OPTIMIZATION TECHNOLOGY CENTER

TOWARDS A DISCRETE NEWTON METHOD WITH MEMORY FOR LARGE-SCALE OPTIMIZATION [1]

by

Richard H. Byrd[1], *Jorge Nocedal*[2] *and Ciyou Zhu*[2]

Technical Report OTC 95/01

February 15, 1996

ABSTRACT

A new method for solving large nonlinear optimization problems is outlined. It attempts to combine the best properties of the discrete-truncated Newton method and the limited memory BFGS method, to produce an algorithm that is both economical and capable of handling ill-conditioned problems. The key idea is to use the curvature information generated during the computation of the discrete Newton step to improve the limited memory BFGS approximations. The numerical performance of the new method is studied using a family of functions whose nonlinearity and condition number can be controlled.

Key words: discrete Newton method, variable metric method, truncated Newton method, large scale optimization, nonlinear optimization, limited memory method.

[1]To appear in **Nonlinear Optimization and Applications**, G. Di Pillo, F. Giannessi, eds, Plenum.

[1] This author was supported by NSF grant CCR-9101795, ARO grant DAAL 03-91-G-0151, and AFOSR grant AFOSR-90-0109.

[2] These authors were supported by National Science Foundation Grants CCR-940081 and ASC-9213149, and by Department of Energy Grant DE-FG02-87ER25047-A004.

1. Introduction.

In some important classes of large unconstrained optimization problems, the objective function f and its gradient g are available, but the Hessian matrix cannot be computed or stored. Problems of this type are often solved by the discrete-truncated Newton method or the limited memory BFGS method, but both methods have significant drawbacks that sometimes lead to unacceptable computing times. The discrete Newton method can require many gradient evaluations per iteration, whereas the limited memory method can be very slow on ill-conditioned problems. In order to remedy these limitations we propose an algorithm that attempts to combine, in a novel way, the best properties of each method. The purpose of this paper is to outline our approach and test it in a controlled setting.

It is convenient to begin our discussion by reviewing the truncated Newton method. Each iteration of this type of method generates a search direction d_k that approximately solves the Newton equations

$$\nabla^2 f(x_k) d_k = -g(x_k). \tag{1.1}$$

For large problems it is not cost effective to solve the system (1.1) exactly since this may require excessive computing time, and an approximation may result in a sufficiently good search direction. Therefore an inner iterative method such as the linear conjugate method [5] is used to solve the system (1.1) approximately. The accuracy of the solution of (1.1) can be controlled by the norm of the residual,

$$r_k \equiv \nabla^2 f(x_k) d_k + g(x_k),$$

which is forced to decrease as the iterates converge to a solution [1],[15]. An alternative is to terminate the inner iteration when sufficient decrease in the quadratic model

$$\frac{1}{2} d_k^T \nabla^2 f(x_k) d_k + g(x_k)^T d_k$$

has been obtained [11].

When the Hessian $\nabla^2 f(x_k)$ is not positive definite, the inner conjugate gradient iteration may generate a direction v of negative curvature, i.e. a vector v such that $v^T \nabla^2 f(x_k) v < 0$. If this occurs, the inner iteration is terminated and the last estimate obtained prior to the generation of the negative curvature direction is taken as the search direction d_k. Because the inner conjugate gradient iteration is truncated (when either the desired accuracy is reached, or negative curvature is detected), this method is called the *truncated Newton method.*

A modification of the truncated Newton method allows us to solve problems where the Hessian $\nabla^2 f(x_k)$ is not available. It is based on the observation that the conjugate gradient method only needs the product of $\nabla^2 f(x_k)$ with certain displacement vectors v – and does not require the Hessian matrix itself. These products can be approximated by finite differences,

$$\nabla^2 f(x_k) v \approx \frac{g(x_k + \epsilon v) - g(x_k)}{\epsilon}, \tag{1.2}$$

where ϵ is a small differencing parameter. Since each iteration of the conjugate gradient method performs one product $\nabla^2 f(x_k) v$, it requires a new evaluation of the gradient g of the objective function. A method that uses (1.2) is called a *discrete Newton method* [13],[10]. The combination of these two ideas gives the discrete-truncated Newton method, which has been implemented, for example, in the codes of Nash [9] and Schlick and Fogelson [14].

After computing the search direction d_k, we define the new iterate by

$$x_{k+1} = x_k + \alpha_k d_k, \tag{1.3}$$

where α_k is a step length parameter. In this paper we will assume that α_k satisfies the strong Wolfe conditions (cf. [2] or [4])

$$f(x_k + \alpha_k d_k) \le f(x_k) + c_1 \alpha_k g(x_k)^T d_k \tag{1.4}$$

$$|g(x_k + \alpha_k d_k)^T d_k| \le c_2 |g(x_k)^T d_k| \tag{1.5}$$

where $0 < c_1 < c_2 < 1$ are constants.

We should note that the product $\nabla^2 f(x_k)v$ can be computed by automatic differentiation techniques [6] instead of the finite difference (1.2). Automatic differentiation has the important advantage of being accurate. However it is at least as expensive as finite differences in terms of computing time. The algorithms presented below can use the automatic differentiation technique, but for concreteness the details of the discussion will be framed in terms of the discrete version (1.2) of the method.

The accuracy with which (1.1) is solved has a marked effect on the behavior of the discrete-truncated Newton method: if only one iteration of the inner conjugate gradient method is performed, the method reduces to steepest descent, whereas high accuracy results in an approximation to Newton's method.

The discrete-truncated Newton method can be very useful for solving large problems but suffers from two drawbacks. It is not easy to guess the accuracy with which the Newton equations (1.1) are to be solved, so as to obtain a good rate of convergence and at the same time avoid an unnecessarily large number of gradient evaluations. If the accuracy in the solution of (1.1) is too low, convergence will be very slow, but if the inner convergence test is too stringent, an excessive number of gradient evaluations will be made during the inner conjugate gradient iteration. In each of these extreme cases the discrete-truncated Newton method is likely to be less efficient than the limited memory BFGS method described in the next section. The second deficiency is that, in comparison with a quasi-Newton method, discrete-truncated Newton spend a great amount of functional information into each major iteration, but this information is not used in subsequent iterations. In contrast, a quasi-Newton method always carries the information gathered about the Hessian of objective function from one step to the next.

How can we improve the cost-effectiveness of the discrete-truncated Newton method? An idea that comes to mind is to save the curvature information generated by the inner conjugate gradient iteration at the iterate x_k and use it at the next iterate x_{k+1}. This can be done in various ways.

Our approach consists of viewing the inner cycle of conjugate gradient iterations from the point of view of variable metric methods. We note that the gradient differences (1.2) contain useful information about the curvature of the objective function f which can be saved in the form of a limited memory quasi-Newton matrix. Once this information has been stored, it may be unnecessary to perform a discrete Newton step at the next iteration since a limited memory step using this iteration matrix may produce good progress towards the solution. Before we can explain this idea in more detail, we need to digress and review the main features of the limited memory BFGS method.

2. Limited Memory BFGS

The limited memory BFGS method (L-BFGS) is an adaptation, to large problems, of the standard BFGS method. Like the discrete-truncated Newton method, it does not require knowledge about the Hessian matrix, but uses only function and gradient information. The iteration of the L-BFGS method is of the form (1.3), where α_k is a steplength satisfying the Wolfe conditions (1.4)-(1.5), and where the search direction d_k is of the form

$$d_k = -H_k g(x_k). \tag{2.1}$$

The iteration matrix H_k, which is not formed explicitly, is based on the BFGS updating formula. In the standard BFGS method the inverse Hessian approximation H_k is updated at every iteration by means of the formula

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T, \tag{2.2}$$

where

$$\rho_k = 1/y_k^T s_k, \qquad V_k = I - \rho_k y_k s_k^T, \tag{2.3}$$

and

$$s_k = x_{k+1} - x_k, \qquad y_k = g(x_{k+1}) - g(x_k).$$

We say that the matrix H_{k+1} is obtained by updating H_k using the pair $\{s_k, y_k\}$.

The $n \times n$ BFGS matrices H_k will generally be dense, so that storing and manipulating them is prohibitive when the number of variables is large. To circumvent this problem, the limited memory BFGS method does not form these matrices but only stores a certain number, say m, of pairs $\{s_k, y_k\}$ that define them implicitly through the BFGS update formula (2.2)-(2.3). The parameter m therefore determines how much memory is used by the algorithm. The product $H_k g(x_k)$ is obtained by performing a sequence of inner products involving $g(x_k)$ and these m pairs $\{s_k, y_k\}$. After computing the new iterate, we save the most recent correction pair $\{s_k, y_k\}$ – unless the storage is full, in which case we delete the oldest pair from the set $\{s_i, y_i\}$ to make room for the newest one, $\{s_k, y_k\}$. In summary, limited memory BFGS algorithm always keeps the m most recent pairs $\{s_i, y_i\}$ to define the iteration matrix. This approach is suitable for large problems because it has been observed in practice that small values of m (say $m \in [3, 20]$) very often give satisfactory results [7], [3].

Let us now describe the updating process in more detail. Suppose that the current iterate is x_k and that we have stored the m pairs $\{s_i, y_i\}$, $i = k-m, ..., k-1$. We first define the "basic matrix" $H_k^{(0)} = \gamma_{k-1} I$ where

$$\gamma_{k-1} = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}. \tag{2.4}$$

We then update $H_k^{(0)}$ m times using the BFGS formula (2.2)-(2.3) and the m pairs $\{s_i, y_i\}$, $i = k-m, ..., k-1$. From (2.2) we see that H_k can be written as

$$\begin{aligned} H_k &= \left(V_{k-1}^T \cdots V_{k-m}^T\right) H_k^{(0)} \left(V_{k-m} \cdots V_{k-1}\right) \\ &+ \rho_{k-m} \left(V_{k-1}^T \cdots V_{k-m+1}^T\right) s_{k-m} s_{k-m}^T \left(V_{k-m+1} \cdots V_{k-1}\right) \\ &+ \rho_{k-m+1} \left(V_{k-1}^T \cdots V_{k-m+2}^T\right) s_{k-m+1} s_{k-m+1}^T \left(V_{k-m+2} \cdots V_{k-1}\right) \\ &+ \vdots \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T. \end{aligned} \tag{2.5}$$

A recursive formula described in [12] takes advantage of the symmetry of this expression to compute the product $H_k g(x_k)$ efficiently.

The numerical performance of the limited memory method L-BFGS is often very good in terms of total computing time. However the method suffers from two major drawbacks: it is not rapidly convergent, and on ill-conditioned problems it can require an excessive number of function and gradient evaluations. In contrast, the discrete-truncated Newton method can be designed to give fast convergence, and to cope with ill-conditioned problems; its drawback is that it often requires too many gradient evaluations in (1.2).

3. The New Approach

Our aim is to design an algorithm that is as economical as L-BFGS in terms of gradient evaluations, but is efficient on ill-conditioned problems. There are two ways of viewing our goal. One of them is to try to improve the L-BFGS method by interleaving discrete-truncated Newton steps and using the curvature information generated during the inner conjugate gradient iteration to improve the L-BFGS matrix H_k. The second view is to note that the discrete Newton method does not possess memory: the gradient differences used at an iterate are not used at the next one. Therefore the idea is to save the information from these gradient differences in the form of a limited memory matrix, and avoid performing discrete-truncated Newton steps during the next few iterations.

Let us first introduce some notation that will help us explain the idea of saving information from the conjugate gradient cycle into a limited memory matrix.

Let $\{x_l\}$ be the iterates generated by the optimization algorithm. Suppose that at the iterate x_k we perform a discrete-truncated Newton step, i.e. we approximately solve (1.1) by the conjugate gradient (CG) method, using the finite difference (1.2). Let us write the CG iteration at x_k as

$$z^{i+1} = z^i + \sigma^i v^i, \tag{3.1}$$

where σ^i is the steplength and where the directions $\{v^i\}$ are conjugate with respect to the Hessian matrix $\nabla^2 f(x_k)$; see e.g. [5]. The initial guess in the CG iteration is set to be $z^1 = 0$. Suppose that p iterations of the CG method are performed; then the search direction of the optimization algorithm at x_k is given by $d_k = z^p$. To perform the multiplication $\nabla^2 f(x_k)v^i$, which is needed at every CG iteration, we compute the finite difference

$$\frac{g(x_k + \epsilon v^i) - g(x_k)}{\epsilon}.$$

The inner CG iteration (3.1) is terminated when

$$r^i \equiv \nabla^2 f(x_k) z^i + g(x_k) \tag{3.2}$$

satisfies $\|r^i\| \leq c\|g(x_k)\|$, for some constant $c > 0$.

Let us also define

$$s_k^i = \epsilon v^i, \qquad y_k^i = g(x_k + \epsilon v^i) - g(x_k), \quad i = 1, \dots, p. \tag{3.3}$$

Since the CG iteration is terminated if negative curvature is detected, these vectors satisfy the curvature condition

$$(s_k^i)^T y_k^i > 0,$$

and can be used to define a limited memory BFGS matrix H_k as described in the previous section. We conjecture that the information saved in this manner can be quite useful because the conjugate gradient iteration will explore the function along directions that would normally not be seen by the limited memory method.

The new algorithm could proceed as follows. At the starting point x_0 we perform a discrete-truncated Newton step leading to x_1, and construct a limited memory matrix H_1 using the vectors (3.3) generated during the inner conjugate gradient iteration. The matrix H_1 will also incorporate the pair

$$s_0 = x_1 - x_0, \quad y_0 = g_1 - g_0 \tag{3.4}$$

corresponding the total step. Thus H_1 is obtained by updating a multiple of the identity matrix $p+1$ times. We now proceed with the limited memory method generating iterates $x_2, x_3, \dots$, and updating the limited memory matrices $H_2, H_3, \dots$, at each iteration using the information $(s_1, y_1), (s_2, y_2), \dots$, from the total steps.

At some iterate x_t it is judged that a discrete-truncated Newton step should be computed again. We then approximately solve the linear system

$$\nabla^2 f(x_t) d_t = -g(x_t) \tag{3.5}$$

using the conjugate gradient method, and store the new inner vectors (3.3). This produces the search direction d_t leading to the new iterate x_{t+1}. At this point we remove from the limited memory matrix H_t the information obtained from the previous cycle of conjugate gradient iterations and replace it with the latest set of vectors (3.3) generated by the inner CG iterations used to solve (3.5). Thus the matrix H_{t+1} is obtained by updating a multiple of the identity matrix using the most recent inner steps (3.3) plus (possibly) a few previous outer pairs $(s_{t-j}, y_{t-j}), \dots, (s_t, y_t)$.

We can think that the storage is divided in two: one part contains the pairs $\{s^i, y^i\}$ generated during the inner CG iteration, and the other contains pairs $\{s_k, y_k\}$ corresponding to the outer (or total) steps of the algorithm. The information corresponding to the outer steps is continuously updated as in the standard limited memory method: the latest correction vector replaces the oldest one. The information from the inner CG iteration could be kept intact until a new truncated Newton step is performed, and then completely refreshed, or it could be lumped in with the outer iteration pairs and refreshed in the same last-in-first-out fashion.

In summary the algorithm combines discrete-truncated Newton and limited memory steps. But the algorithm does not simply alternate them. The key idea is to preserve the information generated during the inner CG iteration to improve the quality of the limited memory matrix used in subsequent steps.

3.1. Variations

Even though we have outlined our approach in some detail, many variations of it are possible and some details of implementation have to be made more precise. We now list and discuss some of these points.

1. We need to develop a criterion to determine when to trigger a new discrete-truncated Newton step. This could be done with a regular frequency – say 1 discrete-truncated Newton step for every 10 limited memory steps – but an automatic criterion based on the observed behavior of the objective function would be more appropriate.

2. As in any truncated Newton method, the termination test on the inner CG iteration needs to be chosen carefully. But in our approach we also need to determine if all, or only some, of the pairs generated during the inner CG iteration should be stored.

3. The limited memory correction pairs available at an iterate where a discrete-truncated Newton step is to be performed could be used as a preconditioner for the CG iteration. This poses the additional question of what types of pairs should be used in the preconditioner: only the inner pairs $\{s^i, y^i\}$, only the outer pairs $\{s_k, y_k\}$, or both? A simple form of preconditioning of this type is used by Nash [10].

4. How long the two blocks information, from the inner and outer iterations, are to be kept in memory must be determined, as discussed above. Also, the *order* in which limited memory updating is performed has not been specified. The limited memory matrix H_{k+1} could be obtained by updating a multiple of the identity matrix using first the inner information $\{s^i, y^i\}$, followed by the most recent outer pairs (s_k, y_k). Or the order could be reversed, updating first with the outer information and then with the inner information. (In the limited memory method L-BFGS the criterion is simple: the oldest pairs are used first.)

5. This brings up the choice of the scaling parameter γ_k that initializes limited memory updating; see (2.4). It could be based on inner or outer information pairs.

Since there appear to be too many algorithmic choices, many of which could have an important impact on performance, we will not attempt to experiment with different combinations. Instead we would like to test the main idea underlying in this approach: *is it beneficial to save the inner pairs* $\{s^i, y^i\}$ *in the limited memory matrix?* To try to answer this question, we will perform a set of controlled experiments.

4. Numerical Investigation

We will test three algorithms. The first one is the limited memory BFGS method (L-BFGS) as described in [7]. It is used mainly as a benchmark. The other two algorithms combine discrete-truncated Newton and limited memory steps in a regular manner: a discrete Newton step is performed at iterations $6, 16, 26, \ldots$, and all other steps are limited memory BFGS steps. The inner CG iteration that computes the discrete Newton step is terminated either when the residual (3.2) satisfies $\|r^i\|_2 \leq 10^{-2}$, or when the total number of CG iterations is 20. (Since all the test functions will be strongly convex, there is no need to include a test that terminates the iteration when negative curvature is detected). The difference between these two methods lies in the type of information they save, as we now discuss in detail.

DINEMO. (Discrete Newton method with Memory). The first iterations are identical to those of the L-BFGS method. When we reach the first point x_k at which a discrete-truncated Newton step is to be performed, we clear the storage by discarding all the correction pairs saved in the limited memory matrix. We then compute the discrete-truncated Newton step and save all the pairs $\{s^i, y^i\}$ $i = 1, \ldots, p$ generated during the inner CG iteration, as well as the pair (s_k, y_k) corresponding to the outer step. At the new iterate x_{k+1} we construct a limited memory BFGS matrix H_{k+1} as follows: (i) the scaling parameter (2.4) is defined by the outer pair (s_k, y_k); (ii) we update the matrix $\gamma_k I$ $p+1$ times using the inner pairs $\{s^i, y^i\}$ $i = 1, \ldots, p$, and the outer pair (s_k, y_k) (in that order). We then use H_{k+1} to perform a limited memory BFGS step. We continue generating limited memory steps – and adding the new correction pairs to storage – until a new discrete-truncated Newton step is to be performed.

The maximum number of limited memory corrections stored is $m = 29$. Since the maximum number of inner CG steps is $maxcg = 20$, and since a discrete-truncated Newton step is performed at every 10 iterations, limited memory corrections are only discarded prior to taking a discrete-truncated Newton step.

ALTERNATE. In this method the limited memory and discrete-truncated Newton steps are alternated, but the information from the inner CG iteration is *not saved.* This method is identical to DINEMO, except for the construction of the limited memory matrix H_{k+1} immediately following a discrete-truncated Newton step. In ALTERNATE the scaling parameter (2.4) is defined by the outer pair (s_k, y_k), and H_{k+1} is obtained by updating $\gamma_k I$ once using the pair (s_k, y_k).

We will measure the effect of saving information from the inner CG cycle by comparing DINEMO, which saves this information, with ALTERNATE, which does not. The three methods use the same line search. It is performed by the routine of Moré and Thuente [8] with parameters $c_1 = 10^{-4}$ and $c_2 = 0.9$ in (1.4)-(1.5).

We chose the following quartic objective function in $n = 100$ variables to perform our tests,

$$\min \frac{1}{2}(x-1)^T D(x-1) + \frac{\sigma}{4}\Big((x-1)^T B(x-1)\Big)^2 + 1,$$

where D is a positive definite diagonal matrix, σ is a parameter that controls the deviation from quadratic, and

$$B = U^T U, \quad \text{with} \quad U = \begin{bmatrix} 1 & \dots & 1 \\ & \ddots & \vdots \\ & & 1 \end{bmatrix}.$$

The starting point was chosen as $(-1)^i \times 50$ for $i = 1, \dots, 100$.

In the first experiment, the matrix D was chosen as

$$D = \operatorname{diag}\Big[(1+\epsilon)^{-50}, (1+\epsilon)^{-49}, \dots, (1+\epsilon)^{49}\Big], \tag{4.1}$$

where $\epsilon > 0$ is a variable parameter that determines the condition number of D. In our tests we used the values $\epsilon = 0$, $\epsilon = 0.05$, and $\epsilon = 0.09$ which give rise, respectively, to the condition numbers of $1, 125$, and 5073 in D. We also tried several values for the parameter σ to observe the behavior of the methods on non-quadratic functions.

Since we are interested in highlighting the differences between the three methods, we used a very stringent stopping test. The runs were terminated when both

$$f(x_k) \le 1 + 10^{-14} \tag{4.2}$$

and

$$g(x_k)^T g(x_k) \le 10^{-14} \tag{4.3}$$

(note that the optimal objective function value is 1). On some runs, only one of the stopping tests was satisfied, and this is indicated by a * in the tables below, but this is of no particular importance since all the runs achieved essentially the same accuracy. Our code always evaluates the function and gradient simultaneously, and the tables below give the total number of function/gradient evaluations.

The results of the first experiment are given in Table 1. When $\epsilon = \sigma = 0$ the objective function is a quadratic with unit Hessian, and would be minimized using only 1 function/gradient evaluation if an exact line search were used. However our code always tries the unit steplength and accepts it if it satisfies the Wolfe conditions (1.4)-(1.5). Because of this, 6 function/gradient evaluations were required by each of the three methods in this case.

	ϵ/σ			
Algorithm	0/0	0/.06	0/.12	0/.18
L-BFGS	6	131	138	151
DINEMO	6	110	115	115
ALTERNATE	6	133	136	148

	ϵ/σ			
Algorithm	.05/0	.05/.06	.05/.12	.05/.18
L-BFGS	134	208	211	218
DINEMO	153	212	211	210
ALTERNATE	154	246	248	274

	ϵ/σ			
Algorithm	.09/0	.09/.06	.09/.12	.09/.18
L-BFGS	683	607	607	600
DINEMO	899	922	*740	926
ALTERNATE	1084	*936	*811	868

Table 1. Number of function/gradient evaluations when D is given by (4.1). The symbol * indicates that only the stopping test (4.3) was met.

Let us analyze the results of Table 1. When the problem is very well conditioned ($\epsilon = 0$) the new algorithm (DINEMO) performs quite well. However, as ϵ increases, its performance relative to L-BFGS deteriorates. DINEMO does well compared with ALTERNATE (the method that does not save the information generated by the inner CG iteration), except for large values of ϵ and σ. We may be tempted to draw the following conclusion from this test: saving the information from the inner conjugate gradient cycle accelerates the iteration, but for ill-conditioned problems the cost incurred does not pay off – since L-BFGS ends up being the winner when $\epsilon = .09$. It turns out, however, that the condition number of the problem is not the determining factor in the efficiency of the methods, but that the distribution of eigenvalues plays a crucial role, as we will show below. Note that when D is given by (4.1), its eigenvalues are more or less evenly spread, and that this situation is very disadvantageous to the (unpreconditioned) inner conjugate gradient iteration, since the problem has 100 distinct eigenvalues and we only allow 20 conjugate gradient iterations.

In the second experiment we alter D so that its eigenvalue distribution changes but its condition number remains the same. We let the 5 smallest and 6 largest elements be as in (4.1), but make all the other elements 1. Thus, if the new diagonal matrix is denoted by $\hat{D} = \text{diag}(\hat{d}_1, ..., \hat{d}_n)$ we have

$$\begin{aligned} \hat{d}_i &= d_i \quad i = 1, ..., 5 \\ \hat{d}_i &= 1 \quad i = 6, ..., 94 \\ \hat{d}_i &= d_i \quad i = 95, ..., 100, \end{aligned} \tag{4.4}$$

where $D = \text{diag}(d_1, ..., d_n)$ is defined by (4.1). The results are given in Table 2 and are strikingly different. We have omitted the results for $\epsilon = 0$, since they are the same as in Table 1. All three methods required substantially fewer function evaluations, but the new method (DINEMO) performs consistently better than the other two. It is interesting to note that DINEMO has a clear advantage of ALTERNATE, indicating that the strategy of saving inner CG information is useful.

	ϵ/σ			
Algorithm	.05/0	.05/.06	.05/.12	.05/.18
L-BFGS	56	172	177	182
DINEMO	43	139	142	166
ALTERNATE	44	198	206	215
	ϵ/σ			
Algorithm	.09/0	.09/.06	.09/.12	.09/.18
L-BFGS	96	291	291	288
DINEMO	52	175	180	180
ALTERNATE	72	256	253	248

Table 2. Number of function/gradient evaluations when D is given by (4.4).

The matrix $\hat{D}$ given by (4.4) has only 12 distinct eigenvalues, and since the inner CG cycle is allowed to perform 20 steps it will completely solve the Newton equations (1.1) when $\sigma = 0$. Therefore this case may be too simple.

In the third experiment we alter D further so that the inner CG iteration is not able to solve the Newton equations. We leave the 5 smallest and 6 largest eigenvalues as in (4.4) but now split the eigenvalue of 1 into 89 eigenvalues which are contained in the interval $[0.6, 9.4]$. The new diagonal matrix, which we denote by $\bar{D}$ is given by

$$\begin{aligned} \bar{d_i} &= d_i \quad i = 1, ..., 5 \\ \bar{d_i} &= i/10 \quad i = 6, ..., 94 \\ \bar{d_i} &= d_i \quad i = 95, ..., 100. \end{aligned}$$

The eigenvalues of $\bar{D}$ are still clustered into three groups, but the middle cluster has a relatively wide spread of eigenvalues which prevents the CG iteration from "seeing" the cluster of smallest eigenvalues. Note that the condition numbers of $D, \hat{D}$ and $\bar{D}$ remain the same. The results of the third experiment are given in Table 3.

	ϵ/σ			
Algorithm	.05/0	.05/.06	.05/.12	.05/.18
L-BFGS	102	194	190	191
DINEMO	98	178	208	208
ALTERNATE	123	218	219	247
	ϵ/σ			
Algorithm	.09/0	.09/.06	.09/.12	.09/.18
L-BFGS	264	415	359	354
DINEMO	222	333	274	331
ALTERNATE	364	405	*444	435

Table 3. Number of function/gradient evaluations when D is given by (4.1). The symbol * indicates that only the stopping test (4.3) was met.

Table 3 shows that when D is well conditioned ($\epsilon = .05$, which corresponds to a condition number of 125) L-BFGS and DINEMO are comparable. This is in contrast to Table 2 where DINEMO had a clear advantage in this case. Thus collecting incomplete information during the inner CG iteration is not as beneficial for the new method DINEMO. On the other hand, when the condition number of D is large ($\epsilon = .09$, corresponding to a condition numer of 5073) DINEMO performs better than L-BFGS. Since from Table 1 we know that DINEMO does very well when $\epsilon = 0$, we see that, overall, it performs quite well in this test. We note once more that DINEMO outperforms ALTERNATE by a wide margin, indicating again that the inner CG information is useful.

In the tables we have reported only the number of function/gradient evaluations since this is the key measure of performance. But we should mention that DINEMO and ALTERNATE required a substantially smaller number of iterations than L-BFGS.

We performed several other experiments. In one of them the 25 smallest and 26 largest eigenvalues of the diagonal matrix were as in (4.1), and all the other eigenvalues were 1. In that test DINEMO clearly outperformed the two other methods for all values of ϵ. We also repeated the first experiment in which D is given by (4.1), using various stopping criteria for the inner conjugate gradient iteration. We observed that the more information is collected in the inner cycle, the better DINEMO performed.

We have drawn the following conclusions from these experiments. Collecting information from the inner CG cycle and saving it in the form of a limited memory matrix is certainly beneficial since DINEMO clearly outperformed ALTERNATE in the great majority of the tests. The benefits of saving this information depend on the eigenvalue structure of the Hessian. The more information is captured during the inner CG iteration, the more competitive DINEMO is with respect to L-BFGS. Not only is DINEMO faster in terms of number of outer iterations, but its cost effectiveness in terms of function/gradient evaluations increases by capturing more eigenvalue information. We believe that a fully developed version of DINEMO is likely to become a powerful general-purpose optimization algorithm. However to achieve this goal, all the implementation questions listed in §3.1 need to be investigated. We intend to do so in the near future.

5. *

References

[1] R.S. Dembo, S.C. Eisenstat, and T. Steihaug, "Inexact Newton methods," *SIAM J. Numer. Anal.* 19 (1982), pp. 400–408.

[2] J. E. Dennis, Jr. and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, N.J., Prentice-Hall, 1983.

[3] J.C. Gilbert and C. Lemaréchal, "Some numerical experiments with variable storage quasi-Newton algorithms," *Mathematical Programming* 45 (1989), pp. 407–436.

[4] P. E. Gill, W Murray and M. H. Wright, *Practical Optimization*, London, Academic Press, 1981.

[5] G.H. Golub and C.F. Van Loan, *Matrix Computations* (Second Edition), The John Hopkins University Press, Baltimore and London, 1989.

[6] A. Griewank, "On automatic differentiation," *Mathematical Programming* (M. Iri and K. Tanabe, eds.), Kluwer Academic Publishers, (Tokyo, 1989), pp. 83–107.

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

[8] J. J. Moré and D.J. Thuente, "Line search algorithms with guaranteed sufficient decrease," *ACM Transactions on Mathematical Software* 20 (1994), no. 3, pp. 286-307.

[9] S.G. Nash. User's guide for TN/TNBC: FORTRAN routines for nonlinear optimization, Report 397, Mathematical Sciences Dept., The Johns Hopkins University, 1984.

[10] S.G. Nash, "Preconditioning of truncated-Newton methods," *SIAM Journal on Scientific and Statistical Computing* 6 (1985), pp. 599–616.

[11] S.G. Nash and J. Nocedal, "A Numerical Study of the Limited Memory BFGS Method and the Truncated-Newton Method for Large Scale Optimization," *SIAM Journal on Optimization* 1 (1991), no. 3, pp. 358-372.

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

[13] D.P. O'Leary, "A discrete Newton algorithm for minimizing a function of many variables," *Mathematical Programming* 23 (1982), pp. 20–33.

[14] T. Schlick and A. Fogelson, "TNPACK - A truncated Newton package for large-scale problems: I. Algorithms and usage," *ACM Transactions on Mathematical Software* 18 (1992), no. 1, pp. 46-70.

[15] T. Steihaug (1983), "The conjugate gradient method and trust regions in large scale optimization," *SIAM J. Num. Anal.* 20 (1983), pp. 626–637.