A Preconditioned L-BFGS Algorithm with Application to Molecular Energy Minimization *

Lianjun Jiang   Richard H. Byrd   Elizabeth Eskow   Robert B. Schnabel

Nov. 21, 2004

Technical Report CU-CS-982-04
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

The limited-memory BFGS method has been widely used in large scale unconstrained optimization problems, such as the protein structure prediction problem. A major weakness of the L-BFGS method is that it may converge very slowly for ill-conditioned problems. We propose a preconditioned L-BFGS method, where we form the preconditioner from parts of the partially separable objective function. We report results of experiments in the context of the protein structure prediction problem for four different proteins, using a protein energy model as the objective function and multiple initial configurations for each protein. The results show speed-ups with factors between 3 and 10 in terms of function evaluations and with factors between 2 and 7 in terms of CPU time. The difference between CPU time and function evaluation speed-up is due to the extra overhead of calculating and applying the preconditioner. We also compare the performance of this method to the preconditioned truncated Newton method.

1 Introduction.

In the field of large scale unconstrained optimization, one of the most versatile, effective and widely used classes of methods are limited-memory quasi-Newton methods. These are variants of quasi-Newton methods that do not require calculation or storage of a full Hessian matrix, which can be extremely expensive for many large scale problems. Limited memory methods are often very effective, but they can be slow and sometimes have limited accuracy, especially for ill-conditioned problems. Another class of methods, truncated Newton methods, are similar to limited memory methods in terms of storage costs and computational overhead, and it is well known that truncated Newton methods can be greatly improved by preconditioning the inner conjugate gradient iteration. For limited memory methods, it is possible to perform a procedure

*Supported by National Science Foundation Grants CHE-0205170 and CCR-0290190 and Army Research Office Grant DAAD19-02-1-0407

analogous to preconditioning, although there is much less computational experience with this procedure.

In this paper, we examine some ways to precondition limited memory methods, focusing on problems arising in predicting protein structure by potential energy minimization. Our experimental results, with several proteins, show that preconditioning the limited memory update significantly improves the performance of the method, and that this improvement can be enhanced by making advantageous choices in the implementation. The preconditioning considered here is very similar to that considered by Xie and Schlick in [17], where they applied the preconditioned truncated-Newton method to minimize the potential energy of proteins. They also did a performance comparison between preconditioned L-BFGS and truncated-Newton. Their results showed that the preconditioned L-BFGS reduced the number of iterations, but not significantly enough to balance the cost of preconditioner calculation overhead. Our method forms the preconditioner differently, which leads to much greater reduction in function evaluation, and thus makes the preconditioning extremely effective. Furthermore, we predict that the benefit of using a preconditioner will increase with the size of the problem.

1.1 Limited-Memory BFGS Method

The limited-memory BFGS method belongs to a class of methods called limited-memory quasi-Newton methods. This class of methods is especially useful in solving large scale problems with Hessian matrices that are too expensive to compute or too dense to store and manipulate easily. The major advantage is that these methods don't require the storage of the $N \times N$ full Hessian matrix, where N is the number of variables; instead they only store $2m$ vectors of length N. Usually, these vectors are applied to update a simple initial matrix, such as a multiple of the identity, to form an approximation to the current Hessian. The same idea can be applied to the inverse of the Hessian. Suppose at step k, we have stored m most recent correction pairs $(s_i, y_i), i = k-m, \ldots, k-1$, where

$$s_i = x_{i+1} - x_i, \tag{1.1}$$

$$y_i = \nabla f(x_{i+1}) - \nabla f(x_i) \tag{1.2}$$

We can construct an approximation to the inverse of $\nabla^2 f_k$ by first selecting an initial inverse Hessian approximation H_k^0, and then applying the update pairs to it:

$$H_k^1 = update(H_k^0, s_{k-m}, y_{k-m}) \tag{1.3}$$

$$\vdots$$

$$H_k^{j+1} = update(H_k^j, s_{k-m+j}, y_{k-m+j}) \tag{1.4}$$

Finally we set $H_k = H_k^m$, and use H_k to find the new direction p_k:

$$p_k = -H_k * \nabla f(x_k) \tag{1.5}$$

The next iteration is then given by

$$x_{k+1} = x_k + \alpha p_k \tag{1.6}$$

where α is chosen by a line search method.

This approach works with a variety of update formulas. The most widely used in this context is the BFGS inverse updating formula, given by:

$$update(H, s, y) = V^T H V + \rho s s^T, \tag{1.7}$$

where

$$\rho = 1/y^T s, \ \ V = I - \rho y s^T. \tag{1.8}$$

The matrix H_0^k in (1.3) should be a rough approximation to the inverse of the real Hessian, and usually is given by:

$$H_k^0 = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I. \tag{1.9}$$

Previous studies have shown that $3 \leq m \leq 7$ is a reasonable choice, and increasing m doesn't necessarily imply improved performance. The main weakness of the L-BFGS method is that it may converge very slowly in terms of number of iterations for ill-conditioned problems, which will require a large number of potentially costly function evaluations to be performed. Our goal is to find better choices of H_k^0 that improve the speed of the method.

1.2 The protein structure prediction problem

We will discuss ways to compute an initial H_k^0 in the L-BFGS method specifically in the context of energy minimization for protein structure prediction, which is to predict the 3-dimensional structure, or native state, of a protein, given its amino acid sequence. It is believed that, in most cases, the native state corresponds to the minimum free energy of the protein. However, the energy landscape of a realistic-sized protein has thousands of parameters and an enormous number of local minimizers. This means that an efficient, large-scale global optimization method is required to solve the problem. These global optimization approaches, including the one developed by our group [10, 6], often rely heavily on local minimizations in the full parameter space (and possibly smaller sub-spaces) of the problem. As this is a major cost in the global optimization method, reducing that cost enables these expensive problems to be worked on much more efficiently.

Our global optimization approach is based on the use of an empirical energy potential, AMBER [4] , which is one of several commonly used potentials for proteins. E_{AMBER} represents the N Cartesian coordinates of the n atoms of the protein as a vector of length $N = 3n$, and approximates the potential energy by a function of the form,

$$E_{\text{AMBER}} = \sum_{\text{bonds}} K_{r_i}(r_i - r_{i,eq})^2 + \sum_{\text{angles}} K_{\theta_\ell}(\theta_\ell - \theta_{\ell,eq})^2 + \sum_{\text{dihedrals}} \frac{V_k}{2}[1 + \cos(n_k\phi_k - \gamma_k)] + \sum_{i<j}\left(\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - 2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + C\frac{q_i q_j}{r_{ij}}\right). \tag{1.10}$$

The energy function contains two types of interactions: bonded (the first three terms) and non-bonded (the last two). The first term, the bond length potential, is a sum over all the bonds between neighboring atoms, and depends on the deviation of the distance $||r_i||$ from its equilibrium value. For each three atoms such that two atoms are both bonded to the third atom, the second sum has a term depending on the bond angle θ formed by the three atoms. The last bonded summation includes terms depending on certain dihedral angles ϕ formed by sets of four atoms connected by bonds. The dihedral angle is formed by two planes each containing a specified set of three of the four atoms, and is a measure of the torsion of the configuration. The non-bonded interactions are comprised of Lennard-Jones and electrostatic terms, respectively. These non-bonded interactions occur between every pair (i, j) of atoms and depend on the distance r_{ij} between the pair of atoms, and on their charges $q_i q_j$.

An additional component of the energy model that is necessary for a good model is an empirical solvation free energy term used to model the interaction of the protein with an aqueous environment. We use a term, $E_{SOLVATION}$, which models the hydrophobic (adverse to interactions with water) effects as a two-body interaction between certain atoms of the protein, using a sum of Gaussians.

$$E_{\text{SOLVATION}} = \sum_{i,j \leq N_c} \sum_{k \leq 3} h_k \exp\left(- \left[\frac{(r_{ij} - c_k)}{w_k} \right]^2 \right), \tag{1.11}$$

where the sum over i and j is over the aliphatic carbon centers, and each of the 3 Gaussians is parameterized by position (c_k), depth (h_k), and width (w_k) so as to describe the minima and barrier of the hydration energy [10]. The total energy that we minimize is thus $E_{AMBER} + E_{SOLVATION}$.

2 Preconditioned L-BFGS Method

The performance of the L-BFGS method relies upon having a good approximation to the actual Hessian. The correction pairs are used to correct the behavior of H_k^0. So if we start with a better H_k^0, we should expect a better approximation to the actual $\nabla^2 f^{-1}(x_k)$, and hopefully, a more quickly convergent L-BFGS method.

2.1 The algorithm

The L-BFGS algorithm does not need to form H_k explicitly. It only needs the vector $H_k \nabla f(x_k)$ at step k. This can be computed by a two-loop recursion algorithm:

$$\begin{aligned}
&q = \nabla f(x_k) \\
&\text{For } i = m-1, \ldots, 0 \\
&\left[\begin{array}{rcll} \alpha_i & = & \rho_i s_i^T q & \quad (\text{store } \alpha_i) \\ q & := & q - \alpha_i y_i & \end{array} \right. \\
&\qquad\quad r = H_k^0 q \\
&\text{For } i = 0, 1, \ldots, m-1 \\
&\left[\begin{array}{rcl} \beta & = & \rho_i y_i^T r \\ r & := & r + s_i(\alpha_i - \beta) \end{array} \right. \\
&H_k \nabla f(x_k) = r \ ,
\end{aligned} \tag{2.12}$$

which is described in [12].

This two-loop recursion algorithm is an inexpensive computation, costing only $4mN$ multiplications and additions, plus one product with H_k^0 in (2.12). It also has the advantage that the computation involving H_k^0 is isolated from the rest of the computations. Therefore one can chose H_k^0 freely at each L-BFGS iteration. Of course if we choose an initial matrix more complex than a diagonal it is more practical to choose a matrix M_k that is like $\nabla^2 f(x_k)$ in some way and replace (2.12) with the solution of the linear system:

$$M_k r = q. \tag{2.13}$$

It is worth noting that using M_k as the initial matrix in this way is mathematically equivalent to making the change of variables $x \to M_k^{\frac{1}{2}} x$, and then applying the correspondingly transformed updates to an initial matrix equal to the identity. Therefore, it is appropriate to refer to this approach as a *preconditioned* limited memory method.

For this approach to be appealing, M must have the following properties:

1. M must be inexpensive to calculate.
2. M must allow equation (2.13) to be solved easily.
3. M must be positive definite.
4. M should be a significant part of the real Hessian.

We will discuss how we resolve these issues in the following sections. Even though we are using the protein structure prediction problem as our example, the approaches we take should be applicable to other large scale problems.

2.2 Preconditioner calculation

From the equation (1.10), the potential energy can be separated into two parts; bonded terms (local terms) and non-bonded terms. Specifically, the bonded terms include the first three sums of the equation(1.10). Correspondingly the Hessian may be serarated into

$$\nabla^2 f(x) = \nabla^2 f_B + \nabla^2 f_{NB} \tag{2.14}$$

We will form the preconditioner M to approximate each part of the Hessian respectively. Since the bonded term only involves adjacent atoms, the Hessian $\nabla^2 f_B$ is expected to be banded and sparse, if the variables are ordered properly. Therefore it is reasonable to include $\nabla^2 f_B$ in M, and we do so. On the other hand, $\nabla^2 f_{NB}$ is dense, and expensive to compute. Rather than ignoring $\nabla^2 f_{NB}$ entirely, we initially approximate $\nabla^2 f_{NB}$ by term of the form βI. Because

$$\nabla^2 f_{NB} \approx y - \nabla^2 f_B s \tag{2.15}$$

we let

$$\beta = \frac{||y - \nabla^2 f_B\, s||}{||s||} \tag{2.16}$$

We then form the preconditioner M by

$$M = \nabla^2 f_B + \beta I \tag{2.17}$$

Including the βI term in M makes a significant improvement in the performance of the algorithm, as we show in Section 3.

The Hessian $\nabla^2 f_B$ can be calculated either analytically or numerically. Two common approaches to obtain a numerical Hessian are finite differences and automatic differentiation. Both techniques require in general O(N) gradient evaluations. However, since our preconditioner is sparse, one can apply a coloring algorithm [5], to figure out the independent columns of the Hessian, and thus greatly reduce the cost of both methods. For example, since $\nabla^2 f_B$ is banded, the number of coloring groups is dependent on the bandwidth. We found that no matter how big the protein is, the upper bound for the number of colors is 66. That is, rather than needing N gradient evaluations to calculate $\nabla^2 f_B$, it can be calculated in 66 gradient evaluations. In our experiments, we used the automatic differentiation program ADIFOR [1] to automatically generate the Hessian code from the gradient code for bonded terms, because ADIFOR is more accurate than the finite difference method.

This preconditioner has the properties 1 and 4 mentioned above. Furthermore, since the preconditioner should vary slowly between iterations near the solution, we also suggest that M only be calculated at every Tth L-BFGS iteration. For the next $T-1$ iterations, we continue to use the same M for equation (2.13). This allows the total cost of calculating the preconditioner to be reduced by a factor of T. The effect of using different T values is shown in Section 3.

2.3 Modification to the preconditioner

The matrix M formed by (2.17) is not guaranteed to be positive-definite, which is critical for the L-BFGS method to generate decent directions. The preconditioner may need to be modified to ensure positive definiteness. One way to do this is to use a modified Cholesky factorization to form the triangular factor L such that

$$LL^T = M + E \quad \text{where } E \text{ is a diagonal matrix.} \tag{2.18}$$

Sparsity can be be maintained by using symmetric pivoting. Another approach is to use an incomplete Cholesky factorization, such that the amount of fill-in occurring in L is limited, and many of the nonzero elements of L in (2.18) are suppressed.

Using either approach allows the preconditioner to satisfy requirement 3. Furthermore if $||E||$ is small, then property 4 also is maintained. Both modified Cholesky and incomplete Cholesky can be implemented to keep a certain degree of sparsity in L. This limits the cost of the back-solving, which allows the preconditioner to retain property 2. As discussed below we can further control this cost by reusing the preconditioner and its factorization for several iterations.

The code MA57 [9] uses a version of the modified Cholesky method of Schnabel and Eskow [16] to determine the diagonal modification. Sparsity is preserved by a pivoting strategy. The incomplete Cholesky factorization code, ICF, by Lin and Moré [3] restricts fill-in explicitly by allowing no more than pn elements to fill in, where p is user-defined. It uses the following algorithm to find α such that $M + \alpha D$ is positive definite.

Choose $\alpha_s > 0$
Compute $\hat{M} = D^{-1/2} M D^{-1/2}$ where $D = diag(||Me_i||_2)$
Set $\alpha_0 = 0$ if $\min(\hat{m}_{ii}) > 0$; otherwise $\alpha_0 = -\min(\hat{m}_{ii}) + \alpha_s$
For k=0,1,. . . ,
    Use incomplete Cholesky factorization on $\hat{M}_k = \hat{M} + \alpha_k I$;
    If successful set $\alpha = \alpha_k$ and exit
    Else set $\alpha_{k+1} = max(2\alpha_k, \alpha_s)$

A drawback of this approach is that it may need to run the factorization algorithm multiple times before determining the correct α. In these problems, since the $\nabla^2 f_B$ is banded, there is little fill-in to the factor L, so we set p large enough to keep all the entries in L.

MA57 is a sparse solver which performs pivoting to maintain sparsity, and then applies the modified Cholesky without pivoting to achieve positive-definiteness. Since ICF may need to run multiple times to figure out the proper α, it is more expensive then MA57. However, since MA57 uses modified Cholesky without pivoting to maintain sparsity, the final E in (2.18) is much larger than that computed by ICF, on average. The larger modification makes the preconditioner a worse approximation to the Hessian and thus leads to more iterations. The experimental results on the performance of these two approaches are shown in the Section 3.1. We concluded that incomplete Cholesky is the better choice for our problem, at least until a version of modified Cholesky is developed that balances pivoting for sparsity with pivoting to minimize the size of E.

2.4 Line search modification

We are using the line search algorithm by Moré and Thuente [13] in our L-BFGS code. During out initial experiments, there were several line search failures. These appear to be caused by very high objective function values near poles of the objective. The problem is that the new upper bound u of the line search interval $[l, u]$ is sometimes set equal to α_c, the minimizer of a cubic interpolant. If one of the interpolating values is extremely large, this sometimes shrinks too close to the lower bound l, causing the line search to fail. We added a safeguard parameter *minshrink* to prevent this from happening, and compute the new upper bound by the formula:

$$u \to \max(l + minshrink * (u - l), \alpha) \tag{2.19}$$

Setting *minshrink* to 0.001 eliminates all the line search failures which occurred in our experiments. We later discovered that Xie and Schlick had also observed this problem and suggested the same solution in [18].

3 Computational results

We performed our experiments on a cluster of 65 nodes, each containing two Intel Xeon 2GHz CPUs and 2 GB memory. We selected 4 different proteins with between 1779 and 13728 variables to do experiments on. We denote these proteins by P1(SH3 prototype 1E0M), P2(HI0073 H. influenzae 1N05), P3(HI1034, H. influenzae 1IN0) and P4(F-actin capping protein a-1 subunit, chicken 1IZN) [11]. Proteins P2, P3, and P4 were targets T130, T148, and T162 from the CASP5 competition. Several different starting points were used; however, due to the extreme nonlinearity of the energy function, and the number of different local minima, it is very common for different versions of a method with the same starting point to converge to different local minimizers. We can get two strategies to converge to the same minimizer if the starting point is sufficiently good (i.e. $\nabla f(x)$ is small), and this provides the most precise comparisons of the strategies. However, in practice much poorer initial points are used, and it would be more realistic to use many far starting points and average the behavior for a given strategy.

To resolve this dilemma, we prepared two test beds for each protein. The first one contains starting points which will converge to the same minimizer with respect to different methods. These starting points are already close to the solution, in fact close enough that further minimization produces changes in the objective in the third decimal digit, or smaller. However, this does allow us to compare the various methods as they approach the same minimum. In this set, we used 16 starting points for P1 and P2, 8 starting points for P3 and only 4 starting points for P4. The second test bed contains 16 starting points which are far from the solutions, and may therefore converge to different minima. All the starting points were intermediate points from our global optimizer described in [10]. For each experiment we computed the average run statistics over the starting points. All the methods use the following stopping condition:

$$\frac{||\nabla f(x_k)||}{\max(1.0,\ ||x_k||)} < \varepsilon, \quad \varepsilon = 1.0E-5 \tag{3.20}$$

3.1 Preconditioner factorization and positive definiteness

As discussed above, we tried two codes for sparse matrix factorization, the positive definite modified Cholesky option of MA57 and the incomplete Cholesky factorization code of Moré. For all of our tests, ICF resulted in smaller quantities added to the diagonal and fewer P-LBFGS iterations. We believe this is due to the more expensive method used by ICF for computing the multiple of the identity to add. Xie and Schlick [17], working with truncated Newton, also observed that several different modified Cholesky approaches all add too much to the original preconditioner. The following table shows the results from experiments on target protein P2 using the close starting points.

method	iter	nfg	time	add-on	factor time
ICF	337.38	454.08	280.05	20.44	6.62
MA57	500.38	720.77	395.30	2194.32	3.27

Table 1: MA57 vs ICF. Values are averages of number of iterations, number of function and gradient evaluations and CPU time in seconds for the run. The column add-on is the average value of the maximum amount added to the diagonal, and factor time is the time in seconds for each factorization and back-solve.

The modification made by MA57 clearly is much larger than that made by ICF. It appears that ICF generates a better preconditioner, which leads to convergence in fewer P-LBFGS iterations. Since the cost of either factorization is relatively small we use ICF in all subsequent experiments.

3.2 Time break down for the algorithm

One strength of the L-BFGS method is the low cost of the linear algebra for each iteration. The cost of the standard L-BFGS method on these problems is dominated by the function and gradient evaluation cost. However, in the preconditioned L-BFGS method, two other costs are involved:

preconditioner calculation and using the preconditioner (factoring and solving). Table(2) shows the time (in seconds) for each operation for different problem sizes, based on average values over a single run.

Targets	N	Nonzero(%)	fg eval	Adifor	ICF
P1	1779	1.77	0.038	0.15	0.03
P2	5676	0.57	0.438	0.52	0.15
P3	7944	0.40	0.820	0.72	0.23
P4	13728	0.23	2.62	1.26	0.24

Table 2: Time break down in seconds. N is the number of variables.Nonzero is the nonzero percentage in the preconditioner. fg evals is the time for each function and gradient evaluation. Adifor is the time for each preconditioner calculation using Adifor. ICF is the time for each factorization and back-solve.

It is clear from Table 2 that function and gradient evaluation is an $O(n^2)$ operation, so as the problem size doubled, the function and gradient evaluation cost quadrupled. However, the preconditioner calculation cost is increasing linearly, so the ratio of evaluation cost to preconditioner cost will increase as the problem size grows. In other words, the extra overhead we put into the preconditioned L-BFGS method will be less significant as the problem size increases. The factorization cost is clearly a minor cost compared to other two major costs. Hence for this problem, it makes sense to choose ICF over MA57, even though ICF requires more factorizations than MA57, because it turns out to save P-LBFGS iterations. However, in a different context, if factorization cost were more significant than the function and gradient evaluation cost, then MA57 might be a better choice. Finally, using the ratio of evaluation cost to preconditioner cost, one can easily calculate the desired iteration reduction to make P-LBFGS a useful method. For example, let n_1 = number of iterations to solve using L-BFGS, n_2 = number of iterations to solve using P-LBFGS, then for P1, P-LBFGS will be a better method only if:

$$\frac{n_2}{n_1} < \frac{Cost_{fgeval}}{Cost_{fgeval} + Cost_{adifor} + Cost_{ICF}} = \frac{0.038}{0.038 + 0.1511 + 0.03} = 0.174 \tag{3.21}$$

ie, a speed up of about factor 5.7 in terms of iterations. For P4, factor (3.21) is 0.636, thus requiring a speed-up of 1.6 in iterations. Fortunately, as shown in the next section, this factor of speed-up is commonly achieved, and thus makes P-LBFGS a competitive method.

3.3 Performance results

Now we consider a straightforward comparison between the performance of the standard L-BFGS method and two preconditioned versions. Table 3 shows results for the standard method and for L-BFGS with preconditioners based on a positive definite modification of $\nabla^2 f_B$ alone, and based on a positive definite modification of (2.17) (P-LBFGS). For all these tests, we set the number of correction pairs, $m = 30$, and we compute the preconditioner at each iterate. The following two sections will show that greater improvements result from adjusting these parameters. For P1 with close starting points, the $\frac{n_2}{n_1}$ ratio is certainly much smaller than 0.174, and thus we have a factor of two speed up in time. For the bigger problem, the function and gradient evaluation costs become more dominant, and even though we see a bigger $\frac{n_2}{n_1}$ ratio, the time reduction becomes greater still, rising to nearly a factor of 3.

Preconditioning by (2.17), which includes the approximation βI to the non-bonded terms, compared to preconditioning by $\nabla^2 f_B$ alone results in an additional reduction of more than 25% in time and iterations for the close starting points and substantial but smaller reductions for the farther start points. In [17]), Xie and Schlick use only $\nabla^2 f_B$ in their preconditioned L-BFGS tests, and this difference appears to be part of the reason that our preconditioning results in a net time saving, while in [17] the preconditioning results in more computational time. Another factor is the use of a modified Cholesky to make the preconditioner positive definite in [17], as opposed to the more accurate and expensive approach used here.

Targets(Close):	P1			P2			P3			P4		
Methods	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
L-BFGS	1758	1804	78.2	2537	2598	1124.7	3098	3168	2610.1	4239	4270	11424
L-BFGS w/$\nabla^2 f_B$	157	175	46.2	409	467	564.0	621	822	1418.9	887	1063	4698.5
P-LBFGS	119	143	35.9	295	389	442.9	451	613	1062	645	988	4039
Targets(Far):	P1			P2			P3			P4		
Methods	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
L-BFGS	21016	21433	962.0	19187	19558	8623.7	18174	18598	15778.4	43469	44282	117472.0
L-BFGS w/$\nabla^2 f_B$	2118	2949	598.3	3119	4524	4390.0	3204	5033	7627.2	7644	10868	42862.4
P-LBFGS	1765	2462	489.5	2425	3576	3397.3	2985	4368	6628.1	6792	10226	39247.1

Table 3: Performance results for plain L-BFGS, and L-BFGS with two preconditioners. Shown for each problem are: the number of L-BFGS iterations, the number of function and gradient evaluations, and the total CPU time in seconds for the method to converge.

3.4 Preconditioner recalculation interval

Examination of Table 3 indicates that the reduction in time between L-BFGS and P-LBFGS is not as sharp as the reduction in number of iterations, and from Table 2 it is clear that this difference is due to the cost of computing the preconditioning matrix at each iteration. One way to reduce the cost of computing $\nabla^2 f_B$ is not to compute it at each iteration, but to use the matrix $\nabla^2 f_B(x_k)$ as a preconditioner at iterates $x_{k+i}; i = 0, \ldots, T-1$. Thus with a *recalculation interval* of T we only compute $\nabla^2 f_B$ every T iterations. The results of our experiments with this strategy are summarized in Table 4 below.
From these results, we make the following observations:

- increasing T can sometimes make the preconditioner less accurate and may increase the number of iterations. This is especially evident for the close starting points, but the effect is not large or consistent.
- Increasing T results in the performance of much fewer preconditioner calculations, This outweighs any increase in number of iterations, and significantly reduces total computational time.
- Our results indicate that for problems of the sizes used in our experiments, $T = 20$ is a reasonable value, but performance is not extremely sensitive to this parameter.

3.5 Number of saved update pairs m

Previous study has suggested that increasing m past about 30 may not improve the performance of the L-BFGS method (see, e.g. [12]). Table 5 shows the performance results for the preconditioned

Targets(Close)	P1			P2			P3			P4		
T	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
1	119	143	35.9	295	389	442.9	451	613	1062.0	645	988	4039.0
3	132	163	21.7	331	442	331.4	528	717	878.7	642	955	3192.0
10	131	161	15.3	337	454	280.1	536	740	773.3	603	943	2889.7
20	140	171	14.5	333	459	270.3	517	732	738.7	569	889	2719.9
30	155	197	15.3	349	457	266.2	571	805	792.5	600	958	2857.9

Targets(Far)	P1			P2			P3			P4		
T	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
1	1765	2462	489.4	2425	3576	3396.9	2985	4368	6612.2	6792	10226	39247.1
3	1497	2104	204.9	2226	3222	2016.2	2870	4185	4551.0	5366	7998	24674.2
10	1535	2179	138.0	2295	3469	1757.4	2957	4376	4080.3	5919	9242	25889.7
20	1422	2072	115.4	2294	3555	1711.1	3214	4790	4292.5	5411	8393	23094.8
30	1485	2238	118.6	2171	3373	1605.7	2799	4241	3776.6	5960	9120	24895.2
60	1401	2289	114.5	2448	4015	1859.0	2925	4565	4003.2	5296	8730	23654.5

Table 4: Varying the recalculation interval T

version on P1-P3 for different m values:

Targets(Close)	P1			P2			P3		
M	iter	nfg	time	iter	nfg	time	iter	nfg	time
1	156	186	42.9	525	646	686.4	1344	1714	2779.9
3	159	227	45.5	461	706	683.9	866	1292	2013.5
5	156	186	42.8	405	636	620.2	605	903	1446.9
10	130	180	39.0	367	548	558.1	509	725	1212.0
30	119	143	35.9	295	389	439.5	451	613	1062.0
60	118	134	35.8	303	367	439.3	492	659	715.0
120	111	122	33.9	267	302	388.5	521	686	739.6
180	103	112	31.7	258	282	373.6	420	543	610.8
240	101	110	31.3	253	272	368.0	424	551	621.5

Targets(Far)	P1			P2			P3		
M	iter	nfg	time	iter	nfg	time	iter	nfg	time
1	5049	5909	1235.4	7924	9616	9592.8	15237	20498	31416.5
3	2307	3168	610.4	4030	6187	5554.1	7539	12915	17877.5
5	2000	2706	530.0	3291	5007	4573.9	6035	9610	13775.0
10	1826	2441	482.2	2844	4277	3948.7	4580	6822	10180.9
30	1765	2462	473.6	2425	3576	3397.3	3142	4567	6946.0
60	1611	2484	449.3	2201	3184	3062.7	2351	3363	5273.6
120	1658	2868	486.2	1965	3066	2884.8	2392	3583	5521.4
180	1668	2981	496.9	2024	3127	3011.4	2411	3747	5739.2
240	1633	2957	492.8	1941	3181	2970.1	2248	3529	5455.0
300	1647	2999	513.0	1935	3222	3101.7	2308	3671	5803.0

Table 5: Varying the number of saved update pairs m for P-LBFGS

It is clear that performance of P-LBFGS improves steadily as we increase m up to about 240.

The number of iterations decreases with m, and the extra computational cost is small enough that computational time decreases also. (Performance improves with m for the unpreconditioned LBFGS also, but the improvement is less consistent.) Since performance stops improving in this examples for m around 240, we use the value $m = 240$ in the rest of the paper.

3.6 Performance gain

Based upon the results of the previous two sections, we conclude that $T = 20$ and $m = 240$ are good overall choices for P-LBFGS. We compared P-LBFGS with these settings to the unpreconditioned LBFGS. For plain LBFGS, we either used the previous value $m = 30$ or $m = 240$ whichever was better. (We did notice the unpreconditioned method getting better times for $m = 1$ for close starting points on two problems, but this appears to be an artifact of the stopping tolerance, as the function values were higher than with $m = 30$.) The following table shows the results of this comparison.

	P1			P2			P3			P4		
targets(C)	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
L-BFGS	1758	1804	78.2	2537	2598	1124.7	3098	3168	2610.1	4239	4370	11424
P-LBFGS	106	117	12.0	266	293	199.7	358	418	482.0	468	570	1880.7
ratio	16.53	15.40	6.56	9.54	8.88	5.63	8.67	7.59	5.42	9.06	7.67	6.07
	P1			P2			P3			P4		
targets(F)	iter	nfg	time	iter	nfg	time	iter	nfg	time	iter	nfg	time
L-BFGS	12473	12575	686.7	14879	15020	7086.3	15698	15845	13927.5	32783	33096	89952.2
P-LBFGS	1751	3291	188.3	2081	3499	1730.8	2351	3879	3585.7	4344	7324	20456.9
ratio	7.12	3.82	3.65	7.15	4.29	4.09	6.68	4.09	3.88	7.55	4.52	4.40

Table 6: Performance gain

The difference between the nfg ratio and time ratio reflects the overhead of the preconditioned L-BFGS method. Note that, these two ratios are much closer than in the results shown in Table 3, a result primarily of using a larger recalculation interval.

3.7 Comparison with truncated Newton method

Another method used very commonly for molecular energy minimization is the truncated-Newton method. This method computes an inexact solution to the classic Newton equation:

$$\nabla^2 f(x_k) p_k = -\nabla f_k \tag{3.22}$$

Normally the method uses an iterative solver, and terminates at

$$||\nabla^2 f(x_k) p_k + \nabla f(x_k)|| \leq \eta_k ||\nabla f(x_k)||, \tag{3.23}$$

or because of an inner iteration limit. The most common choice of the iterative solver is the conjugate gradient method, which doesn't require the computation of $\nabla^2 f(x_k)$, but only requires the Hessian-vector product. This can be computed either analytically or simply by finite differencing:

$$\nabla^2 f(x) d \approx \frac{\nabla f(x + \varepsilon d) - \nabla f(x)}{\varepsilon} \tag{3.24}$$

In the CG method, preconditioning is crucial to improve the speed of the convergence. Schlick and Fogelson have implemented a preconditioned truncated Newton package TNPACK, and we have made some comparisons with PLBFGS. Following [17], we used the UMC option in TNPACK and the more lenient line search option [18] to get the best performance of TNPACK. TNPACK also allows the user to set the maximum number of conjugate gradient iterations allowed per outer iteration. This parameter, called here $maxcg$, can have a significant effect on efficiency and can be difficult to determine. Xie and Schlick suggest that the optimal value for $maxcg$ is problem dependent, and that a bad $maxcg$ value can lead to slow convergence. We used their suggested value of 40 and also tried another value 100 in our experiments. For the P-LBFGS method, we used the preconditioner recalculation interval $T = 20$ and number of correction pairs $m = 240$. We only used the close starting points for this experiment, and for P3, 2 out of 8 starting points fail in TNPACK due to line search failures, so we only average the results for 6 starting points of P3. The results are shown in Table (7):

	P1			P2			P3		
targets	iter	nfg	time	iter	nfg	time	iter	nfg	time
TNPACK(40)	11.4	362.63	23.92	29.62	1006.08	546.6	35.5	1256.67	1226.11
TNPACK(100)	7.31	389.69	24.23	15.46	983.62	503.61	16.5	1152.67	1100.27
P-LBFGS	106.31	117.13	11.92	265.85	292.62	199.7	357.5	417.5	481.96

Table 7: Comparison with TNPACK. TNPACK(40) and TNPACK(100) use maxcg values of 40 and 100. iter is the number of L-BFGS iterations for P-LBFGS method and the number of outer iterations for TNPACK.

The results shows that TNPACK has a much lower outer iteration count, which would mean fewer preconditioner calculations than straightforward P-LBFGS with $T = 1$. However, using the larger recalculation interval $T = 20$, P-LBFGS has a similar number of preconditioner calculations to TNPACK(100) on these problems. Then the performance difference is reflected on the number of function and gradient evaluations, which TNPACK requires at each inner CG iteration. We see P-LBFGS is two times faster on these problems than TNPACK in terms of CPU time. However, these results are not meant to offer a conclusive comparison between P-LBFGS and TNPACK on this class of problems, as this is best done by more comprehensive testing. It is possible that we have not made optimal use of TNPACK in spite of considerable experimentation. In [17] the relative performance of TNPACK is better than preconditioned LBFGS. The difference is probably due to the more careful positive definite modification used in ICF, the extra βI term in the preconditioner here, and certainly the larger recalculation interval used here. It is also possible the TNPACK performance would be better if an exact Hessian were used, as it was in [17]. The results appear to indicate, at the least, that P-LBFGS is a competitive choice on these problems.

3.8 Impact on global minimization algorithm

Our global minimization algorithm presented in [10, 6] is heuristic and requires large amounts of computation time. The dominant cost is the full dimensional local searches, which take more than 90% of the total cost. Use of the faster local minimization method described in this paper, should result in significant performance improvement on our global algorithm as well. We made two experiments to test this. In the first experiment, we require the old method and the new method to do a similar number of local searches (around 180). The new method utilizing P-LBFGS reduces the total computational time by a factor of 4. This factor is very close to the factor seen in the performance gain section. In the second experiment, we set the number of local searches so that both methods take similar computational time (around 16 hours). The new method is able to do 115 local searches whereas the old method only does 35. Since using the P-LBFGS method in our global algorithm enables it to explore more space with the same amount of time compared to the old method, the new method should be more likely to get lower minimizers. In the experiment, we found 7 minimizers that are lower in energy than the lowest one found by the old method.

4 Conclusion

We have presented a preconditioned L-BFGS algorithm and its application to the protein structure prediction problem. P-LBFGS utilized the preconditioner created from the partially separable objective function, and greatly reduced the number of function and gradient evaluations. Because of that, we got a speed up factor of at least 3.6 in terms of CPU time. The results indicate the importance of preconditioning in L-BFGS method. Overall, we believe the following conclusions can be drawn from this study.

- Preconditioning can significantly enhance the performance of limited-memory quasi-Newton methods.
- When gradients are expensive, it is worth spending effort to get a precise positive definite modification to the preconditioner factorization.
- Evaluating the preconditioner only every few iterations can significantly improve efficiency.
- Large values (200) of the number of saved secant pairs can be very effective for the preconditioned limited memory method.
- Preconditioned limited memory is at least competitive with preconditioned truncated Newton on molecular problems.

Given the impressive efficiency gains shown in this paper we believe that more effort can be put into improving preconditioned limited memory methods, particularly in using preconditioners involving more of the Hessian, in more precise positive definite factorization modification, and in more efficient methods for preconditioner modification.

Acknowledgments

Computer time was provided by equipment purchased under NSF ARI Grant #CDA-9601817 and NSF sponsorship of the National Center for Atmospheric Research.

The authors appreciate several helpful conversations with Ian Duff, Jorge More and Jorge Nocedal.

References

[1] Christian Bischof, Alan Carle, Paul Hovland, Peyvand Khademi, Andrew Mauer, "ADIFOR 2.0 User's Guide (Revision D)," March 1995. Revised: June, 1998.

[2] R. H. Byrd, J. Nocedal and R. B. Schnabel, "Representation of quasi-Newton matrices and their use in limited memory methods", *Mathematical Programming* 63, 4, 1994, pp. 129-156

[3] Chih-Jen Lin and Jorge J. More', Incomplete Cholesky factorizations with limited memory, SIAM Journal on Scientific Computing, 21, pages 24-45, 1999.

[4] W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz, D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell and P.A. Kollman, A second generation force field for the simulation of proteins, nucleic acids, and organic molecules, J. Am. Chem. Soc. **117**, (1995) 5179-5197.

[5] T.F.Coleman and J.J.More, "Estimating of sparse Jacobian matrices and graph coloring problems", SIAM Journal on Numerical Analysis, 20(1983), pp. 187-209

[6] S. Crivelli, E. Eskow, B. Bader, V. Lamberti, R. Byrd, R. Schnabel, and T. Head-Gordon, "A physical approach to protein structure prediction", Biophysical J. **82**, (2002), 36-49.

[7] Das B, Meirovitch H, Navon IM, "Performance of hybrid methods for large-scale unconstrained optimization as applied to models of proteins" Journal of Computational Chemistry 24 (10): 1222-1231 JUL 30 2003

[8] P. Derreumaux, G. Zhang, B. Brooks, and T. Schlick, "A Truncated-Newton Method Adapted for CHARMM and Biomolecular Applications", J. Comp. Chem., 15:532-552, April (1994)

[9] I. Duff, "MA57 – a code for the solution of sparse symmetric definite and indefinite systems", *ACM Transactions on Mathematical Software* 30 (2004): 118-144.

[10] E. Eskow, B. Bader, R. Byrd, S. Crivelli, T. Head-Gordon, V. Lamberti and R. Schnabel "An optimization approach to the problem of protein structure prediction," to appear in Mathematical Programming series A (2004).

[11] L.N. Kinch, Y. Qi, T.J.P. Hubbard, and N.V. Grishin, "CASP5 target classification", *Proteins: Structure, Function, and Genetics* 53(Supplement 6):340-351, 2003.

[12] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization methods", *Mathematical Programming* 45 (1989): 503-528.

[13] J. J. Moré and D.J. Thuente (1990), "On line search algorithms with guaranteed sufficient decrease", Mathematics and Computer Science Division Preprint MCS-P153-0590, Argonne National Laboratory (Argonne, IL).

[14] J. Nocedal, "Updating quasi-Newton matrices with limited storage", *Mathematics of Computation* 35 (1980): 773-782.

[15] T. Schlick and A. Fogelson, "TNPACK - A Truncated Newton Minimization Package for Large Scale Problems: I. Algorithm and Usage", ACM Trans. Math. Softw., 18:46-70, March (1992)

[16] R.B. Schnabel and E. Eskow, "A revised modified Cholesky factorization algorithm", SIAM Journal on Optimization, volume 9, number 4, pp. 1135-1148

[17] D. Xie and T. Schlick, "Efficient Implementation of the Truncated Newton Method for Large Scale Chemistry Applications", SIAM J. Opt., 10: 132-154 October (1999)

[18] D. Xie and T. Schlick, "Remark on the Updated Truncated Newton Minimization Package, Algorithm 702", ACM Trans. Math. Softw., 25, 108-122, March (1999)

[19] X. Zou, I. M. Navon, M. Berger, P. K. H. Phua, T. Schlick and F. X. Le Dimet, "Numerical Experience with Limited-Memory and Truncated Newton Methods", SIAM J. Opt., 3:582-608, August (1993)