ADIFOR Working Note #8: Hybrid Evaluation of Second Derivatives in ADIFOR

Christian Bischof
George Corliss
Andreas Griewank

CRPC-TR92238
May 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

ANL/MCS-TM-166

ADIFOR Working Note #8:
Hybrid Evaluation of Second Derivatives in ADIFOR

by

Christian Bischof, George Corliss, and Andreas Griewank

May 1992

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Contents

ADIFOR Working Note #8:

Hybrid Evaluation of Second Derivatives in ADIFOR

by

Christian Bischof, George Corliss, and Andreas Griewank

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 in 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.

2 Need for Second Derivatives

The primary motivation for adding to ADIFOR the ability to compute second derivatives comes from optimization. Given $f : \mathrm{R}^n \mapsto \mathrm{R}$, unconstrained optimization algorithms minimize f locally by solving $\nabla f = 0$ using a Newton or a secant-type iterative method [7, 8]. 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, namely, 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 make up 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 compute the gradient and the dense Hessian of f by propagating the first- and second-derivative objects strictly in the forward mode of automatic differentiation [10]. 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 for dense, forward-mode Hessians

To Evaluate	+'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 using overall reverse-mode propagation of adjoint values [9] 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 of 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 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. Computing the gradient in the forward mode

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. Computing the Hessian in the forward mode

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 operator for forward-mode, dense Hessians as partial derivatives

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-tuple

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

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

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

We wish 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

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

We can achieve this goal by propagating the 6-tuples 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

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

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 identities

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

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 described in [5] for an arbitrary number of independent variables and for arbitrary order mixed partial derivatives.

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 similar to the complexity of the operators for full, dense Hessians described in Section 3.1.

To see 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 2. 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 2 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 $1+n+n*(n+1)/2$, 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 propagate univariate Taylor series only 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 3 gives the computational complexity of the univariate Taylor × operator, assuming the Hessian matrix is dense.

Table 3. Maximum computational complexity of the univariate Taylor × operator

To Evaluate	+'s	×
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. Compared with the complexity of propagating forward-mode Hessians (see Table 1), univariate Taylor series save about 1/3 of the + and 1/4 of the × operations. 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 number 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 Generation of Code for Second Derivatives

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

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 optimization that we are doing now concerning zeros and ones. Specifically, in the code generated by ADIFOR

- we propagate three-term Taylor series, one series for each nonzero element in the Hessian.
- we propagate series in an overall forward mode similar to current gradients.
- For each composite assignment statement, we
  - 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 as follows:

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.

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). We 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 the 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. Because of known limitations of the current ADIFOR implementation, we made 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 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

We have identified three data structures in which the p Taylor series could 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.

We prefer alternative 1.1, for two reasons:

- it generalizes to higher derivatives, and
- the h$routine should also return values directly, as the original routine did.

6.2 In-line vs Subroutine Call

This paper illustrates the code in a conceptual way. We have identified two possible implementations:

Alternative 2.1: Generate code in line.

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

We prefer alternative 2.1, for three reasons:

- Better code optimization. In particular, the code generated to compute local second derivatives can be heavily optimized.
- Better parallelization and vectorization scope.
- No bloat in the number of subroutines, even though the code is large.

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 other simple expressions which can be folded into the code using conventional compiler constant-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 Drivers

We need at least four drivers for ADIFOR:

- Given sparsity pattern, compute Hessian and return in sparse data structure
- Compute dense Hessian
- Compute Hessian × vector
- Compute Hessian × 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 role in the ADIFOR development project.

Appendix A. Unoptimized ADIFOR-generated Code for Listing 1

```
C  Purpose:  Explore 2nd derivative code.
C            Raw Fortran to be hand translated a la ADIFOR
C            Hand ADIFORed with no optimization.
C  Author:   George Corliss, 06-NOV-1991
C  Results:
C  X :   1.0000000E+00  2.0000000E+00  3.0000000E+00  4.0000000E+00
C  F :  -4.1666668E-02

      integer xdim, fdim, pmax, i, j
      parameter (xdim = 4, fdim = 1, pmax = xdim)
      real x(xdim), f(fdim), jac_f(fdim,xdim),
     +     g$x(pmax,xdim), g$f(pmax,fdim)

      do 10 i = 1, xdim
   10    x(i) = i
      write (6, 1010) (x(i), i = 1, xdim)
 1010 format ('X       : ', 1p10e15.7)
      call fcn (x, xdim, f, fdim)
      write (6, 1020) (f(i), i = 1, fdim)
 1020 format ('F       : ', 1p10e15.7)
      call jac (x, xdim, f, fdim, jac_f)
      do 20 i = 1, fdim
         write (6, 1030) (jac_f(i,j), j = 1, xdim)
 1030    format ('Jacobian: ', 1p10e15.7)
   20 continue

      call eye (g$x, pmax)
      call g$fcn$3 (xdim, x, g$x, pmax, xdim, f, g$f, pmax, fdim)
      write (6, 1020) (f(i), i = 1, fdim)
      do 30 i = 1, fdim
         write (6, 1030) (g$f(j,i), j = 1, xdim)
   30 continue
      do 40 i = 1, fdim
         write (6, 1040) ((jac_f(i,j) - g$f(j,i)), j = 1, xdim)
 1040    format ('Jac err : ', 1p10e15.7)
   40 continue

      stop
      end


      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


      subroutine jac (x, xdim, f, fdim, jac_f)
      integer xdim, fdim
      real x(xdim), f(fdim), jac_f(fdim,xdim)

      f(1) = -x(1) / (x(2) * x(3) * x(4))
      jac_f(1,1) = -1.0 / (x(2) * x(3) * x(4))
      jac_f(1,2) = x(1) / (x(2) * x(2) * x(3) * x(4))
      jac_f(1,3) = x(1) / (x(3) * x(2) * x(3) * x(4))
      jac_f(1,4) = x(1) / (x(4) * x(2) * x(3) * x(4))

      return
      end
```

```
      subroutine g$fcn$3 (g$p$, x, g$x, ldg$x, xdim, f, g$f, ldg$f, fdim)
      integer xdim, fdim
      real x(xdim), f(fdim)
      integer g$p$, ldg$x, ldg$f
      real g$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 = 0.0
      r$1bar = 0.0
      r$2bar = 0.0
      r$3bar = 0.0
      r$4bar = 0.0
      r$5bar = 0.0
      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


      subroutine eye (x, xdim)
      integer xdim, i
      real x(xdim, xdim)
      do 10 i = 1, xdim
         do 10 j = 1, xdim
            x(i,j) = 0.0
   20    continue
         x(i,i) = 1.0
   10 continue
      return
      end


      subroutine saxpy4 (pmax, LenVec,
     +                   Weigh1, Vectr1, Weigh2, Vectr2,
     +                   Weigh3, Vectr3, Weigh4, Vectr4,
     +                   Result)
```

```
c  Purpose:  SAXPY of 4 vectors.
c  Input parameters:
c     pmax
c     LenVec   Length of all vectors
c     Weighn   Scalar weights
c     Vectrn   Vectors
c  Output parameter
c     Result(i)  = sum Weigh_n * Vectr_n(i)
c                   n
c  Author:  George Corliss, 06-NOV-1991
c  Assumptions:
c     All vectors have the same logical lengths
c        (Although allocated lengths may vary)
c     1 <= LenVec <= allocated sizes of vectors

      integer pmax, LenVec, i
      real Result(*), Vectr1(*), Vectr2(*), Vectr3(*), Vectr4(*)

      do 10 i = 1, LenVec
   10    Result(i) = Weigh1 * Vectr1(i) + Weigh2 * Vectr2(i)
     +              + Weigh3 * Vectr3(i) + Weigh4 * Vectr4(i)
      return
      end
```

Appendix B. Main Program for Computing Dense Hessians as Partial Derivatives

```
c  Purpose:  Illustrate code that ADIFOR might generate for
c              forward mode gradients and Hessians.
c           Not intended as a rigorous test.
c  Author:   George Corliss, 05-FEB-1992
c  Discussion:
c    This simulates program fragments from an ADIFOR-generated
c    subroutine.
c       u, v   active, not necessarily independent
c       g$x    gradient object
c       h$x    Hessian object
c    Operators for full, dense, forward mode for unary and binary
c    operations.  We assume that the target of the assignment
c    DOES NOT ALSO APPEAR ON THE RIGHT HAND SIDE.
c  Results:
c    For u    :  value:   3.400000E+00
c    Gradient :    1.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For v    :  value:  -2.100000E+00
c    Gradient :    0.000000E+00   1.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Addit:  value:   1.300000E+00
c    Gradient :    1.000000E+00   1.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Multi:  value:  -7.140000E+00
c    Gradient :   -2.100000E+00   3.400000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   1.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Divid:  value:  -1.619048E+00
c    Gradient :   -4.761905E-01  -7.709752E-01   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2  -2.267574E-01  -7.342621E-01
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Exp  :  value:   2.996410E+01
c    Gradient :    2.996410E+01   0.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   2.996410E+01
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c
c=============================================================

      integer p, pmax, g$j$, g$i$, i, j
      parameter (pmax = 4)
```

```
      real u, g$u(pmax), h$u(pmax,pmax),
     +     v, g$v(pmax), h$v(pmax,pmax),
     +     r$0, g$r$0(pmax), h$r$0(pmax,pmax),
     +     r$1, g$r$1(pmax), h$r$1(pmax,pmax),
     +     r$2, g$r$2(pmax), h$r$2(pmax,pmax),
     +     r$3, g$r$3(pmax), h$r$3(pmax,pmax)


c=============================================================
c
c  INITIALIZATION:
c
      p = pmax
      u =  3.4
      v = -2.1
      do j = 1, p
         g$u(j) = 0.0
         g$v(j) = 0.0
         do i = 1, j
            h$u(j,i) = 0.0
            h$v(j,i) = 0.0
         end do
      end do
      g$u(1) = 1.0
      g$v(2) = 1.0
      call prthes ('u    ', p, pmax, u, g$u, h$u)
      call prthes ('v    ', p, pmax, v, g$v, h$v)


c=============================================================
c
c  ADDITION: f = u + v
c
c    df     du     dv
c    --  =  --  +  --
c    dx     dx     dx
c
c      2        2        2
c     d f      d u      d v
c    -----  =  -----  +  -----
c    dx dy     dx dy     dx dy
c
      r$0 = u + v
      do g$j$ = 1, p
         g$r$0(g$j$) = g$u(g$j$) + g$v(g$j$)
         do g$i$ = 1, g$j$
            h$r$0(g$j$,g$i$) = h$u(g$j$,g$i$) + h$v(g$j$,g$i$)
         end do
      end do


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


c=============================================================
c
c  DIVISION: f = u / v, v * f = u
c
c        df     dv          du
c    v * --  +  -- * f  =  --
c        dx     dx          dx
c       ====
c
c          2                                    2             2
c         d f      dv   df    dv   df       d v           d u
c    v * -----  +  -- * --  + -- * --  +  ----- * f  =  -----
c        dx dy     dy   dx    dx   dy      dx dy         dx dy
c       =======
c
      r$2 = u / v
      do g$j$ = 1, p
         g$r$2(g$j$) =  (g$u(g$j$) - g$v(g$j$) * r$2) / v
       end do
c                  Two separate loops are necessary because
c                  the Hessian requires the gradient of f.
      do g$j$ = 1, p
         do g$i$ = 1, g$j$
            h$r$2(g$j$,g$i$) = (h$u(g$j$,g$i$)
     +                            - g$v(g$i$) * g$r$2(g$j$)
     +                         .  - g$v(g$j$) * g$r$2(g$i$)
     +                            - h$v(g$j$,g$i$) * r$2)
     +                        / v
         end do
       end do


c=============================================================
c
c  EXPONENTIAL: f = exp (u),
c
c    df         du
c    --  =  f * --
c    dx         dx
c
c      2            2
c     d f          d u       df   du
c    -----  =  f * -----  +  -- * --
c    dx dy        dx dy     dy   dx
c
      r$3 = exp (u)
      do g$j$ = 1, p
         g$r$3(g$j$) =  r$3 * g$u(g$j$)
         do g$i$ = 1, g$j$
            h$r$3(g$j$,g$i$) = r$3 * h$u(g$j$,g$i$)
     +                         + g$r$3(g$i$) * g$u(g$j$)
         end do
```

```
      end do

c=============================================================
c
c  RESULTS:
c
      call PrtHes ('Addit', p, pmax, r$0, g$r$0, h$r$0)
      call PrtHes ('Multi', p, pmax, r$1, g$r$1, h$r$1)
      call PrtHes ('Divid', p, pmax, r$2, g$r$2, h$r$2)
      call PrtHes ('Exp  ', p, pmax, r$3, g$r$3, h$r$3)

      stop
      end


      subroutine PrtHes (name, length, ldg_x, x, grad_x, Hess_x)
      character*6 name
      integer length, ldg_x, i, j
      real x, grad_x(ldg_x), Hess_x(ldg_x,ldg_x)

      write (6, 1010) name, x, (grad_x(i), i = 1, length)
      write (6, 1020)
      do j = 1, length
         write (6, 1030) j, (Hess_x(j,i), i = 1, j)
      end do
 1010 format (/ 'For ', A6, ':  value:', 1PE15.6,
     +        / 'Gradient : ', 1P5E15.6, / 100(10X, 1P5E15.6, /))
 1020 format ('Hessian  :')
 1030 format (I11, 1P5E15.6)

      return
      end
```

Appendix C. Main Program for Computing Dense Hessians as Univariate Taylor Series

```
c  Purpose:  Illustrate code that ADIFOR might generate for
c              univariate Taylor series to generate
c              gradients and Hessians
c            Not intended as a rigorous test.
c  Author:   George Corliss, 12-FEB-1992
c  Modifications:
c    12-FEB-1992 George Corliss
c                Adapted from hes_oprs.f
c  Discussion:
c    This simulates program fragments from an ADIFOR-generated
c    subroutine.
c       u, v   active, not necessarily independent
c       h$x    Taylor series object.  Ordering:
c                1
c                2   3
c                4   5   6
c                7   8   9  10
c    We compute Taylor coefficients u^(i)/i!.
c    Function values are computed redundantly.  For sequential
c      computation, that is easier to read.  For parallel
c      computation, each processor computes its own copy.
c    Operators for univariate Taylor series for unary and binary
c      operations.  We assume that the target of the assignment
c      DOES NOT ALSO APPEAR ON THE RIGHT HAND SIDE.
c  Results:
c    For u    :  value:   3.400000E+00
c    Gradient :    1.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For v    :  value:  -2.100000E+00
c    Gradient :    0.000000E+00   1.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Addit:  value:   1.300000E+00
c    Gradient :    1.000000E+00   1.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Multi:  value:  -7.140000E+00
c    Gradient :   -2.100000E+00   3.400000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2   1.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Divid:  value:  -1.619048E+00
c    Gradient :   -4.761905E-01  -7.709752E-01   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   0.000000E+00
c              2  -2.267573E-01  -7.342621E-01
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c    For Exp  :  value:   2.996410E+01
```

```
c    Gradient :    2.996410E+01   0.000000E+00   0.000000E+00   0.000000E+00
c    Hessian  :
c              1   2.996410E+01
c              2   0.000000E+00   0.000000E+00
c              3   0.000000E+00   0.000000E+00   0.000000E+00
c              4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
c
c=============================================================

c         NumInd  Number of independent variables (= p)
c         NumSer  Number of series
c                   for second derivatives, NumSer = p(p+1)/2
c         SerOrd  Series Order
      integer NumInd, NumSer, SerOrd, j
      parameter (NumInd = 4, NumSer = 10, SerOrd = 2)
      real u, h$u(NumSer, 0:SerOrd), v, h$v(NumSer, 0:SerOrd),
     +     r$0, h$r$0(NumSer, 0:SerOrd), r$1, h$r$1(NumSer, 0:SerOrd),
     +     r$2, h$r$2(NumSer, 0:SerOrd), r$3, h$r$3(NumSer, 0:SerOrd)

c  Auxiliaries for comparison:
      integer pmax
      parameter (pmax = 4)
      real g$u(pmax), Hes$u(pmax,pmax),
     +     g$v(pmax), Hes$v(pmax,pmax),
     +     g$r$0(pmax), Hes$r$0(pmax,pmax),
     +     g$r$1(pmax), Hes$r$1(pmax,pmax),
     +     g$r$2(pmax), Hes$r$2(pmax,pmax),
     +     g$r$3(pmax), Hes$r$3(pmax,pmax)

c=============================================================
c
c  INITIALIZATION:
c
      u =  3.4
      v = -2.1
      do j = 1, NumSer
c                              Function values:
         h$u(j,0) = u
         h$v(j,0) = v
c                              First derivatives:
         h$u(j,1) = 0.0
         h$v(j,1) = 0.0
c                              Second derivatives:
         h$u(j,2) = 0.0
         h$v(j,2) = 0.0
      end do
      h$u(1,1) = 1.0
      h$u(2,1) = 1.0
      h$u(4,1) = 1.0
      h$u(7,1) = 1.0
      h$v(2,1) = 1.0
      h$v(3,1) = 1.0
      h$v(5,1) = 1.0
      h$v(8,1) = 1.0

      call Ser2He (NumSer, SerOrd, h$u, u, g$u, Hes$u, pmax)
      call Ser2He (NumSer, SerOrd, h$v, v, g$v, Hes$v, pmax)

      call PrtSer ('u    ', NumSer, SerOrd, h$u)
      call PrtHes ('u    ', pmax, pmax, u, g$u, Hes$u)
      call PrtSer ('v    ', NumSer, SerOrd, h$v)
      call PrtHes ('v    ', pmax, pmax, v, g$v, Hes$v)
```

```
c=============================================================
c
c  ADDITION: f = u + v
c
c     df     du     dv
c     --  =  --  +  --
c     dx     dx     dx
c
c      2      2      2
c     d f    d u    d v
c     ---  = ---  + ---
c       2      2      2
c     dx     dx     dx
c
      r$0 = u + v
      do g$j$ = 1, NumSer
         h$r$0(g$j$,0) = h$u(g$j$,0) + h$v(g$j$,0)
c  or                 = r$0
         h$r$0(g$j$,1) = h$u(g$j$,1) + h$v(g$j$,1)
         h$r$0(g$j$,2) = h$u(g$j$,2) + h$v(g$j$,2)
      end do


c=============================================================
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.
c
      r$1 = u * v
      do g$j$ = 1, NumSer
         h$r$1(g$j$,0) = h$u(g$j$,0) * h$v(g$j$,0)
c  or                 = r$1
         h$r$1(g$j$,1) = u * h$v(g$j$,1) + h$u(g$j$,1) * v
         h$r$1(g$j$,2) = u * h$v(g$j$,2) + h$u(g$j$,1) * h$v(g$j$,1)
     +                   + h$u(g$j$,2) * v
      end do


c=============================================================
c
c  DIVISION: f = u / v, v * f = u
c
c          df     dv          du
c     v * --  +  -- * f  =  --
c          dx     dx          dx
c         ====
c
c          2                          2          2
c         d f          dv    df      d v        d u
c     v * ----  +  2 * -- * --  +  --- * f  =  ---
c            2                        2          2
c         dx           dx    dx      dx         dx
c         ====
```

```
c
c    We divide both sides by 2.
c
      r$2 = u / v
      do g$j$ = 1, NumSer
         h$r$2(g$j$,0) = h$u(g$j$,0) / h$v(g$j$,0)
c  or                 = r$2
         h$r$2(g$j$,1) = (h$u(g$j$,1) - h$v(g$j$,1) * r$2) / v
         h$r$2(g$j$,2) = (h$u(g$j$,2) - h$v(g$j$,1) * h$r$2(g$j$,1)
     +                   - h$v(g$j$,2) * r$2) / v
c  Better parallelism from:
c         h$r$2(g$j$,2) = (h$u(g$j$,2)
c     +                    - h$v(g$j$,1) * (h$u(g$j$,1)
c     +                                    - h$v(g$j$,1) * r$2) / v
c     +                    - h$v(g$j$,2) * r$2) / v

      end do


c=============================================================
c
c  EXPONENTIAL: f = exp (u),
c
c    df         du
c    --  =  f * --
c    dx         dx
c
c     2           2
c    d f         d u      df   du
c    ---  =  f * ---  +   -- * --
c      2           2
c    dx          dx       dx   dx
c
c    We divide both sides by 2.
c
      r$3 = exp (u)
      do g$j$ = 1, NumSer
         h$r$3(g$j$,0) = exp (h$u(g$j$,0))
c  or                 = r$3
         h$r$3(g$j$,1) = r$3 * h$u(g$j$,1)
         h$r$3(g$j$,2) = r$3 * h$u(g$j$,2)
     +                   + h$r$3(g$j$,1) * h$u(g$j$,1) / 2.0
c  Better parallelism from:
c     +                    + r$3 * h$u(g$j$,1)
c     +                      * h$u(g$j$,1) / 2.0
      end do


c=============================================================
c
c  RESULTS:
c
      call Ser2He (NumSer, SerOrd, h$r$0, r$0, g$r$0, Hes$r$0, pmax)
      call Ser2He (NumSer, SerOrd, h$r$1, r$1, g$r$1, Hes$r$1, pmax)
      call Ser2He (NumSer, SerOrd, h$r$2, r$2, g$r$2, Hes$r$2, pmax)
      call Ser2He (NumSer, SerOrd, h$r$3, r$3, g$r$3, Hes$r$3, pmax)

      call PrtSer ('r$0  ', NumSer, SerOrd, h$r$0)
      call PrtHes ('Addit', pmax, pmax, r$0, g$r$0, Hes$r$0)
      call PrtSer ('r$1  ', NumSer, SerOrd, h$r$1)
      call PrtHes ('Multi', pmax, pmax, r$1, g$r$1, Hes$r$1)
      call PrtSer ('r$2  ', NumSer, SerOrd, h$r$2)
      call PrtHes ('Divid', pmax, pmax, r$2, g$r$2, Hes$r$2)
      call PrtSer ('r$3  ', NumSer, SerOrd, h$r$3)
```

```
      call PrtHes ('Exp  ', pmax, pmax, r$3, g$r$3, Hes$r$3)

      stop
      end


      subroutine Ser2He (NumSer, SerOrd, Ser_V,
     +                   Value, Grad_V, Hess_V, pmax)

c  Purpose:  Convert univariate Taylor series form
c            to value, gradient, and Hessian form.

      integer NumSer, SerOrd, pmax, i, j, index(10,10)
      real Ser_V(NumSer,0:SerOrd),
     +     Value, Grad_V(pmax), Hess_V(pmax,pmax)

c  Index a lower triangular matrix in a sequential order:
      if (pmax .gt. 10) then
         write (*, *) 'Error in Ser2He.'
         STOP
      end if
      do j = 1, pmax
         do i = 1, j
            index(j,i) = i + j * (j-1) / 2
         end do
      end do

      Value = Ser_V(1,0)
      do j = 1, pmax
         Grad_V(j) = Ser_V(j*(j+1)/2,1)
         Hess_V(j,j) = 2.0 * Ser_V(index(j,j),2)
         do i = 1, j-1
            Hess_V(j,i) = Ser_V(index(j,i),2) - Ser_V(index(j,j),2)
     +                    - Ser_V(index(i,i),2)
         end do
      end do

      return
      end


      subroutine PrtSer (name, ldg_x, Len_x, Ser_x)
      character*6 name
      integer ldg_x, Len_x, i, j
      real Ser_x(ldg_x,*)

      write (6, 1010) name
      do j = 1, ldg_x
         write (6, 1020) j, (Ser_x(j,i+1), i = 0, Len_x)
      end do
 1010 format (/ 'Series for ', A6, ':')
 1020 format (I11, 1P5E15.6, 100(14X, 1P5E15.6/))

      return
      end


      subroutine PrtHes (name, length, ldg_x, x, grad_x, Hess_x)
      character*6 name
      integer length, ldg_x, i, j
      real x, grad_x(ldg_x), Hess_x(ldg_x,ldg_x)

      write (6, 1010) name, x, (grad_x(i), i = 1, length)
      write (6, 1020)
```

```
      do j = 1, length
         write (6, 1030) j, (Hess_x(j,i), i = 1, j)
      end do
1010  format (/ 'For ', A6, ':  value:', 1PE15.6,
     +        / 'Gradient : ', 1P5E15.6, / 100(10X, 1P5E15.6, /))
1020  format ('Hessian  :')
1030  format (I11, 1P5E15.6)

      return
      end
```

Appendix D. Driver Program to Call Undifferentiated Code

```
C  Purpose:  Explore 2nd derivative code.
C            Driver program to call undifferentiated code.
C  Author:   George Corliss, 25-FEB-1992
C  Reference:   Simple example from Working Note 1, Section 2.
C  Results:
C       Undifferentiated code.
C          F =   -0.12500

      program driver

      integer xdim
      parameter (xdim = 4)
      real x(xdim), f

      do i = 1, xdim
         x(i) = i
      end do
      call examp2 (x, xdim, f)
      write (6, 1010) f
 1010 format ('Undifferentiated code.' / '   F = ', f10.5)

      stop
      end


      subroutine examp2 (x, xdim, f)
      integer xdim
      real x(xdim)
      real 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
```

Appendix E. Driver Program to Call ADIFOR-generated First Derivative Code

```
C  Purpose:  Explore 2nd derivative code.
C            Driver program to call ADIFOR-generated first derivative code.
C  Author:   George Corliss, 25-FEB-1992
C  Reference:   Simple example from Working Note 1, Section 2.
C  Results:
C       ADIFOR-generated code.
C          F      =   -0.12500
C          grad F =  -1.250000E-01  1.875000E-01  0.000000E+00  0.000000E+00

      program driver

c Declare:
      integer xdim
      parameter (xdim = 4)
      real x(xdim), f
      real g$x(xdim,xdim), g$f(xdim)

c Initialize:
      do i = 1, xdim
         x(i) = i
      end do
      call eye (g$x, xdim)

c Compute derivatives:
      call g$examp2$5 (xdim, x, g$x, xdim, xdim, f, g$f, xdim)

c Report:
      write (6, 1010) f, (g$f(i), i = 1, xdim)
 1010 format ('ADIFOR-generated code.' /
     +        '   F      = ', f10.5 /
     +        '   grad F = ', 1p8e14.6)

      stop
      end


      subroutine g$examp2$5(g$p$, x, g$x, ldg$x, xdim, f, g$f, ldg$f)
C
C       Formal f is active.
C       Formal x is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 4)
        integer g$i$
        real r$2bar
        real r$1bar
        real zbar
        real r$3
        real r$2
        real r$1
        integer ldg$f
C
        real f
        real g$f(ldg$f)
        integer xdim
        real x(xdim)
        real g$x(ldg$x, xdim)
        integer ldg$x
        real y, z, w
        real g$y(g$pmax$), g$z(g$pmax$), g$w(g$pmax$)
```

```
C       y and z depend in some way on x(1..xdim)
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
C       y = x(1)
        do 99995 g$i$ = 1, g$p$
          g$y(g$i$) = g$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)
99994   continue
        z = x(2)
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$1bar * z
        do 99993 g$i$ = 1, g$p$
          g$w(g$i$) = -(1.0d0 / r$2) * g$y(g$i$) + (zbar * g$z(g$i$))
99993   continue
        w = r$3
C       f depends in some way on w
        f = w
        do 99992 g$i$ = 1, g$p$
          g$f(g$i$) = g$w(g$i$)
99992   continue
        return
      end
```

Appendix F. Driver Program to Code Extracted for Single Assignment

```
C  Purpose:  Explore 2nd derivative code.
C            Driver program to code extracted for single assignment.
C  Author:   George Corliss, 25-FEB-1992
C  Results:
C       Code extracted for single assignment.
C          W      =   -0.12500
C          grad W =  -1.250000E-01  1.875000E-01  0.000000E+00  0.000000E+00

      program driver

c Declare:
      integer xdim
      parameter (xdim = 4)
      real y, g$y(xdim), z, g$z(xdim), w, g$w(xdim)

c Initialize:
      y = 1.0
      z = 2.0
      g$y(1) = 1.0
      g$z(2) = 1.0

c Compute derivatives:
      call examp2G (2, y, g$y, z, g$z, w, g$w)

c Report:
      write (6, 1010) w, (g$w(i), i = 1, xdim)
 1010 format ('Code extracted for single assignment.' /
     +        '   W      = ', f10.5 /
     +        '   grad W = ', 1p8e14.6)

      stop
      end


      subroutine examp2G (gQpQ, y, gQy, z, gQz, w, gQw)

C  Purpose:  Explore 2nd derivative code.
C            Extract ADIFOR-generated code for one assignment
C            statement from examp2.5.f
C  Author:   George Corliss, 25-FEB-1992
C  Modifications:
C    1.0d0  --> 1.0
C    $ --> Q
C    Remove redundant parentheses
C    Combine ADIFOR-generated declarations
C    bar --> B

        integer gQpQ, gQpmaxQ, gQiQ
        parameter (gQpmaxQ = 4)
        real rQ2B, rQ1B, zB, rQ3, rQ2, rQ1
        real y, z, w, gQy(gQpmaxQ), gQz(gQpmaxQ), gQw(gQpmaxQ)

C       Consider the assignment statement
C       w = -y / (z * z * z)
        rQ1 = z * z
        rQ2 = rQ1 * z
        rQ3 = -y / rQ2
        rQ2B = -rQ3 / rQ2
        rQ1B = rQ2B * z
        zB = rQ2B * rQ1
        zB = zB + rQ1B * z
```

```
        zB = zB + rQ1B * z
        do 99993 gQiQ = 1, gQpQ
          gQw(gQiQ) = -1.0 / rQ2 * gQy(gQiQ) + zB * gQz(gQiQ)
99993   continue
        w = rQ3

        return
      end
```

Appendix G. Driver Program for Hessian by ADIFOR (ADIFOR (w = -y / (z*z*z)))

```
C  Purpose:  Explore 2nd derivative code.
C            Driver program Hessian by Adifor (Adifor (examp2.f))
C  Author:   George Corliss, 25-FEB-1992
c Description:
c    We want to compute the Hessian of W with respect to y and z.
c    Subroutine examp2g.74 computes g$w as the gradient of W with
c    respect to y and z.  Hence, we want to differentiate g$w with
c    respect to y and z.
C  Results:
C        Hessian by Adifor (Adifor (examp2.f)).
C          W      =   -0.12500
C          grad W =  -1.250000E-01  1.875000E-01
C          Hessian W =
C           1      0.000000E+00  1.875000E-01
C           2      1.875000E-01 -3.750000E-01

      program driver

c Declare:
      integer xdim
      parameter (xdim = 2)
      real y, gQy(4), z, gQz(4), w, gQw(4)
      real g$y(xdim), g$z(xdim), g$gQw(xdim,4)

c Initialize:
      y = 1.0
      z = 2.0
      gQy(1) = 1.0
      gQz(2) = 1.0
      g$y(1) = 1.0
      g$z(2) = 1.0

c Compute derivatives:
      call g$examp2g$74 (xdim, xdim, y, g$y, xdim, gqy, z, g$z, xdim,
     *     gqz, w, gqw, g$gqw, xdim)

c Report:
      write (6, 1010) w, (gQw(i), i = 1, xdim)
 1010 format ('Hessian by Adifor (Adifor (examp2.f)).' /
     +        '   W      = ', f10.5 /
     +        '   grad W = ', 1p8e14.6)
      write (6, 1015)
 1015 format ('   Hessian W =')
      do j = 1, xdim
         write (6, 1020) j, (g$gqw(j,i), i = 1, xdim)
 1020    format (i5, 4x, 1p8e14.6)
      end do

      stop
      end


      subroutine g$examp2g$74(g$p$, gqpq, y, g$y, ldg$y, gqy, z, g$z, ld
     *g$z, gqz, w, gqw, g$gqw, ldg$gqw)
C
C       Formal gqw is active.
C       Formal z is active.
C       Formal y is active.
C
        integer g$p$
        integer g$pmax$
```

```
      parameter (g$pmax$ = 2)
      integer g$i$
      real r$1
      real r$0
      integer ldg$y
      integer ldg$z
C
C     Purpose:  Explore 2nd derivative code.
C     Extract ADIFOR-generated code for one assignment
C     statement from examp2.5.f
C     Author:   George Corliss, 25-FEB-1992
C     Modifications:
C     1.0d0  --> 1.0
C     $ --> Q
C     Remove redundant parentheses
C     Combine ADIFOR-generated declarations
C     bar --> B
      integer gqpq, gqpmaxq, gqiq
      parameter (gqpmaxq = 4)
      real rq2b, rq1b, zb, rq3, rq2, rq1
      real g$rq2b(g$pmax$), g$rq1b(g$pmax$), g$zb(g$pmax$), g$rq3(g$pm
     *ax$), g$rq2(g$pmax$), g$rq1(g$pmax$)
      real y, z, w, gqy(gqpmaxq), gqz(gqpmaxq), gqw(gqpmaxq)
      real g$y(ldg$y), g$z(ldg$z), g$gqw(ldg$gqw, gqpmaxq)
      integer ldg$gqw
C     Consider the assignment statement
C     w = -y / (z * z * z)
      if (g$p$ .gt. g$pmax$) then
        print *, 'Parameter g$p is greater than g$pmax.'
        stop
      endif
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
```

Appendix H. Driver Program to Call ADIFOR-like Hessian Code

```
C  Purpose:  Explore 2nd derivative code.
C            Driver program to call ADIFOR-like Hessian code
C  Author:   George Corliss, 26-FEB-1992
C  Reference:   Simple example from Working Note 1, Section 2.
C  Discussion:
C     Compute the DENSE Hessian by Taylor Series
C  Results:
C     Series for F     :
C                1  -1.250000E-01  -1.250000E-01   0.000000E+00
C                2  -1.250000E-01   6.250000E-02   0.000000E+00
C                3  -1.250000E-01   1.875000E-01  -3.750000E-01
C                4  -1.250000E-01  -1.250000E-01   0.000000E+00
C                5  -1.250000E-01   1.875000E-01  -3.750000E-01
C                6  -1.250000E-01   0.000000E+00   0.000000E+00
C                7  -1.250000E-01  -1.250000E-01   0.000000E+00
C                8  -1.250000E-01   1.875000E-01  -3.750000E-01
C                9  -1.250000E-01   0.000000E+00   0.000000E+00
C               10  -1.250000E-01   0.000000E+00   0.000000E+00
C     For F      :  value:  -1.250000E-01
C     Gradient :    -1.250000E-01   1.875000E-01   0.000000E+00   0.000000E+00
C     Hessian  :
C                1   0.000000E+00
C                2   1.875000E-01  -3.750000E-01
C                3   0.000000E+00   0.000000E+00   0.000000E+00
C                4   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00

      program driver

c Declare:
      integer xdim, order, ser_p, P
      parameter (xdim = 4, order = 2, ser_p = 10)
      real x(xdim), f
      real hx(ser_p,order,xdim), hf(ser_p,order)
      real temp, g$temp(xdim), h$temp(xdim,xdim)

c Initialize:
      do i = 1, xdim
         x(i) = i
      end do
      call SerEye (ser_p, order, xdim, hx)
      p = ser_p

c Compute derivatives:
      call g$examp2$5 (p, x, hx, ser_p, xdim, f, hf, ser_p)

c Report:
      write (6, 1010)
 1010 format ('Hessian by hand-coded second derivatives.')
      call PrtSer ('F      ', ser_p, order, f, hf)
      call Ser2He (ser_p, order, hf, temp, g$temp, h$temp, xdim)
      call PrtHes ('F      ', xdim, xdim, temp, g$temp, h$temp)
      stop
      end
```

Appendix I. Library Utility Routines

```
c  File:  SEREYE.f                          26-FEB-1992

      subroutine SerEye (NumSer, SerOrd, V_Size, Ser_V)

c  Purpose: Initialize univariate Taylor series to compute
c           a dense Hessian stored in the order
c                1
c                2   3
c                4   5   6
c  Assumptions:
c    V_Size <= 10
c    NumSer = V_Size * (V_Size + 1) / 2
c    SerOrd = 3

      integer NumSer, SerOrd, V_Size, i, j, index(10,10)
      real Ser_V(NumSer,SerOrd,V_Size)

c  Index a lower triangular matrix in a sequential order:
      if (V_Size .gt. 10) then
         write (*, *) 'Error in SerEye.'
         STOP
      end if
      do j = 1, V_Size
         do i = 1, j
            index(j,i) = i + j * (j-1) / 2
         end do
      end do

      do k = 1, V_Size
         do i = 1, NumSer
c Second derivative:
            Ser_V(i,2,k) = 0.0
         end do
c First derivative:
         do j = 1, V_Size
            do i = 1, j
               if ((i .eq. k) .or. (j .eq. k))
     +            Ser_V(index(j,i),1,k) = 1.0
            end do
         end do
      end do

      return
      end


c  File:  PRTSER.f                          12-FEB-1992

      subroutine PrtSer (name, ldg_x, Len_x, x, Ser_x)
      character*6 name
      integer ldg_x, Len_x, i, j
      real x, Ser_x(ldg_x,*)

      write (6, 1010) name
      do j = 1, ldg_x
         write (6, 1020) j, x, (Ser_x(j,i+1), i = 1, Len_x)
      end do
 1010 format (/ 'Series for ', A6, ':')
 1020 format (I11, 1P5E15.6, 100(14X, 1P5E15.6/))

      return
```

```
      end

C File:  PRTHES.f                          06-FEB-1992

      subroutine prthes (name, length, ldg_x, x, grad_x, Hess_x)
      character*6 name
      integer length, ldg_x, i, j
      real x, grad_x(ldg_x), Hess_x(ldg_x,ldg_x)

      write (6, 1010) name, x, (grad_x(i), i = 1, length)
      write (6, 1020)
      do j = 1, length
         write (6, 1030) j, (Hess_x(j,i), i = 1, j)
      end do
 1010 format (/ 'For ', A6, ':  value:', 1PE15.6,
     +        / 'Gradient : ', 1P5E15.6, / 100(10X, 1P5E15.6, /))
 1020 format ('Hessian  :')
 1030 format (I11, 1P5E15.6)

      return
      end


C File:  SER2HE.f                          11-MAR-1992

      subroutine Ser2He (NumSer, SerOrd, Ser_V, Grad_V, Hess_V, pmax)

c  Purpose:  Convert univariate Taylor series form
c            to value, gradient, and Hessian form.

      integer NumSer, SerOrd, pmax, i, j, index(10,10)
      real Ser_V(NumSer,SerOrd), Grad_V(pmax), Hess_V(pmax,pmax)

c  Index a lower triangular matrix in a sequential order:
      if (pmax .gt. 10) then
         write (*, *) 'Error in Ser2He.'
         STOP
      end if
      do j = 1, pmax
         do i = 1, j
            index(j,i) = i + j * (j-1) / 2
         end do
      end do

      do j = 1, pmax
         Grad_V(j)   = Ser_V(j*(j+1)/2,1)
         Hess_V(j,j) = Ser_V(index(j,j),2)
         do i = 1, j-1
            Hess_V(j,i) = (Ser_V(index(j,i),2) - Ser_V(index(j,j),2)
     +                   - Ser_V(index(i,i),2)) * 0.5
         end do
      end do

      return
      end
```

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., September 1991, and as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Tex., 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., March 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 Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, N.J., 1983.

[8] John E. Dennis and Robert B. Schnabel. A view of unconstrained optimization. In G. L. Nemhauser, editor, *Handbooks in Operations Research and Mathematical Software*, volume 1, pages 1–72. Elsevier, 1989.

[9] 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.

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