SIAM J. OPTIM.
Vol. 29, No. 2, pp. 1282–1299

ADAPTIVE, LIMITED-MEMORY BFGS ALGORITHMS FOR UNCONSTRAINED OPTIMIZATION*

PAUL T. BOGGS† AND RICHARD H. BYRD‡

Abstract. The limited-memory BFGS method (L-BFGS) has become the workhorse optimization strategy for many large-scale nonlinear optimization problems. A major difficulty with L-BFGS is that, although the memory size m can have a significant effect on performance, it is difficult to know what memory size will work best. Importantly, a larger m does not necessarily improve performance, but may, in fact, degrade it. There is no guidance in the literature on how to choose m. In this paper, we briefly review L-BFGS and then suggest two computationally efficient ways to measure the effectiveness of various memory sizes, thus allowing us to adaptively choose a different memory size at each iteration. Our overall numerical results illustrate that our approach improves the performance of the L-BFGS method. These results also indicate some further directions for research.

Key words. quasi-Newton methods, BFGS, limited-memory methods, adaptive strategies, compact form

AMS subject classifications. 65K05, 90-08, 90C53

DOI. 10.1137/16M1065100

1. Introduction.

Large-scale unconstrained optimization problems continue to arise in numerous applications, ranging from simple data fitting problems to complicated PDE-constrained optimization problems. Thus it is important to have an efficient solver for such problems that does not require the tuning of parameters for each application area. We show that a popular unconstrained optimization method can be modified to overcome one of its main deficiencies and thus can be used to meet our goal of developing such a method.

Since its introduction, the limited-memory BFGS (L-BFGS) algorithm has emerged as one of the workhorse methods for large-scale unconstrained optimization. (See, e.g., [10] and [15].) It has several advantages:

- It does not require computing the second derivative matrix (the Hessian) of the objective.
- It does not require the problem to have any special structure, or Hessian sparsity.
- It uses an approximation to the Hessian based on the BFGS quasi-Newton method, which is the most successful of the quasi-Newton methods that maintain hereditary positive definiteness.
- It stores only a modest amount of recently computed information (the limited memory), which is used to compute the new search direction efficiently.

*Received by the editors March 10, 2016; accepted for publication (in revised form) February 20, 2019; published electronically May 9, 2019. This work was performed by an employee of the U.S. Government or under U.S. Government contract. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Copyright is owned by SIAM to the extent not limited by these rights.

http://www.siam.org/journals/siopt/29-2/M106510.html

Funding: The second author was supported by National Science Foundation grant DMS-1620070.

†Sandia National Laboratories (Retired), Livermore, CA 94550 (paulboggs@comcast.net).

‡Department of Computer Science, University of Colorado, Boulder, CO 80309 (richard.byrd@colorado.edu).

- It has been used successfully on a large number of problems arising from many applications.

A major disadvantage of L-BFGS is that the user must specify the amount of memory that is used. There is little guidance provided in the literature to make the choice and, even worse, the performance of the algorithm can vary dramatically as the memory size is changed, often in erratic ways. It is certainly not the case that the use of more memory is always better.

In subsequent sections, we suggest several adaptive strategies for dynamically changing the memory size as the computation proceeds; i.e., we allow the memory size to adaptively change at each iteration. We show that these strategies can be effective in creating an algorithm that performs consistently well on a collection of test problems. We begin in section 2 by stating the general unconstrained optimization problem and reviewing the details of the L-BFGS method. In section 3, we suggest a strategy for dynamically choosing the memory size, and we report on its performance on a collection of problems. Although this method has an acceptable additional computational cost, the success of the idea led us to consider a different method that is substantially cheaper. Remarkably, this method did even better on many, but not all, of our test collection (see section 4). In section 5 we summarize the results, showing all of our problems with each of the methods. We also provide some further comparisons. Some details of the implementation of our methods are discussed in section 6. Finally, in section 7 we give some conclusions and future research directions.

2. The limited-memory BFGS method. Consider the unconstrained minimization problem

$$\underset{x}{\text{minimize}} \quad f(x), \tag{2.1}$$

where $x \in \mathcal{R}^n$, and f is a twice differentiable function. Both standard quasi-Newton methods (that use positive definite Hessian approximations) and their limited-memory counterparts compute an approximate solution to (2.1) by an iteration of the form

$$x_{k+1} = x_k - \alpha_k H_k \nabla f(x_k), \tag{2.2}$$

where x_0 and H_0 are given and H_k is a symmetric positive definite matrix whose inverse approximates the Hessian, $\nabla^2 f(x_k)$. The scalar parameter α_k is chosen by a linesearch strategy.

A few remarks about this algorithm are in order. First, to obtain guaranteed convergence, conditions need to be placed on the steplength α_k used in (2.2). We require a positive value of α_k that satisfies the strong Wolfe conditions

$$f(x_k + \alpha_k p) \le f(x_k) + c_1 \alpha_k \nabla f(x_k)^{\mathsf{T}} p, \tag{2.3}$$

$$|\nabla f(x_k + \alpha_k p)^{\mathsf{T}} p| \le c_2 |\nabla f(x_k)^{\mathsf{T}} p|, \tag{2.4}$$

where $p = H_k \nabla f(x_k)$, for specified values of $c_1 \in (0, 1/2)$ and $c_2 \in (c_1, 1)$. Specifically, in our code we use the linesearch strategy of Moré and Thuente given in [13], using $c_1 = 10^{-4}$ and $c_2 = 0.9$. We set the max number of iterations at each step to be 20. The overall convergence criteria are standard, requiring that either two successive iterates are relatively close (xTol), or that two successive function values are relatively close (fTol), or that $\nabla f(x_k)$ is sufficiently small (gTol). Specifically, we use the following

values for all of our tests:

$$\begin{aligned}\mathsf{xTol} &= 5.0 \times 10^{-08},\\ \mathsf{fTol} &= 5.0 \times 10^{-10},\\ \mathsf{gTol} &= 5.0 \times 10^{-07}.\end{aligned}$$

All the methods considered in this paper use the above linesearch, stopping conditions, and parameter values.

In most standard quasi-Newton methods the inverse Hessian approximation H_k is updated from the previous approximation H_{k-1} using the *secant vectors*

$$\begin{aligned}s_k &= x_k - x_{k-1},\\ y_k &= \nabla f(x_k) - \nabla f(x_{k-1}).\end{aligned}$$

H_k is then chosen to satisfy the secant equation

$$H_k y_k = s_k.$$

As noted, the most successful positive definite quasi-Newton update is the BFGS update given by

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

where

$$\begin{aligned}\rho_k &= \frac{1}{s_k^{\mathsf{T}} y_k},\\ V_k &= I - \rho_k y_k s_k^{\mathsf{T}},\end{aligned}$$

and I is the identity matrix.

One difficulty with this method is that it requires the storage of an $n \times n$ matrix. Even if the underlying true Hessian is sparse, the BFGS approximation will, in general, be dense.

A quite successful modification of the quasi-Newton methods is the *limited-memory* BFGS method (L-BFGS) of Nocedal (see [10], [15]). In this method, no Hessian approximation is ever actually formed, but rather a collection of the last several (s_k, y_k) pairs is stored and used to compute the step. Let m, the memory size, be the number of (s, y) pairs stored. Then, given an initial matrix H_k^0, the matrix H_k can be defined as follows:

$$\begin{aligned}&H \leftarrow H_k^0\\ &\text{for } i = m, \ldots, 1\\ &\qquad H \leftarrow V_{k-i}^{\mathsf{T}} H V_{k-i} + \rho_{k-i} s_{k-i} s_{k-i}^{\mathsf{T}}\\ &\text{end for}\\ &H_k \leftarrow H.\end{aligned}$$

We simplify the notation by eliminating the iteration counter k and choosing to store the most recent value of s, that is, s_{k-1}, in s_{m-1} and the oldest value, s_{k-m}, in s_0. The vectors y_i, $i = 0, \ldots, m-1$, are stored similarly. Let x_c be the current value of x. For the initial matrix H_k^0 we override our previous notation and take H_0

1. Set $q = \nabla f(x_c)$
2. For $i = m-1, m-2, \ldots, 0$
   - $\eta_i = \rho_i s_i^{\mathsf{T}} q$
   - $q := q - \eta_i y_i$
3. Set $p = H_0 q$
4. for $i = 0, 1, \ldots, m-1$
   - $\beta = \rho_i y_i^{\mathsf{T}} p$
   - $p := p + (\eta_i - \beta) s_i$
5. Return p

FIG. 1. *The two-loop recursion to compute $H\nabla f(x_c)$.*

to be the initial matrix at the current iteration and let H be the updated version of H_0. With these values, it can be shown that the search direction given by

$$p = -H\nabla f(x_c)$$

can be computed by the well-known "two-loop recursion" given in Figure 1. Note that H is never actually formed. This algorithm was introduced in [14] and, interestingly, was shown in [7] to be derivable from an instance of the reverse mode of automatic differentiation. This algorithm is computationally efficient, requiring only approximately $4nm$ operations plus the computation of $H_0 q$ in step 3. (We use the term "operations" to refer to one multiplication and one addition.) In most of the implementations of this method, the initial matrix H_0 is taken to be γI, where

$$\gamma = \frac{s_{m-1}^{\mathsf{T}} y_{m-1}}{y_{m-1}^{\mathsf{T}} y_{m-1}}. \tag{2.5}$$

As noted above, the L-BFGS has emerged as the leading method for solving large unconstrained optimization problems, but the choice of m is problematic. Naively, one might be tempted to use a value of m as large as possible, but often much smaller sizes of m perform better.

To test our ideas, we consider a set of problems chosen from the CUTE set (see [1]). Our original set consisted of 228 problems. We noted, however, that quite a few of these problems were not of interest. First, since these methods were designed for large problems, we eliminated all 129 problems with dimension less than 50. Next, we eliminated the problems where all of the methods—L-BFGS for all m and the new methods described in the next two sections—converged within two iterations of each other; there were 43 such problems. Finally, there were four problems for which all of the methods failed in the linesearch, usually after only a few iterations, and four problems that failed to converge in our cutoff of 3000 iterations. This left a set of 48 problems that we used to test our ideas. These, along with their dimensions, are given in Table 1.

The performance of L-BFGS with m varying from 5 to 50 in increments of 5 is given in Table 2. The maximum and minimum number of function evaluations for each of the problems is given in Table 3. (The number of gradient evaluations is the same as the number of function evaluations in our implementation.) As one can easily see, there are considerable variations over this range, but there is no clear pattern to the best performing value of m. Clearly, no one value of m was uniformly preferable.

Intuitively, one might argue that a small memory size might be preferable when the iterates are far from the solution and are changing rapidly. In this case, it may be

TABLE 1
Problem names and number of variables, n. We will use the problem number in the rest of the paper.

No.	Name	n
1	arwhead	5000
2	bdqrtic	1000
3	bratu1d	1001
4	broydn7d	1000
5	chnrosnb	50
6	clplatea	4970
7	clplateb	4970
8	cragglvy	5000
9	curly10	10000
10	curly20	10000
11	curly30	10000
12	cvxbqp1	10000
13	deconvu	51
14	dixmaane	3000
15	dixmaanf	3000
16	dixmaang	3000

No.	Name	n
17	dixmaanh	3000
18	dixmaani	3000
19	dixmaanj	3000
20	dixmaank	3000
21	dixmaanl	3000
22	eigenals	110
23	eigena	110
24	eigenbls	110
25	eigenb	110
26	errinros	50
27	fletcbv2	100
28	fletchcr	100
29	flosp2hl	691
30	fminsrf2	1024
31	fminsurf	1024
32	freuroth	5000

No.	Name	n
33	genrose	500
34	liarwhd	10000
35	msqrtals	1024
36	msqrtbls	1024
37	noncvxu2	1000
38	noncvxun	1000
39	nondquar	10000
40	nonscomp	10000
41	qr3dbd	127
42	scurly10	10000
43	scurly20	10000
44	scurly30	10000
45	sinquad	10000
46	tointqor	50
47	tridia	10000
48	woods	10000

argued that the (s, y) pairs from even a few previous iterations would not be useful in an approximation to the Hessian at the current point. Similarly, as the iterates near the solution, it would seem reasonable that a larger value of m would be better since the function should be nearly quadratic, and thus we want the L-BFGS to come as closely as possible to the full BFGS. Actually, as we shall see later, the situation is more complicated. In the next section we suggest our first way to use the current data to choose an appropriate memory size to use.

3. A strategy for memory-size selection. The underlying idea for our algorithm is to choose a maximum memory size M that is reasonably large, e.g., 50, and to always store M (s, y) pairs. Then, at each iteration, we choose a value of $m \leq M$ of pairs to compute the search direction. The value m is chosen to optimize some measure of the value of using the most recent m pairs.

In order to consider ways of selecting the memory size to use in L-BFGS it is helpful to introduce some notation. We will use the symbol $H^{i,j}$, where $i \geq j$, to denote the quasi-Newton matrix formed from update pairs $(s_{M-i}, y_{M-i}), \ldots (s_{M-j}, y_{M-j})$, applied to the initial H_0. Thus L-BFGS with memory size m takes the step direction $-H^{m,1}\nabla f(x_c)$, where x_c is the current iterate.

The essence of our first strategy for selecting which value of m to use in $H^{m,1}$ is to estimate which matrix is closest to the inverse Hessian at x_c. An ideal way to do this would be, for some test vector v and for some integer $j \in [1, M]$, to find which value of m results in the smallest discrepancy

$$\| \left(H^{m,j} - \nabla^2 f(x_c)^{-1}\right) v\|.$$

Since the only information we have on the exact Hessian is the quasi-Newton pairs, it is natural to take v to be one of the secant vectors y_p. Thus, since since $\nabla^2 f(x_c)s_p \approx y_p$ we consider the quantities

$$\left\|H^{m,j}y_p - s_p\right\|.$$

The available pair with the most current Hessian information at x_c is, of course, (s_{M-1}, y_{M-1}). However, if we are using y_{M-1} as a test vector, we cannot use it in forming our trial matrix since $H^{m,1}y_{M-1} = s_{M-1}$ for all m. Therefore, we temporarily

TABLE 2

Function evaluations for given memory size; min in each row is boxed. Failure to converge in 3000 *iterations is indicated by a negative number in the table.*

Prob	Memory Size									
	5	10	15	20	25	30	35	40	45	50
1	**15**	19	16	16	16	16	16	16	16	16
2	107	80	69	47	50	38	**37**	**37**	**37**	**37**
3	−3091	**3044**	−3069	−3070	−3064	−3067	−3061	−3082	−3066	−3071
4	**356**	370	374	375	375	380	380	377	378	374
5	257	222	214	209	208	201	197	193	189	**183**
6	810	585	451	435	575	423	412	402	**391**	403
7	482	482	468	444	**436**	448	472	463	472	469
8	71	72	71	68	**66**	67	72	76	77	81
9	1022	1001	1091	1247	**776**	1199	1134	1044	1261	1165
10	1438	1514	2168	1972	1560	1689	1634	**1357**	1771	1651
11	2201	2263	1757	2462	2090	**1711**	2419	2371	2297	2261
12	2162	1966	2002	2051	2022	1892	**1664**	1832	2139	1785
13	122	93	65	69	64	60	61	61	60	**59**
14	**228**	241	243	241	237	236	240	239	235	241
15	211	178	178	176	176	180	169	**166**	173	174
16	214	171	174	174	172	175	**169**	172	172	**169**
17	218	176	172	172	173	173	174	172	176	**171**
18	1766	1847	**1653**	1985	2031	1721	1905	1878	1763	1837
19	250	238	253	252	249	251	249	257	257	**223**
20	224	233	235	215	**190**	229	199	222	226	229
21	**177**	223	184	215	202	205	199	217	203	203
22	300	325	234	161	120	109	116	106	**95**	97
23	319	458	238	161	120	109	116	106	**95**	97
24	1292	1126	1004	963	898	885	891	**816**	843	849
25	1280	1178	980	934	907	850	**573**	819	894	596
26	122	112	110	101	97	97	100	95	96	**94**
27	150	110	103	105	101	103	**100**	104	**100**	**100**
28	414	350	316	**256**	298	282	293	274	276	281
29	1199	**1101**	1208	−3102	1162	1166	1236	1227	2728	−3091
30	**235**	264	243	245	244	245	240	250	242	**235**
31	204	217	215	216	189	192	185	181	181	**176**
32	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**	**24**
33	**2845**	2895	2911	2893	2883	2906	2907	2884	2893	2879
34	29	**27**	**27**	**27**	**27**	**27**	**27**	**27**	**27**	**27**
35	2288	2317	2081	1851	1880	1860	**1800**	1819	1835	1832
36	1656	1614	1533	1566	1587	1514	**1240**	1441	1500	1489
37	2505	1509	1467	1815	1292	1453	**948**	1690	1363	1397
38	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**
39	**839**	1126	1333	1294	1052	1174	1079	1109	1150	874
40	**45**	**45**	46	46	46	46	46	46	46	46
41	−3286	2694	2879	2087	2641	2855	1906	1497	1299	**1184**
42	**1445**	2710	2067	2417	−3077	2495	2443	1642	−3068	−3074
43	**530**	786	652	591	1067	968	998	997	2058	834
44	358	330	**262**	463	395	540	782	1088	788	1370
45	268	**230**	264	277	276	262	284	272	271	271
46	31	30	28	**27**	**27**	**27**	**27**	**27**	**27**	**27**
47	1400	1407	1349	1434	1404	1361	**1088**	1349	1104	1128
48	122	110	**102**	109	106	106	109	107	109	107
Tot	38641	38146	36616	39093	36685	36050	34454	34664	38504	37014

TABLE 3

Minimum and maximum number of function evaluations for given memory size and ratio of max/min. Note that the ratio of max/min does not make sense for problems where there was no convergence; we left a blank in these cases.

Prob	n	Min	Max	max/min
1	5000	15	19	1.2667
2	1000	37	107	2.8919
3	1001	3044	−3091	
4	1000	356	380	1.0674
5	50	183	257	1.4044
6	4970	391	810	2.0716
7	4970	436	482	1.1055
8	5000	66	81	1.2273
9	10000	776	1261	1.625
10	10000	1357	2168	1.5976
11	10000	1711	2462	1.4389
12	10000	1664	2162	1.2993
13	51	59	122	2.0678
14	3000	228	243	1.0658
15	3000	166	211	1.2711
16	3000	169	214	1.2663
17	3000	171	218	1.2749
18	3000	1653	2031	1.2287
19	3000	223	257	1.1525
20	3000	190	235	1.2368
21	3000	177	223	1.2599
22	110	95	325	3.4211
23	110	95	458	4.8211
24	110	816	1292	1.5833

Prob	n	Min	Max	Max/Min
25	110	573	1280	2.2339
26	50	94	122	1.2979
27	100	100	150	1.5
28	100	256	414	1.6172
29	691	1101	−3102	
30	1024	235	264	1.1234
31	1024	176	217	1.233
32	5000	24	24	1
33	500	2845	2911	1.0232
34	10000	27	29	1.0741
35	1024	1800	2317	1.2872
36	1024	1240	1656	1.3355
37	1000	948	2505	2.6424
38	1000	33	33	1
39	10000	839	1333	1.5888
40	10000	45	46	1.0222
41	127	1184	3286	2.7753
42	10000	1445	−3077	
43	10000	530	2058	3.883
44	10000	262	1370	5.229
45	10000	230	284	1.2348
46	50	27	31	1.1481
47	10000	1088	1434	1.318
48	10000	102	122	1.1961

remove the pair (s_{M-1}, y_{M-1}) from the set used to form the quasi-Newton matrices and compare the matrices $H^{m,2}$. That is, we compute the values

$$e_m = \left\| H^{m,2} y_{M-1} - s_{M-1} \right\|_2^2, \ m = 2, \ldots, M. \tag{3.1}$$

(This approach is somewhat analogous to the technique of cross-validation in statistics.) We could then pick m as the $\arg\min_{2 \le m \le M}\{e_m\}$. But we go one step further and take

$$e_1 = \left\| H_0 y_{M-1} - s_{M-1} \right\|_2^2, \tag{3.2}$$

which allows us to consider not using more than the most recent pair. Now we can set

$$m^* = \arg \min_{1 \le m \le M} \{e_m\}$$

as the memory size to use at this iteration. In this way we will always use at least the most recent pair (s_{M-1}, y_{M-1}) so that $m^* \ge 1$ and the step taken is $-H^{m^*,1} \nabla f(x_c)$.

Recall that the work to compute the two-loop recursion of Figure 1 using m pairs is approximately $4nm$ operations. Here we compute this for

$$m = 1, 2, \ldots, M,$$

so we do $4nM(M+1)/2 \approx 2nM^2$ operations. However, by closely examining the two-loop recursion, we see that if the first loop is run for $i = M-1, \ldots, 0$, the value of q computed when $i = m$ is the final value of q needed to compute $H^{m,1}$. Thus we can eliminate half of this work by computing and saving all M of the intermediate q

vectors. Thus the first loop needs to be executed only once, but M additional vectors of length n need to be saved. This is a significant increase over the normally small cost of computing an L-BFGS step. However, there is still only one gradient evaluation required per iteration (plus the possible additional evaluations in the linesearch), so for a problem where f is very expensive to evaluate, this may be a minor cost. In section 4, we describe an approach that can reduce the computational overhead of this method significantly.

Given x_c, an approximation to x^*

1. Compute $H_0 = \gamma I$ with γ from (2.5) (or compute another appropriate initial matrix);
2. Compute $e_m, m = 1, \ldots, M$ by (3.1) and (3.2);
3. Set $m^* = \arg\min_m \{e_m\}$;
4. Compute the search direction
$$p = -H^{m^*,1} \nabla f(x_c);$$
5. Linesearch: Choose α to satisfy the strong Wolfe conditions (2.3);
6. Set $x_+ = x_c + \alpha p$;
7. Evaluate $f(x_+)$ and $\nabla f(x_+)$;
8. If convergence test is satisfied then stop;
9. Set $(s_i, y_i) = (s_{i+1}, y_{i+1}),\ i = 0, \ldots, M-2$;
10. Set $s_{M-1} = \alpha p$ and $y_{M-1} = \nabla f(x_+) - \nabla f(x_c)$;
11. Set $x_c = x_+$; go to 1;

FIG. 2. *AL-BFGS, the adaptive L-BFGS algorithm. Obvious modifications will be made until we have M pairs stored.*

Our adaptive algorithm, which we call AL-BFGS, is given in Figure 2. In Table 4 we give the results of applying this strategy to the problems of Table 1. Here, and in all subsequent runs, we have used a value of $M = 50$. For each problem we give the total number of function evaluations and the ratio of that number to the minimum and maximum values from Table 3. We see that AL-BFGS gives a significant improvement. First, the total number of iterations (32910) is between 5 and 16 percent better than any fixed memory-size strategy. However, we did fail to solve two problems within 3000 iterations, one of which (33) was solved by L-BFGS with all m and one (3) by L-BFGS for only one value of m. More importantly, the median of the ratios of the adaptive to the minimum of the L-BFGS is 1.09, so the median performance is almost as good as the best over all fixed strategies. And we note that we actually did better than or equal to the best on 16 of the problems. Finally, when we did worse than the worst, it was not by very much.

It is interesting to observe the memory size, m, used at each iteration to determine if there are any discernible trends. Thus, for each problem we kept track of the memory size used at each iteration. This is much too much information to display here, so we give our summary. To see if there was a trend towards increasing memory size as convergence was approached, we divided the iterations into 15 bins as follows: Let I be the total number of iterations for a given problem. Then bin 1 contains iterations 1 through $(I/15)$, bin 2 iterations $(I/15 + 1)$ through $(2I/15)$, etc. For each bin, we computed the average memory size used divided by the maximum possible average. In looking at these results, we did not see any trend towards larger memory usage as

TABLE 4

Evaluations and ratios of evaluations of AL-BFGS to the minimum and maximum number of evaluations for L-BFGS. A negative number in the number of function evaluations implies nonconvergence in 3000 *iterations; in these cases a blank was entered in the table, and these values were not used in the average or mean values.*

Prob	Evals	Adapt/Min	Adapt/Max	Prob	Evals	Adapt/Min	Adapt/Max
1	13	0.86667	0.68421	25	743	1.2967	0.58047
2	59	1.5946	0.5514	26	129	1.3723	1.0574
3	−3152			27	114	1.14	0.76
4	396	1.1124	1.0421	28	280	1.0938	0.67633
5	273	1.4918	1.0623	29	403	0.36603	0.12992
6	672	1.7187	0.82963	30	258	1.0979	0.97727
7	506	1.1606	1.0498	31	183	1.0398	0.84332
8	70	1.0606	0.8642	32	23	0.95833	0.95833
9	759	0.97809	0.6019	33	−3623		
10	1022	0.75313	0.4714	34	27	1	0.93103
11	1874	1.0953	0.76117	35	2551	1.4172	1.101
12	1521	0.91406	0.70352	36	1704	1.3742	1.029
13	94	1.5932	0.77049	37	2911	3.0707	1.1621
14	251	1.1009	1.0329	38	29	0.87879	0.87879
15	186	1.1205	0.88152	39	652	0.77712	0.48912
16	179	1.0592	0.83645	40	47	1.0444	1.0217
17	182	1.0643	0.83486	41	2074	1.7517	0.63116
18	1610	0.97399	0.79271	42	464	0.32111	0.1508
19	262	1.1749	1.0195	43	233	0.43962	0.11322
20	207	1.0895	0.88085	44	137	0.5229	0.1
21	211	1.1921	0.94619	45	216	0.93913	0.76056
22	111	1.1684	0.34154	46	27	1	0.87097
23	111	1.1684	0.24236	47	1484	1.364	1.0349
24	728	0.89216	0.56347	48	149	1.4608	1.2213

Total Evals	Avg. Adapt/Min	Avg. Adapt/Max	Median Adapt/Min	Median Adapt/Max
32910	1.132	0.7984	1.0945	0.84332

TABLE 5

For each problem, we divided the iterations into 15 *bins as described above. Then, for each bin, we computed the ratio of memory used vs. maximum possible memory to use. In this table we show the average of these ratios for each problem.*

Prob	Avg	Prob	Avg	Prob	Avg	Prob	Avg
1	0.0	13	0.3	25	0.2	37	0.3
2	0.5	14	0.4	26	0.2	38	0.2
3	0.3	15	0.4	27	0.5	39	0.4
4	0.1	16	0.3	28	0.3	40	0.3
5	0.2	17	0.3	29	0.4	41	0.4
6	0.3	18	0.3	30	0.2	42	0.3
7	0.3	19	0.3	31	0.3	43	0.3
8	0.3	20	0.4	32	0.5	44	0.3
9	0.4	21	0.3	33	0.1	45	0.2
10	0.3	22	0.5	34	0.4	46	0.7
11	0.3	23	0.5	35	0.3	47	0.3
12	0.3	24	0.3	36	0.4	48	0.2

convergence was neared. Overall, the memory usage tended to be quite small. We show in Table 5 the average of the averages of relative memory size over the 15 bins. As is clear, this average is rarely over .5, and the average of the averages in Table 5 is .32. This is in contrast to the results given in the next section.

4. An efficient weighted measure for determining m. In this section we consider a different measure for determining the number of previous (s, y) pairs to use at the current iteration. We are motivated by wanting to use the compact forms of the update formulas for H_k and its inverse B_k (see [3]), since this would allow us to create a more efficient test. Recall that the number of operations per iteration to use AL-BFGS is $\approx 2nM^2$ along with the storage of $3M$ additional vectors of length n ($2M$ for the M (s, y) pairs and M additional vectors for the q vectors in Figure 1). This is significantly more than the $\approx 4Mn$ operations required and $2Mn$ storage locations to compute the L-BFGS step (using the largest value of memory). We show here that, using this metric with the compact form, we can reduce this extra overhead to $\approx 2Mn+O(M^3)$ operations with the additional storage of only $O(M^2)$ storage locations, using only existing data. For large n, this is about 50% additional work compared to L-BFGS. We summarize these results in Table 6 after we give the details of the new measure.

As above, we drop the iteration index k to simplify notation. Then, for $m = 2, \ldots, M$, we consider the error measure weighted by $B^{1/2}$ given by

$$e^{m,2} = \left\| (B^{m,2})^{1/2}(s_{M-1} - H^{m,2}y_{M-1}) \right\|_2^2 = s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1} - 2s_{M-1}^{\mathsf{T}} y_{M-1} + y_{M-1}^{\mathsf{T}} H^{m,2} y_{M-1}. \tag{4.1}$$

Using the compact forms of the update formulas for B and H, the scalar quantities $s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1}$ and $y_{M-1}^{\mathsf{T}} H^{m,2} y_{M-1}$ can be computed much more efficiently than the quantities needed for the measure in the previous section. The weighting by $B^{1/2}$ is further justified by recalling that BFGS can be derived as a least-change secant update using a similar norm [4].

To define the compact form for the update $B^{m,2}$ and its inverse we proceed as follows. At the current iteration we have available M of the (s, y) pairs. For $j \geq i$ we let

$$S^{i,j} = \{s_{M-j}, \ldots, s_{M-i}\}.$$

Define $Y^{i,j}$ analogously. Now define

$$\begin{aligned} \mathcal{Z}^{i,j} &= (S^{i,j})^{\mathsf{T}} Y^{i,j}, \\ \mathcal{S}^{i,j} &= (S^{i,j})^{\mathsf{T}} S^{i,j}, \\ \mathcal{Y}^{i,j} &= (Y^{i,j})^{\mathsf{T}} Y^{i,j} \end{aligned}$$

and then let

$$\begin{aligned} L^{i,j} &= \text{StrictLowerTriangle}(\mathcal{Z}^{i,j}), \\ D^{i,j} &= \text{Diag}(\mathcal{Z}^{i,j}), \\ R^{i,j} &= \text{UpperTriangle}(\mathcal{Z}^{i,j}), \text{ and} \\ \mathcal{L}^{i,j} &= L^{i,j}(D^{i,j})^{-1}(L^{i,j})^{\mathsf{T}}. \end{aligned}$$

Using these quantities, we restate the formula in [3] for $s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1}$. An examination of this formula shows that it will be efficient for our purposes only if B_0 is taken to be σI, where $\sigma = 1/\gamma$. Otherwise there is no good way to update the information used for $s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1}$ to get $s_{M-1}^{\mathsf{T}} B^{m+1,2} s_{M-1}$. Using this notation,

the formula becomes

$$s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1} = \sigma s_{M-1}^{\mathsf{T}} s_{M-1}$$
$$- s_{M-1}^{\mathsf{T}} \begin{bmatrix} \sigma S^{m,2} & Y^{m,2} \end{bmatrix} \begin{bmatrix} \sigma (S^{m,2})^{\mathsf{T}} S^{m,2} & L^{m,2} \\ (L^{m,2})^{\mathsf{T}} & -D^{m,2} \end{bmatrix}^{-1} \begin{bmatrix} \sigma (S^{m,2})^{\mathsf{T}} \\ (Y^{m,2})^{\mathsf{T}} \end{bmatrix} s_{M-1}. \tag{4.2}$$

It is shown in [3] that the inverse matrix in (4.2) can be conveniently factored to obtain

$$\begin{aligned} s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1} = \sigma s_{M-1}^{\mathsf{T}} s_{M-1} & \\ - s_{M-1}^{\mathsf{T}} W^{\mathsf{T}} \begin{bmatrix} -(D^{m,2})^{1/2} & (D^{m,2})^{-1/2} (L^{m,2})^{\mathsf{T}} \\ 0 & J \end{bmatrix}^{-1} & \\ \times \begin{bmatrix} (D^{m,2})^{1/2} & 0 \\ -L^{m,2} (D^{m,2})^{-1/2} & J^{\mathsf{T}} \end{bmatrix}^{-1} W s_{M-1}, & \end{aligned} \tag{4.3}$$

where

$$W = \begin{bmatrix} (Y^{m,2})^{\mathsf{T}} \\ \sigma (S^{m,2})^{\mathsf{T}} \end{bmatrix},$$
$$J = \mathrm{Chol}(\sigma * \mathcal{S}^{m,2} + \mathcal{L}^{m,2}),$$

and $\mathrm{Chol}(A)$ denotes the upper right triangular Cholesky factor of A.

An efficient strategy for choosing m using the new metric requires the quantities $\mathcal{Z}^{m,2}$, $\mathcal{Y}^{m,2}$, and $\mathcal{S}^{m,2}$ for each value of m. To efficiently calculate these matrices, we update the $(M \times M)$ version for each, once at each iteration, and then extract the appropriate $(m \times m)$ submatrix when required. The work in updating these $(M \times M)$ matrices is $\approx 4Mn$ for the four new columns or rows. The work in all of the subsequent computations does not depend on n, and so it is relatively inexpensive, but we still have to be careful.

By writing out the explicit forms for the inverses in (4.3) and multiplying, we can show that we can obtain $s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1}$ by the following: Define

$$a = (Y^{m,2})^{\mathsf{T}} s_{M-1},$$
$$b = \sigma * (S^{m,2})^{\mathsf{T}} s_{M-1}.$$

The quantities a and b are easily extracted from $\mathcal{Z}$ and $\mathcal{S}$, respectively. Let

$$r = b + L^{m,2} (D^{m,2})^{-1} a.$$

Now solve

$$Jq = r,$$

and, finally, compute

$$s_{M-1}^{\mathsf{T}} B^{m,2} s_{M-1} = \sigma s_{M-1}^{\mathsf{T}} s_{M-1} - q^{\mathsf{T}} q + a^{\mathsf{T}} (D^{m,2})^{-1} a.$$

Proceeding as above we obtain the formula for the computation of $y_{M-1}^{\mathsf{T}} H^{m,2} y_{M-1}$:

$$\begin{aligned} y_{M-1}^{\mathsf{T}} H^{m,2} y_{M-1} = \ & \gamma y_{M-1}^{\mathsf{T}} y_{M-1} \\ & + \left[(R^{m,2})^{-1} (S^{m,2})^{\mathsf{T}} y_{M-1} \right]^{\mathsf{T}} \left[D^{m,2} + \gamma (\mathcal{Y}^{m,2})^{\mathsf{T}} \right] \left[(R^{m,2})^{-1} (S^{m,2})^{\mathsf{T}} y_{M-1} \right] \\ & - 2\gamma y_{M-1}^{\mathsf{T}} Y (R^{m,2})^{-1} (S^{m,2})^{\mathsf{T}} y_{M-1}. \end{aligned} \tag{4.4}$$

To compute this we perform the following: Define

$$c = (S^{m,2})^{\mathsf{T}} y_{M-1},$$
$$d = (Y^{m,2})^{\mathsf{T}} y_{M-1}.$$

Again, these are easily extracted from $\mathcal{Z}$ and $\mathcal{Y}$, respectively. Then solve

$$Rz = c$$

and compute

$$y_{M-1}^{\mathsf{T}} H^{m,2} y_{M-1} = \gamma y_{M-1}^{\mathsf{T}} y_{M-1} + z^{\mathsf{T}}(D^{m,2} + \gamma \mathcal{Y}^{m,2})z - 2\gamma d^{\mathsf{T}} z.$$

We see that the work is $O(nM)$ to do the updates plus, for a given value of m, $O(m^3)$, where the $O(m^3)$ arises due to the need to compute the Cholesky factorization of an $(m \times m)$ matrix. Although this appears to be quite efficient, we need to do this for each value of $m = 1, \ldots, M$. Thus, proceeding as above would require $O(m^3)$ operations for each value of m and the total work would be $O(M^4)$. We can, however, easily update the Cholesky factor J for testing $(m, 2)$ pairs from the previous calculations, computing $(m-1, 2)$ pairs in $O(m^2)$ operations, resulting in total work of $O(M^3)$.

It is important to keep in mind that the above estimates of the work assume that B_0 and, by implication, H_0 are scalar multiples of the identity matrix. If not, the work is considerably more since little of the work at one iteration can be saved for use at subsequent iterations. Note, in contrast, the matrix H_0 enters the two-loop recursion in one isolated place. Thus, if we have a complicated initial matrix, the formulas in this section may not be appropriate. We call our final algorithm using the new metric given by (4.1) A_WL-BFGS.

Having determined the best value m^* for the memory by the above procedure we still need to compute the search direction $p = -H^{m^*,1}\nabla f(x_c)$. This can be done by simply applying the two-loop recursion [1] for a cost of about $4m^*n \leq 4Mn$ operations, making a total cost of about $8Mn$. We have taken this approach in our implementation. However, the paper [3] describes a procedure for using the compact form and two saved vectors of length M to compute the direction p in approximately $2m^*n$ additional operations. This is a total step computational cost bounded by $6Mn$, a 50% increase over L-BFGS.

We summarize the storage and work per iteration for the original L-BFGS algorithm and for each of our two adaptive methods in Table 6. Notice that all of the methods require $2Mn$ storage for the (s, y) pairs (we are using the maximum value of M for the L-BFGS method), but that the storage beyond that is much less for A_WL-BFGS than for AL-BFGS since $M \ll n$. Similarly for the extra operations.

TABLE 6

A comparison of the storage and the work per iteration for L-BFGS and for each of our two adaptive algorithms. Note that all require $2Mn$ to store the (s, y) pairs.

Algorithm	Storage	Work/Iteration
L-BFGS	$2Mn$	$4Mn$
AL-BFGS	$2Mn + Mn$	$\approx 2M^2n$
A_WL-BFGS	$2Mn + O(M^2)$	$6Mn + O(M^3)$

TABLE 7

Evaluations and ratios of evaluations of A_W L-BFGS to the minimum and maximum number of evaluations for L-BFGS. A negative number in the number of function evaluations implies nonconvergence in 3000 *iterations, and blanks were entered for the ratios.*

Prob	Evals	Adapt/Min	Adapt/Max
1	13	0.86667	0.68421
2	40	1.0811	0.37383
3	−3063		
4	376	1.0562	0.98947
5	196	1.071	0.76265
6	393	1.0051	0.48519
7	459	1.0528	0.95228
8	69	1.0455	0.85185
9	1129	1.4549	0.89532
10	2195	1.6175	1.0125
11	2448	1.4307	0.99431
12	2377	1.4285	1.0994
13	62	1.0508	0.5082
14	240	1.0526	0.98765
15	176	1.0602	0.83412
16	170	1.0059	0.79439
17	173	1.0117	0.79358
18	1780	1.0768	0.87642
19	247	1.1076	0.96109
20	218	1.1474	0.92766
21	200	1.1299	0.89686
22	98	1.0316	0.30154
23	98	1.0316	0.21397
24	785	0.96201	0.60759
25	806	1.4066	0.62969
26	95	1.0106	0.77869
27	108	1.08	0.72
28	273	1.0664	0.65942
29	1328	1.2062	0.42811
30	233	0.99149	0.88258
31	174	0.98864	0.80184
32	22	0.91667	0.91667
33	3045	1.0703	1.046
34	29	1.0741	1
35	1761	0.97833	0.76003
36	1498	1.2081	0.90459
37	1433	1.5116	0.57206
38	30	0.90909	0.90909
39	476	0.56734	0.35709
40	44	0.97778	0.95652
41	1194	1.0084	0.36336
42	2355	1.6298	0.76536
43	1003	1.8925	0.48737
44	640	2.4427	0.46715
45	272	1.1826	0.95775
46	27	1	0.87097
47	1356	1.2463	0.94561
48	123	1.2059	1.0082

Total Evals	Avg. Adapt/Min	Avg. Adapt/Max	Median Adapt/Min	Median Adapt/Max
35330	1.1564	0.78262	1.0664	0.86141

In Table 7 we see that the number of function evaluations for A_WL-BFGS is better than L-BFGS on all but two fixed values of m, 35 and 40. The median of the ratios of the number of evaluations to the minimum of the L-BFGS, however, is 1.07, which is quite good. We note also that A_WL-BFGS was better than or equal to the best of the L-BFGS methods on 10 of the problems and worse than the worst on only 4 problems. It also had just one failure due to iteration count compared to two for the unweighted method.

As for AL-BFGS in section 3 we analyze the memory used at each iteration. We performed the same operation of creating 15 bins and computing the ratio of the average memory size used versus the maximum possible memory size. Here, there is a strong tendency to use most, if not all, of the available pairs. We show in Table 8 the average memory usage is often $\geq .9$ with an average of .74. (Recall that for AL-BFGS the average was .32.) It is not completely clear why this difference should occur using the weighted measure, but the fact that the weighting by $B^{1/2}$ used here is similar to the weighting used to derive the BFGS method may be relevant. We also note that the best of the L-BFGS methods occurred for $m = 35$ and that $.74 \times 50 = 37$.

In the next section, we summarize the results from our computations and do some additional comparisons.

TABLE 8
For each problem, we show the average of the memory usage ratios, as in Table 5.

Prob	Avg	Prob	Avg	Prob	Avg	Prob	Avg
1	0.0	13	0.6	25	0.9	37	0.9
2	0.6	14	0.8	26	0.6	38	0.2
3	1.0	15	0.8	27	0.9	39	0.9
4	0.5	16	0.8	28	0.6	40	0.4
5	0.5	17	0.8	29	1.0	41	1.0
6	1.0	18	1.0	30	0.7	42	0.9
7	0.9	19	0.9	31	0.7	43	0.9
8	0.5	20	0.8	32	0.4	44	0.9
9	1.0	21	0.8	33	0.3	45	0.3
10	0.9	22	0.8	34	0.3	46	0.9
11	1.0	23	0.8	35	0.9	47	0.9
12	1.0	24	0.9	36	0.9	48	0.3

5. Summary. In this section we make an evaluation of the comparative performance of our new adaptive methods and the standard fixed-memory L-BFGS methods. First we show in Table 9 a summary of the computational results of sections 2, 3, and 4. Both the unweighted $Hy - s$ and weighted adaptive methods often have function-evaluation counts close to those of the best L-BFGS method (see Table 2). The median value of Ev/Min for the weighted method is smaller than for the unweighted method. However, the average value for this ratio is slightly worse for the weighted method, as is the total number of function evaluations. Also as noted above the unweighted method fails on problems 3 and 33, and the weighted method on problem 3. It is noteworthy that the median values of Ev/Min for the weighted method imply that on half of the problems, the unweighted adaptive method, AL-BFGS, has an evaluation count within 9.5% of L-BFGS with the best choice of m for that problem. For the weighted method, A_WL-BFGS, performance on half the problems is within 6.4% of the best m.

To further compare our adaptive methods with the classic L-BFGS methods, we show a Dolan–Moré performance plot [5] comparing AL-BFGS, A_WL-BFGS, and the L-BFGS method with the best-performing L-BFGS method, namely, L-BFGS with $m = 35$, as well as $m = 5$ (see Figure 3). On this plot a data point for a given method at some value of performance ratio gives the fraction of test problems for which the function evaluations count for the given method is less than the performance value times the function evaluations count of the best method on the problem. It is clear that all three methods are pretty close. The weighted method is best on more problems, and seems to slightly dominate LBFGS-35 overall. The important fact is that both adaptive methods are at least competitive with standard L-BFGS using the a posteriori best value of m for this problem set. In our experience, many users of L-BFGS simply pick a small value of m. Thus we also include the LBFGS-5 to show how much our strategies improve performance over this common choice.

6. Implementation. A code to implement these methods was written in C++, making extensive use of the Sundance package (see [12] and [11]), which is part of Trilinos (see [8]). Sundance is primarily designed for solving partial differential equations, but it has an associated package called Playa (see [9]) that provides a high-level interface for many of the linear algebraic computations that we needed. We also made use of AMPL (see [6]) in which the CUTE test problems were written.

One of the complicating factors in the code is the necessity to keep track of the most recent (s, y) pairs. We have implemented this using the C++ Standard Template

TABLE 9

L-BFGS summary, Hy-s summary, and weighted summary, with total evaluations and average and median of ratios.

		L-BFGS			Hy-s			Weighted		
Prob	n	Min Ev	Max Ev	Max/Min	Ev	Ev/Min	Ev/Max	Ev	Ev/Min	Ev/max
1	5000	15	19	1.2667	13	0.86667	0.68421	13	0.86667	0.68421
2	1000	37	107	2.8919	59	1.5946	0.5514	40	1.0811	0.37383
3	1001	3044	3091	1.0154	−3152	1.0355	1.0197	−3063	1.0062	0.99094
4	1000	356	380	1.0674	396	1.1124	1.0421	376	1.0562	0.98947
5	50	183	257	1.4044	273	1.4918	1.0623	196	1.071	0.76265
6	4970	391	810	2.0716	672	1.7187	0.82963	393	1.0051	0.48519
7	4970	436	482	1.1055	506	1.1606	1.0498	459	1.0528	0.95228
8	5000	66	81	1.2273	70	1.0606	0.8642	69	1.0455	0.85185
9	10000	776	1261	1.625	759	0.97809	0.6019	1129	1.4549	0.89532
10	10000	1357	2168	1.5976	1022	0.75313	0.4714	2195	1.6175	1.0125
11	10000	1711	2462	1.4389	1874	1.0953	0.76117	2448	1.4307	0.99431
12	10000	1664	2162	1.2993	1521	0.91406	0.70352	2377	1.4285	1.0994
13	51	59	122	2.0678	94	1.5932	0.77049	62	1.0508	0.5082
14	3000	228	243	1.0658	251	1.1009	1.0329	240	1.0526	0.98765
15	3000	166	211	1.2711	186	1.1205	0.88152	176	1.0602	0.83412
16	3000	169	214	1.2663	179	1.0592	0.83645	170	1.0059	0.79439
17	3000	171	218	1.2749	182	1.0643	0.83486	173	1.0117	0.79358
18	3000	1653	2031	1.2287	1610	0.97399	0.79271	1780	1.0768	0.87642
19	3000	223	257	1.1525	262	1.1749	1.0195	247	1.1076	0.96109
20	3000	190	235	1.2368	207	1.0895	0.88085	218	1.1474	0.92766
21	3000	177	223	1.2599	211	1.1921	0.94619	200	1.1299	0.89686
22	110	95	325	3.4211	111	1.1684	0.34154	98	1.0316	0.30154
23	110	95	458	4.8211	111	1.1684	0.24236	98	1.0316	0.21397
24	110	816	1292	1.5833	728	0.89216	0.56347	785	0.96201	0.60759
25	110	573	1280	2.2339	743	1.2967	0.58047	806	1.4066	0.62969
26	50	94	122	1.2979	129	1.3723	1.0574	95	1.0106	0.77869
27	100	100	150	1.5	114	1.14	0.76	108	1.08	0.72
28	100	256	414	1.6172	280	1.0938	0.67633	273	1.0664	0.65942
29	691	1101	3102	2.8174	403	0.36603	0.12992	1328	1.2062	0.42811
30	1024	235	264	1.1234	258	1.0979	0.97727	233	0.99149	0.88258
31	1024	176	217	1.233	183	1.0398	0.84332	174	0.98864	0.80184
32	5000	24	24	1	23	0.95833	0.95833	22	0.91667	0.91667
33	500	2845	2911	1.0232	−3623	1.2735	1.2446	3045	1.0703	1.046
34	10000	27	29	1.0741	27	1	0.93103	29	1.0741	1
35	1024	1800	2317	1.2872	2551	1.4172	1.101	1761	0.97833	0.76003
36	1024	1240	1656	1.3355	1704	1.3742	1.029	1498	1.2081	0.90459
37	1000	948	2505	2.6424	2911	3.0707	1.1621	1433	1.5116	0.57206
38	1000	33	33	1	29	0.87879	0.87879	30	0.90909	0.90909
39	10000	839	1333	1.5888	652	0.77712	0.48912	476	0.56734	0.35709
40	10000	45	46	1.0222	47	1.0444	1.0217	44	0.97778	0.95652
41	127	1184	3286	2.7753	2074	1.7517	0.63116	1194	1.0084	0.36336
42	10000	1445	3077	2.1294	464	0.32111	0.1508	2355	1.6298	0.76536
43	10000	530	2058	3.883	233	0.43962	0.11322	1003	1.8925	0.48737
44	10000	262	1370	5.229	137	0.5229	0.1	640	2.4427	0.46715
45	10000	230	284	1.2348	216	0.93913	0.76056	272	1.1826	0.95775
46	50	27	31	1.1481	27	1	0.87097	27	1	0.87097
47	10000	1088	1434	1.318	1484	1.364	1.0349	1356	1.2463	0.94561
48	10000	102	122	1.1961	149	1.4608	1.2213	123	1.2059	1.0082
Tot		34454	39093		32910			35330		
Avg						1.1329	0.7814		1.1532	0.77048
Med						1.0945	0.83988		1.0633	0.84299

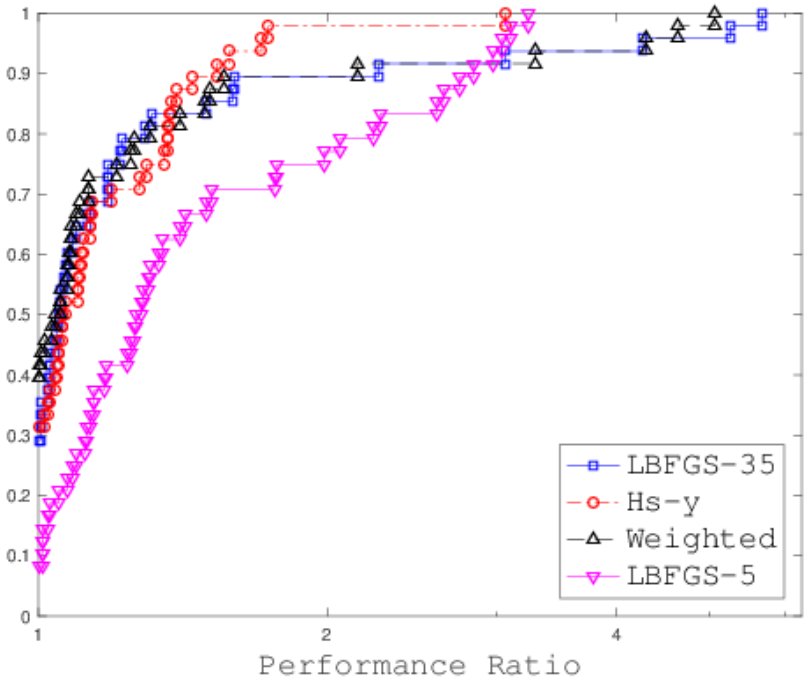

FIG. 3. *Dolan–Moré plot comparing evaluation cost for the basic adaptive method, the weighted adaptive method, and classic BFGS with* $m = 35$.

Library (see [16]) deque data structure. The deque guarantees a constant amount of work when adding or deleting items from either end of the list. This has greatly simplified the code.

Finally, we have made extensive use of MATLAB to test the code.

7. Conclusions and future work. We have addressed the problem that there is no general guidance for how to choose the memory size in the L-BFGS method. We have provided two measures that assess the value of using the individual (s, y) pairs and thus have shown how to create adaptive L-BFGS methods. Without our modifications, the performance of the L-BFGS method often varies widely as a function of the memory size. With our adaptive schemes, the performance is quite stable; the algorithms are often much more efficient than any fixed-size memory. On individual problems, they are almost never as bad as the worst performance of the standard L-BFGS method over a range of memory sizes, and they are often better than the best performance. One of our algorithms, AL-BFGS, adds some modest computational overhead, while the other one, A_WL-BFGS, adds considerably less. We think that the success demonstrated here suggests some future efforts, some examples of which follow:

- We investigated, with mixed results, some strategies for using more than one test pair. We think that there may be some promise here, but more research and testing are necessary to determine good, general strategies for using more than the most recent (s, y).
- We need to understand better why AL-BFGS tends to use a smaller value of m than A_WL-BFGS.

- At this point it is hard to make a firm general recommendation for which measure to use. Clearly, if $H_0 \neq \gamma I$, then AL-BFGS is the obvious choice. Often in optimization we need to solve many related problems, so doing some preliminary testing may yield a superior choice of method.
- Since some of our calculations can be performed in parallel, we need to examine carefully the potential parallel performance of any resulting algorithm.
- It would be interesting to test the algorithm on larger problems, especially those arising from PDE-constrained optimization. Such problems sometimes give rise to the ability to use preconditioners, which, in our context, amounts to a more suitable initial H_0 that is not a scalar multiple of the identity.
- The compact form used extensively in the weighted measure was developed originally for the extension of L-BFGS to handle simple bounds. (See [2].) Since we have many of the same computations here, it would be interesting to combine the two methods to create a more efficient algorithm to handle bound constraints.
- All of our runs were done with a fixed maximum memory size of 50. We observed in some of our runs that the algorithm nearly always chose m to be 50, so it may be of value to try to develop an adaptive scheme to dynamically increase (or decrease) M depending on how m varies.

Acknowledgments. The authors would like to thank Mehiddin Al-Baali for helpful comments on an early draft of the paper and Albert Berahas for help with numerical testing and with Dolan–Moré plots. We also thank the referees for many suggestions that led to certain clarifications in the presentation. We are also grateful to Sandia National Laboratories at Livermore for providing a visiting position to R. H. Byrd during which much of this work was done.

REFERENCES

[1] I. Bongartz, A. R. Conn, N. Gould, and Ph. L. Toint, *CUTE: Constrained and unconstrained testing environment*, ACM Trans. Math. Softw., 21 (1995), pp. 123–160, https://doi.org/10.1145/200979.201043.

[2] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization*, SIAM J. Sci. Comput., 16 (1995), pp. 1190–1208, https://doi.org/10.1137/0916069.

[3] R. H. Byrd, J. Nocedal, and R. B. Schnabel, *Representations of quasi-Newton matrices and their use in limited-memory methods*, Math. Programming Ser. A, 63 (1994), pp. 129–156.

[4] J. E. Dennis, Jr., and J. J. Moré, *Quasi-Newton methods, motivation and theory*, SIAM Rev., 19 (1977), pp. 46–89, https://doi.org/10.1137/1019005.

[5] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Math. Programming, 91 (2002), pp. 201–213.

[6] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Thomson/Brooks/Cole, 2003, https://books.google.com/books?id=Ij8ZAQAAIAAJ.

[7] J. C. Gilbert and J. Nocedal, *Automatic differentiation and the step computation in the limited memory BFGS method*, Appl. Math. Lett., 6 (1993), pp. 47–50.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[9] V. E. Howle, R. C. Kirby, K. Long, B. Brennan, and K. Kennedy, *Playa: High performance programmable linear algebra*, Sci. Programming, 20 (2012), pp. 257–273.

[10] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Math. Programming, 45 (1989), pp. 503–528.

[11] K. Long, P. T. Boggs, and B. G. van Bloemen Waanders, *Sundance: High-level software for PDE-constrained optimization*, Sci. Programming, 20 (2012), pp. 293–310.

[12] K. Long, R. Kirby, and B. van Bloemen Waanders, *Unified embedded parallel finite element computations via software-based Fréchet differentiation*, SIAM J. Sci. Comput., 32 (2010), pp. 3323–3351, https://doi.org/10.1137/09076920X.
[13] J. J. Moré and D. J. Thuente, *Line search algorithms with guaranteed sufficient decrease*, ACM Trans. Math. Softw., 20 (1994), pp. 286–307.
[14] J. Nocedal, *Updating quasi-Newton matrices with limited storage*, Math. Comp., 35 (1980), pp. 773–782.
[15] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., Springer, New York, 2006.
[16] P. J. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Library*, 1st ed., Prentice–Hall, Upper Saddle River, NJ, 2000.