LIMITED-MEMORY REDUCED-HESSIAN METHODS FOR LARGE-SCALE UNCONSTRAINED OPTIMIZATION

PHILIP E. GILL* AND MICHAEL W. LEONARD†

Abstract. Limited-memory BFGS quasi-Newton methods approximate the Hessian matrix of second derivatives by the sum of a diagonal matrix and a fixed number of rank-one matrices. These methods are particularly effective for large problems in which the approximate Hessian cannot be stored explicitly.

It can be shown that the conventional BFGS method accumulates approximate curvature in a sequence of expanding subspaces. This allows an approximate Hessian to be represented using a smaller *reduced* matrix that increases in dimension at each iteration. When the number of variables is large, this feature may be used to define *limited-memory* reduced-Hessian methods in which the dimension of the reduced Hessian is limited to save storage. Limited-memory reduced-Hessian methods have the benefit of requiring half the storage of conventional limited-memory methods.

In this paper, we propose a particular reduced-Hessian method with substantial computational advantages compared to previous reduced-Hessian methods. Numerical results from a set of unconstrained problems in the CUTE test collection indicate that our implementation is competitive with the limited-memory codes L-BFGS and L-BFGS-B.

Key words. Unconstrained optimization, quasi-Newton methods, BFGS method, reduced-Hessian methods, conjugate-direction methods

AMS subject classifications. 65K05, 90C30

1. Introduction.

BFGS quasi-Newton methods have proved reliable and efficient for the unconstrained minimization of a smooth nonlinear function $f : \mathbb{R}^n \to \mathbb{R}$. However, the need to store an $n \times n$ approximate Hessian has limited their application to problems with a small-to-moderate number of variables (say, less than 500). For larger n it is necessary to use methods that do not require the storage of a full $n \times n$ matrix. Sparse quasi-Newton updates can be applied if the Hessian has a significant number of zero entries (see, e.g., Powell and Toint [29], Fletcher [10]). However, if the Hessian is dense, as is often the case for certain subproblems arising in nonlinearly constrained optimization, other methods must be used. Such methods include conjugate-gradient methods, limited-memory quasi-Newton methods, and limited-memory reduced-Hessian quasi-Newton methods.

Conjugate-gradient methods require storage for only a few n-vectors (see, e.g., Gill, Murray, and Wright [17, pp. 144–150]). These methods can be equivalent to the BFGS quasi-Newton method on a quadratic function, but they are generally acknowledged to be less robust on general nonlinear problems (see Gill and Murray [14] for some numerical comparisons). Limited-memory quasi-Newton methods also require storage of few n-vectors but have a more explicit relationship with quasi-Newton methods. Limited-memory methods exploit the fact that the approximate Hessian (or its inverse) can be written as the sum of a diagonal matrix and a number of rank-one matrices. This allows the search direction to be calculated as a simple linear combination of the vectors that define each rank-one update. The idea of

*Department of Mathematics, University of California, San Diego, La Jolla, California 92093-0112, USA (pgill@ucsd.edu). Research supported by the National Science Foundation grants DMI-9424639, CCR-9896198, and DMS-9973276 and Office of Naval Research grant N00014-96-1-0274.

†Department of Mathematics, University of California, San Diego, La Jolla, California 92093-0112, USA (mleonard@na-net.ornl.gov). Research supported by National Science Foundation grant DMI-9424639.

a limited-memory method is to store a fixed number m ($m \ll n$) of pairs of update vectors and to discard older pairs as new ones are computed. These methods appeared in the early 1980s (see, e.g., Shanno [30] and Nocedal [26]), and they have now been developed to a considerable level of sophistication (see Byrd, Nocedal, and Schnabel [5] and Kaufman [19]). Limited-memory approximate Hessians may be used directly in a conventional quasi-Newton method, or they may be used as preconditioners for a nonlinear conjugate-gradient method (see, e.g., Buckley [2, 3], Gill, Murray, and Wright [17, pp. 151–152], Morales and Nocedal [22], and Nazareth [25]).

A different approach has been taken by Fenelon [8] and Siegel [33], who independently proposed methods in which the curvature is accumulated in a subspace spanned by a set of m independent vectors. These *reduced-Hessian* methods exploit the fact that quasi-Newton methods accumulate approximate curvature in a sequence of expanding subspaces (see Gill and Leonard [13]). Reduced-Hessian methods represent the approximate Hessian using a smaller *reduced* matrix that increases in dimension at each iteration. This reduced matrix incorporates curvature information that has been accumulated during earlier iterations and allows the search direction to be calculated from a linear system that is smaller than that used in conventional methods.

In this paper we propose the limited-memory method L-RHR, which may be viewed as a limited-memory variant of the reduced-Hessian method RHR of Gill and Leonard [13]. L-RHR has two features in common with the limited-memory method of Siegel [33]: a basis of search directions is maintained for the sequence of m-dimensional subspaces, and an implicit orthogonal decomposition is used to define an orthonormal basis for each subspace. However, L-RHR is different from Seigel's algorithm in several ways: (i) L-RHR updates the Cholesky factor of the reduced Hessian instead of updating an explicit reduced inverse Hessian; (ii) the formulation of L-RHR as a modification of RHR allows the application of *Hessian reinitialization*, which is shown to greatly enhance performance on large problems; and (iii) L-RHR employs *selective basis reorthogonalization* to improve robustness for moderate values of the subspace dimension. Property (i) implies that in exact arithmetic, even if implemented without Hessian reinitialization and selective reorthogonalization, the L-RHR iterates are different from those of Siegel's method (see section 3.6). Properties (i)–(iii) not only provide substantial improvements in efficiency compared to Siegel's method, but also make reduced-Hessian methods competitive with the state-of-the-art limited-memory method L-BFGS-B of Zhu et al. [34]. L-RHR requires the storage of an $n \times m$ matrix, two $m \times m$ nonsingular upper-triangular matrices, and a fixed number of n- and m-vectors. For a given m, this is approximately half the storage required for L-BFGS-B to represent essentially the same amount of second-derivative information. Moreover, L-RHR requires fewer floating-point operations per iteration, which results in smaller overall computation times on many problems.

The paper is organized as follows. In section 2 we briefly review various theoretical aspects of reduced-Hessian quasi-Newton methods, including the definition of Algorithm RHR, a reduced-Hessian method with Hessian reinitialization. Algorithm RHR provides the theoretical framework for the limited-memory algorithm L-RHR proposed in section 3. We give algorithms for maintaining both gradient- and search-direction subspace bases, and it is shown that L-RHR has the property of finite termination on a strictly convex quadratic function. To simplify the discussion, the algorithms of sections 2–3 are stated with the assumption that all computations are performed in exact arithmetic. The effects of rounding error and the use of reorthogonalization are discussed in sections 4.1, 4.2, and 4.3. Finally, section 5 includes some numerical re-

sults obtained when various limited-memory reduced-Hessian algorithms are applied to test problems from the CUTE test collection of Bongartz et al. [1]. It is shown that reinitialization and selective reorthogonalization (in conjunction with an explicit factorization for the subspace basis) give, respectively, significantly fewer function evaluations and increased robustness compared to Siegel's method. Section 5 also includes comparisons of L-RHR with two alternative implementations of the conventional limited-memory BFGS method.

Unless explicitly indicated otherwise, $\|\cdot\|$ denotes the vector two-norm or its subordinate matrix norm.

2. Motivation. The BFGS method generates a sequence of iterates $\{x_k\}$ such that $x_{k+1} = x_k + \alpha_k p_k$, where p_k is the search direction and α_k is a scalar step length. The search direction satisfies $H_k p_k = -\nabla f(x_k)$, where H_k is an approximate Hessian. The application of the BFGS update to H_k gives a matrix H_k' such that

$$H_k' = H_k - \frac{1}{\delta_k^T H_k \delta_k} H_k \delta_k \delta_k^T H_k + \frac{1}{\gamma_k^T \delta_k} \gamma_k \gamma_k^T, \tag{2.1}$$

where $\delta_k = x_{k+1} - x_k$, $g_k = \nabla f(x_k)$, and $\gamma_k = g_{k+1} - g_k$. A conventional BFGS method then defines $H_{k+1} = H_k'$ (another choice for H_{k+1} is discussed in section 2.2). If H_0 is symmetric and positive definite, and if α_k is such that the approximate curvature $\gamma_k^T \delta_k$ is positive, then H_k is symmetric positive definite for all $k \geq 0$. Conditions imposed on the step length by practical step-length algorithms can ensure both positivity of the approximate curvature and sufficient descent. This is the case, for example, for any α_k satisfying the Wolfe conditions

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \mu \alpha_k g_k^T p_k \quad \text{and} \quad g_{k+1}^T p_k \geq \eta g_k^T p_k, \tag{2.2}$$

where the constants μ and η are chosen so that $0 \leq \mu < \eta < 1$ and $\mu < \frac{1}{2}$.

The need to solve a linear system for p_k makes it convenient to use the upper-triangular Cholesky factor C_k such that $H_k = C_k^T C_k$. In this case, the Cholesky factor C_{k+1} of H_{k+1} is obtained from a rank-one change to C_k (see Dennis and Schnabel [7]). We omit the details of this procedure and simply write $C_{k+1} = \textbf{\textit{update}}(C_k, \delta_k, \gamma_k)$.

2.1. Reduced-Hessian methods. Reduced-Hessian methods provide an alternative way of implementing the BFGS method. Let $\mathcal{G}_k$ denote the subspace $\mathcal{G}_k = \text{span}\{g_0, g_1, \ldots, g_k\}$, and let $\mathcal{G}_k^\perp$ denote the orthogonal complement of $\mathcal{G}_k$ in $\mathbb{R}^n$. Reduced-Hessian methods are based on the following result (see, e.g., Fletcher and Powell [11], Fenelon [8], and Siegel [33]).

LEMMA 2.1. *Consider the BFGS method applied to a general nonlinear function. If $H_0 = \sigma I$ ($\sigma > 0$) and $H_k p_k = -g_k$, then $p_k \in \mathcal{G}_k$ for all k. Moreover, if $z \in \mathcal{G}_k$ and $w \in \mathcal{G}_k^\perp$, then $H_k z \in \mathcal{G}_k$ and $H_k w = \sigma w$.*

Let r_k denote $\dim(\mathcal{G}_k)$, and let B_k (B for "basis") denote an $n \times r_k$ matrix whose columns form a basis for $\mathcal{G}_k$. An orthonormal basis Z_k can be defined from the QR decomposition $B_k = Z_k T_k$, where T_k is a nonsingular upper-triangular matrix. Let the $n - r_k$ columns of W_k define an orthonormal basis for $\mathcal{G}_k^\perp$. If Q_k is the orthogonal matrix $Q_k = \begin{pmatrix} Z_k & W_k \end{pmatrix}$, then the transformation $x = Q_k x_Q$ defines a transformed approximate Hessian $Q_k^T H_k Q_k$ and a transformed gradient $Q_k^T g_k$. If $H_0 = \sigma I$ ($\sigma > 0$), it follows from (2.1) and Lemma 2.1 that the transformation induces a block-diagonal structure, with

$$Q_k^T H_k Q_k = \begin{pmatrix} Z_k^T H_k Z_k & 0 \\ 0 & \sigma I_{n-r_k} \end{pmatrix} \quad \text{and} \quad Q_k^T g_k = \begin{pmatrix} Z_k^T g_k \\ 0 \end{pmatrix}. \tag{2.3}$$

The positive-definite matrix $Z_k^T H_k Z_k$ is known as a reduced approximate Hessian (or just reduced Hessian). The vector $Z_k^T g_k$ is known as a reduced gradient.

If we write the equation for the search direction as $(Q_k^T H_k Q_k) Q_k^T p_k = -Q_k^T g_k$, it follows from (2.3) that

$$p_k = Z_k q_k, \text{ where } q_k \text{ satisfies } Z_k^T H_k Z_k q_k = -Z_k^T g_k. \tag{2.4}$$

If the Cholesky factorization $Z_k^T H_k Z_k = R_k^T R_k$ is known, q_k can be computed from the forward substitution $R_k^T d_k = -Z_k^T g_k$ and back-substitution $R_k q_k = d_k$. The practical benefit of this approach is that, if $k \ll n$, the matrices Z_k and R_k require much less storage than H_k.

There are a number of alternative choices for B_k (see Gill and Leonard [13, Theorem 2.3]). Both Fenelon and Siegel propose that B_k be formed from a linearly independent subset of $\{g_0, g_1, \ldots, g_k\}$. With this choice, the orthonormal basis can be accumulated columnwise as the iterations proceed using Gram–Schmidt orthogonalization (see, e.g., Golub and Van Loan [18, pp. 218–220]). During iteration k, the number of columns of Z_k either remains unchanged or increases by one, depending on the value of the scalar ρ_{k+1} such that $\rho_{k+1} = \|(I - Z_k Z_k^T) g_{k+1}\|$. If $\rho_{k+1} = 0$, the new gradient has no component outside range(Z_k) and g_{k+1} is said to be *rejected*. Thus, if $\rho_{k+1} = 0$, Z_k already provides a basis for $\mathcal{G}_{k+1}$ with $r_{k+1} = r_k$ and $Z_{k+1} = Z_k$. Otherwise, $r_{k+1} = r_k + 1$ and the gradient g_{k+1} is said to be *accepted*. In this case, Z_k gains a new column z_{k+1} defined by the identity $\rho_{k+1} z_{k+1} = (I - Z_k Z_k^T) g_{k+1}$. The calculation of z_{k+1} also provides the r_k-vector $u_k = Z_k^T g_{k+1}$ and the scalar $z_{k+1}^T g_{k+1}$ $(= \rho_{k+1})$, which are the components of the reduced gradient $Z_{k+1}^T g_{k+1}$ for the next iteration. For simplicity, we write $(Z_{k+1}, u_k, \rho_{k+1}, r_{k+1}) = \textbf{\textit{orthog}}(Z_k, g_{k+1}, r_k)$ in the algorithms that follow. This orthogonalization procedure requires approximately $2nr_k$ flops. Gram–Schmidt orthogonalization may be considered as an algorithm for computing the QR decomposition of B_k without storing T_k. Suppose that at the start of iteration k there exists a nonsingular T_k with $B_k = Z_k T_k$. If g_{k+1} is accepted, then

$$B_{k+1} = \begin{pmatrix} B_k & g_{k+1} \end{pmatrix} = \begin{pmatrix} Z_k & z_{k+1} \end{pmatrix} \begin{pmatrix} T_k & Z_k^T g_{k+1} \\ 0 & \rho_{k+1} \end{pmatrix} = Z_{k+1} T_{k+1}, \tag{2.5}$$

where the last equality defines T_{k+1}, which is nonsingular since $\rho_{k+1} \neq 0$. Otherwise, $T_{k+1} = T_k$.

Definition (2.4) of each search direction implies that $p_j \in \mathcal{G}_k$ for all $0 \leq j \leq k$. This leads naturally to another basis for $\mathcal{G}_k$ based on orthogonalizing the search directions $p_0, p_1, \ldots, p_k$. The next theorem implies that the columns of Z_k constitute an orthonormal basis for $\mathcal{P}_k$, the span of *all* search directions $\{p_0, p_1, \ldots, p_k\}$ (for a proof, see Gill and Leonard [13]).

THEOREM 2.2. *If $H_0 = \sigma I$ $(\sigma > 0)$, then the subspaces $\mathcal{G}_k$ and $\mathcal{P}_k$ generated by the gradients and search directions of the conventional BFGS method are identical.*

This result implies that Z_k can be generated from either gradients or search directions, a point that will be used to advantage in section 3.

Given Z_{k+1} and H_k, the calculation of the search direction for the next iteration requires the Cholesky factor of $Z_{k+1}^T H_{k+1} Z_{k+1}$.[1] This factor can be obtained from R_k in a two-step process without the need to know H_k. The first step, which is not

[1] As mentioned earlier, H_{k+1} is usually H_k', which is defined by (2.1) However, we will implicitly alter H_k' further, as described in section 2.2.

needed if g_{k+1} is rejected, is to compute the factor R'_k of $Z_{k+1}^T H_k Z_{k+1}$. This step involves adding a row and column to R_k to account for the new last column of Z_{k+1}. It follows from Lemma 2.1 and (2.3) that

$$Z_{k+1}^T H_k Z_{k+1} = \begin{pmatrix} Z_k^T H_k Z_k & Z_k^T H_k z_{k+1} \\ z_{k+1}^T H_k Z_k & z_{k+1}^T H_k z_{k+1} \end{pmatrix} = \begin{pmatrix} Z_k^T H_k Z_k & 0 \\ 0 & \sigma \end{pmatrix},$$

giving an expanded block-diagonal factor R'_k defined by

$$R'_k = \begin{cases} R_k & \text{if } r_{k+1} = r_k; \\ \begin{pmatrix} R_k & 0 \\ 0 & \sigma^{1/2} \end{pmatrix} & \text{if } r_{k+1} = r_k + 1. \end{cases} \tag{2.6}$$

The algorithm that defines R'_k from R_k will be denoted by ***expand*** for obvious reasons. This expansion also involves vectors $v_k = Z_k^T g_k$, $u_k = Z_k^T g_{k+1}$, and $q_k = Z_k^T p_k$, which are updated to give $v'_k = Z_{k+1}^T g_k$, $u'_k = Z_{k+1}^T g_{k+1}$, and $q'_k = Z_{k+1}^T p_k$. As both p_k and g_k lie in range(Z_k), if g_{k+1} is accepted, the vectors v'_k and q'_k are trivially defined from v_k and q_k by appending a zero component (see (2.3)). Similarly, the vector u'_k is formed from u_k and ρ_{k+1}. If g_{k+1} is rejected, $v'_k = v_k$, $u'_k = u_k$, and $q'_k = q_k$. In either case, v_{k+1} is equal to u'_k and need not be calculated at the start of iteration $k+1$ (see Algorithm 2.1 below).

The second step of the modification alters R'_k to reflect the BFGS update to H_k. This update gives a modified factor $R''_k = \textbf{\textit{update}}(R'_k, s_k, y_k)$, where $s_k = Z_{k+1}^T(x_{k+1} - x_k) = \alpha_k q'_k$, and $y_k = Z_{k+1}^T(g_{k+1} - g_k) = u'_k - v'_k$.

2.2. Reinitialization. The initial approximate Hessian can greatly influence the practical performance of quasi-Newton methods. The usual choice $H_0 = \sigma I$ ($\sigma > 0$) can result in many iterations and function evaluations—especially if the iterates tend toward a minimizer at which the Hessian of f is ill-conditioned (see, e.g., Powell [27] and Siegel [33]). This is sometimes associated with "stalling" of the iterates, a phenomenon that can greatly increase the overall cpu time for solution (or termination). The form of the transformed Hessian $Q_k^T H_k Q_k$ (see (2.3)) reveals the influence of H_0 on the approximate Hessian. In particular, the scale factor σ represents the approximate curvature along all directions in $\mathcal{G}_k^{\perp}$. However, in the reduced-Hessian formulation, this initial approximate curvature is not installed until the end of iteration k, when it is used in the ***expand*** procedure according to (2.6). Our idea is to replace σ whenever g_{k+1} is accepted with a value more representative of the approximated curvature. This has the effect of *reinitializing* the approximate curvature along z_{k+1} and is meant to alleviate inefficiencies resulting from poor choices of H_0. An estimate σ_k of the approximate curvature is maintained and updated as new curvature information is obtained. Some popular choices for σ_k are considered by Leonard [20, pp. 44–48]. In section 5 we discuss values that have been proposed for limited-memory methods.

The initial approximate curvature can be reinitialized by using some σ_{k+1} in place of σ_k in the ***expand*** procedure. Gill and Leonard [13] show that reinitialization can be done either before or after the ***expand***, but they recommend the latter because it results in a simpler convergence result. Here, the reinitialization is performed after ***update*** to be consistent with that article. The procedure ***reinitialize*** involves simply changing the trailing diagonal element of R''_k from $\sigma_k^{1/2}$ to $\sigma_{k+1}^{1/2}$ whenever g_{k+1} is accepted.

2.3. Summary. We conclude this section by defining a generic reduced-Hessian method that is the basis of the limited-memory method proposed in section 3. As described above, the reduced-Hessian method involves four main procedures: an *orthogonalize*, which determines Z_{k+1} using the Gram–Schmidt QR process; an *expand*, which increases the order of the reduced Hessian by one; an *update*, which applies a BFGS update directly to the reduced Hessian; and a *reinitialize*, which reinitializes the last diagonal of the reduced Hessian factor.

ALGORITHM 2.1. (RHR) REDUCED-HESSIAN METHOD WITH REINITIALIZATION.
Choose x_0 and σ_0 ($\sigma_0 > 0$);
$k = 0$; $r_0 = 1$; $g_0 = \nabla f(x_0)$;
$Z_0 = \begin{pmatrix} g_0/\|g_0\| \end{pmatrix}$; $R_0 = \begin{pmatrix} \sigma_0^{1/2} \end{pmatrix}$; $v_0 = \|g_0\|$;
while not converged **do**
  Solve $R_k^T d_k = -v_k$; $R_k q_k = d_k$;
  $p_k = Z_k q_k$;
  Find α_k satisfying the Wolfe conditions (2.2);
  $x_{k+1} = x_k + \alpha_k p_k$; $g_{k+1} = \nabla f(x_k + \alpha_k p_k)$;
  $(Z_{k+1}, u_k, \rho_{k+1}, r_{k+1}) = \boldsymbol{orthog}(Z_k, g_{k+1}, r_k)$;
  $(R_k', u_k', v_k', q_k') = \boldsymbol{expand}(R_k, u_k, v_k, q_k, \rho_{k+1}, \sigma_k)$;
  $s_k = \alpha_k q_k'$; $y_k = u_k' - v_k'$; $R_k'' = \boldsymbol{update}(R_k', s_k, y_k)$;
  Compute σ_{k+1}; $R_{k+1} = \boldsymbol{reinitialize}(R_k'', \sigma_{k+1})$;
  $v_{k+1} = u_k'$;
  $k = k + 1$;
end do

When no reinitialization is done, this algorithm generates the same iterates as the conventional BFGS method with $H_0 = \sigma_0 I$. In exact arithmetic, the methods differ only in the storage needed and the number of operations per iteration. It can be shown that both with and without reinitialization, the algorithm retains two important properties of the BFGS method: it has quadratic termination, and it converges globally and Q-superlinearly on strongly convex functions (see Gill and Leonard [13]).

Algorithm RHR implicitly defines a full-sized BFGS approximate Hessian H_k. Let Z_k and R_k be defined at the start of the kth iteration, and let $Q_k = \begin{pmatrix} Z_k & W_k \end{pmatrix}$ denote an orthogonal matrix whose first r_k columns are the columns of Z_k. The full-sized approximate Hessian is given by

$$H_k = Q_k \begin{pmatrix} R_k^T & 0 \\ 0 & \sigma_k^{1/2} I_{n-r_k} \end{pmatrix} \begin{pmatrix} R_k & 0 \\ 0 & \sigma_k^{1/2} I_{n-r_k} \end{pmatrix} Q_k^T. \tag{2.7}$$

Given R_k, Z_k, and any n-vector v, the identity

$$H_k v = Z_k R_k^T R_k Z_k^T v + \sigma_k (I - Z_k Z_k^T) v$$

implies that products $H_k v$ can be calculated. This allows the reduced-Hessian approach to be used in constrained optimization algorithms that use H_k as an operator via products of the form $H_k v$ (see, e.g., Gill, Murray, and Saunders [15]).

3. A limited-memory reduced-Hessian method. In this section we propose a limited-memory method that may be viewed as a reduced-Hessian method in which only the most recent curvature information is retained. As in Algorithm RHR, a

triangular factor of the reduced Hessian is updated and reinitialized at each iteration—the crucial difference is that the number of basis vectors (and hence the dimension of the reduced Hessian) is limited by a preassigned value m. For problems with many variables, a choice of $m \ll n$ gives significant savings in storage compared to conventional quasi-Newton methods.

A simple limited-memory version of RHR can be defined by discarding the oldest gradient when the storage limit is reached. However, algorithms based on this idea have proved to be inefficient in practice. One explanation of this inefficiency is that discarding the oldest gradient invalidates RHR's property of finite termination on a quadratic function (see Theorem 3.1 and the concluding remarks of section 3.7). There is considerable numerical evidence that quadratic termination is beneficial when minimizing general functions; see, e.g., Siegel [31] and Leonard [20]. The limited-memory method proposed here retains the property of finite termination by following Siegel's suggestion of using a basis of search directions rather than gradients. This strategy is sufficient to maintain quadratic termination when the oldest basis vector is discarded (see section 3.3).

An important feature of the method is that it is necessary to store and update the triangular factor T_k associated with the orthogonal factorization $B_k = Z_k T_k$. In practice, we store T_k and either Z_k or B_k.

3.1. The search-direction basis and its factorization. We start by describing how the orthogonal factorization is maintained as directions are added to the basis. Initially, this procedure is described in the context of building an m-dimensional basis before a search direction is discarded, where m is assumed to satisfy $m \geq 2$. The usual context is to add and remove a vector at every iteration. The procedure for removing a direction from the basis is described in section 3.2. To simplify the discussion, we assume that every gradient is accepted.

In order to allow for the fact that Z_k is used in the equations that define p_k, the gradient g_k is used as a temporary basis vector until it can be replaced by p_k. This implies that the kth iteration involves three basis matrices: B_k, B'_k, and B''_k. The starting basis is $B_k = (p_0 \quad \cdots \quad p_{k-1} \quad g_k)$. The matrix B'_k is obtained from B_k by replacing g_k by p_k as soon as it is computed, and B''_k is found by adding the accepted gradient to B'_k. The matrices B_k and B'_k differ by a single column, yet, by Theorem 2.2, their columns span the same subspace during the build process.

The procedure starts with $B_0 = \begin{pmatrix} g_0 \end{pmatrix}$, $T_0 = \begin{pmatrix} \|g_0\| \end{pmatrix}$, and $Z_0 = \begin{pmatrix} z_0 \end{pmatrix}$, where $z_0 = g_0/\|g_0\|$. Once p_0 is calculated, it is swapped into the basis to give $B'_0 = \begin{pmatrix} p_0 \end{pmatrix}$ and $T'_0 = \begin{pmatrix} \|p_0\| \end{pmatrix}$. After the line search, g_1 is accepted (by assumption) and we define

$$B''_0 = (p_0 \quad g_1), \quad T''_0 = \begin{pmatrix} \|p_0\| & u_0 \\ 0 & \rho_1 \end{pmatrix}, \quad \text{and} \quad Z'_0 = (z_0 \quad z_1)$$

(see (2.5) and recall that $u_k = Z_k^T g_{k+1}$). With our assumption that $m \geq 2$, no vector need be discarded and these matrices define B_1, T_1, and Z_1. The kth iteration ($1 \leq k \leq m-1$) proceeds in a similar way, with

$$B_k = (p_0 \quad \cdots \quad p_{k-1} \quad g_k), \quad T_k = (\underline{T}'_{k-1} \quad v_k), \quad \underline{T}'_{k-1} = \begin{pmatrix} T'_{k-1} \\ 0 \end{pmatrix},$$

and $Z_k = (z_0 \quad \cdots \quad z_{k-1} \quad z_k)$ (the form of the last column of T_k follows from the definition of v_k as $Z_k^T g_k$). Once p_k is computed, it is swapped with g_k in the basis

with no computation required, yielding

$$B'_k = (\, p_0 \quad \cdots \quad p_{k-1} \quad p_k \,) \quad \text{and} \quad T'_k = \begin{pmatrix} \underline{T}'_{k-1} & q_k \end{pmatrix},$$

where $q_k = Z_k^T p_k$. The matrix Z_k is unchanged. While building the basis, the last component of q_k is nonzero and the swap can always be done (see Leonard [20, pp. 94–99]). After the line search, g_{k+1} is accepted and the orthogonalization procedure yields

$$B''_k = (\, B'_k \quad g_{k+1} \,) = (\, p_0 \quad \cdots \quad p_{k-1} \quad p_k \quad g_{k+1} \,), \qquad T''_k = \begin{pmatrix} T'_k & u_k \\ 0 & \rho_{k+1} \end{pmatrix},$$

and $Z'_k = (\, z_0 \ \cdots \ z_{k-1} \ z_k \ z_{k+1} \,)$. These matrices are then passed to iteration $k+1$ as B_{k+1}, T_{k+1}, and Z_{k+1}, respectively.

3.2. Discarding the oldest basis vector. Now suppose that $k = m-1$. Given the assumption that every gradient is accepted, there are $m+1$ vectors in the basis at the end of this iteration. At this point, p_0, the oldest search direction, must be discarded before starting iteration $k+1$. This gives the new basis

$$B_{k+1} = (\, p_1 \quad \cdots \quad p_k \quad g_{k+1} \,).$$

(On the other hand, if at least one gradient is rejected, then no vector is discarded at iteration m. In this case, B_{k+1} contains r_{k+1} ($r_{k+1} \leq m$) linearly independent vectors consisting of at most one gradient (the vector g_{k+1}) and a linearly independent set of search directions. If g_{k+1} is rejected, B_{k+1} will consist of r_{k+1} linearly independent search directions.) Discarding a vector from the basis will decrease the rank by one. Hence, a symbol r'_k is needed for the intermediate rank determined by the orthogonalization procedure ***orthog***. The final rank r_{k+1} is then either $r'_k - 1$ or r'_k depending upon whether or not a basis vector is discarded.

When the oldest direction p_0 is discarded, the removal of its associated column from the basis must be reflected in all factorizations associated with B''_k. To simplify the description, the subscript k is suppressed, and a bar is used to denote quantities with subscript $k+1$.

The relationship between the old and new bases B'' and $\bar{B}$ ($= B_{k+1}$) is given by $B'' = (\, p_0 \quad \bar{B} \,)$, where $\bar{B}$ is $n \times m$. Associated with $\bar{B}$, we require $\bar{Z}$ and $\bar{T}$ such that $\bar{B} = \bar{Z}\bar{T}$. Moreover, the change from Z' to $\bar{Z}$ induces a corresponding change to the Cholesky factor. If R''' denotes the factor defined by the ***reinitialize*** procedure, then we require the factor $\bar{R}$ such that $\bar{R}^T\bar{R} = \bar{Z}^T H'' \bar{Z}$, where H'' is defined as in (2.7) but in terms of $\bar{\sigma}$, Z', and R'''. The matrix H'' is the H_{k+1} defined in section 2.2.

Daniel et al. [6] give the following method for updating Z' and T''. Given any orthogonal S, the orthogonal factorization of B'' may be written as $B'' = Z'T'' = Z'S^T S T'' = Z_S T_S$ (say). The matrix S is constructed so that T_S has the partitioned form

$$T_S = \begin{pmatrix} t & \bar{T} \\ \tau & 0 \end{pmatrix},$$

where $\bar{T}$ is the desired $m \times m$ upper-triangular matrix, t is an m-vector, and τ is a scalar. In particular, $S = P_{m,m+1} P_{m-1,m} \cdots P_{12}$, where $P_{i,i+1}$ is an $(m+1) \times (m+1)$ plane rotation in the $(i, i+1)$ plane that annihilates the $(i+1, i+1)$ element of $P_{i-1,i} \cdots P_{12} T''$.

The matrix $\bar{Z}$ consists of the first m columns of Z_S, i.e., $Z_S = (\bar{Z} \quad z)$, where z is an n-vector. From the definition of B'', we have

$$B'' = (p_0 \quad \bar{B}) = Z_S T_S = (\bar{Z}t + \tau z \quad \bar{Z}\bar{T}),$$

and it follows that $\bar{B} = \bar{Z}\bar{T}$ is the required orthogonal factorization.

Next, we propose how to update R''' when the first column of B'' is discarded. The old and new orthogonal bases are related by the identity $\bar{Z} = Z_S E_m$, where E_m comprises the first m columns of the identity matrix of order $m+1$. From the definitions of $\bar{Z}$ and H'', the new reduced Hessian is given by

$$\bar{Z}^T H'' \bar{Z} = E_m^T Z_S^T H'' Z_S E_m = E_m^T S Z'^T H'' Z' S^T E_m = E_m^T S R'''^T R''' S^T E_m.$$

In general, $R'''S^T$ is not upper triangular, but it may be restored to upper-triangular form by a second sweep of plane rotations $\widetilde{S}$. The $(m+1) \times (m+1)$ matrix $\widetilde{S}$ is orthogonal and is chosen so that $\widetilde{S}R'''S^T$ is upper triangular. If $R_S = \widetilde{S}R'''S^T$ denotes the resulting product, then $\bar{Z}^T H'' \bar{Z} = E_m^T R_S^T R_S E_m$, which implies that the leading $m \times m$ block of R_S is the required factor $\bar{R}$. Note that at this point, we can define $\bar{H}$ $(= H_{k+1})$ in terms of $\bar{R}$, $\bar{Z}$, and $\bar{\sigma}$.

The matrix $\widetilde{S}$ is the product $\widetilde{P}_{m,m+1} \cdots \widetilde{P}_{23} \widetilde{P}_{12}$, where $\widetilde{P}_{i,i+1}$ is an $(m+1) \times (m+1)$ plane rotation in the $(i, i+1)$ plane that annihilates the $(i, i+1)$ element of $\widetilde{P}_{i-1,i} \cdots \widetilde{P}_{12} R''' P_{12}^T \cdots P_{i,i+1}^T$. In practice, the two sweeps $\widetilde{S}$ and S are interlaced so that only $\mathcal{O}(m^2)$ operations are required.

It remains to show how u' $(= Z'^T \bar{g})$ is updated, thereby avoiding the mn operations necessary to compute the new reduced gradient u'' $(= \bar{Z}^T \bar{g})$ from scratch. The identity $u'' = \bar{Z}^T \bar{g} = (Z_S E_m)^T \bar{g} = E_m^T S (Z'^T \bar{g}) = E_m^T S u'$ implies that u'' comprises the first m components of Su'.

3.3. Comparison of the bases. We now revert to using subscripts to denote iteration indices. Under the assumption that every gradient is accepted, the gradient and search-direction bases at the start of iteration $m-1$ are given by $G_{m-1} = (g_0 \quad g_1 \quad \cdots \quad g_{m-1})$ and $P_{m-1} = (p_0 \quad p_1 \quad \cdots \quad p_{m-2} \quad g_{m-1})$. Theorem 2.2 implies that $\text{range}(G_{m-1}) = \text{range}(P_{m-1})$, and we can expect that the value of p_{m-1} is independent of the choice of basis. However, the following argument shows that this is not necessarily true for p_m, and hence the gradient and search-direction bases are not necessarily the same in the limited-memory context.

At the end of iteration $m-1$, both bases will have $m+1$ vectors. In the limited-memory context, the oldest basis vector must be discarded, giving bases $G_m = (g_1 \quad g_2 \quad \cdots \quad g_m)$ and $P_m = (p_1 \quad p_2 \quad \cdots \quad p_{m-1} \quad g_m)$. These bases do not include g_0 and p_0 (which is parallel to g_0), respectively. Note that p_{m-1} is not necessarily in $\text{range}(G_m)$ because p_{m-1} may have a nonzero component of g_0, which has been discarded from the gradient basis. Since $p_{m-1} \in \text{range}(P_m)$ by construction, it follows that $\text{range}(G_m) \neq \text{range}(P_m)$. We will discuss a specific implication of this phenomenon in section 3.7.

3.4. An implicit representation of Z. When a basis vector is discarded, the application of the plane rotations to the right of Z_k' requires approximately $4mn$ operations. Although it is possible to reduce this to $3mn$ operations (see Daniel et al. [6]), the update to Z_k' dominates the time to perform an iteration and reduces the efficiency compared to other methods. For example, the *total* number of operations for an iteration of the limited-memory method of Nocedal [26] is approximately $4mn$.

Our limited-memory reduced-Hessian method is substantially faster if, as proposed by Siegel, the basis matrix B_k is stored instead of Z_k. In this case, products involving Z_k are computed as needed using T_k and B_k, and the number of operations required to drop a column from the basis is reduced to $\mathcal{O}(r_k^2)$.

With an implicit definition of Z_k, the orthogonalization procedure becomes a method for updating the orthogonal factorization of B_k *without storing* Z_k. Given B_k and a new gradient g_{k+1}, the first step is to compute $u_k = Z_k^T g_{k+1}$ from the equations $T_k'^T u_k = B_k'^T g_{k+1}$. Once u_k is known, ρ_{k+1} can be computed from the identity $\rho_{k+1}^2 = \|g_{k+1}\|^2 - \|u_k\|^2$. The updated triangular factor T_k'' is defined by augmenting T_k' by a column formed from $u_k = Z_k^T g_{k+1}$ and ρ_{k+1} (see (2.5)). Note that the column z_{k+1} is not needed. The implicit form of Z_k reduces the cost of the orthogonalization procedure by half to approximately nr_k operations.

3.5. The limited-memory algorithm. We have described a reduced-Hessian limited-memory algorithm that needs three procedures in addition to those needed by Algorithm RHR: a *swap*, which replaces an accepted gradient g_k with p_k in the definition of B_k, giving a basis defined by B_k' and T_k'; a new *orthogonalize*, which orthogonalizes g_{k+1} with respect to Z_k, giving a new orthonormal basis defined by B_k'' and T_k''; and a *discard*, which drops the oldest search direction from the basis. As in Algorithm RHR, statements of the form $(B_k', T_k') = \boldsymbol{swap}(B_k, T_k)$ indicate computed quantities and their dependencies associated with a given procedure. Similarly, the results of the implicit orthogonalization and discard procedures are denoted by $(B_k'', T_k'', u_k, \rho_{k+1}, r_k') = \boldsymbol{iorthog}(B_k', T_k', g_{k+1}, r_k)$ and $(B_{k+1}, T_{k+1}, R_{k+1}, u_k'') = \boldsymbol{drop}(B_k'', T_k'', R_k''', u_k')$.

ALGORITHM 3.1. (L-RHR) LIMITED-MEMORY VERSION OF ALGORITHM RHR.
Choose x_0, σ_0 ($\sigma_0 > 0$), and m ($m \geq 2$);
$k = 0$; $r_0 = 1$; $g_0 = \nabla f(x_0)$;
$B_0 = \begin{pmatrix} g_0 \end{pmatrix}$; $T_0 = \begin{pmatrix} \|g_0\| \end{pmatrix}$; $v_0 = \|g_0\|$; $R_0 = \begin{pmatrix} \sigma_0^{1/2} \end{pmatrix}$;
while not converged **do**
  Solve $R_k^T d_k = -v_k$; $R_k q_k = d_k$;
  Solve $T_k w = q_k$; $p_k = B_k w$;
  if g_k was accepted **then** $(B_k', T_k') = \boldsymbol{swap}(B_k, T_k)$;
  Find α_k satisfying the Wolfe conditions (2.2);
  $x_{k+1} = x_k + \alpha_k p_k$; $g_{k+1} = \nabla f(x_k + \alpha_k p_k)$;
  $(B_k'', T_k'', u_k, \rho_{k+1}, r_k') = \boldsymbol{iorthog}(B_k', T_k', g_{k+1}, r_k)$;
  $(R_k', u_k', v_k', q_k') = \boldsymbol{expand}(R_k, u_k, v_k, q_k, \rho_{k+1}, \sigma_k)$;
  $s_k = \alpha_k q_k'$; $y_k = u_k' - v_k'$; $R_k'' = \boldsymbol{update}(R_k', s_k, y_k)$;
  Compute σ_{k+1}; $R_k''' = \boldsymbol{reinitialize}(R_k'', \sigma_{k+1})$;
  if r_k' equals $m+1$ **then**
    $(B_{k+1}, T_{k+1}, R_{k+1}, u_k'') = \boldsymbol{drop}(B_k'', T_k'', R_k''', u_k')$; $r_{k+1} = m$;
  else
    $R_{k+1} = R_k'''$; $B_{k+1} = B_k''$; $T_{k+1} = T_k''$; $u_k'' = u_k'$; $r_{k+1} = r_k'$;
  end if
  $v_{k+1} = u_k''$;
  $k = k + 1$;
end do

Iteration k of Algorithm L-RHR requires $2nr_k+2n+\mathcal{O}(r_k^2)$ operations. (This total includes the work required for the ***swap***, ***iorthog***, ***update***, and ***drop*** procedures but does not include any overhead incurred during the line search.)

3.6. Keeping the reduced Hessian vs. the reduced inverse Hessian. If Siegel's algorithm and an un-reinitialized version of L-RHR are applied with the same line search, then the same search directions and subspace bases are generated for the first m iterations. During these iterations the full $n\times n$ Hessian of L-RHR (see (2.7)) is the inverse of the full inverse Hessian of Siegel's method. However, once a basis vector is discarded, the methods generate different search directions. In both algorithms, the updating procedures associated with a discard relegate curvature information associated with the oldest basis vector to the last row and column of their respective reduced matrices. The off-diagonal entries of this row and column are replaced by zero, and the diagonal element is set to either σ or $1/\sigma$ depending on the method. At this point the two basis matrices generate the same subspace, but because the leading $m\times m$ principal submatrices of a symmetric matrix and its inverse are not generally the inverse of each other, the full Hessian of L-RHR is no longer the inverse of the full inverse Hessian of Siegel's method. At the next iteration, the methods generate different search directions, and subsequent bases and reduced matrices are no longer related.

3.7. Finite termination on quadratics. Next we briefly discuss the properties of Algorithm L-RHR when it is applied with an exact line search to a strictly convex quadratic function.

THEOREM 3.1. *Consider Algorithm* L-RHR *implemented with an exact line search and $\sigma_0 = 1$. If this algorithm is applied to a strictly convex quadratic function, then R_k, B_k, and T_k $(k \geq 1)$ satisfy*

$$R_k = \begin{pmatrix} a_l/h_l & b_l & 0 & \cdots & 0 \\ & a_{l+1} & b_{l+1} & \ddots & \vdots \\ & & \ddots & \ddots & 0 \\ & & & a_{k-1} & b_{k-1} \\ & & & & \sigma_k^{1/2} \end{pmatrix}, \quad B_k = \begin{pmatrix} p_l & p_{l+1} & \cdots & p_{k-1} & g_k \end{pmatrix},$$

and

$$T_k = \begin{pmatrix} h_l d_l & h_l t_{l,l+1} & h_l t_{l,l+2} & \cdots & h_l t_{l,k-1} & 0 \\ & d_{l+1} & t_{l+1,l+2} & \cdots & t_{l+1,k-1} & 0 \\ & & d_{l+2} & & \vdots & \vdots \\ & & & \ddots & t_{k-2,k-1} & 0 \\ & & & & d_{k-1} & 0 \\ & & & & & \|g_k\| \end{pmatrix},$$

where $l = \max\{0, k-m+1\}$ and the scalars a_j, b_j, t_{ij}, d_j, and h_j are given by

$$a_j = \frac{\|g_j\|}{(y_j^T s_j)^{1/2}}, \quad b_j = -\frac{\|g_{j+1}\|}{(y_j^T s_j)^{1/2}}, \quad t_{ij} = -\frac{\|g_j\|^2}{\sigma_j \|g_i\|}, \quad d_j = -\frac{\|g_j\|}{\sigma_j}, \quad h_j = \delta_j \frac{\|p_j\|\sigma_j}{\|g_j\|}$$

with $\delta_j = 1$ if $j = 0$, and $\delta_j = -1$ otherwise. Furthermore, the search directions are given by

$$p_0 = -g_0; \qquad p_k = -\frac{1}{\sigma_k} g_k + \beta_{k-1} p_{k-1}, \quad \beta_{k-1} = \frac{\sigma_{k-1}}{\sigma_k} \frac{\|g_k\|^2}{\|g_{k-1}\|^2}, \quad k \geq 1.$$

Proof. See Leonard [20]. □

COROLLARY 3.2. *If Algorithm* L-RHR *is used to minimize a strictly convex quadratic under the conditions of Theorem* 3.1*, then the method converges to the minimizer in at most n iterations.*

Proof. We show by induction that the search directions are parallel to the conjugate-gradient directions $\{d_k\}$. Specifically, $\sigma_k p_k = d_k$ for all k. This is true for $k = 0$ since $1 \cdot p_0 = -g_0 = d_0$. Assume that $\sigma_{k-1} p_{k-1} = d_{k-1}$. Using Theorem 3.1 and the inductive hypothesis, we find

$$\sigma_k p_k = -g_k + \sigma_{k-1} \frac{\|g_k\|^2}{\|g_{k-1}\|^2} p_{k-1} = -g_k + \frac{\|g_k\|^2}{\|g_{k-1}\|^2} d_{k-1} = d_k,$$

which completes the induction. The result now follows from the quadratic termination property of the conjugate-gradient method. □

We remark that the specific form of L-RHR discussed in Theorem 3.1 defines a "rescaled" form of the classical Fletcher–Reeves conjugate-gradient method [12].

Let p_m^G and p_m^P denote the search directions defined during iteration m of the gradient- and search-direction variants of the limited-memory algorithm. Observe that p_{m-1} is parallel to d_{m-1}, regardless of which basis is used. However, since $g_m \in \text{range}(G_m)$, but possibly $p_{m-1} \notin \text{range}(G_m)$, the vector p_m^G may not be parallel to d_m and the gradient-basis variant does not have quadratic termination.

4. Implementation details. In this section, we describe some details associated with a particular implementation of Algorithm L-RHR. We outline a method for improving the orthonormal basis and discuss a practical criterion for accepting a gradient. We also provide information about the line search, the BFGS update, and restarts.

4.1. Reorthogonalization. For general applications the implicit QR version of Z_k is recommended, with default memory size $m = 5$. For larger values of m (e.g., $m \geq 15$), it is often beneficial to use *reorthogonalization* in combination with the *explicit* QR. L-RHR employs the following reorthogonalization scheme proposed by Daniel et al. [6]. Let u_k and w_k denote the computed values of $Z_k^T g_{k+1}$ and $g_{k+1} - Z_k u_k$, respectively. The vectors u_k and w_k may be improved using one or more steps of the iterative refinement scheme:

$$\Delta u_k = Z_k^T w_k, \qquad u_k \leftarrow u_k + \Delta u_k;$$

and

$$\Delta w_k = -Z_k \Delta u_k, \qquad w_k \leftarrow w_k + \Delta w_k.$$

L-RHR uses criteria suggested by Daniel et al. [6] for invoking and terminating the reorthogonalization. Each step of reorthogonalization adds approximately $2nr_k$ operations to the cost of an iteration. Moreover, the use of an explicit Z_k requires an additional nr_k operations for the calculation of z_{k+1} and an extra $3nr_k$ operations when a basis vector is discarded (since the plane rotations associated with a discard must be applied to Z_k as well as T_k).

4.2. The criterion for gradient acceptance. In exact arithmetic, a gradient is accepted for the basis if $\rho_{k+1} > 0$, where ρ_{k+1} is the norm of $(I - Z_k Z_k^T)g_{k+1}$. This condition ensures that the basis vectors are linearly independent, and hence that T_k'' is nonsingular. When ρ_{k+1} is computed in finite-precision, gradients with small (but nonzero) ρ_{k+1} must be rejected to prevent T_{k+1} and B_{k+1} from being too ill-conditioned. In practice, an accepted gradient must satisfy $\rho_{k+1} \geq \epsilon \|g_{k+1}\|$, where ϵ is a preassigned positive constant. In the results of section 5, ϵ was set to 10^{-4}. Rounding error in the calculation of ρ_{k+1} is exacerbated by the use of an implicit form for Z_k—for example, it is necessary to reject g_{k+1} if the computed value of $\rho_{k+1}^2 = \|g_{k+1}\|^2 - \|u_k\|^2$ is negative. However, a negative computed value of ρ_{k+1}^2 rarely occurred in our experiments, and when it did, it did not prevent the method from terminating successfully (see section 5 for the criterion used). For example, a negative value was computed 268 times during the 69747 iterations in the runs of Table 5.6 below. Moreover, of the 10 problems in which a negative value occurred, all were solved successfully.

4.3. The line search, the BFGS update, and restarts. The line search is a slightly modified version of the one used in the package NPSOL [16]. It is designed to ensure that α_k satisfies the so-called strong Wolfe conditions,

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \mu \alpha_k g_k^T p_k \quad \text{and} \quad |g_{k+1}^T p_k| \leq \eta |g_k^T p_k|, \tag{4.1}$$

where the constants μ and η are chosen so that $0 \leq \mu < \eta < 1$ and $\mu < \frac{1}{2}$ (see Gill et al. [16] or Fletcher [9, pp. 26–30]). The step-length parameters are $\mu = 10^{-4}$ and $\eta = 0.9$. The line search is based on using a safeguarded polynomial interpolation to find an approximate minimizer of the univariate function

$$\phi_k(\alpha) = f(x_k + \alpha p_k) - f(x_k) - \mu \alpha g_k^T p_k$$

(see Moré and Sorensen [23]). The step α_k is the first member of a minimizing sequence $\{\alpha_k^i\}$ that satisfies the Wolfe conditions. The sequence is usually started with $\alpha_k^0 = 1$ (see below).

If α_k satisfies the strong Wolfe conditions, it follows that $y_k^T s_k \geq -(1-\eta) g_k^T s_k > 0$ and the BFGS update can be applied without difficulty. On very difficult problems, however, the combination of a poor search direction and rounding error in f may prevent the line search from satisfying the line search conditions within 20 function evaluations. In this case, the search terminates with the step corresponding to the best value of f found so far. If this α_k defines a strict decrease in f, the minimization continues. In this case, the BFGS update is skipped unless $y_k^T s_k \geq \epsilon_M |g_k^T s_k|$, where ϵ_M is the machine precision. If a strict decrease is not obtained after 20 function evaluations, the algorithm is restarted with $T_k = \big(\, \|g_k\| \,\big)$, $v_k = \|g_k\|$, and $R_k = \big(\, \sigma_k^{1/2} \,\big)$. To prevent the method from degenerating into steepest descent, no more restarts are allowed until the reduced Hessian has built up to its full size of m rows and columns. In practice, a restart is rarely invoked. For example, in the experiments of Table 5.6, L-RHR used only one restart (on problem *freuroth*, which was not solved successfully). For comparison, L-BFGS-B used two restarts (on problem *bdqrtic*, again without success).

If p_k is a poorly scaled version of the steepest-descent direction, the step to a minimizer of $\phi_k(\alpha)$ may be very small relative to one, and a large number of function evaluations may be needed to find an acceptable step length. To prevent this inefficiency, the initial step for the first line search and each line search immediately

following a restart is limited so that $\alpha_k^0 \leq \min\{\Delta/\|p_k\|, 1\}$, where Δ is a preassigned constant ($\Delta = 2$ in the experiments described in the next section). This procedure ensures that the initial change in x does not exceed Δ.

5. Numerical results. In this section, we give numerical results for most of the large unconstrained problems in the CUTE[2] collection (see Bongartz et al. [1]). After some discussion of the test problems, we compare L-RHR with and without reinitialization. Next, we illustrate the differences between L-RHR and Siegel's Algorithm 6 [33], which we refer to as ALG6. This is followed by results that compare L-RHR with L-BFGS and L-BFGS-B, which are two alternative implementations of the limited-memory BFGS method. L-BFGS is based on an algorithm that maintains an implicit approximate inverse Hessian as a sequence of update pairs. L-BFGS-B employs an algorithm that updates an approximate Hessian in factored form $\theta I - WMW^T$, where θ is a scalar and WMW^T is a matrix of low rank. L-BFGS-B is intended for problems with upper and lower bounds on the variables but is also recommended over L-BFGS-B for unconstrained problems (see Zhu et al. [34]).

Throughout, we use m_{LB} to denote the number of update pairs to be kept in memory by L-BFGS and L-BFGS-B. This should not be confused with m, the number of vectors stored by L-RHR.

5.1. Test problem selection. The test set was constructed using the CUTE interactive `select` tool, which allows the identification of groups of problems with certain features. In our case, the `select` tool was first used to locate the twice-continuously differentiable unconstrained problems for which the number of variables in the data file can be varied. Of these problems, the number of variables was set to a value in the range $100 \leq n \leq 1500$ according to criteria that we discuss below. The input for the `select` tool was as follows:

```
Objective function type         : *
Constraints type                : U  (No constraints)
Regularity                      : R  (twice-cont. differentiable)
Degree of available derivatives : *
Problem interest                : *
Explicit internal variables     : *
Number of variables             : v  (variable dimension)
Number of constraints           : 0.
```

A total of 87 problems was obtained from this selection. Six fixed-dimension problems were obtained by using the `select` tool with the number of variables set as follows:

```
Number of variables             : in [    50,   1000 ].
```

Additional criteria were used to determine the suitability of these 93 problems, as we now explain.

After using the `select` tool, it remained to determine a suitable value of n for the problems with variable dimension. The value $n = 1500$ was used for the twelve problems *dixmaana–dixmaanl*, as suggested by Zhu et al. [34]. Values $n \approx 1000$ were used for most of the remaining problems, but it was necessary to choose significantly smaller values of n in some cases. The problems *chnrosnb*, *errinros*, and *watson* have limits on the size of n, and the mandated maximum values of 50, 50, and 31, respectively, were used in these cases. It was also necessary to limit n to be less than

[2]The version of CUTE used was obtained September 7, 2001.

1000 if a problem could not be decoded using the CUTE decoder `sifdec` (compiled with the option `tobig`). For any such problem, the values $n = 300$ and $n = 100$ were tried successively until the decoding succeeded. Problems in this category were *arglina–arglinc*, *brownal*, *hilberta*, *hilbertb*, *mancino*, *penalty3*, and *sensors*. The value $n = 300$ was used for *arglina*, *brownal*, *hilberta*, and *hilbertb*. The value $n = 100$ was used for *mancino*, *penalty3*, and *sensors*. The problems *arglinb* ($n = 300$) and *arglinc* ($n = 100$) and *penalty3* ($n = 100$) were successfully decoded but were removed from the set for reasons described below.

A value of n such that $n < 1000$ was also used if both L-BFGS-B and L-RHR failed to meet the termination criterion with m, $m_{LB} = 5$, 15, 30, and 45. (The termination criterion will not be satisfied if there is a failure in the line search or 40,000 iterations are completed.) In this case, 300 and 100 were tried successively to determine an acceptable value for n. The problems *arglinb*, *arglinc*, *curly10*, *curly20*, *curly30*, *fletchbv*, *hydc20ls* (fixed $n = 99$), *indef*, *nonmsqrt*, *penalty3*, *sbrybnd*, *scosine*, *scurly10*, *scurly20*, and *scurly30* were removed from the test set since, even with $n = 100$, neither method could meet the termination criterion. The value $n = 100$ was used for *penalty2* since neither L-RHR nor L-BFGS-B could achieve the termination criterion with $n = 1000$ or $n = 300$.

These selection criteria had the effect of removing 15 problems from the list generated by the select tool. This left 78 problems suitable for testing. For completeness, we list the problems not already mentioned, with their associated values of n. There were 44 variable-dimension problems with $n = 1000$: *arwhead*, *bdqrtic*, *broydn7d*, *brybnd*, *chainwoo*, *cosine*, *cragglvy*, *dixon3dq*, *dqdrtic*, *dqrtic*, *edensch*, *engval1*, *extrosnb*, *fletcbv2*, *fletcbv3*, *fletchcr*, *freuroth*, *genhumps*, *genrose*, *liarwhd*, *morebv*, *ncb20*, *ncb20b*, *noncvxu2*, *noncvxun*, *nondia*, *nondquar*, *penalty1*, *powellsg*, *power*, *quartc*, *schmvett*, *sinquad*, *sparsine*, *sparsqur*, *spmsrtls*, *srosenbr*, *testquad*, *tointgss*, *tquartic*, *tridia*, *vardim*, *vareigvl*, and *woods*. There were four problems with $n = 1024$: *fminsrf2*, *fminsurf*, *msqrtals*, and *msqrtbls*. The remaining three variable-dimension problems were *eigenals* ($n = 1056$), *eigenbls* ($n = 1056$), and *eigencls* ($n = 1122$). Finally, the names and numbers of variables of the five fixed-dimension problems included in the test set were *deconvu* ($n = 61$), *eg2* ($n = 1000$), *tointgor* ($n = 50$), *tointpsp* ($n = 50$), and *tointqor* ($n = 50$).

All runs were made on a Sun UltraSPARC-IIi (single cpu at 333MHz) with 256MB of RAM. The algorithms L-RHR, L-BFGS-B, and L-BFGS are coded in Fortran and were compiled using g77. ALG6 is coded in C and was compiled using gcc. Full compiler optimization was used in all cases. The caption of each table specifies the amount of limited memory used and indicates whether or not reinitialization and/or reorthogonalization was used. All methods were terminated when $\|g_k\|_\infty < 10^{-5}$, as proposed by Zhu et al. [34].

5.2. L-RHR with and without reinitialization. Table 5.1 gives the results of running L-RHR both with and without reinitialization. Without reinitialization, the parameter σ was fixed at $y_0^T s_0 / \|s_0\|^2$ (see (2.6)).[3] This is the scheme proposed by Siegel [33]. With reinitialization, $\sigma_0 = 1$ and $\sigma_k = y_k^T y_k / y_k^T s_k$ ($k \geq 1$), which are the reciprocals of the parameters used by Liu and Nocedal [21]. Of the 78 problems attempted, L-RHR with reinitialization solved 74 problems satisfactorily and reduced the gradient to within two orders of magnitude of the 10^{-5} target value on three

[3]The steepest-descent direction is used for the first iteration. After the first step, σ is set to $y_0^T s_0 / \|s_0\|^2$ and R is defined accordingly.

others (*bdqrtic*, *freuroth*, and *noncvxun*). The algorithm was unable to reduce $\|g_k\|_\infty$ below 1.5×10^{-2} for *fletcbv3*. Without reinitialization, L-RHR was able to solve only 70 problems and required considerably more function and gradient evaluations on almost every problem attempted. (The additional four unsolved problems were *chainwoo*, *cragglvy*, *edensch*, and *penalty2*.) Table 5.1 gives the total number of iterations, function evaluations and cpu seconds for L-RHR with and without reinitialization on the 70 problems that could be solved by both versions. These results indicate that reinitialization provides substantial practical benefits and indicates an advantage of L-RHR compared to Siegel's method, which does not include reinitialization. A direct comparison between L-RHR and Siegel's method is given in the next section.

TABLE 5.1
L-RHR[a] *with and without reinitialization on* 70 *CUTE problems.*

L-RHR	Itns	Fncs	Cpu	Fail
with reinitialization	65115	66914	1567	4
without reinitialization	83611	107253	1884	8

[a] $m = 5$; without reorthogonalization.

5.3. L-RHR compared with ALG6. Next we compare L-RHR with Siegel's ALG6. In the first set of runs, ALG6 uses the recommended value of $\epsilon = 10^{-3}$ for the gradient acceptance parameter (see [32, p. 8]). The line search in ALG6 is a slightly modified version of the one in Powell's Fortran package TOLMIN [28]. It attempts to satisfy the Wolfe conditions (see (2.2)) with $\mu = 10^{-2}$ and $\eta = 0.9$ but allows $f(x_{k+1}) \geq f(x_k)$ to within a small tolerance. ALG6 is not optimized for cpu time, and it may be possible to improve the performance by making appropriate changes to the code. However, the *relative* differences in cpu times are unlikely to be altered by recoding because many of the run times are dominated by the cumulative cost of the function evaluations (see section 5.5).

Table 5.2 gives the results of comparing ALG6 with a version of L-RHR implemented *without* reinitialization. Algorithm L-RHR succeeded on 70 of the 78 test problems and reduced the gradient norm to at most 10^{-3} on seven others: *bdqrtic*, *chainwoo*, *cragglvy*, *edensch*, *freuroth*, *noncvxun*, and *penalty2*. On the other unsuccessful case, *fletcbv3*, the final gradient norm was 1.1×10^{-1}. ALG6 succeeded on 74 out of the 78 problems. On *arwhead*, *bdqrtic*, and *noncvxun*, ALG6 was able to reduce the gradient norm to at most 10^{-3}. On *fletcbv3*, ALG6 reduced the gradient norm to 4.7×10^{-2}. Table 5.2 summarizes the results for the 69 problems that both methods were able to solve successfully. If the L-RHR line search is made to conform to ALG6 by allowing $f(x_{k+1}) \geq f(x_k)$ to within a prescribed tolerance, then L-RHR is able to solve three more problems, *chainwoo*, *edensch*, and *penalty2*.

TABLE 5.2
L-RHR[a] *compared with* ALG6[b] *on* 69 *CUTE problems.*

Method	Itns	Fncs	Cpu	Fail
L-RHR	83601	107237	1884	8
ALG6	101959	194091	5744	4

[a] $m = 5$; with no reinitialization and no reorthogonalization; $\epsilon = 10^{-4}$.
[b] $m = 5$; with Siegel's version of the TOLMIN line search; $\epsilon = 10^{-3}$.

Since the L-RHR and ALG6 directions have similar definitions when L-RHR does not use reinitialization, it might seem surprising that L-RHR requires significantly fewer function evaluations than ALG6. This phenomenon can be partly explained by differences in the line search and the different choice of ϵ. To illustrate these effects, Table 5.3 gives a comparison between L-RHR and ALG6 when both algorithms are implemented with the NPSOL line search and $\epsilon = 10^{-4}$. Note that the number of function evaluations for ALG6 decreases dramatically, though the stricter requirement that $f(x_{k+1}) < f(x_k)$ results in a few more failures.

TABLE 5.3
L-RHR[a] *compared with* ALG6[b] *on* 69 *CUTE problems.*

Method	Itns	Fncs	Cpu	Fail
L-RHR	82363	105997	1884	8
ALG6	86970	138512	3620	8

[a] $m = 5$; with no reinitialization and no reorthogonalization; $\epsilon = 10^{-4}$.
[b] $m = 5$; with the line search from NPSOL; $\epsilon = 10^{-4}$.

The results are closer, but Table 5.3 illustrates that the methods are still generating different iterates. This is because, in the limited-memory context, an algorithm based on updating a reduced Hessian is fundamentally different from an algorithm based on updating a reduced *inverse* Hessian. The directions generated by ALG6 and an un-reinitialized version of L-RHR are only the same until a basis vector is discarded. From this point, the Hessian of L-RHR is no longer related to the inverse Hessian of ALG6 (see section 3.6). For example, Table 5.4 illustrates that if L-RHR and ALG6 are applied to problem *msqrtals* with $m = 30$, the function values and gradient norms are in close agreement at iteration 30. At the next iteration the first discard is made and most of the agreement is lost. By iteration 50, only 1 significant digit of agreement remains. The total numbers of iterations required are 2556 and 3133 for L-RHR and ALG6, respectively. It follows that L-RHR's significant advantage in the "Fncs" column of Table 5.3 results from the use of a reduced Hessian instead of a reduced inverse Hessian.

TABLE 5.4
L-RHR[a] *and* ALG6[b] *applied to msqrtals with* $m = 30$.

	L-RHR		ALG6	
k	f_k	$\|g_k\|_\infty$	f_k	$\|g_k\|_\infty$
30	0.387222166177969	0.583990052575414	0.387222166177971	0.583990052575427
31	0.341636067001799	0.569093782828364	0.341694835004789	0.569019238515868
50	0.096802831009406	0.126889757489794	0.097640019163964	0.140754040816826

[a] $m = 5$; with no reinitialization and no reorthogonalization; $\epsilon = 10^{-4}$.
[b] $m = 5$; with the line search from NPSOL; $\epsilon = 10^{-4}$.

We provide one final comparison in which L-RHR uses reinitialization. In this case, both L-RHR and ALG6 succeed on 74 problems, and there are 73 problems that are solved by both methods. Table 5.5 shows the overall results for these 73 problems. Note that, overall, the use of reinitialization by L-RHR results in significantly fewer function evaluations compared to ALG6.

TABLE 5.5
L-RHR[a] *compared with* ALG6[b] *on* 73 *CUTE problems.*

Method	Itns	Fncs	Cpu	Fail
L-RHR	69737	71782	1592	4
ALG6	103217	195405	5752	4

[a] $m = 5$; with reinitialization; without reorthogonalization.
[b] $m = 5$; with Siegel's implementation of the TOLMIN line search.

5.4. L-RHR compared with L-BFGS-B. L-RHR and Seigel's method are related to the limited-memory BFGS method of Byrd et al. [4] because all three methods consolidate the quasi-Newton updates into dense matrices. By contrast, L-BFGS keeps an implicit inverse Hessian by storing a fixed number of vector pairs (γ_k, δ_k) (see (2.1) for the definitions of γ_k and δ_k). Products of the inverse Hessian with a vector are then formed without the need to keep an explicit H_k (see Nocedal [26]). Consolidation of the updates is crucial for efficiency if a limited-memory method is to be extended to handle upper and lower bounds on the variables. In this section we compare L-RHR with Version 2.1 of the code L-BFGS-B (see Zhu et al. [34]), which is an implementation of the method of Byrd, Lu, Nocedal and Zhu. L-BFGS-B applies the strong Wolfe conditions (4.1) using the line search of Moré and Thuente [24] with line search parameters $\mu = 10^{-4}$ and $\eta = 0.9$. The memory for L-BFGS-B was limited to $m_{LB} = 5$ pairs of vectors, which is twice the storage used by L-RHR.

Table 5.6 summarizes the performance of L-RHR and L-BFGS-B on the 74 CUTE problems on which both methods succeed. On these 74 problems, L-RHR requires fewer function evaluations than L-BFGS-B on 27 problems and more function evaluations on 43 problems. L-RHR requires less cpu time than L-BFGS-B on 55 problems and more cpu time than L-BFGS-B on 16 problems. L-BFGS-B was able to solve one more problem than L-RHR, namely, *fletcbv3* (L-RHR reduced the gradient norm to 1.5×10^{-2} in this case). Neither method was able to satisfy the termination criterion on *bdqrtic*, *freuroth*, and *noncvxun*. In these cases, the final gradient norms for L-RHR (L-BFGS-B) were 2.90×10^{-4} (6.08×10^{-4}), 3.40×10^{-4} (1.60×10^{-5}), and 1.00×10^{-3} (1.66×10^{-3}), respectively. Overall, L-RHR requires a comparable number of function evaluations and has a significant advantage in terms of cpu time.

TABLE 5.6
L-RHR[a] *and* L-BFGS-B[b] *on* 74 *CUTE problems.*

Method	Itns	Fcns	Cpu	Fail
L-RHR	69747	71798	1592	4
L-BFGS-B	66717	72264	1916	3

[a] $m = 5$ n-vectors; with reinitialization; without reorthogonalization.
[b] $m_{LB} = 5$ pairs of n-vectors.

Although the values $m = 5$ and $m_{LB} = 5$ are recommended for L-RHR and L-BFGS-B, it is of interest to investigate the relative performance of the algorithms as the memory size is increased. For example, the overhead required to solve an unsymmetric $2m_{LB} \times 2m_{LB}$ system every iteration of L-BFGS-B might suggest that L-RHR would have a greater cpu time advantage as m and m_{LB} are increased. Table 5.7 gives the performance of L-RHR and L-BFGS-B with increasing memory-size parameters m and m_{LB}. When $m \geq 15$ it is recommended that L-RHR uses reorthogonalization to maintain a good basis and improve robustness (see section 4.1). However, to illustrate

the difference when L-RHR with $m = 5$ does and does not use reorthogonalization, the results of Table 5.7 use reorthogonalization for all memory sizes. For $m = 5$, 15, 30, and 45, the numbers of failures for L-RHR without reorthogonalization are 4, 5, 7, and 11, respectively. With reorthogonalization, these numbers drop to 4, 4, 3, and 3, respectively. Table 5.7 provides the total number of reorthogonalizations required for each value of m. Although the amount of work per iteration is more than doubled with reorthogonalization, the cpu seconds required for $m = 5$ *decreases* from 1590 to 1580. In this case, the reductions in iterations and function evaluations compensates for the increased cost of computing the search direction.

All 78 problems were attempted, but the totals in each row include only the statistics for problems that could be solved by both methods. The cpu time advantage for L-RHR increases from 82% of the time required by L-BFGS-B to 59% as m and m_{LB} are increased from 5 to 45. However, L-BFGS-B gains an advantage in terms of the number of function evaluations. The interpretation of these results is complicated by the sharp increase in function evaluations when m is increased from 15 to 30. This increase occurs because both methods are able to solve problem *noncvxun* when $m = 30$. If *noncvxun* is removed from the problem set, a total of 69,522 (72,264), 67,293 (62,470), 64,808 (57,114), and 58,075 (54,391) evaluations are required by L-RHR (L-BFGS-B) for m (m_{LB}) with the values 5, 15, 30, and 45, respectively. These results indicate that for both methods, the number of function evaluations generally decreases as the limited-memory size is increased.

TABLE 5.7
L-RHR[a] *and* L-BFGS-B *with various memory sizes for* 78 *CUTE problems.*

	L-RHR					L-BFGS-B			
Mem	Itns	Fcns	Reors	Cpu	Fail	Itns	Fcns	Cpu	Fail
5	67573	69522	42018	1580	4	66717	72264	1916	3
15	65050	67293	57662	1951	4	57302	62470	2155	5
30	99122	101432	95445	2866	3	80669	86304	3744	2
45	78344	80550	76173	2782	3	65859	70962	4686	3

[a] With reinitialization and reorthogonalization.

5.5. L-RHR compared with L-BFGS. We conclude this section by providing a comparison of L-RHR with L-BFGS, which is a limited-memory BFGS method that maintains an implicit inverse approximate Hessian as a sequence of m_{LB} update pairs (see Nocedal [26] and Liu and Nocedal [21]). The recommended memory size is $m_{LB} = 5$. The search direction requires approximately $4nm_{LB}$ operations, which is roughly twice the work required by L-RHR. L-BFGS uses the Moré and Thuente line search with the same parameter settings as L-BFGS-B.

Table 5.8 summarizes the performance of L-RHR and L-BFGS on the 74 problems that both methods solved successfully. Of these 74 problems, L-RHR required fewer function evaluations on 27 problems and L-BFGS required fewer evaluations on 43 problems. L-RHR required less cpu time on 30 problems and more time on 41 problems. Overall, L-BFGS and L-RHR required comparable numbers of function evaluations and iterations. However, even though L-RHR requires roughly half as much work to compute the search direction, the overall cpu time was very close to that of L-BFGS. Further investigation using a performance profiler indicated that the cost of evaluating the objective for seven of the problems (*eigenals*, *eigenbls*, *eigencls*, *msqrtals*, *msqrtbls*, *ncb20*, and *ncb20b*) dominated the overall computation time. The

cpu time required by L-RHR to compute the search direction was less than 60% of that needed by L-BFGS. However, on these seven critical problems, the calculation of the search direction constitutes less than 10% of the solve time.

TABLE 5.8
L-RHR[a] *and* L-BFGS[b] *on* 74 *CUTE problems.*

Method	Itns	Fcns	Cpu	Fail
L-RHR	69747	71798	1592	4
L-BFGS	66445	71978	1670	4

[a] $m = 5$ n-vectors; with reinitialization; without reorthogonalization.
[b] $m_{LB} = 5$ pairs of n-vectors.

In order to compare L-RHR with L-BFGS as m and m_{LB} are increased, L-RHR uses reorthogonalization for improved robustness. As m and m_{LB} take on the values 5, 15, 30, and 45, the total function evaluations on the problems solved by both methods L-RHR (L-BFGS) are 69,522 (71,978), 67,293 (63,466), 101,432 (80,832), and 80,550 (88,717). As in the comparison with L-BFGS-B, L-RHR is competitive in terms of function evaluations, but in this case L-RHR requires more cpu time when $m \geq 15$. Although some of the functions dominate the overall cpu time when $m = 5$, their evaluation carries less weight with increasing m. This effect, when combined with the cost of reorthogonalization, is why L-RHR is slower than L-BFGS when $m = 45$, even though L-RHR requires significantly fewer function evaluations. We reemphasize that L-RHR requires approximately half the storage of L-BFGS.

6. Summary and conclusions. We have presented theoretical and practical details of a limited-memory reduced-Hessian method for large-scale smooth unconstrained optimization problems for which first derivatives are available. The method maintains the Cholesky factor of a reduced Hessian and requires roughly half the storage of conventional limited-memory methods.

The numerical results of Section 5 confirm that L-RHR is efficient and reliable on a set of large test problems from the CUTE collection. Moreover, it is shown that Hessian reinitialization and selective reorthogonalization are vital components of an efficient and robust reduced-Hessian method.

When compared to the state-of-the-art code L-BFGS-B on our test set, L-RHR converges in less cpu time, while requiring comparable numbers of function evaluations. Compared to the code L-BFGS, L-RHR required comparable numbers of function evaluations and iterations when both algorithms were applied with their default memory sizes.

Acknowledgments. We thank Dirk Siegel for graciously providing a copy of his limited-memory code. We also appreciate many suggestions from the referees and Associate Editor Jorge Nocedal.

REFERENCES

[1] I. Bongartz, A. R. Conn, N. I. M. Gould, and P. L. Toint, CUTE*: Constrained and unconstrained testing environment*, ACM Trans. Math. Software, 21 (1995), pp. 123–160.
[2] A. G. Buckley, *A combined conjugate-gradient quasi-Newton minimization algorithm*, Math. Program., 15 (1978), pp. 200–210.
[3] ———, *Extending the relationship between the conjugate-gradient and BFGS algorithms*, Math. Program., 15 (1978), pp. 343–348.
[4] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization*, SIAM J. Sci. Comput., 16 (1995), pp. 1190–1208.

[5] R. H. Byrd, J. Nocedal, and R. B. Schnabel, *Representations of quasi-Newton matrices and their use in limited-memory methods*, Math. Program., 63 (1994), pp. 129–156.
[6] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization*, Math. Comput., 30 (1976), pp. 772–795.
[7] J. E. Dennis, Jr. and R. B. Schnabel, *A new derivation of symmetric positive definite secant updates*, in Nonlinear programming, 4 (Proc. Sympos., Special Interest Group on Math. Programming, Univ. Wisconsin, Madison, Wis., 1980), Academic Press, New York, 1981, pp. 167–199.
[8] M. C. Fenelon, *Preconditioned Conjugate-Gradient-Type Methods for Large-Scale Unconstrained Optimization*, PhD thesis, Department of Operations Research, Stanford University, Stanford, CA, 1981.
[9] R. Fletcher, *Practical Methods of Optimization*, John Wiley and Sons, Chichester and New York, second ed., 1987.
[10] ———, *An optimal positive definite update for sparse Hessian matrices*, SIAM J. Optim., 5 (1995), pp. 192–218.
[11] R. Fletcher and M. J. D. Powell, *A rapidly convergent descent method for minimization*, Computer Journal, 6 (1963), pp. 163–168.
[12] R. Fletcher and C. M. Reeves, *Function minimization by conjugate gradients*, Computer Journal, 7 (1964), pp. 149–154.
[13] P. E. Gill and M. W. Leonard, *Reduced-Hessian quasi-Newton methods for unconstrained optimization*, SIAM J. Optim., 12 (2001), pp. 209–237.
[14] P. E. Gill and W. Murray, *Conjugate-gradient methods for large-scale nonlinear optimization*, Report SOL 79-15, Department of Operations Research, Stanford University, Stanford, CA, 1979.
[15] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM J. Optim., 12 (2002), pp. 979–1006.
[16] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *User's guide for NPSOL (Version* 4.0*): a Fortran package for nonlinear programming*, Report SOL 86-2, Department of Operations Research, Stanford University, Stanford, CA, 1986.
[17] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, London and New York, 1981.
[18] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, second ed., 1989.
[19] L. Kaufman, *Reduced storage, quasi-Newton trust region approaches to function optimization*, SIAM J. Optim., 10 (1999), pp. 56–69.
[20] M. W. Leonard, *Reduced Hessian Quasi-Newton Methods for Optimization*, PhD thesis, Department of Mathematics, University of California, San Diego, 1995.
[21] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Math. Program., 45 (1989), pp. 503–528.
[22] J. L. Morales and J. Nocedal, *Automatic preconditioning by limited memory quasi-Newton updating*, SIAM J. Optim., 10 (2000), pp. 1079–1096.
[23] J. J. Moré and D. C. Sorensen, *Newton's method*, in Studies in Mathematics, Volume 24. Studies in Numerical Analysis, Math. Assoc. America, Washington, DC, 1984, pp. 29–82.
[24] J. J. Moré and D. J. Thuente, *Line search algorithms with guaranteed sufficient decrease*, ACM Trans. Math. Software, 20 (1994), pp. 286–307.
[25] L. Nazareth, *A relationship between the BFGS and conjugate gradient algorithms and its implications for new algorithms*, SIAM J. Numer. Anal., 16 (1979), pp. 794–800.
[26] J. Nocedal, *Updating quasi-Newton matrices with limited storage*, Math. Comput., 35 (1980), pp. 773–782.
[27] M. J. D. Powell, *Updating conjugate directions by the BFGS formula*, Math. Program., 38 (1987), pp. 693–726.
[28] ———, *TOLMIN: A Fortran package for linearly constrained optimization calculations*, Report DAMTP/1989/NA2, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 1989.
[29] M. J. D. Powell and P. L. Toint, *On the estimation of sparse Hessian matrices*, SIAM J. Numer. Anal., 16 (1979), pp. 1060–1074.
[30] D. F. Shanno, *Conjugate-gradient methods with inexact searches*, Math. Oper. Res., 3 (1978), pp. 244–256.
[31] D. Siegel, *Modifying the BFGS update by a new column scaling technique*, Report DAMTP/1991/NA5, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, May 1991.

[32] ———, *Implementing and modifying Broyden class updates for large scale optimization*, Report DAMTP/1992/NA12, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, December 1992.
[33] ———, *Modifying the BFGS update by a new column scaling technique*, Mathematical Programming, 66 (1994), pp. 45–78. Ser. A.
[34] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, *Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization*, ACM Trans. Math. Software, 23 (1997), pp. 550–560.