Solving Limited-Memory BFGS Systems with Generalized Diagonal Updates

Jennifer Erway, Member, IAENG, and Roummel F. Marcia, Member, IAENG

Abstract—In this paper, we investigate a formula to solve systems of the form \((B_k + D)x = y\), where \(B_k\) comes from a limited-memory BFGS quasi-Newton method and \(D\) is a diagonal matrix with diagonal entries \(d_{ii} \geq \sigma\) for some \(\sigma > 0\). These types of systems arise naturally in large-scale optimization. We show that provided a simple condition holds on \(B_k\) and \(\sigma\), the system \((B_k + D)x = y\) can be solved via a recursion formula that requires only vector inner products. This formula has complexity \(M^2n\), where \(M\) is the number of limited-memory BFGS updates and \(n > M\) is the dimension of \(x\). We do not assume anything about the distribution of the values of the diagonal elements in \(D\), and our approach is particularly for robust non-clustered values, which proves problematic for the conjugate gradient method.

Index Terms—L-BFGS, quasi-Newton methods, diagonal modifications, Sherman-Morrison-Woodbury

I. INTRODUCTION

LIMITED-memory (L-BFGS) quasi-Newton methods are powerful tools within the field of optimization [1], [2], [3], [4] for solving problems where second derivative information is not available or computing the second derivative is too computationally expensive. In addition, L-BFGS matrices can be used to precondition iterative methods for solving large linear systems of equations. For both L-BFGS methods and preconditioning schemes, being able to solve linear systems with L-BFGS matrices are of utmost importance. While there is a well-known two-loop recursion (cf. [4], [5]) for solving linear systems with L-BFGS matrices, little is known about solving systems involving matrix modifications of L-BFGS matrices.

In this paper, we develop a recursion formula to solve linear systems with positive-definite diagonal modifications of L-BFGS matrices, i.e., systems of the form

\[ (B_k + D)x = y, \]  

where \(B_k\) is the \(k\)-th step \(n \times n\) limited-memory (L-BFGS) quasi-Newton matrix, \(D\) is a positive-definite diagonal matrix with each diagonal element \(d_{ii} \geq \sigma\) for some \(\sigma > 0\), and \(x, y \in \mathbb{R}^n\). Systems of the form (1) arise naturally in constrained optimization (see, e.g., [6], [7], [8], [9]) and are often a block component of so-called KKT systems (see, e.g., [10]). In previous work [11], we developed a direct recursion formula for solving (1) in the special case when \(D\) is a scalar multiple of the identity matrix, i.e., \(D = \alpha I\) for some constant \(\alpha > 0\), and we stated that the formula can be generalized to diagonal matrices. Here, we explicitly show how to generalize this approach to positive-definite diagonal matrices \(D\). Furthermore, we compare this generalized formula to a popular direct method (the Matlab “backslash” command) and an indirect method (the conjugate gradients). Numerical results suggest that our approach offers significant computational time savings while maintaining high accuracy.

II. THE LIMITED-MEMORY BFGS METHOD

Let \(f(x) : \mathbb{R}^n \rightarrow \mathbb{R}\) a continuously differentiable function. The BFGS quasi-Newton method for minimizing \(f(x)\) works by minimizing a sequence of convex, quadratic models of \(f(x)\). Specifically, the method generates a sequence of positive-definite matrices \(\{B_k\}\) to approximate \(\nabla^2 f(x)\) from a sequence of vectors \(\{y_k\}\) and \(\{s_k\}\) defined as

\[ y_k = \nabla f(x_{k+1}) - \nabla f(x_k) \quad \text{and} \quad s_k = x_{k+1} - x_k, \]

respectively. (See, e.g., [12] for further background on quasi-Newton methods).

The L-BFGS quasi-Newton method can be viewed as the BFGS quasi-Newton method where only at most \(M (\ll n)\) recently computed updates are stored and used to update the initial matrix \(B_0\). The number of updates \(M\) is generally kept very small; for example, Byrd et al. [1] suggest \(M \in [3, 7]\). The L-BFGS quasi-Newton approximation to \(\nabla^2 f(x)\) is implicitly updated as follows:

\[
B_k = B_0 - \sum_{i=0}^{k-1} \frac{1}{s_i^T B_i s_i} B_i s_i s_i^T B_i + \sum_{i=0}^{k-1} \frac{1}{y_i^T y_i} y_i y_i^T \]  

where \(B_0 = \gamma_k^{-1} I\) with \(\gamma_k > 0\) a constant. In practice, \(\gamma_k\) is often taken to be \(\gamma_k \approx s_{k-1}^T y_{k-1}/\|y_{k-1}\|^2\) (see, e.g., [2] or [4]). Using the Sherman-Woodbury-Morrison formula, the inverse of \(B_k\) is given by

\[
B_k^{-1} = B_{k-1}^{-1} + \frac{s_{k-1}^T y_{k-1} + y_{k-1}^T B_{k-1}^{-1} y_{k-1} - s_{k-1}^T B_{k-1}^{-1} s_{k-1}}{(s_{k-1}^T y_{k-1})^2} s_{k-1} s_{k-1}^T \]  

There is an alternative representation of \(B_k^{-1}\) from which a two-term recursion can be established:

\[
B_k^{-1} = (V_{k-1}^T V_{k-1}^T) B_{0}^{-1}(V_0 \cdots V_{k-1})^T + \sum_{j=1}^{k-1} (V_{k-1}^T V_j^T) s_{j-1} s_{j-1}^T (V_j \cdots V_{k-1})^T + \frac{1}{y_{k-1}^T s_{k-1}} s_{k-1} s_{k-1}^T. 
\]
where

\[ V_j = I - \frac{1}{y_j^T s_j} y_j s_j^T \]

(see, e.g., [5]). For the L-BFGS method, there is an efficient way for computing products with \( B_k^{-1} \). Given a vector \( z \), the following algorithm (cf. [4], [5]) terminates with \( q = B_k^{-1} z \):

**Algorithm 2.1**: Two-loop recursion to compute \( q = B_k^{-1} z \)

\[
q \leftarrow q - \alpha_i \beta_i \gamma_i \\
\text{for } i = k - 1, \ldots, 0 \\
\alpha_i \leftarrow (s_i^T q)/(y_i^T s_i) \\
q \leftarrow q - \alpha_i \beta_i \gamma_i \\
\text{end}
\]

Then, \( B_1 + D \) becomes

\[ B_1 + D = (\gamma_1^{-1} I + D) - a_0 a_0^T + b_0 b_0^T \]

We apply the SMW formula twice to compute \( (\gamma_1^{-1} I + D)^{-1} \).

More specifically, let

\[
C_0 = (\gamma_1^{-1} I + D), \\
C_1 = (\gamma_1^{-1} I + D) - a_0 a_0^T, \\
C_2 = (\gamma_1^{-1} I + D) - a_0 a_0^T + b_0 b_0^T.
\]

Applying the SMW formula yields:

\[
C_1^{-1} = C_0^{-1} + \frac{1}{1 - \text{trace}(C_0^{-1} a_0 a_0^T)} C_0^{-1} a_0 a_0^T C_0^{-1} \\
C_2^{-1} = C_1^{-1} - \frac{1}{1 + \text{trace}(C_1^{-1} b_0 b_0^T)} C_1^{-1} b_0 b_0^T C_1^{-1},
\]

giving an expression for \( (B_1 + D)^{-1} = C_2^{-1} \), as desired. This is the basis for the following recursion method that appears in [14]:

**Theorem 1.** Let \( G \) and \( G + H \) be nonsingular matrices and let \( H \) have positive rank \( M \). Let

\[ H = E_0 + E_1 + \cdots + E_{M-1}, \]

where each \( E_k \) has rank one and

\[ C_{k+1} = G + E_0 + \cdots + E_k \]

is nonsingular for \( k = 0, \ldots, M - 1 \). Then if \( C_0 = G \), then for \( k = 0, 1, \ldots, M - 1 \), the inverse of \( C_{k+1}^{-1} \) can be computed recursively and is given by

\[ C_{k+1}^{-1} = C_k^{-1} - \tau_k C_k^{-1} E_k C_k^{-1}, \]

where

\[ \tau_k = \frac{1}{1 + \text{trace}(C_k^{-1} E_k)}. \]

**In particular,**

\[ (G + H)^{-1} = C_{M-1}^{-1} - \tau_{M-1} C_{M-1}^{-1} E_{M-1} C_{M-1}^{-1}. \]

**Proof:** See [14].

We now show that applying the above recursion method to \( B_1 + D \), the product \( (B_1 + D)^{-1} z \) can be computed recursively, assuming \( \gamma_k \sigma \) is bounded away from zero. The proof follows [11] very closely.

**Theorem 2.** Let \( B_0 = \gamma_k^{-1} I \), where \( \gamma_k > 0 \), and let \( D \) be a diagonal matrix with \( d_{i, i} \geq \sigma \) for some constant \( \sigma > 0 \). The Sherman-Morrison-Woodbury (SMW) formula gives the following formula for computing the inverse of \( B + u v^T \), where \( B \) is invertible and \( u v^T \) is a rank-one update (see [13]):

\[
(B + u v^T)^{-1} = B^{-1} - \frac{1}{1 + v^T B^{-1} u} B^{-1} u v^T B^{-1} \\
= B^{-1} - \frac{1}{1 + \text{trace}(B^{-1} u v^T)} B^{-1} u v^T B^{-1}
\]

where \( u \) and \( v \) are both \( n \)-vectors. For simplicity, consider computing the inverse of an L-BFGS quasi-Newton matrix after only one update, i.e., the inverse of \( B_1 + D \). First, we simplify (2) by defining

\[
a_i = \frac{B_i s_i}{\sqrt{s_i^T B_i s_i}} \quad \text{and} \quad b_i = \frac{y_i}{\sqrt{y_i^T s_i}}.
\]

(5)
where
\[ \tau_j = \frac{1}{1 + \text{trace}(C_j^{-1}E_j)}. \] (7)

In particular, since \( C_{2k} = B_k + D \),
\[ (B_k + D)^{-1} = C_{2k-1}^{-1} - \tau_{2k-1}C_{2k-1}^{-1}E_{2k-1}C_{2k-1}^{-1}. \] (8)

Proof: Notice that this theorem follows from Theorem 1, provided we satisfy its assumptions. First, we let
\[ G = B_0 + D \quad \text{and} \quad H = E_0 + E_1 + \cdots + E_{2k-1}. \]

Note that \( G = B_0 + D \) is nonsingular because \( B_0 \) and \( D \) are positive definite and that \( G + H = B_k + D \) is also nonsingular since \( B_k \) and \( D \) are positive definite. Thus, it remains only to show that \( C_j \), which is given by
\[ C_j = G + \sum_{i=0}^{j-1} E_i = \left( B_0 + \sum_{i=0}^{j-1} E_i \right) + D, \]
is nonsingular for \( j = 1, \ldots, 2k \), for which we use induction on \( j \).

For the base case \( j = 1 \), since
\[ C_1 = C_0 - a_0a_0^T = C_0(I - a_0C_0^{-1}a_0^T), \]
the determinant of \( C_1 \) and \( C_0 \) are related as follows [15]:
\[ \det(C_1) = \det(C_0)(1 - a_0^T C_0^{-1} a_0). \]

In other words, \( C_1 \) is invertible if \( C_0 \) is invertible and \( a_0 C_0^{-1} a_0 \neq 1 \). Since \( C_0 \) is positive definite, it is therefore invertible. To show the latter condition, we use the definition of \( a_0 = B_0 s_0 / \sqrt{s_0^TB_0s_0} \) together with \( C_0^{-1} = (\gamma_k I + D)^{-1} \) to obtain the following:
\[ a_0^T C_0^{-1} a_0 = \frac{1}{s_0^TB_0s_0} \gamma_k^{-1} I + D)^{-1} s_0 \]
\[ = \frac{1}{\gamma_k s_0^TB_0s_0} \left( \sum_{i=1}^{n} \frac{1}{\gamma_k s_0^TB_0s_0} \right) \]
\[ \leq \frac{1}{\gamma_k s_0^TB_0s_0} \left( \frac{1}{\gamma_k + \sigma} \right) \]
\[ = \frac{1}{1 + \gamma_k \sigma}. \] (9)

By hypothesis, \( \gamma_k \sigma > \epsilon \), which implies that \( a_0^T C_0^{-1} a_0 < 1 \) and that \( \det(C_1) \neq 0 \). Therefore, \( C_1 \) must be invertible.

Now we assume that \( C_j \) is invertible and show that \( C_{j+1} \) is invertible. If \( j \) is odd, then \( j + 1 = 2i \) for some \( i \) and \( C_{j+1} = B_i + D \), which is positive definite because both \( B_i \) and \( D \) are positive definite. Therefore \( C_{j+1} \) must be nonsingular. If \( j \) is even, i.e., \( j = 2i \) for some \( i \), then \( C_j = B_i + D \), and
\[ C_{j+1} = C_j - a_i a_i^T = B_i - \frac{1}{s_i^TB_i s_i} B_i s_i s_i^TB_i^T + D. \]

We will demonstrate that \( C_{j+1} \) is nonsingular by showing that it is positive definite. Consider \( z \in \mathbb{R}^n \) with \( z \neq 0 \). Then
\[ z^T C_{j+1} z = z^T \left( B_i - \frac{1}{s_i^TB_i s_i} B_i s_i s_i^TB_i^T \right) z + z^T Dz \]
\[ = z^T B_i z - \left( \frac{z^T B_i s_i}{s_i^TB_i s_i} \right)^2 + z^T Dz \]
\[ = \| B_i^{1/2} z \|_2^2 - \left( \frac{\| B_i^{1/2} s_i \|_2}{\| B_i^{1/2} s_i \|_2} \right)^2 + z^T Dz \]
\[ = \| B_i^{1/2} z \|_2^2 \left( 1 - \cos^2(\angle(B_i^{1/2} z, B_i^{1/2} s_i)) \right) + z^T Dz \]
\[ \geq \sigma \| z \|_2^2 \]
\[ > 0, \] (10)

since each \( d_{i,i} \geq \sigma > 0 \).

This demonstrates that \( C_{j+1} \) is positive definite, and therefore, it is nonsingular.

We have now thus satisfied all the assumptions of Theorem 1. Therefore, \( (B_k + D)^{-1} \), which is equal to \( C_{2k}^{-1} \), can be computed recursively and is specifically given using (6) and (8).

Now, we show that computing \( C_{k+1}^{-1} z \) can be done efficiently as is done in [11]. We note that using (6), we have
\[ C_{k+1}^{-1} z = C_k^{-1} z - \gamma_k C_k^{-1} E_k C_k^{-1} z \]
\[ = \begin{cases} C_k^{-1} z + \gamma_k C_k^{-1} a_k^T C_k^{-1} z & \text{if } k \text{ is even} \\ C_k^{-1} z - \gamma_k C_k^{-1} b_k^T C_k^{-1} z & \text{if } k \text{ is odd} \end{cases} \] (11)
We define $p_k$ according to the following rules:

$$p_k = \begin{cases} C_k^{-1} a \frac{z}{k} & \text{if } k \text{ is even} \\ C_k^{-1} b \frac{z}{k} & \text{if } k \text{ is odd} \end{cases} \quad (12)$$

Thus, (11) simplifies to

$$C_{k+1}^{-1} z = C_k^{-1} z + (-1)^k \tau_k (p_k^T z) p_k.$$  

Applying this recursively to $C_1^{-1} z$ yields the following formula:

$$C_{k+1}^{-1} z = C_k^{-1} z + \sum_{i=0}^k (-1)^i \tau_i (p_i^T z) p_i, \quad (13)$$

with $C_0^{-1} z = (\gamma_1^{-1} I + D)^{-1} z$. Thus, the main computational effort in forming $C_k^{-1} z$ involves the inner product of $z$ with the vectors $p_i$ for $i = 0, 1, \ldots, k$.

What remains to be shown is how to compute $\tau_k$ in (7) and $p_k$ in (12) efficiently. The quantity $\tau_k$ is obtained by computing $\text{trace}(C_k^{-1} E_k)$, which after substituting in the definition of $E_k$, is given by

$$\text{trace}(C_k^{-1} E_k) = \begin{cases} -a_k^T C_k^{-1} a \frac{z}{k} & \text{if } k \text{ is even} \\
\frac{b_k^T}{k} C_k^{-1} b \frac{z}{k} & \text{if } k \text{ is odd} \end{cases}.$$  

using (12). Thus, $\tau_k$ simplifies to

$$\tau_k = \begin{cases} \frac{1}{k} & \text{if } k \text{ is even} \\
1 - \frac{a_k^T p_k}{k} & \text{if } k \text{ is odd} \end{cases}.$$  

Finally, notice that we can compute $p_k$ in (12) by evaluating (13) at $z = a \frac{z}{k}$ or $z = b \frac{z}{k}$:

$$p_k = \begin{cases} C_0^{-1} a \frac{z}{k} + \sum_{i=0}^{k-1} (-1)^i \tau_i (p_i^T a \frac{z}{k}) p_i & \text{if } k \text{ is even} \\
C_0^{-1} b \frac{z}{k} + \sum_{i=0}^{k-1} (-1)^i \tau_i (p_i^T b \frac{z}{k}) p_i & \text{if } k \text{ is odd} \end{cases}.$$  

Thus, by computing and storing $a_k^T p_k$ and $b_k^T p_k$, we can form $\tau_k$ and $p_k$ rather easily. The following pseudocode summarizes the algorithm for computing $C_{k+1}^{-1} z$:

**Algorithm 3.1:** Proposed recursion to compute $q = C_{k+1}^{-1} z$.

$$q \leftarrow [(\gamma_1^{-1} I + D)^{-1} z];$$  

for $j = 0, \ldots, k$

if $j$ even

$\quad c \leftarrow a_{j/2};$

else

$\quad c \leftarrow b_{(j-1)/2};$

end

$p_j \leftarrow (-1)^j c p_j c$;

for $i = 0, \ldots, j - 1$

$p_j \leftarrow p_j + (-1)^i v_i (p_i^T c) p_i;$

end

We implemented the proposed method in Matlab on a Two 2.4 GHz Quad-Core Intel Xeon “Westmere” Apple Mac Pro and compared it to a direct method using the Matlab “backslash” command and the built-in conjugate-gradient (CG) method (pcg.m). Because of limitations in memory, we were only able to use the direct method for problems where $n \leq 20,000$. For our numerical experiments, we let the entries in $D$ to be evenly distributed between 1 and $n/10$. The relative residuals for the recursion formula are used as the criteria for convergence for CG. In other words, the time reported in this table reflects how long it takes for CG to achieve the same accuracy as the proposed recursion method. Often, CG terminated without converging to the desired tolerance because the method stagnated (flag = 3 in Matlab).

### Results

The three methods were run on numerous problems with varying problems sizes. Tables I–IV show the time and the relative residuals for each method. In Tables I and II, $n$ ranges from 1,000 to 20,000; Tables III and IV contain results for $n \geq 20,000$. We note that all methods achieve very small relative residual errors for each of the problems we considered. Besides from memory issues, the direct method suffers from significantly longer computational
that requires at most \( M^2 \) vector inner products. (Note: We assume \( M \ll n \), and thus, \( M^2 \) is also significantly smaller than \( n \).) Numerical results demonstrate that the proposed method is not only accurate (the resulting relative residuals are comparable to the Matlab direct solver and the conjugate gradient method), but it is also efficient with respect to computational time (often up to about 50 times faster than CG). The algorithm proposed in this paper can be found at [http://www.wfu.edu/~erwayjb/software](http://www.wfu.edu/~erwayjb/software).

V. CONCLUSION

In this paper, we proposed an algorithm based on the SMW formula to solve systems of the form \( B_k + D \), where \( B_k \) is an \( n \times n \) L-BFGS quasi-Newton matrix. We showed that as long as the diagonal elements of \( D \) are bounded away from zero, the algorithm is well-defined. The algorithm requires at most \( M^2 \) vector inner products. (Note: We assume that \( M \ll n \), and thus, \( M^2 \) is also significantly smaller than \( n \).) Numerical results demonstrate that the proposed method is not only accurate (the resulting relative residuals are comparable to the Matlab direct solver and the conjugate gradient method), but it is also efficient with respect to computational time (often up to about 50 times faster than CG). The algorithm proposed in this paper can be found at [http://www.wfu.edu/~erwayjb/software](http://www.wfu.edu/~erwayjb/software).

References


