NORTHWESTERN UNIVERSITY
Department of Electrical Engineering
and Computer Science

ON THE LIMITED MEMORY BFGS METHOD FOR LARGE SCALE OPTIMIZATION[1]

by

Dong C. Liu[2] *and Jorge Nocedal*[3]

this paper has appeared in
Mathematical Programming 45 (1989), pp. 503-528.

[1]This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-FG02-87ER25047, and by National Science Foundation Grant No. DCR-86-02071.

[2]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston Il 60208.

[3]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston Il 60208.

ON THE LIMITED MEMORY BFGS METHOD FOR LARGE SCALE OPTIMIZATION

by

Dong C. Liu and Jorge Nocedal

ABSTRACT

We study the numerical performance of a limited memory quasi-Newton method for large scale optimization, which we call the L-BFGS method. We compare its performance with that of the method developed by Buckley and LeNir (1985), which combines cyles of BFGS steps and conjugate direction steps. Our numerical tests indicate that the L-BFGS method is faster than the method of Buckley and LeNir, and is better able to use additional storage to accelerate convergence. We show that the L-BFGS method can be greatly accelerated by means of a simple scaling. We then compare the L-BFGS method with the partitioned quasi-Newton method of Griewank and Toint (1982a). The results show that, for some problems, the partitioned quasi-Newton method is clearly superior to the L-BFGS method. However we find that for other problems the L-BFGS method is very competitive due to its low iteration cost. We also study the convergence properties of the L-BFGS method, and prove global convergence on uniformly convex problems.

Key words: large scale nonlinear optimization, limited memory methods, partitioned quasi-Newton method, conjugate gradient method.

Abbreviated title: Limited memory BFGS.

1. Introduction.

We consider the minimization of a smooth nonlinear function $f : \mathbf{R}^n \to \mathbf{R}$,

$$\min f(x), \tag{1.1}$$

in the case where the number of variables n is large, and where analytic expressions for the function f and the gradient g are available. Among the most useful methods for solving this problems are: (i) Newton's method and variations of it; see for example

Steihaug (1983), O'Leary (1982), Toint (1981) and Nash (1985); (ii) the partitioned quasi-Newton method of Griewank and Toint (1982a); (iii) the conjugate gradient method; see for example Fletcher (1980) and Gill, Murray and Wright (1981); (iv) limited memory quasi-Newton methods.

This paper is devoted to the study of limited memory quasi-Newton methods for large scale optimization. These methods can be seen as extensions of the conjugate gradient method, in which additional storage is used to accelerate convergence. They are suitable for large scale problems because the amount of storage required by the algorithms (and thus the cost of the iteration) can be controlled by the user. Alternatively, limited memory methods can be viewed as implementations of quasi-Newton methods, in which storage is restricted. Their simplicity is one of their main appeals: they do not require knowledge of the sparsity structure of the Hessian, or knowledge of the separability of the objective function, and as we will see in this paper, they can be very simple to program.

Limited memory methods originated with the work of Perry (1977) and Shanno (1978b), and were subsequently developed and analyzed by Buckley (1978), Nazareth (1979), Nocedal (1980), Shanno (1978a), Gill and Murray (1979), and Buckley and LeNir (1983). Numerical tests performed during the last ten years on medium size problems have shown that limited memory methods require substantially fewer function evaluations than the conjugate gradient method, even when little additional storage is added. However little is known regarding the relative performance of these methods with respect to Newton's method or the partitioned quasi-Newton algorithm, when solving large problems. Moreover, since the study by Gill and Murray (1979), there have been no attempts to compare the various limited memory methods with each other, and it is therefore not known which is their most effective implementation.

In this paper we present and analyze the results of extensive numerical tests of two limited memory methods and of the partitioned quasi-Newton algorithm. We compare the combined CG-QN method of Buckley and LeNir (1983) as implemented in Buckley and LeNir (1985), the limited memory BFGS method described by Nocedal (1980), and the partitioned quasi-Newton method, as implemented by Toint (1983b). The results indicate that the limited memory BFGS method (L-BFGS) is superior to the method of Buckley and LeNir. They also show that for many problems the partitioned quasi-Newton method is extremely effective, and is superior to the limited memory methods. However we find that for other problems the L-BFGS method is very competitive, in terms of cpu time, with the partitioned quasi-Newton method.

We briefly review the methods to be tested in §2, where we also describe the problems used in our experiments. In §3 we present results that indicate that the limited memory BFGS method is faster than the method of Buckley and LeNir (1985), and is better able to use additional storage to accelerate convergence. In §4 we explore ways of improving the performance of the L-BFGS method, by choosing suitable diagonal scalings, and study its behavior on very large problems (where the number of variables is in the thousands). In §5 we compare the L-BFGS method with two well-known conjugate gradient methods, paying particular attention to execution times. In §6 we compare the L-BFGS method and the partitioned quasi-Newton method, and in §7 we give a convergence analysis of the L-BFGS method.

While this work was in progress we became aware that Gilbert and Lemaréchal (1988) had performed experiments that are similar to some of the ones reported here. They used a newer implementation by Buckley (1987) of the Buckley-LeNir method; this new code is more efficient than the ACM TOMS code of Buckley and LeNir (1985) used in our tests. Gilbert and Lemaréchal's implementation of the L-BFGS method is almost identical to ours. They conclude that the L-BFGS method performs better than Buckley's new code, but the differences are less pronounced than the ones reported in this paper.

Our L-BFGS code will be made available through the Harwell library under the name VA15.

2. Preliminaries

We begin by briefly reviewing the methods tested in this paper.

The method of Buckley and LeNir combines cycles of BFGS and conjugate gradient steps. It starts by performing the usual BFGS method, but stores the corrections to the initial matrix separately to avoid using $O(n^2)$ storage. When the available storage is used up, the current BFGS matrix is used as a fixed preconditioner, and the method performs preconditioned conjugate gradient steps. These steps are continued until the criterion of Powell (1977) indicates that a restart is desirable; all BFGS corrections are then discarded and the method performs a restart. This begins a new BFGS cycle.

To understand some of the details of this method one must note that Powell's restart criterion is based on the fact that, when the objective function is quadratic and the line search is exact, the gradients are orthogonal. Therefore to use Powell restarts, it is necessary that the line search be exact for quadratic objective functions, which means that the line search algorithm must perform at least one interpolation. This is expensive in terms of function evaluations, and some alternatives are discussed by Buckley and LeNir (1983).

The method of Buckley and LeNir generalizes an earlier algorithm of Shanno (1978b), by allowing additional storage to be used, and is regarded as an effective method; see Dennis and Schnabel (1987) and Toint (1986).

The limited memory BFGS method (L-BFGS) is described by Nocedal (1980), where it is called the SQN method. It is almost identical in its implementation to the well known BFGS method. The only difference is in the matrix update: the BFGS corrections are stored separately, and when the available storage is used up, the oldest correction is deleted to make space for the new one. All subsequent iterations are of this form: one correction is deleted and a new one inserted. Another description of the method, which will be useful in this paper, is as follows. The user specifies the number m of BFGS corrections that are to be kept, and provides a sparse symmetric and positive definite matrix H_0, which approximates the inverse Hessian of f. During the first m iterations the method is identical to the BFGS method. For $k > m$, H_k is obtained by applying m BFGS updates to H_0 using information from the m previous iterations.

To give a precise description of the L-BFGS method we first need to introduce some notation. The iterates will be denoted by x_k, and we define $s_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$. The method uses the inverse BFGS formula in the form

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T, \tag{2.1}$$

where $\rho_k = 1/y_k^T s_k$, and

$$V_k = I - \rho_k y_k s_k^T;$$

see Dennis and Schnabel (1983).

Algorithm 2.1 (L-BFGS Method)

(1) Choose x_0, m, $0 < \beta' < 1/2$, $\beta' < \beta < 1$, and a symmetric and positive definite starting matrix H_0. Set $k = 0$.

(2) Compute

$$d_k = -H_k g_k, \tag{2.2}$$

$$x_{k+1} = x_k + \alpha_k d_k, \tag{2.3}$$

where α_k satisfies the Wolfe conditions:

$$\begin{aligned} f(x_k + \alpha_k d_k) &\leq f(x_k) + \beta' \alpha_k g_k^T d_k, && (2.4)\\ g(x_k + \alpha_k d_k)^T d_k &\geq \beta g_k^T d_k. && (2.5) \end{aligned}$$

(We always try the steplength $\alpha_k = 1$ first).

(3) Let $\hat{m} = \min\{k, m-1\}$. Update H_0 $\hat{m}+1$ times using the pairs $\{y_j, s_j\}_{j=k-\hat{m}}^{k}$, i.e. let

$$\begin{aligned} H_{k+1} &= \left(V_k^T \cdots V_{k-\hat{m}}^T\right) H_0 \left(V_{k-\hat{m}} \cdots V_k\right) \\ &+ \rho_{k-\hat{m}} \left(V_k^T \cdots V_{k-\hat{m}+1}^T\right) s_{k-\hat{m}} s_{k-\hat{m}}^T \left(V_{k-\hat{m}+1} \cdots V_k\right) \\ &+ \rho_{k-\hat{m}+1} \left(V_k^T \cdots V_{k-\hat{m}+2}^T\right) s_{k-\hat{m}+1} s_{k-\hat{m}+1}^T \left(V_{k-\hat{m}+2} \cdots V_k\right) \\ &\;\;\vdots \\ &+ \rho_k s_k s_k^T. \end{aligned} \tag{2.6}$$

(4) Set $k := k+1$ and go to 2.

We note that the matrices H_k are not formed explicitly, but the $\hat{m}+1$ previous values of y_j and s_j are stored separately. There is an efficient formula, due to Strang, for computing the product $H_k g_k$; see Nocedal (1980). Note that this algorithm is very simple to program; it is similar in length and complexity to a BFGS code that uses the inverse formula.

This implementation of the L-BFGS method coincides with the one given in Nocedal (1980), except for one detail: the line search is not forced to perform at least one cubic interpolation, but the unit steplegth is always tried first, and if it satisfies the Wolfe conditions, it is accepted. Our aim is that the limited memory method resemble BFGS as much as possible, and we disregard quadratic termination properties, which are not very meaningful, in general, for large dimensional problems.

The partitioned quasi-Newton method of Griewank and Toint assumes that the objective function has the form

$$f(x) = \sum_{i=1}^{ne} f_i(x), \tag{2.7}$$

where each of the ne element functions f_i depends only on a few variables (more generally, it assumes that the Hessian matrix of each element function has a low rank compared with n). The method updates an approximation B_k^i to the Hessian of each element function using the BFGS or SR1 formulas. These small dense matrices, which often contain excellent curvature information, can be assembled to define an approximation to the Hessian of f. The step is determined by an inexact linear conjugate gradient iteration, and a trust region is kept to safeguard the length of the step.

The partitioned quasi-Newton method (PQN) requires that the user supply detailed information about the objective function, and is particularly effective if the correct range of the Hessian of each element function is known. Since in many practical applications the objective function is of the form (2.7), and since it is often possible to supply the correct range information, the method is of great practical value. For a complete description of this algorithm, and for an analysis of its convergence properties see Griewank and Toint (1982a), (1982b), (1984), and Griewank (1987). The tests of the PQN method reported in this paper were performed with the Harwell routine VE08 written by Toint (1983b).

2.1 The Test Problems

The evaluation of optimization algorithms on large scale test problems is more difficult than in the small dimensional case. When the number of variables is very large (in the hundreds or thousands), the computational effort of the iteration sometimes dominates the cost of evaluating the function and gradient. However there are also many practical large scale problems for which the function evaluation is exceedingly expensive. In most of our test problems the function evaluation is inexpensive. We therefore report both the number of function and gradient evaluations and the time required by the various parts of the algorithms. Using this information we will try to identify the classes of problems for which a particular method is effective.

We have used the following 16 test problems with dimensions ranging from 49 to 10000.

Problem	Problem's name	Reference
1	Penalty I	Gill and Murray (1979)
2	Trigonometric	Moré et al. (1981)
3	Extended Rosenbrock	Moré et al. (1981)
4	Extended Powell	Moré et al. (1981)
5	Tridiagonal	Buckley and LeNir (1983)
6	QOR	Toint (1978)
7	GOR	Toint (1978)
8	PSP	Toint (1978)
9	Tridiagonal	Toint (1983a)
10	Linear Minimum Surface	Toint (1983a)
11	Extended ENGVL1	Toint (1983a)
12	Matrix Square Root 1	
13	Matrix Square Root 2	
14	Extended Freudenstein and Roth	Toint (1983a)
15	Sparse Matrix Square Root	
16	u1ts0	Gilbert and Lemaréchal (1988)

Table 1. Set of test problems

Problems 12, 13 and 15, and the starting points used for them, are described in Liu and Nocedal (1988). They derive from the problem of determining the square root of a given matrix A, i.e. finding a matrix B such that $B^2 = A$. For all the other problems we used the standard starting points given in the references. All the runs reported in this paper were terminated when

$$\|g_k\| < 10^{-5} \times \max(1, \|x_k\|), \tag{2.8}$$

where $\|\cdot\|$ denotes the Euclidean norm. We require low accuracy in the solution because this is common in practical applications.

Since we have performed a very large number of tests, we describe the results fully in an accompanying report (Liu and Nocedal (1988)). In this paper we present only representative samples and summaries of these results, and the interested reader is referred to that report for a detailed description of all the tests performed. We should note that all the comments and conclusions made in this paper are based on data presented here and in the accompanying report.

3. Comparison with the method of Buckley and LeNir

In this section we compare the method of Buckley and LeNir (B-L) with the L-BFGS method. In both methods the user specifies the amount of storage to be used, by giving a number m, which determines the number of matrix updates that can be stored. When $m = 1$, the method of Buckley and LeNir reduces to Shanno's method, and when $m = \infty$ both methods are identical to the BFGS method. For a given value of m, the two methods require roughly the same amount of storage, but the L-BFGS method requires slightly less arithmetic work per iteration than the B-L method (as implemented by Buckley and LeNir (1985)).

In both codes the line search is terminated when (2.4) and

$$|g(x_k + \alpha_k d_k)^T d_k| \leq -\beta g_k^T d_k \tag{3.1}$$

are satisfied ((3.1) is stronger than (2.5), which is useful in practice). We use the values $\beta' = 10^{-4}$ and $\beta = 0.9$, which are recommended by Buckley and LeNir (1985), and are also used by Nocedal (1980). All other parameters in the code of Buckley and LeNir were set to their default values, and therefore the method was tested precisely as they recommend. For the L-BFGS method we use a line search routine based on cubic interpolation, developed by J. Moré.

In Table 2 we give the amount of storage required by the two limited memory methods for various values of m and n, and compare it to the storage required by the BFGS method. For example, for a problem with 50 variables, if m= 5, 660 locations are required by each limited memory method.

$n \backslash m$	5	7	15	BFGS
50	660	864	1680	1425
100	1310	1714	3330	5350
1000	13010	17014	33030	503500

Table 2. Storage locations

The tests described below were made on a SUN 3/60 in double-precision arithmetic, for which the unit roundoff is approximately 10^{-16}. For each run we verified that both methods converged to the same solution point. We tested three methods: (1) The combined CG-QN method of Buckley and LeNir (1985) using analytical gradients; (2) the L-BFGS method; (3) the BFGS method, using the line search routine of J. Moré.

The initial Hessian approximation was always the identity matrix, and after one iteration was completed, all methods update $\gamma_0 I$ instead of I, where

$$\gamma_0 = y_0^T s_0 / \|y_0\|^2. \tag{3.2}$$

This is a simple and effective way of introducing a scale in the algorithm; see Shanno and Phua (1978).

In the following tables, P denotes the problem number, N the number of variables and m the number of updates allowed. The results are reported in the form

number of iterations/number of function evaluations
iteration time/function time/total time

where "iteration time" includes the time needed to generate the search direction, perform the line search and test convergence, but excludes the time to evaluate the function and gradient. For all methods the number of gradient evaluations equals the number of function evaluations.

In Table 3 we compare the performance of the two limited memory methods when m=5,7,9. Results for m=15 are given in Table 4, where the runs for the BFGS method are also included for comparison.

		Buckley-LeNir			L-BFGS		
P	N	m=5	m=7	m=9	m=5	m=7	m=9
1	1000	19/88 74/49/123	19/87 79/48/127	19/75 95/41/136	45/55 147/27/174	44/54 179/27/206	44/54 215/27/242
2	1000	48/102 174/675/849	44/94 162/603/765	45/96 187/652/839	53/58 165/337/502	55/58 237/394/631	57/59 288/381/669
4	100	52/108 17/7/24	45/98 17/6/23	38/79 16/4/20	106/111 35/3/38	94/98 42/5/47	57/61 27/2/29
5	100	73/147 52/13/65	72/145 70/11/81	72/145 82/12/94	134/168 43/14/57	126/147 55/10/65	111/131 51/17/68
7	50	82/165 15/48/63	81/163 21/47/68	79/160 17/44/61	162/164 25/50/75	148/150 35/40/75	150/152 39/41/80
10	961	171/343 526/782 1308	183/367 549/858 1407	172/346 544/806 1350	168/280 516/630 1146	167/274 669/606 1275	163/267 680/610 1290
11	1000	14/42 55/38/93	15/44 72/38/110	13/40 71/35/106	36/42 116/37/153	35/41 139/35/174	34/40 162/35/197
12	100	231/467 161/531/692	235/478 175/535/710	225/452 180/507/687	254/260 93/145/238	245/251 112/146/258	246/252 133/149/282

Table 3. Comparison of the two limited memory methods for m=5,7,9. In each box, the two numbers in the top represent iterations/function-evaluations, and the three numbers below give iteration-time/function-time/total-time.

		Buckley-LeNir	L-BFGS	BFGS
P	N	m=15	m=15	
1	1000	19/84 164/54/218	44/54 308/30/338	44/54
2	1000	52/110 278/727/1005	54/56 392/359/751	54/56
4	100	42/87 24/6/30	46/50 33/3/36	41/45
5	100	71/143 108/16/124	110/124 86/9/95	72/77
7	50	147/148 130/42/172	127/129 51/37/88	121/123
10	961	170/341 612/810/1422	155/255 934/578/1512	147/238
11	1000	13/40 99/35/134	29/35 186/32/218	29/35
12	100	229/464 189/533/722	263/269 222/161/383	179/185

Table 4. Limited memory methods using m=15, and the BFGS method.

Tables 3 and 4 give only a small sample of our results, but it is representative of what we have observed (see Liu and Nocedal (1988)). We see that the BFGS method usually requires the fewest function calls, and that for some problems, L-BFGS approaches the performance of the BFGS method. For other problems, however, there remains a gap in terms of function calls, between the BFGS and L-BFGS. In Table 5 we summarize the performance of the two limited memory methods on our whole set of problems, as measured by the number of function evaluations. We give the number of wins, i.e the number of runs for which a method required fewer function calls than the other one.

Method	m=3	m=5	m=7	m=9	m=15	Total
B–L	13	10	5	4	8	39
L-BFGS	17	20	24	26	22	110

Table 5. Number of wins on the whole set of problems.

We see from these results that the L-BFGS method usually requires fewer function calls than the method of Buckley and LeNir (B-L). This is also true if we consider only problems with a very large number of variables ($n \approx 1000$). Only for m=3 are the two methods comparable, and we see that as m increases, the differences between the two become large. To investigate the reason for this, we measure in Tables 6 and 7 the effect of increasing the storage. We define "speed-up" to be the ratio NFUN(m=3)/NFUN(m=7), where NFUN(m=s) denotes the number of function evaluations needed when m=s. Thus if the speed-up is near 1 the method does not gain much from additional storage, whereas a large number means a substantial improvement. In the tables we give the number of test problems for which a certain speed-up was obtained.

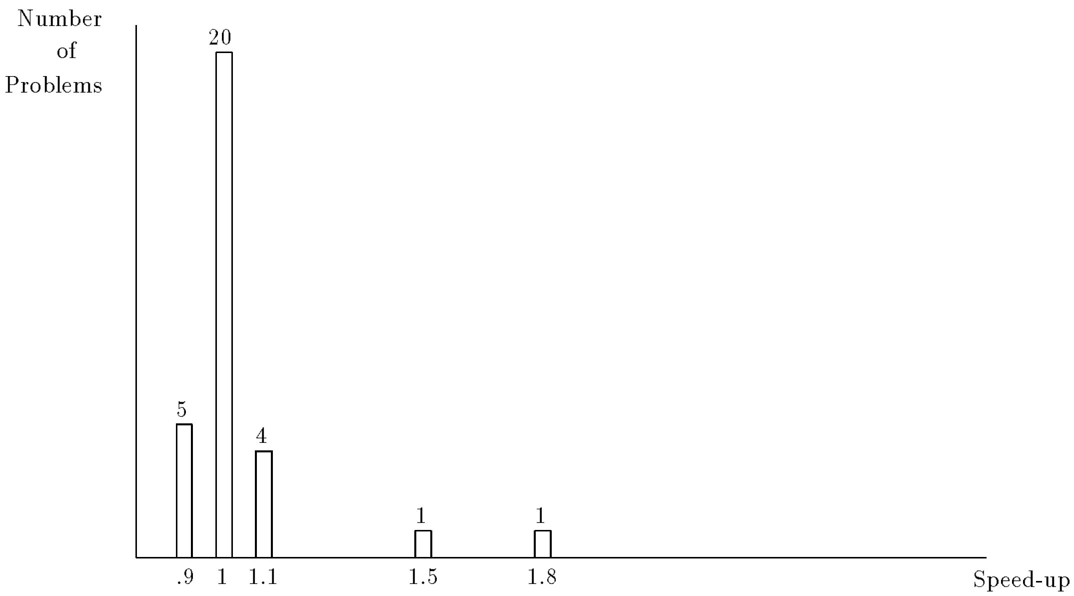

Table 6. Speed-up, NFUN(3)/NFUN(7), for B-L method.

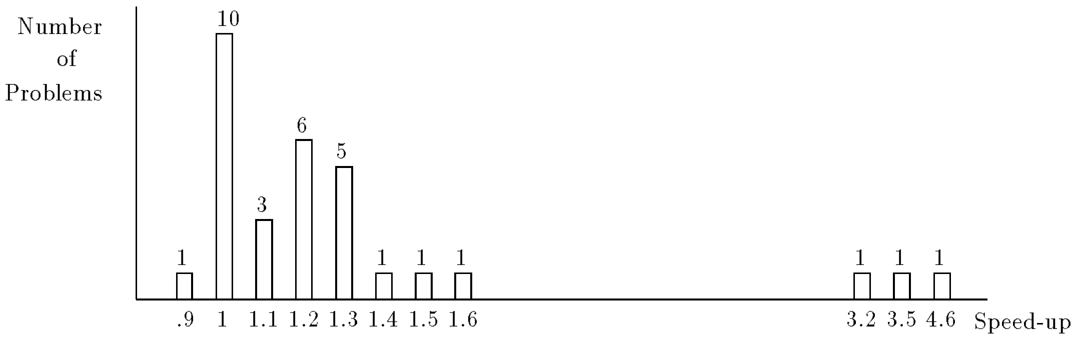

Table 7. Speed-up, NFUN(3)/NFUN(7), for L-BFGS method.

The method of Buckley and LeNir gives little or no speed-up in most of the problems. This is very disappointing because m=7 represents a substantial increase in storage. (The picture is only a slightly better if we define speed-up as NFUN(3)/NFUN(15)). In contrast, the L-BFGS method gives a substantial speed-up in 70% of the problems. We have observed that the L-BFGS method usually reduces the number of function calls as storage is increased, and that this property is true both for medium size and large problems (Liu and Nocedal (1988)). These observations agree with the experience of Gilbert and Lemaréchal (1988).

In our view the method of Buckley and LeNir is not able to use increased storage effectively for the following reason. During the CG cycle, the method uses all m corrections to define the preconditioner. However the restarts are usually performed after only a few iterations of this cycle, and the m corrections are discarded to begin the BFGS cycle. The average number of corrections used during the BFGS cycle is only $(m+1)/2$, since corrections are added one by one. Indeed, what may be particularly detrimental to the algorithm is that the first two or three iterations of the BFGS cycle use a small amount of information. We should add that the relatively accurate line searches performed by the implementation of Buckley and Lenir (1985) also contribute to the inefficiency of the method (this, however, has been corrected in a recent update of the method; see Buckley (1987)).

In practice we would rarely wish to use m greater than 15. However it is interesting to observe the behavior of the L-BFGS method when storage is increased beyond this point. In Table 8 we give the results of using the L-BFGS method with m=15, 25, 40.

Again we see that the number of function calls usually decreases with m, but the gain is not dramatic. The problems given in Table 8 are of medium size, but similar results where obtained when the number of variables was large ($n \approx 1000$).

So far we have concentrated only on the number of function calls, but as we have mentioned earlier, there are practical large scale problems for which the function and gradient evaluation is inexpensive. We will therefore now consider the number of iterations and the total amount of time required by the two limited memory methods.

P	N	L-BFGS m=15	m=25	m=40
4	100	46/50 33/3/36	41/45 36/2/38	41/45 43/2/45
5	100	110/124 86/9/95	109/115 137/7/144	96/104 167/5/172
7	50	127/129 51/37/88	133/135 82/37/119	122/124 107/34/141
10	121	43/49 33/16/49	42/48 36/16/52	41/47 41/14/55
11	100	31/37 21/2/23	30/36 22/4/26	30/36 24/4/28
12	100	263/269 222/161/383	235/241 301/135/436	220/226 420/126/546

Table 8. The L-BFGS method with a large amount of storage.

From Tables 3 and 4 we see that the method of Buckley and LeNir usually requires fewer iterations; when using cpu time as a measure, there is no clear winner. We therefore cannot conclude that the L-BFGS method, as implemented so far, is superior to the method of Buckley and LeNir for problems in which the function evaluation is cheap. However there is a simple way to improve the L-BFGS method in this case.

First, we note that the reason Buckley and LeNir's method requires fewer iterations is that it performs a more accurate line search. The implementation recommended by Buckley and LeNir (1985), i.e. the one obtained by setting all parameters to their default values, ensures that at least one cubic interpolation is applied at every iteration of the algorithm, which usually results in a very good estimate of the one dimensional minimizer. It is therefore natural to perform a more accurate line search in the L-BFGS method in order to decrease the number of iterations. In Table 9 we give the results for the L-BFGS method, when the line search is forced to perform at least one cubic interpolation.

P	N	L-BFGS m=5	m=9	P	N	m=5	m=9
1	1000	16/46 45/27/72	16/46 66/27/93	7	50	97/195 15/57/72	91/183 25/53/78
2	1000	44/89 137/589/726	44/89 218/580/798	12	100	229/461 81/261/342	222/447 132/248/380
11	1000	19/41 60/37/97	18/39 77/36/123	10	961	172/347 512/777/1289	157/317 770/729/1499

Table 9. L-BFGS method with a more accurate line search.

For most problems the number of iterations is markedly reduced (compare Tables 3 and 9). We now compare this implementation of the L-BFGS method with the method of Buckley and LeNir, and for simplicity we will use total cpu time as a measure. In Table 10 we give the number of wins, i.e the number of runs for which a method required less time than the other one, on our whole set of problems.

Method	m=5	m=9	Total
B–L	5	6	11
L-BFGS	24	24	48

Table 10. Number of wins – counting total time

This Table shows that the L-BFGS method is faster on most of the problems. Furthermore an examination of the results given in Liu and Nocedal (1988) shows that the differences are very substantial in many cases. We conclude from these experiments that the L-BFGS method should have two options: (i) when the function and gradient evaluation is expensive, the method should perform an inaccurate line search, like the one described earlier in this section; (ii) otherwise it should perform a more accurate line search, by forcing at least one interpolation, or by using a small value for the parameter β in (3.1).

For the rest of the paper we will consider only the L-BFGS method, since we have seen that it outperforms the method of Buckley and LeNir.

4. Scaling the L-BFGS method

It is known that simple scalings of the variables can improve the performance of quasi-Newton methods on small problems. It is, for example, common practice to scale the initial inverse Hessian approximation in the BFGS method by means of formula (3.2). For large problems scaling becomes much more important; see Beale (1981), Griewank and Toint (1982a) and Gill and Murray (1979). Indeed, Griewank and Toint report that a simple scaling can dramatically reduce the number of iterations of their partitioned quasi-Newton method in some problems. We have observed that this is also the case when using limited memory methods, as we shall discuss in this section.

In the basic implementation of the L-BFGS method given in Algorithm 2.1, the initial matrix H_0, or its scaled version $\gamma_0 H_0$, is carried throughout the iterations. So far we have assumed only that H_0 is sparse, and in our test we have set it to the identity matrix. The choice of H_0 clearly influences the behavior of the method, and a natural question is how best to choose it. If the objective function is mildly nonlinear and if the diagonal entries of the Hessian are all positive, an excellent choice would be to let H_0 be the diagonal of the inverse Hessian matrix at x_0. In general, however, it is preferable to change this matrix as we proceed, so that it incorporates more up-to-date information. Let us therefore replace the matrix H_0 in (2.6) by $H_k^{(0)}$, and consider strategies for computing this matrix at every step.

One simple idea is to use the scaling (3.2) at each iteration and set

$$H_k^{(0)} = \gamma_k H_0, \tag{4.1}$$

where $\gamma_k = y_k^T s_k / \|y_k\|^2$. Another possibility is to try to find a diagonal matrix that approximately satisfies the secant equation with respect to the last m steps. Let x_k be the current iterate, and assume that $k > m$. We find the diagonal matrix D_k which minimizes

$$\|D_k Y_{k-1} - S_{k-1}\|_F, \tag{4.2}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, and $Y_{k-1} = [y_{k-1}, \cdots, y_{k-m}]$, $S_{k-1} = [s_{k-1}, \cdots, s_{k-m}]$. The solution is $D_k \equiv \mathrm{diag}(d_k^i)$ where

$$d_k^i = \frac{s_{k-1}^i y_{k-1}^i + \cdots + s_{k-m}^i y_{k-m}^i}{(y_{k-1}^i)^2 + \cdots + (y_{k-m}^i)^2}, \quad i = 1, ..., n. \tag{4.3}$$

Since an element d_k^i can be negative or very close to zero, we use the following safeguard: formula (4.3) is used only if the denominator in (4.3) is greater than 10^{-10}, and if all the diagonal elements satisfy $d_k^i \in [10^{-2}\gamma_k, 10^2\gamma_k]$; otherwise we set $d_k^i = \gamma_k$.

We have tested the L-BFGS method using the following scalings.

Scaling M1 : $H_k^{(0)} = H_0$ (no scaling).

Scaling M2 : $H_k^{(0)} = \gamma_0 H_0$ (only initial scaling).

Scaling M3 : $H_k^{(0)} = \gamma_k H_0$.

Scaling M4 : Same as M3 during the first m iterations. For $k > m$, $H_k^{(0)} = D_k$; see (4.3).

In Table 11 we give the performance of these scalings on a few selected problems. H_0 was set to the identity matrix, and the method used $m = 5$. The results were also obtained on a SUN 3/60.

P	N	M1	M2	M3	M4
1	1000	34/72 111/35/146	45/55 147/27/174	26/35 87/18/105	29/39 114/20/134
2	1000	51/54 165/330/495	53/58 165/337/502	48/50 160/329/489	50/55 175/332/507
7	50	89/179 14/52/66	162/164 25/50/75	111/119 18/34/52	119/121 25/35/60
10	961	214/569 674/1318/1992	168/280 516/630/1146	190/197 592/435/1027	174/179 544/405/949
11	1000	35/83 112/71/183	36/42 116/37/153	15/22 45/18/63	16/22 54/20/74
12	100	233/482 78/286/364	254/260 93/145/238	308/322 110/183/293	263/270 109/151/260
16	403	41/41 61/1205/1266	26/26 36/806/842	24/27 35/825/860	26/26 38/808/846

Table 11. The L-BFGS method with different scalings, when $m = 5$.

Note the dramatic reduction of function evaluations given by M3 and M4, with respect to M1. We have ranked the performance of the four scalings on each of our test problems, and tallied the rankings for all the problems. The result of such a tally is presented in Tables 12 and 13.

	M1	M2	M3	M4
Best	0	3	12	10
2nd	6	2	6	7
3rd	4	12	4	1
Worst	12	5	0	4

Table 12. Relative performance of scaling methods, counting function calls, on all problems, when m=5.

	M1	M2	M3	M4
Best	4	6	8	6
2nd	8	0	7	8
3rd	3	8	6	2
Worst	7	8	1	6

Table 13. Relative performance of scaling methods, counting cpu time, on all problems, when m=5.

We can see from these tables that M3 and M4 are the most effective scalings. We performed the same tests using $m = 9$ corrections and the results are very similar. M4 seldom required safeguarding; this was needed in only about 5% of the iterations. Our numerical experience appears to indicate that these two scalings are comparable in efficiency, and therefore M3 should be preferred since it is less expensive to implement.

There are many other strategies for dynamically computing scalings. Gill and Murray (1979) have suggested a scaling based on recurring the diagonal of the Hessian approximation produced by the direct BFGS formula. In our tests this formula performed well sometimes, but was very inefficient in many problems. Its behavior seemed erratic, even if one included the safeguards suggested by Gill and Murray, and therefore we do not report these results. It may be very fruitful to study other dynamic scaling strategies — perhaps this is one of the most important topics of future research in large scale optimization.

4.1. Solving Very Large Problems

The largest problems considered so far have 1000 variables. To be able to perform a complete set of tests with larger problems, we had to use a more powerful machine than the SUN 3/60. In Table 14 we describe the performance of the L-BFGS method on problems with 5000 and 10000 variables, using the Alliant FX/8 at Argonne National Laboratory. Double precision arithmetic in this machine has a unit round-off of approximately 10^{-16}. The results are reported in the form:

number of iterations/number of function evaluations
total time

We see that increasing the storage beyond $m = 5$ has little effect on the number of function evaluations, in most of the problems. An improvement is more noticeable if one uses scalings M1 or M2, but the change is still small. We have observed, in general, that when solving very large problems, increasing the storage from $m = 5$ or $m = 7$ gives only a

P	N	m=3	m=5	m=9	m=15	m=40
1	5000	31/46 48	30/45 48	30/45 80	30/45 105	30/45 109
1	10000	37/52 117	35/50 142	35/50 199	35/50 263	35/50 289
2	5000	50/53 96	44/49 105	46/48 148	45/48 192	42/45 218
2	10000	44/46 168	41/43 195	42/44 273	41/43 347	40/42 394
3	5000	34/52 52	33/48 64	35/50 96	35/50 127	35/50 141
3	10000	34/52 105	33/48 130	35/50 195	35/50 258	35/50 284
4	5000	78/99 119	52/61 102	48/58 135	49/55 191	44/49 222
4	10000	183/224 565	52/61 207	50/61 289	53/60 427	51/56 612
11	5000	15/22 24	15/22 28	15/22 34	15/22 34	15/22 34
11	10000	15/22 47	14/21 53	14/21 63	14/21 61	14/21 61
15	4999	150/157 387	147/156 457	146/152 597	143/152 795	142/150 1500
15	10000	149/160 784	149/157 932	144/153 1200	140/147 1570	145/154 3130

Table 14. L-BFGS method with scaling strategy M3.

marginal improvement of performance. Gilbert and Lemaréchal (1988) report similar results. The reason for this is not clear to us. Note, from Table 14, that in all problems the number of iterations needed for convergence is much smaller than the dimension n. In fact, for several problems the number of iterations is a small multiple of m, which would lead one to believe that the value of m is significant. We feel that an explanation of this requires further research.

5. Comparison with conjugate gradient methods.

At this point it is reasonable to ask whether the L-BFGS method, using a scaling such as M3, is faster in terms of cpu time than some of the well-known conjugate gradient methods. We tested three methods: (1) the algorithm CONMIN developed by Shanno and Phua (1980); (2) the conjugate gradient method (CG) using the Polak-Ribière formula (see for example Powell (1977)), restarting every n steps, and with $\beta' = 10^{-4}$ and $\beta = 0.1$ in (2.4) and (3.1); (3) the L-BFGS method M3, for which we tried both accurate and inaccurate line searches. By an accurate line search we mean one in which at least one interpolation was forced; an inaccurate line search does not enforce it. The results are presented in the form

number of iterations/number of function evaluations
iteration time/function time/total time

P	N	CONMIN	CG	L-BFGS (M3)			
				Normal Line Search		Accurate Line Search	
				m=3	m=5	m=3	m=5
1	100	7/15 2/1/3	9/39 1/2/3	16/21 3/1/4	16/21 3/1/4	7/18 2/1/3	7/18 2/1/3
1	1000	11/23 39/14/53	11/58 15/44/59	28/37 64/23/87	26/35 87/18/105	12/32 27/17/44	12/32 40/18/58
2	100	46/98 16/67/83	47/108 8/73/81	52/56 12/35/47	50/57 15/37/52	43/88 10/59/69	44/89 13/59/72
2	1000	47/100 167/653/820	46/102 73/664/737	49/54 110/334/444	48/50 160/329/489	49/99 108/654/762	46/94 153/614/767
3	100	21/54 7/3/10	23/78 4/5/9	34/52 7/2/9	33/48 9/2/11	29/70 6/4/10	31/73 8/4/12
3	1000	30/74 107/26/133	23/78 38/29/67	34/52 78/19/97	33/48 105/17/122	29/70 66/25/91	31/73 98/26/124
4	100	47/95 16/4/20	125/287 18/19/37	70/89 17/3/20	46/54 15/2/17	33/70 8/3/11	25/54 7/2/9
4	1000	41/83 147/48/195	205/465 330/230/560	76/100 174/55/229	50/58 176/30/206	34/72 76/34/110	37/79 130/44/174
5	100	74/149 27/11/38	75/151 11/11/22	129/141 30/8/38	109/114 37/9/46	73/147 17/10/27	74/149 25/10/35
5	1000	280/561 1010/418 1428	280/561 440/418 858	459/483 1056/348 1404	422/443 1530/320 1850	281/563 646/420 1066	281/563 1018/420 1438
6	50	23/47 5/2/7	23/47 2/2/4	37/42 4/2/6	34/38 5/1/6	23/47 2/2/4	23/47 3/2/5
7	50	105/213 20/57/77	92/186 8/54/62	116/124 14/35/49	111/119 18/34/52	87/175 10/52/62	90/181 14/53/67
8	50	84/173 16/7/23	83/211 7/9/16	110/135 14/6/20	101/120 17/5/22	91/190 11/9/20	83/169 15/7/22
9	100	72/145 26/11/37	73/147 12/11/23	112/119 26/7/33	105/112 36/7/43	73/147 17/12/29	72/145 23/11/34
9	1000	275/551 1000/405 1405	275/551 437/405 842	423/451 972/328 1300	367/387 1324/284 1608	276/553 632/409 1041	276/553 938/407 1345
10	121	49/99 21/25/46	45/91 8/22/30	49/52 13/12/25	47/51 17/12/29	42/87 11/20/31	42/87 13/22/35
10	961	163/329 610/731 1341	186/379 280/886 1166	201/206 444/468 912	190/197 592/435 1027	165/338 364/740 1104	165/339 510/746 1256
11	100	14/29 5/3/8	18/47 3/4/7	18/25 4/1/5	15/21 4/1/5	17/37 4/4/8	15/33 4/4/8
11	1000	13/27 47/25/72	18/49 29/43/72	15/22 34/20/54	15/22 45/18/63	15/33 34/27/61	14/31 43/27/70
12	100	231/466 90/278/368	239/482 38/290/328	272/288 63/165/228	308/322 110/183/293	236/475 54/281/335	234/471 79/280/359
13	100	200/403 74/235/309	225/454 35/254/289	290/308 66/182/248	281/289 98/161/259	217/435 50/240/290	224/449 76/243/319
16	403	25/52 36/1520 1556	25/52 16/1518 1534	27/29 25/871 896	24/27 35/825 860	25/50 23/1494 1517	25/50 34/1501 1535

Table 15. CONMIN, CG and L-BFGS methods.

The next two tables summarize the results of Table 15. The performance in terms of function calls is as expected: L-BFGS with inaccurate line search is best, CONMIN is second and CG is worst.

	CONMIN	CG	L-BFGS (M3)			
			Normal		Accurate	
			m=3	m=5	m=3	m=5
Best	2	0	2	19	0	1
2nd	0	0	16	1	3	2
3rd	10	3	1	1	8	7
4th	3	0	1	1	4	7
5th	5	4	2	0	7	5
Worst	2	15	0	0	0	0

Table 16. Relative performance of CONMIN, CG and L-BFGS methods, counting function calls.

	CONMIN	CG	L-BFGS (M3)			
			Normal		Accurate	
			m=3	m=5	m=3	m=5
Best	1	9	10	2	4	2
2nd	1	0	2	7	8	1
3rd	2	6	4	2	5	5
4th	4	3	3	4	5	6
5th	4	2	3	2	0	6
Worst	10	2	0	5	0	2

Table 17. Relative performance of CONMIN, CG and L-BFGS methods, counting cpu time.

Some of the timing results of Table 15 are very surprising. The CG method is in general faster than CONMIN. The best timings of L-BFGS are obtained when m=3; in this case its performance is only slighlty better than that of the CG method.

Examining the results of Table 15 closely we observe that in most of our problems the function and gradient evaluation is inexpensive, which explains why the times of CG are good in spite of its large number of function evaluations. However for a few problems, notably problem 16, the function and gradient are very expensive to compute. We see that in this case the L-BFGS method with an inaccurate line search is much better than CG.

We conclude that the L-BFGS method performs well in comparison with the two conjugate gradient methods, both for expensive and inexpensive objective functions. We also conclude that for large problems with inexpensive functions the simple CG method can still be considered among the best methods available to date. Based on our experience we recommend to the user of Harwell code VA15, which implements the M3 L-BFGS method, to use low storage and accurate line searches, when function evaluation is inexpensive, and to set $3 \leq m \leq 7$ and use an inaccurate line search when the function is expensive.

6. Comparison with the partitioned quasi-Newton method

We now compare the performance of the L-BFGS method with that of the partitioned quasi-Newton method (PQN) of Griewank and Toint, which is also designed for solving large problems. The PQN method is described in detail in Griewank and Toint (1984), and the code VE08 implementing it has been published by Toint (1983b). We will only discuss one feature of the algorithm that is important in practice.

Suppose that one of the element functions in (2.7) is of the form

$$f_i(x) = (x_1 - x_2)^2 + x_3^3.$$

Even though f_i depends on three variables, the rank of its Hessian matrix is only two. One can introduce the linear transformation of variables $y_1 = x_1 - x_2$, $y_2 = x_3$, so that this element function depends on only two variables. In VE08 the user must specify the element function, and is given the option of providing a rule for reducing the number of variables on which this function depends. Two of our test problems allow for a variable reduction, and since we believe that in some cases the user may not wish (or may not be able) to supply the variable reduction rule, we tested the PQN method with and without this option.

Two choices for the starting matrix were used in the PQN method: the identity matrix scaled at the end of the first iteration by the dual of (3.2), $\sigma = y_0^T s_0/\|s_0\|^2$, ($B_0 = \sigma I$), and the Hessian matrix at x_0, estimated by finite differences (B_{diff}). The L-BFGS method was run using the scaling M3, storing $m = 5$ corrections. Stg stands for the amount of storage required by each method, "it" denotes the number of iterations, and nf the number of function/gradient calls. We report three times: iteration-time/function-time/total-time. The runs were performed on a SUN 3/60.

In Table 18 we compare the two methods on two problems that allow for variable reduction, and take advantage of this in the PQN method.

P	N	PQN					L-BFGS		
		$B_0 = \sigma I$			$B_0 = B_{diff}$				
		Stg	it/nf	time	it/nf	time	Stg	it/nf	time
9	100	1005	3/5	5/1/6	3/5	5/1/6	1310	105/112	36/7/43
9	1000	10005	3/5	49/4/53	4/6	57/5/62	13010	367/387	1324/284 1608
10	121	1696	10/13	26/2/28	10/17	26/3/29	1583	47/51	17/12/29
10	961	14656	15/22	834/19 853	15/26	830/24 854	12503	190/197	529/435 964

Table 18. Partitioned quasi-Newton method with variable reduction, and L-BFGS method with M3 scaling and $m = 5$.

In these two problems the PQN method is vastly superior, in terms of function evaluations, to the L-BFGS method. We see that the additional information supplied to the PQN method has been used very effectively. Note that the storage requirements of the two methods are similar. In terms of cpu time the advantage of PQN is less dramatic: PQN is much faster for problem 9, but the two methods have comparable times for the linear minimum surface problem (problem 10).

Table 19 compares the two methods on several other problems. We include the two problems used in Table 18, but this time the PQN method did not use variable reduction.

P	N	PQN					L-BFGS		
		$B_0 = \sigma I$			$B_0 = B_{diff}$				
		Stg	it/nf	time	it/nf	time	Stg	it/nf	time
3	100	906	19/34	8/3/11	40/55	23/4/27	1310	33/48	9/2/11
3	1000	9006	19/34	106/13 119	40/55	231/15 246	13010	33/48	105/17 122
4	100	987	39/46	29/4/33	31/39	24/2/26	1310	46/54	15/2/17
4	1000	9762	42/49	317/26 343	31/39	228/18 246	13010	50/58	176/30 206
9	100	1203	12/14	16/1/17	4/7	7/1/8	1310	105/112	36/7/43
9	1000	12003	12/14	157/10 167	8/11	96/8 104	13010	367/387	1324/284 1608
10	121	2396	28/40	88/3/91	10/19	57/2/59	1583	47/51	17/12/29
10	961	20956	73/107	3373/106 3479	15/28	1411/28 1439	12503	190/197	529/435 964
11	100	1200	12/18	13/1/14	9/12	8/1/9	1310	15/21	4/1/5
11	1000	12000	10/16	95/12/107	9/12	79/8/87	13010	15/22	45/18/63
12	100	23357	95/109	14828/43 14871	116/183	21216/74 21290	1310	308/322	110/183 293
14	100	1200	23/30	23/4/27	10/13	12/1/13	1310	21/28	5/6/11
14	1000	12000	19/25	180/48 228	10/13	96/24 120	13010	18/26	54/58 112
15	100	2643	23/32	103/4/107	25/53	77/5/82	1310	63/71	22/15/37
15	1000	26643	34/58	1032/176 1208	47/88	1431/266 1697	13010	106/113	385/230 615

Table 19. PQN and L-BFGS on several other problems.

The L-BFGS method is very competitive in these problems, in terms of computing time. Even though it usually requires more iterations, this is off set by the low cost of computing the search direction. On the other hand, in terms of function evaluations, the PQN method is clearly the winner. Problem 12 does not really belong in this Table because its Hessian matrix is dense, and therefore it is not suitable for the PQN method. We have included it, however, to show what happens when a problem like this is solved by the PQN method: the results are very poor. This problem has an objective function that may appear at first to be partially separable, and it requires some attention to notice that the Hessian matrix is, in fact, dense.

To analyze these results further, we give in Table 20 more information about the test problems. The number of element functions is denoted by *ne*. The number of variables entering into the element functions is *nve*, and *nve-vr* is the number obtained after applying variable reduction. Using the results of Table 19, we give the average time required to perform an iteration (it-time). For the PQN method we have used the results corresponding to $B_0 = \sigma I$, and we recall that the L-BFGS method used scaling M3 and $m = 5$.

P	N	ne	nve	nve-vr	PQN it-time	L-BFGS it-time
3	100	50	2	2	.42	.27
3	1000	500	2	2	5.58	3.18
4	100	33	4	4	.74	.33
4	1000	333	4	4	7.55	3.52
9	100	100	2	1	1.33	.34
9	1000	1000	2	1	13.1	3.61
10	121	100	4	2	3.14	.36
10	961	900	4	2	46.21	2.78
11	100	99	2	2	1.08	.27
11	1000	999	2	2	9.5	3.0
14	100	99	2	2	1.0	.24
14	1000	999	2	2	9.47	3.0
15	100	164	5	5	4.48	.35
15	1000	1664	5	5	30.35	3.63

Table 20. Separability of the objective functions, and average iteration time.

The iteration time of the L-BFGS method is, of course, quite predictable (it is a function of n). We observe large variations in the iteration time of PQN: for most problems it is 2 to 5 times larger than that of L-BFGS. However for problem 10 (minimum surface problem without variable reduction) and problem 15 (sparse matrix square root problem) the PQN iteration time is 10 to 15 times that of L-BFGS.

The PQN method usually requires less storage than L-BFGS with $m = 5$, except for problem 15, where PQN requires twice as much storage. Note that in this problem the element functions depend on 5 variables. It thus appears from these results that the PQN method becomes less attractive when the number of variables entering into the element functions is greater than 4 or 5.

7. Convergence Analysis

In this section we show that the limited memory BFGS method is globally convergent on uniformly convex problems, and that its rate of convergence is R-linear. These results are easy to establish after noting that all Hessian approximations H_k are obtained by updating a bounded matrix m times using the BFGS formula. Because we prefer to analyze the direct BFGS formula, in what follows we assume that the algorithm updates B_k – the inverse of H_k.

Algorithm 6.1 (General limited memory BFGS algorithm).

(1) Choose x_0, m, $0 < \beta' < 1/2$, $\beta' < \beta < 1$, and a symmetric and positive definite starting matrix B_0. Set $k = 0$.

(2) Compute

$$d_k = -B_k^{-1} g_k, \tag{6.1}$$

$$x_{k+1} = x_k + \alpha_k d_k, \tag{6.2}$$

where α_k satisfies (2.4) and (2.5).

(3) Let $\tilde{m} = \min\{k+1, m\}$, and define a symmetric and positive definite matrix $B_k^{(0)}$. Choose a set of increasing integers $\mathcal{L}_k = \{j_0, ..., j_{\tilde{m}-1}\} \subseteq \{0, ...k\}$. Update $B_k^{(0)}$ $\tilde{m}$ times using the pairs $\{y_{j_l}, s_{j_l}\}_{l=0}^{\tilde{m}-1}$, i.e. for $l = 0, ..., \tilde{m}-1$ compute

$$B_k^{(l+1)} = B_k^{(l)} - \frac{B_k^{(l)} s_{j_l} {s_{j_l}}^T B_k^{(l)}}{{s_{j_l}}^T B_k^{(l)} s_{j_l}} + \frac{y_{j_l} {y_{j_l}}^T}{{y_{j_l}}^T s_{j_l}}. \tag{6.3}$$

Set $B_{k+1} = B_k^{(\tilde{m})}$, $k := k+1$, and go to (2).

There are many possible choices of $B_k^{(0)}$ in step (3) as discussed in §4. For example we could have $B_k^{(0)} = B_0$, or $B_k^{(0)} = B_0/\gamma_k$. We will assume only that the sequence of matrices $B_k^{(0)}$, and the sequence of their inverses, are bounded. Since the elements of $\mathcal{L}_k$ defined in step (3) form an increasing sequence, Algorithm 6.1 is identical to the BFGS method when $k < m$. For $k \geq m$, $\mathcal{L}_k$ can be chosen without this monotonicity restriction, but this may not be advantageous in practice. Note that Algorithms 2.1 and 6.1 are mathematically equivalent. In our code we implement Algorithm 2.1 because it allows us to avoid storing a matrix; Algorithm 6.1 is given only for the purposes of the analysis.

We make the following assumptions about the objective function. The matrix of second derivatives of f will be denoted by G.

Assumptions 6.1

(1) The objective function f is twice continuously differentiable.

(2) The level set $D = \{x \in \mathbf{R}^n : f(x) \leq f(x_0)\}$ is convex.

(3) There exist positive constants M_1 and M_2 such that

$$M_1 \|z\|^2 \leq z^T G(x) z \leq M_2 \|z\|^2 \tag{6.4}$$

for all $z \in \mathbf{R}^n$ and all $x \in D$. Note that this implies that f has a unique minimizer x_* in D.

Theorem 6.1 *Let x_0 be a starting point for which f satisfies Assumptions 6.1, and assume that the matrices $B_k^{(0)}$ are chosen so that $\{\|B_k^{(0)}\|\}$ and $\{\|{B_k^{(0)}}^{-1}\|\}$ are bounded. Then for any positive definite B_0, Algorithm 6.1 generates a sequence $\{x_k\}$ which converges to x_*. Moreover there is a constant $0 \leq r < 1$ such that*

$$f_k - f_* \leq r^k [f_0 - f_*], \tag{6.5}$$

which implies that $\{x_k\}$ converges R-linearly.

Proof: If we define

$$\overline{G}_k = \int_0^1 G(x_k + \tau s_k) d\tau, \tag{6.6}$$

then

$$y_k = \overline{G}_k s_k. \tag{6.7}$$

Thus (6.4) and (6.7) give

$$M_1 \|s_k\|^2 \leq y_k^T s_k \leq M_2 \|s_k\|^2, \tag{6.8}$$

and

$$\frac{\|y_k\|^2}{y_k^T s_k} = \frac{s_k^T \overline{G}_k^2 s_k}{s_k^T \overline{G}_k s_k} \leq M_2. \tag{6.9}$$

Let $\mathrm{tr}(B)$ denote the trace of B. Then from (6.3), (6.9) and the boundedness of $\{\|B_k^{(0)}\|\}$

$$\begin{aligned} \mathrm{tr}(B_{k+1}) &\leq \mathrm{tr}(B_k^{(0)}) + \sum_{l=0}^{\tilde{m}-1} \frac{\|y_{j_l}\|^2}{{y_{j_l}}^T s_{j_l}} \\ &\leq \mathrm{tr}(B_k^{(0)}) + \tilde{m} M_2 \\ &\leq M_3, \end{aligned} \tag{6.10}$$

for some positive constant M_3. There is also a simple expression for the determinant (see Pearson (1969) or Powell (1976))

$$\begin{aligned} \det(B_{k+1}) &= \det(B_k^{(0)}) \prod_{l=0}^{\tilde{m}-1} \frac{{y_{j_l}}^T s_{j_l}}{{s_{j_l}}^T B_k^{(l)} s_{j_l}} \\ &= \det(B_k^{(0)}) \prod_{l=0}^{\tilde{m}-1} \frac{{y_{j_l}}^T s_{j_l}}{{s_{j_l}}^T s_{j_l}} \frac{{s_{j_l}}^T s_{j_l}}{{s_{j_l}}^T B_k^{(l)} s_{j_l}}. \end{aligned} \tag{6.11}$$

Since by (6.10) the largest eigenvalue of $B_k^{(l)}$ is also less than M_3, we have, using (6.8) and the boundedness of $\{\|{B_k^{(0)}}^{-1}\|\}$,

$$\begin{aligned} \det(B_{k+1}) &\geq \det(B_k^{(0)}) \left(\frac{M_1}{M_3}\right)^{\tilde{m}} \\ &\geq M_4, \end{aligned} \tag{6.12}$$

for some positive constant M_4. Therefore from (6.10) and (6.12) we conclude that there is a constant $\delta > 0$ such that

$$\cos \theta_k \equiv \frac{s_k^T B_k s_k}{\|s_k\| \|B_k s_k\|} \geq \delta. \tag{6.13}$$

One can show that the line search conditions (2.4)-(2.5) and Assumptions 6.1 imply that there is a constant $c > 0$ such that

$$f(x_{k+1}) - f(x_*) \leq (1 - c \cos^2 \theta_k)(f(x_k) - f(x_*)),$$

see for example Powell (1976). Using (6.13) we obtain (6.5).

From (6.4)

$$\frac{1}{2} M_1 \|x_k - x_*\|^2 \leq f_k - f_*,$$

which together with (6.5) implies $\|x_k - x_*\| \leq r^{k/2}[2(f_0 - f_*)/M_1]^{1/2}$, so that the sequence $\{x_k\}$ is R-linearly convergent also.

□

It is possible to prove this result for several other line search strategies, including backtracking, by adapting the arguments of Byrd and Nocedal (1987); see the proof of their Theorem 3.1. Note from (6.4), (6.9) and (4.1) that $M_1 \leq \gamma_k \leq M_2$. Thus the L-BFGS method using strategy M3 satisfies the conditions of Theorem 6.1.

One can implement the method of Buckley and LeNir so that it is n-step quadratically convergent on general problems, which implies an R-superlinear rate of convergence. The L-BFGS method does not have this property, and R-linear convergence is the best we can expect. Finally we note that the algorithms of Shanno and Phua and Buckley and LeNir are special cases of Algorithm 6.1, if we let the integer m vary at each iteration in the interval $[1, m_{max}]$, where m_{max} is the maximum number of corrections allowed (see Buckley and LeNir (1983)). Therefore Theorem 6.1 applies also to these two methods.

8. Final Remarks

Our tests indicate that a simple implementation of the L-BFGS method performs better than the code of Buckley and LeNir (1985), and that the L-BFGS method can be greatly improved by means of a simple dynamic scaling, such as M3. Our tests have convinced us that the partitioned quasi-Newton method of Griewank and Toint is an excellent method for large scale optimization. It is highly recommended if the user is able and willing to supply the information on the objective function that the method requires, and it is particularly effective when the element functions depend on a small number of variables (less than 4 or 5, say). The L-BFGS method is appealing for several reasons: it is very simple to implement, it requires only function and gradient values – and no other information on the problem – and it can be faster than the partitioned quasi-Newton method on problems where the element functions depend on more than 3 or 4 variables. In addition, the L-BFGS method appears to be preferable to PQN for large problems in which the Hessian matrix is not very sparse, or for problems in which the information on the separablity of the objective function is difficult to obtain.

Our tests also indicate that L-BFGS with dynamic scalings performs better than the CONMIN code of Shanno and Phua (1980) and than the standard conjugate gradient method (CG), except in one case: for large problems with inexpensive functions, CG is competitive with L-BFGS.

Acknowledgements. We would like to thank Andreas Griewank and Claude Lemaréchal for several helpful conversations, and Richard Byrd for suggesting the scaling used in method M4. We are grateful to Jorge Moré who encouraged us to pursue this investigation, and who made many valuable suggestions, and to the three referees for their helpful comments.

References

E.M.L. Beale, "Algorithms for very large nonlinear optimization problems," in: M.J.D. Powell, ed., *Nonlinear Optimization 1981* (Academic Press, London, 1981) 281-292.

A. Buckley, "A combined conjugate gradient quasi-Newton minimization algorithm," *Mathematical Programming* 15 (1978) 200-210.

A. Buckley, "Update to TOMS Algorithm 630," Rapports Techniques No. 91, Institut National de Recherche en Informatique et en Automatique, Domaine Voluceau, Rocquencourt, B.P. 105, Le Chesnay Cedex, (France, 1987).

A. Buckley and A. LeNir, "QN-like variable storage conjugate gradients," *Mathematical Programming* 27 (1983) 155-175.

A. Buckley and A. LeNir, "BBVSCG – A variable storage algorithm for function minimization," *ACM Transactions on Mathematical Software* 11/2 (1985) 103-119.

R.H. Byrd and J. Nocedal, "A tool for the analysis of quasi-Newton methods with application to unconstrained minimization," (1987) to appear in *SIAM Journal on Numerical Analysis.*

J.E. Dennis, Jr. and R.B. Schnabel, *Numerical methods for unconstrained optimization and nonlinear equations* (Prentice-Hall, Inc., New Jersey, 1983).

J.E. Dennis Jr. and R.B. Schnabel, "A view of unconstrained optimization," University of Colorado Tech. Rpt. CU-CS-376-87, to appear in: G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J. Todd, eds., *Handbooks in Operations Research and Management Science, Vol 1, Optimization* (North-Holland, Amsterdam, 1987).

R. Fletcher, *Practical Methods of Optimization, Vol.1, Unconstrained Optimization* (John Wiley & Sons, New York, 1980).

J.C. Gilbert and C. Lemaréchal, "Some numerical experiments with variable storage quasi-Newton algorithms," IIASA Working Paper WP-88, A-2361 (Laxenburg, Austria, 1988).

P.E. Gill and W. Murray, "Conjugate-gradient methods for large-scale nonlinear optimization," Technical report SOL 79-15, Department of Operations Research, Stanford University, (Stanford, CA, 1979).

P.E. Gill, W. Murray and M.H. Wright, *Practical Optimization* (Academic Press, London, 1981).

A. Griewank, "The global convergence of partitioned BFGS on semi-smooth problems with convex decompositions," ANL/MCS-TM-105, Mathematics and Computer Science Division, Argonne National Laboratory (Argonne, IL, 1987).

A. Griewank and Ph.L. Toint, "Partitioned variable metric updates for large structured optimization problems," *Numerische Mathematik* 39 (1982a) 119-137.

A. Griewank and Ph.L. Toint, "Local convergence analysis of partitioned quasi-Newton updates," *Numerische Mathematik* 39 (1982b) 429-448.

A. Griewank and Ph.L. Toint, "Numerical experiments with partially separable optimization problems," in: D.F. Griffiths, ed., *Numerical Analysis: Proceedings Dundee 1983,* Lecture Notes in Mathematics 1066 (Springer Verlag, Berlin, 1984) 203-220.

D.C. Liu and J. Nocedal, "Test results of two limited memory methods for large scale optimization," Technical Report NAM 04, Department of Electrical Engineering and Computer Science, Northwestern University (Evanston, IL, 1988).

J.J. Moré, B.S. Garbow and K.E. Hillstrom, "Testing unconstrained optimization software," *ACM Transactions on Mathematical Software* 7 (1981) 17-41.

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

L. Nazareth, "A relationship between the BFGS and conjugate gradient algorithms and its implications for new algorithms," *SIAM Journal on Numerical Analysis* 16 (1979) 794-800.

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

J.D. Pearson, "Variable metric methods of minimization, *Computer Journal* 12 (1969) 171-178

J.M. Perry, "A class of conjugate gradient algorithms with a two-step variable-metric memory," Discussion Paper 269, Center for Mathematical Studies in Economics and Management Science, Northwestern University (Evanston, IL, 1977).

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

M.J.D. Powell, "Some global convergence properties of a variable metric algorithm for minimization without exact line search," in: R.W. Cottle and C.E. Lemke eds., *Nonlinear Programing, SIAM-AMS Proceedings IX* (SIAM Publications, 1976).

M.J.D. Powell, "Restart procedures for the conjugate gradient method," *Mathematical Programming* 12 (1977) 241-254.

D.F. Shanno, "On the convergence of a new conjugate gradient algorithm," *SIAM Journal on Numerical Analysis* 15 (1978a) 1247-1257.

D.F. Shanno, "Conjugate gradient methods with inexact searches," *Mathematics of Operations Research* 3 (1978b) 244-256.

D.F. Shanno and K.H. Phua, "Matrix conditioning and nonlinear optimization," *Mathematical Programming* 14 (1978) 149-160.

D.F. Shanno and K.H. Phua, "Remark on algorithm 500: minimization of unconstrained multivariate functions," *ACM Transactions on Mathematical Software* 6 (1980) 618- 622.

T. Steihaug, "The conjugate gradient method and trust regions in large scale optimization," *SIAM Journal on Numerical Analysis* 20 (1983) 626-637.

Ph.L. Toint, "Some numerical results using a sparse matrix updating formula in unconstrained optimization," *Mathematics of Computation* 32 (1978) 839-851.

Ph.L. Toint, "Towards an efficient sparsity exploiting Newton method for minimization," in: I.S. Duff, ed., *Sparse Matrices and their Uses* (Academic Press, New York, 1981), 57-87.

Ph.L. Toint, "Test problems for partially separable optimization and results for the routine PSPMIN," Report Nr 83/4, Department of Mathematics, Facultés Universitaires de Namur (Namur, Belgium, 1983a).

Ph.L. Toint, "VE08AD, a routine for partially separable optimization with bounded variables," Harwell Subroutine Library, A.E.R.E. (UK, 1983b).

Ph.L. Toint, "A view of nonlinear optimization in a large number of variables," Report Nr 86/16, Department of Mathematics, Facultés Universitaires de Namur (Namur, Belgium, 1986).