ADIFOR: Generating Derivative Codes from Fortran Programs

Christian Bischof
Alan Carle
George Corliss
Andreas Griewank
Paul Hovland

CRPC-TR91185-S
December 1991

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Revised: August, 1995. Appears in *Scientific Programming*, 1(1), 1992.

ADIFOR — Generating Derivative Codes from Fortran Programs[*]

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

Argonne Preprint MCS–P263–0991

CRPC Technical Report CRPC–TR91185

Published in Scientific Programming, 1(1), pp. 1–29, 1992.

Abstract. The numerical methods employed in the solution of many scientific computing problems require the computation of derivatives of a function $f : \mathbf{R}^n \rightarrow \mathbf{R}^m$. Both the accuracy and the computational requirements of the derivative computation are usually of critical importance for the robustness and speed of the numerical solution. 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. ADIFOR employs the data analysis capabilities of the ParaScope Parallel Programming Environment, which enable us to handle arbitrary Fortran 77 codes and to exploit the computational context in the computation of derivatives. 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 to compute derivatives by orders of magnitude.

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 Introduction

The methods employed for the solution of many scientific computing problems require the evaluation of derivatives of some function. Probably best known are gradient methods for optimization [22], Newton's method for the solution of nonlinear systems [15, 22], and the numerical solution of stiff ordinary differential equations [8, 21]. Other examples can be found in [17]. In the context of optimization, for example, given a function

$$f : \mathbf{R}^n \rightarrow \mathbf{R},$$

one can find a minimizer x_* of f using variable metric methods that involve the iteration

```
for $i = 1, 2, ....$ do
    Solve $B_i s_i = -\nabla f(x_i)$
    $x_{i+1} = x_i + \alpha_i s_i$
end for
```

for suitable step multipliers $\alpha_i > 0$. Here

$$\nabla f(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{pmatrix} \tag{1}$$

[*]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.

is the *gradient* of f at a particular point x_0, and B_i is a positive definite matrix that may change from iteration to iteration.

In the context of finding the root of a nonlinear function

$$f : \mathbf{R}^n \to \mathbf{R}^n, \quad f = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix},$$

Newton's method requires the computation of the *Jacobian matrix*

$$f'(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \cdots & \frac{\partial}{\partial x_n} f_1(x) \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_1} f_n(x) & \cdots & \frac{\partial}{\partial x_n} f_n(x) \end{pmatrix}. \tag{2}$$

Then, we execute the following iteration:

```
**for** $i = 1, 2, ....$ **do**
    Solve $f'(x_i)s_i = -f(x_i)$
    $x_{i+1} = x_i + s_i$
**end for**
```

Another important application is the numerical solution of initial value problems in stiff ordinary differential equations. Methods such as implicit Runge-Kutta [9] and backward differentiation formula (BDF) [35] methods require a Jacobian which is either provided by the user or approximated by divided differences. Consider a system of ODEs

$$y' = f(t, y), \quad y(t_0) = y_0. \tag{3}$$

System (3) is called *stiff* if its Jacobian $J = \frac{\partial f}{\partial y}$ (in a neighborhood of the solution) has eigenvalues λ_i with $\mathrm{Re}(\lambda_i) << 0$ in addition to eigenvalues of moderate size. Equations of this type arise frequently in chemical reaction models, for example. They can be of arbitrarily large dimension, because they also arise as discretizations of partial differential equations, where J is large and sparse. If explicit methods (multistep, Runge-Kutta, Taylor, or extrapolation) are applied, the step size must be very small in order to retain desirable stability properties of the method. That is why authors as early as the 1920's [19] and again in the late 1940's [20] were led to consider implicit methods in which the approximate solution y_{i+1} at $t = t_{i+1}$ is given by the solution to some nonlinear system

$$\Phi(y_{i+1}) = 0,$$

which is solved by a Newton-type iteration requiring the Jacobian $\frac{\partial \Phi}{\partial y}$. The exact form of Φ differs from one implicit method to another, but for many methods,

$$\Phi(y_{i+1}) = y_{i+1} - \alpha h f(t_{i+1}, y_{i+1}) + \text{ terms not involving } y_{i+1},$$

so the user is asked to supply the Jacobian $J = \frac{\partial f}{\partial y}$.

These methods are examples of a large class of methods for numerical computation, where the computation of derivatives is a crucial ingredient in the numerical solution process. The function f is not usually represented in closed form, but in the form of a computer program.

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.* 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 (these issues are discussed in more detail in [29]):

By Hand: Hand coding is increasingly 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*

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

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 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 independent variables.

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. For example, if

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

we obtain

$$\begin{aligned}
\frac{\partial\, f}{\partial\, x_1} &= x(2) * x(3) * x(4) * x(5) \\
\frac{\partial\, f}{\partial\, x_2} &= x(1) * x(3) * x(4) * x(5) \\
\frac{\partial\, f}{\partial\, x_3} &= x(1) * x(2) * x(4) * x(5) \\
\frac{\partial\, f}{\partial\, x_4} &= x(1) * x(2) * x(3) * x(5) \\
\frac{\partial\, f}{\partial\, x_5} &= x(1) * x(2) * x(3) * x(4).
\end{aligned}$$

This is correct, yet it does not represent a very efficient way to compute the derivatives, since there are a lot of common subexpressions in the different derivative expressions. Symbolic differentiation is a powerful technique, but it may not derive good computational recipes, and it 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

$$\left.\frac{\partial}{\partial t} f(g(t))\right|_{t=t_0} = \left(\left.\frac{\partial}{\partial s} f(s)\right|_{s=g(t_0)}\right) \left(\left.\frac{\partial}{\partial t} g(t)\right|_{t=t_0}\right) \tag{4}$$

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 transforms Fortran 77 programs using this approach. 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, **gp** to 5, and the array **g$x** to the 5×5 identity matrix, then on exit the vector **g$f** 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 like fashion, and the proper pieces of the product are reused in the derivative computation. The rationale underlying this code will be given in Section 2.

Symbolic differentiation uses the rules of calculus in a more or less mechanical way, although some efficiency can be recouped by back-end optimization techniques [13, 28]. In contrast, automatic differentiation is intimately related to the program for the computation of the function to be differentiated. By applying the chain-rule step by step to the elementary operations executed in the course of computing the "function", automatic differentiation computes exact derivatives (up to machine precision, of course) and avoids the potential pitfalls of divided differences. The techniques of automatic differentiation are directly applicable to functions with branches and loops.

ADIFOR is a tool to provide automatic differentiation for programs written in Fortran 77. Given a Fortran subroutine (or collection of subroutines) for a function f, ADIFOR produces Fortran 77 subroutines for the computation of the derivatives of this function. ADIFOR differs from other approaches to automatic differentiation (see [39] for a survey) by being based on a source translator paradigm and by having been designed from the outset with large-scale codes in mind. ADIFOR provides several advantages:

Portability: ADIFOR produces vanilla Fortran 77 code. ADIFOR-generated derivative code does not require any run-time support and can easily be ported between different computing environments.

Generality: ADIFOR supports almost all of Fortran 77, including arbitrary calling sequences, nested subroutines, common blocks, and equivalences. Fortran 77 functions and statement functions will be supported in the next version of ADIFOR. We do not anticipate support for i/o, alternate returns for subroutines, or entry statements.

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.

Extensability: 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, exploit vendor-supplied libraries, and reduce 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.

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.

In the next section, we shall give a brief introduction to automatic differentiation. Section 3 describes how ADIFOR provides this functionality in the context of a source transformation environment, and gives the rationale for choosing such an approach. Section 4 gives a brief introduction into the use of ADIFOR-generated derivative codes, including the exploitation of sparsity structure in the derivative matrices. In Section 5, we present some experimental results which show that the run-time required for ADIFOR-generated exact derivative codes compares very favorably with divided-difference derivative approximations. Lastly, 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 magnitudes.

2 Automatic Differentiation

We illustrate automatic differentiation with an example. Assume that we have the sample program shown in Figure 2 for the computation of a function $f : \mathbf{R}^2 \rightarrow \mathbf{R}^2$. Here, the vector $\mathtt{x}$ contains the independent variables, and the vector $\mathtt{y}$ 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)
end if
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 $f : \mathrm{x} \mapsto \mathrm{y}$

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

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

We can propagate those derivatives by using elementary differentiation arithmetic based on the chain rule (see [45] for more details). For example, the statement

$$\mathtt{a} = \mathtt{x}(1) + \mathtt{x}(2)$$

implies

$$\nabla \mathtt{a} = \nabla \mathtt{x(1)} + \nabla \mathtt{x(2)}.$$

The chain rule, applied to the statement

$$\mathtt{y(1)} = \mathtt{a/x(2)},$$

implies that

$$\nabla \mathtt{y(1)} = \frac{\partial\, \mathtt{y(1)}}{\partial\, \mathtt{a}} * \nabla \mathtt{a} + \frac{\partial\, \mathtt{y(1)}}{\partial\, \mathtt{x(2)}} * \nabla \mathtt{x(2)} = \mathtt{1.0/x(2)} * \nabla \mathtt{a} + (-\mathtt{a}/(\mathtt{x(2)} * \mathtt{x(2)})) * \nabla \mathtt{x(2)}.$$

Care has to be taken when the same variable appears on both the left- and the right-hand sides of an assignment statement. For example, the statement

$$\mathtt{a} = \mathtt{a} * \mathtt{x(i)}$$

implies

$$\nabla \mathtt{a} = \mathtt{x(i)} * \nabla \mathtt{a} + a * \nabla \mathtt{x(i)}.$$

However, simply combining these two statements leads to the wrong results, since the value of `a` referred to in the right-hand side of the ∇`a` assignment is the value of `a` *before* the assignment `a = a*x(i)` has been executed. We avoid this difficulty in the ADIFOR-generated code by using a temporary variable as shown in Figure 3.

Elementary functions are easy to deal with. For example, the statement

$$\mathtt{y(2)} = \mathtt{sin(x(2))}$$

implies

$$\nabla \mathtt{y(2)} = \frac{\partial\, \mathtt{y(2)}}{\partial\, \mathtt{x(2)}} * \nabla \mathtt{x(2)} = \mathtt{cos(x(2))} * \nabla \mathtt{x(2)}.$$

Straightforward application of the chain rule in this fashion then leads to the pseudo-code shown in Figure 3 for computing the derivatives of `y(1)` and `y(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)
end if
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 situation gets more complicated when the source statement is not just a binary operation. For example, 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 as shown in Figure 4,

```
t1 = - y
t2 = z * z
t3 = t2 * z
w = t1 / t3
```

Figure 4. Expansion of `w = - y / (z * z * z)` in unary and binary operations

we could simply apply the mechanism that was used in Figure 3 and associate a derivative computation with each binary or unary statement (the resulting pseudo-code is shown in the left half of Figure 6).

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. Here we *maintain the derivative of the final result with respect to an intermediate quantity.* These quantities are usually called *adjoints.* They 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 [10, 11], in weather forecasting [41], and even in neural networks [50].

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 simple recipe (see [29, 45]), we generate the code shown in Figure 5 for computing `w` and its gradient.

```
/* Compute function values */
     t1 = - y
     t2 = z * z
     t3 = t2 * z
     w = t1 / t3
/* Initialize adjoint quantities */
     wbar = 1.0; t3bar = 0.0; t2bar = 0.0;
     t1bar = 0.0; zbar = 0.0; ybar = 0.0;
/* Adjoints for w = t1 / t3 */
     t1bar = t1bar + wbar * (1 / t3)
     t3bar = t3bar + wbar * (- t1 / t3)
/* Adjoints for t3 = t2 * z */
     t2bar = t2bar + t3bar * z
     zbar  = zbar + t3bar * t2
/* Adjoints for t2 = z * z */
     zbar  = zbar + t2bar * z
     zbar  = zbar + t2bar * z
/* Adjoints for t1 = - y */
     ybar  = - t1bar
     ∇ w = ybar * ∇ y + zbar * ∇ z
     w = t4
```

Figure 5. Reverse mode computation of ∇w

In Figure 6, we juxtapose the derivative computations for `w = -y/(z*z*z)` based on the pure forward mode and those based on the reverse mode for computing ∇`w`. For the reverse mode, we performed some simple optimizations such as eliminating multiplications by 1 and additions to 0.

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

The forward mode code in Figure 6 requires that space be allocated for three auxiliary gradient objects, and the code contains four gradient computation loops. In contrast, the reverse mode code requires only five scalar auxiliary derivative objects and has only one gradient loop. In either case, the storage required by automatic differentiation is at most the amount of storage required by the original function evaluation times the length of the gradient objects computed.

Figures 5 and 6 illustrate a very simple example of using the reverse mode. The reverse mode requires fewer operations 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 [29, 31, 33].

Despite the advantages of the reverse mode with regard to 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 [39]) achieve this by storing a record of every computation performed. Then an interpreter performs a backward pass on this "tape." The resulting overhead often annihilates the complexity advantage of the reverse mode in an actual implementation (see [23, 24]).

ADIFOR uses a hybrid approach. It is generally based on the forward mode, but uses the reverse mode to compute the gradients of assignment statements, since for this restricted case the reverse mode can easily be implemented by a source-to-source translation. 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 multivariate higher-order derivatives [14, 45, 30].

The derivatives computed by automatic differentiation are highly accurate, unlike those computed by divided differences. Griewank and Reese [34] showed that the derivative objects computed in the presence of round-off corresponds to the exact result of a nonlinear system whose partial derivatives have been perturbed by factors of at most $(1 + \varepsilon_{i,j})^2$, where $|\varepsilon_{i,j}| \leq \varepsilon$, the relative machine precision.

3 ADIFOR Design Philosophy

The examples in the preceding section have shown 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 [39] for a survey).

Most of these implementations implement automatic differentiation by means of *operator overloading,* which is a language feature in C++, Ada, Pascal-XSC, and Fortran 90 [18]. Operator overloading provides the possibility of associating side-effects with arithmetic operations. For example, with an addition "+" we now could associate the addition of the derivative vectors that is required in the forward mode. Operator overloading also allows for a simple implementation of the reverse mode, since as a by-product of the computation of f we can store a record of every computation performed and then have an interpreter perform a backward pass on this "tape." The only drawback is that for straightforward implementations, the length of the tape is proportional to the number of arithmetic operations performed [33, 5]. Recently, Griewank [31] suggested an approach to overcome this limitation through clever checkpointing.

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

Loss of context: Since all computation is performed as a by-product of an elementary operation, it is very difficult, if not impossible, to perform optimizations that transcend one elementary operation (such as the constant-folding techniques that simplified the reverse mode shown in Figure 5 into that shown in Figure 6). 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, which does not support operator overloading. While we can emulate operator overloading by associating a subroutine call with each elementary operation, this approach slows 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 [43, 49], GRESS/ADGEN [37, 38], and JAKEF [36]. Experiments with some of those systems are described in [48].

The lack of efficiency of previously existing 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 essentially mandated the use of automatic differentiation techniques and prompted the development of custom-tailored automatic differentiation systems (see [32]). For the majority of applications, however, automatic differentiation techniques were substantially slower than divided-difference approximations, discouraging potential users.

The issues of ease of use and portability have received scant attention in software for automatic differentiation as well. In many applications, the "function" of which we wish to compute derivatives is a collection of subroutines, and all that really should be expected of the user is to specify which of the variables correspond to the independent and dependent variables. In addition, the automatic differentiation code should be easily transportable 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 is based on the ParaScope programming environment [12] which combines dependence analysis with interprocedural analysis to support the semi-automatic parallelization of Fortran programs. While our primary goal is not the parallelization of Fortran programs, the ParaScope environment provides us with a Fortran parser, data abstractions for representing Fortran programs, and tools for constructing and manipulating those representations. In particular, ParaScope tools gather

- data flow facts for scalars and arrays,
- dependence graphs for array elements,
- control flow graphs, and
- constant and symbolic facts.

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. Such a matrix represents a bipartite graph. Input parameters or variables in common blocks are connected with output parameters or variables in common blocks whose values they influence. This dependency 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 following 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 a compiler-based approach is that we have the mechanism in place for simplifying the derivative code that has been generated by application of the simple statement-by-statement rules. For example, consider the reverse mode code shown in Figure 5. By applying constant folding and eliminating variables that are used only once, we eliminate multiplications by 1.0 and additions to 0, and we reduce the number of variables that must be allocated.

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 intra- 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, the forward mode 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 users' 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 is included in [4].

4 The Functionality of ADIFOR-Generated Derivative Codes

The functionality provided by ADIFOR is best understood through an example. Our example is adapted from problem C2 in the STDTST set of test problems for stiff ODE solvers [25]. The routine `FCN2` shown in Figure 7 that computes the right-hand side of a system of ordinary differential equations $y' = f(x,y)$ by calling a subordinate routine `FCN`. In the numerical solution of the ordinary differential equation, the Jacobian $\frac{\partial f}{\partial y}$ is required.

```
      SUBROUTINE FCN2(M,X,Y,YP)

      INTEGER  N
      DOUBLE PRECISION X, Y(M), YP(M)
      INTEGER        ID, IWT
      DOUBLE PRECISION W(20)
      COMMON         /STCOM5/W, IWT, N, ID

      CALL FCN(X,Y,YP)
      RETURN
      END

      SUBROUTINE FCN(X,Y,YP)

C     ROUTINE TO EVALUATE THE DERIVATIVE F(X,Y) CORRESPONDING TO THE
C     DIFFERENTIAL EQUATION:
C                    DY/DX = F(X,Y) .
C     THE ROUTINE STORES THE VECTOR OF DERIVATIVES IN YP(*). THE
C     DIFFERENTIAL EQUATION IS SCALED BY THE WEIGHT VECTOR W(*)
C     IF THIS OPTION HAS BEEN SELECTED (IF SO IT IS SIGNALLED
C     BY THE FLAG IWT).

      DOUBLE PRECISION X, Y(20), YP(20)
      INTEGER        ID, IWT, N
      DOUBLE PRECISION W(20)
      COMMON         /STCOM5/W, IWT, N, ID
      DOUBLE PRECISION SUM, CPARM(4), YTEMP(20)
      INTEGER        I, IID
      DATA           CPARM/1.D-1, 1.D0, 1.D1, 2.D1/

      IF (IWT.LT.0) GO TO 40
      DO 20 I = 1, N
         YTEMP(I) = Y(I)
         Y(I) = Y(I)*W(I)
   20 CONTINUE
   40 IID = MOD(ID,10)

C     ADAPTED FROM PROBLEM C2
      YP(1) = -Y(1) + 2.D0
      SUM = Y(1)*Y(1)
      DO 50 I = 2, N
         YP(I) = -10.0D0*I*Y(I) + CPARM(IID-1)*(2**I)*SUM
         SUM = SUM + Y(I)*Y(I)
   50 CONTINUE

      IF (IWT.LT.0) GO TO 680
      DO 660 I = 1, N
         YP(I) = YP(I)/W(I)
         Y(I) = YTEMP(I)
  660 CONTINUE
  680 CONTINUE
      RETURN
      END
```

Figure 7. Original code for problem C2

Nominating **Y** as independent variables, and **YP** as dependent ones, ADIFOR produces the code shown in Figures 8 and 9. We use the dollar sign \$ to indicate ADIFOR-generated names. In practice, ADIFOR generates variable names which do not conflict with any names appearing in the original program.

```
      subroutine g$fcn2$6(g$p$, m, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
C
C       ADIFOR: runtime gradient index
        integer g$p$
C       ADIFOR: translation time gradient index
        integer g$pmax$
        parameter (g$pmax$ = 20)
C       ADIFOR: gradient iteration index
        integer g$i$
C
        integer ldg$y
        integer ldg$yp
        integer n
        double precision x, y(m), yp(m)
        integer id, iwt
        double precision w(20)
        common /stcom5/ w, iwt, n, id
C
C       ADIFOR: gradient declarations
        double precision g$y(ldg$y, m), g$yp(ldg$yp, m)
        if (g$p$ .gt. g$pmax$) then
          print *, "Parameter g$p$ is greater than g$pmax."
          stop
        endif
        call g$fcn$6(g$p$, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
        return
      end
```

Figure 8. ADIFOR-generated code for problem C2 (part 1)

```
      subroutine g$fcn$6(g$p$, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
C
C       ADIFOR: runtime gradient index
        integer g$p$
C       ADIFOR: translation time gradient index
        integer g$pmax$
        parameter (g$pmax$ = 20)
C       ADIFOR: gradient iteration index
        integer g$i$
C
        integer ldg$y
        integer ldg$yp
C       ROUTINE TO EVALUATE THE DERIVATIVE F(X,Y) CORRESPONDING TO THE
C       DIFFERENTIAL EQUATION:
C       DY/DX = F(X,Y) .
C       THE ROUTINE STORES THE VECTOR OF DERIVATIVES IN YP(*). THE
C       DIFFERENTIAL EQUATION IS SCALED BY THE WEIGHT VECTOR W(*)
C       IF THIS OPTION HAS BEEN SELECTED (IF SO IT IS SIGNALLED
C       BY THE FLAG IWT).
        double precision x, y(20), yp(20)
        integer id, iwt, n
        double precision w(20)
        common /stcom5/ w, iwt, n, id
        double precision sum, cparm(4), ytemp(20)
        integer i, iid
        data cparm /1.d-1, 1.d0, 1.d1, 2.d1/
C
C       ADIFOR: gradient declarations
        double precision g$y(ldg$y, 20), g$yp(ldg$yp, 20)
        double precision g$sum(g$pmax$), g$ytemp(g$pmax$, 20)
        if (g$p$ .gt. g$pmax$) then
          print *, "Parameter g$p$ is greater than g$pmax."
          stop
        endif
        if (iwt .lt. 0) then
          goto 40
        endif
        do 99999, i = 1, n
```

```
C           ytemp(i) = y(i)
          do g$i$ = 1, g$p$
            g$ytemp(g$i$, i) = g$y(g$i$, i)
          enddo
          ytemp(i) = y(i)
C           y(i) = y(i) * w(i)
          do g$i$ = 1, g$p$
            g$y(g$i$, i) = w(i) * g$y(g$i$, i)
          enddo
          y(i) = y(i) * w(i)
20        continue
99999   continue
40      iid = mod(id, 10)
C       ADAPTED FROM PROBLEM C2
C       yp(1) = -y(1) + 2.d0
        do g$i$ = 1, g$p$
          g$yp(g$i$, 1) = -g$y(g$i$, 1)
        enddo
        yp(1) = -y(1) + 2.d0
C       sum = y(1) * y(1)
        do g$i$ = 1, g$p$
          g$sum(g$i$) = y(1) * g$y(g$i$, 1) + y(1) * g$y(g$i$, 1)
        enddo
        sum = y(1) * y(1)
        do 99998, i = 2, n
C           yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
          do g$i$ = 1, g$p$
            g$yp(g$i$, i) = cparm(iid - 1) * (2 ** i) * g$sum(g$i$) + -1
     *0.0d0 * i * g$y(g$i$, i)
          enddo
          yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
C           sum = sum + y(i) * y(i)
          do g$i$ = 1, g$p$
            g$sum(g$i$) = g$sum(g$i$) + y(i) * g$y(g$i$, i) + y(i) * g$y
     *(g$i$, i)
          enddo
          sum = sum + y(i) * y(i)
50        continue
99998   continue
        if (iwt .lt. 0) then
          goto 680
        endif
        do 99997, i = 1, n
C           yp(i) = yp(i) / w(i)
          do g$i$ = 1, g$p$
            g$yp(g$i$, i) = (1 / w(i)) * g$yp(g$i$, i)
          enddo
          yp(i) = yp(i) / w(i)
C           y(i) = ytemp(i)
          do g$i$ = 1, g$p$
            g$y(g$i$, i) = g$ytemp(g$i$, i)
          enddo
          y(i) = ytemp(i)
660       continue
99997   continue
680     continue
        return
      end
```

Figure 9. ADIFOR-generated code for problem C2 (part 2)

We see that the derivative codes have a gradient object associated with every dependent variable. Our convention is to associate a gradient `g$<var>` of leading dimension `ldg$<var>` with variable `<var>`. The calling sequence of **`g$foo$n`** is derived from that of `foo` by inserting an argument **`g$p$`** denoting the length of the gradient vectors as the first argument, and then copying the calling sequence of `foo`, inserting **`g$<var>`** and **`ldg$<var>`** after every active variable **`<var>`**. Passive variables or those not of REAL or DOUBLE PRECISION type are left untouched.

Subroutine **g$fcn2$6** relates to the Jacobian

$$J_{yp} = \begin{pmatrix} \frac{\partial yp_1}{\partial y_1} & \dots & \frac{\partial yp_1}{\partial y_m} \\ \vdots & & \vdots \\ \frac{\partial yp_m}{\partial y_1} & \dots & \frac{\partial yp_m}{\partial y_m} \end{pmatrix}$$

as follows: Given input values for **gp, m, x, y, gy, ldgy,** and **ldg$yp**, the routine **g$fcn2$6** computes **yp** and **g$yp**, where

$$\texttt{g\$yp(1:g\$p\$,1:m)} = (J_{yp}(\texttt{g\$y(1:g\$p\$,1:m)}^T))^T$$

The superscript T denotes matrix transposition. While the implicit transposition may seem awkward at first, this is the only way to handle assumed-size arrays (like `real a(*)`) in subroutine calls. It is the responsibility of the user to allocate **g$yp** and **g$y** with leading dimensions **ldg$yp** and **ldg$y** that are at least **gp**.

For example, to compute the Jacobian of **yp** with respect to **y**, we initialize **g$y** to be an $\mathtt{m} \times \mathtt{m}$ identity matrix and set **p** to **m**. After the call to **g$fcn2$6**, **g$yp** contains the transpose of the Jacobian of **yp** with respect to **y**. If we wish to compute only a matrix-vector product (as is often the case when iterative schemes are applied to solve equation systems with the Jacobian as the coefficient matrix), we set $p = 1$ and **g$y** to the vector by which the Jacobian is to be multiplied.

From the forementioned discussion, ADIFOR-generated code is well suited for computing dense Jacobian matrices. We will now show that it can also exploit the sparsity structure of Jacobian matrices. Remember that the forward mode of automatic differentiation upon which ADIFOR is mainly based requires roughly **gp** operations for every assignment statement in the original function. Thus, if we compute a Jacobian J with n columns by setting gp $= n$, its computation will require roughly n times as many operations as the original function evaluation, independent of whether J is dense or sparse. However, it is well known [16, 27] that the number of function evaluations that are required to compute an approximation to the Jacobian by divided differences can be much less than n if J is sparse. The same idea can be applied to greatly reduce the running time of ADIFOR-generated derivative code as well.

As an example, consider the swirling flow problem, which comes from Parter [42] and is part of the MINPACK–2 test problem collection [1]. The problem is 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. For example, for $n = 56$ the Jacobian of F has the structure shown in Figure 10.

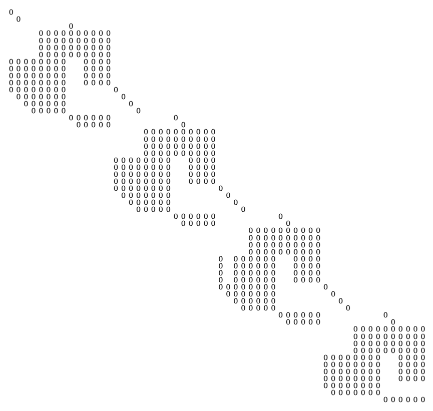
Figure 10: Structure of the swirling flow Jacobian, n = 56

By using a graph coloring algorithm designed to identify structurally orthogonal columns (we used the one described in [16]), we can determine that this Jacobian can be grouped into 14 sets of structurally orthogonal columns, independent of the size of the problem. As a result, we initialize a 56×14 matrix $\texttt{g\$x}^T$ to the structure shown in Figure 11. Here every circle denotes the value 1.0. The structure of the resulting compressed Jacobian $\texttt{g\$Fval}^T$ is shown in Figure 11 as well. Here every circle denotes a nonzero entry. Now, instead of $\texttt{g\$p\$} = 56$, a size of $\texttt{g\$p\$} = 14$ is sufficient, a sizeable reduction in cost. The proper and efficient initialization of ADIFOR-generated derivative codes is described in detail in [4].

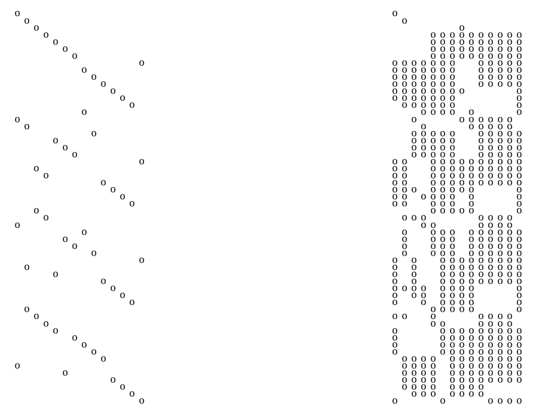

Figure 11: Left: Structure of $\texttt{g\$x}^T$ Right: Structure of $\texttt{g\$Fval}^T$

One issue that deserves some attention is that of error handling. Exceptional conditions arise because of branches in the code or because subexpressions may be defined but not be differentiable ($\sqrt{(x)}$ at $x = 0$, for example). ADIFOR knows when Fortran intrinsics are nondifferentiable, and traps to an error handler if we wish to compute derivatives at a point where the derivatives do not exist. The current error-handling mechanism of ADIFOR is described in [3].

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 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 [6]. The code 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 [46]. The Jacobian has a banded structure. The compressed Jacobian has 28 columns, compared to 190 for the "normal" Jacobian. The code named "adiabatic" is from Larry Biegler, Chemical Engineering, Carnegie-Mellon University and implements adiabatic flow, a common module in chemical engineering [47]. Lastly, 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.

Tables 1 and 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. We used different machines because the codes were submitted from different computing environments. The numbers

Table 1: Performance of ADIFOR-generated derivative codes compared to divided-difference approximations on orthogonal-distance regression examples for 10,000 Jacobian evaluations

Problem Name	Jacobian Size	Code Size (lines)	Div Diff Run time (seconds)	ADIFOR Run time (seconds)	ADIFOR Improve-ment	Machine
Camera	2×13	97	1.82	1.81	0.5%	RS6000/550
Camera	2×13	97	8.19	13.87	-69%	SPARC 4/490
Micro	4×20	153	6.39	3.35	47%	RS6000/550
Micro	4×20	153	23.0	16.17	30%	SPARC 4/490
Polymer	2×6	34	3.12	1.20	62%	RS6000/550
Polymer	2×6	34	9.18	4.84	47%	SPARC 4/490
Psycho	1×5	26	0.70	0.38	46%	RS6000/550
Psycho	1×5	26	2.95	1.49	49%	SPARC 4/490
Sand	1×4	24	0.16	0.07	56%	RS6000/550
Sand	1×4	24	0.36	0.18	50%	SPARC 4/490

Table 2: 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 Improve-ment	Machine
Reactor	3×29	1455	42.34	36.14	15%	SPARC 4/490
Reactor	3×29	1455	13.34	8.33	38%	RS6000/550
Adiabatic	6×6	1089	0.54	0.18	67%	SPARC 1
Heart	1×8	1305	11641.1	13941.30	-20%	SPARC 1
Shock	190×190	1403	0.041	0.023	44%	RS6000/550
Shock	190×190	1403	0.46	0.31	33%	SPARC 1

reported in Table 1 are 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 approach is conservative, yet fairly 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 a result of the SPARC compiler's missing 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)
```

If we edit the ADIFOR-generated code by hand to extract the invariant expression, we get a similar performance on the SPARC. Moving loop-invariant code outside of loops is one of the performance improvements that we will implement in later versions.

We see that already in its current version, ADIFOR performs well in competition with divided difference approximations. It is up to a factor of three faster, and never worse by more than a factor of 1.69. This improvement was 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. The ADIFOR-generated derivative codes had up to four times as many lines of code as the code that was submitted to ADIFOR.

The performance of ADIFOR-generated derivatives can even be better than that of hand-coded derivatives. For example, for the swirling flow problem mentioned in the preceding section, we obtain the performance shown Figure 12.

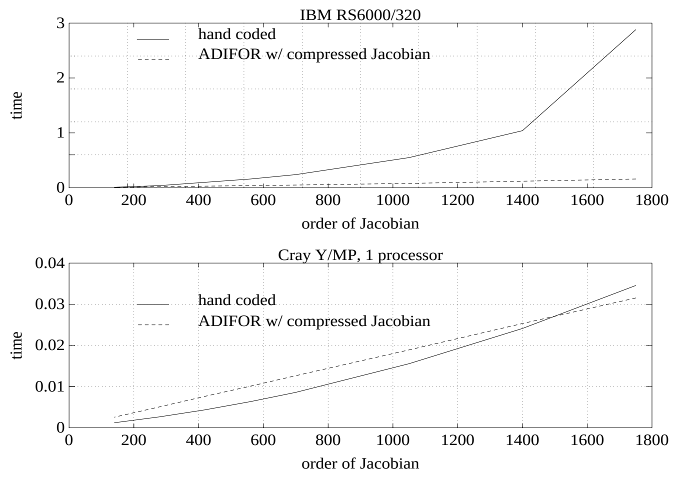

Figure 12: Swirling flow Jacobian

Figure 12 shows the performance of the hand-derived derivative code supplied as part of the MINPACK-2 test set collection [40], and that 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, the two approaches 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. The accuracy of the finite difference approximations, on the other hand, depends on the user's careful choice of a step size.

We conclude that ADIFOR-generated derivatives are a more than suitable substitute for hand-coded or divided-difference derivatives. Virtually no time investment is required by the user to generate the codes. In most of our example codes, ADIFOR-generated codes outperform divided-difference derivative approximations. In addition, the fact that ADIFOR computes highly accurate derivatives 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 improvements for ADIFOR. The most important are

- second- and higher-order derivatives,
- automatic detection of sparsity,
- increased use of the reverse mode for better performance, and
- integration with Fortran parallel programming environments such as Fortran-D [26].

Second-order derivatives are a natural extension, and this functionality is required for many applications in numerical optimization. In addition, for sensitivity analysis applications, second derivatives reveal correlations between various parameters. While we currently can just process the ADIFOR-generated code for first derivatives, much can be gained by computing both first- and second-order derivatives at the same time [30, 44].

The automatic detection of sparsity is a functionality that is unique to automatic differentiation. Here we exploit the fact that in automatic differentiation, the computation of derivatives is intimately related to the computation of the function itself. The key observation is that all our gradient computations have the form

$$\text{vector} = \sum_i \text{scalar}_i * \text{vector}_i.$$

By merging the structure of the vectors on the right-hand side, we can obtain the structure of the vector on the left-hand side. In addition, the proper use of sparse vector data structures will ensure that we perform computations only with the non-zero components of the various derivative vectors.

We can improve the speed of ADIFOR-generated derivative code through increased use of the reverse mode. The reverse modes 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 do 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, another problem in the MINPACK-2 test set collection, we obtained the performance shown in Figure 13. This figure shows the ratio of gradient/function evaluation on a Solbourne 5E/900 for the current ADIFOR version, and for a hand-modified ADIFOR code that uses the reverse mode for the bodies of parallel loops. If *nint* is the number of grid points in each dimension, then the gradients are of size $nint * nint$.

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 stop at a problem of size 4900 only because at that size, we ran out of memory.

These examples for which we have "compiled" ADIFOR-generated code by hand show again the promise of viewing automatic differentiation as a syntax transformation process. By taking advantage of the context (parallel loops, in this case) of a piece of code, we can choose whatever automatic differentiation technique is most applicable, and generate the most efficient code for the computation of derivatives. In many applications where the computation of derivatives currently requires the dominant portion of the running time, the use

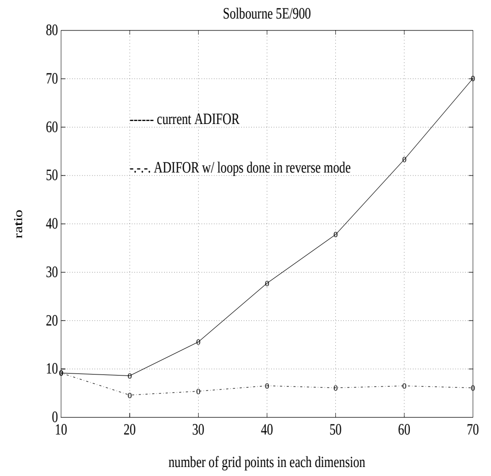

Figure 13: Ratio of gradient/function evaluation

of ADIFOR-generated derivatives will then lead to dramatic improvements, *without having to change the algorithm that uses the derivative information, or the coding of the 'function' for which derivatives are required.*

References

[1] Brett Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK–2 test problem collection (preliminary version). Technical Memorandum MCS–TM–150, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, Ill. 60439, May 1991.

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

[3] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Memorandum ANL/MCS–TM–159, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, Ill. 60439, 1991.

[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, 9700 S. Cass Ave., Argonne, Ill. 60439, 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, 9700 South Cass Ave., Argonne, Ill. 60439, 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, Houston, Tex., November 1990.

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

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

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

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

[12] 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.

[13] Bruce W. Char. Computer algebra as a toolbox for program generation and manipulation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 53 – 60. SIAM, Philadelphia, Penn., 1991.

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

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

[16] 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.

[17] George F. Corliss. Applications of differentiation arithmetic. In Ramon E. Moore, editor, *Reliability in Computing*, pages 127 – 148. Academic Press, London, 1988.

[18] George F. Corliss. Overloading point and interval Taylor operators. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 139 – 146. SIAM, Philadelphia, Penn., 1991.

[19] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100:32 – 74, 1928.

[20] J. Crank and P. Nicholson. A practical method for numerical integration of solutions of partial differential equations of heat conduction type. *Proc. Cambridge Philos. Soc.*, 43:50 ff, 1947.

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

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

[23] 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.

[24] 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.

[25] Wayne H. Enright and John D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Trans. Math. Software*, 13(1):1 – 22, 1987.

[26] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran d language specification. CRPC Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, Houston, Tex., December 1990.

[27] D. Goldfarb and P. Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, pages 69 – 88, 1984.

[28] Victor V. Goldman, J. Molenkamp, and J. A. van Hulzen. Efficient numerical program generation and computer algebra environments. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 74 – 83. SIAM, Philadelphia, Penn., 1991.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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, 9700 S. Cass Ave., Argonne, Ill. 60439, 1990.

[34] 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.

[35] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II (Stiff and Differential-Algebraic Problems)*, volume 14 of *Springer Series in Computational Mathematics.* Springer Verlag, New York, 1991.

[36] 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, 9700 South Cass Ave., Argonne, Ill. 60439, 1982.

[37] 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.

[38] 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. 37830, 1988.

[39] 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.

[40] 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.

[41] 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.

[42] Seymour V. Parter. On the swirling flow between rotating coaxial disks: A survey. In W. Eckhaus and E. M. de Jager, editors, *Theory and Applications of Singular Perturbations*, volume 942 of *Lecture Notes in Mathematics*, pages 258 – 280. Springer Verlag, New York, 1982.

[43] 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.

[44] Louis B. Rall. Applications of software for automatic differentiation in numerical computation. In G. Alefeld and R. D. Grigorieff, editors, *Fundamentals of Numerical Computation (Computer Oriented Numerical Analysis)*, Computing Supplement No. 2, pages 141 – 156. Springer Verlag, Berlin, 1980.

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

[46] 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.

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

[48] 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*, pages 297 – 306. SIAM, Philadelphia, Penn., 1991.

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

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