ADIFOR: Automatic Differentiation in a Source Translator Environment

Christian Bischof, Argonne National Laboratory,
Alan Carle, Rice University,
George Corliss, Argonne National Laboratory, and
Andreas Griewank, Argonne National Laboratory

Abstract The numerical methods employed in the solution of many scientific computing problems require the computation of derivatives of a function $f : \mathbf{R}^n \to \mathbf{R}^m$. ADIFOR (Automatic Differentiation In FORtran) is a source transformation tool that accepts Fortran 77 code for the computation of a function and writes portable Fortran 77 code for the computation of the derivatives. In contrast to previous approaches, ADIFOR views automatic differentiation as a source transformation problem and employs the data analysis capabilities of the ParaScope Fortran programming environment. Experimental results show that ADIFOR can handle real-life codes and that ADIFOR-generated codes are competitive with divided-difference approximations of derivatives. In addition, studies suggest that the source-transformation approach to automatic differentation may improve the time required to compute derivatives by orders of magnitude.

Keywords. Derivative, gradient, Jacobian, automatic differentiation, chain rule, ParaScope Parallel Programming Environment, source transformation and optimization.

1 Introduction

The methods employed for the solution of many scientific computing problems require the evaluation of derivatives of some function. Probably the best known are gradient methods for optimization [11], Newton's method for the solution of nonlinear systems [9, 11], and the numerical solution of stiff ordinary differential equations [6, 10]. The function f to be differentiated is usually represented in the form of a computer program, not in a closed form as a single expression.

For purposes of illustration, we assume that $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}$ and that we wish to compute the derivatives of y with respect to x. We call x the *independent variable* and y the *dependent variable*. There are four approaches to computing derivatives (these issues are discussed in more detail in [14]):

Hand-Coded: Computing derivatives by hand is difficult and error-prone, especially as the problem complexity increases.

Divided Differences: The derivative of f with respect to the ith component of x at a particular point x_0 is approximated by either *one-sided differences*

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x=x_0} \approx \frac{f(x_0 + h * e_i) - f(x_0)}{h}$$

or *central differences*

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x=x_0} \approx \frac{f(x_0 + h * e_i) - f(x_0 - h * e_i)}{2h}.$$

Here e_i is the ith Cartesian basis vector. Computing derivatives by divided differences has the advantage that we need only the function as a "black box." The principle disadvantage of divided differences is that their accuracy is hard to assess. A small step size, h, is needed for properly approximating derivatives, yet may lead to numerical cancellation and the loss of many digits of accuracy. In addition, different scales of the x_i's may require different step sizes for the various parameters.

Symbolic Differentiation: Given a string describing the definition of a function, symbolic manipulation packages such as Maple, Reduce, Macsyma, or Mathematica provide exact derivatives, expressing the derivatives in terms of the intermediate variables. For example, if

$$f(x) = x(1) * x(2) * x(3) * x(4) * x(5),$$

we obtain

$$\nabla f(x) = \begin{pmatrix} x(2)*x(3)*x(4)*x(5) \\ x(1)*x(3)*x(4)*x(5) \\ x(1)*x(2)*x(4)*x(5) \\ x(1)*x(2)*x(3)*x(5) \\ x(1)*x(2)*x(3)*x(4) \end{pmatrix}.$$

The evaluation of ∇f in this form is very inefficient, so modern symbolic processors devote considerable attention to efficient evaluations of common subexpressions. Branches and loops in the code defining f present formidable challenges, which are being addressed by developers of symbolic systems on a basis similar to that presented here. The principle disadvantage of symbolic processors is that they may run into resource limitations when the function description is sufficiently long and complicated.

Automatic Differentiation: Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos. By applying the chain rule

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right) \tag{1}$$

over and over again to the composition of those elementary operations, one can compute derivative information of f exactly and in a completely mechanical fashion. ADIFOR uses this approach to transform Fortran 77 programs. For example, if we have a program for computing $f = \prod_{i=1}^{5} x(i)$

```
subroutine prod5(x,f)
real x(5), f
f = x(1) * x(2) * x(3) * x(4) * x(5)
return
end
```

ADIFOR produces a program whose computational section is shown in Figure 1.

```
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$f(g$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)
end do
f = r$3 * x(5)
```

Figure 1. ADIFOR-generated code

The $ sign is used to emphasize ADIFOR-generated variables. To improve readability, we deleted continuation line characters. If the variable `x` is initialized to the desired value x_0, `g$p` to 5, and the array `g$x` to the 5×5 identity matrix, then on exit the vector `g$y` contains $\frac{\partial f(x)}{\partial x}|_{x=x_0}$. No redundant subexpressions are computed here, since the overall product is computed in a binary-tree fashion, and the proper pieces of the product are reused in the derivative computation.

In the next section, we shall give a brief introduction to automatic differentiation. Section 3 describes the overall use of the forward mode of automatic differentiation, while employing the reverse mode for efficiency within assignment statements. Section 4 describes how ADIFOR provides this functionality in the context of a source transformation environment, and gives the rationale for choosing such an approach. In Section 5, we present some experimental results which show that the run time required for ADIFOR-generated exact derivative codes compares favorably with divided-difference derivative approximations. In Section 6, we outline ongoing work and present evidence that the source-transformation approach to automatic differentiation may reduce the time to compute derivatives by orders of magnitude.

2 Automatic Differentiation

Automatic differentiation takes advantage of the fact that the source code also contains information about derivatives of the function. ADIFOR (Automatic Differentiation In FORtran) [3] augments the original source code with additional statements that propagate values of derivative objects in addition to the values of the variables computed in the original code. Given a Fortran subroutine (or a collection of subroutines) for a function f, ADIFOR produces Fortran 77 subroutines for the computation of the derivatives of f.

We illustrate automatic differentiation with an example. Assume that we have the sample program shown in Figure 2 for the computation of a function $f : \mathbf{R}^2 \mapsto \mathbf{R}^2$. Here, the vector `x` contains the independent variables, and the vector `y` contains the dependent variables. The function described by this program is defined except at $\mathtt{x}(2) = 0$ and is differentiable except at $\mathtt{x}(1) = 2$.

```
if x(1) > 2 then
   a = x(1) + x(2)
else
   a = x(1) * x(2)
endif
do i = 1, 2
   a = a * x(i)
end do
y(1) = a / x(2)
y(2) = sin (x(2))
```

Figure 2. Sample program for a function $f : \mathrm{x} \mapsto \mathrm{y}$

We can transform this program into one for computing derivatives by associating a derivative object $\nabla\mathtt{t}$ with every variable $\mathtt{t}$. Assume that $\nabla\mathtt{t}$ contains the derivatives of $\mathtt{t}$ with respect to the independent variables $\mathbf{x}$,

$$\nabla\mathtt{t} = \begin{pmatrix} \frac{\partial\,\mathtt{t}}{\partial\,\mathbf{x}(1)} \\ \frac{\partial\,\mathtt{t}}{\partial\,\mathbf{x}(2)} \end{pmatrix}.$$

We can propagate these derivatives by using elementary differentiation arithmetic based on the chain rule [14, 20] for computing the derivatives of `y(1)` and `y(2)`, as shown in Figure 3. In this example, each assignment to a derivative is actually a vector assignment of length 2.

```
if x(1) > 2.0 then
   a = x(1) + x(2)
   ∇a = ∇x(1) + ∇x(2)
else
   a = x(1) * x(2)
   ∇a = x(2) * ∇x(1) + x(1) * ∇x(2)
endif
do i = 1, 2
   temp = a
   a = a * x(i)
   ∇a = x(i) * ∇a + temp * ∇x(i)
end do
y(1) = a / x(2)
∇y(1) = 1.0 / x(2) * ∇a
        - a / (x(2) * x(2)) * ∇x(2)
y(2) = sin (x(2))
∇y(2) = cos (x(2)) * ∇x(2)
```

Figure 3. Sample program of Figure 2 augmented with derivative code

This mode of automatic differentiation, where we maintain the derivatives with respect to the independent variables, is called the *forward mode* of automatic differentiation. The *reverse mode* of automatic differentiation maintains the derivative of the final result with respect to an intermediate quantity. These quantities, usually referred to as *adjoints,* measure the sensitivity of the final result with respect to some intermediate quantity.

The reverse mode requires fewer operations than the forward mode if the number of independent variables is larger than the number of dependent variables. This is exactly the case for computing a gradient, which can be viewed as a Jacobian matrix with only one row. This issue is discussed in more detail in [14, 16, 17].

Wolfe observed [23], and Baur and Strassen confirmed [2], that if care is taken in handling quantities which are common to the (rational) function and its derivatives, then the cost of evaluating a gradient with n components is a small multiple of the cost of evaluating the underlying scalar function. Despite the advantages of the reverse mode from the viewpoint of complexity, the implementation for the general case is quite complicated. It requires the ability to access *in reverse order* the instructions performed for the computation of f and the values of their operands and results. Current tools (see [18]) achieve this by storing a record of every computation performed. An interpreter performs a backward pass on this "tape." The resulting overhead often dominates the complexity advantage of the reverse mode in an actual implementation (see [12, 13]).

We also note that even though we showed the computation only of first derivatives, the automatic differentiation approach can easily be generalized to the computation of univariate Taylor series or Hessians and multivariate higher-order derivatives [8, 15, 20].

This discussion is intended to demonstrate that the principles underlying automatic differentiation are not complicated: We just associate extra computations (which are entirely specified on a statement-by-statement basis) with the statements executed in the original code. As a result, a variety of implementations of automatic differentiation have been developed over the years (see [18] for a survey).

3 A Hybrid Approach

For efficiency in ADIFOR, we have adopted a hybrid approach to computing derivatives that is generally based on the forward mode, but uses the reverse mode to compute the gradients of assignment statements containing complex expressions. The hybrid mode is effective because assignment statements often compute a single dependent variable given the values of multiple independent variables, an ideal case for the reverse mode. For this restricted case, the reverse mode code can be implemented entirely as inline code, thereby avoiding potentially recursive programming implied by the Baur and Strassen proof [2].

A simple example will illustrate the advantages of the hybrid mode. Consider the statement

$$\mathtt{w} = -\mathtt{y}/(\mathtt{z} * \mathtt{z} * \mathtt{z}),$$

where y and z depend on the independent variables. We have already computed ∇y and ∇z and now wish to compute ∇w. By breaking up this compound statement into unary and binary statements and applying the chain rule to each statement, we get the forward mode code shown in Figure 4.

There is another way, though. The chain rule tells us that

$$\nabla \mathtt{w} = \frac{\partial\, \mathtt{w}}{\partial\, \mathtt{y}} * \nabla \mathtt{y} + \frac{\partial\, \mathtt{w}}{\partial\, \mathtt{z}} * \nabla \mathtt{z}.$$

Hence, if we know the "local" derivatives $(\frac{\partial\, w}{\partial\, y}, \frac{\partial\, w}{\partial\, z})$ of w with respect to z and y, we can easily compute ∇w, the derivatives of w with respect to x. The local derivatives $(\frac{\partial\, w}{\partial\, y}, \frac{\partial\, w}{\partial\, z})$ can be computed efficiently by using the reverse mode of automatic differentiation. In the reverse mode, let tbar denote the adjoint object corresponding to t. The goal is for tbar to contain the derivative $\frac{\partial\, w}{\partial\, t}$. We know that wbar $= \frac{\partial\, \mathtt{W}}{\partial\, \mathtt{W}} = 1.0$. We can compute ybar and zbar by applying the following simple rule to the statements executed in computing w, but in reverse order:

if $s = f(t)$,	then	tbar += sbar * (df/dt)
if $s = f(t, u)$,	then	tbar += sbar * (df/dt)
		ubar += sbar * (df/du)

Using this recipe (and some simple optimizations), we generate the reverse mode code shown in Figure 4.

Forward Mode:

```
t1 = - y
∇ t1 = - ∇ y
t2 = z * z
∇ t2 = ∇ z * z + z * ∇ z
t3 = t2 * z
∇ t3 = ∇ t2 * z + t2 * ∇ z
w = t1 / t3
∇ w  = (∇ t1 - ∇ t3 * w) / t3
```

Reverse Mode:

```
t1 = - y
t2 = z * z
t3 = t2 * z
w = t1 / t3
t1bar = (1 / t3)
t3bar = (- t1 / t3)
t2bar = t3bar * z
zbar  = t3bar * t2
zbar  = zbar + t2bar * z
zbar  = zbar + t2bar * z
ybar  = - t1bar
∇ w = ybar * ∇ y + zbar * ∇ z
```

Figure 4. Forward versus reverse mode in computing derivatives of `w = -y/(z*z*z)`

The forward mode code in Figure 4 requires space for three auxiliary gradient vectors and contains four vector assignments. In contrast, the reverse mode code requires space for five scalar auxiliary adjoint objects and has only one vector assignment.

4 ADIFOR Design: Principles and Advantages

ADIFOR has been developed within the context of the ParaScope Parallel Programming Environment [7], which combines dependence analysis with interprocedural analysis to support ambitious interprocedural code optimization and semi-automatic parallelization of Fortran programs. While our primary goal is not code optimization or parallelization of Fortran programs, ParaScope provides us with a Fortran parser, data abstractions for representing Fortran programs and sophisticated facts derived from Fortran programs, and tools for constructing and manipulating those representations.

In particular, ParaScope tools compute data flow information, dependence graphs, control flow graphs, and a call graph. The data-dependence analysis capabilities are critical for determining which variables need to have derivative objects associated with them, a process we call *variable nomination.* Only those variables z whose values depend on an independent variable x and influence a dependent variable y need to have derivative information associated with them.

Another advantage of basing ADIFOR within a sophisticated code optimization framework is that mechanisms are already in place for simplifying the derivative code that we generate by application of the statement-by-statement hybrid mode translation rules. By applying constant folding and forward substitution, we eliminate multiplications by 1.0 and additions of 0.0, and we reduce the number of variables that must be allocated to hold derivative values [1].

In summary, ADIFOR proceeds as follows:

1. The user specifies the subroutine that corresponds to the "function" for which he wishes derivatives, as well as the variable names that correspond to dependent and independent variables. These names can be subroutine parameters or variables in common blocks. In addition to the source code for the "function" subroutine, the user must submit the source code for all subroutines that are directly or indirectly called from this subroutine.

2. ADIFOR parses the code, builds the call graph, collects intraprocedural and interprocedural dependency information, and determines active variables.

3. ADIFOR allocates derivative objects.

4. The original source code is augmented with derivative statements. The forward mode is used overall, while the reverse mode is used for assignment.

5. The augmented code is optimized, eliminating unnecessary arithmetic operations and temporary variables.

The resulting code generated by ADIFOR can be called by user programs in a flexible manner to be used in conjunction with standard software tools for optimization, solving nonlinear equations, or for stiff ordinary differential equations. A discussion of calling the ADIFOR-generated code from users' programs in included in [4].

The ease of use of ADIFOR follows from its basis in a sophisticated compilation environment. In many applications, the "function" whose derivatives we wish to compute is a collection of subroutines, and all that is expected of the user is to specify which of the variables correspond to the independent and dependent variables. In addition, the code generated by automatic differentiation is easy to transport between different machines. ADIFOR takes those requirements into account. Its user interface is simple, and the ADIFOR-generated code is efficient and portable. In comparison with other implementations of automatic differentiation (see [18] for a survey), ADIFOR provides the following features:

Portability: ADIFOR produces vanilla Fortran 77 code. ADIFOR-generated derivative code requires no run-time support and can easily be ported between different computing environments.

Generality: ADIFOR supports almost all of Fortran 77, including nested subroutines, common blocks, and equivalences.

Efficiency: ADIFOR-generated derivative code is competitive with codes that compute the derivatives by divided differences. In most applications we have run, the ADIFOR-generated code is faster than the divided-difference code.

Preservation of Software Development Effort: The code produced by ADIFOR respects the data flow structure of the original program. That is, if the user invested the effort to develop code that vectorizes and parallelizes well, then the ADIFOR-generated derivative code also vectorizes and parallelizes well. In fact, the derivative code offers more scope for vectorization and parallelization.

Extensibility: ADIFOR employs a consistent subroutine-naming scheme that allows the user to supply his own derivative routines. In this fashion, the user can exploit domain-specific knowledge, utilize vendor-supplied libraries, and minimize computational bottlenecks.

Ease of Use: ADIFOR requires the user to supply the Fortran source code for the subroutine representing the function to be differentiated and for all lower-level subroutines. The user then selects the variables (in either parameter lists or common blocks) that correspond to the independent and dependent variables. ADIFOR then determines which other variables throughout the program require derivative information. A detailed description of the use of ADIFOR-generated code appears in [4].

Intuitive Interface: An X-windows interface for ADIFOR (called xadifor) makes it easy for the user to create the ASCII script file that ADIFOR reads. This functional division makes it easy both to set up the problem and to rerun ADIFOR if changes in the code for the target function require a new translation.

Using ADIFOR, one then need not worry about the accurate and efficient computation of derivatives, even for complicated functions. As a result, the computational scientist can concentrate on the more important issues of algorithm design or system modeling.

5 Experimental Results

In this section, we report on the execution time of ADIFOR-generated derivative codes in comparison with divided-difference approximations of first derivatives on some larger codes. While the ADIFOR system runs on a SPARC platform, the ADIFOR-generated derivative codes are portable and can run on any computer that has a Fortran 77 compiler.

The "heart" problem was given to us by Janet Rogers, National Institute of Standards and Technology in Boulder, Colorado. The code submitted to ADIFOR computes elementary Jacobian matrices which are then assembled to a large sparse Jacobian matrix used in an orthogonal-distance regression fit [5]. The code named "adiabatic" is from Larry Biegler, Chemical Engineering Department, Carnegie-Mellon University, and implements adiabatic flow, a common module in chemical engineering [22]. The code named "reactor" was given to us by Hussein Khalil, Reactor Analysis and Safety Division, Argonne National Laboratory. While the other codes were used in an optimization setting, the derivatives of the "reactor" code are used for sensitivity analysis to ensure that the model is robust with respect to certain key parameters. Finally, the code

Table 1. Performance of ADIFOR-generated derivative codes compared to divided-difference approximations for a single Jacobian evaluation

Problem Name	Jacobian Size	Code Size (lines)	Div. Diff. Run time (seconds)	ADIFOR Run time (seconds)	ADIFOR Improvement	Machine
Heart	1×8	1305	11641.1	13941.30	-20%	SPARC 1
Adiabatic	6×6	1089	0.54	0.18	67%	SPARC 1
Reactor	3×29	1455	42.34	36.14	15%	SPARC 4/490
Reactor	3×29	1455	13.34	8.33	38%	RS6000/500
Shock	190×190	1403	0.041	0.023	44%	RS6000/500
Shock	190×190	1403	0.46	0.31	33%	SPARC 1

named "shock" was given to us by Greg Shubin, Boeing Computer Services, Seattle, Washington. This code implements the steady shock tracking method for the axisymmetric blunt body problem [21]. The Jacobian has a banded structure. The "normal" Jacobian has 190 columns, although the Jacobian compression techniques outlined in [4] requires only 28 columns.

Table 1 summarizes the time required by the ADIFOR-generated derivative codes with respect to divided differences. These tests were run on a SPARCstation 1, a SPARC 4/490, or an IBM RS6000/550.

Different machines are cited because of the different sources of the codes being run. The column of the table labeled "ADIFOR Improvement" indicates the percentage improvement of the running time of the ADIFOR-generated derivative code over an approximation of the divided-difference running times. This column contains the machine-independent comparison data. For the "shock" code, we had a derivative code based on sparse divided differences supplied to us. In the other cases, we estimated the time for divided differences by multiplying the time for one function evaluation by the number of independent variables. This approach is conservative, yet typical in an optimization setting, where the function value already has been computed for other purposes. An improvement greater than 0% indicates that the ADIFOR-generated derivatives ran faster than divided differences.

We see that already in its current version, ADIFOR performs well in competition with divided-difference approximations. For all codes that we processed, ADIFOR-generated code is up to a factor of three faster, and never worse by more than a factor of 1.82. This improvement in the speed of derivative computations is obtained without the user having to make any modifications to the code. We also see that ADIFOR can handle problems where symbolic techniques would be almost certain to fail, such as the "shock" or "reactor" codes.

We conclude that ADIFOR-generated derivatives are more than suitable as a substitute for handcoded or divided-difference derivatives. Virtually no time investment is required by the user to generate the codes. In most codes, ADIFOR-generated codes outperform divided-difference derivative approximations. In addition, the fact that ADIFOR computes *exact* derivatives (up to machine precision) may significantly increase the robustness of optimization codes or ODE solvers, where good derivative values are critical for the convergence of the numerical scheme.

6 Future Work

We are planning many enhancements to improve the performance of ADIFOR-generated code. The most important seems to be the increased use of the reverse mode for better performance. The reverse mode requires us to reverse the computation from a trace of at least part of the computation, which we later interpret. If we can accomplish the code reversal at compile time, we can truly exploit the reverse mode, since we shall not incur the overhead that is associated with run-time tracing.

ADIFOR currently does a compile-time reversal of composite right-hand sides of assignment statements, but there are other syntactic structures such as parallel loops for which this could be performed at compile time. In a parallel loop, there are no dependencies between different iterations. Thus, in order to generate code for the reverse mode, it is sufficient to reverse the computation inside the loop body. This can easily be done if the loop body is a basic block. The potential of this technique is impressive. Hand-compiling reverse mode code for the loop bodies of the torsion problem, a problem in the MINPACK-2 test set collection [19], we obtained the performance shown in Figure 5. This figure shows the ratio of gradient/function evaluation on a Solbourne SE/900 for the current ADIFOR version and

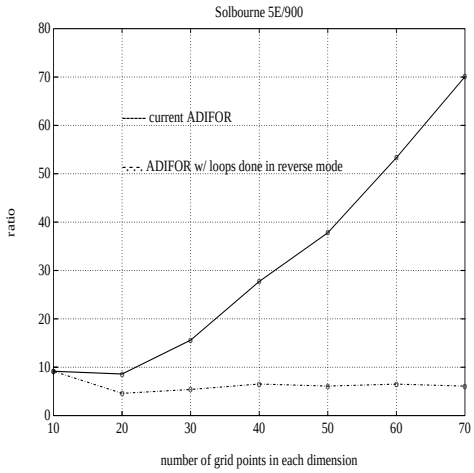

Figure 5: Ratio of gradient/function evaluation

for a hand-modified ADIFOR code that uses the reverse mode for the bodies of parallel loops. The gradients are of size $nint * nint$, where $nint$ is the number of grid points in each dimension.

Approximation of the gradient by divided differences costs $nint * nint$ function evaluations. Hence, we see that

- the current ADIFOR is faster than divided-difference approximations by a factor of 70 on a problem of size 4900; and
- using the reverse mode for loop bodies, we can compute the gradient in about six to seven times the cost of a function evaluation, independent of the size of the problem.

Taken together, these points mean that for the problem of size 4900, we can improve the speed of derivative computation by over two orders of magnitude compared to divided-difference computations.

We also plan to develop a better understanding of the techniques that are used for symbolic approaches for computing derivatives, especially with respect to reasoning about mathematical identities. For example, if presented with a statement like `x = sin(y)**2 + cos(y)**2`, ADIFOR will dutifully apply the chain rule, while the mathematical reasoning built into a symbolic system might recognize this identity and simplify it. The chain-rule based automatic differentiation approach underlying ADIFOR is a perfect overall framework for the computation of derivatives since it is more or less insensitive to the overall size of the code. On the other hand, symbolic techniques, whose execution time depends significantly on the size of the problem presented, fit in well as simplification or optimization techniques at the statement or basic-block level. We intend to explore this issue further.

Acknowledgement

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W–31–109–Eng–38; through NSF Cooperative Agreement No. CCR–8809615; and by the W. M. Keck Foundation.

References

[1] Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., second edition, 1986.

[2] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317 – 330, 1983.

[3] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Generating derivative codes from Fortran programs. Preprint MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. 60439, 1991. Also appeared as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Tex. 77251.

[4] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Memorandum ANL/MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. 60439, October 1991. ADIFOR Working Note # 2.

[5] Paul T. Boggs and Janet E. Rogers. Orthogonal distance regression. *Contemporary Mathematics*, 112:183 – 193, 1990.

[6] J. C. Butcher. Implicit Runge-Kutta processes. *Math. Comp.*, 18:50 – 64, 1964.

[7] D. Callahan, K. Cooper, R. T. Hood, Ken Kennedy, and Linda M. Torczon. ParaScope: a parallel programming environment. *International Journal of Supercomputer Applications*, 2(4), December 1988.

[8] Bruce D. Christianson. Automatic Hessians by reverse accumulation. Technical Report NOC TR228, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., April 1990.

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

[10] G. Dahlquist. A special stability problem for linear multistep methods. *BIT*, 3:27 – 43, 1963.

[11] John Dennis and R. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, N.J., 1983.

[12] Lawrence C. W. Dixon. Automatic differentiation and parallel processing in optimisation. Technical Report No. 180, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., 1987.

[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] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83 – 108. Kluwer Academic Publishers, 1989.

[15] Andreas Griewank. Automatic evaluation of first- and higher-derivative vectors. In R. Seydel, F. W. Schneider, T. Küpper, and H. Troger, editors, *Proceedings of the Conference at Würzburg, Aug. 1990, Bifurcation and Chaos: Analysis, Algorithms, Applications*, volume 97, pages 135 – 148. Birkhäuser Verlag, Basel, Switzerland, 1991.

[16] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, to appear. Also appeared as Preprint MCS–P228–0491, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, Ill. 60439, 1991.

[17] Andreas Griewank, David Juedes, Jay Srinivasan, and Charles Tyner. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, to appear.

[18] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315 – 329. SIAM, Philadelphia, Penn., 1991.

[19] Jorge J. Moré. On the performance of algorithms for large-scale bound constrained problems. In T. F. Coleman and Y. Li, editors, *Large-Scale Numerical Optimization*, pages 32 – 45. SIAM, 1991.

[20] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[21] 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.

[22] J. M. Smith and H. C. Van Ness. *Introduction to Chemical Engineering*. McGraw-Hill, New York, 1975.

[23] Philip Wolfe. Checking the calculation of gradients. *ACM Trans. Math. Softw.*, 6(4):337 – 343, 1982.