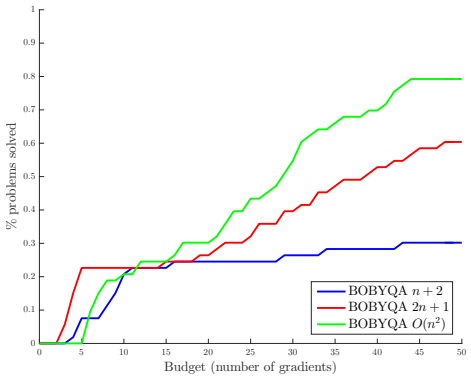


(i) Stochastic noise

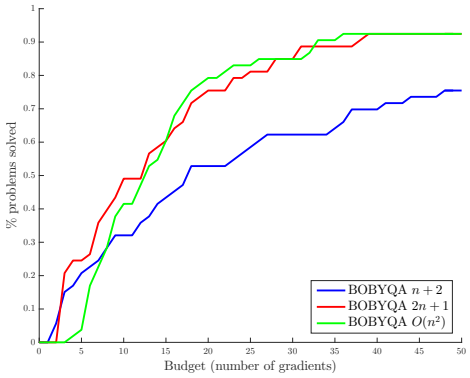
Figure 3: Comparison of solvers: DFO-basic, BOBYQA with $(n+1)(n+2)/2$ interpolation points, E04JC-LS with $2n+1$ interpolation points, Gauss Newton (line-search parameter $\eta = 0.5$), and Nelder-Mead. Noise level is $\sigma = 10^{-3}$.



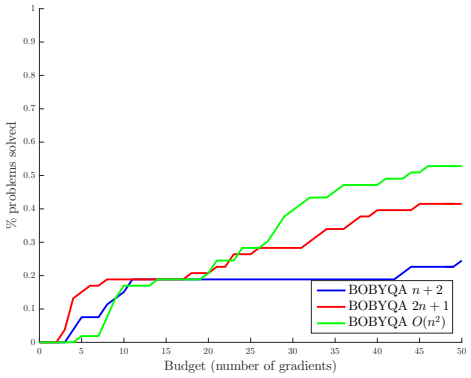
(a) Noiseless objective, $\tau = 10^{-3}$



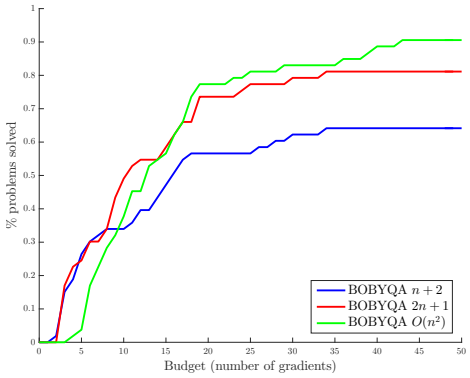
(b) Noiseless objective, $\tau = 10^{-7}$



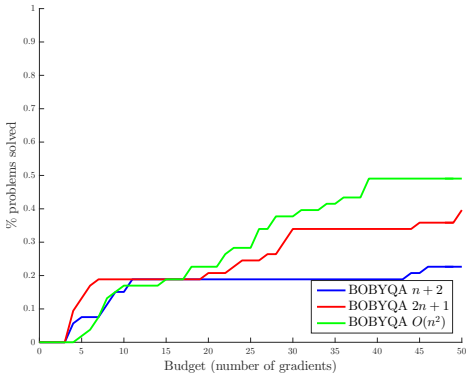
(c) Deterministic noise, $\tau = 10^{-3}$



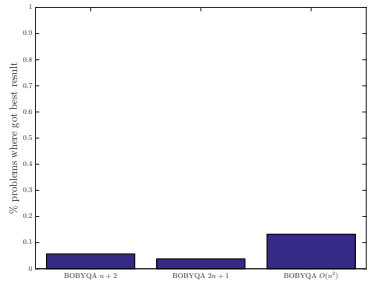
(d) Deterministic noise, $\tau = 10^{-7}$



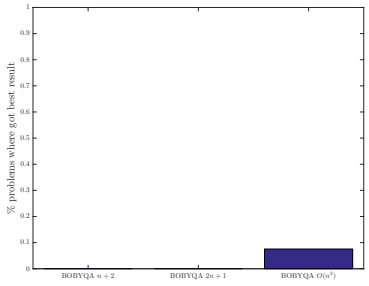
(e) Stochastic noise, $\tau = 10^{-3}$



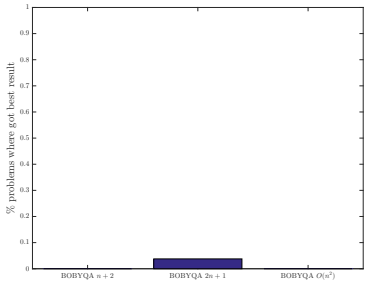
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective

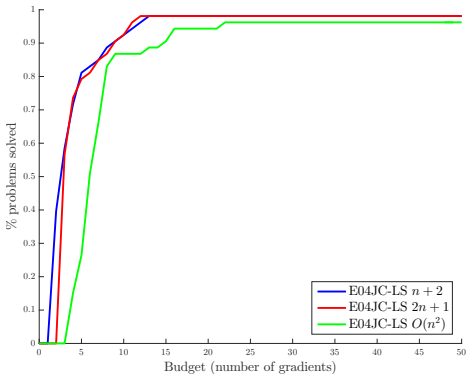


(h) Deterministic noise

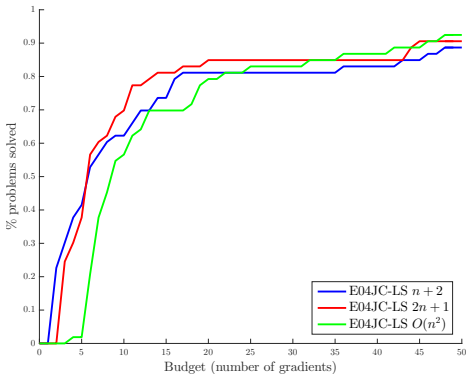


(i) Stochastic noise

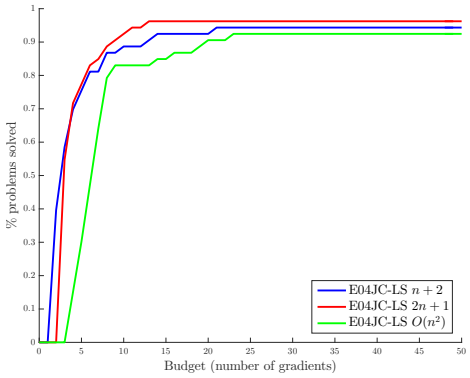
Figure 4: Comparison of BOBYQA with interpolation set sizes $n + 2$, $2n + 1$ and $(n + 1)(n + 2)/2$. Noise level is $\sigma = 10^{-3}$.



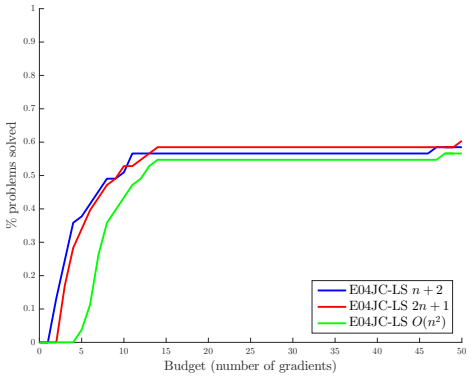
(a) Noiseless objective, $\tau = 10^{-3}$



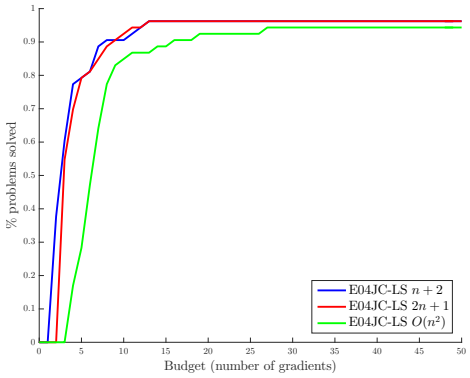
(b) Noiseless objective, $\tau = 10^{-7}$



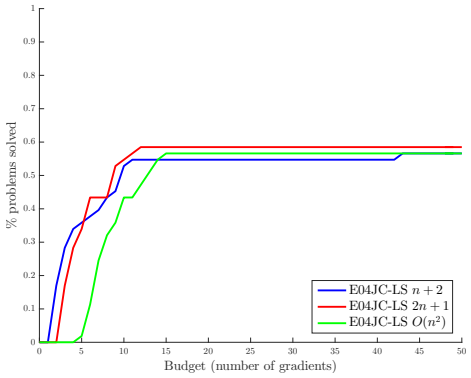
(c) Deterministic noise, $\tau = 10^{-3}$



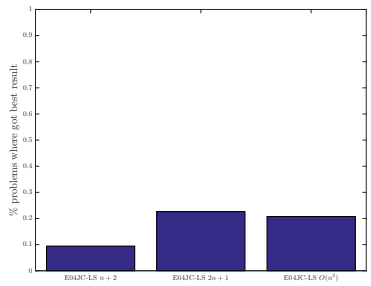
(d) Deterministic noise, $\tau = 10^{-7}$



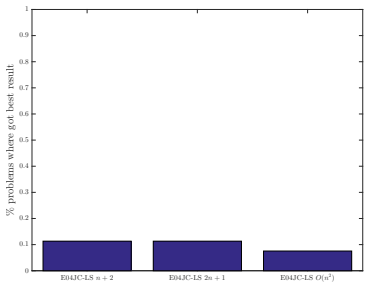
(e) Stochastic noise, $\tau = 10^{-3}$



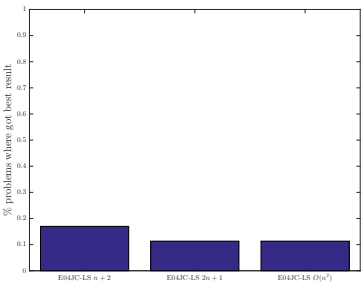
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective

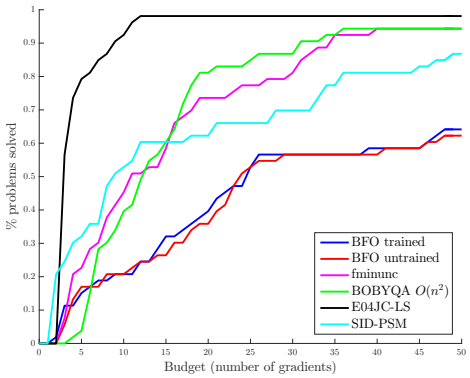


(h) Deterministic noise

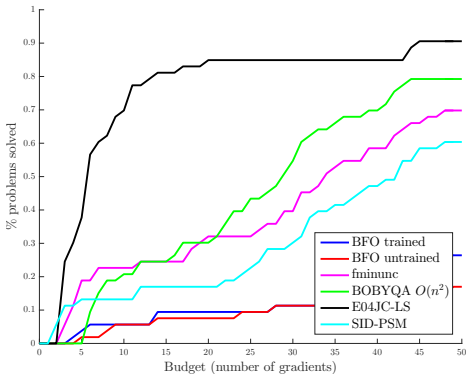


(i) Stochastic noise

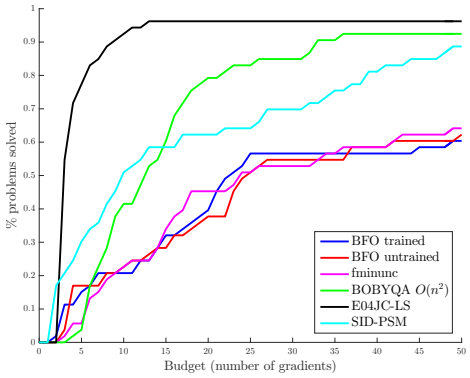
Figure 5: Comparison of E04JC-LS with interpolation set sizes $n + 2$, $2n + 1$ and $(n + 1)(n + 2)/2$. Noise level is $\sigma = 10^{-3}$.



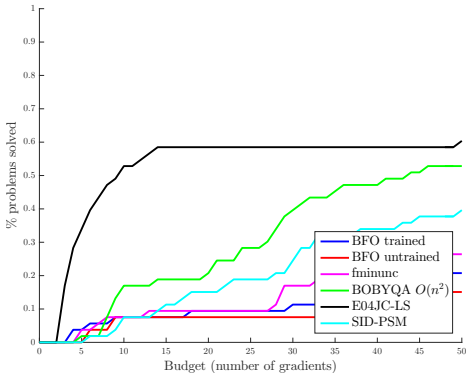
(a) Noiseless objective, $\tau = 10^{-3}$



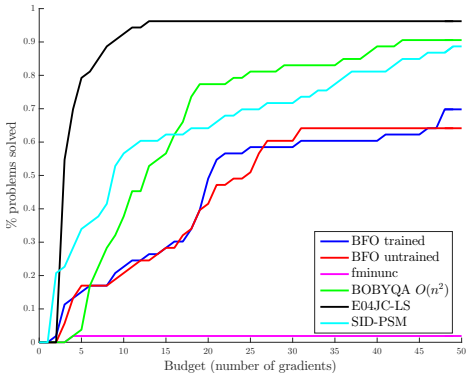
(b) Noiseless objective, $\tau = 10^{-7}$



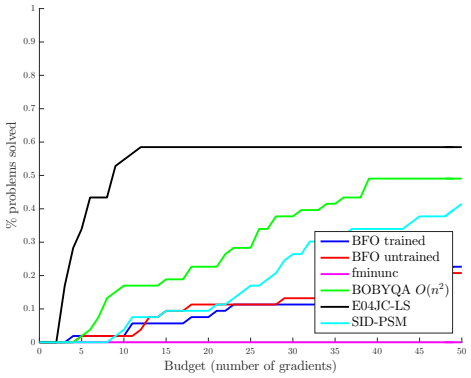
(c) Deterministic noise, $\tau = 10^{-3}$



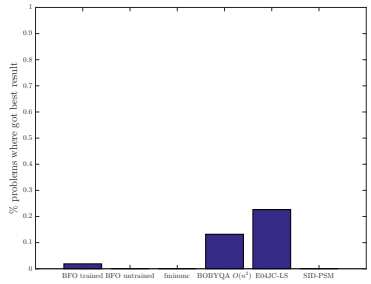
(d) Deterministic noise, $\tau = 10^{-7}$



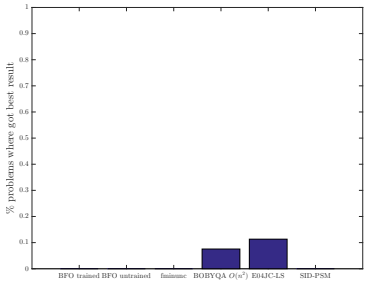
(e) Stochastic noise, $\tau = 10^{-3}$



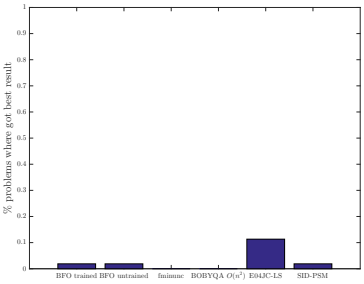
(f) Stochastic noise, $\tau = 10^{-7}$



(g) Noiseless objective



(h) Deterministic noise



(i) Stochastic noise

Figure 6: Comparison of solvers: fminunc, BFO and SID-PSM, compared to E04JC-LS with $2n+1$ points and BOBYQA with $(n+1)(n+2)/2$ points. Noise level is $\sigma = 10^{-3}$.

7 Future Research Directions

From the work described in previous sections, there are several ideas that provide us with some future research directions. In this section, we will examine some of these ideas, and how further work may improve upon current DFO algorithms. This work has the potential improve both DFO for general problems and least-squares problems.

7.1 Startup Cost

One important use case of DFO algorithms is when function evaluations are computationally expensive compared to the cost of the algorithm. This situation occurs for instance in data fitting for finance and data assimilation for weather forecasting.

In all DFO algorithms mentioned previously, the first step is to generate a set of sample points and evaluate the objective at each of these. The set of points is of size $\mathcal{O}(n)$ or $\mathcal{O}(n^2)$. Only after these function evaluations have been performed does the algorithm properly commence and progress is made. We see this in, e.g. Figure 1, where no problems are solved for very small budgets (e.g. less than 1 gradient).

Although this step may be parallelisable, if function evaluation is expensive then we may not have time for even these initial evaluations. Also, in this scenario, practitioners may wish to see some progress immediately, and only be interested in finding some reduction in the objective, rather than a point close to a local minimum. It is therefore desirable to consider ways we can make progress before reaching the minimum $n + 2$ function evaluations required for BOBYQA and E04JC-LS.

One approach is, if we only have a small number of points, to limit our search directions to a subspace of $\mathbb{R}^n$ – the span of directions in which we already have sampled the objective. In the least-squares case, we also have the option of constructing a simpler model by only evaluating a subset of the residual functions r_i. Together, we can think of approximating our true Jacobian with a subset of its rows and columns (and setting all other elements to zero). Similar approaches have recently been considered for machine learning problems, e.g. [15, 16, 21]. A modification of the Gauss-Newton method (using derivatives) in which only a subset of directions were searched was studied in [6], and good numerical performance was observed.

7.2 Performance under noise

Throughout the numerical results, each solver solved substantially fewer problems (especially to high accuracy) once noise was introduced. Function evaluations may be subject to noise in reality from the incorporation of input uncertainty, or the use of Monte Carlo simulations. As we have seen in Section 6, finite differences no longer produce accurate values, so derivative-free algorithms are crucial. However, E04JC-LS with $2n + 1$ interpolation points solves (in a 50 gradient budget, to within accuracy $\tau = 10^{-7}$) only approximately 60% of problems with stochastic noise, compared to over 90% of problems without noise.

The model-based methods currently construct models based on interpolation conditions such as (2.8). However if evaluation is noisy, then the exact numerical values

of $f(\mathbf{y})$ are unreliable. This leads to the question – why impose exact interpolation conditions when function evaluations are noisy? In general, the relative success of DFO algorithms compared to finite difference-based algorithms is related to the fact that noise affects the interpolation (over points separated by relatively large gaps) less than derivative calculations (over points very close together). However, as we converge to a solution, this becomes less true. It is worth noting that [3, Chapter 4] already considers the case of an over-determined interpolation problem, so the linear system from (2.8) is solved in a least-squares minimisation sense. That is, the interpolation can never be solved exactly.

One research direction would be to consider when and how to build approximate interpolation models. This would allow uncertainty in function evaluations to be naturally captured, and likely have the extra benefit of improving the speed of the interpolation step (currently the most expensive part of BOBYQA and E04JC-LS).

By analogy, the trust region subproblem (2.6) is often not solved to high precision. In fact, for a globally convergent algorithm, the only requirement is that the solution produces at least as much reduction as steepest descent. It is possible that similar errors may also be allowable for the interpolation step without breaking any convergence guarantees.

7.3 Large-scale problems

Many important optimisation problems, such as oil well modelling and data assimilation for weather forecasting, have high dimension. Currently, DFO software is designed to work on problems of small dimension, e.g. up to $n = \mathcal{O}(100)$. In higher dimensions, several problems can occur:

- Evaluating the function even n times may be prohibitively expensive;
- There are many more possible search directions, so more iterations are likely to achieve the same improvement; and
- For very large n, even simple operations such as matrix-vector multiplication may be too expensive to perform at each iteration. Similarly, the memory required to store a $n \times n$ matrix may be more than is available on a given machine.

Although parallelisation can offset some of these difficulties, DFO algorithms do not have this ability at the moment.

One approach for improving DFO algorithms in high dimensions would be to exploit sparsity in the problem. For instance, efficient construction of models using techniques from compressive sensing was considered in [1].

Also, the techniques discussed in Section 7.1 would also be applicable here, where we search a tractable subspace. In the least-squares case, we also have the option of building models and evaluating objectives for a subset of residuals.

8 Conclusions

Derivative-free optimisation is an important area that has seen substantial progress over the last 10-15 years. It is most useful when function evaluation is noisy (so finite differences are inaccurate) or expensive (so progress must be made in few iterations, and finding high-accuracy solutions is not appropriate). There are numerous real-world situations where such methods are appropriate, for instance in finance, energy, climate and engineering design.

One important class of derivative-free algorithms are extensions of classical trust-region methods, where models approximating the objective are constructed by interpolation rather than by Taylor series expansions. To do this, one maintains a set of points, and must consider the geometry of these points and how to appropriately solve the resulting interpolation problem (e.g. when it is underdetermined). Several algorithms of this type exist, and the NAG library has one such method, called BOBYQA.

Least-squares problems are one of the most common optimisation problems, and are particularly important for data fitting. We can combine general-purpose derivative-free methods such as BOBYQA with classical gradient-based least-squares methods to build derivative-free methods which properly exploit the underlying structure in the problem. One example is DFBOLS, based on the methods described in Section 4 and [22]. The same approach has been used to extend the NAG implementation of BOBYQA to produce a new solver, called E04JC-LS, which will feature in a future release of the NAG library.

By considering a well-known set of test least-squares problems, we have been able to compare different solvers. We see that when computational budget is the main limiting factor for a user, that derivative-free solvers outperform comparable derivative-based solvers. This is particularly noticeable when function evaluation is noisy, and derivative-based solvers fail to make any progress. We have also seen a substantial benefit from exploiting the least-squares structure of problems. Ultimately, DFBOLS and E04JC-LS perform very well for a wide range of problems in a wide range of settings.

This work has also raised questions and topics which could form the basis of future research work. Of particular relevance are problems with noisy function evaluations and/or high underlying dimension. We may find both of these features in problems in oil well modelling and weather forecasting, for instance.

Since computational budget is often the limiting factor, particularly for large scale problems, we wish to make the best possible use of every function evaluation. Currently the initial function evaluations are at a predetermined set of points, but sometimes just performing these evaluations may be prohibitively expensive. Thus we should consider how to more efficiently build an initial interpolation set, using all the information available as we receive it, even if we only have 2 or 3 function values.

Also, the imposition of exact interpolation when building our model is computationally expensive and inappropriate for noisy functions. We should investigate methods for building approximate interpolation models, or performing interpolation under noise. This could improve the robustness of DFO methods for noisy functions, and improve the tractability of DFO algorithms for large scale problems.

8.1 Postscript (November 2018)

The code E04JC-LS has been incorporated into the NAG Library, and is available as routine E04FFF; see

`https://www.nag.co.uk/content/derivative-free-optimization-data-fitting`

for details.

References

[1] A. S. Bandeira, K. Scheinberg, and L. N. Vicente, *Computation of sparse low degree interpolating polynomials and their application to derivative-free optimization*, Math. Program., 134 (2012), pp. 223–257.

[2] A. R. Conn, K. Scheinberg, and P. L. Toint, *A Derivative Free Optimization Algorithm in Practice*, in Proc. 7th AIAA/USAF/NASA/ISSMO Symp. Multidiscip. Anal. Optim., St. Louis, MO, 1998.

[3] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to Derivative-Free Optimization*, vol. 8 of MPS-SIAM Series on Optimization, MPS/SIAM, Philadelphia, 2009.

[4] A. L. Custódio, H. Rocha, and L. N. Vicente, *Incorporating minimum Frobenius norm models in direct search*, Comput. Optim. Appl., 46 (2010), pp. 265–278.

[5] A. L. Custódio and L. N. Vicente, *Using Sampling and Simplex Derivatives in Pattern Search Methods*, SIAM J. Optim., 18 (2007), pp. 537–555.

[6] N. W. Eizenberg, *Parameter Estimation for Climate models*, msc thesis, University of Oxford, 2015.

[7] G. Fasano, J. L. Morales, and J. Nocedal, *On the geometry phase in model-based algorithms for derivative-free optimization*, Optim. Methods Softw., 24 (2009), pp. 145–154.

[8] P. E. Gill and W. Murray, *Algorithms for the Solution of the Nonlinear Least-Squares Problem*, SIAM J. Numer. Anal., 15 (1978), pp. 977–992.

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

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

[11] NAG, *E04FC Unconstrained Least-Squares (function values only)*, 2016.

[12] ——, *E04JC Bound-constrained derivative-free optimisation (BOBYQA)*, 2016.

[13] J. A. Nelder and R. Mead, *A simplex method for function minimization*, Comput. J., 7 (1964), pp. 308–313.

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

[15] M. Pilanci and M. J. Wainwright, *Randomized Sketches of Convex Programs With Sharp Guarantees*, IEEE Trans. Inf. Theory, 61 (2015), pp. 5096–5115.

[16] ——, *Iterative Hessian sketch: Fast and accurate solution approximation for constrained least-squares*, J. Mach. Learn. Res., 17 (2016), pp. 1–38.

[17] M. Porcelli and P. L. Toint, *BFO, a trainable derivative-free Brute Force Optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables*, tech. rep., University of Namur, 2015.

[18] M. J. D. Powell, *On the use of quadratic models in unconstrained minimization without derivatives*, Optim. Methods Softw., 19 (2004), pp. 399–411.

[19] ——, *The NEWUOA software for unconstrained optimization without derivatives*, vol. 83 of Nonconvex Optimization and Its Applications, Springer, Boston, MA., 2006, pp. 255–297.

[20] ——, *The BOBYQA algorithm for bound constrained optimization without derivatives*, tech. rep., University of Cambridge, 2009.

[21] F. Roosta-Khorasani and M. W. Mahoney, *Sub-Sampled Newton Methods I: Globally Convergent Algorithms*, arXiv Prepr. arXiv1601.04737, (2016).

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