Automatic Differentiation, Tangent Linear Models, and (Pseudo)Adjoints

Christian H. Bischof

CRPC-TR95511
January, 1995

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

To appear in *Proceedings of the Workshop on High-Performance Computing in the Geosciences*

Automatic Differentiation, Tangent Linear Models, and (Pseudo)Adjoints[1]

Christian H. Bischof

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL 60439-4843
`bischof@mcs.anl.gov`

Abstract

This paper provides a brief introduction to automatic differentiation and relates it to the tangent linear model and adjoint approaches commonly used in meteorology. After a brief review of the forward and reverse mode of automatic differentiation, the ADIFOR automatic differentiation tool is introduced, and initial results of a sensitivity-enhanced version of the MM5 PSU/NCAR mesoscale weather model are presented. We also present a novel approach to the computation of gradients that uses a reverse mode approach at the time loop level and a forward mode approach at every time step. The resulting "pseudoadjoint" shares the characteristic of an adjoint code that the ratio of gradient to function evaluation does not depend on the number of independent variables. In contrast to a true adjoint approach, however, the nonlinearity of the model plays no role in the complexity of the derivative code.

Keywords: Automatic Differentiation, Adjoint, Tangent Linear Model, MM5, ADIFOR, SparsLinC, Data Assimilation.

1 Introduction

Let $F(p)$ denote the (vector-valued) output of a model F produced by a particular (vector-valued) input p. Employing the Taylor expansion of F around a reference state p_o, we have

$$F(p_o + \triangle p) = F(p_o) + \frac{\partial F(p_o)}{\partial p}\triangle p + \frac{1}{2}(\triangle p)^T \frac{\partial^2 F(p_o)}{\partial p^2}\triangle p + HO(p_o, \triangle p), \qquad (1)$$

where the higher-order terms $HO(p_o, \triangle p)$ satisfy $||HO(p_o, \triangle p)|| = O(||\triangle p||^3)$. 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 first- or second-order approximation of the change of F in response to a perturbation of the input p from its base state p_o. In particular, the first-order Taylor series approximation

$$F(p_o) + \frac{\partial F(p_o)}{\partial p}\triangle p \qquad (2)$$

provides a linear approximation to the (usually nonlinear) behavior of F around the point p_o.

[1]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 by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

Derivatives provide a way for computing a relatively simple approximation of F, thus allowing one to inexpensively explore the behavior of f in the neighborhood of p_o. Hence, derivatives are ubiquitous in numerical computing.

There are four main 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, 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 p at a particular point p_0 by either *one-sided differences*

$$\left.\frac{\partial F(p)}{\partial p_i}\right|_{p=p_0} \approx \frac{F(p_0 \pm h * e_i) - F(p_0)}{h} \tag{3}$$

or *central differences*

$$\left.\frac{\partial F(p)}{\partial p_i}\right|_{p=p_0} \approx \frac{F(p_0 + h * e_i) - F(p_0 - h * e_i)}{2h}. \tag{4}$$

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. Divided difference approximations have the advantage that we need only the function as a "black box." A disadvantage, however, is that their accuracy is hard to assess (see, e.g., [13]).

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. 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 reassemble 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, [15, 28]). By applying the chain rule

$$\left.\frac{\partial}{\partial t} f(g(t))\right|_{t=t_0} = \left(\left.\frac{\partial}{\partial s} f(s)\right|_{s=g(t_0)}\right)\left(\left.\frac{\partial}{\partial t} g(t)\right|_{t=t_0}\right) \tag{5}$$

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 [18]. 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.

In this paper, we explore different ways of computing first-order derivatives employing automatic differentiation tools. The paper is structured as follows. In the next section we explore the relationship between the tangent linear model and the adjoint approach for computing a gradient from a derivative point of view. In Section 3, we briefly review the forward and reverse mode of automatic differentiation when viewing the automatic differentiation as a code rewriting problem. We then briefly describe the ADIFOR tool for the automatic differentiation of Fortran 77 programs, and present some preliminary results from a sensitivity-enhanced version of MM5 developed with ADIFOR. We then suggest a novel approach to computing large gradients which we call the "pseudoadjoint" approach. It combines what we consider to be the good features of the forward and reverse modes while trying to avoid some of their shortcomings. Lastly, we summarize our results.

2 The Tangent Linear Model and the Adjoint

Two approaches are typically employed in the computation of the linearized model (2)—the tangent linear model (TLM) and the adjoint. They form the basis of the following commonly employed techniques.

Sensitivity analysis techniques — here one tries to asses the sensitivity of the responses of a computational model with respect to perturbations in its parameters or initial conditions (see, for example, [27, 25, 26, 36]).

Data assimilation techniques — here one tries to adjust the initial state of a model to best reproduce some observed behavior (see, for example, [34, 33, 35]).

A collection of papers on this subject can be found in [24].

To illustrate the tangent linear model and the adjoint, we assume that the state X of the system at time t satisfies the simple equation

$$X(t) = H(X(t-1)),\ t = 0, \ldots T \tag{6}$$

and that prognostic and diagnostic variables are the same. Further, let

$$J(i) := \left.\frac{\partial H}{\partial X}\right|_{X=X(i)} \tag{7}$$

be the $n \times n$ Jacobian of H with respect to the state at time step i. The tangent linear model (TLM) describes the linearized evolution of errors about the trajectory of a particular nonlinear solution. Denoting by $\delta X(t)$ the sensitivity of the state at time t with respect to perturbations in the initial state $X(0)$, we have

$$\begin{array}{rcl} \delta X(1) & = & J(0) * \delta X(0) \\ \delta X(2) & = & J(1) * \delta X(1) = J(1) * J(0) * \delta X(0) \\ & \vdots & \\ \delta X(t) & = & J(t-1) * \delta X(t-1) = J(t-1) * \cdots * J(0) * \delta X(0). \end{array} \tag{8}$$

Here $\delta X(t)$ should be interpreted as a column vector. The tangent linear model is defined as

$$X(T) + \delta X(T). \tag{9}$$

Comparing this with (2), we see that the TLM and the first-order Taylor approximation with $\triangle p = \delta X(0)$ are identical. We also point out that by initializing $\delta X_i(0) = 1$, $\delta X_j(0) = 0$ for $j \neq i$, $\delta X(T)$ can be interpreted as the sensitivity of all output variables $X(T)$ with respect to a unit change in the ith component $X_i(0)$ of the initial state. In the literature the expression "development of the TLM" is used somewhat loosely to denote either a code for $\delta X(T)$ or $X(T) + \delta X(T)$ as the computation of $\delta X(i)$ and $X(i)$ is usually intertwined.

In contrast, the adjoint integrates the model back in time. Denote the sensitivity of the final state with respect to a change in an intermediate state by $\overline{\delta X(t)}$. Then

$$\begin{aligned}
\overline{\delta X(T-1)} &= \overline{\delta X(T)} * J(T-1) \\
\overline{\delta X(T-2)} &= \overline{\delta X(T-1)} * J(T-2) = \overline{\delta X(T)} * J(T-1) * J(T-2) \\
&\vdots \\
\overline{\delta X(t)} &= \overline{\delta X(T)} * J(T-1) * \cdots * J(t)
\end{aligned} \tag{10}$$

Here $\overline{\delta X(t)}$ should be interpreted as a row vector. In particular, initializing $\overline{\delta X_i(T)} = 1$, $\overline{\delta X_j(0)} = 0$ for $j \neq i$, $\overline{\delta X(0)}$ can be interpreted as the sensitivity of the ith component of the final state with respect to a unit change in all components of the initial state. That is, $\overline{\delta X(0)}$ is the gradient

$$\overline{\delta X(0)} = \frac{\partial X_j(T)}{\partial X(0)}$$

and hence can be viewed as another approach for computing the derivatives required for the first-order linear approximation (2).

Let e_i be the ith canonical unit vector, namely, $e_i(i) = 1$, and $e_i(j) = 0$ for $j \neq i$. Then

$$\delta X(0) = e_j \text{ and } \overline{\delta X(T)}^T = e_i$$

implies

$$\forall\, 1 \leq t \leq T: \ \overline{\delta X(t)}^T * \delta X(t) = e_i^T \frac{\partial X(T)}{\partial X(0)} e_j = \frac{\partial X_i(T)}{\partial X_j(0)}.$$

Hence, combining TLM and adjoint codes, we have many possibilities for computing the same derivative values [8].

To summarize, if we discount numerical instabilities arising from the complementary stability behavior of forward- and backward-integration of dynamical systems (see, for example, [32]), when properly initialized, the tangent linear model and the adjoint will compute the same sensitivities and provide a mechanism for developing a linear approximation of the model. The computational complexity of these two approaches is quite different, though, as we will see in the next section.

3 Automatic Differentiation

Traditionally, two approaches to automatic differentiation (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

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

Figure 1: Sample Code Fragment

variables with respect to the independent variables, corresponding to the forward sensitivity formalism [11, 12], whereas the reverse mode propagates the derivatives of the final values with respect to intermediate variables corresponding to the adjoint sensitivity formalism [11, 12]. In either case, automatic differentiation produces code that computes the *values* of the analytical derivatives accurate to machine precision.

We illustrate the difference between these two 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 "◇" is either "$*$" or "+."

3.1 The Forward Mode

The forward mode of automatic differentiation computes derivatives as shown in Figure 2, 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 $\frac{\partial y(i)}{\partial x(1:n)}$. 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, we initialize ∇`x(i) = 0.0` for $i \neq 3$ and ∇`x(3) = 1.0`. In this case, then, each statement involving a derivative object is a scalar instruction, and we emerge with ∇`y(i)` $= \frac{dy(i)}{dx(3)}$. 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

$$\frac{dy}{dx} = \begin{pmatrix} \frac{\partial y(1)}{\partial x(1)} & \cdots & \frac{\partial y(1)}{\partial x(n)} \\ \frac{\partial y(2)}{\partial x(1)} & \cdots & \frac{\partial y(2)}{\partial x(n)} \end{pmatrix} \tag{11}$$

in that

$$\begin{pmatrix} \nabla y(1) \\ \nabla y(2) \end{pmatrix} = \frac{dy}{dx} * \begin{pmatrix} \nabla x(1) \\ \vdots \\ \nabla x(n) \end{pmatrix}. \tag{12}$$

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

Forward Mode for $\diamond = *$

Figure 2: Forward Mode Code for Code Fragment of Figure 1

In particular, initializing `∇x(i) = d(i)`, we compute the directional derivative

$$\frac{dy}{dx} * d = \lim_{h \to 0} \frac{y(x + h * d) - y(x)}{h}. \tag{13}$$

Forward mode code is easy to generate, preserves any parallelizable or vectorizable structures within the original code, and is readily generalized to higher-order derivatives [4]. If we wish to compute m directional derivatives, then running forward-mode code requires at most on the order of m times as much time and memory as the original code.

We also note that what we called the forward mode is the approach often employed when deriving the tangent linear model of a computer code by hand. This should not come as a surprise. Applying the forward mode of automatic differentiation to a code computing (6) will result in the same derivatives as the TLM accumulation (8) when $\nabla X_i(0) = \delta X_i(0)\,, i = 1, \ldots, n$.

3.2 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, [18]) that the statement $s = f(v, w)$ in the original code implies

$$\begin{aligned} \overline{v} &\mathrel{+}= \tfrac{\partial s}{\partial v}\overline{s} \\ \overline{w} &\mathrel{+}= \tfrac{\partial s}{\partial w}\overline{s} \end{aligned}$$

```
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;
      y1value(c1) = y1value(c1 - 1) ◇ x(i)
   else
      jump(i) = 'right'; c2 = c2 + 1;
      y2value(c2) = y2value(c2 - 1) ◇ x(i)
   endif
enddo
```

Figure 3: Code Fragment of Figure 1 Modified in Preparation for Reverse-Mode Code Generation

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 or subtraction, $\frac{\partial s}{\partial v} = \frac{\partial s}{\partial w} = 1$, and hence $\frac{\partial s}{\partial v}$ and $\frac{\partial s}{\partial w}$ do not depend on the values of their operands. On the other hand, when f is a nonlinear operation such as a multiplication, both $\frac{\partial s}{\partial v}$ and $\frac{\partial s}{\partial w}$ do depend on the values of their operands, and one must remember either these derivative 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. To this end, we transform the code from Figure 1 to the form shown in Figure 3, where we generate a trace of the branch history in the "`jump`" array, and save intermediate values of the variables `y(1)` and `y(2)` in `y1value(:)` and `y2value(:)`. The counters `c1` and `c2` are used to keep track of when a variable was last assigned. We are now in a position to automatically generate reverse mode code for this computation. The result is shown in Figure 4. We use the notation $\overline{\texttt{s}}$ to denote the derivative object associated with the program variable `s`.

We can easily convince ourselves that when we initialize $\overline{\texttt{y(1)}} = 1.0$, $\overline{\texttt{y(2)}} = 0.0$, and all other derivative objects to zero, then by running the codes in Figures 3 and 4, we emerge with $\overline{\texttt{x(i)}} = \frac{\partial x(i)}{\partial y(1)}$. Similarly, initializing $\overline{\texttt{y(1)}} = 0.0$, $\overline{\texttt{y(2)}} = 1.0$, and all other derivative objects to zero, we compute $\overline{\texttt{x(i)}} = \frac{\partial x(i)}{\partial y(2)}$. In general, if we view the adjoint vector associated with a program variable as column vector, the linearity of differentiation implies that

$$\left(\overline{x(1)}, \cdots, \overline{x(n)}\right) = \left(\ \overline{y(1)}\ ,\ \overline{y(2)}\ \right) * \frac{dy}{dx}, \tag{14}$$

where $\frac{dy}{dx}$ is as defined in equation (11). That is, reverse mode code allows us to compute arbitrary linear combinations of the rows of the Jacobian. Initializing $\overline{\texttt{y(i)}}$ `= d(i)`, we compute the derivative

$$\frac{\partial\,(d^T * y(x))}{\partial\, x}. \tag{15}$$

```
$\overline{\texttt{y1value(c1)}}$ = $\overline{y(1)}$; $\overline{\texttt{y2value(c2)}}$ = $\overline{y(2)}$;
do i = n to 1 step -1
   if (jump(i) = 'left') then
      $\overline{\texttt{y1value(c1 - 1)}}$ += $\overline{\texttt{yvalue1(c1)}}$
      $\overline{\texttt{x(i)}}$ += $\overline{\texttt{y1value(c1)}}$
      c1 = c1 - 1
   else
      $\overline{\texttt{y2value(c2 - 1)}}$ += $\overline{\texttt{y2value(c2)}}$
      $\overline{\texttt{x(i)}}$ += $\overline{\texttt{y2value(c2)}}$
      c2 = c2 - 1
   endif
enddo
```

Reverse Mode for $\diamond = +$

```
$\overline{\texttt{y1value(c1)}}$ = $\overline{y(1)}$; $\overline{\texttt{y2value(c2)}}$ = $\overline{y(2)}$;
do i = n to 1 step -1
   if (jump(i) = 'left') then
      $\overline{\texttt{y1value(c1 - 1)}}$ += x(i)*$\overline{\texttt{y1value(c1)}}$
      $\overline{\texttt{x(i)}}$ += y1value(c1-1)*$\overline{\texttt{y1value(c1)}}$
      c1 = c1 - 1
   else
      $\overline{\texttt{y2value(c2 - 1)}}$ += x(i)*$\overline{\texttt{y2value(c2)}}$
      $\overline{\texttt{x(i)}}$ += y2value(c2-1)*$\overline{\texttt{y2value(c2)}}$
      c2 = c2 - 1
   endif
enddo
```

Reverse Mode for $\diamond = *$

Figure 4: Reverse Mode Code Generated from Code in Figure 3

Note that it is quite an involved process to generate reverse mode code. While the complexity of the forward-mode code generation in Figure 2 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)` and `y(2)` 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 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 $m = 1$.

We also note that what we called the reverse mode of automatic differentiation is what is usually performed by hand when deriving an adjoint code. Applying the reverse mode of automatic differentiation to a code computing (6) will result in the same derivatives as the adjoint accumulation (10) when $\overline{X_i(T)} = \overline{\delta X_i(T)}, i = 1, \ldots, n$.

4 The ADIFOR Automatic Differentiation Tool and an Application to the MM5 Mesoscale Weather Model

There have been various implementations of automatic differentiation; an extensive survey can be found in [22]. In particular, we mention GRESS [20], and PADRE-2 [23] for Fortran Programs and ADOL-C [17] for C programs. 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 [31].

4.1 The ADIFOR (Automatic Differentiation of Fortran) Tool

Recently, a "source transformation" approach to automatic differentiation has been explored in the ADIFOR [2], ADIC [7], and Odyssee [29, 30] tools. ADIFOR and Odyssee transform Fortran 77 code and ADIC transforms ANSI-C code. By applying the rules of automatic differentiation, these tools generate new code, which, when executed, computes derivatives without the overhead associated with trace interpretation schemes. ADIFOR and ADIC mainly use the forward mode[2]. In contrast, Odyssee employs the reverse mode.

Given a Fortran subroutine (or collection of subroutines) describing a "function," and an indication of which variables in parameter lists or common blocks correspond to "independent" and "dependent" variables with respect to differentiation, ADIFOR performs a data flow analysis to determine which statements in the code have to be augmented with derivative computations and then produces Fortran 77 code that computes the derivatives of the dependent variables with respect to the independent ones. ADIFOR produces portable Fortran 77 code, and accepts almost all of Fortran 77; in particular, it can deal with arbitrary calling sequences, nested subroutines, common blocks, and equivalences. The ADIFOR-generated code tries to preserve vectorization and parallelism in the original code and employs a consistent subroutine-naming scheme, which allows for code tuning, the exploitation of domain-specific knowledge, and the use of vendor-supplied libraries.

ADIFOR employs a hybrid forward/reverse mode approach to generating derivatives. For each assignment statement, it uses the reverse mode to generate code that computes the partial derivatives of the result with respect to the variables on the right-hand side and then employs the forward mode to propagate overall derivatives. For example, the single Fortran statement

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

gets transformed into the code segment shown in Figure 5. Note that none of the common subexpressions $x(i) * x(j)$ are recomputed in the reverse mode section for $\frac{\partial \mathtt{y}}{\partial \mathtt{x(i)}}$. The variable **g$p$** denotes the number of (directional) derivatives being computed. For example, if **gp** $= 5$, and **g$x(1:5,1:5)** is the 5×5 identity matrix (i.e., **g$x(i,j)** $= \frac{\partial x(i)}{\partial x(j)}$) then upon execution of these statements, **g$y(1:5)** equals $\frac{dy}{dx}$. On the other hand, assume that we wished only to compute derivatives with respect to a scalar parameter **s**, so **gp** $= 1$, and, on entry to this code segment, **g$x(1,i)** $= \frac{\partial x(i)}{\partial s}$, $i = 1, \ldots, 5$. Then the do-loop in Figure 5 implicitly computes $\frac{dy}{ds} = \frac{dy}{dx}\frac{dx}{ds}$ without ever forming $\frac{\partial y}{\partial x}$ explicitly. Note that the cost of computing y is amortized over all the derivatives being computed. Thus, 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.

We also see that ADIFOR-generated code provides the directional derivative computation possibilities associated with the forward mode of automatic differentiation [6]. Instead of simply producing code to compute the Jacobian J, ADIFOR produces code to compute $J * S$, where the "seed matrix" S is

[2]Information on ADIFOR and ADIC can be found on the world-wide web under `http://www.mcs.anl.gov/Projects/autodiff/index.html`

```
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
```

Reverse Mode for computing $\frac{\partial\,\mathtt{y}}{\partial\,\mathtt{x(i)}}$:

$$\mathtt{r\$jbar} = \frac{\partial\,\mathtt{y}}{\partial\,\mathtt{x(i)}},\ i = 1,\ldots,4$$

$$\mathtt{r\$3} = \frac{\partial\,\mathtt{y}}{\partial\,\mathtt{x(5)}}$$

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

Forward Mode:
Assembling ∇y from $\frac{\partial\,\mathtt{y}}{\partial\,\mathtt{x(i)}}$ and $\nabla x(i)$, $i = 1,\ldots,5$.

```
y = r$3 * x(5)
```

Computing function value

Figure 5: Sample Segment of an ADIFOR-generated Code

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. In [5] the flexibility of the ADIFOR interface is exploited in a "stripmining" approach to decrease turnaround time for derivative computations by spawning several independent subprocesses computing parts of the desired gradient or Jacobian. The seed matrix also provides a powerful mechanism for decreasing the computational complexity of derivative codes through judicious use of the chain rule [8, 21]. The running time and storage requirements of the ADIFOR-generated code are roughly proportional to the numbers of columns of S, which equals the **gp** variable in the sample code above.

ADIFOR has been successfully applied to codes from various domains of science. Experiences with meteorological codes, for example, have been reported in [10, 25, 26, 27]. Typically, 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 superior performance is due to the reverse/forward hybrid mode and a dependence analysis that tries to avoid computing derivatives of expressions that do not affect the "dependent variables". We also note that in order to take full advantage of reduced complexity of ADIFOR-generated code, it is advantageous to compute several directional derivatives at the same time — so the ADIFOR-generated code may require significantly more memory than the original simulation code.

4.2 First Results with a Sensitivity-Enhanced Version of the MM5 Mesoscale Weather Model

The development of a sensitivity-enhanced version of the Fifth-Generation Penn State/NCAR mesoscale weather model (MM5) [14] using the ADIFOR automatic differentiation tool is in progress. ADIFOR expects code that complies with the Fortran 77 standard. MM5 does not comply with this standard —

in particular, it makes much use of "pointer variables." We circumvented this difficulty by developing an MM5-specific tool to map the pointer handling to standard-conforming Fortran77 code acceptable to ADIFOR, and to remap ADIFOR's output to obtain the desired sensitivity-enhanced code.

To date, we have developed sensitivity-enhanced versions of the nonhydrostatic dynamics and most of the physics, including the Blackadar high-resolution planetary boundary layer parameterization, the Grell and Kuo cumulus parametrizations, the Dudhia long- and short-wave radiation scheme, and the models representing explicit moisture with treatment of mixed phase processes (ice), shallow convection, and dry convective adjustment.

To verify our ADIFOR-generated sensitivities, we used the sensitivity-enhanced version of MM5 to compute the time-evolution of first-order perturbations of pressure, temperature, water vapor, and convective rain, in response to perturbations in the initial-pressure data. Our perturbation scheme introduces an artificial perturbation parameter ϵ. To properly address the leapfrog scheme employed in MM5, we perturbed the pressure[3] on the two initial time-slices t_0 and t_1 as follows:

$$\begin{array}{lcl} p'(x,y,z,t_0;\,\epsilon) & := & (1+\epsilon)\,p'(x,y,z,t_0) \\ p'(x,y,z,t_1;\,\epsilon) & := & (1+\epsilon)\,p'(x,y,z,t_1) \end{array} . \tag{16}$$

Then ϵ parameterizes a family of perturbed solutions for the MM5 variables, with the unperturbed solution obtained at $\epsilon = 0$. Any quantity (Q, say) influenced by the initial pressure therefore acquires an implicit dependence on ϵ. Pressure, temperature, water vapor, and convective rain, are particular instances of such quantities. By definition, the first-order perturbation-theoretic sensitivity of Q is

$$\delta Q(x,y,z,t) := \left.\frac{\partial\, Q(x,y,z,t;\,\epsilon)}{\partial\,\epsilon}\right|_{\epsilon=0} . \tag{17}$$

Given our choice of the perturbation of p', $\delta Q(x,y,z,t)$ can be interpreted as the sensitivity of Q to a uniform relative change in the initial pressure fields. We note that both $p'(t_0)$ and $p'(t_1)$ were perturbed, since a leapfrog timestepping approach is employed in MM5.

The sensitivity $\delta Q(x,y,z,t)$ can easily be computed by using automatic differentiation. We slightly modify MM5 to include ϵ as an input parameter entering into the computation of the initial pressre, as shown above. We then employ an automatic differentiation tool (ADIFOR, in our case) to differentiate the code with respect to ϵ. Finally, we evaluate the sensitivity-enhanced code at $\epsilon = 0$. In this fashion, we can compute quantities equivalent to those obtained in the tangent linear model *without any further hand-modification of the code.*

We compared the results of our perturbation AD-based method with the results of perturbing the initial-pressure data by one part in 10^3 in the fully nonlinear model, which corresponds to choosing $\epsilon = \pm 10^{-3}$ in Equation (16). For each of pressure (MM5 variable `ppa`), temperature (`ta`), water vapor (`qva`), and convective rain (`rainc`), we computed the following quantities:

- Its value using the initial pressure distribution.
- Its sensitivity $\delta Q(x,y,z,t)$ provided by the ADIFOR-generated code.

[3]Following standard notation, $p' = p - p_0$ is the deviation of the absolute pressure p from the reference-state pressure p_0; this is the quantity actually computed by MM5 in nonhydrostatic mode.

- An approximation to this sensitivity via a central divided-difference approximation (4). Hence, for example, choosing $Q = \texttt{ppa}$, we also compute the values $\texttt{ppa}_{\pm}$ obtained by a relative perturbation of the initial pressure distribution by $\pm\epsilon = 10^{-3}$, as in Equation (16). We then computed the second-order divided difference approximation $\tilde{\delta}\texttt{ppa}$ to $\delta\texttt{ppa}$:

$$\tilde{\delta}\texttt{ppa}(x, y, z, t, \epsilon) := \frac{\texttt{ppa}_{+} - \texttt{ppa}_{-}}{2\epsilon}.$$

- The difference

$$\Delta Q := \tilde{\delta}Q - \delta Q$$

  between the divided-difference approximation and the first-order sensitivity. It follows from (1) that, assuming that the model is continuously differentiable in the vicinity of the base state, ΔQ should be of the order $O(\epsilon^3)$ and hence negligible except for areas of high nonlinearity. It should be noted, however, that there is no guarantee that the model is continuously differentiable—in particular, in light of the switching behavior in the moisture physics modeling.

We note that for this particular case the ADIFOR-generated code exhibits run times and storage requirements on the order of twice those of the original MM5 code, and hence it obtains first-order sensitivities at about two-thirds the cost of the divided-difference approximation.

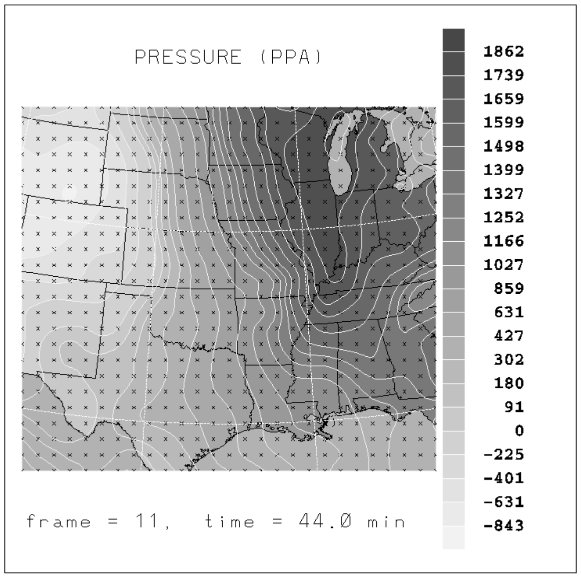
The data set we had at our disposal for testing purposes uses a $28 \times 25 \times 23$ grid, without nesting. We ran the model for 12 time steps of 4 minutes each, terminating at 48 minutes. In Figure 6, for the final time-step, we show for surface pressure

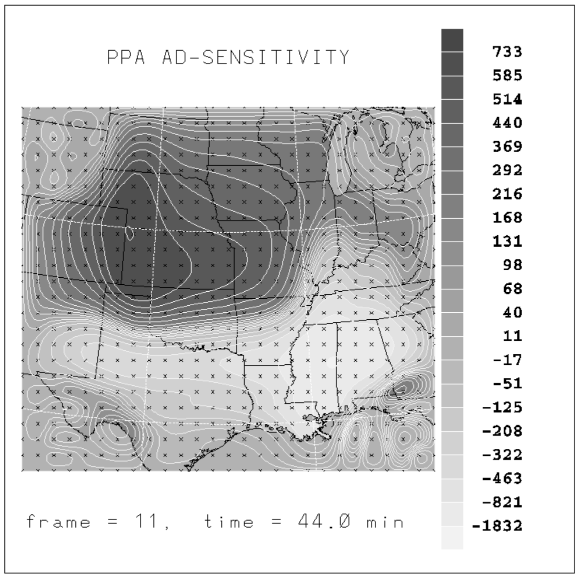
- the value of the forecast variable,
- the sensitivity of the forecast variable with respect to a uniform relative change of the pressure at time zero, and
- the difference between the first-order sensitivity and the divided-difference approximation.

We mention that this perturbation may violate some consistency conditions enforced by the MM5 preprocessors. We would have had to perturb the vertical wind velocity as well, in order not to excite sound waves. This issue will also be addressed by generating sensitivity-enhanced versions of the MM5 preprocessors. We did observe these waves, but they quickly propagate out of the system. At any rate, the purpose of this exercise was to validate our ADIFOR-generated derivatives. Since pressure couples with all the other variables, it still is an appropriate choice to check the correctness of the ADIFOR-generated derivative propagation code.

The upper left plot in Figure 6 shows the value of ppa after the twelfth time step, the upper right plot shows the sensitivity $\delta\texttt{ppa}$, and the bottom plots shows $\Delta\texttt{ppa}$. Pressure is in Pascals, and $\delta\texttt{ppa}$ and $\Delta\texttt{ppa}$ are in Pascals per unit ϵ. The labels in the legend denote the boundaries between contour intervals.

As we can see, excellent first-order agreement is achieved everywhere for surface pressure. We see that the fractional differences between the first-order sensitivity and its divided-difference approximation are only a few parts in 10^3. Since the perturbation is the sensitivity times ϵ, it follows that the absolute differences between the tangent-linear and fully nonlinear models of the surface-pressure field are only of the order of a few parts in 10^6.





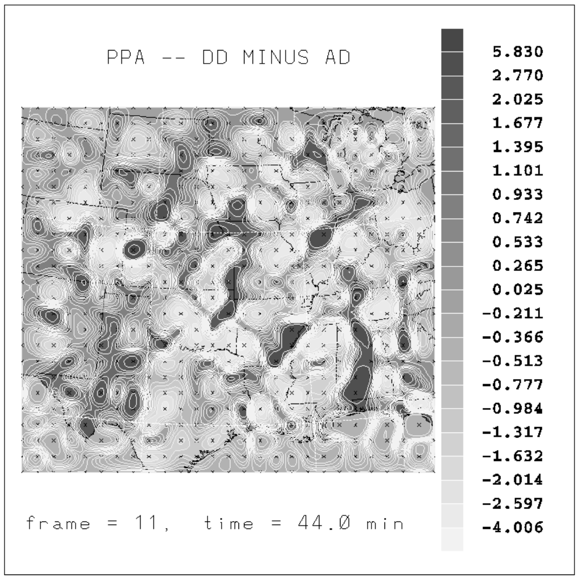


Figure 6: Plot showing the surface-pressure field **ppa** (upper left), the first-order sensitivity δ**ppa** (upper right), and the difference Δ**ppa** (bottom)

5 Pseaudoadjoints

Let us now assume that we have a computation where we repeatedly update a (large) state X and at the end use a merit function that summarizes the relevant features of the final state in a few numbers. That is, we can view the computation as follows:

$$X(0) \xrightarrow{H} X(1) \xrightarrow{H} \cdots \xrightarrow{H} X(T) \xrightarrow{R} r, \tag{18}$$

where, as in (6), H denotes the update operator and R denotes the merit function. We further assume for simplicity that $1 = \dim(r) \ll n := \dim(X)$.

We are interested in efficiently evaluating the gradient

$$\frac{\partial\, r}{\partial\, X(0)} = \frac{\partial\, r}{\partial\, X(T)} * \frac{\partial\, X(T)}{\partial\, X(T-1)} * \ldots * \frac{\partial\, X(1)}{\partial\, X(0)}. \tag{19}$$

A forward-mode based approach like that used in ADIFOR will require a run time that is proportional to n, whereas a reverse mode approach might require a run time that is comparable to a few function evaluations. Of course, the complexity of the reverse mode approach greatly depends on the operator H. If H is "mostly linear," then applying the reverse mode to H does not require much storage for intermediate values. A "highly nonlinear" H, on the other hand, may pose considerable storage demands.

Let us further assume that

- $\frac{\partial\, R}{\partial\, x}$ can be computed easily (often it is a weighted sum of squares) and that
- $\frac{\partial\, H}{\partial\, X}$ is sparse. This situation is fairly typical because of the local nature of stencil-based approximations.

5.1 Exploiting Sparsity in Forward-Mode Derivative Computations

Forward-mode approaches can be used advantageously to compute large sparse Jacobians. One can employ a so-called compressed Jacobian approach, which, given the sparsity pattern of the Jacobian, derives a graph coloring that identifies which columns of the Jacobian can be computed with the same directional derivative. To illustrate, let us assume that we have a function

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

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

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

Columns 1 and 2, as well as columns 3 and 4, are structurally orthogonal. In divided-difference approximations one could exploit that structure by perturbing both x_1 and x_2 in one function evaluation, and both x_3 and x_4 in the other. In an automatic differentiation system like ADIFOR, one can exploit this fact by setting

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

which results in the computation of the compressed Jacobian

$$JS = \begin{pmatrix} \bigcirc & \\ \bigcirc & \diamond \\ \triangle & \diamond \\ \triangle & \Box \\ \triangle & \Box \end{pmatrix}$$

at roughly half the cost. For most grid problems, the width p of the compressed Jacobian is independent of the problem size and depends only on the local stencil chosen. Experimental results with this approach in computing large sparse Jacobians as they arise in large-scale nonlinear equations have been reported in [1].

This compressed Jacobian approach is also applicable to the computation of gradients of so-called partially separable functions [19], which are functions f that can be represented in the form

$$f(x) = \sum_{i=1}^{np} f_i(x), \tag{20}$$

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. It can be shown [19] that any function with a sparse Hessian is a partially separable one. The computation of the gradient of a partially separable function can be reduced to the problem of computing a sparse Jacobian [9] by realizing that the gradient of f can easily be obtained by summing the rows of the *sparse* Jacobian $\frac{dG}{dx}$, where

$$G(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_{np}(x) \end{pmatrix}. \tag{21}$$

Another approach is based on the realization that the workhorse of any mainly forward-mode first-order automatic differentiation approach is a "vector linear combination," for example, ∇`y(1) = x(i)*`∇`y(1) + y(1)*`∇`x(i)` in Figure 3. Here ∇`y(1)` and ∇`x(i)` are vectors of length p, where, as in Subsection 4, p denotes the number of directional derivatives to be computed, and `y(1)` and `x(i)` are scalars. This operation is a particular instantiation of

$$w = \sum_{i=1}^{k} \alpha_i * v_i, \tag{22}$$

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 the initial seed matrix is sparse (e.g., the identity), then, if we ignore exact numerical cancellation, the left-hand side vector w in (22) 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, all intermediate vectors must be sparse. That is, by expressing the derivative linear combinations with algorithms and data structures tailored toward to sparse vectors, we can exploit sparsity in a transparent fashion, *even if the sparsity pattern of the derivative matrix is not known beforehand.* Also note that the sparsity structure of J or ∇f_i is computed as a byproduct of the derivative computation.

The SparsLinC (Sparse Linear Combination) Library [3, 8] addresses the scenario where p is large and most of the vectors involved in vector linear combination are sparse. It provides 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.

Efficient Memory Allocation Scheme: SparsLinC employs a "bucket" memory allocation scheme which, in effect, provides a buffered memory allocation mechanism.

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 (20).

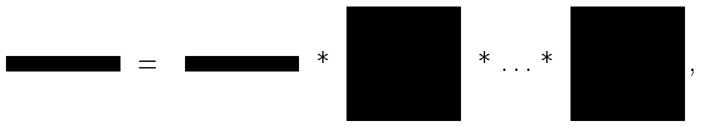
Full Precision Support: single and double precision computations are provided for both real- and complex-valued computations.

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 (20). We also mention that almost no memory is allocated for derivative objects that are all zeros. SparsLinC will be fully integrated in the forthcoming release of the ADIFOR 2.0 system.

Note that neither the run time of the compressed Jacobian nor that of the SparsLinC approach is affected by nonlinearities in the program. That is, changing all additions in the code to multiplications does not increase the required storage, and the run time increases at most by a factor of two.

5.2 Pseudoadjoints

Now, coming back to the problem of computing $\frac{\partial r}{\partial X(0)}$, we realize that if we consider the computation of $\frac{\partial H}{\partial X}$ as a "black box," we can interpret equation (19) as a series of matrix-matrix and vector-matrix multiplications, namely,



where the vector on the left-hand side corresponds to $\frac{\partial r}{\partial X(0)}$, the vector on the right-hand side corresponds to $\frac{\partial r}{\partial X}|_{X=X(T)}$, and the matrices on the right-hand side correspond to

$$J(i) := \frac{\partial H}{\partial X}|_{X=X(i)}, i = 1, \ldots, T-1.$$

An automatic differentiation tool that employs the forward mode throughout can be interpreted as accumulating this product from the right, forming a large matrix corresponding to $\frac{\partial X(T)}{\partial X(0)}$, which then is multiplied by the vector $\frac{\partial r}{\partial X(T)}$ to result in the final gradient. Obviously, it would be much more advantageous to accumulate the product from the left, computing $\frac{\partial r}{\partial X(i)}, i = T, \ldots, 1$ through a series of matrix-vector multiplies. The first approach can be interpreted as the forward mode in an algebra where H and R are elementary operators, the latter one as the reverse mode in this algebra.

However, using the approaches outlined in the previous subsection, we are in a position to inexpensively compute $\frac{\partial H}{\partial X}$, with predictable storage and runtime requirements that do not depend on the nonlinearity of the model. Hence, by storing all intermediate derivatives $J(i)$ in a forward sweep. we can compute $\frac{dr}{dX(0)}$ through a series of sparse matrix-vector multiplies at a runtime complexity that is of the order

$$T * \text{runtime}(\frac{\partial H}{\partial X}) + T * \text{runtime(sparse matrix-vector multiply)}$$

and the storage complexity is of the order

$$T * \text{storage}(\frac{\partial H}{\partial X}).$$

If we only store snapshots of X in the forward pass and regenerate individual Jacobians $\frac{\partial H}{\partial X}|_{X=X(i)}$ when needed, the runtime complexity is of the order

$$T * \text{runtime}(\frac{\partial H}{\partial X}) + T * \text{runtime}(H) + T * \text{runtime(sparse matrix-vector multiply)}$$

and storage complexity that is of the order

$$\text{storage}(\frac{\partial H}{\partial X}) + T * \text{storage}(X).$$

When we employ the snapshotting scheme proposed by Griewank [16] to regenerate the $X(i)$ from a series of checkpoints, the time complexity of the latter approach becomes of the order

$$\begin{aligned} T * \text{runtime}(\frac{\partial H}{\partial X}) & + & T * (1 + \log(T)) * \text{runtime}(H) \\ & + & T * \text{runtime(sparse matrix-vector multiply)} \end{aligned}$$

and storage complexity is of the order

$$\text{storage}(\frac{\partial H}{\partial X}) + (1 + \log(T)) * \text{storage}(X).$$

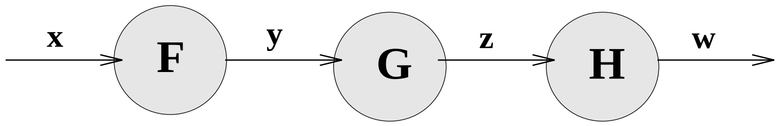


Figure 7: Serial Simulation

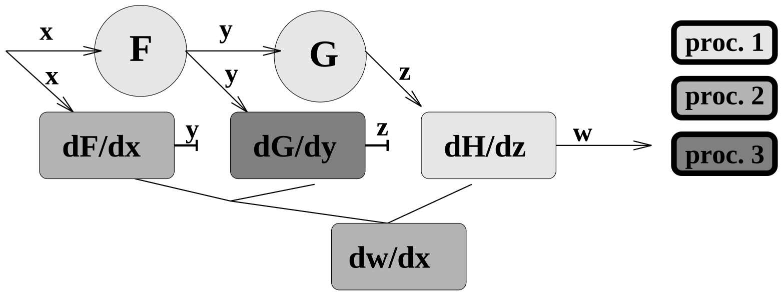


Figure 8: Parallel Derivative Computation

Given that we can compute sparse Jacobians efficiently, at a storage and memory cost that is a moderate multiple of the cost for evaluating H itself, the last approach may be an attractive approach to developing a gradient code. It shares with a "usual" adjoint the characteristic that the ratio of gradient to function evaluation does not depend on the number of input parameters. This approach capitalizes on the advantages of the forward mode (efficient computation of sparse Jacobians with predictable complexity independent of the nonlinearity of the model) and the reverse mode (lower arithmetic complexity). It also avoids the drawbacks of the forward mode (the ratio of gradient to function evaluation depends on the number of input variables) and the reverse mode (a highly nonlinear model may lead to excessive storage demands).

However, if execution time is of the essence, we can do even better by exploiting chain rule associativity to break the time dependency in the derivative computation. To illustrate this idea, we consider the situation shown in Figure 7: 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 8. 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, with minimal synchronization requirements, could be mapped to a network of workstations.

In particular, if we now apply this idea to our problem (18), we could, if enough memory and processors were available, spawn processes to independently compute $J(i)$ in parallel, and would only have to wait for an *additional* run time of the order of

$$\text{runtime}(\frac{\partial H}{\partial X}) + T * \text{runtime}(\text{sparse matrix-vector multiply}),$$

to obtain the desired gradient $\frac{\partial r}{\partial X(0)}$.

6 Conclusions and Future Work

This paper gave a brief introduction into automatic differentiation and related it to the tangent linear model and adjoint approaches commonly used in meteorology. We briefly reviewed the forward and reverse mode of automatic differentiation and introduced the ADIFOR automatic differentiation tool. We also presented first results of a sensitivity-enhanced version of the MM5 PSU/NCAR mesoscale weather model, thus demonstrating that automatic differentiation can generate results equivalent to a tangent linear model for sophisticated weather models, with minimal recourse to laborious and error-prone hand-coding.

We presented a novel approach to the computation of gradients, which used a reverse mode approach at the timestep level and a forward mode approach at every time step. The resulting "pseudoadjoint" shared the characteristic of an adjoint code that the ratio of gradient to function evaluation did not depend on the number of independent variables, but, in contrast to a true adjoint approach, the nonlinearity of the model played no role in the complexity of the derivative code. Lastly, we motivated how chain rule associativity could be employed to break time dependencies in the derivative computation.

The pseudoadjoint strategy is a particular instantiation of what we call a "hybrid mode," where both forward and reverse modes of automatic differentiation are employed at various levels in the derivative computation. The strategy employed in the ADIFOR and ADIC tools is a particularly simple instance of such a hybrid strategy which, in some sense, is at the opposite spectrum of the "pseudoadjoint" approach suggested here. In the pseudoadjoint approach suggested in Section 5 we are using the reverse mode at the outermost loop level, and a forward based technique in the rest, whereas ADIFOR and ADIC employs the reverse mode at the lowest level, within the scope of an assignment statement. Clearly, there are many other alternatives, and we are beginning to explore them in a systematic fashion with an eye toward capitalizing on the benefits of the forward and reverse mode approaches while avoiding their respective drawbacks.

Acknowledgments

We thank Andreas Griewank for many stimulating discussions on the subject, Alan Carle for his instrumental role in the ADIFOR project, and Gordon Pusch for his essential role in the development of the sensitivity-enhanced version of the MM5 code. We also thank Kelvin Droegemeier and Seon Ki Park for introducing us to the use of tangent linear and adjoint models in meteorology.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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*, pages 73–84. American Institute of Aeronautics and Astronautics, 1994. AIAA 94-4261.

[6] 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.

[7] Christian Bischof and Andrew Mauer. ADIC – a tool for the automatic differentiation of C programs. Unpublished Information, Argonne National Laboratory, 1994.

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

[9] 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.

[10] 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.

[11] D. G. Cacuci. Sensitivity theory for nonlinear systems, I: Nonlinear functional analysis approach. *Journal of Mathematical Physics*, 22(12):2794–2802, 1981.

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

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

[14] G. A. Grell, J. Dudhia, and D. R. Stauffer. A description of the fifth-generation Penn State/NCAR mesoscale weather model (MM5). Technical Report NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado, June 1994.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Paul Hovland, Christian Bischof, Donna Spiegelman, and Mario Casella. Efficient derivative codes through automatic differentiation and interface contraction and an application in biostatistics. in preparation.

[22] 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.

[23] 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.

[24] Special Issue on Adjoint Applications in Dynamic Meterology. Tellus, 45a(5), 1993.

[25] 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.

[26] Seon Ki Park and Kelvin Droegemeier. On the use of automatic differentiation to assess parametric sensitivity in convective-scale variational data assimilation. In *Preprints, Proc. Int. Symp. on Assimilation of Observations in Meteorology and Oceanography, Tokyo, Japan, March 1995.* World Meteorological Organization.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] Karl Svozil. *Randomness and Undecidability in Physics.* World Scientific, Singapore, 1993.

[33] Olivier Talagrand. The use of adjoint equations in numerical modelling of the atmospheric circulation. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms*, pages 169–180, Philadelphia, 1991. SIAM.

[34] Jean-Noël Thépaut, Drasko Vasiljevic, Philippe Courtier, and Jean Pailleux. Variational assimilation of conventional meteorological observations with a multilevel primitive-equation model. *Q. J. R. Meteorol. Soc.*, 119:153–186, 1993.

[35] Z. Wang, I. M. Navon, X. Zou, and K. J. Ingles. 4-D variational data assimilation with a global multilevel primitive equation model. In *Preprints, Proc. Int. Symp. on Assimilation of Observations in Meteorology and Oceanography, Tokyo, Japan, March 1995.* World Meteorological Organization.

[36] Zhi Wang, Kelvin Droegemeier, Ming Xue, and Seon Ki Park. Sensitivity analysis of a 3-D compressible storm-scale model to input parameters. In *Preprints, Proc. Int. Symp. on Assimilation of Observations in Meteorology and Oceanography, Tokyo, Japan, March 1995.* World Meteorological Organization.