Computational Differentiation and Multidisciplinary Design*

Christian Bischof† and **Andreas Griewank**‡

Preprint MCS-P420-0294

to appear in Proceedings of the Symposium on Inverse Problems and Optimal Design in Industry, J. McLaughlin and H. Engl, Eds, Teubner Verlag, Stuttgart, Germany

Abstract

Multidisciplinary Design Optimization (MDO) by means of formal sensitivity analysis requires that each single-discipline analysis code supply not only the output functions for the (usually constrained) optimization process and other discipline analysis inputs, but also the derivatives of all of these output functions with respect to its input variables. Computational differentiation techniques and automatic aifferentiation tools enable MDO by providing accurate and efficient derivatives of computer programs with little human effort. We discuss the principles behind automatic differentiation and give a brief overview of automatic differentiation tools and how they can be employed judiciously, for example, for sparse Jacobians and to exploit parallelism. We show how, and under what circumstances, automatic differentiation applied to iterative solvers delivers the mathematically desired derivatives. We then show how derivatives that can now be feasibly obtained by computational differentiation techniques can lead to improved solution schemes for nonlinear coupled systems and multidisciplinary design optimization.

1 Introduction

Computational differentiation (CD) provides a new foundation for sensitivity analysis and subsequent design optimization of complex systems by reliably computing derivatives of large computer codes. The goal is to free the computational scientist from worrying about the accurate and efficient computation of derivatives, even for complicated "functions," thereby enabling him to concentrate on the more important issues of system modeling and algorithm design.

Almost as soon as the first programmable systems became available, scientists observed that the mechanical application of the chain rule to obtain derivatives can be *automatically* and reliably performed by computers. The basic *forward*, or *bottom-up*, mode of automatic differentiation (AD) was apparently first proposed by R. E. Wengert [41]. The mathematically more interesting *reverse*, or *top-down*, mode was first published by G. M. Ostrowskii [36]. This observation gives rise to *automatic differentiation* (AD), which can compute derivatives of a function defined by a computer code in a black-box fashion, without any knowledge of the application beyond what are considered "dependent" and "independent" variables with respect to differentiation.

While these original results have been repeatedly rediscovered and extended, the full potential of automatic differentiation as a general-purpose computational tool has not yet

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Aerospace Agency under Purchase Order L25935D.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, U.S.A., `bischof@mcs.anl.gov`.

‡Institute of Scientific Computing, Technical University Dresden, Mommsenstr. 13, 01062 Dresden, Germany, `griewank@math.tu-dresden.de`.

been realized. Historically, this is due both to a lack of general-purpose tools and a lack of scientific communication and cooperation. It was not until 1991 that the first meeting focusing on Computational Differentiation was held [24]. Since then, much has changed as automatic differentiation tools have acquired a level of maturity that enables them to deliver the promises of the theory.

As researchers have gained experience with automatic differentiation tools, it also has become apparent that the process of generating suitable derivative codes is not necessarily automatic. For example, the user of an AD tool can capitalize, in a high-level fashion, on the structure of his program to decrease the computational cost of computing derivatives, leading to an "automated," or "computer-supported" approach to generating derivative code.

Moreover, there is the issue of AD versus "do-what-I-mean," where one has to relate knowledge about the algorithms underlying the program in question with automatic differentiation to ensure that the derivatives computed are the ones that are mathematically desired. Computational differentiation is the term we chose to denote all these closely related efforts.

This paper is structured as follows. In the next section we review automatic differentiation and comment on the mathematical and computer science challenges that remain. In Section 3 we give examples that show how judicious use of AD tools can have a significant effect on the speed with which one computes derivatives. We then explore, in Section 4, whether, and under what circumstances, AD techniques, when applied to iterative solvers, deliver the desired derivatives. Section 5 illustrates how computational differentiation techniques can be employed to arrive at faster and more robust procedures for multidisciplinary design analysis and optimization. Lastly, we give a brief outlook on the field.

2 Automatic Differentiation

Because accurate and efficient derivative values are so crucial for the speed and robustness of numerical methods, many developers have expended great efforts to derive, by hand, code for the derivatives of particular modeling functions. Obviously there would be little point in discussing the importance of AD if there were other approaches for obtaining derivatives accurately and cheaply. We have already mentioned that divided differences are of uncertain quality in terms of accuracy and are also quite costly if there are many independent variables. Fully symbolic differentiation, as provided, for example, by the "diff" operator in MAPLE, is generally not a realistic alternative, because with the possible exception of some right-hand sides in ordinary differential equations, the problem functions in the applications mentioned above are evaluated by computational procedures of some length. Consequently, the direct representation and differentiation of the dependent variables in terms of the independent variables are generally very resource demanding or impossible, while they pose no difficulties for automatic differentiation (see, for example, [9, 28]).

In this section, we give an overview of the classical approaches to automatic differentiation, as well as some of the more recent research questions relating to the computational complexity for computing derivatives. We also give a brief overview automatic differentiation tools, in particular the ADIFOR Fortran77 tool.

2.1 The Forward and Reverse Mode of Automatic Differentiation

In contrast to fully symbolic packages, automatic differentiation applies the chain rule to numbers rather than algebraic expressions. These real numbers are, of course, rounded after each application of the chain rule, so that the complexity of representing and manipulating

$$\begin{array}{lll}
v_1 = x_2 * x_3 & & \\
v_2 = v_1 x_1 & // & = x_1 x_2 x_3 \\
v_3 = v_1 x_4 & // & = x_2 x_3 x_4 \\
y_1 = c_1 v_2 + d_1 v_3 & // & = c_1 x_1 x_2 x_3 + d_1 x_2 x_3 x_4 \\
y_2 = c_2 v_2 + d_2 v_3 & // & = c_2 x_1 x_2 x_3 + d_2 x_2 x_3 x_4 \\
y_3 = c_3 v_2 + d_3 v_3 & // & = c_3 x_1 x_2 x_3 + d_3 x_2 x_3 x_4
\end{array}$$

Figure 1: Example program with 4 independents and 3 dependents

$$\begin{array}{lll}
\nabla v_1 = x_2 \nabla x_3 + x_3 \nabla x_2 & // & (0, x_3, x_2, 0) \\
\nabla v_2 = x_1 \nabla v_1 + v_1 \nabla x_1 & // & (v_1, x_1 x_3, x_1 x_2, 0) \\
\nabla v_3 = x_4 \nabla v_1 + v_1 \nabla x_4 & // & (0, x_4 x_3, x_4 x_2, v_1) \\
\nabla y_1 = c_1 \nabla v_2 + d_1 \nabla v_3 & // & (c_1 v_1, c_1 x_1 x_3 + d_1 x_4 x_3, c_1 x_1 x_2 + d_1 x_4 x_2, d_1 v_1) \\
\nabla y_2 = c_2 \nabla v_2 + d_2 \nabla v_3 & // & (c_2 v_1, c_2 x_1 x_3 + d_2 x_4 x_3, c_2 x_1 x_2 + d_2 x_4 x_2, d_2 v_1) \\
\nabla y_3 = c_3 \nabla v_2 + d_3 \nabla v_3 & // & (c_3 v_1, c_3 x_1 x_3 + d_3 x_4 x_3, c_3 x_1 x_2 + d_3 x_4 x_2, d_3 v_1)
\end{array}$$

Figure 2: Forward mode evaluation of the 4×3 Jacobian

each partial derivative does not grow at all. It has been shown that automatic differentiation is backward stable in the sense that the derivative values obtained correspond to the exact results for slightly perturbed independent variables and intermediates [26].

As an example let us consider the following simple program with four independent variables x_1, x_2, x_3, x_4 and three dependent variables y_1, y_2, y_3, where the c_i and d_i are distinct constants.

To calculate the Jacobian $\partial y / \partial x$ one could associate with each scalar variable a gradient of derivatives with respect to the three independent variables. Starting from the Cartesian basis vectors $\nabla x_i = e_i \in \mathbf{R}^3$, one can calculate derivatives according to the usual interpretation of the chain rule, as shown in Figure 2. The 4-vector $\nabla v_1 = (0, x_3, x_2, 0)$ has only two nonzero components and can be composed without performing any arithmetic operations. Consequently the calculation of the gradients ∇v_2 and ∇v_3 costs 4 multiplications. Since both of these vectors have three nonzero components each the calculation of the gradients ∇y_i representing the three rows of the Jacobian $\partial y / \partial x$ costs 18 more multiplications and 6 additions. The total count is thus 22 multiplications and 6 additions.

The calculation performed above represents the so-called forward mode of automatic differentiation, with an operations count corresponding to a sparse implementation of this basic procedure. To see how a few operations can be saved by the so-called reverse mode, let us first consider a graphical representation of the dependence between the x_j and the y_i.

In the graph displayed in Figure 3 the nodes represent the independent, intermediate, and dependent variables and the edges represent the elementary partial derivatives between them. In other words, an edge connects v_i and v_j exactly when v_i occurs on the righthand side of the statement that defines v_j on the left. In that case the edge is annotated with the partial derivative of v_j with respect to v_i, evaluated at the current argument. Since all operations

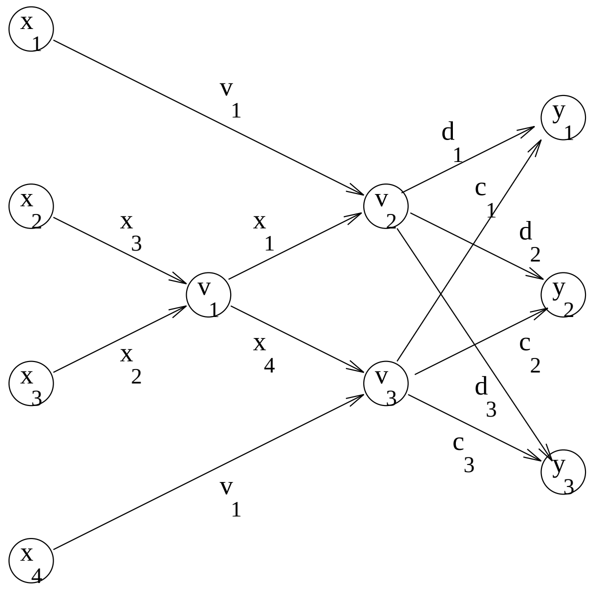


Figure 3: Computational graph with 3 intermediate nodes

in our example involve two nonconstant arguments, there are two edges arriving at each node; but in the general situation there are often univariate functions such as exponentials and trigonometric intrinsics, which generate nodes with a single incoming arc. In those situations the calculation of the elementary partials themselves may be nontrivial, though its cost is always bounded by a small multiple of the cost for evaluating the elementary functions themselves. This is true for any realistic measure of computational cost and may account for memory accesses.

Clearly the collection of elementary partials in the computational graph uniquely determines the overall Jacobian. The sparse forward procedure of Figure 2 for calculating the Jacobian combines the elementary partials to arrive at the overall Jacobian by applying the chain rule in one particular way. Contrary to what one might expect, there are several other ways in which the Jacobian can be accumulated from the elementary partials. The results are the same up to round-off, but the computational effort in terms of the number of arithmetic operations may differ.

For example, rather than successively calculating the gradients of intermediates and dependents with respect to the independents, as is done in the forward mode, one can instead compute vectors of sensitivities of the dependents with respect to the intermediate and the independents. More specifically, let us define the *adjoint* vectors

$$\bar{v}_i \quad = \quad (\partial y / \partial v_i) \in \mathbf{R}^3$$

and correspondingly $\bar{x}_i$ and $\bar{y}_i = e_i$. This time the derivative objects associated with the dependents are initialized to Cartesian basis vectors, and we proceed backwards as shown in Figure 4, using the so-called reverse mode of automatic differentiation.

We note that the results are consistent in that the four columns $\triangle x_i$ obtained in the reverse mode form exactly the same matrix as the one formed by the three rows ∇y_i computed earlier in the forward mode. This time, however, the whole procedure requires 18 rather than 22

$$\begin{array}{lll}
\bar{v}_3 = d_1\bar{y}_1 + d_2\bar{y}_2 + d_3\bar{y}_3 & // & (d_1, d_2, d_3) \\
\bar{v}_2 = c_1\bar{y}_1 + c_2\bar{y}_2 + c_3\bar{y}_3 & // & (c_1, c_2, c_3) \\
\bar{v}_1 = x_1\bar{v}_2 + x_4\bar{v}_3 & // & (x_1c_1 + x_4d_3, x_1c_2 + x_4d_2, x_1c_3 + x_4d_3) \\
\bar{x}_4 = v_1\bar{v}_3 & // & (v_1d_1, v_1d_2, v_1d_3) \\
\bar{x}_3 = x_2\bar{v}_1 & // & (x_2x_1c_1 + x_2x_4d_1, x_2x_1c_2 + x_2x_4d_2, x_2x_1c_3 + x_2x_4d_3) \\
\bar{x}_2 = x_3\bar{v}_1 & // & (x_3x_1c_1 + x_3x_4d_1, x_3x_1c_2 + x_3x_4d_2, x_3x_1c_3 + x_3x_4d_3) \\
\bar{x}_1 = v_1\bar{v}_2 & // & (v_1c_1, v_1c_2, v_1c_3)
\end{array}$$

Figure 4: Reverse mode accumulation of the 4×3 Jacobian

multiplications and 3 rather than 6 additions. Hence the reverse mode has a lower operations count on this example.

Unfortunately, apart from the computation of gradients, for which the reverse mode is optimal [19], there is no reliable rule to determine a priori whether the forward or reverse mode is better, and there are in fact even more variations on the chain rule. We also note that although it seems as if the reverse mode requires storage that is proportional to the runtime of the undifferentiated code, Griewank [21] has shown that much more reasonable compromises between temporal and spatial complexity can be achieved by a recursive checkpointing approach.

2.2 Variations of Derivative Accumulation

In the "explicit" expressions on the right margins of the informal program listed above we have deliberately used the intermediate values rather than their expansion, for example v_1 instead of x_2x_3. This way one can observe that for each pair (x_j, y_i) of an independent and a dependent variable, the corresponding entry in the Jacobian is a sum over all directed paths connecting them in the computational graph (Figure 3). The additive term associated with each path is simply the product of all edge values involved, which we will call the path product. In our simple example, x_1 and x_4 are connected by only one path to each of the three dependents, whereas x_2 and x_3 impact the same dependents via the intermediates v_2 or v_3.

In general, the number of different paths grows exponentially with the diameter of the graph, that is, the length of the longest directed path in the graph. Therefore, it is in general impossible or at least very expensive to evaluate the additive terms for each path separately and then to add them to the appropriate Jacobian entry. In our example this approach would require 2 multiplications each for the 6 partials with respect to x_1 and x_4 and 3 multiplications each for the 6 partials with respect to x_2 and x_3, yielding a total of 30 multiplications and 6 additions. Both the forward and reverse mode do better because they utilize the fact that certain subproducts like x_3x_1 or x_1d_1 occur repeatedly and can therefore be reused. The forward mode is based on partial path products starting from the independents, and the reverse mode accumulates path products starting from the dependents. In graphs with paths of length four or greater, one can also start forming partial path products in the middle, that is, without involving either independents or dependents initially.

To visualize some particular choices in the usually bewildering variety of accumulating path products, one can think of successively eliminating intermediate nodes by connecting all intermediate predecessor/successor pairs with the products of the corresponding edge values. If a direct edge already exists, its value is incremented by the product of the two arcs running through the intermediate node. In the example above, one could first eliminate the node v_1 by connecting the predecessors x_2 and x_3 with the successors v_2 and v_3 by four edges with the values x_3x_1, x_3x_4, x_2x_1, and x_2x_4. This is essentially what the forward mode does in calculating the gradients ∇v_2 and ∇v_3. Subsequently it eliminates the nodes v_2 and v_3 by multiplying the newly obtained edge values with the constant multipliers c_i and d_i, respectively. The reverse mode first eliminates the node pair (v_2, v_3) and then v_1.

Because of the symmetric role of v_2 and v_3 in Figure 3, there exists only one significantly different third elimination order, namely, v_2, v_1, v_3. Starting from Figure 3, this is depicted in Figure 5, where vw denotes the value of the edge (v, w) and $a+ = b$ is shorthand for $a = a + b$. The thick edges in the graphs on the righthand side are the ones being created or updated in the current elimination step. The total count for this elimination ordering is 23 multiplications and 6 additions, which means it is slightly worse than the forward mode but still a lot better than the explicit evaluation of all path products.

The number of multiplications in each elimination step equals exactly the product of the number of predecessors and successors of the vertex being eliminated. For all neighbors this so-called Markowitz degree is changed as some incoming or outgoing edges may merge and others are created because of fill-in. Thus we see that the problem of calculating the Jacobian of a particular vector function using the minimal number of arithmetic operations ([20]) is closely related to the corresponding task in sparse Gaussian elimination. It is therefore not surprising that the Jacobian accumulation problem can be shown to be NP-complete as welll [29]. Just as in the sparse matrix case, the development of near-optimal heuristics for accumulating Jacobians by successively eliminating intermediates provides a rich field for future research.

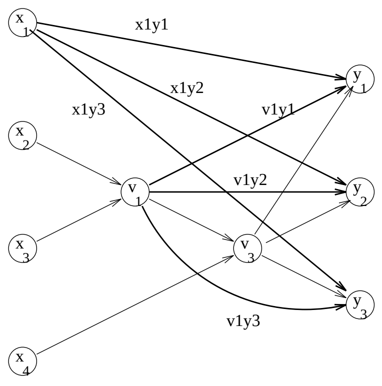
The classic greedy heuristic is the Markowitz rule, which requires that one always eliminates a vertex of minimal degree, thus minimizing the operations at the current stage. In the long run this may not be the best strategy, as one can already see from the small example given above. Here v_1 has initially the smallest Markowitz degree, namely 4, whereas v_2 and v_3 both have the degree 6. Yet eliminating v_1 first corresponds to the forward mode, which uses more operations than the reverse mode, which eliminates v_3 and v_2 before v_1. While this example is merely of academic interest, there are certain classes of discretized PDEs for which the Markowitz heuristic is a good one [26]. In general, however, these various accumulation strategies may exhibit vastly differing computational complexity. These complexity issues are further explored in [22].

The computational graphs from which the corresponding Jacobians can be obtained by successively eliminating all intermediate vertices contains one node for each arithmetic operation or intrinsic function call. Therefore, the graphs are usually so large that they cannot be stored in core; and even when they can, the interpretive overhead of manipulating them as linked data structures may easily upset the gain in theoretical complexity. Consequently, if one wishes to employ this approach in an AD tool, one must develop a constructive theory of decomposing the graph hierarchically and processing it in pieces. This approach is particularly promising in codes where subroutines with fixed-dimensional inputs (for example finite element stencil evaluators) are called many thousands of times. Since in most applications Jacobians are used only as a means of solving nonlinear systems iteratively, it may be preferable to directly attack the larger problem of computing Newton-steps or their approximations [20], [13].

Like derivatives, many other numerically useful pieces of information about a composite function can be derived from the properties and mutual relations of its elementary constituents. For example, one may determine error bounds, Lipschitz constants, trust region radii, and even parallel evaluation schedules. In this wider sense the term “automatic dif-

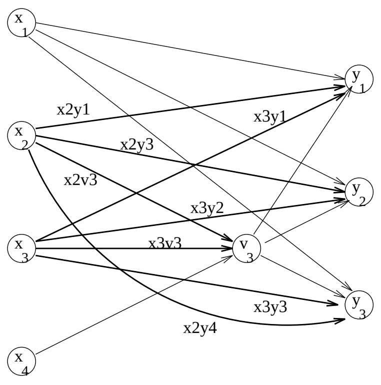
Eliminate node v_2

$$\begin{aligned}
x1y1 &= v_1 d_1\\
x1y2 &= v_1 d_2\\
x1y3 &= v_1 d_3\\
v1y1 &= x_1 d_1\\
v1y2 &= x_1 d_2\\
v1y3 &= x_1 d_3
\end{aligned}$$



Eliminate node v_1

$$\begin{aligned}
x2y1 &= x_3 v1y1\\
x2y2 &= x_3 v1y2\\
x2y3 &= x_3 v1y3\\
x3y1 &= x_2 v1y1\\
x3y2 &= x_2 v1y2\\
x3y3 &= x_2 v1y3\\
x2v3 &= x_3 x_4\\
x3v3 &= x_2 x_4
\end{aligned}$$



Eliminate node v_3

$$\begin{aligned}
x2y1 &\mathrel{+}= c_1 x2v3\\
x2y2 &\mathrel{+}= c_2 x2v3\\
x2y3 &\mathrel{+}= c_3 x2v3\\
x3y1 &\mathrel{+}= c_1 x3v3\\
x3y2 &\mathrel{+}= c_2 x3v3\\
x3y3 &\mathrel{+}= c_3 x3v3\\
x4y1 &= v_1 c_1\\
x4y2 &= v_1 c_2\\
x4y3 &= v_1 c_3
\end{aligned}$$

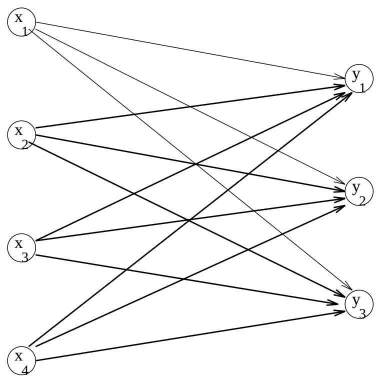


Figure 5: Another possibility for accumulation of the 4×3 Jacobian

ferentiation" is unduly restrictive, as it does not indicate the ability to build much more general models of nonlinear functions near a given argument. Unfortunately, no convincing alternative terminology has yet emerged.

2.3 Automatic Differentiation Tools

There have been various implementations of automatic differentiation, and an extensive survey can be found in [31]. In particular, we mention GRESS [30] and PADRE-2 [34] for Fortran Programs, and ADOL-C [25] for C and C++ programs. ADOL-C employs the operator overloading capabilities of the C++ language to transparently augment the original code with computations for derivative of arbitrary order. Operator overloading also allows for a flexible and easily customizable system. These issues are further explored in [12].

GRESS, PADRE-2, and ADOL-C implement both the forward and reverse mode. The reverse mode requires that one saves or recomputes all intermediate values that nonlinearly impact the final result, and to this end these tools generate, a trace of the computation. The interpretation overhead of this trace and its potentially very large size can be a serious computational bottleneck [39].

Recently, a "source transformation" approach to automatic differentiation has been explored in the ADIFOR [3] and ODYSSEE [37] projects. Both tools transform Fortran code, applying the rules of automatic differentiation and generating new Fortran code that, when executed, computes derivatives without the overhead associated with "tape interpretation" schemes. ODYSEE generates reverse mode; ADIFOR uses a hybrid forward/reverse mode strategy.

ADIFOR (Automatic Differentiation in Fortran) [3, 7] provides automatic differentiation for 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 portable Fortran 77 code that allows the computation of the derivatives of the dependent variables with respect to the independent ones.

ADIFOR 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 and employs a consistent subroutine-naming scheme that allows for code tuning, the exploitation of domain-specific knowledge, and the exploitation of vendor-supplied libraries.

ADIFOR employs a hybrid forward/reverse mode approach to generating derivatives. For each assignment statement, it generates code for computing the partial derivatives of the result with respect to the variables on the right-hand side using the reverse mode approach, and then employs the forward mode to propagate overall derivatives. For example, the statement

$$\mathtt{y} = \mathtt{x(1)} * \mathtt{x(2)} * \mathtt{x(3)} * \mathtt{x(4)} * \mathtt{x(5)}$$

gets transformed into the code shown in Figure 6. Note that none of the common subexpressions $x(i) * x(j)$ is recomputed in the reverse mode section.

ADIFOR-generated code can be used in various ways [6]: Instead of simply producing code to compute the Jacobian J, ADIFOR produces code to compute $J * S$, where the "seed matrix" S is initialized by the user. So if S is the identity, ADIFOR computes the full Jacobian, and if S is just a vector, ADIFOR computes the product of the Jacobian by a vector. The running time and storage requirements of the ADIFOR-generated code are roughly proportional to the number of columns of S, which equals the **gp** variable in the sample code above.

```
    r$1 = x(1) * x(2)
    r$2 = r$1 * x(3)
    r$3 = r$2 * x(4)
    r$4 = x(5) * x(4)
    r$5 = r$4 * x(3)
    r$1bar = r$5 * x(2)
    r$2bar = r$5 * x(1)
    r$3bar = r$4 * r$1
    r$4bar = x(5) * r$2
do g$i$ = 1, g$p$
   g$y(g$i$) = r$1bar * g$x(g$i$, 1)
              + r$2bar * g$x(g$i$, 2)
              + r$3bar * g$x(g$i$, 3)
              + r$4bar * g$x(g$i$, 4)
              + r$3 * g$x(g$i$, 5)
enddo
              y = r$3 * x(5)
```

Reverse Mode for computing

$$\frac{\partial \texttt{y}}{\partial \texttt{x(i)}}, i = 1, \ldots, 5$$

Forward Mode:
Assembling $\nabla\texttt{y}$ from $\nabla\texttt{x(i)}$, $i = 1, \ldots, 5$.

Computing function value

Figure 6: Sample of ADIFOR-generated code

3 Exploiting Program Structure and Parallelism

In this section we show how we can exploit high-level program structures and parallelism to speed derivative computations.

3.1 Computing Large, Sparse Jacobians

In the approximation of large, sparse Jacobians bu divided differences, one usually exploits the sparsity structure of these matrices by simultaneously perturbing several "unrelated" input parameters [11, 10]. This structure can also be exploited by a suitable choice of the "seed matrix."

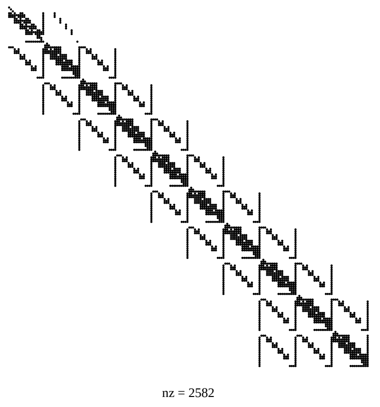
The idea is best understood with an example. Assume that we have a function

$$F = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix} : x \in \mathbf{R}^4 \mapsto y \in \mathbf{R}^5$$

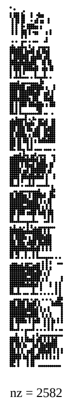
whose Jacobian J has the following structure (symbols denote nonzeros, and zeros are not shown):

$$J = \begin{pmatrix} \bigcirc & & & \\ \bigcirc & & & \Diamond \\ & \triangle & & \Diamond \\ & \triangle & \Box & \\ & \triangle & \Box & \end{pmatrix}.$$

So columns 1 and 2, as well as columns 3 and 4 are structurally orthogonal, and in divided-difference approximations one could exploit that by perturbing both x_1 and x_2 in one function



a) uncompressed



b) compressed

Figure 7: Sparsity structure of 190 × 190 blunt body problem Jacobian

evaluation, and both x_3 and x_4 in the other. We can exploit this fact by setting

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

For a more realistic example, the 190 × 190 Jacobian of the blunt body shock-tracking problem described in [38] has only 2582 nonzero entries; its structure is shown in Figure 7a. Because of its sparsity structure, it can be condensed into the "compressed Jacobian" shown in Figure 7b, which has only 28 columns.

The computation of Jacobian*vector products and compressed Jacobians requires much less time and storage than the generation of the full Jacobian matrix. For example, on the blunt-body problem we observe the performance shown in Table 1 on Sun Microsystems SPARC 2 and IBM RS/6000-550 workstations. The first line gives the run time of a sparse divided-difference approximation, based on the same coloring scheme as the "compressed Jacobian" approach in the second line. The third line shows the running time obtained if one treats this Jacobian as a dense one and ignores sparsity; this means that all derivative operations now are performed with vectors of length 190 instead of 28. As expected, performance suffers, although much less so on the IBM. This is due to the superscalar architecture of this chip and, we suspect, to efficient microcode implementations of multiplications by zero. A comprehensive study of this methodology applied to a collection of optimization model problems can be found in [2].

The compressed Jacobian does not tell the whole story, though. When we keep track of how many operations really have to be performed, avoiding additions and multiplications with zeros, we observe the behavior shown in Table 2. Here we denote the number of additions, multiplications, and the median number of nonzeros in derivative objects for the compressed Jacobian approach and a "sparse" implementation, which avoids operations with zeros. Thus, there is still ample scope for exploiting the temporal sparsity behavior of derivative propagation. This is not surprising since most of the time, the seed matrix is very sparse (e.g., the

	SPARC-2	RS/6000
Sparse DD	0.20	0.130
Compressed ADIFOR	0.13	0.025
Dense ADIFOR	0.85	0.056

Table 1: Performance of ADIFOR-generated derivative code (seconds)

	Adds	Mults	Median ∇ length
Compressed	134,428	185,948	28.0
Sparse	5,537	31,633	4.0

Table 2: Dense versus sparse derivative propagation

identity), and derivative objects fill in slowly as the computation proceeds. This effect is most noticeable in the computation of gradients of so-called partially separable functions [6, 8], a rather common class of functions first described by Griewank and Toint [27]. For example, on the 2-D Ginsburg-Landau Superconductivity problem [1], the sparse gradient computation on a 400×400 grid requires only 7 percent of the floating-point operations required for the dense version.

The dependency information collected and utilized in the sparse propagation of derivative objects also determines the sparsity pattern of the Jacobian, Hessian, and higher-derivative tensors. A sparse derivative implementation of forward differentiation is currently under development.

3.2 Stripmining Derivative Computations

Let us assume, for example, that we wish to compute a 17-element gradient $g = \frac{dF}{dx(1:17)}$. We can compute any subset of the entries of this gradient through proper choice of the "seed matrix." Hence, we can compute several sets of sensitivities one after the other, or we can decrease turnaround time by spawning several copies of the derivative code, as shown, for example, in Figure 8. Here e_i denotes the i-th canonical unit vector. Hence, the first incarnation of the derivative code will compute the first six entries of the gradient, the second one the next six, and the third one the remaining five.

This "stripmining" approach is simple, much like running several simulations in parallel for divided-difference approximations of derivatives, but has the advantage of decreasing wall clock time with minimal human effort. It can also easily be mapped on any collection of compute nodes, be it a MIMD parallel computer or a heterogeneous workstation network. We implemented this approach with the Fortran M system [16, 15], and the resulting "parallel wrapper" can easily be adapted to other codes.

The TLNS3D code [40] is a high-fidelity aerodynamic computer code that solves the time-dependent, 3-D, thin-layer Navier-Stokes equations with a finite-volume formulation. The code employs grid sequencing, multigrid, and local time stepping to accelerate convergence and efficiently obtain steady-state high Reynolds number turbulent flow solutions. Experiences with ADIFOR on TLNS3D are described in [4, 18].

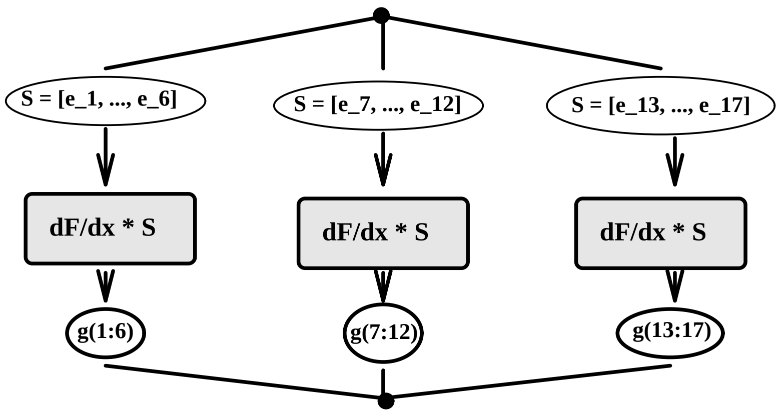


Figure 8: An Example of parallel sensitivity computation

# Processors	Average Time	Std. Deviation
1	34,160 (est.)	0
5	6,832	53.1
15	3,757	11.5

Table 3: Results of parallel sensitivity computation for TLNS3D

We were interested in computing the sensitivities of lift, drag, and pitching moment with respect to 60 shape design parameters on a 97x25x17 grid on a workstation platform. Because of memory limitations, we could only fit the code for the computation of up to 12 sensitivities on an IBM RS6000 workstation with 128 MBytes of memory. On the IBM SP1 parallel computer, which for the purposes of this exercise can be regarded as a network of workstations, we observed the performance shown in Table 3. If we had only one processor at our disposal, the memory limitation would limit us to running the 12-sensitivities job five times, requiring an estimated 34,160 seconds, or 9.5 hours. Employing 15 processors, the same set of sensitivities can now be obtained in just about one hour, a dramatic increase in turnaround time. Details are reported in [5].

4 Differentiation of Iterative Processes

Until recently it was not clear under what conditions automatic differentiation could be expected to yield useful derivative values if the evaluation program is not just a straight-line code but contains branches dependent on argument values. This situation occurs frequently in programs that employ iterative or adaptive numerical procedures, for example to approximate the solutions of nonlinear algebraic systems or differential equations. One can easily construct examples where branching makes the relation between independent and dependent variables nonsmooth so that the demand for derivative values is unrealistic in the strictly mathematical sense. Nevertheless, one can show under reasonable assumption that on piecewise smooth problems automatic differentiation yields one-sided derivatives, which may be quite useful and informative [32],[14]. More important, Gilbert has shown recently that in the case of contractive fixed-point iterations, the corresponding derivative values also converge to their correct values [17]. We have confirmed the validity of this result on a transsonic fluid dynamics code [4], where the previously used semi-analytic approximation to the derivative of the lift coefficient with respect to the Mach number turned out to be off by 50%. We have extended Gilbert's result to a wider class of iterative schemes, including Broyden's method, and other quasi-Newton schemes.

Our basic assumption is that a parameter dependent nonlinear system

$$F(v,x) \;=\; 0 \quad \text{with} \quad F \;: \mathbb{R}^n \times \mathbb{R}^p \mapsto \mathbb{R}^n$$

is, for fixed x, solved for $v(x)$ by an iteration of the form

$$v_{k+1} \;\equiv\; v_k \;-P_k F(v_k,x)\;, \tag{1}$$

where P_k is some $n \times n$ matrix, which we will consider as preconditioner. Total derivatives with respect to $x \in \mathbb{R}^p$ will be denoted by primes, and partial derivatives (with v kept constant) by the subscript x.

Simply differentiating (1), one finds that the derivatives of the iterates satisfy the recurrence

$$v'_{k+1} \;=\; v'_k - P_k\;[F_v(v_k,x)v'_k + F_x(v_k,x)] - P'_k\, F(v_k,x)\;. \tag{2}$$

Now the question arises under what assumptions this recurrence ensures convergence of the v'_k to the implicit derivative $v'(x)$ defined by the linear system

$$F_v(v(x),x)\,v'(x) \;=\; -F_x(v(x),x) \quad , \tag{3}$$

where $v(x)$ is the implicit function defined by $F(v(x),x) = 0$. In addition to the usual local regularity assumptions on the Jacobian F_v, we imposed the condition that the basic iteration is contractive in that $I - P_k F_v$ has a spectral norm bounded below 1 and furthermore that $P'_k\, F(v_k,x)$ converges to zero. This last condition is always met if the matrices P'_k are uniformly bounded, as can be expected in the case for Newton's method, where $P_k = F_v(v_k,x)^{-1}$ is continuously differentiable in x provided F is sufficiently smooth. Yet this example also shows that P'_k really should not matter at all since it is multiplied by the residual $F(v_k,x)$, which tends to zero as the iteration progresses. Moreover, in some cases P'_k may not even exist or grow unbounded because of nonsmooth adjustments of the preconditioner P_k from step to step. Then the convergence of the $v'_k(x)$ may be slowed down or prevented altogether. Therefore, it makes sense to *deactivate* the preconditioner P_k, that is, to suppress its dependence on x by simply setting $P'_k = 0$. The resulting simplified derivative recurrence

$$\tilde{v}'_{k+1} \;=\; \tilde{v}'_k - P_k\;[F_v(v_k,x)\;\tilde{v}'_k +\; F_x(v_k,x)] \tag{4}$$

is in general cheaper to perform and corresponds to the so-called incremental iterative schemes proposed in the context of CFD sensitivity analysis [35, 33].

We could show for a large variety of Newton-like methods that it still yields convergence of the $\tilde{v}'_k$ to $v'(x)$ at the same linear rate as the v'_k, which is determined by the spectral radius of $I - P_k F_v$. The only drawback is that the preconditioner P_k must be identified, which might not be so easy in a complex code. Moreover, multigrid schemes and other iterative solvers almost certainly do not satisfy our assumptions, and even if they did, this fact would be very hard to ascertain.

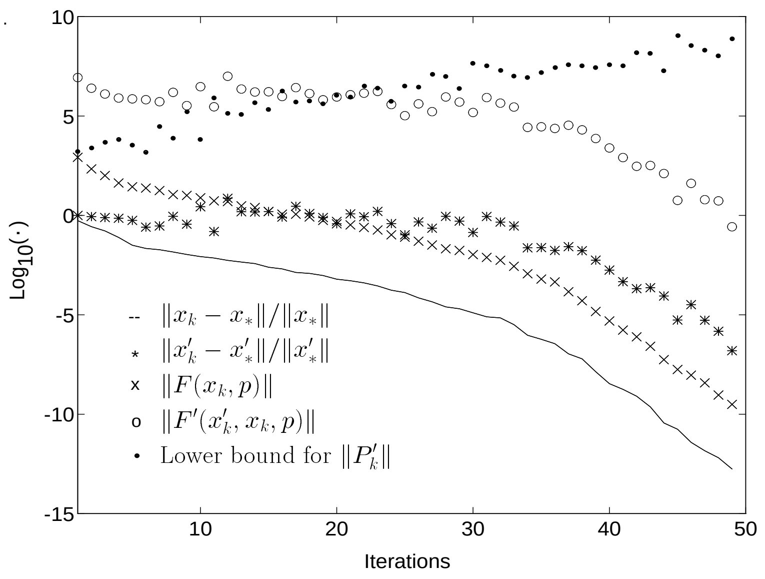


Figure 9: Fully differentiated Broyden's method for Jacobian calculation

Nevertheless, we have found so far that, for all practical solvers, derivatives of the iterates seem to converge to the correct implicit derivatives with or without deactivation of preconditioners. However, the derivatives tend to lag behind the iterates themselves, and whenever possible one should gauge their convergence by evaluating the derivatives' residual

$$F'(v_k, x) \quad = \quad [F_v(v_k, x)v'_k + F_x(v_k, x)]$$

which must vanish by the implicit function theorem when $v_k = v(x)$ and $v'_k = v'(x)$. A typical situation is depicted in Fig. 9 for a parameter estimation problem with three parameters x and a 40-dimensional state vector v. Details are reported in [23].

In Figure 9 it is particularly noteworthy that the P'_k appear to grow unbounded but that the derivative residuals $F'(v_k, x)$ and errors $v'_k - v'(x)$ converge to zero a bit later but essentially at the same rate as the iterates v_k themselves. This seems to be a typical situation.

5 New Approaches in Multidisciplinary Analysis and Design

The task of performing sensitivity analysis on the supersonic or transsonic flow over an elastic wing, for example, combines most of the difficulties that one may encounter in computational differentiation for the purposes of multidisciplinary design analysis and optimization.

1. The two disciplines aerodynamics and structures are mutually dependent, though the coupling is usually assumed to be sufficiently weak that block Gauss-Seydel solvers converge.

2. Both disciplines use their own grids, which usually do not coincide and must in any case be adapted as the calculation proceeds. This requires grid generation and interpolation procedures.

3. The nonlinear systems describing the equilibrium situation are so large and irregular that they cannot be solved directly, but one must employ instead iterative solvers, usually of the multigrid type.

4. The turbulence effects that occur even in subsonic situations must be approximated by algebraic functions, which are typically quite complex and not everywhere differentiable.

Before we discuss how computational differentiation can be made to cope with this kind of situation, let us note that other approaches are even less likely to succeed. Parts of some aerodynamic codes have been differentiated by hand, but since the turbulence models were generally considered too messy for hand manipulation, the results tended to be of very low accuracy. Also, since the outputs of realistic codes are strictly speaking not even continuous functions of the inputs, taking difference quotients is quite dangerous. As we have mentioned before it also has an unnecessarily high computational complexity.

5.1 Solving Coupled Systems

Symbolically we may write the structural and aerodynamic equilibrium conditions as the coupled algebraic system of equations:

$$E(u,v,w,x) \quad = \quad \begin{bmatrix} F(u,v,x) \\ G(u,v,w,x) \\ H(v,w,x) \end{bmatrix} = 0 \quad , \tag{5}$$

where the vectors v and x of linking and design variables may be assumed to have much smaller numbers of components than the vectors u and w representing displacements and flow velocities, respectively. Often the former represent functions in two spatial variables and the latter are discretizations of functions in three spatial variables. Therefore, the overall Jacobian will look roughly like

$$\frac{\partial E(u,v,w,x)}{\partial(u,v,w,x)} \quad = \quad \begin{bmatrix} F_u & F_v & 0 & F_x \\ G_u & G_v & G_w & G_x \\ 0 & H_v & H_w & H_x \end{bmatrix} \tag{6}$$

$$= \begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times & 0 & 0 & 0 & 0 & \times \\
\times & \times & \times & \times & \times & \times & \times & 0 & 0 & 0 & 0 & \times \\
\times & \times & \times & \times & \times & \times & \times & 0 & 0 & 0 & 0 & \times \\
\times & \times & \times & \times & \times & \times & \times & 0 & 0 & 0 & 0 & \times \\
\times & \times & \times & \times & \times & \times & \times & 0 & 0 & 0 & 0 & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times
\end{bmatrix} . \tag{7}$$

We may assume that each discipline has some iterative method for solving the internal system $F(u,v,x) = 0$ and $H(v,w,x) = 0$ for fixed v and x, which amounts to evaluating the implicit functions $u = u(v,x)$ and $w = w(v,x)$. If these methods can be differentiated, then, by the results described in Section 4, one obtains simultaneously approximations to

$$\left[\frac{\partial u}{\partial v}, \frac{\partial u}{\partial x}\right] \quad = \quad -F_u^{-1}[F_v, F_x] \tag{8}$$

and

$$\left[\frac{\partial w}{\partial v}, \frac{\partial w}{\partial x}\right] \quad = \quad -H_w^{-1}[H_v, H_x]. \tag{9}$$

Note that the inner disciplinary Jacobians F_u and H_w need not be formed explicitly even though their inverses appear in the explicit expression for $\frac{\partial u}{\partial v}$ and $\frac{\partial w}{\partial v}$.

The total derivative with respect to v of the reduced function

$$\hat{G}(v, x) \equiv G(u(v, x), v, w(v, x), x)$$

is given by

$$\frac{\partial \hat{G}}{\partial v} = G_u \frac{\partial u}{\partial v} + G_v + G_w \frac{\partial w}{\partial v} = G' \left[\left(\frac{\partial u}{\partial v}\right)^T, I, \left(\frac{\partial u}{\partial w}\right)^T, 0 \right]^T , \tag{10}$$

where G' is shorthand for $[G_u, G_v, G_w, G_x]$. Note that we need not compute G' explicitly to evaluate the derivative (10), but rather we can use the "seed matrix" mechanism to compute it by a forward sweep of automatic differentiation at roughly n_v times the cost of evaluating G by itself, where n_v represents the number of linking variables v. By using the resulting Jacobian $\hat{G}_v$ for a Newton-like method to directly solve $\hat{G}(v, x)$ for fixed x, one can directly apply an iterative method based on directional derivatives of $\hat{G}$ in the domain of v to solve the nonlinear equations (5). In either case the fact that the cross terms F_v, G_u, G_w, H_v are implicitly evaluated through automatic differentiation (rather than simply neglected and assumed to be small) ensures local convergence to a stationary solution, where F_u, H_w, and the overall Jacobian of E with respect to (u, v, w) are nonsingular.

5.2 Design Sensitivities for Coupled Systems

Now suppose we have an objective function $f(u, v, w, x)$ like the lift or the drag coefficient, in which case x incorporates not only geometric and structural parameters but also the free stream boundary conditions. Then the question arises how one can obtain approximations to the total gradient

$$\nabla_x f \quad = \quad f_u \frac{\partial u}{\partial x} + f_v \frac{\partial v}{\partial x} + f_w \frac{\partial w}{\partial x} + f_x \quad . \tag{11}$$

The derivatives $\partial u/\partial x$ and $\partial w/\partial x$ can be obtained by differentiating the individual discipline solvers with respect to x in addition to v. Then one obtains the derivative of $\hat{G}$ with respect to x

$$\frac{\partial \hat{G}}{\partial x} = G_u \frac{\partial u}{\partial x} + G_x + G_w \frac{\partial w}{\partial x} \quad , \tag{12}$$

where we have assumed v to be constant in the differentiation with respect to x. Assuming that $\frac{\partial \hat{G}}{\partial v}$ is small enough to be formed and factored explicitly, one may solve for the root $v = v(x)$ of $\hat{G}(v, x) = 0$ by Newton's method, say, and then evaluate according to the implicit function theorem

$$\frac{\partial v}{\partial x} \quad = \quad -\left(\frac{\partial \hat{G}}{\partial v}\right)^{-1} \frac{\partial \hat{G}}{\partial x} \quad .$$

Even if this is not possible and the iterates v_k are generated instead by an iteration of the form

$$v_{k+1} \quad = \quad v_k - P_k \hat{G}(v_k, x),$$

one may use the simplified derivative recurrence

$$v'_{k+1} = v'_k - P_k \left[\hat{G}_v v'_k + \hat{G}_x\right]$$

with $\hat{G}_v$ and $\hat{G}_x$ as defined above. Provided both levels of the iteration satisfy our contractivity assumption, we can expect that the v'_k will indeed converge to the correct value $\partial v/\partial x$, which can then be used in the expression (11) for the total gradient of the objective function.

6 Outlook

The emergence of robust automatic differentiation tools, together with the increased interest of the scientific community in sensitivity analysis and design optimization has provided a fertile climate for the advancement of computational differentiation techniques. However, despite the ubiquitous nature of derivatives in numerical modelling, computational differentiation techniques as we understand them are still only known or accessible to relatively few computational scientists.

Together with our colleagues in the field, we are working on changing this situation by developing robust, general, and portable tools that provide automatic differentiation technology in a fashion that scales with the problem as well as the computing environment, to support the full scientific development cycle. We are also working on developing templates to guide users in the development of codes embodying certain numerical paradigms to ensure that application of automatic differentiation leads to the desired results.

In the long run, we hope that computational differentiation techniques not only will provide computational scientists with a convenient and efficient means for computing the first- and second order derivatives, but will change the way scientists and algorithm developers view their problems. We expect that the fact that derivatives will be obtainable at a significantly reduced cost and of arbitrary order will spawn new algorithmic approaches as well as new problem-solving environments.

References

[1] Brett Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Report ANL/MCS–TM–150, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[2] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. Technical Report MCS–P348–0193, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[3] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[4] Christian Bischof, George Corliss, Larry Green, Andreas Griewank, Kara Haigler, and Perry Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Journal on Computing Systems in Engineering*, 3(6):625–638, 1992.

[5] Christian Bischof, Larry Green, Kitty Haigler, and Tim Knauff. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. Extended Abstract, submitted to the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, 1994.

[6] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. ADIFOR Working Note #2, MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[7] Christian H. Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Getting started with ADIFOR. ADIFOR Working Note #9, ANL–MCS–TM-164, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[8] Christian H. Bischof and Moe El-Khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. ADIFOR Working Note #7, ANL–MCS–TM-163, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[9] Stephen L. Campbell, Edward Moore, and Yangchun Zhong. Utilization of automatic differentiation in control algorithms. To appear in IEEE Trans. Automatic Control, 1993.

[10] T. F. Coleman, B. S. Garbow, and J. J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Software*, 10:329 – 345, 1984.

[11] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187 – 209, 1984.

[12] George Corliss and Andreas Griewank. Operator overloading as an enabling technology for automatic differentiation. Technical Report MCS-P358-0493, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[13] Lawrence C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114 – 125. SIAM, Philadelphia, Penn., 1991.

[14] Hans-C. Fischer. Differentiation arithmetic and applications in Pascal-XSC. Poster presented at SIAM Workshop on Automatic Differentiation of Algorithms, Breckenridge, Colo., January 1991.

[15] Ian Foster, Robert Olson, and Steven Tuecke. Programming in Fortran M. Technical Report ANL–93/26, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, October 1993.

[16] Ian T. Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS–P327–0992, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[17] Jean-Charles Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–22, 1992.

[18] Lawrence Green, Perry Newman, and Kara Haigler. Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation. In *Proceedings of the 11th AIAA Computational Fluid Dynamics Conference, AIAA Paper 93-3321*. American Institute of Aeronautics and Astronautics, 1993.

[19] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[20] Andreas Griewank. Direct calculation of Newton steps without accumulating Jacobians. In T. F. Coleman and Y. Li, editors, *Large-Scale Numerical Optimization*, pages 115–137, Philadelphia, Pa., 1990. SIAM.

[21] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods & Software*, 1(1):35–54, 1992.

[22] Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. Technical Report MCS-P355-0393, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[23] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence of iterative equation solvers. Technical Report MCS-P333-1192, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[24] Andreas Griewank and George Corliss. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, Penn., 1991.

[25] Andreas Griewank, David Juedes, and Jay Srinivasan. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Technical Report MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, 1990.

[26] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126 – 135. SIAM, Philadelphia, Penn., 1991.

[27] Andreas Griewank and Philippe L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137, 1982.

[28] Uli Häußermann. Automatische Differentiation zur Rekursiven Bestimmung von Partiellen Ableitungen. STUD-102, Institut B für Mechanik, Universität Stuttgart, 1993.

[29] Kieran Herley. On the NP-completeness of optimum accumulation by vertex elimination. Unpublished manuscript, 1993.

[30] Jim E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 243 – 250. SIAM, Philadelphia, Penn., 1991.

[31] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Philadelphia, 1991. SIAM.

[32] G. Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150 – 165, June 1980.

[33] V. M. Korivi, A. C. Taylor, P. A. Newman, G. W. Hou, and H. E. Jones. An incremental strategy for calculating consistent discrete CFD sensitivity derivatives. NASA Technical Memorandum 104207, NASA Langley Research Center, February 1992.

[34] Koichi Kubota. PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 251 – 262. SIAM, Philadelphia, Penn., 1991.

[35] P. A. Newman, G. J.-W. Hou, H. E. Jones, A. C. Taylor, and V. M. Korivi. Observations on computational methodologies for use in large-scale, gradient-based, multidisciplinary design incorporating advanced CFD codes. NASA Technical Memorandum 104206, NASA Langley Research Center, 1992.

[36] G. M. Ostrovskii, Ju. M. Wolin, and W. W. Borisov. Über die Berechnung von Ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382 – 384, 1971.

[37] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[38] G. R. Shubin, A. B. Stephens, H. M. Glaz, A. B. Wardlaw, and L. B. Hackerman. Steady shock tracking, Newton's method, and the supersonic blunt body problem. *SIAM J. on Sci. and Stat. Computing*, 3(2):127–144, June 1982.

[39] Edgar Soulie. User's experience with Fortran compilers for least squares problems. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 297–306. SIAM, Philadelphia, Penn., 1991.

[40] V. N. Vatsa and B. W. Wedan. Development of a multigrid code for 3-D Navier-Stokes equations and its application to a grid-refinement study. *Computers & Fluids*, 18(4):391–403, 1990.

[41] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.