Computing Large-Sparse Jacobian Matrices Using Automatic Differentiation

Brett M. Averick
Jorge J. Moré
Christian H. Bischof
Alan Carle
Andreas Griewank

CRPC-TR93362
April 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

COMPUTING LARGE SPARSE JACOBIAN MATRICES USING AUTOMATIC DIFFERENTIATION

Brett M. Averick*, Jorge J. Moré, Christian H. Bischof, Alan Carle*, and Andreas Griewank

Mathematics and Computer Science Division

Preprint MCS-P348-0193

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

* Work supported in part by the Center for Research on Parallel Computation, under NSF Cooperative Agreement No. CCR-8809615.

ABSTRACT

The computation of large sparse Jacobian matrices is required in many important large-scale scientific problems. We consider three approaches to computing such matrices: hand-coding, difference approximations, and automatic differentiation using the ADIFOR (Automatic Differentiation in Fortran) tool. We compare the numerical reliability and computational efficiency of these approaches on applications from the MINPACK-2 test problem collection. Our conclusion is that automatic differentiation is the method of choice, leading to results that are as accurate as hand-coded derivatives, while at the same time outperforming difference approximations in both accuracy and speed.

COMPUTING LARGE SPARSE JACOBIAN MATRICES USING AUTOMATIC DIFFERENTIATION

Brett M. Averick*, Jorge J. Moré, Christian H. Bischof,
Alan Carle*, and Andreas Griewank

1 Introduction

The solution of large-scale nonlinear problems often requires the computation of the Jacobian matrix $f'(x)$ of a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$. This computation is required, for example, in constrained optimization, parameter identification, sensitivity analysis, and the solution of systems of stiff differential and algebraic equations. In this paper we consider three approaches to computing large, sparse Jacobian matrices.

The most popular approach is to use *function differences* (FD) to approximate the Jacobian matrix. The i-th column of $f'(x)$ can be approximated by first-order accurate *forward differences* and by second-order accurate *central differences*,

$$\frac{f(x+h_ie_i)-f(x)}{h_i}, \qquad \frac{f(x+h_ie_i)-f(x-h_ie_i)}{2h_i},$$

respectively, where h_i is a suitably chosen parameter and e_i is the i-th unit vector. Computing derivatives by differences has the advantage that only the function is needed as a black box; however, the accuracy of such derivative approximations is hard to assess. The choice of the difference parameter h_i can be a source of difficulty for many problems, in particular if the problem is highly nonlinear or if the function is noisy. A small step size h_i is needed to suitably approximate the derivatives yet may lead to numerical cancellation and the loss of accuracy.

The potential inaccuracies of difference approximations can be avoided if one is able to supply derivative code. One approach is to *hand-code* a subroutine to evaluate $f'(x)$. In addition to being accurate, hand-coding usually produces efficient code. However, this approach is often time-consuming and error-prone, especially if the function is complicated. In particular, new coding effort is required whenever the function is modified.

Another way to obtain Jacobian matrices is by using symbolic manipulation packages such as Maple, Reduce, Macsyma, or Mathematica. Given a string describing the definition of a function, symbolic manipulation packages provide algebraic expressions for derivatives expressed in terms of the independent variables. Symbolic differentiation is a powerful technique, but quickly runs into resource limitations when applied to even moderately sized problems (described by 100 lines of code, say). Hence we will not consider it further.

Still another way to obtain derivative code is through *automatic differentiation* (AD). Automatic differentiation techniques rely on the fact that every function, no matter how

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

*Work supported in part by the Center for Research on Parallel Computation, under NSF Cooperative Agreement No. CCR-8809615.

complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as the trigonometric and exponential functions. By applying the chain rule to the composition of those elementary operations, one can compute derivative information for f exactly and in a completely mechanical fashion. The perceived disadvantage of automatic differentiation techniques is that they are not able to handle large problems. In particular, there seems to be a perception that automatic differentiation techniques are not able to compute large sparse Jacobian matrices either accurately or efficiently. The main purpose of this paper is to dispel this perception.

In Sections 2 and 3, we review three approaches to computing sparse Jacobian matrices: hand-coding, difference approximations, and automatic differentitation using the ADIFOR (Automatic Differentiation of Fortran) tool. The rest of the paper uses several of the MINPACK-2 test problems to compare these approaches. The test problems are described in Section 4, and computational results are presented in Sections 5 and 6. We conclude that automatic differentiation is the method of choice, leading to results that are as accurate as hand-coded derivatives, while at the same time outperforming difference approximations in both accuracy and speed.

2 Computing Sparse Jacobian Matrices

Computing the Jacobian matrix $f'(x)$ of a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ can be a very difficult task when $f'(x)$ is large and sparse, particularly if a sparse data structure is used. The difficulty lies in the need to place elements of $f'(x)$ into the correct positions of the sparse data structure once they have been computed. In this section we discuss techniques that avoid this dependency on the data structure.

We assume that code for evaluating the Jacobian-vector product $f'(x)v$ for any $x, v \in \mathbb{R}^n$ is available; for many problems it is relatively easy to develop such code since there is no dependence on data structure. If $f'(x)v$ code is not available, the Jacobian-vector product can be approximated by

$$f'(x)v \approx \frac{f(x + h_v v) - f(x)}{h_v}$$

for some difference parameter h_v.

Each column of $f'(x)$ can be obtained by choosing v to be the corresponding Cartesian basis vector. This can be extremely inefficient as it requires the computation of n Jacobian-vector products or in the case of a difference approximation, n function evaluations. To avoid this inefficiency, we *partition* the columns of $f'(x)$ into groups such that columns in a group do not have nonzeros in the same row position. For example, if a function $f : \mathbb{R}^4 \to \mathbb{R}^4$ has a Jacobian matrix $f'(x)$ with the structure (symbols denote nonzeros, and zeros are not shown)

$$f'(x) = \begin{pmatrix} \bigcirc & & & \\ \bigcirc & & & \diamond \\ & \triangle & & \diamond \\ & \triangle & \square & \\ & \triangle & \square & \end{pmatrix},$$

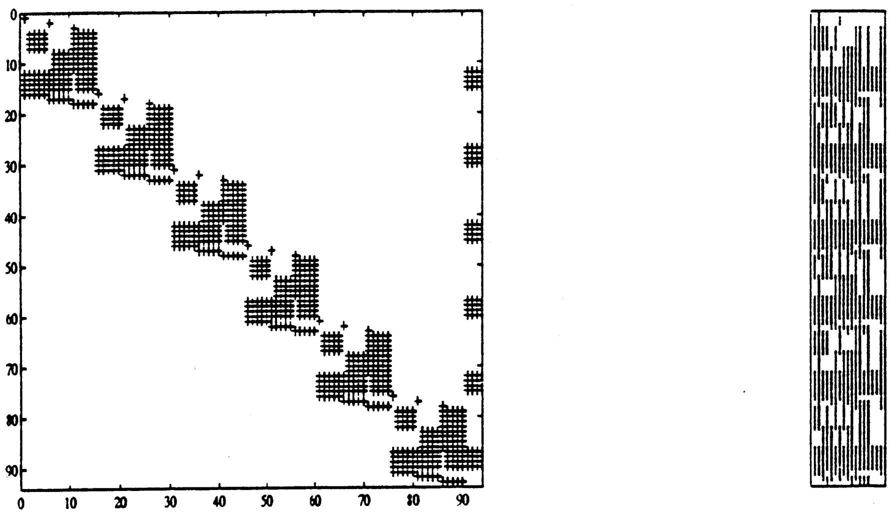

Figure 2.1: IER Jacobian Sparsity (left) and IER Compressed Sparsity (right)

then columns 1 and 2 can be placed in one group, while columns 3 and 4 can be placed in another group. This partitioning identifies *structurally orthogonal* columns of $f'(x)$, that is, columns whose inner product is zero, independent of x.

Given a partitioning of $f'(x)$ into p groups, each group consisting of structurally orthogonal columns, we can determine $f'(x)$ with p evaluations of $f'(x)v$. For each group we compute $f'(x)v$, where $v_i = 1$ if the i-th column is in the group, and $v_i = 0$ otherwise. In the above example, we would compute $f'(x)v_i$ for $v_1 = e_1 + e_2$ and $v_2 = e_3 + e_4$, and obtain

$$f'(x)v_1 = \begin{pmatrix} \bigcirc \\ \bigcirc \\ \triangle \\ \triangle \\ \triangle \end{pmatrix}, \qquad f'(x)v_2 = \begin{pmatrix} \\ \diamond \\ \diamond \\ \square \\ \square \end{pmatrix},$$

at the cost of only two evaluations of $f'(x)v$ (versus four for the naive approach). Because of the structural orthogonality property we can uniquely extract all entries of the Jacobian matrix from the Jacobian-vector products.

When Jacobian-vector products $f'(x)v_i$ are computed by hand-coded programs, it is usually advantageous to calculate them simultaneously in order to avoid the reevaluation of common expressions. This is the motivation for the *compressed Jacobian* approach where we assemble the vectors v_i into an $n \times p$ matrix V and compute $f'(x)V$. Clearly, the advantages of computing the compressed Jacobian tend to increase with p. Figure 2.1 illustrates the difference between the sparsity structure of $f'(x)$ and the sparsity structure of the compressed Jacobian $f'(x)V$ for the Inverse Elastic Rod problem from the MINPACK-2 collection (See Section 4).

Curtis, Powell, and Reid (CPR) [11] were the first to note that a partitioning of the columns into p structurally orthogonal groups allows the approximation of the Jacobian matrix by function differences with p function evaluations. In the CPR algorithm the groups are formed one at a time by scanning the columns in the natural order and including a column in the current group if it has not been included in a previous group and if it does not have a nonzero in the same row position as another column already in the group.

Coleman and Moré [10] showed that the partitioning problem could be analyzed as a *graph coloring* problem and that, by looking at the problem from the graph coloring point of view, it is possible to improve the CPR algorithm by scanning the columns in a carefully selected order. Coleman, Garbow, and Moré [9, 8] describes software for the partitioning problem. Given a representation of the sparsity structure of $f'(x)$, these algorithms produce a partitioning of the columns of $f'(x)$ into p structurally orthogonal groups. For many sparsity patterns, p is small and independent of n. For example, if the sparsity structure has bandwidth β, then $p \leq \beta$. We also note that discretization of an infinite-dimensional problem also leads to sparsity patterns where p is independent of the mesh size.

3 Automatic Differentiation and the ADIFOR Tool

Automatic differentiation [17] is a chain-rule-based technique for evaluating the derivatives of functions defined by computer programs with respect to their input variables. There are two basic modes of automatic differentiation, which are usually referred to as *forward* and *reverse*. As discussed in [12], the reverse mode is closely related to adjoint methods and has a very low operation count for gradients. However, its potentially large memory requirement has been a serious impediment to its application in large-scale scientific computing. When there are several independent and dependent variables, the operation count for evaluating the Jacobian matrix may be lowest for certain mixed strategies [15] rather than for the forward or reverse mode. AD can also be extended to the accurate evaluation of second and higher derivatives [13, 7, 3]. A comprehensive collection on the theory, implementation, and some earlier applications can be found in [14].

In contrast to the approximation of derivatives by function differences, AD incurs no truncation error. Hence, at least for noniterative and branch-free codes, the resulting derivative values are usually obtained with the working accuracy of the original function evaluation. In contrast to fully symbolic differentiation, both operations count and storage requirement can be *a priori* bounded in terms of the complexity of the original function code.

The ADIFOR (Automatic Differentiation in Fortran) tool [2, 5, 4] provides automatic differentiation of programs written in Fortran 77. Given a Fortran subroutine (or collection of subroutines) describing a function, and an indication of which variables in parameter lists or common blocks correspond to independent and dependent variables with respect to differentiation, ADIFOR produces Fortran 77 code that allows the computation of the derivatives of the dependent variables with respect to the independent variables. ADIFOR employs a hybrid of the forward and reverse modes of automatic differentiation. The resulting decrease in complexity compared with an implementation based entirely on the forward mode is usually substantial.

In contrast to some earlier AD implementations [16], the source translator ADIFOR was designed from the outset with large-scale codes in mind. The facilities of the ParaScope Fortran environment [6] control flow and data dependence flow information. ADIFOR produces portable Fortran 77 code and accepts almost all of Fortran 77—in particular, arbitrary calling sequences, nested subroutines, common blocks, and equivalences. The ADIFOR-generated code tries to preserve vectorization and parallelism in the original code. It also employs a consistent subroutine naming scheme that allows for code tuning, the use of domain-specific knowledge, and the exploitation of vendor-supplied libraries. It should be stressed that ADIFOR uses data flow analysis information to determine the set of variables that require derivative information in addition to the dependent and independent ones. This approach allows for an intuitive interface, and greatly reduces the storage requirements of the derivative code.

ADIFOR produces code to compute the Jacobian-vector product $f'(x)v$ or the compressed Jacobian matrix $f'(x)V$. The directional derivative $f'(x)v$ can be evaluated without forming the Jacobian matrix explicitly and thus potentially at a much lower computational cost. The operations count for this calculation is bounded by three times that of f; actual runtime ratios vary depending on the implementation, the computing platform and the nature of f. The compressed Jacobian matrices can be computed by exploiting the same graph coloring techniques discussed in Section 2. As already pointed out, the advantages of computing the compressed Jacobian matrix increase with p because it allows the reuse of expressions. ADIFOR-generated code reuses these expressions automatically; this is not possible with difference approximations, and may not be done in a hand-coded subroutine to evaluate $f'(x)V$.

4 Test Problems

In this section we describe the problems used in comparing the different approaches for computing a sparse Jacobian matrix. The descriptions are brief since the problems are part of the MINPACK-2 test problem collection; the current version of this collection is described by Averick, Carter, Moré, and Xue [1]. For each problem, the MINPACK-2 test collection contains subroutines that define the sparsity pattern of $f'(x)$ and evaluate $f(x)$ and $f'(x)V$ for any $V \in \mathbb{R}^{n \times p}$.

These problems are representative of computational problems found in applications. Collocation and finite differences are used to discretize these problems so as to obtain systems of nonlinear equations $f(x) = 0$, where $f : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear mapping with a sparse Jacobian matrix $f'(x)$.

Flow in a Channel (FIC). The analysis of fluid injection through one side of a long vertical channel leads to the boundary value problem

$$u'''' = R[u'u'' - uu'''], \qquad 0 \le t \le 1,$$

$$u(0) = u'(0) = 0, \quad u(1) = 1, \quad u'(1) = 0,$$

where u is the potential function, u' is the tangential velocity, and R is the Reynolds number of the fluid.

Discretization of this problem by a k-stage collocation method, with $k = 4$, leads to a system of $n = 8n_h$ equations, where n_h is the number of subintervals in the collocation scheme. In this problem there is a maximum of 9 nonzeros per row, independent of n_h.

Swirling Flow between Disks (SFD). The analysis of steady flow of a viscous, incompressible, axisymmetric fluid between two rotating, infinite coaxial disks, located at $t = 0$ and $t = 1$, yields the boundary value problem

$$\epsilon f'''' + ff''' + gg' = 0, \quad \epsilon g'' + fg' + f'g = 0, \qquad 0 \le t \le 1,$$

$$f(0) = f'(0) = f(1) = f'(1) = 0, \quad g(0) = \Omega_0, \quad g(1) = \Omega_1,$$

where f' is radial velocity, g is angular velocity (Ω_0 and Ω_1 are the angular velocities of the infinite disks), and $0 \le \epsilon \ll 1$ is a viscosity parameter.

Discretization of this problem by a k-stage collocation method, with $k = 4$, leads to a system of $n = 14n_h$ equations, where n_h is the number of subintervals in the collocation scheme. In this problem there is a maximum of 14 nonzeros per row, independent of n_h.

Incompressible Elastic Rods (IER). The shape of a thin incompressible elastic rod, or elastica, clamped at the origin and acted on by a vertical force Q, a horizontal force P, and torque M is described by the solution of the boundary value problem

$$\theta'(s) = Qx(s) - Py(s) + M, \quad x'(s) = \cos[\theta(s)], \quad y'(s) = \sin[\theta(s)],$$

$$x(0) = y(0) = \theta(0) = 0,$$

where θ is the local angle of inclination, and s is the arc length along the elastica. We need to determine Q, P, and M such that $x(\cdot), y(\cdot)$, and $\theta(\cdot)$ solve this boundary value problem and satisfy the boundary conditions $x(1) = a$, $y(1) = b$, $\theta(1) = c$.

Discretization by a k-stage collocation method, with $k = 4$, leads to a system of $n = 15n_h + 3$ equations, where n_h is the number of subintervals in the collocation scheme. The sparsity structure of the Jacobian and compressed Jacobian matrix for $n_h = 6$ are shown in Figure 2.1. The nonzeros in the last 3 columns correspond to the variables Q, P, and M. In this problem there is a maximum of 17 nonzeros per row, independent of n_h. Since there are 17 columns in the compressed Jacobian matrix, the coloring is optimal.

Solid Fuel Ignition (SFI). This problem arises in the analysis of a thermal reaction process dependent upon a balance between chemically generated heat addition and heat transfer by conduction, in a rigid material. A steady-state model of solid fuel ignition can be described in terms of the solution u_λ of the boundary value problem

$$-\Delta u(x) = \lambda \exp[u(x)], \quad x \in \Omega, \qquad u(x) = 0, \quad x \in \partial\Omega,$$

where Δ is the Laplace operator, Ω is a domain in $\mathbb{R}^2$ with boundary $\partial\Omega$, and $\lambda \in \mathbb{R}$.

Discretization of this problem by finite differences on the unit square leads to a system of $n = n_x n_y$ equations, where n_x and n_y are the number of interior gridpoints in the coordinate directions, respectively. For this problem there is a maximum of 5 nonzeros per row, independent of n_x and n_y.

Flow in a Driven Cavity (FDC). The steady flow of a viscous incompressible fluid in a planar region Ω is described in terms of a stream function by the boundary value problem

$$\Delta^2 \psi - R\Big[(\partial_y \psi)(\partial_x \Delta\psi) - (\partial_x \psi)(\partial_y \Delta\psi)\Big] = 0,$$

$$\psi(\xi_1, \xi_2) = \partial_x \psi(\xi_1, \xi_2) = 0, \qquad \partial_y \psi(\xi_1, \xi_2) = \begin{cases} 1 & \text{if } \xi_2 = 1 \\ 0 & \text{if } 0 \le \xi_2 < 1. \end{cases}$$

Table 5.1: Accuracy of ADIFOR and Function Differences

		Absolute Error		Relative Error	
Problem	N	FD	ADIFOR	FD	ADIFOR
FIC	96	2.9×10^{-6}	1.4×10^{-14}	5.6×10^{-1}	9.5×10^{-16}
SFD	98	2.5×10^{-8}	0.0	1.0	0.0
IER	93	4.4×10^{-8}	0.0	1.6×10^{-1}	0.0
SFI	100	9.7×10^{-8}	0.0	3.8×10^{-8}	0.0
FDC	100	3.2×10^{-6}	3.5×10^{-14}	5.6×10^{-6}	5.4×10^{-14}

Discretization by finite differences on the unit square leads to a system of $n = n_x n_y$ equations, where n_x and n_y are the number of interior gridpoints in the coordinate directions, respectively. The Jacobian matrix has a maximum of 13 nonzeros per row independent of n_x and n_y.

5 Accuracy

We first compare the accuracy of the Jacobian matrix produced by ADIFOR, with the accuracy of the function difference approximation. As the standard we take the Jacobian matrix included with the MINPACK-2 test problem collection. We measure both the absolute error and the relative error,

$$\max_{ij} \{|\partial_{i,j} f(x) - a_{ij}(x)|\}, \qquad \max_{ij} \left\{ \frac{|\partial_{i,j} f(x) - a_{ij}(x)|}{\max\{|\partial_{i,j} f(x)|, |a_{ij}(x)|\}} \right\},$$

respectively, where $\partial_{i,j} f(x)$ is the derivative produced by the MINPACK-2 software, and $a_{i,j}(x)$ is either the difference approximation or the ADIFOR Jacobian matrix.

The results presented in Table 5.1 were obtained on a Solbourne 5E/902 in double-precision IEEE arithmetic. For these results we evaluated the Jacobian matrices at a random vector with elements in the interval [0,1] and used a value of $h_i = 10^{-8}$ for all the variables. We do not claim that this choice is optimal, but for these problems this choice produces reasonably accurate results. In general, the accuracy of an approximation with differences of function values depends on the choice of the difference parameters h_i, but even with an optimal choice of difference parameter, we can expect a difference approximation that is accurate only up to half the number of possible significant digits.

The results in Table 5.1 clearly indicate the superior accuracy of automatic differentiation as compared with function differences. In terms of absolute error, ADIFOR and hand-coded Jacobians agree up to 16 significant digits, while function differences offer at most half that accuracy.

The same type of observation can be made for the relative error. However, relative error can be misleading if the Jacobian elements are sufficiently small. For FIC and IER, the relative error is usually of order 10^{-7}, but for small Jacobian entries, it may be of order 10^{-1}. In the case of SFD, function differences show a relative error of 1. This is due to

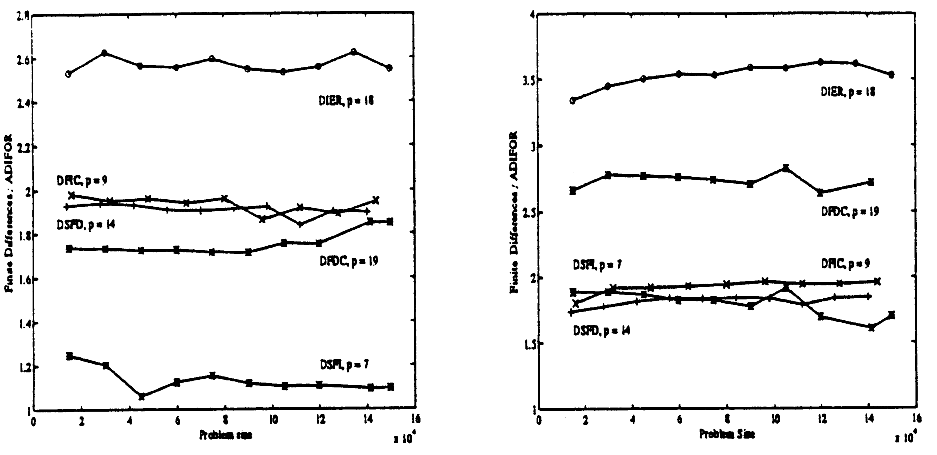

Figure 6.1: FD vs. ADIFOR Timings on the Solbourne (left) and the IBM (right)

entries where the derivative value is of order 10^{-13} but function differences yield zero. Note, however, that ADIFOR accurately computes these small elements.

6 Timing

We now compare the time required to compute a sparse Jacobian matrix by all three approaches. The timing information was obtained for double-precision computations on a Solbourne 5E/902 and an IBM RS6000/580 workstation. Since we are interested in large problems, we used problems with n variables where n ranges between 14,000 and 160,000. Similar results were obtained on smaller problems.

Figure 6.1 shows the ratio of the run times for difference approximations to the Jacobian matrix to ADIFOR-generated derivative code. These results clearly show that the ADIFOR-generated code is faster than the difference approximations in all cases and that the performance advantage is more pronounced on the IBM than on the Solbourne. This can be explained by noting that the Solbourne has a true scalar processor, whereas the IBM employs a pipelined superscalar chip that performs well on the vector operations that constitute the bulk of the ADIFOR-generated derivative code.

In general we expect ADIFOR to perform best if the number of groups p is large because the advantages of computing the compressed Jacobian matrix increase with p. This is borne out by our results. The performance of ADIFOR for small p tends to depend on the particular problem.

Note that the runtime ratios in Figure 6.1 are independent of n. This can be explained by noting that the runtime is proportional to the number of groups p associated with the sparsity structure; for these problems p does not depend on the size n of the Jacobian matrix.

The timing comparison of the ADIFOR-generated code with the hand-coded Jacobian matrix appears in Figure 6.2. We see that the ADIFOR-generated Jacobian code performs somewhat worse than the hand-coded Jacobian matrix, but not by a margin of more than

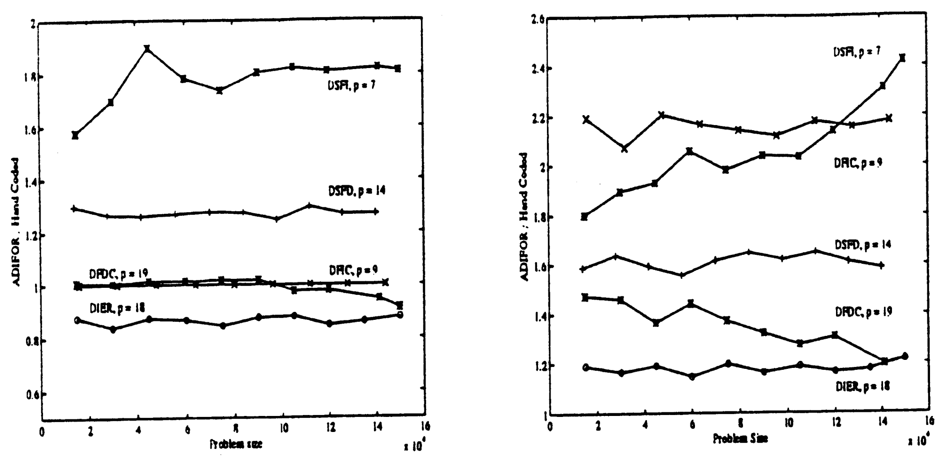

Figure 6.2: ADIFOR vs. Hand-Code Timings on the Solbourne (left) and the IBM (right)

roughly 2.5.

Lastly, Table 6.1 puts these these timings in perspective by comparing the time for computing the Jacobian matrices (FD, ADIFOR, and MINPACK) with the time required to partition the columns into structurally orthogonal groups (DSM), and the time required to convert the compressed Jacobian matrix into a sparse matrix format (FDJS). Subroutines for these tasks are described by Coleman, Garbow, and Moré [9, 8]. Subroutine **dsm** takes the sparsity pattern of the Jacobian matrix and produces a partitioning of the columns of $f'(x)$ into p structurally orthogonal groups, while subroutine `fdjs` converts the compressed Jacobian matrix into a sparse matrix format. Columns N, NNZ, and P show the dimension, number of nonzeros in the Jacobian, and number of groups, respectively, for each of the problems.

Table 6.1: Detailed Solbourne Timings.

				Time (Seconds)				
Problem	N	NNZ	P	DSM	FD	ADIFOR	MINPACK	FDJS
SFI	14884	73932	7	2.38	1.35	0.93	0.59	.19
FIC	16000	123987	9	2.62	5.23	2.55	1.46	.35
SFD	14000	154981	14	3.79	5.48	2.69	2.08	.43
IER	15003	158000	18	6.14	7.43	2.77	3.16	.48
FDC	14884	191056	19	9.11	11.71	6.53	6.50	.54

Table 6.1 shows that the time required by **dsm** is significant when compared with the time required to compute a Jacobian matrix by either ADIFOR or the hand-coded MINPACK subroutines. This is justified for most problems because **dsm** needs to be called only once for each sparsity pattern. Moreover, the runtime of **dsm** only depends on the sparsity pattern,

not on the expense of evaluating the function or Jacobian matrix.

We also note that the runtime of `fdjs` is small compared with the other times in Table 6.1 and is proportional to the number of nonzeros in the Jacobian. This was to be expected because `fdjs` converts the compressed Jacobian matrix into a column-oriented sparse matrix format but does not perform any arithmetic operations. We also note that the time required by `fdjs` becomes less significant as the number of groups p increases.

7 Conclusions

We conclude that ADIFOR-derived derivatives are certainly superior to difference approximations. Not only does the automatic differentiation approach not suffer from truncation error, but its ADIFOR incarnation delivers code that outperforms divided differences by a factor of up to 3.5. The other attraction of ADIFOR is that one can generate derivative code at the touch of the button, whereas the development of a derivative code by hand is tedious and time-consuming. Whether the effort involved in computing a Jacobian by hand is worth the modest speedup that we have observed here obviously depends on the application, but from our perspective it is not.

Acknowledgements

We would like to thank Richard Carter for many stimulating discussions and Jack Dongarra for the use of the IBM RS-6000.

References

[1] B. M. Averick, R. G. Carter, J. J. Moré, and G.-L. Xue, *The MINPACK-2 test problem collection*, Preprint MCS-P153-0692, Argonne National Laboratory, Argonne, Illinois, 1992.

[2] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 1–29.

[3] C. Bischof, G. Corliss, and A. Griewank, *Computing second- and higher-order derivatives through univariate Taylor series*, Preprint MCS-P296-0392, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[4] C. Bischof and A. Griewank, *ADIFOR: A Fortran system for portable automatic differentiation*, in Proceedings of the 4th Symposium on Multidisciplinary Analysis and Optimization, AIAA Paper 92-4744, American Institute of Aeronautics and Astronautics, 1992, pp. 433–441.

[5] C. Bischof and P. Hovland, *Using ADIFOR to compute dense and sparse Jacobians*, Tech. Report MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[6] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren, *A practical environment for scientific programming*, IEEE Computer, 20 (1987), pp. 75–89.

[7] B. Christianson, *Reverse accumulation and accurate rounding error estimates for Taylor series coefficients*, Optimization Methods and Software, 1 (1992), pp. 81–94.

[8] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Fortran subroutines for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 346–347.

[9] ——, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[10] T. F. Coleman and J. J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209.

[11] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.

[12] A. Griewank, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, 1989, pp. 83 - 108.

[13] ——, *Automatic evaluation of first- and higher-derivative vectors*, in Proceedings of the Conference at Würzburg, Aug. 1990, Bifurcation and Chaos: Analysis, Algorithms, Applications, R. Seydel, F. W. Schneider, T. Küpper, and H. Troger, eds., vol. 97, Birkhäuser Verlag, Basel, Switzerland, 1991, pp. 135 - 148.

[14] A. Griewank and G. F. Corliss, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[15] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss, eds., SIAM, Philadelphia, 1991, pp. 126–135.

[16] D. Juedes, *A taxonomy of automatic differentiation tools*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss, eds., SIAM, Philadelphia, 1991, pp. 315–329.

[17] L. B. Rall, *Automatic Differentiation: Techniques and Applications*, vol. 120 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.