ADIFOR Working Note #4:

ADIFOR: Fortran Source Translation for Efficient Derivatives*

Christian Bischof†
Alan Carle‡
George Corliss†
Andreas Griewank†
Paul Hovland†

Argonne Preprint MCS–P278–1291

Abstract. The numerical methods employed in the solution of many scientific computing problems require the computation of derivatives of a function $f : \mathbf{R}^n \to \mathbf{R}^m$. Both the accuracy and the computational requirements of the derivative computation are usually of critical importance for the robustness and speed of the numerical method. ADIFOR (Automatic Differentiation In FORtran) is a source translation tool implemented by using the data abstractions and program analysis capabilities of the ParaScope Parallel Programming Environment. ADIFOR accepts arbitrary Fortran 77 code defining the computation of a function and writes portable Fortran 77 code for the computation of its derivatives. In contrast to previous approaches, ADIFOR views automatic differentiation as a process of source translation that exploits computational context to reduce the cost of derivative computations. Experimental results show that ADIFOR can handle real-life codes, providing exact derivatives with a running time that is competitive with the standard divided-difference approximations of derivatives and that may perform orders of magnitude faster than divided-differences in certain cases. The computational scientist using ADIFOR is freed from worrying about the accurate and efficient computation of derivatives, even for complicated "functions" and hence is able to concentrate on the more important issues of algorithm design or system modeling.

Key words. Large-scale problems, derivative, gradient, Jacobian, automatic differentiation, optimization, stiff ordinary differential equations, chain rule, parallel, ParaScope Parallel Programming Environment, source transformation and optimization.

1 Automatic Differentiation

The methods employed for the solution of many scientific computing problems require the evaluation of derivatives of some function f that is usually represented as a computer program, not in closed form. Probably best known are gradient methods for optimization [13], Newton's method for the solution of nonlinear systems [13], and the numerical solution of stiff ordinary differential equations [8]. These methods are examples of a large class of methods for numerical computation, where the computation of derivatives is a crucial ingredient in the computation of a numerical solution.

A conventional compiler extracts from the Fortran source code for computing a function a sequence of unary and binary operations and elementary functions and decisions that can be executed to compute the function values. More sophisticated compilers extract from the source code information that allows some of the computations to be executed efficiently on vector or parallel computers. Stetter has observed that in many applications, high-quality scientific computing requires the extraction of more mathematical information than just the function values [34]. For example, Neumaier [27] listed 15 mathematical properties (including derivative values, Lipschitz constants, enclosures, and asymptotic expansions) that might be propagated along with the values of the variable.

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

*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.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439.
‡Center for Research on Parallel Computation, Rice University, P. O. Box 1892, Houston, TX 77251.

```
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 1: Sample program for a function f : x $\mapsto$ y

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.

For discussion, 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.* While the terms "dependent", "independent", and "variable" are used in many different contexts, this terminology corresponds to the mathematical use of derivatives. There are four approaches to computing derivatives [16]:

By Hand: As the problem complexity increases, this approach becomes increasingly difficult and error-prone.

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* or *central differences.* Computing derivatives by divided differences has the advantage that we treat the function as a "black box." The main drawback 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: This functionality is provided by symbolic manipulation packages such as Maple, Reduce, Macsyma, or Mathematica. Given a string describing the definition of a function, symbolic manipulation packages provide exact derivatives, expressing the derivatives all in terms of the intermediate variables. Symbolic differentiation is a powerful technique, but it may derive poor computational recipes and may run into resource limitations when the function description is complicated. Functions involving branches or loops cannot be readily handled by symbolic differentiation.

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

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

over and over again to the composition of the elementary operations, one can compute derivative information of f exactly (up to machine precision, of course), in a completely mechanical fashion that avoids the potential pitfalls of divided differences. The techniques of automatic differentiation are directly applicable to functions with branches and loops.

We illustrate automatic differentiation with an example. Assume that we have the sample program shown in Figure 1 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 `x(2)` $= 0$ and is differentiable except at `x(1)` $= 2$.

We can transform this program into one for computing derivatives by associating a derivative object ∇`t` with every variable `t`. Assume that ∇`t` contains the derivatives of `t` with respect to the independent

```
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 2: Sample program of Figure 1 augmented with derivative code

variables `x`,

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

We can propagate these derivatives by using elementary differentiation arithmetic based on the chain rule [16, 29] for computing the derivatives of `y(1)` and `y(2)` as shown in Figure 2. In this example, each assignment to a derivative is actually a vector assignment of length 2.

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. This approach is closely related to the adjoint sensitivity analysis for differential equations that has been used at least since the late sixties, especially in nuclear engineering [9, 10], weather forecasting [26], and neural networks [35].

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 [16, 18, 20]. Despite its advantages from the viewpoint of complexity, the implementation of the reverse mode 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 [24]) 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 [14, 15]).

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 [12, 17, 29].

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 [24] for a survey).

Most of these implementations implement automatic differentiation by means of *operator overloading,* which is a language feature of several modern programming languages, including C++, Ada, Pascal-XSC, and Fortran 90. Operator overloading provides the possibility of associating side-effects with the elementary arithmetic operations. For example, the addition of the derivative vectors that is required in the forward mode can be associated with each addition "+" in the user's program. Operator overloading also allows

for a simple implementation of the reverse mode, since the "tape" can be created as a by-product of the evaluation of f. The only drawback is that for straightforward implementations, the length of the tape is proportional to the number of arithmetic operations performed by f [20, 5]. Recently, Griewank [18] suggested an approach to overcome this limitation through clever checkpointing.

Nonetheless, for all their simplicity and elegance, operator overloading approaches have two fundamental drawbacks:

Loss of context: Since all computation is performed as a by-product of elementary operations, it is very difficult, if not impossible, to perform optimizations that transcend one elementary operation. The resulting disadvantages, especially those associated with the exploitation of parallelism, are discussed in [2].

Loss of Efficiency: The overwhelming majority of codes for which computational scientists want derivatives are written in Fortran 77, which does not support operator overloading. While we can emulate operator overloading by associating a subroutine call with each elementary operation, this approach slows down computation considerably, and usually also imposes some restrictions on the syntactic structure of the code that can be processed. Examples of this approach are DAPRE [28, 33], GRESS/ADGEN [22, 23], and JAKEF [21]. Experiments with some of these systems are described in [32].

2 Hybrid Mode of Automatic Differentiation

We believe that the lack of efficiency of previously existing automatic differentiation tools has prevented automatic differentiation from becoming a standard tool for mainstream high-performance computing, even though there are numerous applications where the need for accurate first- and higher-order derivatives has essentially mandated the use of automatic differentiation techniques and prompted the development of custom-tailored automatic differentiation systems (see [19]). For the majority of applications, however, existing automatic differentiation implementations have provided derivatives substantially slower than divided-difference approximations, discouraging potential users.

Since the efficiency of computing derivatives is so crucial to the success of automatic differentiation for large applications, we are developing ADIFOR, an automatic differentiation tool for Fortran, with the explicit goal of computing derivatives efficiently. Motivated by demands that ADIFOR deliver exact derivatives quickly in order to be considered as a tool for serious high-performance computing, 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, and because, for this restricted case, the reverse mode code can be implemented entirely as inline code. Hence there is no need to construct the tape.

Let us use an example to 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 $\nabla\mathtt{y}$ and $\nabla\mathtt{z}$ and now wish to compute $\nabla\mathtt{w}$. 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 3.

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 $\nabla\mathtt{w}$, 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 $\mathtt{wbar} = \frac{\partial\,\mathtt{W}}{\partial\,\mathtt{W}} = 1.0$. We

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 3: Forward versus reverse mode in computing derivatives of `w = -y/(z*z*z)`

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 simple recipe (and some simple optimizations), we generate the reverse mode code shown in Figure 3.

The forward mode code in Figure 3 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.

3 ADIFOR Design and Implementation

ADIFOR has been developed within the context of the ParaScope Parallel Programming Environment [11], 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 facts for scalars and regular array sections,
- dependence graphs for array elements,
- control flow graphs,
- constant and symbolic facts, 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. Such a variable is called *active.* Variables that do not require derivative information are called *passive.* Interprocedurally, variable nomination proceeds in a series of passes over the program call graph by using an interaction matrix for each subroutine. This interaction matrix represents which input

parameters or variables in common blocks influence which output parameters or variables in common blocks. This analysis is also crucial in determining the sets of active/passive variable binding contexts in which each subroutine may be invoked. For example, consider the code for computing `y = 3.0 * x * x`:

```
subroutine threexx(x,y)
call prod(3.0,x,t)
call prod(t,x,y)
end

subroutine prod(x,y,z)
z = x * y
end
```

In the first call to `prod`, only the second and third of `prod`'s parameters are active, whereas in the second call, all variables are active. ADIFOR recognizes this situation and performs procedure cloning to generate different augmented versions of `prod` for these different contexts. The decision to do cloning based on active/passive variable context will eventually be based on an assessment of the savings made possible by introducing the cloned procedures, in accordance with the goal-directed interprocedural transformation approach being adopted within ParaScope [7].

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. Derivative objects are allocated in a straightforward fashion: Derivative objects for parameters are again parameters. Derivative objects for variables in common blocks and local variables are again allocated in common blocks and as local variables, respectively.

4. The original source code is augmented with derivative statements. The reverse mode is used for assignment statements, and the forward mode is used overall. Subroutine calls are rewritten to propagate derivative information, and procedure cloning is performed as needed.

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].

4 Using ADIFOR

The issues of ease of use and portability have received scant attention in software for automatic differentiation. In many applications, the "function" whose derivatives we wish to compute is a collection of subroutines, and all that should be 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 should be 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. Unlike previous approaches, ADIFOR can deliver this functionality because it views automatic differentiation from the outset as a source transformation problem. The goal is to automate and optimize the source translation process that was shown in very simple examples of the preceding section. By taking a source translator view, we can bring the many man-years of effort of the compiler community to bear on this problem.

ADIFOR differs from other implementations of automatic differentiation (see [24] for a survey) by being based on a source translation paradigm and by having been designed from the outset with large-scale codes and the need for highly efficient derivative computations in mind. 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 set up 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. 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 problems named "camera," "micro," "heart," "polymer," "psycho," and "sand" were given to us by Janet Rogers, National Institute of Standards and Technology in Boulder, Colorado. The test codes submitted to ADIFOR compute elementary Jacobian matrices that are then assembled to form a large sparse Jacobian matrix that is used in an orthogonal-distance regression fit [6]. The code named "adiabatic" is from Larry Biegler, Carnegie-Mellon University Chemical Engineering Department, and implements adiabatic flow, a common module in chemical engineering [31]. The code named "shock" was given to us by Greg Shubin, Boeing Computer Services, Seattle, Washington. This code implements the steady shock tracking

Problem Name	Jacobian Size	Code Size (lines)	Div Diff Run-time (seconds)	ADIFOR Run-time (seconds)	ADIFOR Improve-ment	Machine
Sand	1×4	24	0.16	0.07	56%	RS6000
Sand	1×4	24	0.36	0.18	50%	Sparc 4/490
Psycho	1×5	26	0.70	0.38	46%	RS6000
Psycho	1×5	26	2.95	1.49	49%	Sparc 4/490
Polymer	2×6	34	3.12	1.20	62%	RS6000
Polymer	2×6	34	9.18	4.84	47%	Sparc 4/490
Camera	2×13	97	1.82	1.81	0.5%	RS6000
Camera	2×13	97	8.19	13.87	-69%	Sparc 4/490
Micro	4×20	153	6.39	3.35	47%	RS6000
Micro	4×20	153	23.0	16.17	30%	Sparc 4/490

Table 1: Performance of ADIFOR-generated derivative codes compared to divided-difference approximations on orthogonal-distance regression examples

Problem Name	Jacobian Size	Code Size (lines)	Div Diff Run-time (seconds)	ADIFOR Run-time (seconds)	ADIFOR Improve-ment	Machine
Heart	1×8	1305	11641.1	13941.30	-20%	Sparc1
Adiabatic	6×6	1089	0.54	0.18	67%	Sparc1
Reactor	3×29	1455	42.34	36.14	15%	Sparc 4/490
Reactor	3×29	1455	13.34	8.33	38%	RS6000
Shock	190×190	1403	0.041	0.023	44%	RS6000
Shock	190×190	1403	0.46	0.31	33%	Sparc1

Table 2: Performance of ADIFOR-generated derivative codes compared to divided-difference approximations

method for the axisymmetric blunt body problem [30]. The Jacobian has a banded structure, and the compressed Jacobian has 28 columns, compared with 190 for the "normal" uncompressed Jacobian. Lastly, the code named "reactor" was given to us by Hussein Khalil, Argonne National Laboratory Reactor Analysis and Safety Division. 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 varies gracefully with certain key parameters.

Table 1 and Table 2 summarize the performance of ADIFOR-generated derivative codes with respect to divided differences. These tests were run on a Sparcstation 1, a Sparc 4/400, or an IBM RS6000/550. The numbers reported in Table 1 are actually for 10,000 evaluations of the Jacobian, while those in Table 2 are for a single evaluation of the Jacobian.

The column of the tables 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. 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 conservative approach is 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.

The percentage improvement for the "camera" problem indicates a stronger than expected dependence of running times of ADIFOR-generated code on the choice of compiler and architecture. In fact, the 69% degradation in performance on the "camera" problem is because the Sparc compiler misses an opportunity to move loop-invariant cos and sin invocations outside of loops, as occurs in the following ADIFOR-generated

code:

```
C       cteta = cos(par(4))
        d$0 = par(4)
        do 99969 g$i$ = 1, g$p$
          g$cteta(g$i$) = -sin(d$0) * g$par(g$i$, 4)
99969   continue
        cteta = cos(d$0)
```

ADIFOR will eventually move loop-invariant code outside of the vector loops.

We see that already in its current version, ADIFOR performs well in competition with divided differences. We also see that ADIFOR can handle problems where symbolic techniques would be almost certain to fail, such as the "shock" or "reactor" codes.

ADIFOR-generated derivatives can also outperform hand-coded derivatives. For example, consider the swirling flow problem from the MINPACK–2 test problem collection [25]. The problem consists of a coupled system of boundary value problems describing the steady flow of a viscous, incompressible, axisymmetric fluid between two rotating, infinite coaxial disks. The number of variables in the resulting optimization problem depends on the discretization. Figure 4 shows the performance of the hand-coded derivative code supplied as part of the original swirling flow code and that of the ADIFOR-generated code, properly initialized to exploit the sparsity structure of the Jacobian. On an RS6000/320, the ADIFOR-generated code significantly outperforms the hand-coded derivatives. On one processor of the CRAY Y-MP/18, ADIFOR and the hand-coded derivatives perform comparably. The values of the derivatives computed by the ADIFOR-generated code agree to full machine precision with the values from the hand-coded derivatives. On the other hand, the accuracy of the divided-difference approximations depends on the user's careful choice of a step size.

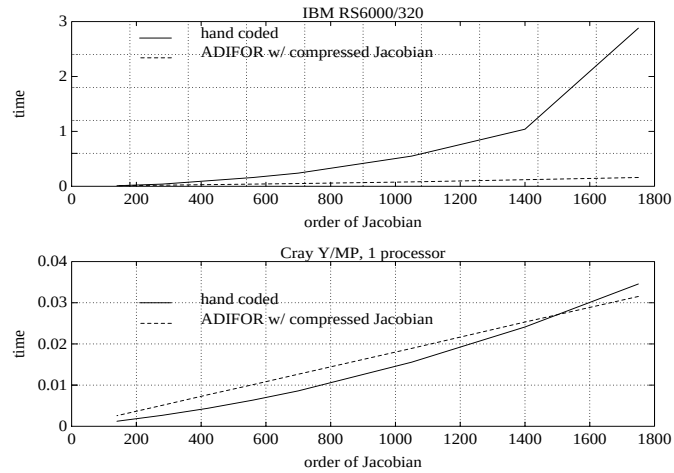

Figure 4: Swirling flow Jacobian

6 Conclusions and Future Work

We conclude that ADIFOR-generated derivatives are an attractive substitute for hand-coded 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.

We are planning many improvements for ADIFOR. The most important are the following:

- generation of code to compute second- and higher-order derivatives as required by many applications in numerical optimization,
- automatic detection of sparsity,
- increased use of the inline version of the reverse mode for better performance, and
- integration with parallel programming models such as Fortran-D.

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] Christian Bischof. Issues in parallel automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penn., 1991, 100–113.

[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., 1991. Also appeared as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Texas.

[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., October 1991.

[5] Christian Bischof and James Hu. Utilities for building and optimizing a computational graph for algorithmic decomposition. Technical Memorandum ANL/MCS–TM–148, Mathematics and Computer Sciences Division, Argonne National Laboratory, Argonne, Ill., April 1991.

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

[7] Preston Briggs, Keith D. Cooper, Mary W. Hall, and Linda Torczon. Goal-directed interprocedural optimization. CRPC Report CRPC–TR90102, Center for Research on Parallel Computation, Rice University, November 1990.

[8] J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations (Runge-Kutta and General Linear Methods).* John Wiley and Sons, 1987.

[9] D. G. Cacuci. Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. *J. Math. Phys.*, 22(12):2794–2802, 1981.

[10] D. G. Cacuci. Sensitivity theory for nonlinear systems. II. Extension to additional classes of responses. *J. Math. Phys.*, 22(12):2803–2812, 1981.

[11] 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.

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

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

[14] 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.

[15] 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*. SIAM, Philadelphia, Penn., 1991, 114–125.

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

[17] 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. Birkhäuser Verlag, Basel, Switzerland, 1991, 135–148.

[18] 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, Argonne, Ill., 1991.

[19] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Penn., 1991.

[20] 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. Also appeared as Preprint MCS–P180–1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1990.

[21] Kenneth E. Hillstrom. JAKEF – A portable symbolic differentiator of functions given by algorithms. Technical Report ANL–82–48, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1982.

[22] 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*. SIAM, Philadelphia, Penn., 1991, 243–250.

[23] Jim E. Horwedel, Brian A. Worley, E. M. Oblow, and F. G. Pin. GRESS version 1.0 users manual. Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, Tenn., 1988.

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

[25] 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*. SIAM, 1991, 32–45.

[26] I. Michael Navon and U. Muller. FESW — A finite-element Fortran IV program for solving the shallow water equations. *Advances in Engineering Software*, 1:77–84, 1970.

[27] Arnold Neumaier. Rigorous recursive calculations with functions. Talk presented at Second International Conference on Industrial and Applied Mathematics (Washington, D.C.), July 1991.

[28] John D. Pryce and Paul H. Davis. A new implementation of automatic differentiation for use with numerical software. Technical Report TR AM-87-11, Mathematics Department, Bristol University, 1987.

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

[30] 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 Journal on Scientific and Statistical Computing*, 3(2):127–144, June 1982.

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

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

[33] Bruce R. Stephens and John D. Pryce. *The DAPRE/UNIX Preprocessor Users' Guide v1.2.* Royal Military College of Science at Shrivenham, 1990.

[34] Hans J. Stetter. Inclusion algorithms with functions as data. *Computing, Suppl.*, 6:213–224, 1988.

[35] P. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Systems Modeling and Optimization*, New York, 1982. Springer Verlag, 762–777.