ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

COMPUTING GRADIENTS IN LARGE-SCALE OPTIMIZATION USING AUTOMATIC DIFFERENTIATION

Christian H. Bischof, Ali Bouaricha, Peyvand M. Khademi, Jorge J. Moré

Mathematics and Computer Science Division

Preprint MCS-P488-0195

January 1995

(Final Revised Version)

June 1996

Work supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

COMPUTING GRADIENTS IN LARGE-SCALE OPTIMIZATION USING AUTOMATIC DIFFERENTIATION*

CHRISTIAN H. BISCHOF ALI BOUARICHA PEYVAND M. KHADEMI
JORGE J. MORÉ

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{bischof,bouarich,khademi,more}@mcs.anl.gov

Abstract

The accurate and efficient computation of gradients for partially separable functions is central to the solution of large-scale optimization problems, since these functions are ubiquitous in large-scale problems. We describe two approaches for computing gradients of partially separable functions via automatic differentiation. In our experiments we employ the ADIFOR (Automatic Differentiation of Fortran) tool and the SparsLinC (Sparse Linear Combination) library. We use applications from the MINPACK-2 test problem collection to compare the numerical reliability and computational efficiency of these approaches with hand-coded derivatives and approximations based on differences of function values. Our conclusion is that automatic differentiation is the method of choice, providing code for the efficient computation of the gradient without the need for tedious hand-coding.

The solution of nonlinear optimization problems often requires the computation of the gradient ∇f_0 of a mapping $f_0 : \mathbb{R}^n \to \mathbb{R}$. If the number of variables n is moderate, we can approximate the components of the gradient by differences of function values, for example,

$$[\nabla f_0(x)]_i \approx \frac{f_0(x + h_i e_i) - f_0(x)}{h_i}, \qquad 1 \le i \le n, \tag{1}$$

where h_i is the difference parameter, and e_i is the i-th unit vector. However, for large-scale problems (even for moderately sized problems with $n = 100$ variables) use of this approximation is prohibitive because it requires n function evaluations for each gradient. Another reason to avoid the use of (1) is that truncation errors in this calculation can mislead an optimization algorithm and cause premature termination far away from a solution. Thus, algorithms for the solution of optimization problems avoid approximations of the gradient by differences, and insist on an accurate and efficient evaluation of the gradient.

In this paper we explore the use of automatic differentiation tools for the computation of ∇f_0 when $f_0 : \mathbb{R}^n \mapsto \mathbb{R}$ is partially separable, that is, f_0 can be represented in the form

$$f_0(x) = \sum_{i=1}^{m} f_i(x), \tag{2}$$

where f_i depends on $p_i \ll n$ variables. This class of functions, introduced by Griewank and Toint [17, 18], plays a fundamental role in the solution of large-scale optimization problems

since, as shown by Griewank and Toint, a function f_0 is partially separable if the Hessian matrix $\nabla^2 f_0(x)$ is sparse.

Algorithms and software that take advantage of the partially separable structure have been developed for various problems (see, for example, [14, 27, 19, 31, 32, 33, 34]). In these algorithms the partially separable structure is used mainly to approximate the (dense) Hessian matrices $\nabla^2 f_i(x)$ by quasi-Newton methods. Partial separability is also used to compute the gradient of f_0 as the sum of the gradients of the element functions f_i, but this is just another method for hand-coding the gradient. In a related paper [11] we discuss the impact of partial separability on optimization software.

The key observation needed to compute the gradient of a partially separable function is that if $f_0 : \mathbb{R}^n \mapsto \mathbb{R}$ is defined by (2), and if the vector-valued function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ is defined by

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}, \tag{3}$$

then the gradient of f_0 is given by

$$\nabla f_0(x) = f'(x)^T e, \tag{4}$$

where $f'(x)$ is the Jacobian matrix of f at x, and $e \in \mathbb{R}^m$ is the vector of all ones. At first sight this approach does not look promising because it requires the computation of the Jacobian matrix $f'(x)$. However, for partially separable functions, f_i depends on $p_i \ll n$ variables, and thus $f'(x)$ is a sparse matrix. We use the sparsity of $f'(x)$ to show that automatic differentiation tools can compute the gradient ∇f_0 so that

$$T\{\nabla f_0(x)\} \leq \Omega_T \, T\{f_0(x)\}, \tag{5}$$

$$M\{\nabla f_0(x)\} \leq \Omega_M \, M\{f_0(x)\}, \tag{6}$$

where $T\{\cdot\}$ and $M\{\cdot\}$ denote computing time and memory, respectively, and Ω_T and Ω_M are constant. We also show that for partially separable functions that arise in applications, the constants Ω_T and Ω_M are independent of n, in contrast to the use of (1).

The approach for computing the gradient of f_0 using (3) and (4) was proposed by Andreas Griewank and can be viewed as a special case of the results discussed by Griewank [21, Section 2]. Preliminary tests of this approach were done by Bischof and El-Khadiri [9]. The results in this paper show that this approach is not only feasible, but highly efficient.

A brief review of automatic differentiation, the ADIFOR (Automatic Differentiation of Fortran) tool [4, 6], and the SparsLinC (Sparse Linear Combination) library [5, 6] is provided in the next section. Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a potentially

long sequence of elementary operations such as additions, multiplications, and elementary functions (e.g., the trigonometric and exponential functions). By applying the chain rule to the composition of those elementary operations, derivative information can be computed exactly and in a completely mechanical fashion [22, 28].

In Section 2 we propose two approaches for the computation of the Jacobian matrix $f'(x)$. The first approach uses the sparsity pattern of $f'(x)$, graph-coloring techniques, and the ADIFOR tool to obtain a compressed Jacobian matrix that contains all the information needed to determine the entire Jacobian matrix. The second approach uses ADIFOR with the SparsLinC library to produce a sparse representation of the Jacobian matrix without a priori knowledge of the sparsity pattern. In fact, the sparsity pattern is a byproduct of the ADIFOR/SparsLinC approach.

Section 3 discusses the formulation of large-scale problems in terms of partially separable functions, and outlines the problems from the MINPACK-2 [1] collection of large-scale problems that we use to validate our approach. Experimental results with problems from the MINPACK-2 collection on Sun SPARC 10, IBM RS 6000 (model 370), and Cray C90 platforms are presented in Section 4.

Our results show that the compressed Jacobian approach with the ADIFOR automatic differentiation tool generally outperforms difference approximations (to the compressed Jacobian matrix) in terms of computing time. We make no comparisons with the standard difference approximation (1) because our results show that the approach based on (1) is roughly n times slower than the approach based on the compressed Jacobian matrix.

The ADIFOR/SparsLinC approach obviates the need for the computation of the sparsity pattern and the compressed Jacobian matrix, but produces slower gradient code in our test problems. This tradeoff between convenience and cost is not always an option. Using ADIFOR/SparsLinC is the only feasible approach for applications where it is desirable to relieve the user of the error-prone task of providing the sparsity pattern or where the assumption that the sparsity pattern of $f'(x)$ is independent of x does not hold.

For both approaches based on automatic differentiation, (5) and (6) hold with constants Ω_T and Ω_M that depend on the number of columns p in the compressed Jacobian matrix. For many sparsity patterns, p is independent of n. For example, Coleman and Moré [13] show that $p \leq \beta$ if there is a permutation of the Jacobian matrix with bandwidth β.

The relationship between p and Ω_M is mainly dependent on the problem. On scalar and vector architectures, the memory requirements for both approaches based on automatic differentiation is comparable with the approach based on difference approximations, with Ω_M somewhere between p and $12p$.

The relationship between p and Ω_T depends on the approach used to compute the gradient, the architecture, and the problem. On scalar architectures, the ADIFOR tool outperforms the approach based on difference approximations with $\Omega_T \leq 2p$; with the

ADIFOR/SparsLinC approach $\Omega_T \leq 30p$. The performance of the various approaches on vector architectures is harder to predict as performance depends on the level of vectorization in the approach for computing the gradient and the code that evaluates the problem. On vector architectures, the performance of the ADIFOR tool is comparable with the approach based on difference approximations, but the higher overhead in the indirect addressing used by the ADIFOR/SparsLinC approach leads to a significant degradation in performance.

The performance of the various approaches also depends on the sparsity pattern, the value of p, and the cost of evaluating the function. In this paper we concentrated on sparsity patterns where $p = 3$ and $p = 8$, but a similar study [1] of ADIFOR for computing sparse Jacobian matrices shows that we can expect $\Omega_T \leq 2p$ on scalar architectures for a wide variety of sparsity patterns with $7 \leq p \leq 19$. In general we expect our results to hold for any sparsity pattern where p is independent of n.

The cost of evaluating the function influences the performance of the various approaches because if the function is not costly to evaluate, then the overhead associated with the various approaches is noticeable. This can be seen in our results since the worst performance of the approaches based on automatic differentiation is obtained on quadratic functions (see problems EPT and PJB in Section 4.3). In particular, $\Omega_T \leq 10p$ with the ADIFOR/SparsLinC approach if we omit the two quadratic functions.

In terms of accuracy, both approaches based on automatic differentiation provide the gradient to full accuracy, while approximations based on differences always suffer from truncation errors and provide, at best, half the accuracy in the function evaluation. We emphasize that the accuracy of the gradient in an optimization algorithm is of paramount importance because the gradient is used to determine the search directions. An inaccurate gradient can easily lead to false convergence.

1 The ADIFOR Tool and the SparsLinC Library

Automatic differentiation [22, 28] (AD) is a chain-rule-based technique for evaluating the derivatives of functions defined by computer programs. AD produces code that, in the absence of floating-point exceptions, computes the values of the analytical derivatives accurate to machine precision. AD avoids the truncation and cancellation errors inherent in approximations of derivatives by differences of function values. Moreover, it is applicable to codes of arbitrary length containing branches, loops, and subroutine calls.

The *forward* and *reverse* modes of automatic differentiation for computing the Jacobian matrix of a mapping $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ are distinguished by how the chain rule is used to propagate derivatives through the computation. The forward mode accumulates the derivatives of intermediate variables with respect to the *independent* variables x, whereas the reverse mode propagates the derivatives of the *dependent* variables $y = f(x)$ with respect to intermediate variables.

Given a *seed* matrix S with n rows and p columns, the forward mode generates code for the computation of the directional derivative

$$f'(x)S. \tag{7}$$

Given the directional derivative $f'(x)S$ we can determine $f'(x)$ for suitable choices of the seed matrix S. Clearly, if we set S to the identity matrix of order n, then the Jacobian matrix can be obtained directly from (7), but this requires $p = n$. We want to have $p < n$ because this reduces the computing time of $f'(x)S$, which is usually possible if $f'(x)$ is sparse. For example, if the Jacobian matrix has the structure

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

where symbols denote nonzeros, and zeros are not shown, then we can determine $f'(x)$ from (7) by setting the seed matrix to

$$S = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

In this case $p = 2$. In general, each column of the seed matrix defines a group of *structurally orthogonal* columns, that is, columns that do not have a nonzero in the same row position. The partitioning of the columns of $f'(x)$ into groups of structurally orthogonal columns can be done with the techniques in Section 2. Given $f'(x)$, we compute the gradient of a partially separable function via (4),

The complexity of the forward mode is rather predictable. If $L\{f\}$ and $M\{f\}$ are, respectively, the number of floating-point operations and the amount of memory required by the computation of $f(x)$, then an AD-generated code employing the forward mode requires

$$L\{f'(x)S\} \leq (2 + 3p)\, L\{f\}, \qquad M\{f'(x)S\} \leq (1 + p)\, M\{f\}. \tag{8}$$

floating-point operations and memory, respectively, to compute $f'(x)S$ (see Griewank [21]). With the reverse mode, on the other hand, we can compute $f'(x)^T Q$ where, Q is a seed matrix with m rows and q columns. The reverse mode requires the ability to reverse the partial order of program execution and to remember (or recompute) any intermediate result that nonlinearly affects the final result. As a result, the complexity of the reverse mode is harder to predict. If no intermediate values are recomputed, a straightforward

implementation of the reverse mode requires $\mathcal{O}(L\{f\})$ floating-point operations and up to $\mathcal{O}(L\{f\} + M\{f\})$ memory, depending on the nonlinearity of the code.

The bounds in (8) are only theoretical and do not take into account the overhead in the generation of $f'(x)S$. As we shall see in Section 4, the overhead in the various approaches to automatic differentiation is of crucial importance.

The reverse mode is attractive when m is small. In particular, if $m = 1$, then $f'(x)$ is a gradient, and the reverse mode needs only $\mathcal{O}(L\{f\})$ operations to compute $f'(x)$. The storage requirement of the reverse mode, however, can be a difficulty because of the possible dependence on $L\{f\} + M\{f\}$. Griewank [20] suggested a snapshot approach to circumvent this difficulty.

There have been various implementations of automatic differentiation; an extensive survey can be found in [25]. In particular, we mention GRESS [24], and PADRE-2 [26] for Fortran programs and ADOL-C [23] for C programs. GRESS, PADRE-2, and ADOL-C implement both the forward and reverse modes. In order to save control flow information and intermediate values, these tools generate a trace of the computation by recording the particulars of every operation performed in the code. The interpretation overhead associated with using this trace for the purposes of automatic differentiation, as well as its potentially very large size, can be a serious computational bottleneck [30]. Recently, a source transformation approach to automatic differentiation has been explored in the ADIFOR [4, 6], ADIC [10], AMC [16], and Odyssee [29] tools. ADIFOR transforms Fortran 77 code, ADIC transforms ANSI-C code, and AMC and Odyssee transform a subset of Fortran 77. ADIFOR and ADIC mainly use the forward mode, with the reverse mode at the statement level, while AMC and Odyssee use the reverse mode.

In our work, we employed the ADIFOR tool, which has been developed jointly by Argonne National Laboratory and Rice University (see the World Wide Web sites `http://www.mcs.anl.gov/adifor` or `http://www.cs.rice.edu/~adifor` for additional information on ADIFOR). 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.

The workhorse of any mainly forward-mode first-order automatic differentiation approach, such as employed in ADIFOR or ADIC, for computing the m directional derivatives in (7) is the vector linear combination

$$\sum_{i=1}^{k} \alpha_i v_i, \tag{9}$$

where α_i is a scalar, v_i is a vector of length p, and k is usually less than 10. By default, this operation is implemented as a `DO` loop; and as long as p is of moderate size and the vectors

are dense, this is an efficient way of expressing a vector linear combination.

The SparsLinC library [5, 6] addresses the situation where the seed matrix S is sparse and most of the vectors involved in the computation of $f'(x)S$ are sparse. This situation arises, for example, in the computation of large sparse Jacobian matrices, since the sparsity of the final Jacobian matrix implies that, with great probability, all intermediate derivative computations involve sparse vectors as well. SparsLinC implements routines for executing the vector linear combination (9) using sparse data structures [6]. It is fully integrated into ADIFOR and ADIC and provides a mechanism for transparently exploiting sparsity in derivative computations. SparsLinC does not require knowledge of the sparsity structure of the Jacobian matrix; indeed, the sparsity structure of the Jacobian matrix is a byproduct of the derivative computation. The SparsLinC routines adapt to the particular situation at hand, providing efficient support for a wide variety of sparsity scenarios.

2 Computing Gradients of Partially Separable Functions

We compute the gradient of a partially separable function as outlined in thee introduction: Given the element functions $f_1, \ldots, f_m$ that define the partially separable function (2), we compute the Jacobian matrix $f'(x)$ of the vector-valued function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ defined by (3). The gradient $\nabla f_0(x)$ of the partially separable function is then obtained via (4); that is, we add the rows of $f'(x)$. In this section we propose two techniques for computing the Jacobian matrix.

If the sparsity pattern of $f'(x)$ is known, then graph-coloring techniques can be used to determine a seed matrix S so that the *compressed Jacobian matrix* $f'(x)S$ contains all the information needed to determine the entire Jacobian matrix $f'(x)$. The compressed Jacobian matrix approach has long been used in connection with the determination of sparse Jacobian matrices by differences of function values; see, for example, [13, 15]. As we mentioned in Section 1, the compressed Jacobian matrix approach requires the determination of a partitioning of the columns of $f'(x)$ into structurally orthogonal columns. Because of the structural orthogonality property, we can uniquely extract all entries of the original Jacobian matrix from the compressed Jacobian.

The partitioning problem can be considered as a graph-coloring problem [13]. Given a graph representation of the sparsity structure of $f'(x)$, these algorithms produce a partitioning of the columns of $f'(x)$ into p structurally orthogonal groups by graph-coloring algorithms for the column-intersection graph associated with $f'(x)$. For many sparsity patterns, p is small and independent of n. For example, if a matrix is banded with bandwidth β or if it can be permuted to a matrix with bandwidth β, it can be shown [13] that $p \leq \beta$. In our experiments we employ the graph-coloring software described in [12] to determine an appropriate partition.

In an optimization algorithm we invariably need to compute a sequence $\{\nabla f_0(x_k)\}$ of

gradients for some sequence $\{x_k\}$ of iterates. This step requires the computation of a sequence of Jacobian matrices $\{f'(x_k)\}$. In most cases we need to do the graph-coloring only once, since we can specify the *closure* of the sparsity pattern, that is, a sparsity pattern that, for every iterate x_k, contains the sparsity pattern of $\{f'(x_k)\}$. If we are not able to specify the closure of the sparsity pattern, the compressed Jacobian approach requires a call to the graph-coloring software at each iteration.

By exploiting the capability to compute directional derivatives (7), we can easily compute compressed Jacobian matrices via automatic differentiation (for additional details, see [3]): Given the seed matrix S, ADIFOR-generated code computes the compressed Jacobian matrix $f'(x)S$. In contrast to the approximation techniques based on the compressed Jacobian matrix approach [13, 15], all columns of the compressed Jacobian matrix are computed at once.

In many situations it is desirable to have a tool for the determination of $f'(x)$ that does not require knowledge of the sparsity pattern of $f'(x)$. This situation arises, for example, while developing interfaces for the solution of large-scale optimization problems [11], where it is desirable to relieve the user of the error-prone task of providing the sparsity pattern. In these situations, a sparse implementation of automatic differentiation, such as provided by the ADIFOR/SparsLinC approach, is the only feasible approach.

We use the term *sparse* ADIFOR for the approach based on the ADIFOR tool employing the SparsLinC library for the computation of vector linear combinations of derivative objects. This approach is extremely simple. We run ADIFOR with instructions to generate calls to SparsLinC. Then, at runtime, we set the seed matrix S to the identity matrix using the SparsLinC interface routines. No knowledge of the sparsity structure is required. On the other hand, this approach is likely to be slower than the compressed Jacobian approach because of the need to maintain dynamic data structures for the representation of the sparse vectors. We also note that, unlike the compressed Jacobian matrix approach, this approach is applicable to Jacobian matrices that have a few dense rows; SparsLinC will allocate a few long vectors for the dense rows and will maintain all others as short vectors.

3 Test Problems

A wide variety of large-scale optimization problems in applications can be formulated as variational problems where we need to find a function $v : \mathcal{D} \mapsto \mathbb{R}^p$ that minimizes a functional of the form

$$\int_{\mathcal{D}} \Phi(x, v, \nabla v)\, dx, \tag{10}$$

where $\mathcal{D}$ is some domain in $\mathbb{R}^2$, and Φ is defined by the application.

Finite element approximations to these problems are obtained by minimizing (10) over the space of piecewise linear functions v with values $v_{i,j}$ at $z_{i,j}$, $0 \le i \le n_y+1$, $0 \le j \le n_x+1$,

where $z_{i,j} \in \mathbb{R}^2$ are the vertices of a triangulation of $\mathcal{D}$ with grid spacings h_x and h_y. The vertices $z_{i,j}$ are chosen to be a regular lattice so that there are n_x and n_y interior grid points in the coordinate directions, respectively. Lower triangular elements T_L are defined by vertices $z_{i,j}, z_{i+1,j}, z_{i,j+1}$, while upper triangular elements T_U are defined by vertices $z_{i,j}, z_{i-1,j}, z_{i,j-1}$. A typical triangulation is shown in Figure 1.

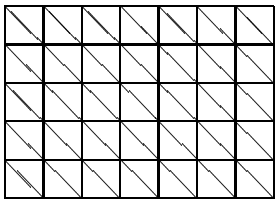

Figure 1: Triangulation of domain $\mathcal{D}$

The finite element approximation to (10) is defined by the values $v_{i,j}$ of a piecewise linear functions at $z_{i,j}$. The values $v_{i,j}$ are obtained by solving the minimization problem

$$\min \left\{ \sum_{(i,j)} \left(f_{i,j}^L(v) + f_{i,j}^U(v) \right) : v \in \mathbb{R}^n \right\}, \tag{11}$$

where $f_{i,j}^L$ and $f_{i,j}^U$ are the finite element approximation to the integrals in the elements T_L and T_U, respectively. This problem can be expressed in partially separable form by setting

$$f(v) = \begin{pmatrix} f_{1,1}^L(v) \\ f_{1,2}^L(v) \\ \vdots \\ f_{1,1}^U(v) \\ f_{1,2}^U(v) \\ \vdots \end{pmatrix}. \tag{12}$$

The Jacobian matrix of this mapping is sparse, since the element functions $f_{i,j}^L(v)$ and $f_{i,j}^U(v)$ depend only on $v_{i,j}, v_{i+1,j}, v_{i,j+1}$ and $v_{i,j}, v_{i-1,j}, v_{i,j-1}$, respectively, and thus the techniques presented in Section 1 are directly applicable to the computation of the Jacobian matrix of this mapping.

There are other ways to express problem (11) in partially separable form. For example, by accumulating the contributions of the lower triangular elements T_L and T_U, we obtain the mapping

$$f(v) = \begin{pmatrix} f_{1,1}^L(v) + f_{1,1}^U(v) \\ f_{1,2}^L(v) + f_{1,2}^U(v) \\ \vdots \end{pmatrix}. \tag{13}$$

A difference between formulations (12) and (13) is that the number of element functions $m \approx 2n$ for (12), while $m \approx n$ for (13). This implies, in particular, that the number of groups p determined by the graph-coloring software is likely to be different, and thus the computing times for the compressed Jacobian matrix may depend on p. In our preliminary experience, however, the computing time of different formulations did not differ significantly.

We selected six problems from the MINPACK-2 test problem collection to compare the different approaches for computing the gradient of a partially separable function. The selected problems are representative of large-scale optimization problems arising from applications in superconductivity, optimal design, combustion, and lubrication. We give only a brief description of two of these problems to illustrate the partially separable structure of these problems. For further information refer to [1].

The Ginzburg-Landau (GL2) problem is of the form (10), where $v : \mathbb{R}^2 \mapsto \mathbb{R}^4$. The first two components of v represent a complex-valued function $\psi : \mathcal{D} \mapsto \mathbb{C}$ (the order parameter), and the other two components represent a vector-valued function $A : \mathcal{D} \mapsto \mathbb{R}^2$ (the vector potential). This problem has the form

$$\min\{f_1(\psi) + f_2(\psi, A) : \psi, A \in H_0^1(\mathcal{D})\},$$

where $\mathcal{D}$ is a two-dimensional region,

$$f_1(\psi) = \int_{\mathcal{D}} \left\{ -|\psi(x)|^2 + \tfrac{1}{2}|\psi(x)|^4 \right\} \, dx,$$

$$f_2(\psi, A) = \int_{\mathcal{D}} \left\{ \left\| [\nabla - iA(x)]\, \psi(x) \right\|^2 + \kappa^2 \left\| (\nabla \times A)(x) \right\|^2 \right\} \, dx,$$

and κ is the Ginzburg-Landau constant.

The minimal surface area (MSA) problem is of the form

$$\min\{f(v) : v \in K\},$$

where $f : K \mapsto \mathbb{R}$ is the functional

$$f(v) = \int_{\mathcal{D}} \left(1 + \|\nabla v(x)\|^2 \right)^{1/2} \, dx,$$

and the set K is defined by

$$K = \left\{ v \in H^1(\mathcal{D}) : v(x) = v_D(x) \text{ for } x \in \partial\mathcal{D} \right\}$$

for the boundary data function $v_D : \partial\mathcal{D} \mapsto \mathbb{R}$ that specifies the Enneper minimal surface.

These two problems are partially separable, but each code is structured distinctly, resulting in a distinctly structured compressed Jacobian in each case (the other four MINPACK-2 problems, SSC, EPT, ODC, and PJB, are all structurally identical to the MSA problem).

Note that the Jacobian matrix of the MINPACK-2 problems is sparse. If $\mathrm{nnz}(f'(x))$ is the number of nonzero entries in the Jacobian matrix, then for the GL2 problem we have $n = 4n_x n_y$, $m = n$, and $\mathrm{nnz}(f'(x)) \approx 4n$, while for the MSA problem, we have $n = n_x n_y$, $m \approx 2n$, and $\mathrm{nnz}(f'(x)) \approx 6n$. On the other hand, the compressed Jacobian matrix is almost dense. For the GL2 problem (where $p = 8$), the compressed Jacobian turns out to be 50% dense, whereas for the MSA problem (where $p = 3$), the compressed Jacobian is almost completely dense. As we shall see, respectively in Sections 4.2 and 4.3, this variance in densities impacts the memory requirements and computing time performance of the ADIFOR/SparsLinC approach relative to that of the ADIFOR approach.

4 Experimental Results

We compare four methods for the computation of the gradient of a partially separable function: hand-coded derivative (HC), approximation of the compressed Jacobian matrix with function differences (FD), computation of the compressed Jacobian matrix with ADIFOR (AD), and computation of the full Jacobian matrix with ADIFOR/SparsLinC (Sparse AD). Our aim is to compare these methods with the cost of computing the function (F) and to show that in all cases (5) and (6) hold with constants Ω_T and Ω_M that are small and independent of n.

Experiments were performed on Sun SPARC 10, an IBM RS 6000 (model 370), and a Cray C90. The Fortran compiler was used with all optimization options enabled (on the Sun, we employed F77 version 1.4 with the -O option; on the IBM, XLF version 3.1.2.3 with the -O option; and on the Cray, CFT77 version 6.0.3.20 with the -O inline3 -Oscalar3 -O task0 -O vector3 -Wf"-dp" options). All computations were done with 64-bit arithmetic.

The MINPACK-2 problems were used as a test set because the availability of hand-coded gradients provides a metric in terms of accuracy, computing time, and memory requirements. The emphasis of our work is to show the effectiveness of automatic differentiation tools for computing gradients, given that for many problems hand-coding of derivatives is non-trivial and prohibitive in cost.

4.1 Numerical Accuracy

In terms of numerical accuracy, the approaches based on automatic differentiation were accurate to near machine precision, while the approach based on function differences were accurate up to at most half of the number of possible significant digits. We do not elaborate further on this point because this contrast in accuracies between automatic differentiation and function differences shows consistency with previously published work [3] on the computation of sparse Jacobian matrices with automatic differentiation.

Table 1: Memory Requirements for GL2 ($n = 160,000, p = 8$)

Platform	F	FD	FD/F	AD	AD/F	Sparse AD	Sparse AD/F
SPARC / IBM	2.59	31.39	12.1	48.13	18.6	38.65	15.0
Cray C90	3.07	42.76	13.9	59.50	19.4	59.74	19.5

Table 2: Memory Requirements for MSA ($n = 160,000, p = 3$)

Platform	F	FD	FD/F	AD	AD/F	Sparse AD	Sparse AD/F
SPARC / IBM	2.57	34.68	13.5	33.38	13.0	39.64	15.4
Cray C90	2.99	49.19	16.5	47.90	16.0	60.37	20.2

Table 3: Memory Requirements for SSC, EPT, ODC, or PJB ($n = 160,000, p = 3$)

Platform	F	FD	FD/F	AD	AD/F	Sparse AD	Sparse AD/F
SPARC / IBM	1.29	33.38	25.8	32.09	24.8	38.55	29.9
Cray C90	1.72	47.91	27.9	46.61	27.1	59.39	34.9

4.2 Memory Requirements

Tables 1 and 2 present, respectively for the GL2 and MSA problems, the total memory required for the computation of the function as well as the various gradient methods, for the case of $n = 160,000$ variables. The remaining four problems have identical memory requirements to each other; these are shown in Table 3.

We measured memory with the Unix command `size` *executable-file*, which reports the total amount of statically allocated memory (memory requirements that can be assessed at compile time) needed to load and run the executable. In the case of SparsLinC, where memory is also allocated dynamically, we call a SparsLinC routine that reports the total amount of dynamically allocated memory, and we add this to the statically allocated memory.

The AD and FD approaches have similar memory requirements for the gradient computation. In both cases, memory requirements for the compressed Jacobian matrix are proportional to the product mp, where m is the number of component functions of f, and p is the number of groups determined by the graph-coloring algorithm. Sparsity pattern and graph-coloring computations, present in both approaches, require memory proportional to $\mathrm{nnz}(f'(x))$, the total number of nonzeros in the Jacobian matrix. Each approach also has some distinct memory requirements which account for the differences between the two in Tables 1–3.

For the Sparse AD approach, much of the memory is allocated dynamically and based

Table 4: Memory Increase Factor Ω_M

	SPARC/IBM	Cray C90
FD	$p \leq \Omega_M \leq 9p$	$2p \leq \Omega_M \leq 9p$
AD	$2p \leq \Omega_M \leq 8p$	$2p \leq \Omega_M \leq 9p$
Sparse AD	$2p \leq \Omega_M \leq 10p$	$2p \leq \Omega_M \leq 12p$

on the need to represent nonzero derivative information. Certainly, the memory needed for representing the sparse Jacobian matrix has a lower bound of $\mathrm{nnz}(f'(x))$. Beyond this, SparsLinC requires additional memory for internal representations as explained in [8].

The first column in Tables 1–3 shows the memory required for running the original function. Memory requirements for the hand-coded MINPACK-2 gradient codes are not shown separately, but are always between a factor of 1.5–2 times the memory requirements of the corresponding function. The next three double columns show the memory requirements of the FD, AD, and Sparse AD approaches in megabytes (Mbytes) and as the ratio of gradient to function memory requirements. The memory requirements on the SPARC 10 and IBM RS 6000 are identical, while the Cray C90 requires more memory because the Cray default length for integer variables is 64 bits, whereas it is 32 bits on the workstation platforms. This is particularly noticeable for the Sparse AD approach, which maintains integer arrays for sparse vector data structures.

The results in Tables 1–3 show that the memory requirements of the different gradient computations are 12–35 times those of the corresponding function computations.

The memory requirements can also be measured in terms of the possible range of the constant Ω_M in (6). Table 4 shows that Ω_M is a small multiple of p. In these results we have rounded the coefficients of p to the nearest integer, since we are interested only in general trends.

All three approaches are comparable in terms of memory requirements. The worst performance is obtained for the problems in Table 3 because the function codes for these problems are relatively simple and require only the storage of the vector x. The results for the GL2 and MSA problems are more representative because these problems have work arrays in the function code. In general we expect the Sparse AD approach to require less memory than AD when the compressed Jacobian matrix is sparse. Indeed, the Sparse AD approach requires about 20% less memory on the workstation platforms for the GL2 problem, where the compressed Jacobian matrix is 50% sparse.

Table 5: Gradient-to-Function Runtime Ratios for Sparse AD on the Cray C90

n	10,000	40,000	90,000	160,000
GL2	1,390	1,790	1,710	1,790
MSA	70.7	72.1	72.2	72.6

4.3 Computing Time

Figure 2 summarizes the GL2 and MSA results for the SPARC 10, IBM RS 6000 and Cray C90. Each figure shows the gradient-to-function computing time ratio for each of the four methods for computing the gradient. We have included data for problems with $n = 2,500$ variables to $n = 160,000$. The solid line indicates the Sparse AD approach, the dotted line the AD approach, the dashed line the FD approach, and the dash-dotted line is the hand-coded derivatives (HC).

The main conclusion that can be drawn from Figure 2 is that the gradient-to-function computing time ratio is independent of the problem size for these two problems. This is an important aspect of these results, since our main goal is to avoid the cost of n function evaluations for approximating the gradient by differences of function values. The gradient-to-function ratios for SparsLinC on the Cray C90 are not shown in Figure 2 because inclusion of these ratios would distort the plots. Table 5 show that these ratios, though larger, are also independent of n.

We are also interested in the ratio of computing times between the various approaches and their relation to the time required for the coloring preprocessing step. These ratios appear in Table 6 for all the problems under consideration, but only for $n = 160,000$. The plots in Figure 2 show that these ratios are essentially independent of the number n of variables, and thus the results in Table 6 are representative for any reasonable number of variables.

Before we analyze the runtime results, we briefly summarize important features of the underlying architectures. The SPARC 10 essentially has a scalar processor and a flat memory hierarchy. Hence, vector operations execute only marginally faster, and memory locality (that is, the reuse of data and the accessing of adjacent memory locations) is not much of an issue. In contrast, the IBM RS 6000 architecture employs a superscalar chip and a cache-based memory architecture. Hence, this machine performs better if executing short vector operations, since these operations can fill the short pipes and take advantage of memory locality. On the other hand, indirect addressing, used extensively in SparsLinC and in the coloring algorithm, while fairly inconsequential on the SPARC, may lead to performance degradation, as memory locality suffers. The Cray C90 is a vector processor without a cache

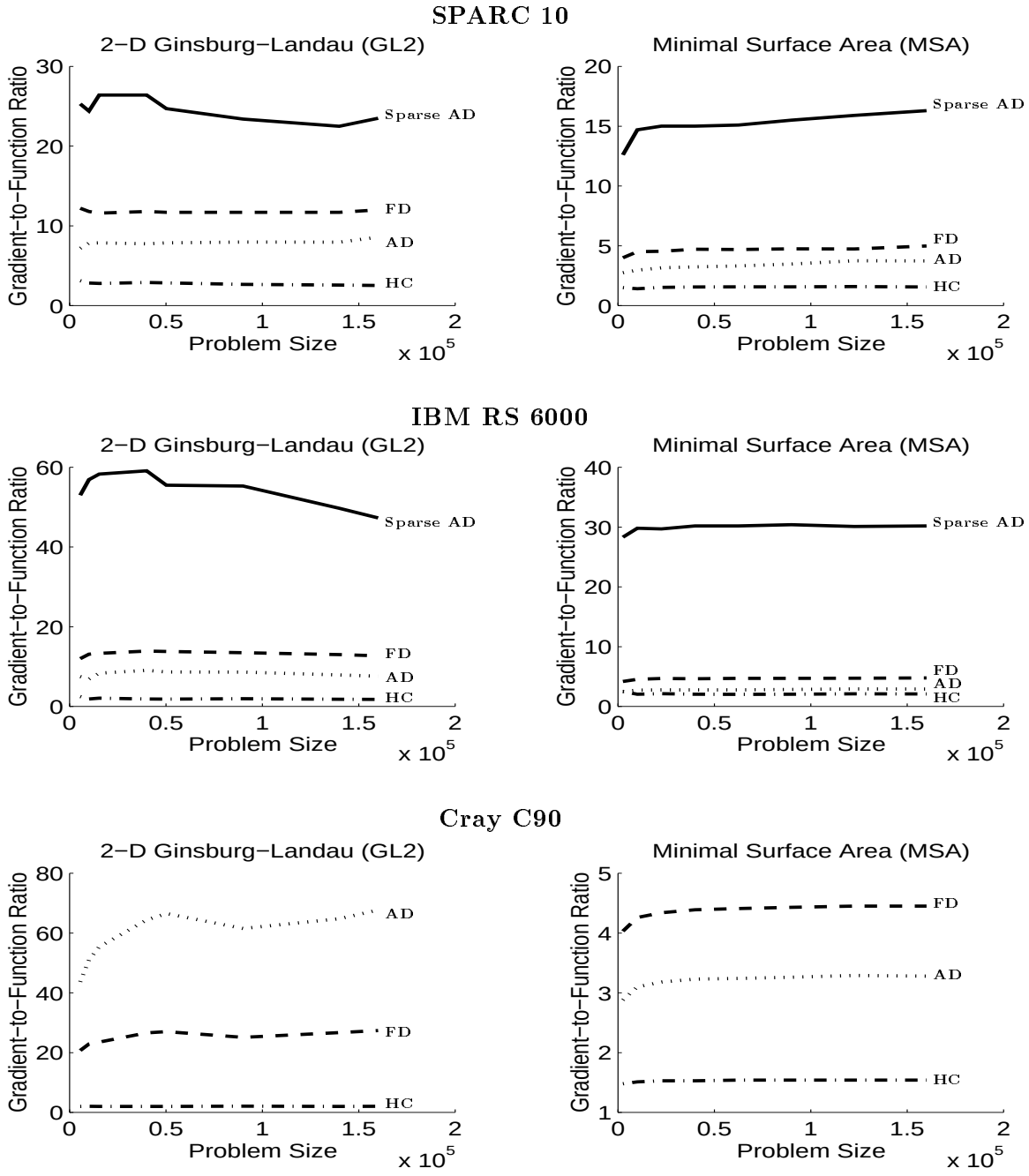

Figure 2: Ratios of computing times between the gradient and the function. FD (dashed), AD (dotted), Sparse AD (solid), HC (dash-dotted)

Table 6: Coloring-to-Function and Gradient-to-Function Runtime Ratios ($n = 160,000$)

Ginzburg-Landau (GL2) problem ($p = 8$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	18.36	2.52	12.00	8.58	23.50
IBM RS 6000	36.24	1.78	12.70	7.58	47.30
Cray C90	664.29	2.05	27.40	67.70	1790.00

Minimal Surface Area (MSA) problem ($p = 3$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	7.18	1.55	4.98	3.74	16.30
IBM RS 6000	11.62	2.09	4.77	2.90	30.20
Cray C90	12.57	1.54	4.45	3.28	72.60

Steady State Combustion (SSC) problem ($p = 3$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	4.43	1.28	4.63	3.08	18.00
IBM RS 6000	5.58	1.48	4.39	2.12	26.50
Cray C90	86.51	18.00	7.54	33.60	902.00

Optimal Design with Composites (ODC) problem ($p = 3$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	5.35	1.28	4.79	3.37	15.70
IBM RS 6000	7.68	1.43	4.55	2.56	26.30
Cray C90	10.25	2.09	4.41	4.95	77.60

Elastic-Plastic Torsion (EPT) problem ($p = 3$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	13.24	1.59	5.99	5.67	43.80
IBM RS 6000	23.88	2.50	5.71	4.46	87.70
Cray C90	331.98	25.5	17.50	63.30	2800.00

Pressure in a Journal Bearing (PJB) problem ($p = 3$)

Platform	Coloring	HC	FD	AD	Sparse AD
SPARC 10	12.64	1.92	5.82	5.06	25.20
IBM RS 6000	18.24	2.13	5.53	4.06	41.50
Cray C90	204.63	64.70	12.20	39.10	1260.00

and achieves its full potential only when the code exhibits long vector operations. Without optimization of the source Fortran code, short vector loops and indirect addressing schemes exhibit much lower performance, since the hardware pipes cannot get filled and the speed of main memory is much slower than that of the CPU.

Based on the architectures used in our testing, we expect computing times to be stable and predictable on workstation platforms but expect that vectorization issues will cause a large variation in computing times on vector architectures. Our experimental results bear out these expectations.

As expected, the hand-coded derivative code is the fastest on the scalar architectures. For the results in Table 6 we have

$$T\{\nabla f_0(x) : \mathrm{HC}\} \leq 3\, T\{f_0\}, \tag{14}$$

where $T\{\nabla f_0(x) : \cdot\}$ is the time required to compute the gradient of the partially separable function by a particular method. The above ratio can be expected for well-coded gradient computations on scalar architectures but requires special techniques on vector and parallel architectures [2].

On vector architectures we can expect the ratio (14) to hold only if both the function and the gradient evaluation codes vectorize or if neither code vectorizes. An examination of Cray C90 results shows that only the MSA and ODC function evaluation codes fail to vectorize, and that the GL2 hand-coded gradient evaluation code is the only HC code that vectorizes. Our results support this remark because we obtain a high gradient-to-function runtime ratio only on problems where only the function evaluation code vectorizes (i.e., SSC, EPT, and PJB).

The results in Table 6 show that the AD approach outperforms the FD approach on scalar architectures. The performance of the various approaches on vector architectures is harder to predict as performance depends on the delicate interplay between the code and the compiler (for examples, see [7, 11]). Note that the results in Table 6 show that the performance of AD is comparable to that of FD on the Cray C90 for those problems (MSA and ODC) where the function evaluation code fails to vectorize.

Our numerical results also show that the AD approach outperforms the Sparse AD approach on all the architectures. From the results in Table 6 we can observe that

$$T\{\nabla f_0(x) : \text{Sparse ADIFOR}\} \leq \kappa\, T\{\nabla f_0(x) : \text{ADIFOR}\},$$

where κ satisfies

SPARC 10	IBM RS 6000	Cray C90
$3 \leq \kappa \leq 8$	$6 \leq \kappa \leq 20$	$15 \leq \kappa \leq 45$

.

In all our experiments with the exception of the GL2 problem, the compressed Jacobian is almost fully dense. It is not surprising that AD outperforms Sparse AD on these problems,

Table 7: Time Increase Factor Ω_T

	SPARC 10	IBM RS 6000	Cray C90
FD	$p \leq \Omega_T \leq 2p$	$p \leq \Omega_T \leq 2p$	$p \leq \Omega_T \leq 6p$
AD	$p \leq \Omega_T \leq 2p$	$p \leq \Omega_T \leq 2p$	$p \leq \Omega_T \leq 20p$
Sparse AD	$3p \leq \Omega_T \leq 15p$	$6p \leq \Omega_T \leq 30p$	$25p \leq \Omega_T \leq 930p$

given that the runtime efficiency of SparsLinC is expected to become apparent for problems that have much sparser compressed Jacobians. Note that Sparse AD performs much better on the GL2 problem, where the compressed Jacobian is 50% sparse, compared with the other problems.

We can compare the performance of the various approaches by computing the range for the constant Ω_T in (5) as a function of p. These results, with the coefficients of p rounded to the nearest integer, are shown in table 7.

This table shows that in most cases Ω_T is a small multiple of p.

We note the wide variation in Ω_T for FD and AD on the vector architecture owing to the code-dependent effects of vectorization, as already discussed. We also note the large variation in Ω_T for the Sparse AD results on the SPARC 10. This results from the way SparsLinC exploits the particular sparsity characteristics of each problem (this issue is explored in [8]). Finally, we note that the performance of Sparse AD degrades on vector computers, as a result of pervasive use of indirect addressing and lack of vector instructions, though this performance could be improved through the use of hardware-supported gather/scatter instructions.

Table 6 also compares the cost of the graph coloring algorithm with the cost of computing the function. The high relative cost of computing the graph coloring is mainly a reflection of the low cost of computing the functions for these problems. We can justify this remark by noting that evaluation of the component functions for the GL2, EPT, and PJB problems only require the evaluation of a low-order polynomial and, that for these problems, the coloring-to-function runtime ratio is high. On the other hand, the problems with a low coloring-to-function runtime ratio are relatively expensive to evaluate; the SSC problem requires the evaluation of the exponential function, while the MSA and SSC problems require a square root.

Another reason for the high relative cost of computing the graph coloring is that the algorithm we employ (subroutine DSM from Coleman, Garbow, and Moré [12]) is intended to produce graph colorings with a small p by employing several heuristics. The runtime of subroutine DSM could be reduced by a factor of two or more without a substantial increase in p by only using one of the heuristics. Also note that the graph coloring algorithms share

many of the characteristics of Sparse AD with respect to indirect addressing and memory locality, and thus the performance of the coloring algorithm deteriorates on the RS 6000 and C90 platforms.

5 Conclusions

We have shown that automatic differentiation outperforms difference approximations of derivatives and offers high numerical accuracy without the need for hand-coding. The approach based on the compressed Jacobian matrix with the ADIFOR tool produces code that is often not more than four times slower than a well-coded hand-derived gradient code on scalar architectures. This approach, however, requires the sparsity pattern of the partially separable function.

The approach based on the ADIFOR/SparsLinC tool set is the ultimate in convenience, as not even the sparsity pattern of the underlying Jacobian matrix is needed. In fact, the sparsity pattern is a byproduct of the ADIFOR/SparsLinC approach. On the other hand, this approach is considerably slower, particularly on vector architectures.

Acknowledgments

We thank Andreas Griewank for stimulating discussions on the subject and Alan Carle for his instrumental role in the ADIFOR project. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

References

[1] AVERICK, B., CARTER, R., MORÉ, J., and XUE, G.-L., 1992. The MINPACK-2 test problem collection, Technical Report ANL/MCS-TM-150, Revised, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] AVERICK, B. and MORÉ, J., 1994. Evaluation of large-scale optimization problems on vector and parallel architectures, *SIAM Journal on Optimization* 4, 708–721.

[3] AVERICK, B., MORÉ, J., BISCHOF, C., CARLE, A., and GRIEWANK, A., 1994. Computing large sparse Jacobian matrices using automatic differentiation, *SIAM Journal Scientific and Statistical Computing* 15, 285–294.

[4] BISCHOF, C., CARLE, A., CORLISS, G., GRIEWANK, A., and HOVLAND, P., 1992. ADIFOR: Generating derivative codes from Fortran programs, *Scientific Programming* 1(1), 11–29.

[5] BISCHOF, C., CARLE, A., and KHADEMI, P., 1994. Fortran 77 interface specification to the SparsLinC library, Technical Report ANL/MCS-TM-196, Mathematics and Computer Science Division, Argonne National Laboratory.

[6] BISCHOF, C., CARLE, A., KHADEMI, P., and MAUER, A., 1994. The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University. Forthcoming in IEEE Computational Science & Engineering.

[7] BISCHOF, C., GREEN, L., HAIGLER, K., and KNAUFF, T., 1994. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation, *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4261, 73–84. American Institute of Aeronautics and Astronautics.

[8] BISCHOF, C., KHADEMI, P., BOUARICHA, A., and CARLE, A., 1996. Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation, *Optimization Methods and Software* 7(1), 1–39.

[9] BISCHOF, C. and EL-KHADIRI, M., 1992. Extending compile-time reverse mode and exploiting partial separability in ADIFOR, Technical Report ANL/MCS-TM-163, Mathematics and Computer Science Division, Argonne National Laboratory.

[10] BISCHOF, C., JONES, W., MAUER, A., and SAMAREH, J., 1996. Experiences with the application of the ADIC automatic differentiation tool to the CSCMDO 3-D volume grid generation code, *Proceedings of the 34th AIAA Aerospace Sciences Meeting*, pages AIAA 96–0716. American Institute of Aeronautics and Astronomics.

[11] BOUARICHA, A., and MORÉ, J., 1995. Impact of partial separability on large-scale optimization. Preprint MCS-P487-0195, Argonne National Laboratory, Argonne, Illinois. Forthcoming in Computational Optimization and Applications.

[12] COLEMAN, T., GARBOW, B., and MORÉ, J., 1984. Fortran subroutines for estimating sparse Jacobian matrices, *ACM Transactions on Mathematical Software* 10,346–347.

[13] COLEMAN, T. and MORÉ, J., 1983. Estimation of sparse Jacobian matrices and graph coloring problems, *SIAM Journal on Numerical Analysis* 20, 187–209.

[14] CONN, A.R., GOULD, N.I.M., and TOINT, P.L., 1992. LANCELOT, Springer Series in Computational Mathematics, Springer-Verlag.

[15] CURTIS, A., POWELL, M., and REID, J., 1974. On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.* 13,117–119.

[16] GIERING, R., 1992. Adjoint model compiler, manual version 0.2, AMC version 2.04, Technical Report, Max-Planck Institut für Meteorologie.

[17] GRIEWANK, A. and TOINT, P.L., 1982. On the unconstrained optimization of partially separable functions, in *Nonlinear Optimization 1981* (M. J. D. Powell, ed.), Academic Press.

[18] GRIEWANK, A. and TOINT, P.L., 1982. Partitioned variable metric updates for large structured optimization problems, *Numerische Mathematik* 39, 119–137.

[19] GRIEWANK, A. and TOINT, P.L., 1984. 'Numerical experiments with partially separable optimization problems, in *Numerical Analysis: Proceedings Dundee 1983* (D. F. Griffiths, ed.), Lecture Notes in Mathematics 1066, Springer-Verlag.

[20] GRIEWANK, A., 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation, *Optimization Methods and Software*, 1(1), 35–54.

[21] GRIEWANK, A., 1993. Some bounds on the complexity of gradients, Jacobians, and Hessians, in P.M. Pardalos, editor, *Complexity in Nonlinear Optimization*, 128–161. World Scientific Publishers.

[22] GRIEWANK, A. and CORLISS, G., eds., 1991. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Society for Industrial and Applied Mathematics.

[23] GRIEWANK, A., JUEDES, D., and UTKE, J., 1996. ADOL-C: A package for the automatic differentiation of algorithms written in C/C+, *ACM Transactions on Mathematical Software* 22, 131–167.

[24] HORWEDEL, J., 1991. GRESS: A preprocessor for sensitivity studies on Fortran programs, [22], 243–250.

[25] JUEDES, D., 1991. A taxonomy of automatic differentiation tools, [22], 315–330.

[26] KUBOTA, K., 1991. PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives, [22], 251–262.

[27] LESCRENIER, M., 1988. Partially separable optimization and parallel computing, *Annals Operations Research* 14, 213–224.

[28] RALL, L., 1981. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

[29] ROSTAING, N., DALMAS, S., and GALLIGO, A., 1993. Automatic differentiation in Odyssee, *Tellus*, 45a(5), 558–568.

[30] SOULIE, E. User's experience with Fortran compilers for least squares problems, [22], 297–306.

[31] TOINT, P.L., 1986. Numerical solution of large sets of algebraic nonlinear equations, *Mathematics of Computation* 46, 175–189.

[32] TOINT, P.L., 1987. On large scale nonlinear least squares calculations, *SIAM Journal Scientific and Statistical Computing* 8, 416–435.

[33] TOINT, P.L., and TUYTTENS, D., 1990. On large-scale nonlinear network optimiz ation, *Mathematical Programming* 48, 125–159.

[34] TOINT, P.L., and TUYTTENS, D., 1992. LSNNO: A Fortran subroutine for solving large-scale nonlinear network optimization problems, *ACM Transactions on Mathematical Software* 18, 308–328.