Science and Technology Facilities Council

Preprint
STFC-P-2024-001

Approximating large-scale Hessian matrices using secant equations

JM Fowkes, NIM Gould, JA Scott

May 2024

Submitted for publication in ACM Transactions on Mathematical Software

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445577
email: library@stfc.ac.uk

Science and Technology Facilities Council reports are available online at:
https://epubs.stfc.ac.uk

Accessibility: a Microsoft Word version of this document (for use with assistive technology) may be available on request.

ISSN 2753-5819

STFC Author Identifiers (ORCIDs)

Author ORCIDs are provided where available.

Jaroslav Fowkes	0000-0002-8048-4572
Nicholas Gould	0000-0002-1031-1588
Jennifer Scott	0000-0002-2130-1091

Approximating large-scale Hessian matrices using secant equations

Jaroslav M. Fowkes* Nicholas I. M. Gould* Jennifer A. Scott*†

May 16, 2024

Abstract

Large-scale optimization algorithms frequently require sparse Hessian matrices that are not readily available. Existing methods for approximating large sparse Hessian matrices either do not impose sparsity or are computationally prohibitive. To try and overcome these limitations, we propose a novel approach that seeks to satisfy as many componentwise secant equations as necessary to define each row of the Hessian matrix. A naive application of this approach is prohibitively expensive on Hessian matrices that have some relatively dense rows but by carefully taking into account the symmetry and connectivity of the Hessian matrix we are able devise an approximation algorithm that is fast and efficient with scope for parallelism. Example sparse Hessian matrices from the CUTEst test problem collection for optimization illustrate the effectiveness and robustness of our proposed method.

Keywords: Sparse nonlinear systems, sparse Hessian matrices, secant equations.

1 Introduction

Let us suppose we are given a smooth objective function $f(x)$ of n variables x, whose gradient $g(x) := \nabla_x f(x)$ is known. Our aim is to compute estimates $B^{(k)}$ of the Hessian matrix $H(x) := \nabla_{xx} f(x)$ at a sequence of given iterates $x^{(k)}$. Such a requirement lies at the heart of both Newton-like methods for minimizing f and methods that try to assess the stability of its gradient. We are particularly interested in the case where $H(x) = \{h_{ij}(x)\}$ is large and sparse with a known sparsity pattern $\mathcal{S}(H(x)) := \{(i,j) : h_{ij}(x) \neq 0\}$. We say that the Hessian matrix has an *entry* in row i and column j if $h_{ij}(x) \neq 0$ for some x.

If we are extremely fortunate, an analytic expression for the Hessian matrix may be available. Alternatively, it may be possible to obtain approximations to $H(x)$ using

*STFC Rutherford Appleton Laboratory, Harwell Campus, Didcot, Oxfordshire, OX11 0QX, UK

†School of Mathematical, Physical and Computational Sciences, University of Reading, Reading RG6 6AQ, UK. Correspondence to: jaroslav.fowkes@stfc.ac.uk. All authors were supported by EPSRC grant number EP/X032485/1.

automatic differentiation [16]. If not, we could consider finite-difference approximations to compute one column of $B^{(k)}$ at a time, for instance using forward differences

$$B^{(k)}e_i \approx \frac{g(x^{(k)} + \Delta e_i) - g(x^{(k)})}{\Delta},$$

where e_i is the i-th unit vector and Δ is an appropriate small scalar [10]; more expensive but more accurate central differences can also be used [6, §5.6]. Note that if the Hessian matrix is dense then $n + 1$ gradient evaluations are needed to find $B^{(k)}$. If, however, it is sparse, it may be possible to partition $\mathcal{N} = \{1, \ldots, n\}$ into a small number $q \ll n$ of disjoint subsets $\mathcal{N} = \bigcup_{j=1}^{q} \mathcal{I}_j$ with $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$ $(1 \leq i < j \leq q)$, such that the rows indexed in each $\mathcal{I}_j$ are orthogonal. In this case,

$$\sum_{i \in \mathcal{I}_j} B^{(k)}e_i \approx \frac{g(x^{(k)} + \Delta \sum_{i \in \mathcal{I}_j} e_i) - g(x^{(k)})}{\Delta},$$

and only $q+1$ gradients are required. Exploiting symmetry reduces the count further [2,20].

Another possible approach is to require that $B^{(k)}$ satisfies the secant equation

$$B^{(k)}(x^{(k)} - x^{(k-1)}) = g^{(k)} - g^{(k-1)},$$

where $g^{(k)} := g(x^{(k)})$. Traditionally, $B^{(k)}$ is obtained from the previous estimate $B^{(k-1)}$ by imposing the secant equation and requiring $B^{(k)} - B^{(k-1)}$ is of low rank [6,19]—usually rank one or two—rather than sparse. Thus it is highly unlikely that $B^{(k)}$ will be sparse, even if its predecessor was, and the computational gains from utilising sparse linear algebra are lost. Although there have been attempts to derive sparse updates [7, 23, 24], their stability is a problem [22].

Secant methods start from an initial estimate $B^{(0)}$ (often $B^{(0)} = I$) and build up the approximation as new points are added. Thus, after k steps, a rank k (for the SR1 method) or $2k$ (for methods like BFGS or DFP) update will have been applied to $B^{(0)}$. A related limited-memory approach is to use dense low-rank updates, but instead of applying them all to $B^{(0)}$, apply the last m updates as if they had been applied to a re-initialized $B^{(k-m)}$. In practice, the sequence of m previous differences $s^{(l)} := x^{(l)} - x^{(l-1)}$ and $y^{(l)} := g^{(l)} - g^{(l-1)}$ $(l = k-1, \ldots, k-m)$ are recorded and then the effect (product with, solve with) of using the relevant $B^{(k)}$ is computed as it is required. Efficient algorithms exist for this [17, 18] but, as before, without attempting to impose the sparsity structure of $H^{(k)} := H(x^k)$ on $B^{(k)}$.

If the function $f(x)$ is partially separable, that is,

$$f(x) = \sum_{i=1}^{p} f_i(x),$$

where each "element" $f_i(x)$ has a large invariant subspace [15], then an approximation

$$B^{(k)} = \sum_{i=1}^{p} B_i^{(k)}$$

can be computed in which each element Hessian estimate $B_i^{(k)}$ satisfies its own secant equation

$$B_i^{(k)}(x^{(k)} - x^{(k-1)}) = g_i(x^{(k)}) - g_i(x^{(k-1)}),$$

where $g_i(x) := \nabla_x f_i(x)$. The invariant subspace assumption implies g_i and $B_i^{(k)}$ are structured. In particular, any differentiable $f(x)$ for which the Hessian matrix is sparse is partially separable [14], and in this case, each element secant equations only involves a few variables, leading to an excellent sparse approximation. This and its generalization to group-partial separability [3] forms the basis of the approximations used in LANCELOT [4]. However, it does not appear to have been widely adopted, and this is apparently because users are either unable or unwilling to provide the necessary separability structure.

From the user's perspective, a more appealing approach, and the one we advocate in this paper, is to use the past m accumulated data pairs $\{s^{(l)}, y^{(l)}\}_{l=k-m+1}^{k}$ and the sparsity structure of $H^{(k)}$ to estimate $B^{(k)}$ directly. The only developments we are aware of in this direction are that of Fletcher, Grothey and Leyffer [8] and our recent sparse linear least squares approach [9]. The original idea of Fletcher et al. was to build an approximation $B^{(k)}$ that satisfies as well as possible the multiple secant conditions

$$B^{(k)}s^{(l)} = y^{(l)}, \qquad l = k - m + 1, \ldots, k, \tag{1.1}$$

together with the symmetry condition

$$B^{(k)} = (B^{(k)})^T,$$

and the sparsity condition

$$\mathcal{S}(B^{(k)}) = \mathcal{S}(H^{(k)}).$$

Because (1.1) will usually be inconsistent, a reasonable compromise is to solve instead the convex quadratic program

$$\min_{B^{(k)}} \sum_{l=k-m+1}^{k} \|B^{(k)}s^{(l)} - y^{(l)}\|_F^2 \text{ such that } B^{(k)} = (B^{(k)})^T \text{ and } \mathcal{S}(B^{(k)}) = \mathcal{S}(H^{(k)}),$$

where $\|B\|_F^2$ denotes the squared Frobenius norm of the matrix B. The solution $B^{(k)}$ may be found by solving a linear system of order n_e, the number of entries in the upper triangle of $H^{(k)}$. As n_e can be large, estimates of the solution $B^{(k)}$ may better be found using an iterative scheme such as conjugate gradients [8], but even this can be prohibitively expensive to compute, especially if such an approximation is to be used within an optimization code.

The idea behind our recent sparse linear least squares approach [9] is to instead stack the nonzero entries in the upper triangular part of $B^{(k)}$ row-by-row above each other in a vector $z^{(k)}$ (equivalently, the entries in the lower triangular part are stacked column-by-column). In this way, we redefine the problem as a large sparse linear system of equations given by

$$A^{(k)}z^{(k)} = c^{(k)}, \tag{1.2}$$

where the matrix $A^{(k)}$ and the vector $c^{(k)}$ are known and depend on the secant conditions (1.1) (see [9] for details). This enables us to find $B^{(k)}$ by computing the least squares solution to (1.2), that is, the $z^{(k)}$ that minimizes

$$\|A^{(k)}z^{(k)} - c^{(k)}\|_2^2,$$

using existing sparse linear algebra solvers. However, as $z^{(k)}$ by construction contains all the nonzero entries in the upper (equiv. lower) triangular part of the Hessian, the resulting linear system can be huge for large Hessian matrices and thus prohibitively expensive to solve, due to the large flop-count required, even by state-of-the-art sparse direct linear solvers. This unfortunately greatly hinders the ability of this approach to be incorporated inside existing optimization solvers and thereby its applicability in practice.

The objective of this paper is therefore to develop a more computationally efficient approach than either [8] or [9] to estimating a sparse Hessian matrix given m past iterates and gradient values. We assume throughout that most of the rows of the Hessian we seek to estimate are very sparse but there may be a small number of rows that are relatively dense. The paper is organised as follows. In Section 2, we lay the groundwork for our sparse Hessian approximation technique and introduce our proposed new approximation algorithm. In Section 3, we report results of numerical experiments that illustrate the potential of the new algorithm for approximating large sparse Hessian matrices in practice. Finally, concluding remarks and suggestions for future work are given in Section 4.

2 Sparse Hessian approximation

2.1 Hessian matrices with all rows sparse

We start by assuming that each row of the Hessian matrix has only a small number of entries; in the next section we allow for the case of practical importance in which some rows have a larger number of entries. As in [8,9], our proposed approach uses accumulated data. But rather than imposing the full secant conditions (1.1) for the previous m steps, for each row i in the approximate Hessian $B^{(k)}$ we aim to satisfy as many *componentwise* equations

$$e_i^T B^{(k)} s^{(l)} = e_i^T y^{(l)}, \quad l = k, k-1, \ldots, \tag{2.3}$$

as are necessary to define the row. We present two algorithms that solve the above componentwise equations in the case where each row has only a small number of entries.

Row-wise independent algorithm

Equation (2.3) can be rewritten as

$$\sum_{j \in \mathcal{S}_i^{(k)}} b_{ij}^{(k)} s_j^{(l)} = y_i^{(l)}, \quad l = k, k-1, \ldots, \tag{2.4}$$

where

$$\mathcal{S}_i^{(k)} := \{j : h_{ij}^{(k)} \neq 0\}$$

is the set of column indices of the unknown entries in row i of the Hessian. Naively (and neglecting any inconsistencies or redundancies in dependencies), to compute row i, we need as many equations (2.4) as there are entries in the row. Let $z_i^{(k)}$ denote the vector of entries in row i and let $nz_i := |\mathcal{S}_i^{(k)}|$ be the number of such entries. Then the equations (2.4) that must be satisfied by $z_i^{(k)}$ can be rewritten as an $nz_i \times nz_i$ dense linear system

$$A_i^{(k)} z_i^{(k)} = c_i^{(k)}, \tag{2.5}$$

where $A_i^{(k)}$ is the matrix whose rows $l = k, k-1, \ldots, k-nz_i+1$ consist of the entries $s_j^{(l)}$ indexed by $j \in \mathcal{S}_i^{(k)}$ and $c_i^{(k)}$ is the vector with entries $y_i^{(l)}$, $l = k, k-1, \ldots, k-nz_i+1$. The following simple example illustrates this matrix formulation.

Example 1 Consider the 3×3 approximate Hessian matrix

$$B^{(k)} = \begin{pmatrix} b_{11}^{(k)} & b_{12}^{(k)} & 0 \\ b_{21}^{(k)} & 0 & b_{23}^{(k)} \\ 0 & b_{32}^{(k)} & b_{33}^{(k)} \end{pmatrix}, \quad \text{with } b_{12}^{(k)} = b_{21}^{(k)} \text{ and } b_{23}^{(k)} = b_{32}^{(k)}.$$

For row $i = 2$, $\mathcal{S}_2^{(k)} = \{1, 3\}$, $nz_2 = 2$ and the linear system (2.5) is given by

$$\underbrace{\begin{pmatrix} s_1^{(k)} & s_3^{(k)} \\ s_1^{(k-1)} & s_3^{(k-1)} \end{pmatrix}}_{A_2^{(k)}} \underbrace{\begin{pmatrix} b_{21}^{(k)} \\ b_{23}^{(k)} \end{pmatrix}}_{z_2^{(k)}} = \underbrace{\begin{pmatrix} y_2^{(k)} \\ y_2^{(k-1)} \end{pmatrix}}_{c_2^{(k)}}.$$

Similarly, for row $i = 3$, $\mathcal{S}_3^{(k)} = \{2, 3\}$, $nz_3 = 2$ and (2.5) is given by

$$\underbrace{\begin{pmatrix} s_2^{(k)} & s_3^{(k)} \\ s_2^{(k-1)} & s_3^{(k-1)} \end{pmatrix}}_{A_3^{(k)}} \underbrace{\begin{pmatrix} b_{32}^{(k)} \\ b_{33}^{(k)} \end{pmatrix}}_{z_3^{(k)}} = \underbrace{\begin{pmatrix} y_3^{(k)} \\ y_3^{(k-1)} \end{pmatrix}}_{c_3^{(k)}}.$$

It may well be the case that we do not have sufficient past data pairs $\{s^{(l)}, y^{(l)}\}_{l=k-m+1}^{k}$ available for the matrices $A_i^{(k)}$ to be non-singular, however there are a number of approaches one can pursue to remedy this. For clarity of exposition we defer the discussion of such approaches for handling the case of insufficient data to Section 2.4.

Using (2.5), we can compute the sparse Hessian approximation $B^{(k)}$ as outlined in Algorithm 2.1. Both the off-diagonal entries $b_{ij}^{(k)}$ and $b_{ji}^{(k)}$ are computed, and then, in the final step, the average is taken to obtain a symmetric approximate Hessian matrix. If the full sparse Hessian approximation $B^{(k)}$ is stored, then the rows can be computed in parallel in any order. However, it is often the case that only the upper triangular (or equivalently lower triangular) part of the sparse Hessian is stored and it may not be desirable to form the full Hessian, in which case the rows cannot be computed in parallel but must be computed in sequence.

Algorithm 2.1 Sparse Hessian approximation (row-wise independent)

1: **for** $i = 1, \ldots, n$ **do**
2: Compute the nz_i entries in row i by constructing and solving linear system (2.5).
3: **end for**
4: Symmetrise $B^{(k)} := (B^{(k)} + (B^{(k)})^T)/2$.

Row-wise dependent algorithm

Taking an average of the two off-diagonal entries (or accepting one of the computed values) does not truly account for symmetry. Instead, consider rewriting (2.4) as

$$\sum_{j \in \mathcal{U}_i^{(k)}} b_{ij}^{(k)} s_j^{(l)} = y_i^{(l)} - \sum_{j \in \mathcal{K}_i^{(k)}} b_{ij}^{(k)} s_j^{(l)}, \quad l = k, k-1, \ldots, \tag{2.6}$$

where

$$\mathcal{K}_i^{(k)} := \{j : h_{ij}^{(k)} \neq 0 \text{ and } b_{ij}^{(k)} \text{ is already known}\} \text{ and } \mathcal{U}_i^{(k)} := \{j : h_{ij}^{(k)} \neq 0\} \setminus \mathcal{K}_i^{(k)},$$

are the column indices of the known and unknown entries, respectively, in row i of the Hessian. It follows that data from (at least) $nu_i := |\mathcal{U}_i^{(k)}|$ previous steps is required. The i-th componentwise equation (2.6) can be rewritten as the dense linear system

$$U_i^{(k)} z_i^{(k)} = c_i^{(k)} - K_i^{(k)} w_i^{(k)}, \tag{2.7}$$

where $w_i^{(k)}$ holds the entries in row i of $B^{(k)}$ that are already known, $z_i^{(k)}$ holds the remaining entries in the row, and $K_i^{(k)}$ and $U_i^{(k)}$ are the corresponding sub-matrices of $A_i^{(k)}$ (note that $K_i^{(k)}$ will in general not be square even if sufficient data is available). This formulation is illustrated in the following simple example.

Example 2 Consider the 5×5 approximate Hessian matrix

$$B^{(k)} = \begin{pmatrix} b_{11}^{(k)} & 0 & 0 & b_{14}^{(k)} & 0 \\ 0 & b_{22}^{(k)} & 0 & b_{24}^{(k)} & 0 \\ 0 & 0 & 0 & b_{34}^{(k)} & 0 \\ b_{41}^{(k)} & b_{42}^{(k)} & b_{43}^{(k)} & b_{44}^{(k)} & b_{45}^{(k)} \\ 0 & 0 & 0 & b_{54}^{(k)} & b_{55}^{(k)} \end{pmatrix} \text{ with } b_{14}^{(k)} = b_{41}^{(k)}, b_{24}^{(k)} = b_{42}^{(k)}, b_{34}^{(k)} = b_{43}^{(k)} \text{ and } b_{45}^{(k)} = b_{54}^{(k)}.$$

Assume the rows are computed in the natural order $1, 2, 3, 4, 5$. Row 4 has five entries and by symmetry $b_{41}^{(k)}, b_{42}^{(k)}, b_{43}^{(k)}$ are already known. The linear system (2.7) for row 4 is therefore

$$\underbrace{\begin{pmatrix} s_4^{(k)} & s_5^{(k)} \\ s_4^{(k-1)} & s_5^{(k-1)} \end{pmatrix}}_{U_4^{(k)}} \underbrace{\begin{pmatrix} b_{44}^{(k)} \\ b_{45}^{(k)} \end{pmatrix}}_{z_4^{(k)}} = \underbrace{\begin{pmatrix} y_4^{(k)} \\ y_4^{(k-1)} \end{pmatrix}}_{c_4^{(k)}} - \underbrace{\begin{pmatrix} s_1^{(k)} & s_2^{(k)} & s_3^{(k)} \\ s_1^{(k-1)} & s_2^{(k-1)} & s_3^{(k-1)} \end{pmatrix}}_{K_4^{(k)}} \underbrace{\begin{pmatrix} b_{14}^{(k)} \\ b_{24}^{(k)} \\ b_{34}^{(k)} \end{pmatrix}}_{w_4^{(k)}}.$$

For this approach, the order in which the rows are computed is crucial. This can be seen from the following two Hessian sparsity patterns with arrow-head structures that are symmetric permutations of each other.

$$\begin{pmatrix} * & & & & * \\ & * & & & * \\ & & . & & \vdots \\ & & & * & * \\ * & * & \cdots & * & * \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} * & * & \cdots & * & * \\ * & * & & & \\ \vdots & & . & & \\ * & & & * & \\ * & & & & * \end{pmatrix}.$$

Assume the rows are processed in the order $1, 2, \ldots n$. For the matrix on the left, for each of the first $n-1$ rows, two entries—one on the diagonal and one in column n—must be computed. The last row has n entries but, by symmetry, all the off-diagonal entries have already been computed. Thus only the diagonal entry is unknown, and hence all the entries can be computed using two data pairs $\{s^{(k)}, y^{(k)}\}$ and $\{s^{(k-1)}, y^{(k-1)}\}$. This is in contrast with the second matrix, where the first row contains n entries. Computing these requires n data pairs $\{s^{(l)}, y^{(l)}\}_{l=k-n+1}^{k}$. Whilst this also requires n computations, the first matrix ordering is clearly much better since it only requires two past data pairs, i.e. past iterates, to be available.

Using (2.7), we can compute the sparse Hessian approximation $B^{(k)}$ as described in Algorithm 2.2, which employs the adjacency (or connectivity) graph $\mathcal{G}(B^{(k)})$. Recall that this is an undirected graph with n vertices and an edge (i, j) if and only if $b_{ij}^{(k)} \neq 0, i \neq j$. If there is an edge (i, j) then i and j are said to be neighbours. The degree of vertex i is the number of neighbours it has, which is equal to the number of off-diagonal entries in row i of $B^{(k)}$. Algorithm 2.2 aims to limit the number of data pairs required at each step. It starts by selecting a vertex v of minimum degree (ties are broken arbitrarily). This can be done using a counting [21, §2.4.6] or bucket [5] sort in $O(n) + O(n_e)$ operations and storage locations (n_e is the number of edges). Once chosen, v is removed from the vertex set and the degree of each neighbour of v is decreased by one (corresponding to removing all edges involving v from the edge set). The process is then repeated for the reduced graph (but a complete reordering of the remaining vertices should be avoided). We define the *current degree* of a vertex to be its degree in the reduced graph (initially, the current degree is equal to the degree).

Algorithm 2.2 Sparse Hessian approximation (row-wise dependent)

1: Compute the adjacency graph $\mathcal{G}(B^{(k)})$ and the (current) degree of each vertex.
2: **for** $i = 1, \ldots, n$ **do**
3: Select vertex v of current minimum degree; assign the corresponding row as row i.
4: Compute the nu_i unknown entries in row i by solving the linear system (2.7).
5: Remove v; decrement the current degree of each of its remaining neighbours by 1.
6: **end for**

While Algorithm 2.2 generally requires fewer floating-point operations than

Algorithm 2.1, it has two related disadvantages. The first is that the steps in Algorithm 2.1 may be performed in parallel (if the full sparse Hessian approximation $B^{(k)}$ is stored) while Algorithm 2.2 is largely sequential—in practice, vertices i and j of minimum current degree with non overlapping sparsity patterns ($\mathcal{I}_i^+ \cap \mathcal{I}_j^+ = \emptyset$) can be processed in parallel. The second more serious defect is that inaccurate estimates from earlier steps in Algorithm 2.2 can be magnified when solving (2.7), leading to error growth even for constant Hessian matrices ($H^{(k)} = H$ for all k). The observed error growth is usually gradual but relentless; the more times an inaccurate early value occurs in later rows, the worse the effect, and this is potentially particularly pernicious for matrices in which some of the rows have a large number of entries. This sparse-dense case is not a problem for Algorithm 2.1 because each row is computed independently.

2.2 Sparse-dense Hessian matrices

We now explore the case where some rows of the Hessian matrix have appreciably more entries than others. We refer to such rows as dense (even though the number of entries may be significantly less than n). We assume the approximate Hessian matrix $B^{(k)}$ has been symmetrically permuted to

$$B^{(k)} = \begin{pmatrix} B_{11}^{(k)} & B_{12}^{(k)} \\ B_{21}^{(k)} & B_{22}^{(k)} \end{pmatrix}, \quad \text{with} \quad B_{21}^{(k)} = (B_{12}^{(k)})^T, \tag{2.8}$$

where the blocks $B_{11}^{(k)}$ and $B_{22}^{(k)}$ are square symmetric matrices, the $n_1 < n$ rows of $(B_{11}^{(k)} \; B_{12}^{(k)})$ are sparse and the remaining $n_2 = n - n_1 \ll n_1$ rows of $(B_{21}^{(k)} \; B_{22}^{(k)})$ are classified as dense. Note that we do not explicitly perform this symmetric permutation, but define it here for ease of presentation of the subsequent algorithms. In practice, we simply need to determine the sets of sparse and dense row indices based on the row densities of $B^{(k)}$.

Combined and thresholding algorithms

An obvious approach is to compute the n_1 sparse rows of $B^{(k)}$ using (2.5) from Algorithm 2.1; set $B_{21}^{(k)}$ by symmetry and then use (2.7) from Algorithm 2.2 to compute the n_2 dense rows. This is outlined in the report [11], but on some examples it suffers from error growth (as discussed earlier for Algorithm 2.2).

An alternative strategy generalises Algorithm 2.2 by using (2.7) to compute the unknown entries in row i only if the number of all entries nz_i in row i is deemed too large to use (2.5). This is also outlined in [11], but again this can suffer from error growth on some examples as documented in the report.

Block parallel algorithm

A better approach that prevents potential growth is the following, which is almost the same as the naive combined algorithm above. The n_1 sparse rows $(B_{11}^{(k)} \; B_{12}^{(k)})$ are computed using

(2.5) from Algorithm 2.1; $B_{21}^{(k)}$ is set by symmetry and then crucially the small $n_2 \times n_2$ block $B_{22}^{(k)}$ is computed using a variant of (2.7). That is, consider rewriting (2.4) as

$$\sum_{j \in \mathcal{V}_i^{(k)}} b_{ij}^{(k)} s_j^{(l)} = y_i^{(l)} - \sum_{j \in \mathcal{L}_i^{(k)}} b_{ij}^{(k)} s_j^{(l)}, \quad l = k, k-1, \ldots, \tag{2.9}$$

where

$$\mathcal{L}_i^{(k)} := \{j : h_{ij}^{(k)} \neq 0 \text{ and } b_{ij}^{(k)} \in B_{21}^{(k)}\} \text{ and } \mathcal{V}_i^{(k)} := \{j : h_{ij}^{(k)} \neq 0\} \setminus \mathcal{L}_i^{(k)},$$

are the column indices of the known entries in $B_{21}^{(k)}$ and unknown entries, respectively, in row i of the Hessian. It follows that data from (at least) $nv_i := |\mathcal{V}_i^{(k)}|$ previous steps is required. The i-th componentwise equation (2.9) can be rewritten as the dense linear system

$$V_i^{(k)} z_i^{(k)} = c_i^{(k)} - L_i^{(k)} w_i^{(k)}, \tag{2.10}$$

where $w_i^{(k)}$ holds the entries in row i of $B^{(k)}$ that are already known (i.e. in $B_{21}^{(k)}$), $z_i^{(k)}$ holds the remaining entries in the row, and $L_i^{(k)}$ and $V_i^{(k)}$ are the corresponding sub-matrices of $A_i^{(k)}$. This formulation is illustrated in the following simple example (note that, as before, $L_i^{(k)}$ will in general not be square even if sufficient data is available).

Example 3 Let $n = 4$ and consider the approximate symmetric Hessian matrix

$$B^{(k)} = \begin{pmatrix} B_{11}^{(k)} & B_{12}^{(k)} \\ B_{21}^{(k)} & B_{22}^{(k)} \end{pmatrix} = \left(\begin{array}{cc|cc} b_{11}^{(k)} & 0 & b_{13}^{(k)} & b_{14}^{(k)} \\ 0 & 0 & b_{23}^{(k)} & b_{24}^{(k)} \\ \hline b_{31}^{(k)} & b_{32}^{(k)} & b_{33}^{(k)} & b_{34}^{(k)} \\ b_{41}^{(k)} & b_{42}^{(k)} & b_{43}^{(k)} & b_{44}^{(k)} \end{array}\right)$$

where the first two rows are considered sparse and the last two rows dense. The 2×2 linear system (2.7) for the dense row 3 is

$$\underbrace{\begin{pmatrix} s_3^{(k)} & s_4^{(k)} \\ s_3^{(k-1)} & s_4^{(k-1)} \end{pmatrix}}_{V_3^{(k)}} \underbrace{\begin{pmatrix} b_{33}^{(k)} \\ b_{34}^{(k)} \end{pmatrix}}_{z_3^{(k)}} = \underbrace{\begin{pmatrix} y_3^{(k)} \\ y_3^{(k-1)} \end{pmatrix}}_{c_3^{(k)}} - \underbrace{\begin{pmatrix} s_1^{(k)} & s_2^{(k)} \\ s_1^{(k-1)} & s_2^{(k-1)} \end{pmatrix}}_{L_3^{(k)}} \underbrace{\begin{pmatrix} b_{13}^{(k)} \\ b_{23}^{(k)} \end{pmatrix}}_{w_3^{(k)}},$$

and for the dense row 4 the 2×2 system (2.7) is

$$\underbrace{\begin{pmatrix} s_3^{(k)} & s_4^{(k)} \\ s_3^{(k-1)} & s_4^{(k-1)} \end{pmatrix}}_{V_3^{(k)}} \underbrace{\begin{pmatrix} b_{43}^{(k)} \\ b_{44}^{(k)} \end{pmatrix}}_{z_3^{(k)}} = \underbrace{\begin{pmatrix} y_4^{(k)} \\ y_4^{(k-1)} \end{pmatrix}}_{c_3^{(k)}} - \underbrace{\begin{pmatrix} s_1^{(k)} & s_2^{(k)} \\ s_1^{(k-1)} & s_2^{(k-1)} \end{pmatrix}}_{L_3^{(k)}} \underbrace{\begin{pmatrix} b_{14}^{(k)} \\ b_{24}^{(k)} \end{pmatrix}}_{w_3^{(k)}}.$$

The final value of entries (3,4) and (4,3) of $B^{(k)}$ is $(b_{34}^{(k)} + b_{43}^{(k)})/2$ (symmetrisation).

The complete algorithm is described in Algorithm 2.3. Here not only the sparse rows, but also the dense rows can be handled in parallel if the full sparse Hessian approximation is stored, i.e. both lower and upper triangles. In fact, a dense row can be computed as soon as all the entries coming from the sparse part are known (and thus it may not be necessary to wait until all the sparse rows have been computed before starting the dense rows, although we have not implemented this).

Algorithm 2.3 Sparse-dense Hessian approximation (block parallel)

1: **parallel for** $i = 1, \ldots, n_1$ **do**
2: Compute all the entries in row i of $(B_{11}^{(k)}\ B_{12}^{(k)})$ by solving the linear system (2.5).
3: **end parallel for**
4: Set $B_{21}^{(k)} := (B_{12}^{(k)})^T$.
5: **parallel for** $i = 1, \ldots, n_2$ **do**
6: Compute all the entries in row i of $B_{22}^{(k)}$ by solving the linear system (2.10).
7: **end parallel for**
8: Symmetrise $B^{(k)} := (B^{(k)} + (B^{(k)})^T)/2$.

2.3 Hessian matrices with variable row densities

Finally, we present an algorithm that can handle general sparse Hessian matrices with variable row densities. For clarity of exposition, we omit the superscript (k), and assume that the approximate Hessian matrix B at the k-th iteration has been symmetrically permuted to

$$B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1N} \\ B_{21} & B_{22} & \cdots & B_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ B_{N1} & B_{N2} & \cdots & B_{NN} \end{pmatrix}, \quad B_{12}^T = B_{21}, \text{etc.}, \tag{2.11}$$

for some $2 < N < n$, where the blocks B_{ij} are all square symmetric matrices with the row blocks ordered according to increasing row density, i.e. the rows in $(B_{11}\ B_{12}\ \cdots\ B_{1N})$ are the sparsest, followed by those in $(B_{21}\ B_{22}\ \cdots\ B_{2N})$, etc. with finally the rows in $(B_{N1}\ B_{N2}\ \cdots\ B_{NN})$ being the densest. We do not explicitly form this symmetric permutation, but we simply determine the sets of row indices belonging to each row block.

Recursive block parallel algorithm

Algorithm 2.3 can be extended to (2.11) by applying it recursively: after the first sparse row block is computed using (2.5) and the corresponding symmetric entries are populated, the dense row block becomes the next sparse row block and is computed using (2.10), where only the entries in the previously computed symmetric blocks are assumed known. The process is repeated until there are no more sparse rows. We illustrate this for two levels of recursion in the diagram below, where the superscript denotes recursion depth.

$$B = \begin{pmatrix} B_{11}^{[0]} & B_{12}^{[0]} \\ B_{21}^{[0]} & B_{22}^{[0]} \end{pmatrix} \implies B_{22}^{[0]} = \begin{pmatrix} B_{11}^{[1]} & B_{12}^{[1]} \\ B_{21}^{[1]} & B_{22}^{[1]} \end{pmatrix} \implies B_{22}^{[1]} = \begin{pmatrix} B_{11}^{[2]} & B_{12}^{[2]} \\ B_{21}^{[2]} & B_{22}^{[2]} \end{pmatrix}$$

where $B_{11}^{[0]} = B_{11}$, $B_{11}^{[1]} = B_{22}$ and $B_{11}^{[2]} = B_{33}$ in (2.11). As before, we assume at each level of recursion r that $n_1^{[r]}$ rows of $(B_{11}^{[r]}\ B_{12}^{[r]})$ are classified as sparse and the remaining $n_2^{[r]}$ rows of $(B_{21}^{[r]}\ B_{22}^{[r]})$ are classified as dense. This approach of course requires a threshold for the sparse/dense split in terms of the unknown entries per row nv_i on which to recurse,

and a natural choice is m the number of past iterates available. Since it is inefficient to solve very small dense linear systems, we also include a minimum unknown entries per row threshold $n_{\min}$. Finally, to prevent the algorithm from recursing excessively (which could hamper parallelism), we also require a maximum recursion depth $r_{\max}$. As before, all the row blocks can be handled in parallel if the full sparse Hessian approximation is stored. The complete algorithm is described in Algorithm 2.4. Note that if there is no recursion ($r = 0$) then Algorithm 2.4 is equivalent to Algorithm 2.3 above.

Algorithm 2.4 Sparse-dense Hessian approximation (recursive block parallel)

Require: $r_{\max}$ maximum recursion depth, $n_{\min}$ minimum entries per row to recurse on.
Find rows i with $nz_i \leq m$ and place them in $(B_{11}^{[0]}\ B_{12}^{[0]})$.
parallel for $i = 1, \ldots, n_1^{[0]}$ **do**
    Compute all the entries in row i of $(B_{11}^{[0]}\ B_{12}^{[0]})$ by solving the linear system (2.5).
end parallel for
Set $B_{21}^{[0]} := (B_{12}^{[0]})^T$.
Initialise recursion counter $r := 0$.
while $r < r_{\max}$ and there exist rows i with $n_{\min} \leq nv_i \leq m$ **do**
    Increment $r := r + 1$
    Find rows i with $n_{\min} \leq nv_i \leq m$ and place them in $(B_{11}^{[r]}\ B_{12}^{[r]})$.
    parallel for $i = 1, \ldots, n_1^{[r]}$ **do**
        Compute all entries in row i of $(B_{11}^{[r]}\ B_{12}^{[r]})$ by solving the system (2.10) where
        only the entries in previously computed symmetric blocks are assumed known.
    end parallel for
    Set $B_{21}^{[r]} := (B_{12}^{[r]})^T$.
end while
parallel for $i = 1, \ldots, n_2^{[r]}$ **do**
    Compute all entries in row i of the remaining $B_{22}^{[r]}$ by solving the system (2.10)
    where only entries in previously computed symmetric blocks are assumed known.
end parallel for
Symmetrise $B := (B + B^T)/2$.

2.4 Implementation details

In each of the four algorithms described above, we need to solve a dense linear system of equations for each row i of the approximate Hessian B at each iteration (here we omit the iteration superscript (k) for clarity). This dense system is either of the form (2.5), i.e.

$$A_i z_i = c_i,$$

or of the form (2.7), i.e.

$$U_i z_i = c_i - K_i w_i,$$

or of the form (2.10), i.e.

$$V_i z_i = c_i - L_i w_i.$$

Thus generically, we can write each of these dense linear system in the form

$$Az = c, \tag{2.12}$$

for $A \in \mathbb{R}^{m_s \times n_s}$ where we solve for the unknowns $z \in \mathbb{R}^{n_s}$ in a particular Hessian row.

Dealing with insufficient data

Ideally, we would have sufficient past data available so that $m \geq m_s$ and the matrix A in (2.12), made up from the m_s most recent $\{s^{(l)}\}_{l=k-m_s+1}^{k}$, would be non-singular. But clearly this may not be the case. Firstly, in the early stages of the optimization algorithm, there may simply not be enough data; this will certainly be the case if $m < m_s$. Secondly, A formed as above may be singular (or close to singular) and in this case either again there will not be enough data to determine z uniquely or, if the objective function f is not quadratic, the gradient data c, made up from the m_s most recent $\{y^{(l)}\}_{l=k-m_s+1}^{k}$, may itself be inconsistent. In such cases, one possible remedy is simply to assign certain components of z to zero, and solve for the remainder (for example, entries far from the diagonal could be dropped). However, this is relatively arbitrary and a better strategy is to find the smallest z consistent with the data by solving the constrained least-squares problem

$$\min_{z \in \mathbb{R}^{n_s}} \|z\|_2 \ \text{ subject to } \ Az = c,$$

using, for example, the singular-value decomposition of A. When the latest data is inconsistent, rather than trying to find earlier data to exchange, we can add earlier data into A and c and then solve a weighted least-squares problem

$$\min_{z \in \mathbb{R}^{n_s}} \|W(Az - c)\|_2,$$

where the diagonal weighting matrix W favours the latest data. Once again, a singular-value decomposition of WA is suitable. However, for simplicity our preference in this paper is to simply find the least-squares solution to $Az = c$ of minimum ℓ_2-norm, i.e.

$$\min_{z \in \mathbb{R}^{n_s}} \|Az - c\|_2,$$

from the singular-value decomposition of A; this is used in all of our implementations.

Solving the linear system

In both the under- and over-determined cases, we compute the compact singular-value decomposition $A = U\Sigma V^T \in \mathbb{R}^{m_s \times n_s}$, where the columns of $U \in \mathbb{R}^{m_s \times r_s}$ and $V \in \mathbb{R}^{n_s \times r_s}$ are orthogonal, $\Sigma \in \mathbb{R}^{r_s \times r_s}$ is non-singular and diagonal, and r_s is the rank of A. We then find the required solution $z = V\Sigma^{-1}U^T c$ using `gelsd`, the SVD divide-and-conquer

algorithm from LAPACK [1]. It is also possible to use a faster but potentially less stable variant based on a QR factorization of A with interchanges using LAPACK `gelsy`, as well as a faster-still LU-based approach when A is square and non-singular using LAPACK `gesv`, but we have found that the resulting dense linear systems are small enough that these offer little advantage (the largest linear system solved by Algorithm 2.3 or 2.4 in the tests is of size 95×94). For numerical stability it may be desirable to add extra data (if available) when solving the linear system and we include such an option in our implementations.

3 Numerical experiments

We now examine how the proposed algorithms perform in practice. To do so, we consider the subset of Hessian matrices from the CUTEst [13] collection that was used in [9] and is listed in Table 3.1. Our experiments are performed on either a single processor core

identifier	n	$nnz(H)$	n_{null}	$nnz(row)$
BQPGAUSS	2,003	9,298	0	552
CURLY30	10,000	309,535	0	61
DRCAV1LQ	4,489	87,635	12	41
JIMACK	3,549	118,824	0	81
NCVXBQP1	50,000	199,984	0	9
SINQUAD	5,000	9,999	0	5,000
SPARSINE	5,000	79,554	0	56
SPARSQUR	10,000	159,494	0	56
WALL100	149,624	1,446,475	0	42
CAR2	5,999	50,964	0	5,999
GASOIL	10,403	8,606	6,998	1,602
LUKVLE12	9,997	22,492	0	2,502
MSQRTA	1,024	33,264	0	64
ORTHREGE	7,506	17,509	2	2,504
TWIRIMD1	1,247	42,197	0	660
YATP1SQ	123,200	368,550	0	352

Table 3.1: CUTEst test problems. The problems in the top (respectively, bottom) half of the table are unconstrained (respectively, constrained). The columns report the CUTEst identifier, the dimension n of H, the number $nnz(H)$ of nonzeros in the lower triangular part of H, the number n_{null} of null rows in H, and the largest number $nnz(row)$ of entries in a row of H.

or 28 processor cores of a dedicated single node on the STFC SCARF cluster, comprising 32 AMD Epyc 7502 CPUs clocked at 2.5GHz with 256 GB of RAM. The algorithms from Section 2 have been implemented in the Fortran 2018 package `SHA` as part of the GALAHAD library [12]. All codes are compiled in double precision using GNU Fortran 13.2 with O2 optimization, znver2 processor architecture, and OpenMP parallelism for the parallel codes.

The necessary LAPACK routines are provided by OpenBLAS 0.3.24 compiled for the cluster node as provided by SCARF (and limited by configuration to 28 OpenMP threads).

In our numerical experiments, we seek to investigate the accuracy attained by the different algorithms under ideal circumstances. We therefore test whether they compute good approximations in the simple case in which the Hessian matrix is fixed, that is, $H(x^{(k)}) = H$ for all k. The method we use to generate our test Hessian matrices is described in Appendix A. Having generated a fixed Hessian matrix H, we randomly generate $s^{(l)} \in (-1, 1)$ and then compute $y^{(l)} = Hs^{(l)}$ for $l = 1, \ldots, m$. We vary the number of past iterates $m = 1, \ldots, 100$ to explore how well the algorithms cope in both the under and over determined cases. For numerical stability, we require 1 extra past iterate if available when solving the linear system. To verify the accuracy of the computed approximations $B = \{b_{ij}\}$, we assume H is known and compute both the maximum relative componentwise error

$$max\_rel\_err = \max_{(i,j)\in\mathcal{S}(H)} |b_{ij} - h_{ij}|/\max(1, |h_{ij}|), \tag{3.13}$$

as well as the median relative componentwise error

$$med\_rel\_err = \operatorname{med}_{(i,j)\in\mathcal{S}(H)} |b_{ij} - h_{ij}|/\max(1, |h_{ij}|), \tag{3.14}$$

where med denotes the median.

3.1 Results when treating all rows as sparse

We start by illustrating the potential shortcomings of the row-wise independent and row-wise dependent algorithms. In Table 3.2, we report the maximum and median relative errors when applying the serial Algorithms 2.1 and 2.2 to the test examples with $m = 100$ past iterates. For Algorithm 2.1, we see that while the median relative error is close to machine precision, the maximum relative error can be large for problems containing dense rows. This is to be expected as Algorithm 2.1 treats each row i independently and therefore requires as many past iterates m as there are entries nz_i in the row. We can see from Table 3.1 that a number of problems in our test set have rows with more than 100 entries and it is precisely on these problems that Algorithm 2.1 struggles. The situation is much worse for Algorithm 2.2, where there is relentless error growth for a significant number of problems. Again this is to be expected, since inaccurate estimates from earlier steps in Algorithm 2.2 are magnified when substituted into subsequent rows (as we have previously discussed). We will therefore not consider Algorithms 2.1 and 2.2 any further.

3.2 Results when allowing for rows of different densities

Next, we start by illustrating how the block parallel and recursive block parallel algorithms are able to remedy the shortcomings of the row-wise independent and row-wise dependent algorithms presented earlier, in the case where sufficient data is available. In Table 3.2, we report the maximum and median relative errors when applying Algorithms 2.3 and 2.4

identifier	Algorithm 2.1		Algorithm 2.2		Algorithms 2.3 and 2.4	
	max_rel_err	*med_rel_err*	*max_rel_err*	*med_rel_err*	*max_rel_err*	*med_rel_err*
BQPGAUSS	2.19E+02	1.42E-15	7.88E+14	2.89E-15	7.95E-12	1.71E-15
CURLY30	8.83E-10	9.69E-15	6.00E-06	9.29E-12	5.41E-11	5.56E-15
DRCAV1LQ	9.38E-07	1.08E-14	‡	‡	2.56E-09	7.02E-15
JIMACK	1.98E-09	2.57E-14	‡	‡	5.32E-10	1.62E-14
NCVXBQP1	1.41E-09	1.22E-15	‡	2.59E+01	3.15E-11	1.07E-15
SINQUAD	9.77E-01	2.30E-16	3.66E-11	2.30E-16	1.99E-11	2.17E-16
SPARSINE	3.38E-09	6.01E-14	‡	‡	6.13E-10	4.40E-14
SPARSQUR	3.24E-09	1.69E-14	‡	‡	7.63E-10	1.28E-14
WALL100	1.05E-07	7.77E-15	‡	1.77E-09	2.32E-09	5.16E-15
CAR2	4.46E-12	0.00	‡	‡	3.81E-14	0.00
GASOIL	1.67E-01	3.33E-16	7.56E-12	3.57E-16	8.84E-12	2.22E-16
LUKVLE12	9.55E-01	6.33E-16	3.22E-09	1.63E-15	4.48E-13	6.66E-16
MSQRTA	3.69E-12	4.66E-15	‡	‡	9.47E-13	2.66E-15
ORTHREGE	4.12	4.76E-16	6.42E+03	4.76E-16	1.25E-12	6.05E-16
TWIRIMD1	1.15	4.33E-15	‡	4.59E-14	2.87E-12	2.60E-15
YATP1SQ	2.10	9.44E-16	2.41E-09	9.44E-16	1.36E-11	9.17E-16

Table 3.2: The maximum and median relative error when applying Algorithms 2.1, 2.2, 2.3 and 2.4 to the test problems with $m = 100$ past iterates. ‡ indicates error exceeds 10^{15}.

in parallel on 28 processor cores to the test problems with $m = 100$ past iterates. For Algorithm 2.4 we set the maximum recursion depth $r_{\max} = 25$ and the minimum entries per row to recurse on $n_{\min} = 10$. For both algorithms we see that the median relative error is close to machine precision and the maximum relative error is acceptably small, even for problems containing dense rows — in contrast to the row-wise independent and row-wise dependent algorithms presented earlier. In fact, in this setting where sufficient past iterates are available, the errors for both Algorithms 2.3 and 2.4 are identical, but this is not always the case as we shall shortly illustrate in the insufficient data regime.

Let us now turn our attention to runtime and illustrate the benefits that parallelism brings to Algorithms 2.3 and 2.4. Note that whilst Algorithm 2.1 is of course also trivially parallelisable, we do not consider it here since it can require an excessive number of past iterates as we have illustrated earlier in Section 3.1. In Table 3.3, we report the runtime in seconds for the serial (single core) and parallel (28 core in this case) variants of Algorithms 2.3 and 2.4. Runtime for each algorithm is measured in seconds using the Fortran `system_clock` routine. We can see from Table 3.3 that Algorithms 2.3 and 2.4 both achieve a significant parallel speedup on problems that are not already fast to solve in serial (i.e. have runtimes less than 0.1 seconds), meaning that in parallel we are able to approximate the Hessian for all examples in less than about 0.5 seconds.

identifier	Algorithm 2.3			Algorithm 2.4		
	serial	*parallel*	*speedup*	*serial*	*parallel*	*speedup*
BQPGAUSS	0.06	0.01	4.00	0.05	0.01	3.50
CURLY30	4.55	0.34	13.23	4.63	0.36	12.68
DRCAV1LQ	0.84	0.08	11.04	0.83	0.10	8.18
JIMACK	1.98	0.19	10.65	1.98	0.21	9.34
NCVXBQP1	0.53	0.05	10.06	0.53	0.05	10.23
SINQUAD	0.01	0.01	2.17	0.01	0.01	1.71
SPARSINE	0.70	0.07	9.94	0.70	0.07	9.67
SPARSQUR	1.40	0.12	11.38	1.40	0.12	11.29
WALL100	8.23	0.44	18.88	8.18	0.52	15.88
CAR2	0.19	0.02	8.22	0.19	0.03	6.37
GASOIL	0.01	0.01	2.00	0.01	0.01	2.17
LUKVLE12	0.04	0.01	4.78	0.04	0.01	4.78
MSQRTA	0.51	0.06	8.10	0.51	0.06	9.00
ORTHREGE	0.02	0.01	3.83	0.02	0.01	1.77
TWIRIMD1	0.60	0.09	6.95	0.60	0.09	6.60
YATP1SQ	0.49	0.07	7.17	0.49	0.06	8.22

Table 3.3: The runtime (in seconds) when applying Algorithms 2.3 and 2.4 in serial (single core) and parallel (28 cores) to the test problems with $m = 100$ past iterates, as well as the actual parallel speedup achieved. For Algorithm 2.4 we set the maximum recursion depth $r_{\max} = 25$ and the minimum entries per row to recurse on $n_{\min} = 10$.

3.3 Results in the low accumulated past data regime

We now turn our attention to the case where we have a low number of past data pairs, i.e. when the number of past iterates m is small. To this end, we will examine the behaviour of Algorithms 2.3 and 2.4 as m varies from 1 to 100.

Firstly, we consider the block parallel algorithm. In Figures 3.1 and 3.2, we report the maximum and median relative errors, respectively, when applying Algorithm 2.3 to the unconstrained (left) and constrained (right) test problems as the number of past iterates m varies from 1 to 100. Corresponding runtimes for Algorithm 2.3 are given in Figure 3.3. We see that when there is insufficient past data (m is too small) there is little the algorithm can do to get an accurate Hessian approximation, however once enough data pairs become available (m is sufficiently large) there is a sharp transition to an accurate approximation. This is most clearly seen for the CURLY30 test problem which has a banded Hessian with a total bandwidth of 61. For this problem there is a sharp transition after $m = 61$ for both the maximum and median relative error as expected. It is also interesting to see that a large maximum relative error is not necessarily indicative of a poor Hessian approximation. For example, the BQPGAUSS test problem has high maximum relative error until $m = 92$ but a low median relative error from $m = 11$ onwards. This suggests that for this problem the Hessian approximation is mostly very good from $m = 11$ onwards, aside from some

outlier Hessian entries that require more data pairs. Looking at the runtime in Figure 3.3, we see that the times are generally reasonable (under one second). However, we note the presence of large runtime spikes for JIMACK and WALL100 as the algorithm does not fully utilise the variable row densities present in these problems, we shall return to this issue in more detail shortly when we discuss the recursive block parallel algorithm.

Secondly, let us consider the recursive block parallel algorithm. We focus here on the test problems for which the recursion in Algorithm 2.4 shows a clear benefit, so that we can compare directly with the non-recursive Algorithm 2.3 (for completeness, we report the full numerical results for Algorithm 2.4 in Appendix B). In Figure 3.4, we report the maximum (left) and median (right) relative errors when applying Algorithms 2.3 (dashed line) and 2.4 (solid line) to the test problems for which recursion shows a clear benefit as the number of past iterates m varies from 1 to 100. Corresponding runtimes for Algorithm 2.4 are given in Figure 3.5. We see that, once again, in the insufficient past data case (m too small) it is not possible for the algorithm to achieve an accurate Hessian approximation, however once enough data pairs become available (m sufficiently large) Algorithm 2.4 outperforms Algorithm 2.3 on several problems. For example, for problem BQPGAUSS, Algorithm 2.3 requires $m = 92$ past iterates to achieve a maximum relative error around 10^{-11} whereas Algorithm 2.4 requires only $m = 39$ past iterates to achieve the same maximum relative error. Similarly, for problem WALL100, Algorithm 2.3 requires $m = 42$ past iterates to achieve a maximum relative error around 10^{-8} whereas Algorithm 2.4 requires only $m = 21$ past iterates. Much the same behaviour can be observed for the median relative error: for example, for problem JIMACK, Algorithm 2.3 requires $m = 81$ past iterates to achieve a median relative error around 10^{-14} whereas Algorithm 2.4 requires only $m = 55$ past iterates. Similarly, for the TWIRIMD1 test problem, Algorithm 2.3 requires $m = 94$ past iterates to achieve a maximum relative error around 10^{-14} whereas Algorithm 2.4 requires only $m = 64$ past iterates. These examples illustrate the primary aim of the recursion in Algorithm 2.4: to reduce the number of past iterates required to achieve an accurate Hessian approximation by reducing the number of unknown Hessian entries that need to be solved for in each row of the Hessian.

Finally, looking at the runtime in Figure 3.5, we can also clearly see the benefits of recursion as the large runtime spikes present for Algorithm 2.3 on the JIMACK and WALL100 test problems are now absent for the recursive Algorithm 2.4. For these two problems, Figure 3.6 shows the number of rows in each level of recursion within Algorithm 2.4. The number of rows in the first 'sparse' level $n_1^{[0]}$ (lines 2-4 in Algorithm 2.4) is depicted in blue, the number of rows in the last 'dense' level $n_2^{[r]}$ (lines 16-19 in Algorithm 2.4) is depicted in black, and the number of rows in the intermediate levels $n_1^{[r]}$ (lines 10-13 in Algorithm 2.4) are depicted using other colours. Note that for the non-recursive Algorithm 2.3 we only have the first and last level (blue and black). We can see from the figure that when m is sufficiently large (i.e. sufficient past iterates are available) no recursion is necessary and all rows are treated as 'sparse' (blue). As m decreases, we see that recursion is required to reduce the number of unknown entries we need to

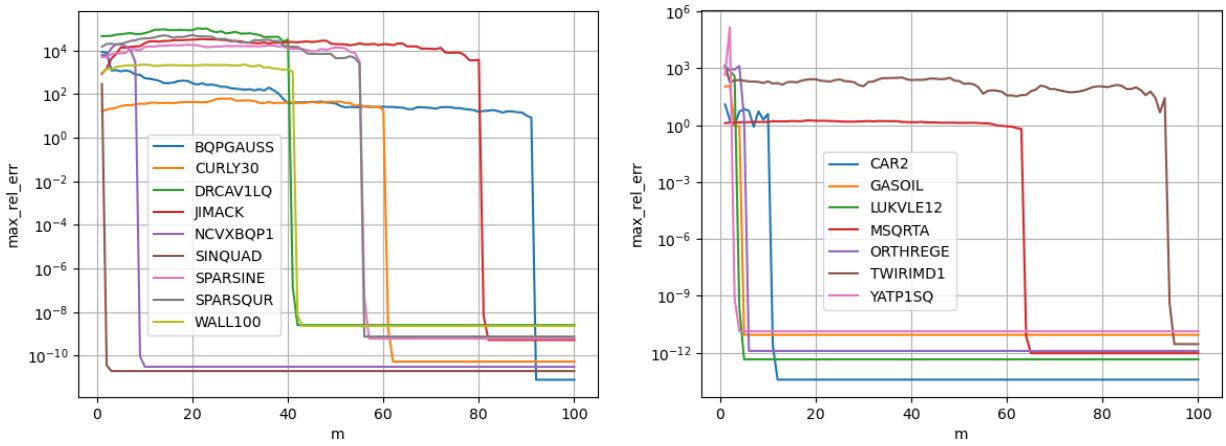

Figure 3.1: Maximum relative error when applying Algorithm 2.3 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.

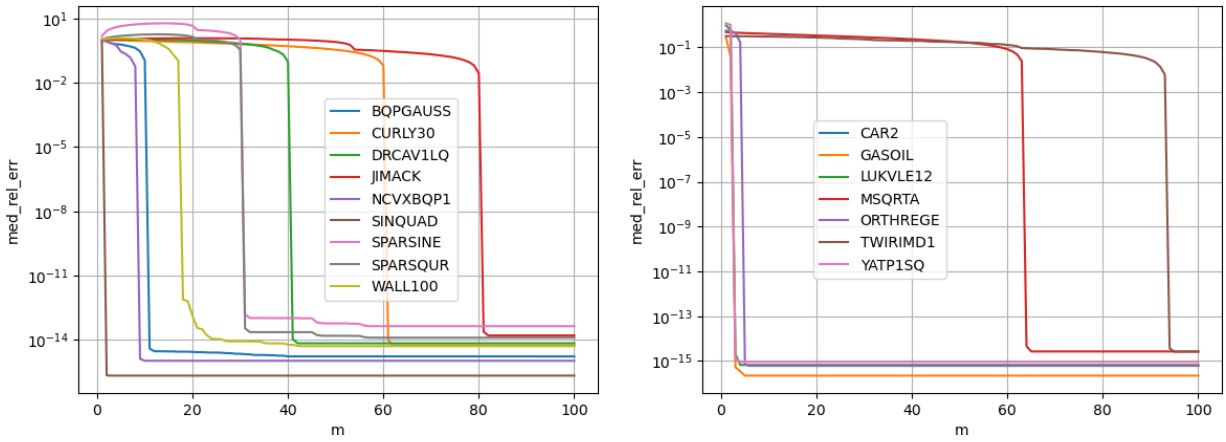

Figure 3.2: Median relative error when applying Algorithm 2.3 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.

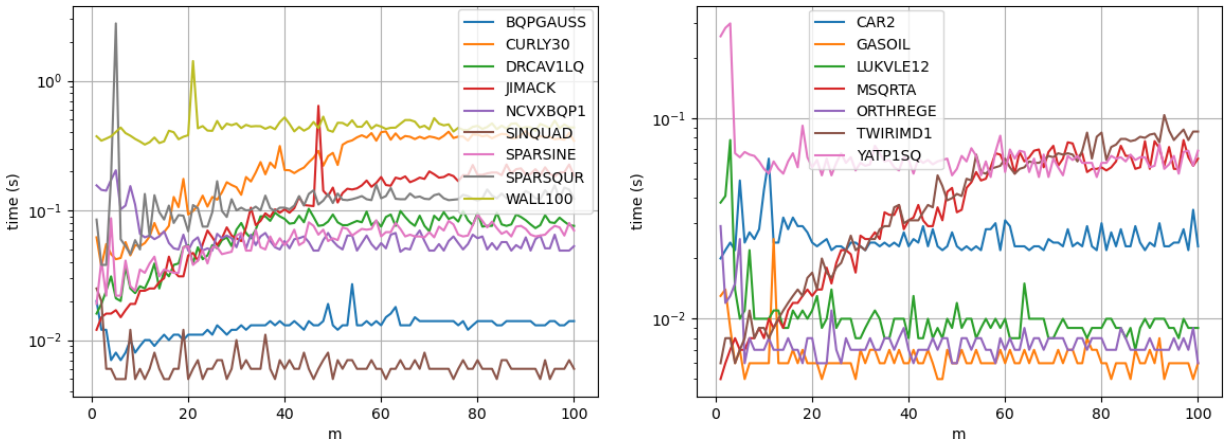

Figure 3.3: Runtime (in seconds) when applying Algorithm 2.3 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.

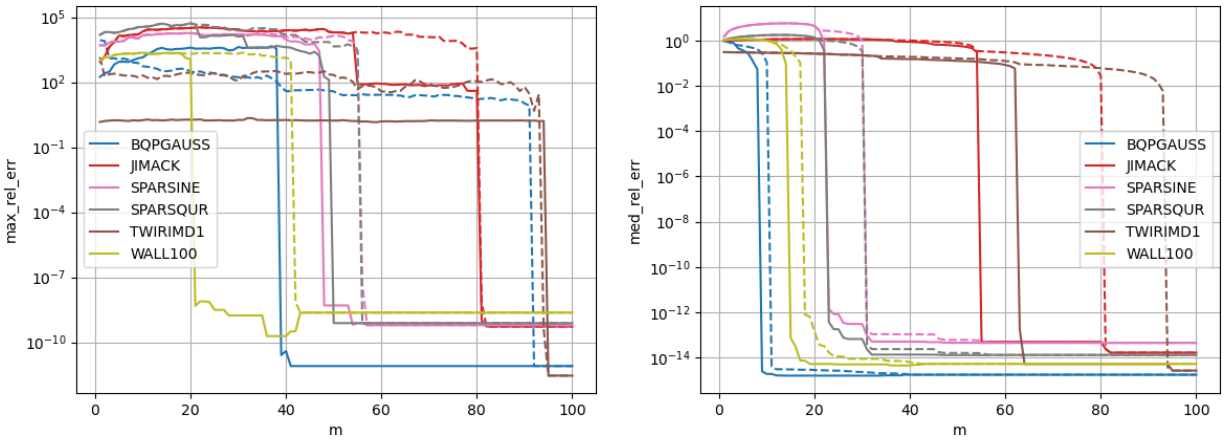

Figure 3.4: Maximum relative error (left) and median relative error (right) when applying Algorithm 2.3 (dashed line) and Algorithm 2.4 (solid line) on problems for which recursion shows a clear benefit as m varies from 1 to 100.

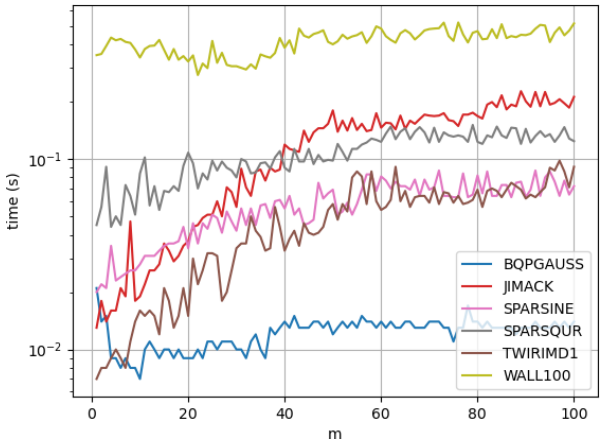

Figure 3.5: Runtime (in seconds) when applying Algorithm 2.4 on problems for which recursion shows a clear benefit as m varies from 1 to 100.

determine in each row of the Hessian, so that we stay within the given budget of m past iterates. Eventually, m becomes so small that there are not enough past iterates available to accurately estimate the Hessian and all rows are treated as 'dense' (black).

4 Conclusions and future work

We have presented a number of new methods for computing approximate Hessian matrices from optimization iterate and gradient differences when the sparsity structure is known in advance. The methods are promising in many cases, and have the potential to exploit reasonable parallel execution on a modest number of processors. Unlike some earlier methods, unwarranted growth in matrix entries due to rounding seems to be avoided in practice. The methods are available in the Fortran module `sha`, with a C interface, as part of the open-source GALAHAD library.

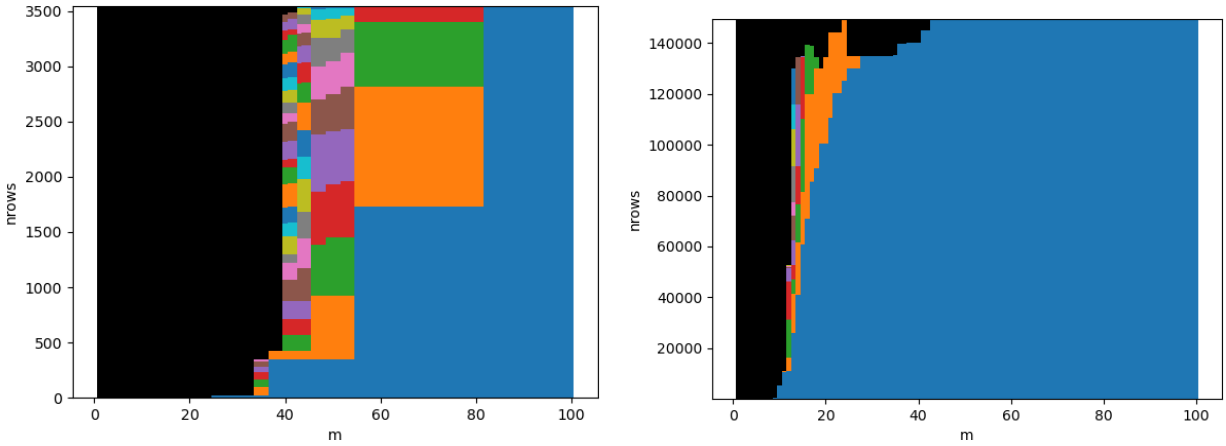

Figure 3.6: Number of rows in the block for each level of recursion (each colour) when applying Algorithm 2.4 on problem JIMACK (left) and WAL100 (right) as m varies from 1 to 100.

The next step is to see how these methods behave in practice when embedded inside actual optimization algorithms. In particular, we would like to know if such accumulated Hessian approximations obtained only from gradients lead to good second-order (Newton-like) algorithms. Or in other words, whether such Hessian approximations capture sufficient curvature information to still enable the higher-order convergence exhibited by second-order optimization algorithms. This is the subject of our current ongoing research.

Finally, although we have motivated our estimation strategies on m past observed optimization steps $\{s^{(l)}\}_{l=k-m+1}^{k}$ and gradient differences $\{y^{(l)}\}_{l=k-m+1}^{k}$ generated as an optimization algorithm proceeds, they apply equally to methods that judiciously choose steps $s^{(l)}$ and compute approximate $y^{(l)} = (g(x^{(k)} + \Delta s^{(l)}) - g(x^{(k)}))/\Delta$ at a particular point $x^{(k)}$, as is typical of the sparse finite-difference schemes [2, 20] we have mentioned in the introduction. The application of our estimation strategies to such methods is an interesting avenue for future research.

Acknowledgments

We are grateful to Coralia Cartis, Lukas Mackinder, Jared Tanner and Philippe Toint for earlier stimulating discussions. In particular, in 2016, Lukas Mackinder submitted a dissertation for a Master of Science degree at the University of Oxford related to the initial ideas behind this work; the dissertation is not publicly available.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, 1999.

[2] T. F. Coleman and J. J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Programming*, 28(3):243–270, 1984.

[3] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. In R. Glowinski and A. Lichnewsky, editors, *Computing Methods in Applied Sciences and Engineering*, pages 42–51. SIAM, 1990.

[4] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Numerical experiments with the LANCELOT package (Release A) for large-scale nonlinear optimization. *Math. Programming*, 73(1, Ser. A):73–110, 1996.

[5] E. Corwin and A. Logar. Sorting in linear time – variations on the bucket sort. *Journal of Computing Sciences in Colleges*, 20(1):197–202, 2004.

[6] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall Series in Computational Mathematics. Prentice Hall, Englewood Cliffs, NJ, 1983.

[7] R. Fletcher. An optimal positive definite update for sparse Hessian matrices. *SIAM J. Optim.*, 5(1):192–218, 1995.

[8] R. Fletcher, A. Grothey, and S. Leyffer. Computing sparse Hessian and Jacobian approximations with optimal hereditary properties. In A.R. Conn, L.T. Biegler, T.F. Coleman, and F.N. Santosa, editors, *Large-scale Optimization with Applications, Part II: Optimal Design and Control,*, volume 93 of *IMA Vol. Math. Appl.*, pages 37–52. Springer, New York, 1997.

[9] J. M. Fowkes, J. A. Scott, and N. I. M. Gould. Approximating sparse Hessian matrices using large-scale linear least squares. *Numerical Algorithms*, pages 1–24, 2023.

[10] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Computing forward-difference intervals for numerical optimization. *SIAM J. Sci. Statist. Comput.*, 4(2):310–321, 1983.

[11] N. I. M. Gould. Computing useful sparse Hessian approximations satisfying componentwise secant equations I: using a known sparsity pattern. Working Note RAL 2013-1, STFC-Rutherford Appleton Laboratory, Oxfordshire, UK, 2013.

[12] N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Transactions on Mathematical Software (TOMS)*, 29(4):353–372, 2003.

[13] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational optimization and applications*, 60:545–557, 2015.

[14] A. Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M.J.D. Powell, editor, *Nonlinear optimization, 1981*, NATO Conf. Ser. II: Systems Sci., pages 301–312. Academic Press, London, 1982.

[15] A. Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numer. Math.*, 39(1):119–137, 1982.

[16] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, second edition, 2008.

[17] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large-scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989.

[18] J. Nocedal. Updating quasi-Newton matrices with limited storage. *Math. Comp.*, 35(151):773–782, 1980.

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

[20] M. J. D. Powell and Ph. L. Toint. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal.*, 16(6):1060–1074, 1979.

[21] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Technical report, Report R-232, Digital Computer Laboratory, Massachusetts Institute of Technology, USA, 1954.

[22] D. C. Sorensen. An example concerning quasi-Newton estimation of a sparse Hessian. *ACM SIGNUM Newsletter*, 16(2):8–10, 1981.

[23] Ph. L. Toint. On sparse and symmetric matrix updating subject to a linear equation. *Math. Comp.*, 31(140):954–961, 1977.

[24] Ph. L. Toint. Some numerical results using a sparse matrix updating formula in unconstrained optimization. *Math. Comp.*, 32(143):839–851, 1978.

A Generation of Hessian matrices using CUTEst

Here we describe how the fixed Hessians H for unconstrained and constrained CUTEst problems used in the numerical experiments in Section 3 are generated.

For each unconstrained CUTEst test example we evaluate its Hessian matrix $H^{cutest}(x)$ at a point x^{pert} that is a random perturbation of the CUTEst starting point x^{start} and set $H = H^{cutest}(x^{pert})$. Specifically, if x_i^{start} $(1 \leq i \leq n)$ is the initial value for component i of

x^{start}, with lower and upper bounds x_i^l and x_i^u, then

$$x_i^{pert} = \begin{cases} x_i^l & \text{if} \quad x_i^l = x_i^u, \\ x_i^l + \rho \min(x_i^u - x_i^l, 1) & \text{if} \quad x_i^{start} \leq x_i^l, \\ x_i^u - \rho \min(x_i^u - x_i^l, 1) & \text{if} \quad x_i^{start} \geq x_i^u, \\ x_i^{start} + \rho \min(x_i^u - x_i^{start}, 1) & \text{otherwise.} \end{cases}$$

Here $\rho \in (0, 1)$ is the pseudo random number returned by the call `rand(seed,.true.,rho)`, where `rand` is from the GALAHAD library [12] and the default `seed` is used.

For the constrained examples, we evaluate the Hessian of the Lagrangian matrix $H_L^{cutest}(x, \mu)$ at a random perturbation x^{pert} of x^{start} (as above) and randomly generated Lagrange multipliers $\mu_q^{rand} \in (-1, 1)$ $(1 \leq q \leq n_c)$, with component i of μ^{rand} returned by `rand(seed,.false.,mu(i))`. We then set $H = H_L^{cutest}(x^{pert}, \mu^{rand})$.

B Complete numerical results for Algorithm 2.4

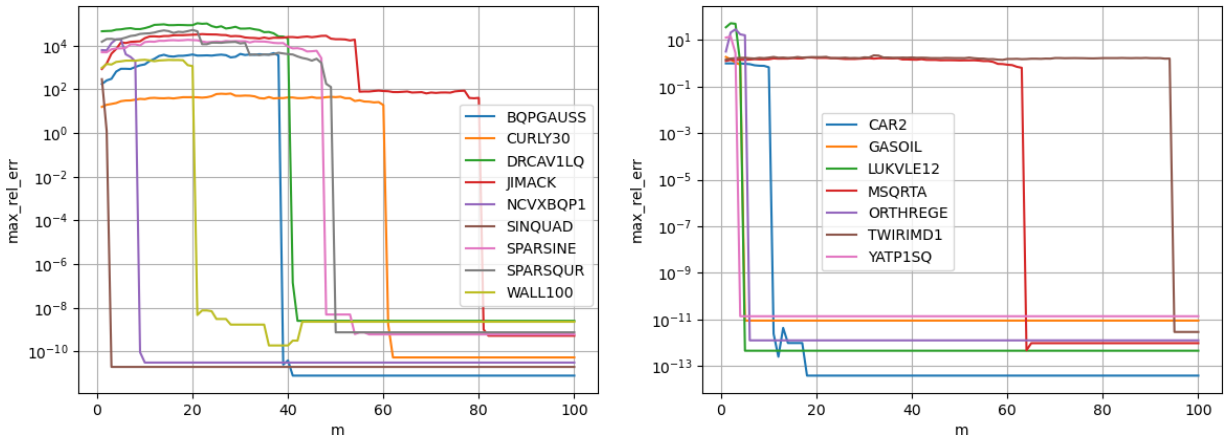

Figure A.1: Maximum relative error when applying Algorithm 2.4 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.

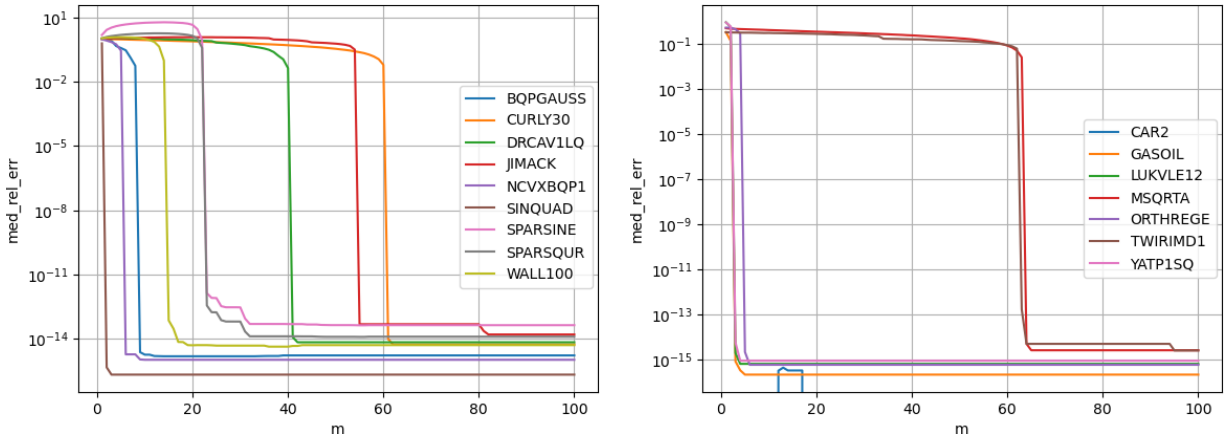

Figure A.2: Median relative error when applying Algorithm 2.4 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.

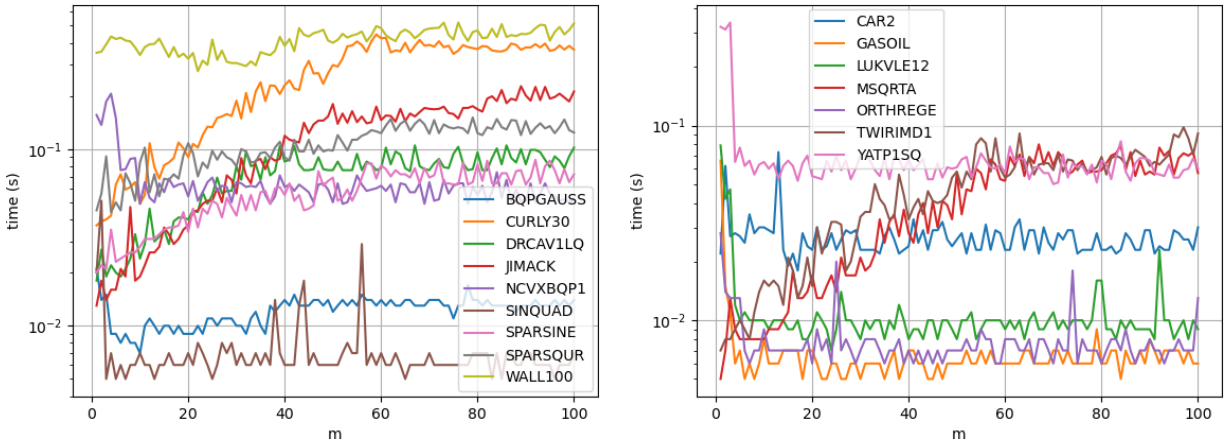

Figure A.3: Runtime (in seconds) when applying Algorithm 2.4 on the unconstrained (left) and constrained (right) examples as m varies from 1 to 100.