SIAM J. SCI. COMPUT.
Vol. 20, No. 4, pp. 1430–1437

THE EFFICIENT COMPUTATION OF STRUCTURED GRADIENTS USING AUTOMATIC DIFFERENTIATION*

THOMAS F. COLEMAN† AND GUDBJORN F. JONSSON‡

Abstract. The advent of robust automatic differentiation tools is an exciting and important development in scientific computing. It is particularly noteworthy that the gradient of a scalar-valued function of many variables can be computed with essentially the same time complexity as required to evaluate the function itself. This is true, in theory, when the "reverse mode" of automatic differentiation is used (whereas the "forward mode" introduces an additional factor corresponding to the problem dimension). However, in practice, performance on large problems can be significantly (and unacceptably) worse than predicted. In this paper we illustrate that when natural structure is exploited, fast gradient computation can be recovered, even for large dimensional problems.

Key words. automatic differentiation, computational differentiation, gradients, optimization, inverse problems

AMS subject classifications. 65K05, 65Y99, 90C06, 90C30

PII. S1064827597320794

1. Introduction.

When solving a nonlinear optimization problem, the gradient ∇f of a function $f : \mathbf{R}^n \to \mathbf{R}$ is usually required. Given a code (in Fortran, C, or C++) that computes this function, automatic differentiation (AD) tools can be used to compute the gradient at a given point. This approach has advantages over both hand-coding and finite differences. Except when the function is fairly simple, hand-coding of the derivative function is a tedious and sometimes error-prone job. Even if symbolic software is used, the resulting expression can be complex and hard to work with. Automatic differentiation gives an accurate result; i.e., there are no truncation errors and no human errors.

Given a code for a function $F : \mathbf{R}^n \to \mathbf{R}^m$, AD uses the chain rule successively to compute the derivative matrix. AD has two basic modes, *forward mode* and *reverse mode*. The difference between these two is the way the chain rule is used to propagate the derivatives.

Given an $n \times t_V$ seed matrix V, forward mode computes the product JV, where J is the Jacobian of the function F. This takes time proportional to $t_V \cdot \omega(F)$, where $\omega(F)$ is the time to evaluate F. Reverse mode computes $W^T J$, where W is an $m \times t_W$ matrix, in time proportional to $t_W \cdot \omega(F)$. In the case of the gradient, W can be chosen as a 1×1 matrix for use with reverse mode, whereas forward mode requires V to be $n \times n$. Hence, reverse mode seems to be preferable to forward mode for gradient computations.

However, when using AD, memory usage is just as important as the number of

*Received by the editors April 28, 1997; accepted for publication (in revised form) March 19, 1998; published electronically March 23, 1999. This research was partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-90ER25013.A000 and by the Advanced Computing Research Institute, a unit of the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute.

http://www.siam.org/journals/sisc/20-4/32079.html

†Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca, NY 14850 (coleman@cs.cornell.edu).

‡Center for Applied Mathematics, Cornell University, Ithaca, NY 14850 (jonsson@cam.cornell.edu).

$$\boxed{\begin{array}{l} \textit{Solve for } y: \ \tilde{F}_E(x,y)=0 \\ \textit{``Solve'' for output } z: \ z-\bar{f}(x,y_1,y_2,\ldots,y_p) \ = \ 0. \end{array}}$$

FIG. 1. *Structured f-evaluation in compact form.*

flops. The memory used by forward mode is proportional to $t_V\mu(F)$, where $\mu(F)$ is the memory used to evaluate F. On the other hand, in reverse mode all the intermediate results have to be stored, so reverse mode often requires a huge amount of memory. Even for moderately sized problems, the memory requirements of reverse mode can surpass the internal memory available and cause greatly reduced performance.

Note that this applies only to the straightforward use of reverse mode, i.e., when a single forward sweep is made, storing all the intermediates, followed by a reverse sweep to compute W^TJ. Fortunately, the memory requirement can be significantly reduced by using techniques like the checkpointing scheme introduced by Griewank [8] and by taking advantage of the underlying structure of the function.

In this paper we will examine how structure can be exploited in the computation of gradients by AD. In particular, our approach is to break the problem into manageable pieces, defined by the natural structure of the problem, and whenever possible take advantage of sparsity. Numerical experiments are made to compare the various approaches, including a straightforward application of reverse mode.

Bischof et al. [2] explore the use of AD to compute the gradient of a partially separable function, i.e., a function $f:\mathbf{R}^n\to\mathbf{R}$ that can be written in the form

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

where each f_i depends on $p_i \ll n$ variables. The ADIFOR tool [1] is used for the numerical experiments in [2], and AD is compared to hand-coding and finite differences.

However, not all functions are partially separable and we will examine two such cases. Furthermore, ADIFOR has only forward mode and we explore the use of reverse mode as well.

When computing a sparse Jacobian matrix using AD, graph-coloring algorithms can be used to significantly reduce the amount of work. The algorithm described in [2] illustrates the use of one-sided coloring. Two-sided coloring [3] combines the powers of both forward and reverse mode by constructing thin matrices V and W so that the Jacobian J can be determined from the pair $(JV,\ W^TJ)$.

Coleman and Verma [4, 6] show how sparsity and structure can be exploited to compute Jacobian and Hessian matrices efficiently using AD. In the scalar-valued case they define a *structured* computation, evaluate $z=f(x), f:\mathbf{R}^n\to\mathbf{R}$, as follows:

$$\begin{array}{l} \text{Solve for } y_1\text{: } F_1(x,y_1)=0, \\ \text{Solve for } y_2\text{: } F_2(x,y_1,y_2)=0, \\ \qquad \vdots \\ \text{Solve for } y_p\text{: } F_p(x,y_1,\ldots,y_p)=0, \\ \text{Solve for } z\text{: } z-\bar{f}(x,y_1,\ldots,y_p)=0, \end{array}$$

where $\bar{f}$ is a scalar-valued function. If we define the "extended" function $\tilde{F}_E^T = (F_1^T, F_2^T, \ldots, F_p^T)$, then the program to evaluate f can be simply written as illustrated in Figure 1, where $y^T=(y_1^T,\ldots,y_p^T)$.

Differentiation of this program with respect to the original variables x as well as the intermediate variables y yields an "extended" Jacobian matrix:

$$J_E = \begin{pmatrix} (\tilde{F}_E)_x & (\tilde{F}_E)_y \\ \nabla_x f^T & \nabla_y f^T \end{pmatrix}. \tag{2}$$

Typically the Jacobian matrix $\tilde{J}_E = ((\tilde{F}_E)_x, (\tilde{F}_E)_y)$ is sparse; hence, sparse AD techniques can be applied to the function $\tilde{F}_E$ to obtain this derivative information efficiently. Note also that $(\tilde{F}_E)_y$ is block lower-triangular.

Obtaining $\nabla_x f$ from (2) boils down to a Schur-complement computation: eliminate the (2, 2)-block, $\nabla_y f^T$, using a block Gaussian transformation; the transformed (2, 1)-block will hold the desired result, i.e., $\nabla_x f^T$.

The two examples illustrated below fall into this general framework. They differ in how J_E is obtained and the tailoring of the Schur complement computation described above. Our experiments were performed on a Sun SPARC10 under Solaris, and the software used was ADOLC, ADMIT, and MATLAB. ADOLC [7] is an automatic differentiation tool for C/C++. ADMIT [5] is a MATLAB interface built on top of ADOLC that also includes the graph-coloring algorithms. It offers both one-sided and two-sided coloring.

For both forward and reverse mode, ADOLC stores all intermediate variables, as well as the operations performed, on *tapes*. (Thus the distinction between memory usage of forward and reverse mode mentioned above does not apply.) There are three tapes for real variables, integers, and operations. The amount of memory allocated for each tape is determined by the parameter "bufsize." We set it to 524288. If the number of entries for one of the three tapes exceeds this number, ADOLC will start writing the tapes on disk.

2. Problem 1: The gradient of an implicit function.

Inverse problems often lead to the minimization of an implicitly defined nonlinear function. Our first function class is in this category, where the evaluation of the objective function depends on the numerical solution of a (linear) system of partial differential equations. The function class has the form

$$f(x) = \bar{f}(y), \tag{3}$$

where y is the solution to the system

$$Ay = \tilde{F}(x) \tag{4}$$

and $A = A(x)$ is a sparse, symmetric, and positive definite matrix. Here $\bar{f}$ and f are scalar-valued. The gradient of f can be written as

$$\nabla_x f = (\tilde{J} - A_x y)^T A^{-1} \nabla_y \bar{f}. \tag{5}$$

Three different methods are used to compute the gradient:

1. *Straightforward application of reverse mode.* Reverse mode is used to take the derivative of the function $f(x)$. The code that is differentiated computes $f(x)$ using a sparse solver for symmetric and positive definite systems to solve the system $Ay = \tilde{F}$.
2. *Extended Jacobian.* Following the structural ideas of Coleman and Verma [4], we form the extended function

$$F_E : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} A(x)y - \tilde{F}(x) \\ \bar{f}(y) \end{pmatrix}. \tag{6}$$

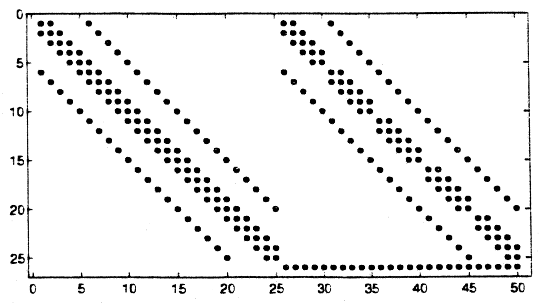

FIG. 2. *The sparsity pattern for the extended Jacobian.*

We use AD with graph-coloring techniques to compute the extended Jacobian

$$J_E = \begin{pmatrix} A_x y - \tilde{J} & A \\ 0 & [\nabla_y \tilde{f}]^T \end{pmatrix} \tag{7}$$

(both the blocks $A_x y - \tilde{J}$ and A are sparse). The blocks of the matrix J_E are then used in formula (5) to compute $\nabla_x f$.

3. *Separate blocks (of extended Jacobian).* In this approach, the block $A_x y - \tilde{J}$ is computed using sparse AD, i.e., AD is used with graph-coloring algorithms to differentiate $A(x)y - \tilde{F}(x)$ with respect to x. The gradient $\nabla_y \tilde{f}$ is computed using reverse mode of AD. Subsequently, formula (5) is used to form $\nabla_x f$.

For all three methods we need to solve the sparse system $Ay = \tilde{F}$, and for methods 2 and 3 we also need to solve $Aw = \nabla_y \tilde{f}$. For this we use SSPD,[1] a package written in C for solving large, sparse, symmetric, and positive definite systems.

The sparsity pattern of matrix A is one of the input parameters. The pattern used for the numerical results in this paper comes from discretization of a square grid with the natural ordering of the grid points. Figure 2 shows the resulting sparsity pattern of the extended Jacobian for a 5×5 grid. When computing J_E, we use coloring by rows, while the coloring is done by columns for $A_x y - \tilde{J}$ in the separate blocks approach. Note that the timing does not include the graph-coloring part, since if the gradient computation is a part of an iterative method, this has to be done only once.

Figure 3 shows timing results for the methods discussed above. The vertical axis is the time $\omega(\nabla f)$ it takes to compute the gradient divided by the time $\omega(f)$ to evaluate the function. According to theory, this ratio should approach a constant, and that is indeed what happens for methods 2 and 3.

However, for the straightforward approach the time ratio increases as the problem gets bigger and there is a jump around $n = 500$. The reason for the jump is that memory limitations have been reached and ADOLC is forced to store the computations on disk. But even in the absence of memory restrictions, this approach is slower than the other two. The explanation lies in the fact that in the straightforward approach we are taking the derivative of the process of solving the system $A(x)y = \tilde{F}(x)$, while in the other two methods this is outside the piece of code that is differentiated. Instead, $A(x)y - \tilde{F}(x)$ is differentiated, which is much simpler, especially since we provide an efficient code for the product $A(x)y$. This also explains why method 1 has much higher memory requirements.

[1]Written by Chunguang Sun, Advanced Computing Research Institute, Cornell University.

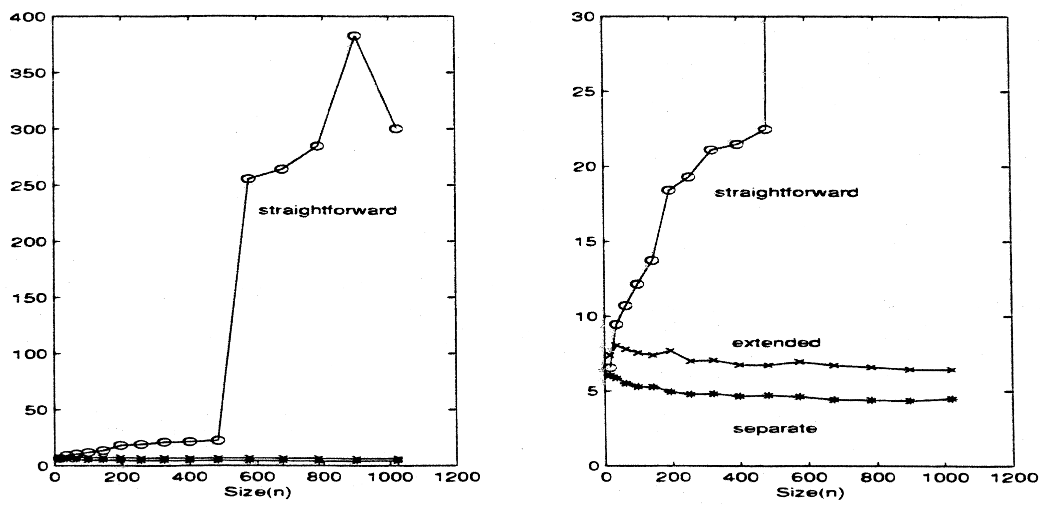

FIG. 3. *The time ratio $\omega(\nabla f)/\omega(f)$ for problem* 1. *The right plot is a blowup of the left plot.*

The separate blocks approach is slightly faster than the extended Jacobian approach. This is because reverse mode is somewhat more expensive than forward mode and because in the extended approach we do an unnecessary differentiation of $A(x)y - \hat{F}(x)$ with respect to y. (The result $A(x)$ has already been computed.)

3. Problem 2: Dynamical system. Consider the autonomous ODE

$$y' = F(y), \tag{8}$$

and suppose for an initial state $y(0) = x$ we use an explicit one-step method (e.g., Euler's method or Runge–Kutta) to compute an approximation y_k to a desired final state $y(T)$. Then we estimate the error $z = f_0(y_k - y(T))$, where f_0 is some appropriate scalar-valued function (e.g., the 2-norm). This leads to the class of functions described below.

The function $z = f(x)$ is defined as follows:

$$\begin{aligned} y_0 &= x, \\ y_i &= S(y_{i-1}) \qquad \text{for } i = 1, 2, \ldots, k, \\ z &= f_0(y_k), \end{aligned} \tag{9}$$

where $S : \mathbf{R}^n \to \mathbf{R}^n$ and $f_0 : \mathbf{R}^n \to \mathbf{R}$. (Here S represents the advancement of one step.) The gradient of $f(x)$ is given by

$$[\nabla f(x)]^T = [\nabla f_0(y_k)]^T J_{k-1} J_{k-2} \cdots J_1 J_0, \tag{10}$$

where J_i is the Jacobian of S at y_i.

We explore several different methods to compute the gradient.

1. *Straightforward use of reverse mode.* Reverse mode of AD is used on the program calculating the whole function.
2. *Dense block reverse mode.* The expression (10) is evaluated from left to right. First, $\nabla f_0(y_k)$ is computed using reverse mode, then it is used as a seed matrix to compute $[\nabla f_0(y_k)]^T J_{k-1}$, whose transpose is used as a seed to compute $[\nabla f_0(y_k)]^T J_{k-1} J_{k-2}$, and so on. Note that at each step the seed matrix has only one column.

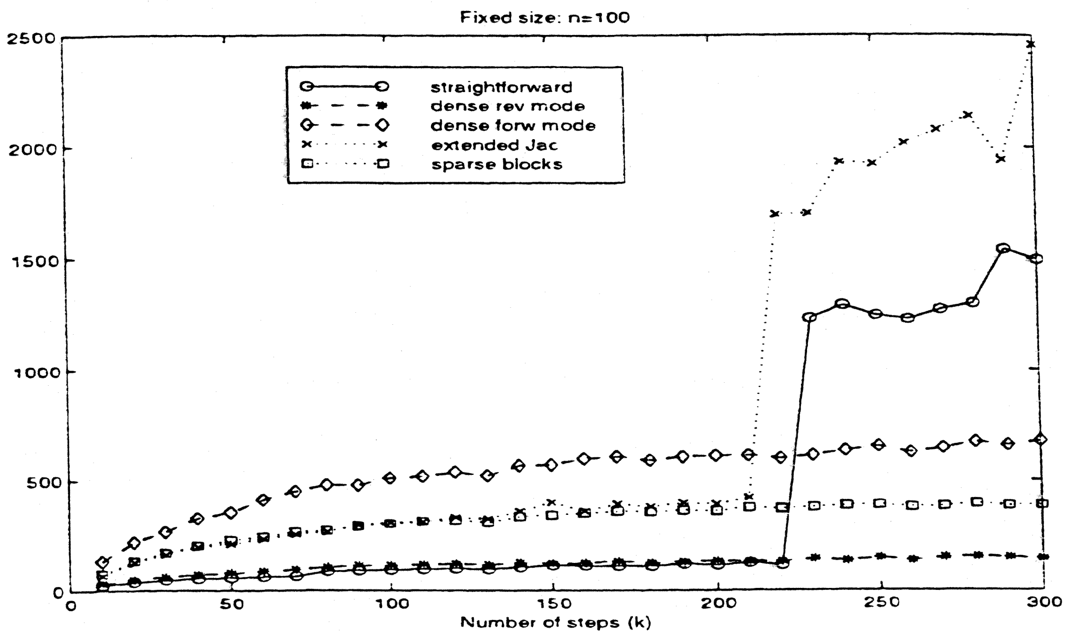

FIG. 4. *The time ratio* $\omega(\nabla f)/\omega(f)$ *for Euler's method.*

3. *Dense block forward mode.* Similar to method 2, but here we compute (10) from right to left. The identity is used as a seed matrix to compute J_0, which is used as a seed matrix to compute $J_1 J_0$, and so on.
4. *Extended Jacobian.* Sparse AD is used on the extended function

$$g_E : \begin{pmatrix} y_0 \\ \vdots \\ y_{k-1} \end{pmatrix} \mapsto \begin{pmatrix} S(y_0) \\ \vdots \\ S(y_{k-1}) \end{pmatrix}, \tag{11}$$

yielding

$$J_E = \begin{pmatrix} J_0 & & \\ & \ddots & \\ & & J_{k-1} \end{pmatrix}, \tag{12}$$

and reverse mode is used to compute $\nabla f_0(y_k)$. Subsequently, the gradient $\nabla f(x)$ is formed using formula (10).
5. *Sparse blocks.* Each block J_i is computed separately using sparse AD, and $\nabla f_0(y_k)$ is computed using reverse mode. Formula (10) is used to compute the gradient $\nabla f(x)$.

In our experiments each elemental Jacobian J_i is tridiagonal; the time-stepping function is Euler's method or fourth-order Runge–Kutta used on a linear constant coefficient ODE. For the two approaches using sparse AD, the coloring is done by columns. The multiplication in (10) is done from left to right rather than from right to left, since the former involves matrix-vector multiplications while the latter involves matrix-matrix multiplications. (Although both matrices are sparse to begin with, we get fill-in causing the latter approach to be costly).

Figures 4 and 5 show the timing for Euler's method and Runge–Kutta, respectively, where the length of the vector x is fixed and the number of steps varies. The

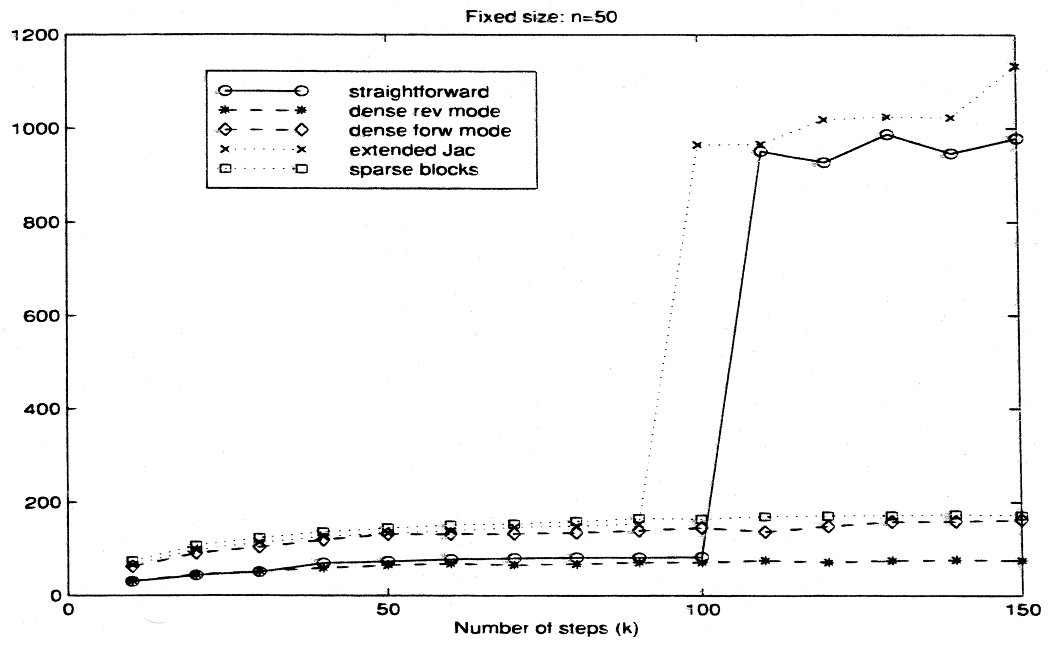

FIG. 5. *The time ratio* $\omega(\nabla f)/\omega(f)$ *for fourth-order Runge–Kutta.*

vertical axis is the time $\omega(\nabla f)$ it takes to compute the gradient divided by the time $\omega(f)$ it takes to evaluate the function. All the calculations are done through ADMIT and MATLAB; the timing does not include the graph-coloring step.

The flop count for the step function S used here is $O(n)$, so it takes $O(nk)$ time to evaluate the function. The time to compute the gradient is also $O(nk)$, except for method 3, where it is $O(n^2k)$. The memory requirement for method 3 is independent of k. For all the other methods, we first need to do a forward sweep to compute and store the y_i's before we do a reverse sweep to compute the derivatives. This requires $O(nk)$ memory.

The memory complexity can be reduced by using Griewank's checkpointing scheme [8] but at the cost of some increase in time complexity. This could be used separately or together with method 2 if there were not enough internal memory to store all the y_i's.

The straightforward approach performs well for small problems, but similar to problem 1, the time blows up when memory limitations are reached. This also happens for the extended Jacobian approach, although forward mode is used. As mentioned before ADOLC records, for both forward and reverse modes, all operations performed and indices used in addition to internal variables. The reason for the jump is that the space allocated for the integer tape fills up. Although the number of real variables stored is $O(nk)$ for all the methods (except method 3), the memory requirement for the other information is $O(nk)$ for methods 1 and 4 but only $O(n)$ for methods 2 and 5.

Ignoring these memory issues, methods 1 and 2 have similar running times, since both are doing essentially the same computation. Methods 4 and 5 are about two or three times slower than the first two methods. This is not surprising, since the seed matrices V used in the forward mode computations J_iV in methods 4 and 5 have three columns, whereas the seed matrices W in the reverse mode computations W^TJ_i in methods 1 and 2 have only one column.

For dense block forward mode the seed matrix has width n, so one would expect this method to always be slower than sparse blocks. On the other hand, the multiplication is implicit in the dense method, while in sparse blocks each J_i is formed and then multiplied by a vector.

4. Conclusions.

Large optimization problems are usually structured. In this paper we have illustrated how structure can be used to enable efficient gradient computation. Specifically, we have explored the use of automatic differentiation to compute the gradient in two important structured function classes.

In some cases, our structured approach improves the running time, but more importantly it has space advantages. In particular, since reverse mode requires that all intermediate results be stored during the gradient computation, it may use a huge amount of memory—this certainly shows up in our numerical experiments. Indeed, when applying reverse mode to the whole function, memory limitations are reached for moderately sized problems. However, memory requirements for the structured approach are relatively modest.

Acknowledgment. We thank Arun Verma for valuable suggestions and help using the ADMIT toolbox.

REFERENCES

[1] C. H. Bischof, A. Carle, P. M. Khademi, and A. Mauer, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Comput. Sci. Engrg., 3 (1996), pp. 18–32.

[2] C. H. Bischof, A. Bouaricha, P. M. Khademi, and J. J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, INFORMS J. Comput., 9 (1997), pp. 185–194.

[3] T. F. Coleman and A. Verma, *The efficient computation of sparse Jacobian matrices using automatic differentiation*, SIAM J. Sci. Comput, 19 (1998), pp. 1210–1233.

[4] T. F. Coleman and A. Verma, *Structure and efficient Jacobian calculation*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., SIAM, Philadelphia, PA, 1996, pp. 149–159.

[5] T. F. Coleman and A. Verma, *ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox*, Tech. Rep. CTC97TR271, Theory Center, Cornell University, Ithaca, NY, 1997.

[6] T. F. Coleman and A. Verma, *Structure and efficient Hessian calculation*, in Advances in Nonlinear Programming, Ya-xiung Yuan, ed., Kluwer Academic Publishers, Norwell, MA, 1998, pp. 57–72.

[7] A. Griewank, D. Juedes, and J. Utke, *ADOL-C: A package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, 22 (1996), pp. 131–167.

[8] A. Griewank, *Achieving logaritmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optim. Methods Softw., 1 (1992), pp. 35–54.