ADIFOR Working Note #8:

Hybrid Evaluation of Second Derivatives in ADIFOR*

by

Christian Bischof, George Corliss,† and Andreas Griewank

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Technical Memorandum ANL/MCS–TM–166

Abstract

Many algorithms for scientific computation require second- or higher-order partial derivatives, which can be efficiently computed by propagating a set of univariate Taylor series. We describe how to implement second-order mixed partial derivative computations in ADIFOR (Automatic Differentiation In FORtran), a Fortran-to-Fortran source transformation tool. Globally, we propagate three-term univariate Taylor series in the forward mode. Locally, we preaccumulate local gradients and Hessians for complicated expressions on the right-hand sides of assignment statements. We describe the source transformations and give an example of the transformed code.

1 Goals

The goals of this paper are

1. to describe the code generated by ADIFOR to compute second derivatives and

2. to document some of the design decisions made in arriving at this implementation.

We assume that the reader is familiar with the Fortran-to-Fortran source transformation tool ADIFOR (Automatic Differentiation In FORtran) as described in [1, 2, 3, 4, 6], as well as with the theoretical framework for computing second- and higher-order mixed partial derivatives by interpolating from sets of univariate Taylor series [5]. Here, we describe the implementation in ADIFOR of the framework outlined in [5].

In Section 2, we outline briefly where second derivatives are required for reliable scientific computation. A more complete survey of algorithms that require second- and higher-order derivatives is in [5]. In Section 3, we discuss components of the algorithm we implement if ADIFOR for computing second derivatives: forward-mode Hessians, interpolation, forward-mode Taylor series, and preaccumulation. Section 4 contains a discussion of the tasks accomplished by ADIFOR in its generation of code to compute second-order derivatives. An example in Section 5 applies ADIFOR's tasks to a simple subroutine. Finally in Section 6, we discuss some implementation decisions.

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†Also affiliated with the Department of Mathematics, Statistics, and Computer Science, Marquette University, Milwaukee, Wisconsin 53233.

2 Need for Second Derivatives

The primary motivation for adding the ability to compute second derivatives comes from optimization. Given $f : \mathbf{R}^n \mapsto \mathbf{R}$, unconstrained optimization algorithms minimize f locally by solving $\nabla f = 0$ using a Newton or a secant-type iterative method [7]. The Newton iteration requires the Hessian $\nabla^2 f$. In nonlinearly constrained optimization, the curvature of the constraint surfaces is represented by the Hessians $\nabla^2 c_i$ of the active constraints $c_i(x) = 0$. Often, all these second derivatives are aggregated into the Hessian of the Lagrangian

$$\nabla^2 L = \nabla^2 f + \sum_i \lambda_i \nabla^2 c_i,$$

where the Lagrange multipliers λ_i are derived in some way from first derivative information, i.e. the gradients of the objective and the active constraints. In most large-scale optimization problems, the Hessians of the objective and constraints are sparse or otherwise structured.

3 Building Blocks

In this section, we discuss three building blocks that together comprise the algorithm we implement in ADIFOR for computing second derivatives:

1. Forward-mode Hessians,

2. Interpolation utilizing forward-mode univariate Taylor series, and

3. Preaccumulation.

Globally at the level of the entire function being differentiated, we can choose either alternative 1 or alternative 2. The second alternative is preferred because it can be used to exploit the sparsity often present in Hessian matrices, it parallelizes and vectorizes, and it generalizes to higher derivatives.

Locally for complicated right-hand sides of assignment statements, we can choose the size of units for which the univariate Taylor series are propagated. We can parse each complicated expression into an equivalent sequence of unary and binary operations as the discussion of alternative 2 in Section 3.2 suggests. Preaccumulating local derivatives as discussed in Section 3.3 allows us to propagate series at the level of the statements in the original code, rather than to the smaller units of binary operations. Preaccumulating local derivatives of complicated expressions saves storage space, code size, and execution time. Eventually, we will generalize the preaccumulation technique to Fortran functions and to some subroutines and basic blocks.

In the rest of this section, we examine in detail the three building blocks listed above.

3.1 Forward-Mode Hessians

One could use the forward mode of automatic differentiation to compute the gradient and the dense Hessian of f by propagating the first- and second-derivative objects strictly in the forward mode [9]. We describe how this would be done to show that the combination of preaccumulation and interpolation yields much more efficient code.

3.1.1 Example – Multiplication

Suppose that u and v are active variables (they depend on values of independent variables). The values of ∇u, ∇v, $\nabla^2 u$, and $\nabla^2 v$ have been computed along with the values for u and v. As an

example of a typical operation, suppose that $f = f(u, v) = u \cdot v$. Then by the chain rule, we have

$$\begin{aligned} f &= uv \\ \nabla f &= u \cdot \nabla v + \nabla u \cdot v \\ \nabla^2 f &= u \cdot \nabla^2 v + \nabla u \cdot (\nabla v)^T + \nabla v \cdot (\nabla u)^T + v \cdot \nabla^2 u. \end{aligned} \tag{1}$$

Table 1 gives the computational complexity for the $\times$ operator.

Table 1. Computational complexity of the $\times$ operator

Cost	+'s	$\times$'s
Function	0	1
Gradient	n	$2n$
Hessian	$1.5n(n+1)$	$2n(n+1)$

The complexity of the other operators is similar, differing only in the constants. The storage complexity for the naive forward propagation of ∇f and $\nabla^2 f$ is proportional to $n^2/2$ times the storage required for computing f. The time and storage complexity for the naive forward propagation contrasts sharply with the corresponding complexities for the univariate Taylor series whose complexities are a small multiple of (the number of nonzero elements of $\nabla^2 f$) $\times$ (the corresponding costs for f).

The alternative of overall reverse-mode propagation of adjoint values [8] is attractive for computing gradients, but for the highly structured Hessians and higher-order derivatives, the global application of the forward mode is satisfactory. We avoid the overhead of run-time recording each operation, while retaining the flexibility to apply compile-time reversal of complicated expressions and eventually some basic blocks of code. The code generated by ADIFOR uses a hybrid of the forward and the reverse modes at the statement level.

3.1.2 Example – Short Subroutine

Here we give a more complete example of the forward propagation of dense Hessians. The ADIFOR-generated code includes many code optimizations.

Suppose that the original subroutine `fcn` provided by the user for the computation of a function $f : x \in \mathbf{R}^n \mapsto f \in \mathbf{R}$ contains an active variable `u`. For the present discussion, we assume that `p` = `pmax` = n. Then the ADIFOR-generated variables `g$u` and `h$u` in `h$fcn` contain

$$\texttt{g\$u(j)} := \frac{\partial \texttt{u}}{\partial x_j}, \quad \text{for } j = 1(1)p$$

$$\texttt{h\$u(j,i)} := \frac{\partial^2 \texttt{u}}{\partial x_j \partial x_i}, \quad \text{for } j = 1(1)p,\ i = 1(1)j$$

We illustrate the code to be generated by ADIFOR by a simple example similar to that used in [3] to motivate the hybrid mode for first derivative objects.

Consider the subroutine in Listing 1.

```
subroutine fcn (x, xdim, f, fdim)
integer xdim, fdim
real x(xdim), f(fdim)
f(1) = -x(1) / (x(2) * x(3) * x(4))
return
end
```

Listing 1. `Subroutine fcn`

We nominate `x` as an independent variable and `f` as a dependent variable. In this example, there are $n = 4$ independent variables.

The raw, unoptimized code segment for computing the gradient in the hybrid mode is shown in Listing 2. The complete subroutine `g$fcn$3`, the subordinate subroutine `saxpy4`, and a main program comparing the Jacobian computed by `g$fcn$3` given in Listing 2 with the hand-coded Jacobian are included in Appendix A. While this code resembles ADIFOR-generated code, we point out that the actual code generated by ADIFOR is much more efficient than the code in Listing 2. We include this code as a basis for building the Hessian code to follow in Listing 3.

```
      subroutine g$fcn$3 (g$p$, x, g$x, ldg$x, xdim, f, g$f, ldg$f, fdim)
      integer xdim, fdim, g$p$, ldg$x, ldg$f
      real x(xdim), f(fdim), g$x(ldg$x,xdim), g$f(ldg$f,fdim)

C      f(1) = -x(1) / (x(2) * x(3) * x(4))
      r$0 = x(1); r$1 = x(2); r$2 = x(3); r$3 = x(4); r$4 = -r$0
      r$5 = r$1 * r$2; r$6 = r$5 * r$3; r$7 = r$4 / r$6
C                                Initialize adjoints
      r$0bar = r$1bar = r$2bar = r$3bar = r$4bar = r$5bar = r$6bar = 0.0
      r$7bar = 1.0
C                                Adjoint for r$7 = r$4 / r$6
      r$4bar = r$4bar + r$7bar * (1.0 / r$6)
      r$6bar = r$6bar + r$7bar * (-r$7 / r$6)
C                                Adjoint for r$6 = r$5 * r$3
      r$5bar = r$5bar + r$6bar * r$3
      r$3bar = r$3bar + r$6bar * r$5
C                                Adjoint for r$5 = r$1 * r$2
      r$1bar = r$1bar + r$5bar * r$2
      r$2bar = r$2bar + r$5bar * r$1
C                                Adjoint for r$4 = -r$0
      r$0bar = r$0bar + r$4bar * (-1.0)

      call saxpy4 (pmax, g$p$, r$0bar, g$x(1,1), r$1bar, g$x(1,2),
     +             r$2bar, g$x(1,3), r$3bar, g$x(1,4), g$f(1,1))
      f(1) = r$7
      return
      end
```

Listing 2. Forward mode code for the gradient

The gradient object `g$x` is initialized to an $n \times n$ identity matrix.

The code that might be generated by ADIFOR to compute both the gradient and the dense Hessian in the forward mode is shown in Listing 3.

```
      subroutine g$fcn$3 (g$p$, x, g$x, h$x, ldg$x, xdim, f, g$f, h$f, ldg$f, fdim)
      integer xdim, fdim
      real x(xdim), f(fdim)
      integer g$p$, pmax, ldg$x, ldg$f, g$i$, g$j$
      parameter (pmax = 4)
      real g$x(ldg$x,xdim), h$x(ldg$x,ldg$x,xdim), g$f(ldg$f,fdim), h$f(ldg$f,ldg$f,fdim),
     +     r$4, g$r$4(pmax), h$r$4(pmax,pmax), r$5, g$r$5(pmax), h$r$5(pmax,pmax),
     +     r$6, g$r$6(pmax), h$r$6(pmax,pmax), r$7, g$r$7(pmax), h$r$7(pmax,pmax), r$8

c  Storage:
c              partial f_k
c     ----------------------  = h$f (j, i, k)
c     partial x_j partial x_i

C  f(1) = -x(1) / (x(2) * x(3) * x(4))
      r$4 = -x(1)
      do 99990 g$j$ = 1, g$p$
```

```
         g$r$4(g$j$) = - g$x(g$j$,1)
         do 99990 g$i$ = g$j$, g$p$
            h$r$4(g$j$,g$i$) = - h$x(g$j$,g$i$,1)
99990 continue

      r$5 = x(2) * x(3)
      do 99980 g$j$ = 1, g$p$
         g$r$5(g$j$) = x(2) * g$x(g$j$,3) + g$x(g$j$,2) * x(3)
99980 continue
      do 99982 g$j$ = 1, g$p$
         do 99982 g$i$ = g$j$, g$p$
            h$r$5(g$j$,g$i$)
     +             = x(2) * h$x(g$j$,g$i$,3) + g$x(g$i$,2) * g$x(g$j$,3)
     +               + g$x(g$j$,2) * g$x(g$i$,3) + h$x(g$j$,g$i$,2) * x(3)
99982 continue

      r$6 = r$5 * x(4)
      do 99970 g$j$ = 1, g$p$
         g$r$6(g$j$) = r$5 * g$x(g$j$,4) + g$r$5(g$j$) * x(4)
99970 continue
      do 99972 g$j$ = 1, g$p$
         do 99972 g$i$ = g$j$, g$p$
            h$r$6(g$j$,g$i$)
     +             = r$5* h$x(g$j$,g$i$,4) + g$r$5(g$i$) * g$x(g$j$,4)
     +               + g$r$5(g$j$) * g$x(g$i$,4) + h$r$5(g$j$,g$i$) * x(4)
99972 continue

C  r$7 = r$4 / r$6
      r$8 = 1.0 / r$6
      r$7 = r$4 * r$8
      do 99960 g$j$ = 1, g$p$
         g$r$7(g$j$) = (g$r$4(g$j$) - g$r$6(g$j$) * r$7) * r$8
99960 continue
      do 99962 g$j$ = 1, g$p$
         do 99962 g$i$ = g$j$, g$p$
            h$r$7(g$j$,g$i$)
     +             = (h$r$4(g$j$,g$i$) - (g$r$6(g$i$) * g$r$7(g$j$)
     +               + g$r$6(g$j$) * g$r$7(g$i$) + h$r$6(g$j$,g$i$) * r$7)) * r$8
99962 continue

      f(1) = r$7
      do 99950 g$j$ = 1, g$p$
         g$f(g$j$,1) = g$r$7(g$j$)
         do 99950 g$i$ = g$j$, g$p$
            h$f(g$j$,g$i$,1) = h$r$7(g$j$,g$i$)
99950 continue

      return
      end
```

Listing 3. Forward mode code for the Hessian

The Hessian object **h$x** is initialized to a $n \times n \times n$ zero matrix because $\frac{\partial^2 x_k}{\partial x_j \partial x_i} = 0$ for all k, j, and i.

Next, we give code for some operators and elementary functions. We generate the first and second derivative objects strictly in the forward mode. This is not the code we will eventually generate, but it is necessary to formulate this code in order to evaluate the relative merits of partial derivatives versus univariate Taylor series for computing dense Hessians (see Section 3.2).

The setting for the operators for computing dense Hessians as their constituent partial derivatives is this: We assume that the user's original code has been parsed into a sequence of assignment statements (as in Listing 4) involving only unary or binary operations or elementary functions.

```
      r$0 = u + v
      r$1 = u * v
      r$2 = u / v
      r$3 = exp (v)
```

Listing 4. Code parsed to unary or binary operations

The variables `u` and `v` are active. We use `exp` as the prototype for all elementary functions for the purpose of specifying code for the operators. When we have evaluated alternatives and settled on a plan for Hessian calculation, then we will give the code for all elementary functions. Listing 5 shows the code for multiplication. A complete program including the code for `+`, `*`, `/`, and `exp` is included as Appendix B.

```
c  MULTIPLICATION: f = u * v
c
c    df         dv     du
c    --  =  u * --  +  -- * v
c    dx         dx     dx
c
c      2            2                                  2
c     d f          d v      du   dv    du   dv       d u
c    -----  =  u * -----  +  -- * --  + -- * --  +  ----- * v
c    dx dy         dx dy     dy   dx    dx   dy     dx dy
c
      r$1 = u * v
      do g$j$ = 1, p
         g$r$1(g$j$) = u * g$v(g$j$) + g$u(g$j$) * v
         do g$i$ = 1, g$j$
            h$r$1(g$j$,g$i$) = u * h$v(g$j$,g$i$) + g$u(g$i$) * g$v(g$j$)
     +                       + g$u(g$j$) * g$v(g$i$) + h$u(g$j$,g$i$) * v
         end do
      end do
```

Listing 5. Multiply operators for forward mode dense Hessians as partial derivatives

Table 2 gives the computational complexity of the $\times$ operator.

Table 2. Computational complexity of the $\times$ operator

Cost	+'s	$\times$'s
Function	0	1
Gradient	p	$2p$
Hessian	$1.5p(p+1)$	$2p(p+1)$

The complexity of the other operators is similar, differing only in the constants.

3.2 Interpolation Utilizing Forward-mode Univariate Taylor Series

As an alternative to the forward-mode propagation of Hessian matrices at the global level of the entire function being differentiated, we prefer to compute second-order partial derivatives by interpolation utilizing forward-mode univariate Taylor series. The mathematical theory of recovering high-order mixed partial derivatives from values propagated as univariate Taylor series is given in [5]. Here, we outline the ideas and sketch an implementation.

3.2.1 Interpolation

Suppose we have a program that evaluates a scalar function $w = f(u, v)$ with two independent variables u and $v \in \mathbf{R}$. In agreement with the design philosophy of ADIFOR, we consider differentiation with respect to a vector of $n = 2$ parameters x and y that are not necessarily the same as u and v. Denoting partial differentiation by subscripts, we will now try to calculate the 6-tupel

$$w, w_x, w_y, w_{xx}, w_{xy}, w_{yy}$$

on the basis of the user supplied data

$$u, u_x, u_y, u_{xx}, u_{xy}, u_{yy} \text{ and } v, v_x, v_y, v_{xx}, v_{xy}, v_{yy}.$$

In other words, the scalar arguments u and v have been replaced by the quadratic polynomials

$$\begin{aligned} P_u(x, y) &= u + u_x x + u_y y + 0.5u_{xx}x^2 + u_{xy}xy + 0.5u_{yy}y^2, \text{ and} \\ P_v(x, y) &= v + v_x x + v_y y + 0.5v_{xx}x^2 + v_{xy}xy + 0.5v_{yy}y^2. \end{aligned}$$

We are trying to calculate the polynomial

$$P_w(x, y) = w + w_x x + w_y y + 0.5w_{xx}x^2 + w_{xy}xy + 0.5w_{yy}y^2$$

that satisfies

$$f(u(x, y), v(x, y)) = P_w(x, y) + \mathcal{O}(x^3 + y^3).$$

The straight-forward way of achieving this goal is to propagate the 6-tupels representing first and second derivatives with respect to x and t through the program that defines f. The storage per intermediate scalar variable is simply $6 = \binom{4}{2}$, and the cost of a convolution is $15 = \binom{6}{2}$ arithmetic operations. Hence, we may assume that the run-time of the code in polynomial arithmetic will be roughly 15 times slower than the evaluation of the function itself.

Next, suppose we wish to determine P_w by propagating only univariate Taylor series through the program. The input expansions

$$u(x) = u + u_x x + 0.5u_{xx}x^2 \text{ and } v(x) = v + v_x x + 0.5v_{xx}x^2$$

yield the coefficients w, w_x and w_{xx}. Differentiating along the y axis yields w_y and w_{yy}. The only coefficient missing is the cross term w_{xy}. To obtain it, we can differentiate along the diagonal by setting $x = y = s$ for a third differentiation parameter s. The input polynomials

$$\begin{aligned} u(s) &= u + (u_x + u_y)s + (u_{xy} + 0.5u_{xx} + 0.5u_{yy})s^2, \text{ and} \\ v(s) &= v + (v_x + v_y)s + (v_{xy} + 0.5v_{xx} + 0.5v_{yy})s^2 \end{aligned}$$

yield some expansion

$$w(s) = f(u(s), v(s)) = w + \alpha s + \beta s^2 + \mathcal{O}(s^3).$$

By using the chain rule, the coefficients α and β satisfy the indentities

$$\begin{aligned} \alpha &= w_s = w_x + w_y, \text{ and} \\ \beta &= w_{ss}/2 = w_{xy} + 0.5w_{xx} + 0.5w_{yy}. \end{aligned}$$

Thus, we can calculate the missing cross term as

$$w_{xy} = \beta - 0.5(w_{xx} + w_{yy}).$$

This is a simple instantiation of the general interpolation procedure for an arbitrary number of independent variables and for arbitrary order mixed partial derivatives described in [5].

3.2.2 Forward-Mode Univariate Taylor Series

Here, we consider how univariate Taylor series provide the values required to compute dense Hessians by the interpolation scheme outlined in Section 3.2.1. The complexity of the operators is very similar to the complexity of the operators for full, dense Hessians described in Section 3.1.

To illustrate how the interpolation scheme works in the special case of second partial derivatives, suppose that x and y are independent variables. Let $s := x + y$. If $f = f(s) = f(x, y)$, then

$$\begin{aligned}
\frac{df}{ds} &= \frac{\partial f}{\partial x} * \frac{\partial x}{\partial s} + \frac{\partial f}{\partial y} * \frac{\partial y}{\partial s} \\
&= \frac{\partial f}{\partial x} * 1 + \frac{\partial f}{\partial y} * 1 \\
\frac{d^2 f}{ds^2} &= \frac{\partial}{\partial s}\left[\frac{\partial f}{\partial x}\right] + \frac{\partial}{\partial s}\left[\frac{\partial f}{\partial y}\right] \\
&= \frac{\partial^2 f}{\partial x^2} * \frac{\partial x}{\partial s} + \frac{\partial^2 f}{\partial x \partial y} * \frac{\partial y}{\partial s} + \frac{\partial^2 f}{\partial x \partial y} * \frac{\partial x}{\partial s} + \frac{\partial^2 f}{\partial y^2} * \frac{\partial y}{\partial s} \\
&= \frac{\partial^2 f}{\partial x^2} + 2 * \frac{\partial^2 f}{\partial x \partial y} + \frac{\partial^2 f}{\partial y^2}.
\end{aligned}$$

Hence, we expand the Taylor series for f with respect to x, y, and $s = x + y$, all at the same expansion point (whose value is suppressed in the notation for clarity):

Table 3. Storage structure for `h$f`

	`f`	`f' = g$f(•)`	`f'' = h$f(•)`
At x :	f	$\frac{\partial f}{\partial x}$	$\frac{\partial^2 f}{\partial x^2}$
At s :	f	$\frac{\partial f}{\partial s}$	$\frac{\partial^2 f}{\partial s^2}$
At y :	f	$\frac{\partial f}{\partial y}$	$\frac{\partial^2 f}{\partial y^2}$

The series for x and for y yield the gradient and the diagonal entries in the Hessian. The off-diagonal entry is

$$\frac{\partial^2 f}{\partial x \partial y} = 0.5 * \left(\frac{\partial^2 f}{\partial s^2} - \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right)\right). \tag{2}$$

We can view the computations implied by Table 3 as vector instructions to be executed for each Taylor series in the table. Alternatively, the number of operations required to compute the values in the second column equals the number of independent variables since $\mathtt{f}'(2) = \mathtt{f}'(1) + \mathtt{f}'(3)$. The number of operations required to compute the values in the third column equals the number of nonzero elements in the Hessian matrix. With these storage optimizations, the storage and operations required by the univariate Taylor polynomials are $1 + n + n * (n + 1)/2$, which is *exactly* the same storage and operations required for f, ∇f, and Hessian (f) in the full, dense mode.

Next, we look at the operators for sets of univariate Taylor polynomials in the hope that they are simpler than the corresponding operators for full, dense Hessians described in Section 3.1.

If the function whose Hessian is sought has n independent variables, then we must compute $n(n + 1)/2$ univariate Taylor series corresponding to the number of possibly distinct entries in the Hessian. If the Hessian is sparse, we need only propagate univariate Taylor series for the nonzero entries in the Hessian. We order the index of `f'` and `f''` as suggested by Table 3 in the column-major order of the lower triangular part of the Hessian matrix.

Listing 6 shows the code for the multiplication operation. A complete program including the code for `+`, `*`, `/`, and `exp` is included as Appendix C.

```
c  MULTIPLICATION: f = u * v
c
c    df         dv     du
c    --  =  u * --  +  -- * v
c    dx         dx     dx
c
c     2           2                       2
c    d f         d v         du   dv     d u
c    ---  =  u * ---  +  2 * -- * --  +  --- * v
c      2           2                       2
c    dx          dx          dx   dx     dx
c
c    We divide both sides by 2 to store the Taylor coefficient.
c
      r$1 = u * v
      do g$j$ = 1, p
         g$r$1(g$j$) = u * g$v(g$j$) + g$u(g$j$) * v
         h$r$1(g$j$) = u * h$v(g$j$) + 2 * g$u(g$j$) * g$v(g$j$) + h$u(g$j$) * v
      end do
```

Listing 6. Multiplication operator for forward Hessians as univariate series

Table 4 gives the computational complexity of the univariate Taylor $\times$ operator, assuming the Hessian matrix is dense.

Table 4. Maximum computational complexity of the univariate Taylor $\times$ operator

Cost	+'s	$\times$
Function	0	1
Gradient	n	$2n$
Hessian	$n(n+1)$	$1.5n(n+1)$

The complexity of the other operators is similar, differing only in the constants. In addition, there is a one-time cost associated with constructing the off-diagonal elements of the Hessian according to Equation (2).

We prefer the technique of interpolation utilizing forward-mode univariate Taylor series to the forward-mode propagation of Hessian matrices (Section 3.1) for implementation in ADIFOR because interpolation

- handles sparse Hessians by generating series only for nonzero entries,
- handles very large Hessians by generating elements in multiple sweeps,
- can generate arbitrary elements with little redundant computation,
- parallelizes and vectorizes,
- uses simple data structures – scalars and vectors, rather than symmetric matrices,
- is easier to understand when coding individual operators, and
- generalizes to higher derivatives.

3.3 Preaccumulation

The discussion in Section 3.2 of interpolation assumed that complicated expressions appearing on the right-hand side of assignment statements are parsed into an equivalent sequence of unary and binary operations. In this section, we show how the preaccumulation of local gradients and Hessians of complicated expressions yields savings of storage space, code size, and execution time by propagating Taylor series at the level of statements in the original code, rather than at the smaller level of binary operations.

Let the variables u and v depend on a vector x of independent variables. The first and second derivatives ∇u, ∇v, $\nabla^2 u$, and $\nabla^2 v$ are available from earlier computations. If $w = f(u, v)$, the chain rule tells us that

$$\begin{aligned} \nabla w &= \frac{\partial w}{\partial u} \cdot \nabla u + \frac{\partial w}{\partial v} \cdot \nabla v, \text{ and} \\ \nabla^2 w &= \frac{\partial w}{\partial u} \cdot \nabla^2 u + \frac{\partial w}{\partial v} \cdot \nabla^2 v \\ &\quad + \frac{\partial^2 w}{\partial u^2} \cdot (\nabla u)^2 + 2\frac{\partial^2 w}{\partial u \partial v} \cdot \nabla u \cdot \nabla v + \frac{\partial^2 w}{\partial v^2} \cdot (\nabla v)^2. \end{aligned} \tag{3}$$

Hence, if we know the "local" derivatives $(\frac{\partial w}{\partial u}, \frac{\partial w}{\partial v})$ and $(\frac{\partial^2 w}{\partial u^2}, \frac{\partial^2 w}{\partial u \partial v}, \frac{\partial^2 w}{\partial v^2})$ of w with respect to v and u, we can easily compute ∇w and $\nabla^2 w$, the derivatives of w with respect to x. An example of Equation (3) is given in Equation (2) for the simple case $w = f(u, v) = u \cdot v$. Equation (3) for propagating Taylor series has the much simpler form given by Equation (5).

The idea is that the large "global" derivatives ∇w are propagated in the forward mode from one assignment statement to another, while the scalar "local" derivatives $(\frac{\partial w}{\partial u}, \frac{\partial w}{\partial v})$ are preaccumulated independently of the larger flow of control from one statement to the next. ADIFOR was the first tool for automatic differentiation to use preaccumulation of local derivatives by applying the reverse mode at the statement level for the efficient computation of first derivatives [3,6]. The hierarchy of "local" and "global" derivatives extends to higher-order derivatives.

If $w = f(s_1, \ldots, s_k)$, let ∇f and $\nabla^2 f$ denote the "local" gradient and Hessian, respectively, of f with respect to $s_1, \ldots, s_k$. If we extend Equation (3) to complicated right-hand sides, we get

$$\begin{aligned} w &= f(s_1, \ldots, s_k) \\ w' &= \sum_{i=1}^{k} (\nabla f)_i \cdot {s_i}' \\ &= \nabla f^T \cdot s' \end{aligned} \tag{4}$$

$$\begin{aligned} w'' &= \sum_{i=1}^{k} \left[(\nabla f)_i \cdot {s_i}'' + {s_i}' \cdot \sum_{j=1}^{k} \left[(\nabla^2 f)_{i,j} \cdot {s_j}' \right] \right] \\ &= \nabla f^T \cdot s'' + s'^T \cdot \nabla^2 f \cdot s'. \end{aligned} \tag{5}$$

Equation (5) represents derivatives in each of the p directions, which may be computed in parallel.

The important point to note in Equation (5) is that there are only two vector loops of length p, independent of the numer of variables or operations on the right-hand side of the assignment statement. The local k-element gradient ∇f and the local k^2-element Hessian $\nabla^2 f$ can be computed in any manner. We may apply preaccumulation again to less complicated subfunctions, or we may use the forward mode, the reverse mode, a combination of the two, or analytic formulas, if they are easy to derive.

4 How ADIFOR Generates Code for Second Derivatives

The central insight for the implementation in ADIFOR of the code for second derivatives is

ADIFOR uses the reverse mode at the statement level to generate code for computing ∇f. By essentially applying ADIFOR again to that generated code, we obtain code for $\nabla^2 f$. The result is code for the preaccumulation of local derivatives.

In this section, we outline how this central insight is implemented in ADIFOR. In the following section, we give an example.

Since the number of independent variables is known at compile-time, extensive scalar code optimizations can be applied in the computation of ∇f. In particular, if all loops are completely unrolled, we prune many computations by exploiting the symmetry in $\nabla^2 f$. But even ignoring the symmetry is not a big issue, since k (the number of active variables appearing on the right hand side of an assignment statement in the user's original code) is usually quite small. In this way, generating the code for computing second derivatives is just an application of the current ADIFOR technology.

In generating the code for Equations (4) and (5), we perform the same kinds of optimizations that we are doing now concerning zeros and ones.

In the code generated by ADIFOR, we

- propagate three-term Taylor series, one series for each nonzero element in the Hessian.
- propagate series in an overall forward mode similar to current gradients.
- For each composite assignment statement
  - generate gradient code for that assignment,
  - pass the generated code to ADIFOR "recursively", and
  - integrate the results.

In general, let us consider an assignment statement with k variables on the right hand side

```
w = f (s1, s2, ..., sk)
```

We wish to transform the code for the assignment statement into code to propagate the first and second derivative objects w' and w''. The tasks for the code transformation algorithm are:

Task 1: Parse the expression on the right hand side into a sequence of m simple assignment statements consisting of at most unary or binary operators or elementary functions:

```
Block 1:
      r$0 = s1
      . . .
      r$m = ...
      w = r$m
```

Task 2: Generate and store the code for the appropriate adjoint objects for the code from `Block 1` in reverse mode:

```
Block 2:
c      r$m = ...
      r$?$bar = r$?$bar + r$m$bar * ...
      r$?$bar = r$?$bar + r$m$bar * ...
      ...
      s1$bar = ...
      ...
      sk$bar = ...
```

For each assignment statement in `Block 1`, we generate one or two statements incrementing a `bar` object. Somewhere in `Block 2`, there must be at least one statement incrementing the `bar` object associated with each of the variables `s1`, `s2`, ..., `sk`.

Task 3: For each variable `x` appearing on the left hand side of an assignment statement in `Block 1` or in `Block 2`, declare a variable `h$x(k)`, (where k is the number of variables on the right hand side of the assignment statement being processed.

Task 4: Call ADIFOR "recursively." That is, take the assignment statements in `Block 1` followed by the assignment statements in `Block 2`, parse them, and generate code for the appropriate adjoint objects in reverse mode. The application of ADIFOR to this code is simpler than in the general case because all assignment statements are *already* parsed into a form with at most a unary or a binary operation or an elementary function, except that the assignment statements for the `bar` object in `Block 2` have a special form with two binary operations `+` and `*`. However, the form of the `bar` assignments is known in advance. For each assignment statement of the form

```
r$j = f (r$1, r$2)
```

in `Block 1`, we generate code of the form

```
c     r$j = f (r$1, r$2)
      do g$i = 1, k
         h$r$j(g$i$) = f_{r$2} * h$r$1(g$i$) + f_{r$1} * h$r$2(g$i$)
      end do
      r$j = f (r$1, r$2)
```

For each assignment statement of the form

```
r$j$bar = r$j$bar + r$1$bar * x
```

in `Block 2`, we generate code of the form

```
c     r$j$bar = r$j$bar + r$1$bar * x
      do g$i = 1, k
         h$r$j$bar(g$i$) = h$r$j$bar(g$i$) + x * h$r$1$bar(g$i$)
                           + r$1$bar * h$x(g$i$)
      end do
      r$j$bar = r$j$bar + r$1$bar * x
```

Task 5: Generate the final loop:

```
do g$i = 1, g$p$
   g$w(g$i$) = s1$bar * g$s1(g$i$) + s2$bar * g$s2(g$i$)
               + ... + sk$bar * g$sk(g$i$)
   h$w(g$i$) = s1$bar * h$s1(g$i$) + s2$bar * h$s2(g$i$)
                  + ... + sk$bar * h$sk(g$i$)
                  + h$s1$bar(1) * g$s1(g$i$)**2
                  + h$s2$bar(2) * g$s2(g$i$)**2
                  + ... + h$sk$bar(k) * g$sk(g$i$)**2
                  + 2.0 * g$s1(g$i$)
                    * (h$s1$bar(2) * g$s2(g$i$)
                       + ... + h$s1$bar(k) * g$sk(g$i$))
                  + 2.0 * g$s2(g$i$)
                    * (h$s2$bar(3) * g$s3(g$i$)
                       + ... + h$s2$bar(k) * g$sk(g$i$))
                  + ... + 2.0 * g$s{k-1}(g$i$)
                          * (h$s{k-1}$bar(k) * g$sk(g$i$))
end do
w = r$m
```

The second assignment inside the `do` loop implements Equation (4); the third implements Equation (5). We might choose to call a subroutine (different for each value of k), but calling a subroutine interferes with code optimization.

Task 6: Apply code optimization. Then write the resulting code.

5 Example of the Generated Code

As an example of the tasks that ADIFOR must perform to generate code to compute second-order derivatives, we take the assignment statement

```
      w = -y / (z * z * z)
```

used as an example in [3] to motivate the generation of code for the hybrid mode. We proceed in incremental steps from a simple subroutine containing this assignment statement to the final subroutine illustrating the code to be generated by ADIFOR. We give the relevant code fragments in the text and relegate listings of the complete programs to appendixes at the end of the paper.

We emphasize that the steps described here are steps to understanding the code to be generated by ADIFOR. We are doing by hand what we expect ADIFOR to do automatically. In operation, the generation of code for second derivatives by ADIFOR is as transparent to the user as running ADIFOR for first derivatives.

5.1 Step 1. Write original code

Listing 7 shows a subroutine containing the example assignment statement. A driving program to call subroutine `examp2` is given in Appendix D.

```
      subroutine examp2 (x, xdim, f)
      integer xdim
      real x(xdim), y, z, w

c     y and z depend in some way on x(1..xdim)
      y = x(1)
      z = x(2)

c  Consider the assignment statement
      w = -y / (z * z * z)

c     f depends in some way on w
      f = w

      return
      end
```

Listing 7. Original code for example assignment statement

5.2 Step 2. Run ADIFOR on `examp2_dr.f` + `examp2.f`

Our intention is to apply ADIFOR to the code generated by ADIFOR. Hence, the second step is to

1. nominate `x` as an independent variable,
2. nominate `f` as an dependent variable,
3. set `pmax` = 4 (the number of locations in `x`), and

4. run ADIFOR on `examp2_dr.f` + `examp2.f` to generate `examp2.5.f`.

Listing 8 shows the portion of `examp2.5.f` that generates the first derivative objects for the example assignment statement. The complete subroutine `examp2.5.f` is given in Appendix E.

```
C       Consider the assignment statement
C       w = -y / (z * z * z)
        r$1 = z * z
        r$2 = r$1 * z
        r$3 = -y / (r$2)
        r$2bar = (-r$3 / (r$2))
        r$1bar = r$2bar * (z)
        zbar = r$2bar * (r$1)
        zbar = zbar + r$1bar * z
        zbar = zbar + r$1bar * z
        ybar = -(1.0d0 / r$2)
        do 99993 g$i$ = 1, g$p$
          g$w(g$i$) = ybar * g$y(g$i$) + zbar * g$z(g$i$)
99993   continue
        w = r$3
```

Listing 8. ADIFOR-generated first derivative code

5.3 Step 3. Run ADIFOR-generated code

As a check on correct programming, we run the ADIFOR-generated code `examp2.5.f` with its driver `examp2_grad.f` (see Appendix E) and get the correct results shown in Listing 9.

```
 ADIFOR-generated code.
    F      =   -0.12500
    grad F =  -1.250000E-01  1.875000E-01  0.000000E+00  0.000000E+00
```

Listing 9. Results from ADIFOR-generated first derivative code

5.4 Step 4. Extract ADIFOR-generated Code for Assignment

Our intention is to implicitly pass to ADIFOR the code it has previously generated for each right hand side. That is, the recursive ADIFOR call is repeated for each assignment statement.

Here, we simulate a recursive ADIFOR call by extracting from `examp2.5.f` the first derivative code for only the example assignment statement under study here. That is, we extract the code shown in Listing 8 from `examp2.5.f` and place it into a subroutine of its own. We add parameters and variable declarations as appropriate. The resulting subroutine `examp2G` is shown in Listing 10. The function of subroutine `examp2G` is to compute the *local* gradient of `w` with respect to the variables `y` and `z` that appear on the right hand side of the example assignment statement. These local derivatives will be assembled later to form the global derivatives of `w` with respect to the independent variables `x` according to Equations (4) and (5).

```
      subroutine examp2G (g$p$, y, g$y, z, g$z, w, g$w)

        integer g$p$, g$pmax$, g$i$
        parameter (g$pmax$ = 4)
        real r$2bar, r$1bar, ybar, zbar, r$1, r$2, r$3
        real y, z, w
        real g$y(g$pmax$), g$z(g$pmax$), g$w(g$pmax$)

C       Consider the assignment statement
```

```
C       w = -y / (z * z * z)
        r$1 = z * z
        r$2 = r$1 * z
        r$3 = -y / (r$2)
        r$2bar = (-r$3 / (r$2))
        r$1bar = r$2bar * (z)
        zbar = r$2bar * (r$1)
        zbar = zbar + r$1bar * z
        zbar = zbar + r$1bar * z
        ybar = -(1.0d0 / r$2)
        do 99993 g$i$ = 1, g$p$
          g$w(g$i$) = ybar * g$y(g$i$) + (zbar * g$z(g$i$))
99993   continue
        w = r$3

        return
      end
```

Listing 10. Subroutine for computing local gradient

5.5 Step 5. Run ADIFOR on `examp2G_dr.f` + `examp2G.f`

We wish to apply ADIFOR to the subroutine `examp2G.f` shown in Listing 10. Due to known limitations of the current ADIFOR implementation, we had to make the following modifications:

1. replace `1.0d0` by `1.0` (Fortran knows about type coersion, but ADIFOR does not),
2. replace `$` by `Q` (xadifor recognizes only characters in the official Fortran character set), and
3. replace `bar` by `B` (ADIFOR can generate variables whose names conflict with variables already present in the code).

These limitations will be removed in subsequent versions of ADIFOR. Then we wrote a driver and ran the resulting code (see Appendix F) to verify correct programming.

We are now ready for the recursive application of ADIFOR. The assignment statements in Listing 10 are relatively simple. Hence, the code generated by ADIFOR is much simpler than for the general case of complicated right hand sides. This relative simplicity allows ADIFOR to perform further code optimizations not illustrated here. To apply ADIFOR the second time, we

1. nominate `y` and `z` as independent variables,
2. nominate `gqw` (renamed `g$w`) as the dependent variable,
3. set `pmax` = 2 (the number of variables on the right hand side of the example assignment statement), and
4. run ADIFOR on `examp2G_dr.f` + `examp2G_.f` to generate `examp2g.74.f`.

Listing 11 shows the portion of `examp2g.74.f` that generates the first derivative objects for `gqw`. The complete subroutine `examp2g.74.f` is given in Appendix G. The code to be generated by subsequent versions of ADIFOR will be much more compact because ADIFOR will use the reverse mode on basic blocks, rather than on individual statements as illustrated here.

```
C       Consider the assignment statement
C       w = -y / (z * z * z)
C       rq1 = z * z
        do 99988 g$i$ = 1, g$p$
          g$rq1(g$i$) = (z + z) * g$z(g$i$)
99988   continue
```

```
        rq1 = z * z
C       rq2 = rq1 * z
        do 99987 g$i$ = 1, g$p$
          g$rq2(g$i$) = z * g$rq1(g$i$) + rq1 * g$z(g$i$)
99987   continue
        rq2 = rq1 * z
C       rq3 = -y / rq2
        r$1 = -y / (rq2)
        do 99986 g$i$ = 1, g$p$
          g$rq3(g$i$) = -(1.0d0 / rq2) * g$y(g$i$) + ((-r$1 / (rq2)) * g$rq2(g$i$))
99986   continue
        rq3 = r$1
C       rq2b = -rq3 / rq2
        r$1 = -rq3 / (rq2)
        do 99985 g$i$ = 1, g$p$
          g$rq2b(g$i$) = -(1.0d0 / rq2) * g$rq3(g$i$) + ((-r$1 / (rq2)) * g$rq2(g$i$))
99985   continue
        rq2b = r$1
C       rq1b = rq2b * z
        do 99984 g$i$ = 1, g$p$
          g$rq1b(g$i$) = z * g$rq2b(g$i$) + rq2b * g$z(g$i$)
99984   continue
        rq1b = rq2b * z
C       zb = rq2b * rq1
        do 99983 g$i$ = 1, g$p$
          g$zb(g$i$) = rq1 * g$rq2b(g$i$) + rq2b * g$rq1(g$i$)
99983   continue
        zb = rq2b * rq1
C       zb = zb + rq1b * z
        do 99982 g$i$ = 1, g$p$
          g$zb(g$i$) = g$zb(g$i$) + z * g$rq1b(g$i$) + rq1b * g$z(g$i$)
99982   continue
        zb = zb + rq1b * z
C       zb = zb + rq1b * z
        do 99981 g$i$ = 1, g$p$
          g$zb(g$i$) = g$zb(g$i$) + z * g$rq1b(g$i$) + rq1b * g$z(g$i$)
99981   continue
        zb = zb + rq1b * z
        do 99999, gqiq = 1, gqpq
C         gqw(gqiq) = -1.0 / rq2 * gqy(gqiq) + zb * gqz(gqiq)
          r$0 = -1.0 / (rq2)
          do 99980 g$i$ = 1, g$p$
            g$gqw(g$i$, gqiq) = gqy(gqiq) * (-r$0 / (rq2)) * g$rq2(g$i$)
     * + gqz(gqiq) * g$zb(g$i$)
99980     continue
          gqw(gqiq) = r$0 * gqy(gqiq) + zb * gqz(gqiq)
99993     continue
99999   continue
        w = rq3
        return
      end
```

Listing 11. Code from recursive ADIFOR call

It is important to understand what we have computed. Subroutine `examp2G` computes `gqw`, the local gradient of first derivatives of `w` with respect to `y` and `z`. By instructing ADIFOR to differentiate `gqw` with respect to `y` and `z`, we have generated subroutine `examp2g.74` to compute the local Hessian of `w` with respect to `y` and `z`.

5.6 Step 6. Run ADIFOR-generated Code

As a check on correct programming, we wrote a driver program and called the ADIFOR-generated subroutine `examp2g.74`. The complete code is contained in Appendix G. The local gradient and

Hessian computed are shown in Listing 12.

```
Hessian by Adifor (Adifor (examp2.f)).
  W      =   -0.12500
  grad W =  -1.250000E-01  1.875000E-01
  Hessian W =
   1      0.000000E+00  1.875000E-01
   2      1.875000E-01 -3.750000E-01
```

Listing 12. Local gradient and Hessian computed by `examp2g.74`

5.7 Step 7. Model Code for ADIFOR-generated Second Derivatives

Now we are finally ready to merge the ADIFOR-generated first derivative code in subroutine `examp2.5.f` with the ADIFOR-generated local second derivative code in subroutine `examp2g.74` to get subroutine `examp2H` shown in Listing 13. The subroutine in Listing 13 is essentially the code ADIFOR generates for second derivatives, except that this code contains explanatory comments, and the ADIFOR-generated code benefits from code optimizations not illustrated here. Comments in this code clarify details of the merging process.

```
      subroutine g$examp2$5(g$p$, x, g$x, h$x, ldg$x, xdim, f, g$f, h$f, ldg$f)

C  Purpose:  Explore 2nd derivative code.
C            Hand-written ADIFOR-like Hessian code
C  Author:   George Corliss, 26-FEB-1992
C  Reference:
C            Simple example from Working Note 1, Section 2.
C  Discussion:
C    Merge examp2.5.f (gradient) + examp2g.74.f (local Hessian)
c    g$...  denote global objects
c    h$...  denote objects local to one assignment statement.

C
C       Formal f is active.
C       Formal x is active.
C
        integer g$p$, g$pmax$, g$i$, ldg$f
        parameter (g$pmax$ = 10)
        real r$1bar, r$2bar, ybar, zbar, r$1, r$2, r$3
c Added ADIFOR-like variables:
        real r$4, r$5, r$6
C
        real f, g$f(ldg$f), h$f(ldg$f)
        integer xdim, ldg$x
        real x(xdim), g$x(ldg$x, xdim), h$x(ldg$x, xdim)
        real y, z, w
        real g$y(g$pmax$), h$y(g$pmax$), g$z(g$pmax$), h$z(g$pmax$),
     +       g$w(g$pmax$), h$w(g$pmax$)

c Declarations for local gradient objects
c Dimension is largest number of variables occurring in any RHS
c in which this variable is involved.
      real h$y(2), h$z(2), h$r$1(2), h$r$2(2), h$r$3(2), h$r$2bar(2),
     +     h$r$1bar(2), h$ybar(2), h$zbar(2)

C       y and z depend in some way on x(1..xdim)
C       y = x(1)
        do 99995 g$i$ = 1, g$p$
          g$y(g$i$) = g$x(g$i$, 1)
          h$y(g$i$) = h$x(g$i$, 1)
```

```
99995   continue
        y = x(1)

C       z = x(2)
        do 99994 g$i$ = 1, g$p$
          g$z(g$i$) = g$x(g$i$, 2)
          h$z(g$i$) = h$x(g$i$, 2)
99994   continue
        z = x(2)

C       Consider the assignment statement
C       w = -y / (z * z * z)
c       r$1 = z * z
c       r$2 = r$1 * z
c       r$3 = -y / (r$2)
c       r$2bar = (-r$3 / (r$2))
c       r$1bar = r$2bar * (z)
c       zbar = r$2bar * (r$1)
c       zbar = zbar + r$1bar * z
c       zbar = zbar + r$1bar * z
c       ybar = -(1.0d0 / r$2)
c       do 99993 g$i$ = 1, g$p$
c         g$w(g$i$) = ybar * g$y(g$i$) + (zbar * g$z(g$i$))
c9993   continue
c       w = r$3

c=========================
c g$p$:  Within this block, the variable named g$p$ is renamed
c        to h$p$.  Its value is equal to the number of variables
c        on the rhs.
c h$y, h$z:  Local gradient objects of dimension = h$p$
c h$u = (u_y, u_z)
c
c For each global univariate Taylor series being propagated,
c     w      = f (y, z)
c     w_u    = f_y * y_u + f_z * z_u
c     w'     = f_y * y'  + f_z * z'
c     w_{uu} = f_y * y_{uu} + f_z * z_{uu}
c                + 2 * f_{yz} * y_u * z_u
c                + f_{yy} * (y_u)^2 + f_{zz} * (z_u)^2
c     w''    = f_y * y'' + f_z * z'' + 2 * f_{yz} * y' * z'
c                + f_{yy} * (y')^2 + f_{zz} * (z')^2

c  Initialize objects local to statement:
        h$p$ = 2
        h$y(1) = 1.0
        h$y(2) = 0.0
        h$z(1) = 0.0
        h$z(2) = 1.0

c  Compute ybar = f_y, zbar = f_z
c          h$w  = f_{yy}, f_{yz}, f_{zz}:
C       r$1 = z * z
        do 99988 g$i$ = 1, h$p$
          h$r$1(g$i$) = (z + z) * h$z(g$i$)
99988   continue
        r$1 = z * z
C       r$2 = r$1 * z
        do 99987 g$i$ = 1, h$p$
          h$r$2(g$i$) = z * h$r$1(g$i$) + r$1 * h$z(g$i$)
99987   continue
        r$2 = r$1 * z
C       r$3 = -y / r$2
        r$4 = -y / (r$2)
```

```
        do 99986 g$i$ = 1, h$p$
          h$r$3(g$i$) = -(1.0d0 / r$2) * h$y(g$i$)
     *                  + ((-r$4 / (r$2)) * h$r$2(g$i$))
99986   continue
        r$3 = r$4
c Re-insert the code that was optimized out:
C       ybar = -1.0 / r$2
        r$6 = -1.0 / r$2
        do g$i$ = 1, h$p$
           h$ybar(g$i$) = -(r$6 / r$2) * h$r$2(g$i$)
        end do
        ybar = r$6

C       r$2bar = -r$3 / r$2
        r$5 = -r$3 / (r$2)
        do 99985 g$i$ = 1, h$p$
          h$r$2bar(g$i$) = ybar * h$r$3(g$i$) + ((-r$5 / (r$2)) * h$r$2(g$i$))
99985   continue
        r$2bar = r$5
C       r$1bar = r$2bar * z
        do 99984 g$i$ = 1, h$p$
          h$r$1bar(g$i$) = z * h$r$2bar(g$i$) + r$2bar * h$z(g$i$)
99984   continue
        r$1bar = r$2bar * z
C       zbar = r$2bar * r$1
        do 99983 g$i$ = 1, h$p$
          h$zbar(g$i$) = r$1 * h$r$2bar(g$i$) + r$2bar * h$r$1(g$i$)
99983   continue
        zbar = r$2bar * r$1
C       zbar = zbar + r$1bar * z
        do 99982 g$i$ = 1, h$p$
          h$zbar(g$i$) = h$zbar(g$i$) + z * h$r$1bar(g$i$)
     *                   + r$1bar * h$z(g$i$)
99982   continue
        zbar = zbar + r$1bar * z
C       zbar = zbar + r$1bar * z
        do 99981 g$i$ = 1, h$p$
          h$zbar(g$i$) = h$zbar(g$i$) + z * h$r$1bar(g$i$)
     *                   + r$1bar * h$z(g$i$)
99981   continue
        zbar = zbar + r$1bar * z

c At this point, in order to generate the statement
C         g$w(g$i$) = -1.0 / r$2 * g$y(g$i$) + zbar * g$z(g$i$)
c the compiler must already know that
c         w_y = ybar = -1.0 / r$2
c         w_z = zbar
c Hence, w_{yy} = h$ybar(1)
c        w_{yz} = h$ybar(2) = h$zbar(1)
c        w_{zz} = h$zbar(2)

c Compute global univariate Taylor series:
c     w   = f (y, z)
c     w'  = f_y * y'  + f_z * z'
c     w'' = f_y * y'' + f_z * z'' + 2 * f_{yz} * y' * z'
c               + f_{yy} * (y')^2 + f_{zz} * (z')^2

        do g$i$ = 1, g$p$
           g$w(g$i$) = ybar * g$y(g$i$) + zbar * g$z(g$i$)
           h$w(g$i$) = ybar * h$y(g$i$) + zbar * h$z(g$i$)
     *             + 2.0 * h$ybar(2) * g$y(g$i$) * g$z(g$i$)
     *             + h$ybar(1) * g$y(g$i$) * g$y(g$i$)
     *             + h$zbar(2) * g$z(g$i$) * g$z(g$i$)
        end do
```

```
        w = r$3

c=============================================================

C       f depends in some way on w
        f = w
        do 99992 g$i$ = 1, g$p$
          g$f(g$i$) = g$w(g$i$)
          h$f(g$i$) = h$w(g$i$)
99992   continue
        return
      end
```

Listing 13. Model code for ADIFOR-generated second derivatives

5.8 Step 8. Run the Model Second Derivative Code

When we write a driver program (see Appendix H) and run the merged subroutine `examp2H` shown in Listing 13, we get the correct global gradient and Hessian shown in Listing 14.

```
Series for F     :
          1  -1.250000E-01  -1.250000E-01   0.000000E+00
          2  -1.250000E-01   6.250000E-02   0.000000E+00
          3  -1.250000E-01   1.875000E-01  -3.750000E-01
          4  -1.250000E-01  -1.250000E-01   0.000000E+00
          5  -1.250000E-01   1.875000E-01  -3.750000E-01
          6  -1.250000E-01   0.000000E+00   0.000000E+00
          7  -1.250000E-01  -1.250000E-01   0.000000E+00
          8  -1.250000E-01   1.875000E-01  -3.750000E-01
          9  -1.250000E-01   0.000000E+00   0.000000E+00
         10  -1.250000E-01   0.000000E+00   0.000000E+00

For F     :  value:  -1.250000E-01
Gradient :   -1.250000E-01   1.875000E-01   0.000000E+00   0.000000E+00
Hessian  :
          1   0.000000E+00
          2   1.875000E-01  -3.750000E-01
          3   0.000000E+00   0.000000E+00   0.000000E+00
          4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
```

Listing 14. Global gradient and Hessian from the hand-written second derivative code.

6 Pending Implementation Issues

In this section, we simply mention some issues that remain to be settled with respect to second derivatives.

6.1 Data Structures for p Taylor Series

In what data structures should the p Taylor series be stored?

Alternative 1.1: Three separate objects: value: `x(p)`, first derivative: `g$x(p)` $= x'$, and second derivative `h$x(p)` $= x''$.

Alternative 1.2: Combined array `x(p,0:2)`.

Alternative 1.3: Value `x(p)` and a combined array `x(p,2)` containing the derivative objects.

Suggest: Alternative 1.1.

- Generalizes to higher derivatives
- The h$routine should also return values directly as routine did

6.2 In-line vs Subroutine Call

This paper illustrates the code in a conceptual way. How should it actually be implemented?

Alternative 2.1: Generate code in line.

Alternative 2.2: Call generated subroutine for each right hand side.

Suggest: Alternative 2.1.

- Better code optimization. In particular, the code generated to compute local second derivatives can be heavily optimized.
- Better parallelization and vectorization scope
- Large code, but no bloat in the number of subroutines

The issue here is that the preaccumulation of local derivatives is an "off-line" process with respect to the broader picture of the overall forward-mode propagation of sets of Taylor series at the statement level. However, compiler technology for code optimization transcends this distinction. In Listing 13, the assignment statements `g$w(g$i$) = ...` and `h$w(g$i$) = ...` contain several •`bar` objects. In many computations (computational models based on grids, for example), many of the corresponding •`bar` objects are 0, 1, 2, or another simple expression which is folded into the code using conventional compiler code-folding techniques. Then, subexpressions of the forms $0 + \bullet$, $0 * \bullet$, and $1 * \bullet$ are simplified appropriately before ADIFOR generates the code for computing the derivatives.

For derivatives higher than second order, custom-generated subroutines might be better.

6.3 What Drivers Do We Need?

- Given sparsity pattern, compute Hessian and return in sparse data structure
- Compute dense Hessian
- Compute Hessian $\times$ vector
- Compute Hessian $\times$ matrix

In Appendix I, we give several prototype library utilities:

sereye.f Initialize univariate series for dense Hessian

prtser.f Print univariate series

prthes.f Print value, gradient, Hessian

ser2he.f Convert univariate series to value, gradient, Hessian form

Acknowledgments

We thank Alan Carle for his helpful suggestions regarding higher derivatives and for his essential roles in the ADIFOR development project.

References

[1] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR : Automatic differentiation in a source translation environment. Preprint MCS–P288–0192, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1992. ADIFOR Working Note # 5. Accepted for the International Symposium on Symbolic and Algebraic Computation, July 27–29, 1992, Berkeley, Calif.

[2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Fortran source translation for efficient derivatives. Preprint MCS–P278–1291, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., December 1991. ADIFOR Working Note # 4.

[3] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Generating derivative codes from Fortran programs. *Scientific Computing*, to appear. ADIFOR Working Note # 1. Also appeared as Preprint MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991, and as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Tex. 77251, 1991.

[4] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Memorandum ANL/MCS–TM–159, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1992. ADIFOR Working Note # 3.

[5] Christian Bischof, George Corliss, and Andreas Griewank. Structured second- and higher-order derivatives through univariate Taylor series. Preprint MCS–P296–0392, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992. ADIFOR Working Note # 6.

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

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

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

[9] Louis B. Rall. Differentiation in Pascal-SC: Type GRADIENT. *ACM Trans. Math. Software*, 10(2):161 – 184, June 1984.