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

Christian Bischof
George Corliss
Andreas Griewank

CRPC-TR92238
May 1992
ADIFOR Working Note #8:
Hybrid Evaluation of Second Derivatives in ADIFOR

by

Christian Bischof, George Corliss, and Andreas Griewank

May 1992
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 \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), 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 $\lambda_i$ 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 $\nabla u$, $\nabla v$, $\nabla^2 u$, 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

$$f = uv$$
$$\nabla f = u \cdot \nabla v + v \cdot \nabla u$$
$$\nabla^2 f = u \cdot \nabla^2 v + \nabla u \cdot (\nabla v)^T + \nabla v \cdot (\nabla u)^T + v \cdot \nabla^2 u.$$

Table 1 gives the computational complexity for the $\times$ operator.
Table 1. Computational complexity of the $\times$ operator for dense, forward-mode Hessians

<table>
<thead>
<tr>
<th>To Evaluate</th>
<th>$+$'s</th>
<th>$\times$'s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Gradient</td>
<td>$n$</td>
<td>$2n$</td>
</tr>
<tr>
<td>Hessian</td>
<td>$1.5n(n+1)$</td>
<td>$2n(n+1)$</td>
</tr>
</tbody>
</table>

The complexity of the other operators is similar, differing only in the constants. The storage complexity for the naive forward propagation of $\nabla f$ 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 $f : x \in \mathbb{R}^n \rightarrow f \in \mathbb{R}$ contains an active variable $u$. For the present discussion, we assume that $p = p_{\text{max}} = n$. Then the ADIFOR-generated variables g\$u and h\$u in h\$fcn contain

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

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

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.

```fortran
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
cend
```

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.
Listing 2. Computing the gradient in the forward mode

The gradient object g$x is initialized to an n × 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.

```fortran
subroutine g$fcn3 (g$p, x, g$x, xdim, f, g$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
E
C Initialize adjoints
r$0bar = r$1bar = r$2bar = r$3bar = r$4bar = r$5bar = r$6bar = r$7bar = 0.0
r$7bar = 1.0
C
r$4bar = r$4bar + r$7bar * (-1.0 / r$6)
r$6bar = r$6bar + r$7bar * (-r$7 / r$6)
C
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$4bar * (-1.0)
C
call saxpy4 (pmax, g$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))
C
f(1) = r$7
C
return
end
```

```fortran
subroutine g$fcn3 (g$p, x, g$x, ldg$x, xdim, f, g$f, ldg$f, fdim)
  integer xdim, 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$4, g$sr$4(pmax), h$sr$4(pmax, pmax), r$5, g$sr$5(pmax), h$sr$5(pmax, pmax),
          + r$6, g$sr$6(pmax), h$sr$6(pmax, pmax), r$7, g$sr$7(pmax), h$sr$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
C f(1) = -x(1) / (x(2) * x(3) * x(4))
C
  r$4 = -x(1)
  do 99990 g$j = 1, g$p
    g$sr$4(g$j) = -g$x(g$j,1)
    do 99990 g$i = g$j, g$p
      h$sr$4(g$j, g$i) = -h$x(g$j, g$i,1)
    99990 continue
    r$5 = x(2) * x(3)
    do 99980 g$j = 1, g$p
      g$sr$5(g$j) = x(2) * g$x(g$j,3) + g$x(g$j,2) * x(3)
    99980 continue
    do 99982 g$j = 1, g$p
      g$sr$6(g$j) = g$j
    do 99982 g$i = g$j, g$p
```

4
h(x(g(j,t),g(i,s)))
  + = x(2) * h_x(g(j,t),g(i,s),3) + g_x(g(j,t),2) * g_x(g(i,s),3)
  + + g_x(g(j,t),2) * g_x(g(i,s),3) + h_x(g(j,t),g(i,s),2) * x(3)

99962 continue

r85 = r85 * x(4)
d0 99970 g(j,t) = 1, g(s)
g(s) = r85 * g_x(g(j,t),4) + g(s)*5(g(j,t)) * x(4)

99970 continue
d0 99972 g(j,t) = 1, g(s)
d0 99972 g(i,s) = g(j,t), g(s)
h(s) = r85 * h_x(g(j,t),g(i,s),4) + g(s)*5(g(i,s)) * g_x(g(j,t),4)
  + + g(s)*5(g(j,t)) * g_x(g(i,s),4) + h(s)*5(g(j,t),g(i,s)) * x(4)

99972 continue

C r87 = r84 / r86
r86 = 1.0 / r86
r87 = r84 * r86
do 99960 g(j,t) = 1, g(s)
g(s) = (g(s)*4(g(j,t)) - g(s)*6(g(j,t)) * r87) * r86

99960 continue
do 99962 g(i,s) = g(j,t), g(s)
h(s) = (h(s)*4(g(j,t),g(i,s)) - (g(s)*6(g(i,s)) * g(s)*7(g(j,t))
  + + g(s)*6(g(j,t)) * g(s)*7(g(i,s)) + h(s)*6(g(j,t),g(i,s)) * r87)) * r86

99962 continue

f(1) = r87
do 99950 g(j,t) = 1, g(s)
g(s) = g(s)*7(g(j,t))
do 99950 g(i,s) = g(j,t), g(s)
h(s) = g(j,t), g(s)
h(s) = h(s)*7(g(j,t),g(i,s))

99950 continue

return
end

Listing 3. Computing the Hessian in the forward mode

The Hessian object h_x is initialized to a n x n x 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.

r80 = u + v
r81 = u * v
r82 = u / v
r83 = 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 MULTIPICATION: f = u * v
c
  df    dv    du
  -- = u * -- + -- * v
  dx    dx    dx

c  d  
  d f = d v

c  ------ = u * ------ + -- * -- + -- * -- + ---- * v
  dx dy  dx dy  dx dy  dx dy  dx dy
  r^1 = u * v
do g^2 = 1, p
  g^2 r^1 (g^2 s^1) = u * g^2 v (g^2 s^1) + g^2 u (g^2 s^1) * v
  do g^2 s^1 = 1, g^2 s^1
    h^2 r^1 (g^2 s^1, g^2 s^1) = u * h^2 s^1 (g^2 s^1, g^2 s^1) + g^2 u (g^2 s^1) * g^2 v (g^2 s^1)
    + g^2 u (g^2 s^1) * g^2 v (g^2 s^1) + h^2 s^1 (g^2 s^1, g^2 s^1) * 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 \( v \in \mathbb{R} \). 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 \( 6 = \binom{3}{2} \), and the cost of a convolution is \( 15 = \binom{5}{2} \) 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

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

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

\[
u(s) = u + (u_x + u_y) s + (u_{xy} + 0.5 u_{xx} + 0.5 u_{yy}) s^2, \quad \text{and} \quad v(s) = v + (v_x + v_y) s + (v_{xy} + 0.5 v_{xx} + 0.5 v_{yy}) s^2
\]

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 \( \alpha \) and \( \beta \) satisfy the identities

\[
\alpha = w_x = w_x + w_y, \quad \text{and} \quad \beta = w_{xx}/2 = w_{xy} + 0.5 w_{xx} + 0.5 w_{yy}.
\]

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

\[
d\frac{df}{ds} = \frac{\partial f}{\partial x} \frac{\partial x}{\partial s} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial s}
\]

\[
= \frac{df}{dx} \frac{\partial x}{\partial s} + \frac{df}{dy} \frac{\partial y}{\partial s} = 1
\]

\[
d\frac{d^2 f}{ds^2} = \frac{\partial}{\partial s} \left[ \frac{\partial f}{\partial x} \right] + \frac{\partial}{\partial s} \left[ \frac{\partial f}{\partial y} \right]
\]

\[
= \frac{\partial^2 f}{\partial x^2} \frac{\partial x}{\partial s} \frac{\partial x}{\partial s} + \frac{\partial^2 f}{\partial x \partial y} \frac{\partial x}{\partial s} \frac{\partial y}{\partial s} + \frac{\partial^2 f}{\partial y \partial x} \frac{\partial y}{\partial s} \frac{\partial x}{\partial s} + \frac{\partial^2 f}{\partial y^2} \frac{\partial y}{\partial s} \frac{\partial y}{\partial s}
\]

\[
= \frac{\partial^2 f}{\partial x^2} + 2 \frac{\partial^2 f}{\partial x \partial y} + \frac{\partial^2 f}{\partial y^2}.
\]

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$\mathbf{f}$

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$f'$ = g$\mathbf{f}$(*)</th>
<th>$f''$ = h$\mathbf{f}$(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>At x</td>
<td>$f$</td>
<td>$\frac{\partial f}{\partial x}$</td>
<td>$\frac{\partial^2 f}{\partial x^2}$</td>
</tr>
<tr>
<td>At s</td>
<td>$f$</td>
<td>$\frac{\partial f}{\partial s}$</td>
<td>$\frac{\partial^2 f}{\partial s^2}$</td>
</tr>
<tr>
<td>At y</td>
<td>$f$</td>
<td>$\frac{\partial f}{\partial y}$</td>
<td>$\frac{\partial^2 f}{\partial y^2}$</td>
</tr>
</tbody>
</table>

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

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

(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 $f'(2) = f'(1) + f'(3)$. 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 $f$, $\nabla f$, and Hessian ($\mathbf{f}$) 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 $\ast$, $\ast$, $/$, and $\exp$ is included as Appendix C.

c MULTIPICATION: $f = u \ast v$
c
df dv du
dx dx dx
c
c 2 2
df dv du dv du
c --- = u * --- + 2 * --- + --- * v
c 2 2 2
dx dx dx dx

c We divide both sides by 2 to store the Taylor coefficient.
c
c r$\mathbf{s}$1 = u * v
do g$\mathbf{s}$j = 1, p
g$\mathbf{s}$r$\mathbf{s}$1(g$\mathbf{s}$j$\mathbf{s}$) = u * g$\mathbf{s}$v(g$\mathbf{s}$j$\mathbf{s}$) + g$\mathbf{s}$u(g$\mathbf{s}$j$\mathbf{s}$) * v
h$\mathbf{s}$r$\mathbf{s}$1(g$\mathbf{s}$j$\mathbf{s}$) = u * h$\mathbf{s}$v(g$\mathbf{s}$j$\mathbf{s}$) + 2 * g$\mathbf{s}$u(g$\mathbf{s}$j$\mathbf{s}$) * g$\mathbf{s}$v(g$\mathbf{s}$j$\mathbf{s}$) + h$\mathbf{s}$u(g$\mathbf{s}$j$\mathbf{s}$) * v
end do

Listing 6. Multiplication operator for forward Hessians as univariate series

Table 3 gives the computational complexity of the univariate Taylor $\times$ operator, assuming the Hessian matrix is dense.
Table 3. Maximum computational complexity of the univariate Taylor \times operator

<table>
<thead>
<tr>
<th>To Evaluate</th>
<th>('s)</th>
<th>(\times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Gradient</td>
<td>(n)</td>
<td>2(n)</td>
</tr>
<tr>
<td>Hessian</td>
<td>(n(n + 1))</td>
<td>1.5(n(n + 1))</td>
</tr>
</tbody>
</table>

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 \(\times\) 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 \(\nabla u\), \(\nabla v\), \(\nabla^2 u\), and \(\nabla^2 v\) are available from earlier computations. If \(w = f(u, v)\), the chain rule tells us that

\[
\nabla w = \frac{\partial w}{\partial u} \cdot \nabla u + \frac{\partial w}{\partial v} \cdot \nabla v, \quad \text{and} \\
\nabla^2 w = \frac{\partial^2 w}{\partial u^2} \cdot (\nabla u)^2 + 2 \frac{\partial^2 w}{\partial u \partial v} \cdot \nabla u \cdot \nabla v + \frac{\partial^2 w}{\partial v^2} \cdot (\nabla v)^2.
\]

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 \(u\) and \(v\), we can easily compute \(\nabla w\) 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 \(\nabla w\) 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.

9
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 \( \nabla f \) 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

\[
\begin{align*}
    w &= f(s_1, \ldots, s_k) \\
    w' &= \sum_{i=1}^{k} (\nabla f)_i \cdot s_i' \\
        &= \nabla f^T \cdot s' \\
    w'' &= \sum_{i=1}^{k} \left( (\nabla f)_i \cdot s_i'' + s_i' \cdot \sum_{j=1}^{k} [(\nabla^2 f)_{ij} \cdot s_j'] \right) \\
        &= \nabla f^T \cdot s'' + s'^T \cdot \nabla^2 f \cdot s'.
\end{align*}
\]

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 \( \nabla f \) 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 \( \nabla f \). 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 \( \nabla f \). 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:

\[
    w = f (s_1, s_2, \ldots, s_k)
\]

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:**

\[
    r^0 = s_1 \\
    \vdots \\
    r^m = \ldots \\
    w = r^m
\]

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

**Block 2:**

\[
    c \quad r^m = \ldots \\
    r^{79bar} = r^{79bar} + r^{35bar} \ast \ldots \\
    r^{79bar} = r^{79bar} + r^{35bar} \ast \ldots \\
    \vdots \quad s^1bar = \ldots \\
    \vdots \quad skbar = \ldots
\]

For each assignment statement in Block 1, we generate one or two statements incrementing a \( \bar{a} \) object. Somewhere in Block 2, there must be at least one statement incrementing the \( \bar{a} \) object associated with each of the variables \( s_1, s_2, \ldots, s_k \).

**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 \( h_sx(k) \), (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 \( \bar{a} \) object in Block 2 have a special form with two binary operations + and \( \ast \). However, the form of the \( \bar{a} \) 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

\[
    c \quad r^j = f (r^1, r^2) \\
    \text{do } g_i = 1, k \\
    \quad hsrj(g_i) = f_-(r^2) \ast hsrj1(g_i) + f_-(r^1) \ast hsrj2(g_i) \\
    \text{end do} \\
    r^j = f (r^1, r^2)
\]
For each assignment statement of the form

\[ r$\bar{y}j$bar = r$\bar{y}j$bar + r$\bar{s}1$bar \times x \]

in Block 2, we generate code of the form

\[
\begin{align*}
\text{c} & \quad \text{r}\overline{y}j\overline{y}bar = r\overline{y}j\overline{y}bar + r\overline{s}1\overline{y}bar \times x \\
\text{do } & \quad g$i = 1, k \\
\text{h}\overline{s}\overline{s}j\overline{y}bar(g$i) = & \quad h\overline{s}r\overline{y}j\overline{y}bar(g$i) + x \times h\overline{s}r\overline{s}1\overline{y}bar(g$i) \\
& \quad + r\overline{s}1\overline{y}bar \times h\overline{s}(g$i) \\
\text{end do} \\
\text{r}\overline{y}j\overline{y}bar = & \quad r\overline{y}j\overline{y}bar + r\overline{s}1\overline{y}bar \times x \\
\end{align*}
\]

Task 5: Generate the final loop:

\[
\begin{align*}
\text{do } & \quad g$i = 1, g$p$ \\
\text{g$s\overline{w}(g$i) = } & \quad s\overline{y}bar + g$s1(g$i) + s2\overline{y}bar \times g$s2(g$i) \\
& \quad + \ldots + s\overline{k}bar \times g$s(k)(g$i) \\
\text{h$s\overline{w}(g$i) = } & \quad s\overline{y}bar + h$s1(g$i) + s2\overline{y}bar + h$s2(g$i) \\
& \quad + \ldots + h$s(k-1)bar(k) \times g$s(k)(g$i) \\
& \quad + 2.0 \times g$s1(g$i) \\
& \quad \times (h$s1\overline{y}bar(2) = g$s2(g$i) \\
& \quad + \ldots + h$s1\overline{y}bar(k) \times g$s(k)(g$i)) \\
& \quad + 2.0 \times g$s2(g$i) \\
& \quad \times (h$s2\overline{y}bar(3) = g$s3(g$i) \\
& \quad + \ldots + h$s2\overline{y}bar(k) \times g$s(k)(g$i)) \\
& \quad + 2.0 \times g$s(k-1)(g$i) \\
& \quad \times (h$s(k-1)\overline{y}bar(k) \times g$s(k)(g$i))) \\
\text{end do} \\
w = & \quad r$s \\
\end{align*}
\]

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

\[ w = - y / (x + z \times z) \]

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)
zbay = r$2bar * (r$1)
zbay = zbar + r$1bar * z
zbay = zbar + r$1bar * z
ybar = -1.0d0 / r$2
do 99993 g$1$ = 1, g$ps$  
g$1$ = ybar * g$y(g$1$) + zbar * g$z(g$1$)
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$pmx$, g$i$
parameter (g$pmx$ = 4)
real r$2bar$, r$bar$, y$bar$, zbar, r$1$, r$2$, r$3$
real y, z, w
real g$y$(g$pmx$), g$z$(g$pmx$), g$w$(g$pmx$)

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$1$)
ybar = 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 coercion, 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 pmxx = 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)
C rq1 = z * z
do 99986 g$i$ = 1, g$p$
g$rq1(g$i$) = (z + z) * g$z(g$i$)
99986 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$)
99987 continue
  rq2 = rq1 * z
C rq3 = -y / rq2
r$s1 = -y / (rq2)
do 99986 g$i$ = 1, g$p$
g$rq3(g$i$) = -(1.0d0 / rq2) * g$y(g$i$) + ((-r$s1 / (rq2)) * g$rq2(g$i$))
99986 continue
  rq3 = r$s1
C rq2b = -rq3 / rq2
r$s1 = -rq3 / (rq2)
do 99985 g$i$ = 1, g$p$
g$rq2b(g$i$) = -(1.0d0 / rq2) * g$rq3(g$i$) + ((-r$s1 / (rq2)) * g$rq2(g$i$))
99985 continue
  rq2b = r$s1
C rq1b = rq2b * z
do 99984 g$i$ = 1, g$p$
g$rq1b(g$i$) = z * g$rq2b(g$i$) + rq2b * g$z(g$i$)
99984 continue
  rq1b = rq2b * z
C z*b = rq2b * rq1
do 99983 g$i$ = 1, g$p$
g$zb(g$i$) = rq1 * g$rq2b(g$i$) + rq2b * g$rq1(g$i$)

15
99983 continue
    zb = rq2b * rql
C    zb = zb + rq1b * z
   do 99982 g$i$ = 1, g$sp$
      g$zb(g$i$) = g$zb(g$i$) + z * g$rqlb(g$i$) + rqlb * g$z(g$i$)
99982 continue
    zb = zb + rq1b * z
C    zb = zb + rq1b * z
   do 99981 g$i$ = 1, g$sp$
      g$zb(g$i$) = g$zb(g$i$) + z * g$rqlb(g$i$) + rqlb * g$z(g$i$)
99981 continue
    zb = zb + rq1b * z
   do 99999, gqiq = 1, gqpq
C    gqw(gqiq) = -1.0 / rq2 * gqy(gqiq) + zb * gqx(gqiq)
    r$q$0 = -1.0 / (rq2)
   do 99980 g$i$ = 1, g$sp$
      g$qw(g$i$) = g$qw(g$i$) + g$qy(gqiq) * (-r$q$0 / (rq2)) * g$qz(rq2(g$i$))
         + g$qz(gqiq) * g$zb(g$i$)
99980 continue
    gqw(gqiq) = r$q$0 * gqy(gqiq) + zb * gqx(gqiq)
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 gsexamp2s5(gsp$, x, g$&i$, h$&i$, ldg$&i$, 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 examp25.f (gradient) + examp2g74.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$i$bar, r$2$bar, 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$2$bar$(2), +
h$r$1$bar$(2), h$y$bar$(2), h$z$bar$(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$2$bar = -(r$3$ / r$2$)
c r$1$bar = r$2$bar + (z)
c zbar = r$2$bar * (r$1$)
c zbar = zbar + r$1$bar * z
c ybar = -(1.0/0 / r$2$)
do 99993 g$&i$ = 1, g$&p$
g$w$(g$&i$) = ybar * g$y$(g$&i$) + (zbar * g$z$(g$&i$))
c99993 continue

17
c \[ w = r^3 \]

---

c.global: Within this block, the variable named g$ps$ is renamed
c.to h$ps$. Its value is equal to the number of variables
c.on the rhs.
c.h$ys$, h$zs$: Local gradient objects of dimension = h$ps$
c.h$u = (u$y$, u$z$)
c.

c c For each global univariate Taylor series being propagated,
c w = f(y, z)
c \[ w_u = f_y \cdot y_u + f_z \cdot z_u \]
c \[ w' = f_y \cdot y' + f_z \cdot z' \]
c \[ w_{(uu)} = f_y \cdot y_{(uu)} + f_z \cdot z_{(uu)} \]
c \[ + 2 \cdot f_{(yz)} \cdot y_u \cdot z_u \]
c \[ + f_{(yy)} \cdot (y_u)^2 + f_{(zz)} \cdot (z_u)^2 \]
c \[ w'' = f_y \cdot y'' + f_z \cdot z'' + 2 \cdot f_{(yz)} \cdot y'' + z'' \]
c \[ + f_{(yy)} \cdot (y')^2 + f_{(zz)} \cdot (z')^2 \]
c.

c Initialize objects local to statement:
h$ps$ = 2
h$ys$(1) = 1.0
h$ys$(2) = 0.0
h$zs$(1) = 0.0
h$zs$(2) = 1.0

C c Compute ybar = f$y$, zbar = f$z$
c h$w = f_{(yy)}, f_{(yz)}, f_{(zz)}:
C r$1 = z * z$
do 99986 g$i$ = 1, h$ps$
h$sr$r$1(g$i$) = (z + z) * h$zs$(g$i$)
99986 continue
r$1 = z * z$
C r$2 = r$1 + z$
do 99987 g$i$ = 1, h$ps$
h$sr$r$2(g$i$) = z + h$sr$r$1(g$i$) + r$1 * h$zs$(g$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$ps$
h$sr$r$3(g$i$) = -(1.0d0 / r$2$) * h$ys$(g$i$)
\[ + ((-r$4 / (r$2)) * h$sr$r$2(g$i$)) \]
99986 continue
r$3 = r$4
C 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$ps$
h$ys$bar(g$i$) = -(r$6 / r$2) * h$sr$r$2(g$i$)
end do
ybar = r$6$
C r$2$bar = -r$3 / r$2$
r$5 = -r$3 / (r$2$)
do 99985 g$i$ = 1, h$ps$
h$sr$r$2$bar(g$i$) = ybar * h$sr$r$3(g$i$) + ((-r$5 / (r$2)) * h$sr$r$2(g$i$))
99985 continue
r$2$bar = r$5$
C r$1$bar = r$2$bar * z
do 99984 g$i$ = 1, h$ps$
h$sr$r$1$bar(g$i$) = z + h$sr$r$2$bar(g$i$) + r$2$bar * h$zs$(g$i$)
99984 continue
r$1$bar = r$2$bar * z
18
C  zbar = r$2bar * r$1
    do 99983 g$i$ = 1, h$sp$
        h$zbar(g$i$) = r$1 * h$r$2bar(g$i$) + r$2bar * h$r$1(g$i$)
    99983 continue
C  zbar = r$2bar * r$1
    do 99982 g$i$ = 1, h$sp$
        h$zbar(g$i$) = h$zbar(g$i$) + z * h$r$1bar(g$i$)
            + r$1bar * h$z(g$i$)
    99982 continue
C  zbar = zbar + r$1bar * z
    do 99981 g$i$ = 1, h$sp$
        h$zbar(g$i$) = h$zbar(g$i$) + z * h$r$1bar(g$i$)
            + r$1bar * h$z(g$i$)
    99981 continue
C  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$2
C  w.z = zbar
C Hence, w.(y) = h$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'
       + f.(yy) * (y')^2 + f.(zz) * (z')^2

    do g$i$ = 1, h$sp$
        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
C  w = r$3

C=================================================================
C
C f depends in some way on w
f = w
    do 99992 g$i$ = 1, g$sp$
        g$w(g$i$) = g$w(g$i$)
        h$w(g$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.
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, \( g(x(p)) = z' \); and second derivative, \( h(x(p)) = z'' \).

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 $g \cdot (g \cdot i) = \ldots$ and $h \cdot (g \cdot i) = \ldots$ contain several $\cdot$bar objects. In many computations (computational models based on grids, for example), many of the corresponding $\cdot$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 $0 + \cdot$, $0 \cdot \cdot$, and $1 \cdot \cdot$ 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 $\times$ vector
- Compute Hessian $\times$ matrix

In Appendix I, we give several prototype library utilities:

ersreve.f Initialize univariate series for dense Hessian
prtsre.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-MAV-1991
C Results:
C X : 1.00000000E+00 2.00000000E+00 3.00000000E+00 4.00000000E+00
C F : -4.16666666E-02

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

do 10 i = 1, xdim
  x(i) = i
10    write (6, 1010) (x(i), i = 1, xdim)
1010    format ('X ', i, l10e15.7)
call fcn (x, xdim, f, fdim)
write (6, 1020) (f(i), i = 1, fdim)
1020    format ('F ', i, l10e15.7)
call jac (x, xdim, f, fdim, jac_f)
do 20 i = 1, fdim
write (6, 1030) (jac_f(i,j), j = 1, xdim)
20    format ('Jacobian: ', l10e15.7)
20 continue
call eye (g$$, pmax)
call g$$$(x, pmax, xdim, f, g$$$, xdim, f, g$$$, pmax, fdim)
do 30 i = 1, fdim
write (6, 1030) (g$$$(j,i), j = 1, xdim)
30    format ('Jac err: ', l10e15.7)
30 continue

do 40 i = 1, fdim
write (6, 1040) ((jac_f(i,j) - g$$$(j,i)), j = 1, xdim)
40    format ('Jac err: ', l10e15.7)
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(2) * x(3) * x(3) * x(4))
jac_f(1,4) = x(1) / (x(4) * x(2) * x(3) * x(4))
return
end

STOP
subroutine g$fcn3 (g$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$0$bar = 0.0
r$1$bar = 0.0
r$2$bar = 0.0
r$3$bar = 0.0
r$4$bar = 0.0
r$5$bar = 0.0
r$6$bar = 0.0
r$7$bar = 1.0
C
adjoint for r$7$ = r$4$ / r$6$

r$4$bar = r$4$bar + r$7$bar * (1.0 / r$6$)

r$6$bar = r$6$bar + r$7$bar * (-r$7$ / r$6$)
C
adjoint for r$5$ = r$5$ * r$3$

r$5$bar = r$5$bar + r$6$bar * r$3$

r$3$bar = r$3$bar + r$6$bar * r$5$
C
adjoint for r$5$ = r$5$ * r$2$

r$1$bar = r$1$bar + r$5$bar * r$2$

r$2$bar = r$2$bar + r$5$bar * r$1$
C
adjoint for r$4$ = -r$0$

r$0$bar = r$0$bar + r$4$bar * (-1.0)

call saxpy4 (pmax, g$p$, r$0$bar, g$x$(1.1), r$1$bar, g$x$(1.2),
+ r$2$bar, g$x$(1.3), r$3$bar, 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 Vctrn   Vectors
c Output parameter
c Result(i) = sum Weigh_n * Vctr_n(i)
c n
Author: George Corliss, 06-NOV-1991
Assumptions:
c All vectors have the same logical lengths
   (Although allocated lengths may vary)
c 1 <= LenVec <= allocated sizes of vectors

integer pmax, LenVec, i
real Result(*), Vctr1(*), Vctr2(*), Vctr3(*), Vctr4(*)

   do 10 i = 1, LenVec
      10  Result(i) = Weigh1 * Vctr1(i) + Weigh2 * Vctr2(i)
       +   + Weigh3 * Vctr3(i) + Weigh4 * Vctr4(i)
      return
   end
Appendix B. Main Program for Computing Dense Hessians as Partial Derivatives

c Purpose: Illustrate code that ADIFOR might generate for
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
subroutine.
c \( u, v \) active, not necessarily independent
c \( g^x \) gradient object
c \( h^x \) Hessian object
Operators for full, dense, forward mode for unary and binary
operations. We assume that the target of the assignment
DOES NOT ALSO APPEAR ON THE RIGHT HAND SIDE.
c Results:
c For \( u \) : value: 3.400000E+00
Gradient : 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For \( v \) : value: -2.100000E+00
Gradient : 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Addit: value: 1.300000E+00
Gradient : 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Multi: value: -7.140000E+00
Gradient : -2.100000E+00 3.400000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 1.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Divid: value: -1.619048E+00
Gradient : -4.761905E-01 -7.709752E-01 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 -2.267574E-01 -7.342621E-01
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Exp : value: 2.996410E+01
Gradient : 2.996410E+01 0.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 1.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00

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$0, g$r0(pmax), h$r0(pmax,pmax),
+ r$1, g$r1(pmax), h$r1(pmax,pmax),
+ r$2, g$r2(pmax), h$r2(pmax,pmax),
+ r$3, g$r3(pmax), h$r3(pmax,pmax)

INITIALIZATION:

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$u(i,j) = 0.0
      h$v(i,j) = 0.0
   end do
end do

$u(1) = 1.0
$w(1) = 1.0

call prthes ('u', p, pmax, u, g$u, h$u)
call prthes ('v', p, pmax, v, g$v, h$v)

ADDITION: f = u + v
df = du + dv
dx = dx + dx

2  2  2
d f = d u + d v

2
dx dy = dx dy + dx dy

r$0 = u + v
do g$i = 1, p
   g$r0(g$i) = g$u(g$i) + g$v(g$i)
do g$i = 1, g$i
   h$r0(g$i,g$i) = h$u(g$i,g$i) + h$v(g$i,g$i)
end do
end do

MULTIPLICATION: f = u * v
df = dv * du
dx = dx + dx

2  2  2
d f = d v * du + dv * dx + dx * du

2
dx dy = u +(dx dy + du)

2
dx = dx dy dy dx dy dx dy

26
r$j = u * v$
do g$j = 1, p
   g$r1(g$j) = u * g$v(g$j) + g$u(g$j) * v$
do g$i = 1, g$j
   h$r1(g$j,g$i) = u * h$v(g$j,g$i)
   + g$u(g$i) * g$v(g$j)
end do
end do

DIVISION: f = u / v, v * f = u

\[
v = \frac{df}{dv} \quad du
\]

\[
dx = \frac{dx}{dv} \quad dx
\]

2
\[
d f \quad dv \quad df \quad dv \quad df
\]

2
\[
v \quad ----- \quad + \quad --- \quad + \quad --- \quad + \quad --- \quad f = ----- 
\]

\[
dx \quad dy \quad dy \quad dx \quad dx \quad dy \quad dy \quad dx \quad dy
\]

r$2 = u / v$
do g$2 = 1, p
   g$r2(g$2) = (g$u(g$2) - g$v(g$2) * r$2) / v
end do

Two separate loops are necessary because the Hessian requires the gradient of f.
do g$j = 1, p
   do g$i = 1, g$j
      h$r2(g$j,g$i) = (h$u(g$j,g$i)
      + g$v(g$i) * g$r2(g$j)
      + g$v(g$i) * g$r2(g$i)
      + h$v(g$j,g$i) * r$2)
      / v
   end do
end do

EXPONENTIAL: f = exp (u),

\[
d f = f \quad du
\]

\[
dx = dx \quad dx
\]

2
\[
d f \quad du \quad df \quad du
\]

2
\[
v = f \quad ----- = f \quad ----- + \quad --- 
\]

\[
dx \quad dy \quad dy \quad dx \quad dy \quad dx
\]

r$3 = exp (u)$
do g$j = 1, p
   g$r3(g$j) = r$3 * g$u(g$j)
   do g$i = 1, g$j
      h$r3(g$j,g$i) = r$3 * h$u(g$j,g$i)
      + g$r3(g$i) * g$u(g$j)
   end do
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: ', 1PE15.6, /100(10X, 1PE15.6, /))
1020 format ('Hessian: ')  
1030 format (I11, 1PE15.6)

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

c Purpose: Illustrate code that ADIFOR might generate for
univariate Taylor series to generate
gradients and Hessians
Not intended as a rigorous test.
c Author: George Corliss, 12-FEB-1992
c Modifications:
12-FEB-1992 George Corliss
Adapted from hes_oprs.f

c Discussion:
This simulates program fragments from an ADIFOR-generated
subroutine.
u, v active, not necessarily independent
h$\times$ Taylor series object. Ordering:
1
2 3
4 5 6
7 8 9 10
We compute Taylor coefficients $u^{-(i)/i!}$.
Function values are computed redundantly. For sequential
computation, that is easier to read. For parallel
computation, each processor computes its own copy.
Operators for univariate Taylor series for unary and binary
operations. We assume that the target of the assignment
DOES NOT ALSO APPEAR ON THE RIGHT HAND SIDE.

c Results:
c For $u$ : value: 3.400000E+00
Gradient : 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For $v$ : value: -2.100000E+00
Gradient : 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Addit: value: 1.300000E+00
Gradient : 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 0.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Multi: value: -7.140000E+00
Gradient : -2.100000E+00 3.400000E+00 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 1.000000E+00 0.000000E+00
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
For Divid: value: -1.619048E+00
Gradient : -4.761905E-01 -7.709752E-01 0.000000E+00 0.000000E+00
Hessian :
1 0.000000E+00
2 -2.287573E-01 -7.342621E-01
3 0.000000E+00 0.000000E+00 0.000000E+00
4 0.000000E+00 0.000000E+00 0.000000E+00
For Exp : value: 2.996410E+01
Gradient : 2.996410E+01  0.0000000E+00  0.0000000E+00  0.0000000E+00
Hessian :
  1  2.996410E+01
  2  0.0000000E+00  0.0000000E+00
  3  0.0000000E+00  0.0000000E+00
  4  0.0000000E+00  0.0000000E+00  0.0000000E+00
  5

NumInd Number of independent variables (= p)
NumSer Number of series
for second derivatives, NumSer = p(p+1)/2
SerOrd Series Order
integer NumInd, NumSer, SerOrd, j
parameter (NumInd = 4, NumSer = 10, SerOrd = 2)
real u, hSu(NumSer, 0:SerOrd), v, hSv(NumSer, 0:SerOrd),
+  r$0, h$r$0(NumSer, 0:SerOrd), r$s1, h$r$s1(NumSer, 0:SerOrd),
+  r$s2, h$r$s2(NumSer, 0:SerOrd), r$s3, h$r$s3(NumSer, 0:SerOrd)

Auxiliaries for comparison:
integer pmax
parameter (pmax = 4)
real g$u(pmax), Hes$u(pmax,pmax),
+  g$u(pmax), Hes$u(pmax,pmax),
+  g$r$0(pmax), Hes$r$0(pmax,pmax),
+  g$s1(pmax), Hes$s1(pmax,pmax),
+  g$r$s2(pmax), Hes$r$s2(pmax,pmax),
+  g$r$s3(pmax), Hes$r$s3(pmax,pmax)

INITIALIZATION:

u = 3.4
v = -2.1
do j = 1, NumSer
  hSu(j,0) = u
  hSv(j,0) = v
  hSu(j,1) = 0.0
  hSv(j,1) = 0.0
end do
hSu(1,1) = 1.0
hSu(2,1) = 1.0
hSu(4,1) = 1.0
hSu(7,1) = 1.0
hSv(2,1) = 1.0
hSv(3,1) = 1.0
hSv(5,1) = 1.0
hSv(8,1) = 1.0

call Ser2He (NumSer, SerOrd, hSu, u, g$u, Hes$u, pmax)
call Ser2He (NumSer, SerOrd, hSv, v, g$u, Hes$u, pmax)
call PrtSer ('u ', NumSer, SerOrd, hSu)
call PrtHes ('u ', pmax, pmax, u, g$u, Hes$u)
call PrtSer ('v ', NumSer, SerOrd, hSv)
call PrtHes ('v ', pmax, pmax, v, g$u, Hes$u)
```c
ADDITION: f = u + v

df = du + dv
dx = dx + dx

df = du + dv
dx = dx + dx

r$0 = u + v
do g$f$ = 1, NumSer
   h$r$0(g$f$) = h$s(u(g$f,0) + h$v(g$f,0)
or = r$0
   h$r$0(g$f,1) = h$s(u(g$f,1) + h$v(g$f,1)
h$r$0(g$f,2) = h$s(u(g$f,2) + h$v(g$f,2)
end do

MULTIPLICATION: f = u * v

df = dv * du
dx = dx * dx

df = dv * du
dx = dx * dx

We divide both sides by 2.

r$1 = u * v
do g$f$ = 1, NumSer
   h$r$1(g$f) = h$s(u(g$f,0) * h$v(g$f,0)
or = r$1
   h$r$1(g$f,1) = u * h$v(g$f,1) + h$s(u(g$f,1) * v
   h$r$1(g$f,2) = u * h$v(g$f,2) + h$s(u(g$f,2) * h$v(g$f,1)
    + h$s(u(g$f,2) * v
end do

DIVISION: f = u / v, v * f = u

v * -- + -- * f = --
dx = dx + dx

v * -- + -- * f = --
dx = dx + dx
```

31
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)
```

Better parallelism from:
```c
h$r$2(g$j, 2) = (h$u(g$j, 2) + h$v(g$j, 1) * r$2) / v
```

EXPERIMENTAL: \( f = \exp(u) \)
```c
def df du
-- = f * --
dx dx
```
```c
2
2
d f d u df du
-- = f * -- + -- * --
2
2
dx dx dx dx
```

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))
```

Better parallelism from:
```c
+ r$3 * h$u(g$j, 1)
+ h$r$3(g$j, 1) / 2.0
```

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 PrtSer ('Add', pmax, pmax, r$0, g$r$0, Hes$r$0)
call PrtSer ('r$1', NumSer, SerOrd, h$r$1)
call PrtSer ('Mult', pmax, pmax, r$1, g$r$1, Hes$r$1)
call PrtSer ('r$2', NumSer, SerOrd, h$r$2)
call PrtSer ('Div', 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
+ to value, gradient, and Hessian form.

c 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)
c 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 (111, 1PE15.6, 100(14X, 1PE15.6/))

return
end

subroutine PrtHes (name, length, ldg_x, x, grad_x, Hess_x)
c 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:', IPE15.6, + / 'Gradient : ', IPE15.6, / 100(I0X, IPE15.6, /))
1020 format ('Hessian : ')
1030 format (I11, IPE15.6)

return
end
Appendix D. Driver Program to Call Undifferentiated Code

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

Results:

Undifferentiated code.

\[ F = -0.12500 \]

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$examp$ (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$examp$(g$p$, x, g$x, l$d$g$x, xdim, f, g$f, l$d$g$f)

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$1
integer l$d$g$f

real f
real g$f(l$d$g$f)
integer xdim
real x(xdim)
real g$(x(l$d$g$x, xdim)
integer l$d$g$x
real y, z, w
real g$y(g$pmax$), g$z(g$pmax$), g$w(g$pmax$)
```
y and z depend in some way on x(1..xdim)
if (g$p > g$pmx) then
    print *, 'Parameter g$p is greater than g$pmx.'
    stop
endif

C
y = x(1)
do 99995 g$i$ = 1, g$p$,
g$y(g$i$) = g$x(g$i$, 1)
99995 continue

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

C
z = x(2)

C
Consider the assignment statement
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$x(g$i$))
99993 continue

C
w = r$3
f depends in some way on w
f = w
do 99992 g$i$ = 1, g$p$
g$z(g$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
C      W     = -0.12500
C grad W = -1.250000E-01  1.875000E-01  0.000000E+00  0.000000E+00

C 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

C 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 --> Q
C Remove redundant parentheses
C Combine ADIFOR-generated declarations
C bar --> B

   integer gQPQ, gQmaxQ, gQ1Q
   parameter (gQmaxQ = 4)
   real rQ2B, rQ1B, zB, rQ3, rQ2, rQ1
   real y, z, w, gQy(gQmaxQ), gQz(gQmaxQ), gQw(gQmaxQ)

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

38
\[ z_B = z_B + rQ1B \cdot z \]
do 99993 gQ1Q = 1, gQpQ
\[ gQw(gQ1Q) = -1.0 / rQ2 \cdot gQy(gQ1Q) + zB \cdot gQz(gQ1Q) \]
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 $g$ as the gradient of $W$ with
C respect to $y$ and $z$. Hence, we want to differentiate $g$ 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

C 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$Qw(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, w, gQw, xdim)

C Report:
write (6, 1010) w, (gQw(i), i = 1, xdim)
1010 format ('Hessian by Adifor (Adifor (examp2.f)).'
      + ' $W = ', 10.5 /
      + ' grad $W = ', 1p8e14.6)
write (6, 1015) 'Hessian $W =$
1015 format (' $W =$
do j = 1, xdim
write (6, 1020) j, (g$Qw(j,i), i = 1, xdim)
1020 format (i5, 4x, 1p8e14.6)
end do
stop
end

C subroutine g$examp2g$74(g$y, g$Qy, y, g$y, ldg$y, gQy, z, g$z, ld
g$z, gQz, w, gQw, g$Qw, ldg$Qw)
C C Formal g$y is active.
C C Formal g$Qy is active.
C C Formal y is active.
C C integer g$y
C integer g$Qymax$
```
parameter (g$pmx$ = 2)
integer g$i$
real r$s$
real r$q$
integer l$dg$
integer l$dg$
y
C

Purpose: Explore 2nd derivative code.
Extract ADIFOR-generated code for one assignment
statement from *examp2.5.f*

Author: George Corliss, 25-FEB-1992

Modifications:
1.0d0 --> 1.0
$S$ --> $Q$

Remove redundant parentheses
Combine ADIFOR-generated declarations
bar --> B

integer gp$q$, gp$pmxq$, g$q$
parameter (gp$pmxq$ = 4)
real r$q$, r$q$1, z$, r$q$3, r$q$2, r$q$
real g$sr$q$2<0.885818955307730082247146462853936773491257324218753e-08>, g$sr$q$1<0.885818955307730082247146462853936773491257324218753e-08>, g$sr$q$3<0.885818955307730082247146462853936773491257324218753e-08>, g$sr$q$<0.885818955307730082247146462853936773491257324218753e-08>
real y, z, w, g$q$y(gp$pmxq$), g$q$z(gp$pmxq$), g$q$w(gp$pmxq$)
real g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q$<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>
integer l$dg$q$

Consider the assignment statement

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

if (g$p$ $>$ g$p$ a) then
print *: 'Parameter g$p$ is greater than g$pmx$.'
stop
endif

C
r$q$1 = z * z
do 99986 g$i$ = 1, g$p$
g$sr$q$1(g$i$) = (z + z) * g$z$(g$i$)
99986 continue
r$q$1 = z * z
C
r$q$2 = r$q$1 * z
do 99987 g$i$ = 1, g$p$
g$sr$q$2(g$i$) = z * g$sr$q$1(g$i$) + r$q$1 * g$z$(g$i$)
99987 continue
r$q$2 = r$q$1 * z
C
r$q$3 = -y / r$q$2
r$s$1 = -y / (r$q$2)
do 99986 g$i$ = 1, g$p$
g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>

** g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>
99985 continue
r$q$2 = r$s$1
C
r$q$1b = r$q$2 + z
do 99984 g$i$ = 1, g$p$
g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>

** g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>
99984 continue
r$q$1b = r$q$2 + z
C
z$b$ = r$q$2 - r$q$
do 99983 g$i$ = 1, g$p$
g$sr$qb(g$i$) = r$q$1 * g$sr$q<.0000000000000000000000000000000000000000000000000000000001068>, r$q$2 * g$z$(g$i$)
99983 continue

41
zb = rq2b * rqi
C
zb = zb + rqi * z
do 99882 g$i$ = 1, g$p$
   g$z$b(g$i$) = g$z$b(g$i$) + z * g$r$q1b(g$i$) + rqi * g$z$(g$i$)
99882 continue
zb = zb + rqi * z
C
zb = zb + rqi * z
do 99881 g$i$ = 1, g$p$
   g$z$b(g$i$) = g$z$b(g$i$) + z * g$r$q1b(g$i$) + rqi * g$z$(g$i$)
99881 continue
zb = zb + rqi * z
do 99999, g$q1q$ = 1, g$p$q
   g$q$(g$q1q$) = -1.0 / g$q$ / g$q$(g$q1q$) + zb * g$q$(g$q1q$)
   r$q$ = -1.0 / (g$q$2)
   do 99980 g$i$ = 1, g$p$
      g$z$q(g$i$, g$q1q$) = g$q$(g$q1q$) * (-r$q$ / (g$q$2)) * g$r$q2(g$i$)
    * + g$q$(g$q1q$) * g$z$b(g$i$)
99980 continue
   g$q$(g$q1q$) = r$q$ * g$q$(g$q1q$) + zb * g$q$(g$q1q$)
99999 continue
99999 continue
w = r$3$
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 :
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
C For F :
value: -1.250000E-01
C Gradient : -1.250000E-01  1.875000E-01  0.000000E+00  0.000000E+00
C 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

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), rf(ser_p, order)
real temp, gtemp(xdim), htemp(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 gtemp2$S (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 Prt2Hx (ser_p, order, hf, temp, gtemp, htemp, xdim)
call PrtHes ('F ', xdim, xdim, temp, gtemp, htemp)
stop
end
Appendix I. Library Utility Routines

File: SEREYE.f  26-FEB-1992

subroutine SerEye (NumSer, SerOrd, V_Size, Ser_V)

c Purpose: Initialize univariate Taylor series to compute
a dense Hessian stored in the order

c
1
2 3
4 5 6

c Assumptions:
c V_Size <= 10
c NumSer = V_Size * (V_Size + 1) / 2
c SerOrd = 3

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

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
      Ser_V(i,2,k) = 0.0
   end do

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),i,k) = 1.0
         end if
      end do
   end do

return
end

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,*)

c 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, IP5E15.6, 100(I4X, IP5E15.6/))

return
C File:  PRTNES.f  06-FEB-1992

subroutine prt hes (name, length, lgd_x, x, grad_x, Hess_x)
character*6 name
integer length, lgd_x, i, j
real x, grad_x(lgd_x), Hess_x(lgd_x, lgd_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: ', 1PSE15.6, / 100(10X, 1PSE15.6, /))
1020 format ('Hessian: ')
1030 format (1I1, 1PSE15.6)

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
to value, gradient, and Hessian form.
c
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)) * 0.5
      Hess_V(i,j) = Hess_V(j,i)
   end do
end do

return
end
References


