The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs

Christian Bischof   Alan Carle
Peyvand Khademi   Andrew Mauer

CRPC-TR94491
1994

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

Revised: May, 1995.

The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs*

Christian Bischof†
Alan Carle‡
Peyvand Khademi†
Andrew Mauer†

Argonne Preprint ANL-MCS-P481-1194
CRPC Technical Report CRPC-TR94491

Abstract. Automatic Differentiation is a technique for augmenting computer programs with statements for the computation of derivatives based on the chain rule of differential calculus. The ADIFOR 2.0 system provides automatic differentiation of Fortran 77 programs for first-order derivatives. The ADIFOR 2.0 system consists of three main components: The ADIFOR 2.0 preprocessor, the ADIntrinsics Fortran 77 exception-handling system, and the SparsLinC library. The combination of these tools provides the ability to deal with arbitrary Fortran 77 syntax, to handle codes containing single- and double-precision real- or complex-valued data, to fully support and easily customize the translation of Fortran 77 intrinsics, and to transparently exploit sparsity in derivative computations. ADIFOR 2.0 has been successfully applied to a 60,000-line code, which we believe to be a new record in automatic differentiation.

Key words. Automatic differentiation, ADIFOR, derivative, gradient, Jacobian, chain rule, source transformation and optimization, ADIntrinsics, ParaScope, SparsLinC.

1 Introduction

Let f be a computer model, and denote by $f(x)$ its output produced for a particular input x. Employing the Taylor expansion of f around a reference state x_o, we have

$$f(x_o + \triangle x) = f(x_o) + \frac{\partial f(x_o)}{\partial x}\triangle x + \frac{1}{2}(\triangle x)^T \frac{\partial^2 f(x_o)}{\partial x^2}\triangle x + HO(x_o, \triangle x), \tag{1}$$

where the higher-order terms $HO(x_o, \triangle x)$ satisfy $||HO(x_o, \triangle x)|| = O(||\triangle x||^3)$. Hence, the value of the first- and second-order derivatives (we also interchangeably use the terms first- and second-order sensitivities) allows us to derive a linear first-order or quadratic second-order approximation of f around the base state x_o.

Derivatives provide a way for computing a relatively simple approximation of f, and thus allow one to inexpensively explore the behavior of f in the neighborhood of x_o. Hence, derivatives are ubiquitous

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D and Cooperative Agreement No. NCCW-0027, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.
†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439-4844, {bischof,khademi,mauer}@mcs.anl.gov.
‡Center for Research on Parallel Computation, Rice University, 6100 S. Main Street, Houston, TX 77251-1892, carle@cs.rice.edu.

in numerical computing. Examples are methods for minimization or the solution of nonlinear systems of equations [31, 59], or the numerical solution of stiff ordinary differential equations [24], partial differential equations [70], and differential-algebraic equations [22].

During the past decade, large computer models have become common, as a consequence of the tremendous expansion of computational capabilities. For such models, the computation of derivatives becomes even more important, as it may be the most compute-intensive part of the overall computation. We mention three examples.

Sensitivity Analysis: Here one tries to assess the sensitivity of a computational model to perturbations in its parameters or initial conditions. Sensitivity analysis usually takes place in the model validation stage, to verify robustness with respect to empirically determined parameters or to verify that the model behaves as suggested by experimental data.

Inverse Problems: The goal here is to calibrate the initial state of a computer model such that its behavior best matches a series of experimentally acquired data. The solution process usually employs some variant of Newton's method. A collection of articles related to this subject can be found, for example, in [33].

(Multidisciplinary) Design Optimization: Here one tries to find the optimal setting of input parameters of a computer model with respect to a cost function that quantifies the quality of the overall design. This approach constitutes probably the most rapidly expanding application area of numerical optimization, since engineers are moving from a "repeated simulation" paradigm for computer-assisted design to one where numerical optimization techniques are employed to explore the design space in a goal-oriented fashion. A collection of articles related to this subject can be found, for example, in [1, 33].

For purposes of illustration, assume that we have a code for the computation of a function f and $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$, and 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:

By Hand: One can differentiate the code by hand and thus arrive at a code that also computes derivatives. However, handcoding of derivatives for a large code is a tedious and error-prone process; moreover, for nonlinear functions, the derivatives are generally more complicated than the function itself. Hence, developing a derivative-code by hand is liable to be a considerable amount of work in comparison with the development of the original code, although it is likely to result in the most efficient code.

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

$$\frac{\partial f(x)}{\partial x_i}\Big|_{x=x_0} \approx \frac{f(x_0 \pm h * e_i) - f(x_0)}{\pm h} \tag{2}$$

or *central differences*

$$\frac{\partial f(x)}{\partial x_i}\Big|_{x=x_0} \approx \frac{f(x_0 + h * e_i) - f(x_0 - h * e_i)}{2h}. \tag{3}$$

Here e_i is the ith Cartesian basis vector. From (1) it can be easily seen that this approach leads to a first- or second-order approximation of the desired derivatives. 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 to minimize the *truncation error* resulting from the omission of higher-order terms in (1), but the resulting subtraction of two almost equal floating-point numbers may lead to significant *cancellation error* (see, for example [39]). At the very best, approach (2), for example, results in a derivative approximation that has half the significant digits of f. These issues, as well as sensible ways of choosing the stepsize, are discussed, for example, in [38].

Symbolic Differentiation: Symbolic manipulators like Maple, Macsyma, or Reduce provide powerful capabilities for manipulating algebraic expressions but are, in general, unable to deal with constructs such as branches, loops, or subroutines that are inherent in computer codes. In addition, for every binary operator (except + or -), the string describing the derivative expression in essence doubles, leading to a combinatorial explosion effect (although some efficiency can be recouped by back-end optimization techniques [28, 40]). Therefore, differentiation using a symbolic manipulator still requires considerable human effort to break down an existing computer code into pieces digestible by a symbolic manipulator and to assemble the resulting pieces into a usable derivative code.

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` (see, for example, [42, 63]. By applying the chain rule

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

over and over again to the composition of those elementary operations, one can compute, in a completely mechanical fashion, derivatives of f that are correct up to machine precision [46]. The techniques of automatic differentiation are directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines. We also note that, unlike handcoding or symbolically assisted approaches, automatic differentiation enables derivatives to be updated easily when the original code changes.

The ADIFOR (Automatic Differentiation of Fortran) system provides 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. The ADIFOR approach provides four benefits:

Ease of Use: ADIFOR requires only that the user supply the Fortran source code and indicate the variables that correspond to the independent and dependent variables.

Portability: ADIFOR produces vanilla Fortran 77 code, which also helps greatly with code verification.

Efficiency: ADIFOR-generated derivative code usually outperforms divided-difference approximations.

Extensibility: ADIFOR employs a consistent subroutine-naming scheme that makes it easy to exploit domain-specific knowledge.

The ADIFOR project began in the summer of 1991. A prototype version of ADIFOR was operational and in use in late 1991 and is described in [6]. Two major revisions of the system were subsequently completed; the June 1993 version is called ADIFOR 1.0. ADIFOR 1.0 was successfully employed in very different areas of science of engineering: aeronautical multidisciplinary design optimization [5, 68], aeronautical computational fluid dynamics [9, 13, 27, 41, 56, 57], weather modeling [19, 25, 61, 62], groundwater contaminant transport [18, 69], aquifer modeling [30, 50], structural engineering [29], statistics [23], mechanical system design [49], power networks [52], reactor modeling [60], and large-scale numerical optimization [4, 14, 64]. The largest of these codes was 25,000 lines long and described 3-D turbulent flow over an airplane wing. The experiences with ADIFOR 1.0 demonstrated that automatic differentiation, properly implemented, is useful for scientists from a wide variety of fields, and applicable to codes of arbitrary length and complexity.

The focus of this paper is on the ADIFOR 2.0 system, which constitutes a major redevelopment effort and provides significantly more functionality. The ADIFOR 2.0 system offers the following new features:

Full Fortran 77 Support: ADIFOR 1.0 did not, for example, support `COMPLEX` arithmetic, `FUNCTION`s (versus `SUBROUTINE`s), statement functions, or procedure parameters. In addition to these features, the ADIFOR 2.0 preprocessor also supports common extensions such as `DOUBLE COMPLEX`, `INCLUDE` statements, and `IMPLICIT NONE`.

Flexible Intrinsic Handler: The ADIntrinsics 1.0 system provides for various reporting levels in response to exceptions such as the differentiation of `sqrt(x)` when `x` is zero, and can easily be customized through the use of template files.

Transparent Sparsity Support: Code generated with ADIFOR 2.0 can perform derivative computations using the SparsLinC (Sparse Linear Combination) library, thus transparently exploiting sparsity arising in large sparse Jacobian computations or gradients of functions that have a sparse Hessian.

Code Customization: ADIFOR 2.0 provides mechanisms to generate code that is particularly suited for the computation of Jacobian$*$vector products.

The paper is structured as follows. In the next section we motivate the classical forward and reverse mode of automatic differentiation, viewing automatic differentiation as a source translation problem, and describe the approach taken by ADIFOR. In Section 3, we describe the new ADIFOR 2.0 system. In Subsection 3.1, we describe the capabilities of the ADIFOR 2.0 preprocessor, which transforms Fortran code into a canonical form suitable for automatic differentiation, determines which variables must be augmented with derivative objects, and generates the derivative code, with templates at call sites of Fortran intrinsics. Subsection 3.2 describes the ADIntrinsics system, which translates the templates into Fortran 77 code, governed by user-customizable prototype files describing the action to be taken, and the desired level of error reporting. The SparsLinC library is described in Subsection 3.3; we present scenarios that suggest its use, and we give some experimental results. Lastly, we summarize our contributions and discuss directions of future work.

```
y(1) = 1.0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      y(1) = x(i) ◇ y(1) ◇ y(1)
   else
      y(2) = x(i) ◇ y(2) ◇ y(2)
   endif
enddo
```

Figure 1: Sample Code Fragment

2 Automatic Differentiation as a Source Transformation and the ADIFOR Approach

The fact that the chain rule can be applied in a mechanical fashion has been rediscovered several times since the 1960s (see, for example, the papers in Part I of [44] and the references therein). Traditionally, two approaches to AD have been developed: the so-called forward and reverse modes. These modes are distinguished by how the chain rule is used to propagate derivatives through the computation. The forward mode accumulates the derivatives of intermediate variables with respect to the independent variables, whereas the reverse mode propagates the derivatives of the final values with respect to intermediate variables. ADIFOR takes an approach that employs both the forward and the reverse mode. In either case, automatic differentiation produces code that, in the absence of floating-point exceptions, computes the values of the analytical derivatives accurate up to machine precision.

We illustrate the differences between these approaches by deriving code for computing $\frac{\partial y}{\partial x(1:n)}$ from the code fragment shown in Figure 1, considering the cases where "$\diamond$" is either "$*$" or "$+$." We take a source transformation approach, rewriting the original code into one that also provides for the computation of derivatives.

2.1 The Forward Mode

To apply the forward mode of automatic differentiation, we first break down the code into elementary unary and binary operations and arrive at the code shown in Figure 2. Now we can compute derivatives as shown in Figure 3, much in the way that the chain rule of differential calculus is usually taught. We use the notation ∇`s` to denote the derivative object associated with the program variable `s`. We can easily convince ourselves that by initializing ∇`x(i)` to the ith canonical unit vector of length n, on exit ∇`y(i)` contains the gradient $\frac{\partial y(i)}{\partial x(1:n)}$, $i = 1, 2$. In this case, each statement involving a derivative object is really a vector instruction involving n-vectors. On the other hand, if we are interested only in sensitivities with respect to `x(3)`, say, then each ∇`x(i)` becomes a scalar rather than a vector, and we initialize ∇`x(i) = 0.0` for $i \neq 3$ and ∇`x(3)`

```
y(1) = 1.0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      temp = x(i) ◇ y(1)
      y(1) = temp ◇ y(1)
   else
      temp = x(i) ◇ y(2)
      y(2) = temp ◇ y(2)
   endif
enddo
```

Figure 2: Sample Code Fragment of Figure 1 Modified in Preparation for Forward-Mode Code Generation

```
∇y(1) = 0
y(1) = 1.0
∇y(2) = 0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      ∇temp = ∇x(i) + ∇y(1)
      temp = x(i) + y(1)
      ∇y(1) = ∇temp + ∇y(1)
      y(1) = temp + y(1)
   else
      ∇temp = ∇x(i) + ∇y(2)
      temp = x(i) + y(2)
      ∇y(2) = ∇temp + ∇y(2)
      y(2) = temp + y(2)
   endif
enddo
```

Forward Mode for $\diamond = +$

```
∇y(1) = 0
y(1) = 1.0
∇y(2) = 0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      ∇temp = y(1)*∇x(i) + x(i)*∇y(1)
      temp = x(i) * y(1)
      ∇y(1) = y(1)*∇temp + temp*∇y(1)
      y(1) = temp * y(1)
   else
      ∇temp = y(2)*∇x(i) + x(i)*∇y(2)
      temp = x(i) * y(2)
      ∇y(2) = y(2)*∇temp + temp*∇y(2)
      y(2) = temp * y(2)
   endif
enddo
```

Forward Mode for $\diamond = *$

Figure 3: Derivative Code Generated from the Code Fragment of Figure 2 by Using the Forward-Mode Approach

`= 1.0`. In this case, then, each statement involving a derivative object is a scalar instruction, and we emerge with ∇`y(i)` $= \dfrac{\partial\, y(i)}{\partial\, x(3)}$, $i = 1, 2$. In general, if we view the derivative vectors ∇ as row vectors, the linearity of differentiation implies that the forward mode allows us to compute arbitrary linear combinations of columns of the Jacobian

$$\frac{d\,y}{d\,x} = \begin{pmatrix} \frac{\partial\, y(1)}{\partial\, x(1)} & \cdots & \frac{\partial\, y(1)}{\partial\, x(n)} \\ \frac{\partial\, y(2)}{\partial\, x(1)} & \cdots & \frac{\partial\, y(2)}{\partial\, x(n)} \end{pmatrix}, \tag{5}$$

in that

$$\begin{pmatrix} \nabla y(1) \\ \nabla y(2) \end{pmatrix} = \frac{d\,y}{d\,x} * \begin{pmatrix} \nabla x(1) \\ \vdots \\ \nabla x(n) \end{pmatrix}. \tag{6}$$

In particular, if, for some n-vector d, we initialize ∇`x(i) = d(i)`, we compute the directional derivative

$$\frac{d\,y}{d\,x} * d = \lim_{h \to 0} \frac{y(x + h * d) - y(x)}{h}. \tag{7}$$

Forward mode code is easy to generate, logically preserves parallelizable or vectorizable structures within the original code, and is readily generalized to higher-order derivatives [12][§]. If we wish to compute p directional derivatives, then running forward-mode code requires at most on the order of p times as much time and memory as the original code.

2.2 The Reverse Mode

In contrast, the so-called reverse mode of automatic differentiation computes adjoint quantities — the derivative of the final result with respect to an intermediate quantity. To propagate adjoints, we have to be able to reverse the flow of the program, and remember or recompute any intermediate value that nonlinearly impacts the final result.

Let $\overline{s}$ denote the adjoint of a particular variable s. As a consequence of the chain rule it can be shown (see, for example, [46]) that the statement $s = f(v, w)$ in the original code implies that code of the form

$$\begin{aligned} \overline{v} \quad &+= \quad \frac{\partial\, s}{\partial\, v}\overline{s} \\ \overline{w} \quad &+= \quad \frac{\partial\, s}{\partial\, w}\overline{s} \end{aligned} \tag{8}$$

should be generated in the reverse mode code. The notation `a += b` is shorthand for `a = a + b`. When f is a linear elementary operation such as addition, $\dfrac{\partial\, s}{\partial\, v} = \dfrac{\partial\, s}{\partial\, w} = 1$, and hence $\dfrac{\partial\, s}{\partial\, v}$ and $\dfrac{\partial\, s}{\partial\, w}$ do not depend on the values of their operands. On the other hand, when f is a nonlinear operation such as a multiplication, both $\dfrac{\partial\, s}{\partial\, v}$ and $\dfrac{\partial\, s}{\partial\, w}$ do depend on the values of their operands, and one must remember either these derivative

[§] Although forward-mode code preserves vectorizable structures within the code, since it inserts additional vectorizable loops into the code, vectorizing compilers may have a difficult time choosing the proper loop level to vectorize.

```
y(1) = 1.0; y(2) = 1.0;
y1value(0) = y(1); c1 = 0;
y2value(0) = y(2); c2 = 0;
do i = 1,n
   if (x(i) > 0.0) then
      jump(i) = 'left'; c1 = c1 + 1;
      tempvalue1(c1) = x(i) ◇ y1value(c1-1)
      y1value(c1) = tempvalue1(c1) ◇ y1value(c1-1)
   else
      jump(i) = 'right'; c2 = c2 + 1;
      tempvalue2(c2) = x(i) ◇ y2value(c2-1)
      y2value(c2) = tempvalue2(c2) ◇ y2value(c2-1)
   endif
enddo
y(1) = y1value(c1); y(2) = y2value(c2);
```

Figure 4: Code Fragment of Figure 2 Modified in Preparation for Reverse-Mode Code Generation

values or the values of the operands. To be able to reverse the flow of the program, one must also remember intermediate values that were overwritten, and trace how branches were taken.

Once we have transformed the code to consist only of elementary unary and binary operations (Figure 2), we introduce trace arrays to record the branch history in the "`jump`" array, and save intermediate values of the variables `y(1)`, `y(2)`, and `temp` in `y1value(:)`, `y2value(:)`, `tempvalue1(:)` and `tempvalue2(:)`. Counters `c1` and `c2` are used to point to the last value set in each of the branches of the if-statement inside the loop. The resulting code is shown in Figure 4. We are now in a position to automatically generate reverse mode code for this computation, employing the recipe described in (8). The result is shown in Figure 5. We use the notation $\overline{\mathtt{s}}$ to denote the adjoint object associated with the program variable `s`.

We can easily convince ourselves that when we initialize $\overline{\mathtt{y(1)}} = 1.0$, $\overline{\mathtt{y(2)}} = 0.0$ and all other adjoint objects to zero, then by running the codes in Figures 4 and 5, we emerge with $\overline{\mathtt{x(i)}} = \frac{\partial\, x(i)}{\partial\, y(1)}$. Similarly, initializing $\overline{\mathtt{y(1)}} = 0.0$, $\overline{\mathtt{y(2)}} = 1.0$, and all other adjoint objects to zero, we compute $\overline{\mathtt{x(i)}} = \frac{\partial\, x(i)}{\partial\, y(2)}$. In general, if we view the adjoint vector associated with a program variable as column vector, the linearity of differentiation implies that

$$\left(\overline{\mathtt{x(1)}}, \cdots, \overline{\mathtt{x(n)}}\right) = \left(\ \overline{\mathtt{y(1)}}\ ,\ \ \overline{\mathtt{y(2)}}\ \right) * \frac{d\,y}{d\,x}, \tag{9}$$

where $\frac{d\,y}{d\,x}$ is as defined in Equation (5). That is, reverse mode code allows us to compute arbitrary linear combinations of the rows of the Jacobian. If, for some vector d, we initialize $\overline{\mathtt{y(i)}}$ `= d(i)`, we compute the

```
y2value(c2) = y(2); y1value(c1) = y(1);
do i = n to 1 step -1
   if (jump(i) = 'left') then
      y1value(c1 - 1) += y1value(c1)
      tempvalue1(c1) += y1value(c1)
      x(i) += tempvalue1(c1)
      y1value(c1 - 1) += tempvalue1(c1)
      c1 = c1 - 1
   else
      y2value(c2 - 1) += y2value(c2)
      tempvalue2(c2) += y2value(c2)
      x(i) += tempvalue2(c2)
      y2value(c2 - 1) += tempvalue2(c2)
      c2 = c2 - 1
   endif
enddo
```

Reverse Mode for $\diamond = +$

```
y2value(c2) = y(2); y1value(c1) = y(1);
do i = n to 1 step -1
   if (jump(i) = 'left') then
      y1value(c1 - 1) += tempvalue1(c1)*y1value(c1)
      tempvalue1(c1) += y1value(c1-1)*y1value(c1)
      x(i) += y1value(c1-1)*tempvalue1(c1)
      y1value(c1 - 1) += x(i)*tempvalue1(c1)
      c1 = c1 - 1
   else
      y2value(c2 - 1) += tempvalue2(c2)*y2value(c2)
      tempvalue2(c2) += y2value(c2-1)*y2value(c2)
      x(i) += y2value(c2-1)*tempvalue2(c2)
      y2value(c2 - 1) += x(i)*tempvalue2(c2)
      c2 = c2 - 1
   endif
enddo
```

Reverse Mode for $\diamond = *$

Figure 5: Derivative Code Generated from Code in Figure 4 by Using the Reverse-Mode Approach

derivative

$$\frac{\partial\,(d^T * y(x))}{\partial\, x}. \tag{10}$$

Note that it is a much more involved process to generate reverse mode code. While the complexity of the forward-mode code generation in Figure 3 changed minimally when we considered an addition instead of a multiplication, the reverse mode code is very sensitive to this change: there is no need to save the intermediate values of `y(1)`, `y(2)`, or `temp` when $\diamond = +$, but we must save them when $\diamond = *$, at the expense of an extra $O(n)$ memory locations. Extra storage is required to remember the way the branches were taken, regardless of whether the loop computed a multiplication or an addition. Hence, the reverse mode can, in extreme cases, require as much memory for the tracing of intermediate values and branches as there are floating-point operations being executed during the run of the program. However, its running time is roughly m times that of the function when computing m linear combinations of the rows of the Jacobian. This is particularly advantageous for gradients, since then $m = 1$.

2.3 The ADIFOR Approach

There have been various implementations of automatic differentiation; an extensive survey can be found in [54]. We are mainly interested in "black-box" tools for automatic differentiation—tools that, given the source code and an indication of which variables correspond to the independent and dependent variables

```
∇y(1) = 0
y(1) = 1.0
∇y(2) = 0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      y1bar = 1.0 + 1.0
      ∇y(1) = y1bar*∇y(1) + ∇x(i)
      y(1) = y(1) + x(i) + y(1)
   else
      y2bar = 1.0 + 1.0
      ∇y(2) = y2bar*∇y(2) + ∇x(i)
      y(2) = y(2) + x(i) + y(2)
   endif
enddo
```

ADIFOR Approach for $\diamond = +$

```
∇y(1) = 0
y(1) = 1.0
∇y(2) = 0
y(2) = 1.0
do i = 1,n
   if (x(i) > 0.0) then
      temp = x(i) * y(1)
      res = temp*y(1)
      xibar = y(1) * y(1)
      y1bar = temp + y(1) * x(i)
      ∇y(1) = y1bar*∇y(1) + xibar*∇x(i)
      y(1) = res
   else
      temp = x(i) * y(2)
      res = temp*y(2)
      xibar = y(2) * y(2)
      y2bar = temp + y(2) * x(i)
      ∇y(2) = y2bar*∇y(2) + xibar*∇x(i)
      y(2) = res
   endif
enddo
```

ADIFOR Approach for $\diamond = *$

Figure 6: Derivative Code for the Code Fragment of Figure 1 Generated by Using the ADIFOR Approach

with respect to differentiation, generate derivative code without further user intervention. Black-box tools in this sense are GRESS [51], PADRE-2 [58] and Odyssee [65, 66] for Fortran programs and ADOL-C [45] and ADIC [16] for C programs. AMC [37], on the other hand, is a tool that supports, in an interactive fashion, the generation of reverse mode code.

GRESS, PADRE-2, and ADOL-C implement both the forward and reverse mode. To save control flow information and intermediate values, these tools generate a "trace" of the computation by writing down the particulars of every operation performed in the code. The interpretation overhead associated with using this trace for the purposes of automatic differentiation and its potentially very large size can be a serious computational bottleneck [67].

ADIFOR, Odyssee, and ADIC take a "source transformation" approach to automatic differentiation. By applying the rules of automatic differentiation, these tools rewrite the original code, inserting statements for the computation of first-order derivatives. Odyssee is the only tool that generates full reverse mode code, and it has been used successfully for the adjoint generation of weather models [65, 66]. It imposes certain restrictions on the Fortran input and on the Fortran runtime environment (e.g., the support of "automatic arrays"). The potential storage explosion associated with applying the reverse mode to highly nonlinear codes has not been addressed in Odyssee yet, but the snapshotting approach suggested in [43] has great potential.

ADIFOR and, more recently, ADIC employ a hybrid forward/reverse mode scheme, and the basic approach taken in ADIFOR 2.0 is unchanged from that of previous versions of ADIFOR. In essence, for each statement, we accumulate the partial derivatives of the variable on the left-hand side with respect to the variables on the right-hand side, and then apply the forward mode to propagate the total derivatives according to the chain rule. The results of this approach for the code of Figure 1 are shown in Figure 6.

For example, the code fragment

```
xibar = y(1) * y(1)
y1bar = temp + y(1) * x(i)
```

is a "cleaned-up" version of the vanilla reverse mode code to compute $\frac{\partial\, y(1)_{new}}{\partial\, x(i)}$ and $\frac{\partial\, y(1)_{new}}{\partial\, y(1)_{old}}$ for $\diamond = *$ which is shown in Figure 7. Note that the cost of computing these "statement derivatives" is amortized

```
resbar = 1.0;
tempbar = y1bar = xibar = 0.0;
y1bar += temp * resbar
tempbar += y1 * resbar
xibar += y1 * tempbar
y1bar += x(i) * tempbar
```

Figure 7: Reverse Mode Code for `y(1) = x(i)*y(1)*y(1)`

over all the derivatives being computed, and hence this approach is more efficient than the normal forward mode or a divided-difference approximation when more than a few derivatives are computed at the same time. Unlike the reverse mode, which is optimal for gradients, but not for general Jacobians, this approach performs well (compared with divided-difference approximations) for a wide variety of problems, and, like the forward mode, it has predictable storage and runtime requirements.

We also see that, from a user's perspective, the ADIFOR-generated code provides the directional derivative computation possibilities associated with the forward mode of automatic differentiation [15]. Instead of simply producing code to compute the Jacobian J, ADIFOR produces code to compute $J * S$, where the "seed matrix" S is initialized by the user. Thus, if S is the identity, ADIFOR computes the full Jacobian; whereas if S is just a vector, ADIFOR computes the product of the Jacobian by a vector. A derivative object ∇`y(1)`, say, contains the derivatives of the scalar `y(1)` with respect to all directions specified in the seed matrix. We call such a vector a *directional gradient vector*, and such a vector is associated with every scalar variable for which we propagate derivatives.

The cost of derivative computation is more or less proportional to the number p of directional derivatives (equal to the number of columns of S) that are computed in one run. Hence, computing a Jacobian-vector product is much less expensive than computing the Jacobian itself. Typically (see the references mentioned in Section 1), ADIFOR-generated code runs two to four times faster than one-sided divided difference approximations when one computes more than 5–10 derivatives at one time. The explanation lies in the hybrid approach and a dependence analysis that tries to avoid computing derivatives of expressions that do not affect the dependent variables (see Subsection 3.1).

The seed matrix mechanism allows for flexible use of ADIFOR-generated code. For example, it can be employed to compute compressed versions of large sparse Jacobians [4], to chain derivatives generated by programs running on different platforms [9, 27], or to decrease turnaround time for derivative computations through a parallel stripmining approach [13].

3 The ADIFOR 2.0 System

The ADIFOR 2.0 system has three major components:

ADIFOR 2.0 preprocessor: The ADIFOR 2.0 preprocessor parses the code, performs certain code normalizations, determines which variables have to be augmented with derivative objects, and generates derivative code with templates at call sites of Fortran 77 intrinsics and, if desired, calls to SparsLinC routines.

ADIntrinsics system: The ADIntrinsics system expands calls to Fortran 77 intrinsic templates to Fortran 77 code guided by a template library defining how each intrinsic is to be translated.

SparsLinC library: The SparsLinC library provides transparent support of sparsity in derivative computations.

The relationship among these components is shown pictorially in Figure 8, and they are described in detail in the following subsections. We point out beforehand that the new ADIFOR 2.0 preprocessor and ADIntrinsics system have correctly handled the 60,000-line CAMRAD helicopter hover code [53] at NASA Langley. Researchers at Langley have verified the derivatives, and we believe this to be a new record in automatic differentiation.

3.1 The ADIFOR 2.0 Preprocessor

ADIFOR takes a source transformation approach to automatic differentiation. That is, in order to augment a given code with derivative computation, we rewrite it, using the principle outlined in the preceding section, generating a new Fortran code that, when compiled and executed, computes derivatives. Compared with implementing automatic differentiation with operator overloading (see, for example, [54]) a source translation approach allows one to view the problem of generating derivative code in a context that is larger than one arithmetic operation, and is the conceptual key to the development of hybrid modes like the one employed in ADIFOR.

In order to be in a position to rewrite "real life" Fortran codes, it is advantageous to base an automatic differentiation tool on existing compiler infrastructure. In this fashion, one can quickly gain access to means for constructing and manipulating an abstract representation of the program. Moreover, one is able to "logically retarget" techniques developed in the compiler community to reason about Fortran programs and generate efficient derivative code. ADIFOR employs compiler infrastructure provided in the ParaScope programming environment [26], which was developed primarily for the semi-automatic parallelization of Fortran programs, and the D system [2], a collection of tools for programming in the Fortran D data

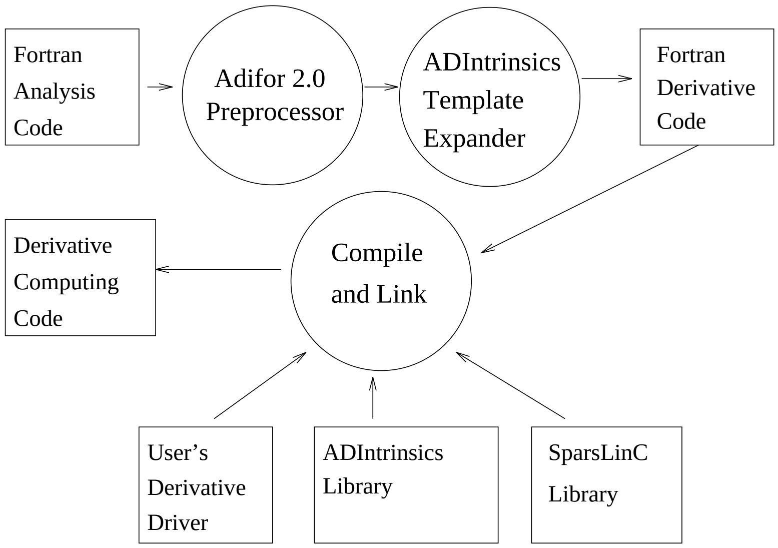

Figure 8: Overview of ADIFOR 2.0 System

parallel language. While our primary goal is not the parallelization of Fortran programs, this compiler infrastructure provides us with a Fortran parser, data abstractions for representing Fortran programs, and tools for constructing and manipulating those representations and for gathering a variety of data flow facts for scalars and arrays, as well as control flow information.

In this section, we describe the ADIFOR preprocessor, which, given an indication of independent and dependent variables, generates a "templatized" Fortran 77 version of the derivative code which will then be processed by the ADIntrinsics system. The preprocessor accomplishes the following tasks:

Code Canonicalization: The original code is rewritten in a fashion that allows for automatic differentiation.

Variable Nomination: We have to decide which variables need to have an associated derivative object. Loosely speaking, any variable whose value could depend on the value of an independent variable and could influence the value of a dependent variable must have a derivative object.

Derivative Code Generation: Derivative code is generated according to the ADIFOR hybrid approach.

We briefly describe these tasks in the next subsections.

3.1.1 Code Canonicalization

In the code canonicalization phase, the Fortran code at hand is, in essence, rewritten to conform to certain standards. For example, expressions appearing as arguments to function or subroutine calls and function calls appearing within conditional tests are hoisted into assignments to new temporary variables. Statement functions are expanded into in-line code. This phase also breaks up long right-hand sides of assignment statements into smaller pieces, and rewrites them such that all variables appearing on the right-hand side of an assignment statement are of the same type. The latter transformation is needed for the code to be able to link in the SparsLinC (see Subsection 3.3) library.

This phase was not present in ADIFOR 1.0, and the occurrence of any of these features in the code required the user to rewrite the code by hand. The canonicalization phase now also supports common Fortran extensions, such as `INCLUDE`, `DOUBLE COMPLEX`, `DO-ENDDO`, and `IMPLICIT NONE` statements. In fact, the only Fortran 77 features that are not supported in ADIFOR 2.0 are Fortran 77 intrinsics passed as procedure parameters, the overriding of Fortran 77 intrinsics by external functions, and I/O statements that contain function invocations. The occurrence of such a statement is, however, flagged by ADIFOR.

3.1.2 Variable Nomination

We associate a derivative object (denoted by ∇ symbols in Figure 6), with every variable whose value may depend on the value of a variable considered "independent" with respect to differentiation, and whose value impacts a variable considered "dependent" with respect to differentiation. Such a variable is called *active.* Variables that do not require derivative information are called *passive.* The easiest solution to this variable nomination problem is to make all variables active at a possibly large additional space penalty (for storage for unneeded derivative objects) and time penalty (for computation of derivative objects that do not depend on the independent variables or which do not impact the values of the dependent variables.)

ADIFOR tries to do better by employing interprocedural analysis techniques. First, it derives a "local interaction graph" for each subroutine. This is a bipartite graph where 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.

Next, an interprocedural analysis is performed, which determines, in essence, all possible program paths through which an independent variable can affect a dependent one and identifies intermediate variables that are involved along such a path. This analysis involves computing a transitive closure of the whole program graph composed from the local interaction graphs. In the presence of common blocks, equivalences, and arbitrary control structures, this is a nontrivial and compute-intensive process. Indeed, in our experience, the transitive closure computation is the most memory- and time-consuming part of the ADIFOR process.

In the ADIFOR 2.0 preprocessor, the dependency analysis code has been substantially improved compared with that in ADIFOR 1.0, In particular, ADIFOR 2.0 now may prune the local interaction matrix to ignore variables that are declared but never used (as if often the case when common blocks are defined in `include` files).

3.1.3 Code Generation

After active variables have been nominated, derivative code is generated for each statement containing an active variable, and derivative objects are allocated. For each statement, we first generate derivative code using the reverse mode approach as illustrated in Figure 7, and then clean up the code by folding constants and by then eliminating variables that are used only once. In this fashion we eliminate multiplications by 1.0 and additions to 0.0, and we reduce the number of variables that must be allocated. For assignment statements containing a Fortran intrinsic, a template is generated that will be instantiated by the ADIntrinsics system.

```
real function diff(x)
real x(3)
square(x) = x*x
diff = abs(x(1)*square(x(2))*x(3))
end
```

Figure 9: Sample Fortran Program

For example, if we designate "x" as an independent variable and "diff" as a dependent one, and if we set a limit of three on the number of directional derivatives that can be computed, by default the function `diff` in Figure 9 is translated into the code shown in Figure 10. In our experience, the code generated by ADIFOR is usually 2-3 times longer than the original code.

The ADIFOR 2.0 preprocessor now also provides a mechanism for customizing the derivative code. For example, one can suppress the leading dimensions of derivative objects, declare the number of directional derivatives to be a constant instead of a runtime parameter, or omit the loops when one is interested only in computing one directional derivative, as, for example, a Jacobian*vector product. By judicious use of these options, one can have ADIFOR generate code that is somewhat less general but may be compiled into faster code on the computer platform at hand, in particular on vector and superscalar platforms. As an example, if we are interested only in computing one directional derivative, then by simply setting some of ADIFOR's variables (details are described in [8]), the ADIFOR 2.0 preprocessor generates the simpler code shown in Figure 11.

3.2 The ADIntrinsics System

Automatic differentiation is based on the application of the chain rule. It gives the correct answer, in the absence of floating-point exceptions, provided that all operators and functions are applied at arguments interior to their domains, so that the operators and functions are smooth in a neighborhood of the point of application. If this assumption is not satisfied, the results computed by an automatic differentiation tool cannot be guaranteed to constitute a valid derivative value.

Let us consider, for example, the situation that in the course of the execution of a program, we compute `z = max(0.0,x)` and the value of `x` happens to be zero. An automatic differentiation tool requires knowledge of $\frac{d\max(0.0,x)}{dx}|_{x=0}$, which is not defined. Other Fortran77 intrinsics that are not everywhere differentiable in their domain are `abs`, `sign`, `aint`, `min`, `dim`, and the power operator `**`. The ADIntrinsics system provides a mechanism to

- define a reasonable default behavior in cases where the derivative of a Fortran 77 intrinsic is not defined,
- provide an error-reporting mechanism that gives various levels of detail for the exceptions that occurred,
- allow the user to easily customize the exception handler, and
- easily extend it to handle new intrinsics.

```
function g_diff(g_p_, x, g_x, ldg_x, g_rres_, ldg_rres_)
real x(3)
integer g_pmax_
parameter (g_pmax_ = 3)
integer g_i_, g_p_, ldg_rres_, ldg_x
real g_diff, r1_p, r6_b, g_rres_(ldg_rres_), r5_b, r4_b, r3_w, r4_v, r2_v
real g_r1_w(g_pmax_), g_x(ldg_x, 3)
save g_r1_w
if (g_p_ .gt.  g_pmax_) then
   print *, 'Parameter g_p_ is greater than g_pmax_'
   stop
endif
r3_v = x(2) * x(2)
r4_v = x(1) * r3_v
r4_b = x(3) * r3_v
r5_b = x(3) * x(1)
r6_b = r5_b * x(2) + r5_b * x(2)
do g_i_ = 1, g_p_
   g_r1_w(g_i_) = r4_b*g_x(g_i_,1) + r6_b*g_x(g_i_,2) + r4_v*g_x(g_i_,3)
enddo
r1_w = r4_v * x(3)
call AD_INTRINSIC_FIRST_ABS_S(r1_w, r2_v, r1_p)
do g_i_ = 1, g_p_
   g_rres_(g_i_) = r1_p * g_r1_w(g_i_)
enddo
g_diff = r2_v
end
```

Figure 10: Derivative Code Generated by ADIFOR 2.0 Preprocessor from the Code in Figure 9 by Using Default Settings

```
function g_diff(x, g_x, g_rres_)
real x(3)
real g_rres_, g_diff, r1_p, r2_v, r1_w, r2_w, r3_w, g_r2_w, g_x(3), g_r3_w
real g_r1_w
save g_r2_w, g_r3_w, g_r1_w
g_r2_w = (x(2) + x(2)) * g_x(2)
r2_w = x(2) * x(2)
g_r3_w = r2_w * g_x(1) + x(1) * g_r2_w
r3_w = x(1) * r2_w
g_r1_w = x(3) * g_r3_w + r3_w * g_x(3)
r1_w = r3_w * x(3)
call AD_INTRINSIC_FIRST_ABS_S(r1_w, r2_v, r1_p)
g_rres_ = r1_p * g_r1_w
g_diff = r2_v
end
```

Figure 11: Derivative Code Generated by ADIFOR 2.0 Preprocessor from the Code in Figure 9 by Using Settings Appropriate for a Jacobian*Vector Product

Tup compute the elementary derivative of a Fortran 77 intrinsic, the ADIFOR preprocessor inserts a call to an *intrinsic template*. For example, the statement `z = max(0.0,x)`, where `x` and `z` are declared as `REAL`s, is translated into

```
call AD_INTRINSIC_FIRST_MAX_S(0.0,x,r2_v,r1_p,r2_p)
do g_i_ = 1,g_p_
       g_z(g_i_) = r2_p * g_x(g_i_)
enddo
z = r2_v
```

The ADIntrinsics system then takes care of translating the `AD_INTRINSIC_FIRST_MAX_S` call into legal Fortran 77 code which provides the value of the intrinsic in `r2_v`, the value of the partial derivative with respect to its first argument in `r1_p` and the value of the partial derivative with respect to its second argument in `r2_p`. Note that neither $\dfrac{\partial \max(x,y)}{\partial x}$ nor $\dfrac{\partial \max(x,y)}{\partial y}$ are defined when $x = y$. As another example, the `AD_INTRINSIC_FIRST_ABS_S` call is inserted to deal with the call to `abs()`, since $\dfrac{d\,|x|}{d\,x}$ is not defined for $x = 0$.

The ADIntrinsics system has three main components:

The Purse Preprocessor: Purse translates `AD_INTRINSIC` calls into legal Fortran 77, governed by the level of exception handling desired.

Template files for all Fortran 77 intrinsics: Purse uses template files as blueprints for how to expand `AD_INTRINSIC` calls. These files can be easily be customized by the user.

The Error Handler library: A collection of Fortran 77 routines is used to record and report runtime errors and to change certain default values.

To illustrate the workings of the ADIntrinsics system, we demonstrate the expansion of the template call `AD_INTRINSIC_FIRST_MAX_S`. A more detailed description can be found in [8].

The translation of an `AD_INTRINSIC` call into Fortran 77 is governed by a *template file*. The template file provided as a default for `max` is shown in Figure 12. The template specifies how the function value is to be computed (`z = max(x,y)`) and how the first-order partials `fx` and `fy` or the second-order partials `fxx`, `fxy`, and `fyy` are computed. It also defines the behavior in "performance mode" and identifies when to invoke an error handler in the other error-reporting modes. While ADIFOR 2.0 does not generate second-order derivative code yet, we expect to add this capability soon; we have already provided for it in the ADIntrinsics system.

Purse uses this file when translating the `AD_INTRINSIC_FIRST_MAX_S` call, governed by the level of error reporting desired. The ADIntrinsics system currently provides terse, counting, and verbose modes:

Performance Mode: Points of nondifferentiability are not checked. `AD_INTRINSIC_FIRST_MAX_S` is translated into

```
r2_v = max (0.0, x)
if (0.0 .gt.  x) then
   r1_p = 1.0e0
   r2_p = 0.0e0
else if (0.0 .lt.  x) then
   r1_p = 0.0e0
   r2_p = 1.0e0
else
   r1_p = 0.5e0
   r2_p = 0.5e0
endif
```

Note that the constants were instantiated with the right type, and no code was generated for evaluating the second-order partial derivatives. At the point of nondifferentiability, both partials are set to 0.5, and no warning is generated. This "tie value" constitutes a subgradient. The rationale for choosing default values for the different intrinsics is provided in [10] and [17].

Terse Mode: At the point of nondifferentiability, an exception handler is called:

```
C     PERFORMANCE
      z = max (x,y)

#ifndef PERFORMANCE
      if (x .gt.  y) then
         fx = TYPE(1.0)
         fy = TYPE(0.0)
      else if (x .lt.  y) then
         fx = TYPE(0.0)
         fy = TYPE(1.0)
      else
         call EXCEPTION_HANDLER
         fy = TYPE(1.0) - fx
      endif
#else
      if (x .gt.  y) then
         fx = TYPE(1.0)
         fy = TYPE(0.0)
      else if (x .lt.  y) then
         fx = TYPE(0.0)
         fy = TYPE(1.0)
      else
c stop 'ADIFOR Exception:  x = y in max(x,y).'
C This is the current value of TieVal
         fx = TYPE(0.5)
         fy = TYPE(0.5)
      endif
#endif
C     PERFORMANCE
      fxx = TYPE(0.0)
C     PERFORMANCE
      fxy = TYPE(0.0)
C     PERFORMANCE
      fyy = TYPE(0.0)
```

Figure 12: Template File for `max` Intrinsic

```
r2_v = max (0.0, x)
if (0.0 .gt.  x) then
   r1_p = 1.0e0
   r2_p = 0.0e0
else if (0.0 .lt.  x) then
   r1_p = 0.0e0
   r2_p = 1.0e0
else
   call ehbfST (7,0.0, x, r2_v, r1_p, r2_p)
   r2_p = 1.0e0 - r1_p
endif
```

When $x = 0$, the terse error handler is invoked, which sets a flag that an exception for the `max` intrinsic occurred (exception no. 7) and returns exceptional values for this occurrence. If, after executing the ADIFOR-generated code, the user calls the `ehrpt` error handler reporting subroutine, and `ehbfST` had been invoked at some point, the message

```
Exception(s) occurred evaluating MIN
```

is generated.

Counting Mode: The translation is like in terse mode, except that the error handler call is now

```
call ehbfSC (7,0.0, x, r2_v, r1_p, r2_p)
```

If `max` was evaluated seven times, say, with both its arguments being equal in the course of the evaluation of the ADIFOR-generated code, then a call to `ehrpt` would result in the message

```
Exception(s) occurred evaluating MIN: 7 times
```

We have both a terse and a counting mode because the counter recurrence required for determining the number of occurrences of each exception inhibits vectorization.

Verbose Mode: The call to the error handler is now something like

```
call ehbfSV (7,0.0, x, r2_v, r1_p, r2_p,'g_main.f',133)
```

where `g_main.f` is the name of the file containing this call on line 133. When this subroutine is called, it results in a message

```
Exception:  MAX ( 0.  0.)
Occurred in g_main.f at line # 133
```

While verbose mode provides the most detailed information, we found that the string handling associated with the file name inhibited compiler optimization on some platforms.

The ADIntrinsics error handler library provides routines for keeping track of and reporting errors as well as changing the default error values, or the error reporting unit.

We are also working on a "report-once" mode. Instead of printing a message every time an exception occurs, this mode will provide summary information about how often the exception at line 133 in file `g_main.f` occurred.

The template mechanism also makes it easy to specify a different way for handling exceptional situations. Say, for example, that the reason for inserting the `max(0.0,x)` call was to ensure that floating-point roundoff errors did not result in a small negative value for a quantity that physically cannot be negative, for example, some energy value. Then we might like to specify that, when the first argument of `max` is zero, we always would like to set the partial of `max` with respect to the second argument to 1. We can easily do this by copying the `max` template above into a file `mymax.T`, say (the .T extension denotes template files), and adding the branch

```
if (x .eq.  TYPE(0.0)) then
   fx = TYPE(0.0)
   fy = TYPE(1.0)
elseif (x .gt.  y) then
...
```

at the beginning of the if-statements. By adding the comment

```
C       AD_EXCEPTION_OVERRIDE_INTRINSIC(MAX,MYMAX)
```

before the call to `max` in the user's code, Purse will then substitute the modified exception handler for dealing with exceptions of this call.

A new intrinsic can be easily supported by adding the name of the new intrinsic to Purse and providing a template file for it. In this fashion, a user of ADIFOR 2.0 has complete control over the handling and reporting of exceptional derivative occurrences. We also note that in the ADIFOR 2.0 system, both the ADIFOR preprocessor and Purse are called in a fashion that is transparent to the user.

3.3 The SparsLinC (Sparse Linear Combination) Library

The workhorse of ADIFOR-generated code (or any other mainly forward-mode first-order automatic differentiation approach) is a "vector linear combination" (e.g., ∇`y(1) = y1bar*`∇`y(1) + xibar*`∇`x(i)` in Figure 6). Here ∇`y(1)` is a vector of length p, where, as in Subsection 2.3, p denotes the number of directional derivatives to be computed, and `y1bar` is a scalar. This operation is a particular instantiation of

$$w = \sum_{i=1}^{k} \alpha_i * v_i, \tag{11}$$

where w and v_i are vectors of length p, the α_i are scalar multipliers, and k is referred to as the "arity." If we choose `AD_FLAVOR = dense`, which is the default, this vector operation is expressed as a Fortran vector loop, e.g.,

```
do g_i_ = 1, g_p_
       g_y(g_i_, 1) = y1bar * g_y(g_i_,1) + xibar * g_x(g_i_,i)
enddo
```

As long as p is moderate, this is an efficient way of expressing a vector linear combination.

The SparsLinC (Sparse Linear Combination) Library addresses the scenario where p is large and most of the vectors involved in vector linear combination are sparse, that is, for the most part they contain zero entries. This situation arises, for example, in the computation of large sparse Jacobians, $J := \frac{dF}{dx}$, or gradients of so-called partially separable functions [47], which are functions f that can be represented in the form

$$f(x) = \sum_{i=1}^{np} f_i(x), \tag{12}$$

where each of the component functions f_i has limited support. Hence, the gradients ∇f_i are sparse, even though the final gradient ∇f is dense. Partially separable functions play a key role in large-scale optimization (for example, all minimization examples in [3] belong to that class), and, in particular, any function with a sparse Hessian is a partially separable one [47].

If the sparsity pattern of J is known, coloring techniques together with the seed matrix mechanism can be employed advantageously to compute a compressed Jacobian matrix efficiently [4]. The computation of the gradient of a partially separable function can be reduced to the problem of computing a sparse Jacobian [20] by realizing that the gradient of f can easily be obtained by summing the rows of the sparse Jacobian $\frac{dG}{dx}$, where

$$G(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_{np}(x) \end{pmatrix}. \tag{13}$$

The compressed Jacobian approach works well for sparse Jacobians under the following assumptions:

- The sparsity pattern of the Jacobian is known.
- The chromatic number of the Jacobian is close to the maximal number of nonzeros in any row, in particular, the Jacobian does not contain a row that can be considered dense.
- The component functions f_i of f are readily accessible.

Note, however, that sparsity is inherent in those problems, no matter how the code is actually formulated. If the initial seed matrix is sparse (e.g., the identity), then, ignoring exact numerical cancellation, the left hand side vector w in (11) has no fewer nonzeros than any of the vectors on the right hand side. Hence, if the final derivative objects, which correspond to a row of the Jacobian J or a component gradient ∇f_i, are sparse, it is very likely that all intermediate vectors are sparse as well. That is, by replacing the dense vector loop as a way of expressing the derivative linear combinations with algorithms and data structures tailored towards exploiting sparsity, we can exploit sparsity in a transparent fashion, even if none of the assumptions for the coloring approach are met. Also note that the sparsity structure of J or ∇f_i is computed as a byproduct of the derivative computation.

The SparsLinC library provides this support for sparse vector linear combination, in a fashion that is well suited to the use of this operation in the context of automatic differentiation. SparsLinC, which is written in ANSI C, includes the following features:

Three data structures for sparse vectors: SparsLinC has different data structures for a vector containing only one nonzero, a few nonzeros, or several nonzeros. In the numerical Linear algebra literature, the latter two data structures are usually referred to as the "single-subscript" and "compressed subscript" representation of a sparse vector (see, for example, [32, 36]).

Efficient Memory Allocation Scheme: SparsLinC employs a "bucket" memory allocation scheme, which in effect provides a buffered memory allocation mechanism, supporting the dynamic nature of the sparse vectors while avoiding the need for system calls most of the time.

Polyalgorithms: SparsLinC switches between vector representations in a transparent fashion and provides special support for the "+=" operation $w = \alpha_1 * w + \alpha_2 * v$, which occurs frequently when computing gradients of partially separable functions, as suggested by (12).

Full-Precision Support: single- and double-precision routines are provided for both real- and complex-valued computations.

In this fashion, SparsLinC can adapt to the dynamic nature of the derivative vectors, efficiently representing derivative vectors that grow from a column of the identity matrix (often occurring in the ADIFOR seed matrix) to a dense vector, such as ∇f in (12). We also mention that almost no memory is allocated for derivative objects that are all zeros. Hence, SparsLinC effectively complements the ADIFOR dependence analysis, which has to make conservative assumptions about what variables are considered active and may, as a result, activate a variable that is not on the computational path from independent to dependent variables. This situation may arise, for example, for large arrays where different portions of the array represent logically different entities.

When invoked with the `AD_FLAVOR = sparse` option, ADIFOR allocates an integer, instead of a vector, for each derivative object, and generates calls to the SparsLinC library to perform the vector linear combinations, for example,

```
call sspg2q(g_y(1),y1bar,g_x(i),xibar,g_y(1))
```

The Fortran interface of SparsLinC is described in [7]. The routines needed for initialization or extraction of data, as well as examples of the use of SparsLinC in the context of ADIFOR 2.0, are provided in [8].

In addition to ease of use, SparsLinC can result in significant performance improvement. For example, we computed gradients for problems from the MINPACK-2 optimization test set [3] where the function was partially separable but was not specified in partially separable form (13). The four codes were

- DGL2: 2-D Ginzburg-Landau model for homogeneous superconductors,
- DMSA: minimal surface area problem,
- DSSC: steady-state combustion model, and
- DEPT: elastic-plastic torsion problem.

We computed derivatives using both the dense and SparsLinC-supported approach. The improvement in run time achieved on an IBM RS 6000-370 workstation through SparsLinC is shown in Figure 13. The execution time of the "dense" routines was extrapolated from the largest problem that we could actually fit onto the machine before running out of memory. We also mention that the derivatives computed with both

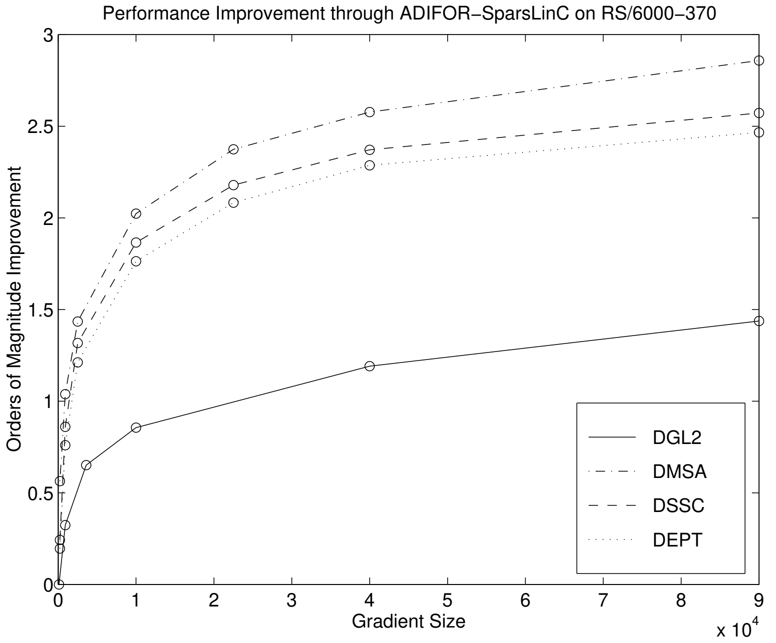

Figure 13: Runtime Improvement in Computing MINPACK-2 Gradients by Using SparsLinC

the "dense" and SparsLinC-supported ADIFOR-generated derivative code agreed with the values produced by the handcoded routines up to machine precision.

We see that, on some problems, we have achieved almost three orders of magnitude improvement over the "dense do-loop" version of the derivative code. We also note that SparsLinC greatly reduces the memory requirements of the code. Instead of requiring n times as much memory for computing a gradient of size n, SparsLinC will allocate only as much memory as is needed, which, in our examples, involves at most three dense vectors of length n into which gradients are accumulated, and only short vectors with at most nine nonzeros for all other derivative objects.

In summary, SparsLinC provides a mechanism for exploiting sparsity in derivative computations in a very easy fashion. The user need not have any knowledge about the particular sparsity structure of the problem. The SparsLinC routines efficiently adapt to the particular situation at hand, providing efficient support for a wide variety of sparsity scenarios. We also mention that, since the sparsity structure of the Jacobian is a byproduct of the derivative computation, we are now in a position to develop, for example, a novel kind of large-scale optimization environment for nonlinear equations, where the user need supply code only for "the function," and automatic differentiation provides both the derivative values and the location of the nonzeros of the Jacobian for a sparse solver, without any user intervention [21].

4 Conclusions and Future Work

The use of ADIFOR in various application domains has demonstrated that automatic differentation

- is applicable to arbitrary codes,
- provides reliable derivatives, and
- can result in considerable speedups with respect to divided-difference approximations.

While these points were always clear to "automatic differentiation believers" and had to some extent been borne out by applications in limited domains, the lack of tools that provided *fast automatic differentiation capabilities for general Fortran 77 codes* effectively prohibited automatic differentiation from becoming part of the mainstream of scientific computing. The ADIFOR project has begun to change this by

- supporting almost all of Fortran 77, thereby lowering the technology accessibility threshold, as the need for code rewriting is reduced;
- employing advanced source transformation infrastructure and algorithms to handle and reason about large and not necessarily well-structured codes;
- employing an approach that resulted in lower complexity than divided differences, and expressing the code in a fashion such that computer run times bore out this complexity advantage;
- providing, through the use of Fortran 77 as an output language, the possibilities for users to become familiar with automatic differentiation through inspection of the generated code and to "tune" the generated code according to their needs; and
- actively promoting and supporting the use of automatic differentiation in applications.

The ADIFOR 2.0 system now goes a step beyond establishing the usefulness of the technology. By providing

- the capability to support all of Fortran 77 as well as common extensions,
- a runtime library for transparent exploitation of sparsity in derivative computations, and
- a completely customizable exception handler,

the ADIFOR 2.0 system affords a new level of ease and flexibility of use in a first-order automatic differentiation tool. While ADIFOR 1.0 was made available mainly through ADIFOR accounts at Argonne National Laboratory, the ADIFOR 2.0 system is intended to be widely distributed.

However, much remains to be done. We are working on extending the capabilities of ADIFOR to directly compute second-order derivatives. While automatic differentiation can easily be extended to arbitrary-order derivatives (see, for example, the ADOL-C [45] system), it is not clear which approach, or combination of approaches, results in the most efficient code. Some of these issues are discussed in [11] and [12].

We are also working on decreasing the complexity of computing first-order derivatives. While ADIFOR-generated code usually beats finite differences, it is usually not yet a match for a derivative code carefully derived by hand, in particular with respect to adjoint codes. To address this issue, we are beginning to

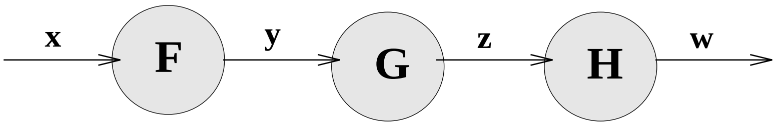

Figure 14: Serial Simulation

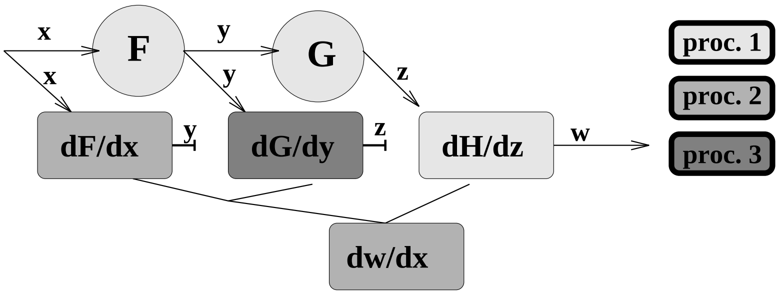

Figure 15: Parallel Derivative Computation

investigate how advanced compiler techniques could be employed to generate efficient adjoint code. As the small example in Figures 4 and 5 suggests, adjoint mode generation requires rather radical code restructuring and advanced analysis capabilities to avoid tracing of unneeded quantities. We are also investigating higher-level hybrid-mode approaches such as "pseudo-adjoints" [19] and the exploitation of parallel derivative codes from serial simulation codes by exploiting chain rule associativity.

To understand the latter idea, consider the situation shown in Figure 14: G cannot start before F has been computed, and H has to wait for the completion of G. That is, none of these processes may execute in parallel. This is not the case in derivative computations, however, because of the associativity of the derivative chain rule. For example, we could proceed as in Figure 15. That is, at the same time that we spawn a process to compute F, we spawn a process to compute $\frac{dy}{dx}$, and at the same time that we start with computation of G, we spawn a process to compute $\frac{dz}{dy}$. Lastly, the computation of $\frac{dw}{dz}$ is initiated. Under the assumption that the computation of derivatives takes significantly longer than the simulation itself, we will, in the end, have the three derivative processes running in parallel. When they have finished, we simply accumulate their outputs to arrive at the desired result, $\frac{dw}{dx}$. Thus, if we are willing to duplicate the computation of y and z, we can in this fashion arrive at a coarse-grained parallel schedule that, as a result of its minimal synchronization requirements, could ideally be mapped to a network of workstations.

We are also investigating the extension of automatic differentiation to parallel communication mechanisms, including MPI [48] and PVM [35], parallel languages such as HPF [55] and Fortran-M [34], and languages such as C++ and Fortran 90. While there is no difficulty in principle in providing automatic differentiation capability for these languages, it is not yet clear to what extent the capabilities of these languages impact (in a positive or negative way) the efficient generation of derivative codes.

Lastly, we remark that the ADIFOR project is, in our view, a good example of the power of interdisciplinary research. The techniques employed in ADIFOR are motivated by classical differential calculus, the theory of automatic differentiation, and discrete complexity theory. The various components of the ADIFOR 2.0 system (the ADIFOR preprocessor, the ADIntrinsics-f77 Fortran system, and the SparsLinC library) draw on compiler and source transformation technology and numerical linear algebra. By combining these approaches, ADIFOR allows us to provide a new and, we believe, superior approach to a problem at the heart of numerical computing, namely, the computation of derivatives.

Acknowledgments

We thank Andreas Griewank, George Corliss, and Paul Hovland for their instrumental role in getting the ADIFOR project off the ground, and for their continued advice. We also thank Gordon Pusch for his contribution to the rationale for the complex exception handler, and Heike Baars for her contribution to the "report-once" mode.

References

[1] Proceedings of the 5th AIAA/NASA/USAF/ISSMO symposium on multidisciplinary analysis and optimization, Panama City, Florida, American Association of Aeronautics and Aerospace Engineers, 1994.

[2] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren, and Chau-Wen Tseng. Requirements for data-parallel programming environments. *IEEE Transactions on Parallel & Distributed Technology*, 2(3):48–58, 1994.

[3] B. M. Averick, R. G. Carter, J. J. Moré, and G. L. Xue. The MINPACK-2 test problem collection. Preprint ANL-MCS-P153-0692, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[4] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[5] J.-F. Barthelemy and L. Hall. Automatic differentiation as a tool in engineering design. In *Proceedings of the 4th AIAA/USAF/NASA/OAI Symp. on Multidisciplinary Analysis and Optimization,* AIAA 92-4743, pages 424–432. American Institute of Aeronautics and Astronautics, 1992.

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

[7] Christian Bischof, Alan Carle, and Peyvand Khademi. Fortran 77 interface specification to the SparsLinC library. Technical Report ANL/MCS-TM-196, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[8] Christian Bischof, Alan Carle, Peyvand Khademi, Andrew Mauer, and Paul Hovland. ADIFOR 2.0 user's guide. Technical Report ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

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

[10] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Report ANL/MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[11] Christian Bischof, George Corliss, and Andreas Griewank. Hybrid evaluation of second derivatives in ADIFOR. Technical Report ANL/MCS-TM-166, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[12] Christian Bischof, George Corliss, and Andreas Griewank. Computing second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, 2:211–232, 1993.

[13] Christian Bischof, Larry Green, Kitty Haigler, and Tim Knauff. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4261, pages 73–84. American Institute of Aeronautics and Astronautics, 1994.

[14] Christian Bischof and Andreas Griewank. Computational differentiation and multidisciplinary design. In H. Engl and J. McLaughlin, editors, *Inverse Problems and Optimal Design in Industry*, pages 187–211, Stuttgart, 1994. Teubner Verlag.

[15] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[16] Christian Bischof and Andrew Mauer. ADIC – a tool for the automatic differentiation of C programs. In Preparation.

[17] Christian Bischof, Gordon Pusch, and Alan Carle. On automatic differentiation of codes with complex arithmetic with respect ot real variables. Technical Report ANL/MCS-TM-188, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[18] Christian Bischof, Greg Whiffen, Christine Shoemaker, Alan Carle, and Aaron Ross. Application of automatic differentiation to groundwater transport models. In Alexander Peters et al., editor, *Computational Methods in Water Resources X*, pages 173–182, Dordrecht, 1994. Kluwer Academic Publishers.

[19] Christian H. Bischof. Automatic differentiation, tangent linear models and pseudo-adjoints. Preprint MCS-P472-1094, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[20] Christian H. Bischof and Moe El-Khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. Technical Report ANL/MCS-TM-163, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[21] Ali Bouaricha and Jorge Moré. Environment for large-scale optimization. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[22] Kathy E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North-Holland, New York, 1989.

[23] Barry W. Brown, F. Martin Spears, Lawrence B. Levy, James Lovato, and Kathy Russell. Algorithm LLDRLF: Log-likelihood and some derivatives for Log-F models. Technical report, Dept. of Biomathematics, The University of Texas M.D. Anderson Cancer Center, Houston, 1994.

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

[25] Daewon W. Byun, Robert Dennis, Dongming Hwang, Jr. Carlie Coats, and M. Talat Odman. Computational modelling issues in next generation air quality models. In *Proceedings of IMACS'94, Atlanta, Georgia*, 1994.

[26] D. Callahan, K. Cooper, R. T. Hood, K. Kennedy, and L. M. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–99, December 1988.

[27] Alan Carle, Lawrence Green, Christian Bischof, and Perry Newman. Applications of automatic differentiation in CFD. In *Proceedings of the 25th AIAA Fluid Dynamics Conference, AIAA Paper 94-2197.* American Institute of Aeronautics and Astronautics, 1994.

[28] 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, 1991.

[29] Shirish Chinchalkar. The application of automatic differentiation to problems in engineering analysis. *Computer Methods in Applied Mechanics and Engineering*, 118:197–207, 1994.

[30] George F. Corliss, Christian Bischof, Andreas Griewank, Steven Wright, and Thomas Robey. Automatic differentiation for PDE's – unsaturated flow case study. In Robert Vichnevetski, Doyle Knight, and Gerard Richter, editors, *Advances in Computer Methods for Partial Differential Equations – VII*, pages 150–156, New Brunswick, 1992. IMACS.

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

[32] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford Press, London, 1987.

[33] H. Engl and J. McLaughlin. Proceedings of the symposium on inverse problems and optimal design in industry, Teubner Verlag, Stuttgart, 1994.

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

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM - Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, 1994.

[36] Alan George and Joseph Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, 1981.

[37] Ralf Giering. Adjoint model compiler, manual version 0.2, AMC version 2.04. Technical report, Max-Planck Institut für Meteorolgie, August 1992.

[38] Phillip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, London, 1981.

[39] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[40] 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, 1991.

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

[42] Andreas Griewank. The chain rule revisited in scientific computing. Preprint MCS–P227–0491, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

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

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

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

[46] 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, 1991.

[47] Andreas Griewank and Philippe L. Toint. On the unconstrained optimization of partially separable objective functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, London, 1981. Academic Press.

[48] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI – Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, 1994.

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

[50] M. Heidari and S. Ranjithan. A hybrid optimization approach to the estimation of distributed parameters in two dimensional confined aquifers under steady state conditions. Draft manuscript, 1994.

[51] 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, 1991.

[52] Amin Ibsais and Venkataramana Ajjarapu. The application of automatic differentiation in the continuation power flow. In *Proc. 26th North American Power Symposium, Part I, Manhattan, Kansas*, pages 329–337, 1994.

[53] W. Johnson. Camrad/ja - a comprehensive analytical model of rotorcraft aerodynamics and dynamics - johnson aeronautics version. Technical report, Johnson Aeronautics, 1988.

[54] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–330, Philadelphia, 1991. SIAM.

[55] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.

[56] V. Korivi, L. Sherman, A. Taylor, G. Hou, L. Green, and P. Newman. First- and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4262, pages 87–120. American Institute of Aeronautics and Astronautics, 1994.

[57] V. Korivi, A. Taylor, and P. Newman. Aerodynamic optimization studies using a 3-D supersonic Euler code with efficient calculation of sensitivity derivatives. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4270, pages 170–194. American Institute of Aeronautics and Astronautics, 1994.

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

[59] Jorge J. Moré and Stephen J. Wright. *Optimization Software Guide*. SIAM, Philadelphia, 1993.

[60] Douglas Muir. Description of covariance data in ENDF-6 format. In C. L. Dunford, editor, *Proc. on Nuclear Data Evaluation Methodology*. World Scientific, 1993.

[61] Seon Ki Park and Kelvin Droegemeier. Effect of a microphysical parameterization on the evolution of linear perturbations in a convective cloud model. In *Preprints, Conference on Cloud Physics, January 1995, Dallas, Texas*. American Meteorological Society.

[62] Seon Ki Park, Kelvin Droegemeier, Christian Bischof, and Tim Knauff. Sensitivity analysis of numerically-simulated convective storms using direct and adjoint methods. In *Preprints, 10th Conference on Numerical Weather Prediction, Portland, Oregon*, pages 457–459. American Meterological Society, 1994.

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

[64] Marcela Rosemblun. Automatic differentiation: Overview and application to systems of parametrized nonlinear equations. Technical Report CRPC-TR92267, Center for Research in Parallel Computation, Rice University, 1992.

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

[66] Nicole Rostaing-Schmidt and Eric Hassold. Basic functional representation of programs for automatic differentiation in the Odyssee system. In Francois-Xavier Le Dimet, editor, *High-Performance Computing in the Geosciences*, Dordrecht, 1994. Kluwer Academic Publishers.

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

[68] E. R. Unger and L. E. Hall. The use of automatic differentiation in an aircraft design problem. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4260, pages 64–73. American Institute of Aeronautics and Astronautics, 1994.

[69] Gregory Whiffen, Christine Shoemaker, Christian Bischof, Aaron Ross, and Alan Carle. Application of automatic differentiation to groundwater transport codes. Preprint MCS-P441-0594, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[70] Erich Zauderer. *Partial Differential Equations of Applied Mathematics*. John Wiley & Sons, Somerset, 1989.