Automation of Nested Matrix and Derivative Operations

Robert Kalaba

Departments of Electrical and Biomedical Engineering and Economics
University of Southern California
Los Angeles, California 90089

Thomas Plum

Department of Biomedical Engineering
University of Southern California
Los Angeles, California 90089

and

Leigh Tesfatsion*

Department of Economics
University of Southern California
Los Angeles, California 90089

ABSTRACT

In 1983 an algorithm FEED was introduced for the systematic exact evaluation of higher-order partial derivatives of functions of many variables. In 1986 the FEED library was extended to permit the automatic differentiation of functions expressed in terms of the derivatives of other functions. Building on this work, the present paper further extends the FEED library to permit the automatic differentiation of expressions involving nested matrix and derivative operations. The need to differentiate such expressions arose naturally in the course of designing sequential filters for a class of nonlinear tracking problems.

*Please address correspondence to Professor Leigh Tesfatsion, Department of Economics, Modelling Research Group MC-0152, University of Southern California, Los Angeles, CA 90089.

52 Vanderbilt Ave., New York, NY 10017 0096-3003/87/$03.50

1. INTRODUCTION

In a recent paper [1] an algorithm is introduced for the systematic exact evaluation of higher-order partial derivatives. Building on a key idea of R. Wengert [3], the evaluation of a complicated kth-order differentiable function $F: R^n \to R$ at a specified domain point x is decomposed into a sequence of simple evaluations of special functions of one or two variables. The evaluation of each special function is accomplished by calling a calculus subroutine which automatically computes and returns the distinct partial derivatives of the special function through order k, along with the special-function value. The final stage in the sequence of special-function evaluations yields the distinct partial derivatives of F at x through order k, along with the function value $F(x)$.

The special functions considered in [1] for function decomposition are the two-variable functions

$$u + v, \qquad u - v, \qquad uv, \qquad u/v, \qquad u^v, \tag{1}$$

and arbitrary one-variable continuously differentiable functions such as

$$\sin u, \qquad \cos u, \qquad \exp u, \qquad \log u, \quad \text{and} \quad au^b + c \tag{2}$$

for arbitrary constants a, b, and c. For example, as detailed in Section 2, evaluation of the function

$$z = F(x, y) = x + \log(xy) \tag{3}$$

and its distinct higher-order partial derivatives at a given domain point (x, y) is accomplished by calling, in order, calculus subroutines for the list of special functions

$$\begin{aligned} a &= x, \\ b &= y, \\ c &= ab, \\ d &= \log(c), \\ z &= a + d. \end{aligned} \tag{4}$$

The calculus subroutines called for the special functions in (4) return well-ordered[1] arrays of function and derivative evaluations, using well-ordered arrays obtained in previous list calculations. For example, the calculus subroutine called for the special function $c = ab$ in (4) returns the well-ordered array $C = (c, c_x, c_y, c_{xx}, c_{xy}, \dots)$ for the value and distinct partial derivatives of c, using the previously obtained well-ordered arrays $A = (a, a_x, a_y, a_{xx}, a_{xy}, \dots)$ and $B = (b, b_x, b_y, b_{xx}, b_{xy}, \dots)$. Thus, $c = ab$, $c_x = a_x b + a b_x$, $c_y = a_y b + a b_y$, and so forth. By construction, the well-ordered array $Z = (z, z_x, z_y, z_{xx}, z_{xy}, \dots)$ obtained for the final special function $z = a + d$ in (4) yields the desired evaluations for the value and distinct partial derivatives of the basic function of interest, $z = x + \log(xy)$.

In a subsequent paper [2] it is shown that this differentiation algorithm, or FEED ("fast efficient evaluation of derivatives") as we now call it, can be applied to a much broader class of functions than envisioned in [1]. Specifically, at each stage in the sequential evaluation of a function via FEED, certain special function *derivatives* are evaluated which in turn can be used as special functions in subsequent stages. Thus, FEED can be used to evaluate and differentiate functions which are defined in terms of the partial derivatives of other functions. The explicit extension of FEED to permit this capability is potentially useful, since the need to differentiate functions of derivatives occurs naturally in a wide variety of applications: for example, power-series expansions, and the solution of Euler-Lagrange equations for calculus of variations problems via quasilinearization or Newton's method.

The present paper takes another step toward the development of a complete, accurate, and user-friendly FEED library. It is shown how expressions involving nested matrix and derivative operations can be differentiated automatically by means of *matrix* FEED subroutines.

---

[1] An array Z containing the value and distinct partial derivatives through order k of a kth-order continuously differentiable real-valued function $z = F(x_1, \dots, x_n)$ at a given domain point $x = (x_1, \dots, x_n)$ will be referred to as *well-ordered* (*for F through order k*) if Z has the form $Z = (F, A_1, A_2, \dots, A_k)$, where A_i is a row vector consisting of all the distinct partial derivatives of F at x of order i which satisfies the following two properties: (a) If $F_{IJ\dots L}$ appears in A_i, then $I \leqslant J \leqslant \cdots \leqslant L$; (b) If $F_{IJ\dots ML}$ and $F_{I'J'\dots M'L'}$ both appear in A_i, then the former term precedes the latter term if and only if either $I < I'$, or $I = I'$ and $J < J'$, or ..., or $I = I'$ and $J = J'$ and ... and $M = M'$ and $L < L'$. For example, if $k = 3$ and $n = 4$, then Z is well-ordered if and only if $Z = (F, A_1, A_2, A_3)$, with $A_1 = (F_1, F_2, F_3, F_4)$, $A_2 = (F_{11}, F_{12}, F_{13}, F_{14}, F_{22}, F_{23}, F_{24}, F_{33}, F_{34}, F_{44})$, $A_3 = (F_{111}, F_{112}, F_{113}, F_{114}, F_{122}, F_{123}, F_{124}, F_{133}, F_{134}, F_{144}, F_{222}, F_{223}, F_{224}, F_{233}, F_{234}, F_{244}, F_{333}, F_{334}, F_{344}, F_{444})$. As is well known, given any kth-order continuously differentiable function $F: R^n \to R$, and any integer m such that $1 \leqslant m \leqslant k$, the number of distinct partial derivatives of F of order m at any domain point x is given by $\binom{n+m-1}{m} = (n+m-1)!/m!(n-1)!$.

For example, let $H: R^n \to R^m$ denote an observation function mapping $n \times 1$ state vectors x into $m \times 1$ observation vectors y, and let $H_x^T(x)$ denote the $n \times m$ transposed Jacobian matrix of H at x. Let K denote an $m \times m$ constant matrix, and let $Z(x)$ denote the $n \times n$ Jacobian matrix defined by

$$Z(x) = \frac{\partial\left(H_x^T(x)K[y - H(x)]\right)}{\partial x}. \tag{5}$$

This expression involves nested matrix and derivative operations. In the course of determining sequential filter solutions for a class of nonlinear tracking problems, expressions such as (5) had to be evaluated. It quickly became apparent that matrix FEED subroutines would have to be devised to handle this evaluation automatically if a practical filter was to be obtained.

To illustrate the extension of FEED to nested matrix and derivative operations, it is shown in Section 3, below, how the Jacobian matrix $Z(x)$ in (5) can be automatically evaluated by means of matrix FEED subroutines for any given x, H, y, and K. The design of general matrix FEED subroutines is considered in Section 4. Numerical tests demonstrating the accuracy of the matrix FEED subroutines are discussed in Section 5. Concluding comments are given in Section 6.

2. THE BASIC FEED ALGORITHM: A SIMPLE ILLUSTRATION

Consider the function $F: R_{++}^2 \to R$ defined by

$$z = F(x, y) = x + \log(xy). \tag{6}$$

Suppose one wishes to evaluate the function value z, the first-order partial derivatives z_x and z_y, and the particular second-order partial derivative z_{xx} at a given domain point (x, y). Consider Table 1.

The first column of Table 1 sequentially evaluates the function value $z = x + \log(xy)$ at the given domain point (x, y). The second, third, and fourth entries in each row give the indicated derivative evaluations of the first entry in the row, using only algebraic operations. The first two rows initialize the algorithm, one row being required for each function variable. The only input required for the first two rows is the domain point (x, y). Each

TABLE 1

SEQUENTIAL EVALUATION OF $z = x + \log(xy)$, z_x, z_y, AND z_{xx}

Special function	$\partial/\partial x$	$\partial/\partial y$	$\partial^2/\partial x^2$
$a = x$	$a_x = 1$	$a_y = 0$	$a_{xx} = 0$
$b = y$	$b_x = 0$	$b_y = 1$	$b_{xx} = 0$
$c = ab$	$c_x = a_x b + ab_x$	$c_y = a_y b + ab_y$	$c_{xx} = a_{xx} b + 2a_x b_x + ab_{xx}$
$d = \log c$	$d_x = c^{-1} c_x$	$d_y = c^{-1} c_y$	$d_{xx} = -c^{-2} c_x^2 + c^{-1} c_{xx}$
$z = a + d$	$z_x = a_x + d_x$	$z_y = a_y + d_y$	$z_{xx} = a_{xx} + d_{xx}$

subsequent row generates a well-ordered array of the form (p, p_x, p_y, p_{xx}), using well-ordered arrays obtained from previous row calculations. The final row yields the desired evaluations (z, z_x, z_y, z_{xx}). These evaluations are exact up to round off error.

It will now be shown how the rows of Table 1 can be sequentially evaluated by means of FORTRAN calculus subroutines at any given domain point (x, y). To facilitate understanding, single-precision subroutines specifically tailored to the example at hand will be presented, using the same nomenclature as in Table 1.

The first two rows of Table 1 can be evaluated by calling a calculus subroutine LIN. The evaluations performed by LIN for rows 1 and 2 can be separately depicted as follows:

```
SUBROUTINE LIN1(X,A)          SUBROUTINE LIN2(Y,B)
DIMENSION A(4)                DIMENSION B(4)
A(1) = X                      B(1) = Y
A(2) = 1.0                    B(2) = 0.0
A(3) = 0.0                    B(3) = 1.0
A(4) = 0.0                    B(4) = 0.0
RETURN                        RETURN
END                           END
```

(7)

Subroutine LIN1 implements the usual calculus formulas to obtain the value and first three partial derivatives $A = (a, a_x, a_y, a_{xx}) = (X, 1, 0, 0)$ of the function a of x and y defined by $a(x, y) = x$, given any particular value X for x. Thus, the array A evaluates the first row of Table 1 at X. Similarly, subroutine LIN2 obtains the value and first three partial derivatives $B =$

$(b, b_x, b_y, b_{xx}) = (Y, 0, 1, 0)$ of the function b of x and y defined by $b(x, y) = y$, given any particular value Y for y. Thus, the array B evaluates the second row of Table 1 at Y.

Row 3 of Table 2.1 can be evaluated by calling the calculus subroutine **MULT** for multiplication:

```
SUBROUTINE MULT(A,B,C)
DIMENSION A(4), B(4), C(4)
C(1) = A(1)*B(1)
C(2) = A(2)*B(1) + A(1)*B(2)                                    (8)
C(3) = A(3)*B(1) + A(1)*B(3)
C(4) = A(4)*B(1) + 2.0*A(2)*B(2) + A(1)*B(4)
RETURN
END
```

Subroutine **MULT** implements the usual calculus formulas to obtain the value and first three partial derivatives $C = (c, c_x, c_y, c_{xx})$ of the function $c = ab$, where a and b are any two real-valued twice differentiable functions of x and y, and $A = (a, a_x, a_y, a_{xx})$ and $B = (b, b_x, b_y, b_{xx})$ are well-ordered arrays for a and b obtained from previous row calculations.

Row 4 of Table 1 can be evaluated by calling the following calculus subroutine for the log function:

```
SUBROUTINE LOGG(C,D)
DIMENSION C(4), D(4)
D(1) = ALOG(C(1))
D(2) = C(2)/C(1)                                                (9)
D(3) = C(3)/C(1)
D(4) = -D(2)**2 + C(4)/C(1)
RETURN
END
```

Subroutine **LOGG** implements the usual calculus formulas to obtain the value and first three partial derivatives $D = (d, d_x, d_y, d_{xx})$ of the function $d = \log c$, where c is any real-valued, positive, twice differentiable function of x and y, and $C = (c, c_x, c_y, c_{xx})$ is a well-ordered array for c obtained from previous row calculations. Note that the calculus subroutine **LOGG** calls the FORTRAN library function subroutine **ALOG** for the log function.

Finally, row 5 of Table 1 can be evaluated by calling the following calculus subroutine for addition:

```
      SUBROUTINE ADD(A,D,Z)
      DIMENSION A(4), D(4), Z(4)
      DO 1 I = 1,4                                    (10)
1       Z(I) = A(I) + D(I)
      RETURN
      END
```

Subroutine **ADD** implements the usual calculus formulas to obtain the value and first three partial derivatives $Z = (z, z_x, z_y, z_{xx})$ of the function $z = a + d$, where a and d are any two real-valued twice differentiable functions of x and y, and $A = (a, a_x, a_y, a_{xx})$ and $D = (d, d_x, d_y, d_{xx})$ are well-ordered arrays for a and d obtained from previous row calculations.

The complete sequential evaluation of the rows of Table 1 is accomplished through use of subroutine **FUN**:

```
SUBROUTINE FUN(X,Y,Z)
DIMENSION A(4), B(4), C(4), D(4), Z(4)
CALL LIN1(X,A)
CALL LIN2(Y,B)
CALL MULT(A,B,C)                                      (11)
CALL LOGG(C,D)
CALL ADD(A,D,Z)
RETURN
END
```

Given any specified domain point $(x, y) = (\mathbf{X}, \mathbf{Y})$, subroutine **FUN** obtains the value and first three partial derivatives (z, z_x, z_y, z_{xx}) of the function $z = x + \log(xy)$ at $(\mathbf{X}, \mathbf{Y})$ by means of the indicated sequence of calls to the calculus subroutines (7) through (10). These evaluations are returned in the array **Z**.

In principle, a **FUN** subroutine can be constructed to calculate the value and partial derivatives through order k, $k \geq 1$, of any function $F: R^n \to R$ which can be sequentially evaluated by means of the special two-variable functions (1) and arbitrary one-variable kth-order continuously differentiable functions such as (2). Systematic rules for constructing general kth-order calculus subroutines for the special functions (1) and (2) are presented in [1].

3. MATRIX FEED: AN EXAMPLE

3.1. *The Purpose of Matrix* FEED

Many practical applications involve multidimensional variables, so the evaluation of partial derivatives evolves into operations on matrices. An efficient algorithm to evaluate partial derivatives of matrix expressions with the same ease as illustrated in Section 2 for the scalar case is thus highly desirable. In this section we will investigate a simple multidimensional example with specific interest in the evaluation of the partial derivatives.

3.2. *Analysis*

3.2.1 Criteria. We will apply the following criteria to the concept of efficient evaluation of derivatives in the following priority:

1. The same accuracy in the numerical calculations as in the one-dimensional case.
2. User-friendliness in the sense of a straightforward, logical, and easily checked representation.
3. Applicability to general arrays, that is, scalars, vectors, and matrices.
4. Low computer processing time.

3.2.2. A Simple Example. In this example we are designing a sequential interpolating filter, and the evaluation of the right-hand side of the resulting differential equations involves the calculation of the following expression:

$$Z(X) = \left\{ H_x^T(X) K [Y - H(X)] \right\}_x, \tag{12}$$

where

$X = (x_1, \ldots, x_n)^T = n \times 1$ vector of independent variables,
$H = (h_1, \ldots, h_m)^T$ maps R^n into R^m,
$H(X) = (h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))^T$,
$Y = (y_1, \ldots, y_m)^T = m \times 1$ observation vector,
$K =$ constant $m \times m$ matrix,
$H_x =$ the $m \times n$ Jacobian matrix

$$\begin{bmatrix} \partial h_1/\partial x_1 & \cdots & \partial h_1/\partial x_n \\ \partial h_2/\partial x_1 & \cdots & \partial h_2/\partial x_n \\ \vdots & & \vdots \\ \partial h_m/\partial x_1 & \cdots & \partial h_m/\partial x_n \end{bmatrix},$$

$H_x^T = n \times m$ transpose of the Jacobian matrix H_x.

We choose a specific example with $n = 3$, $m = 2$, and

$$H: R^3 \to R^2 \quad \text{defined by} \quad H = (h_1, h_2), \tag{13a}$$

where

$$h_1(x_1, x_2, x_3) = x_1 x_2,$$

$$h_2(x_1, x_2, x_3) = x_2 + x_3. \tag{13b}$$

The expression (12) involves matrix multiplications, matrix subtraction, and formation of Jacobian matrices at two levels during the calculation. In the general case, with arbitrarily large values for n and m and with nonlinear expressions for $H = (h_1, \ldots, h_m)$, the evaluation of (12) represents a considerable task.

3.2.3. Operations Involved. The evaluation of expression (12) consists of the following operations:

(1) Form the $m \times n$ Jacobian matrix H_x from the $m \times 1$ column vector H by performing partial differentiation.
(2) Transpose the $m \times n$ matrix H_x to form the $n \times m$ matrix H_x^T.
(3) Postmultiply the $n \times m$ matrix H_x^T by the $m \times m$ matrix K to form an intermediate $n \times m$ matrix $M_3 = H_x^T K$.
(4) Subtract the $m \times 1$ column vector H from the $m \times 1$ column vector Y to form an intermediate column vector $M_4 = Y - H$.
(5) Postmultiply the $n \times m$ matrix M_3 by the $m \times 1$ column vector M_4 to form an $n \times 1$ column vector $M_5 = M_3 M_4 = H_x^T K(Y - H)$.
(6) Form the Jacobian $n \times n$ matrix M_{5x} from the $n \times 1$ column vector M_5 by performing partial differentiation. This is the desired result.

How do we actually implement this in a program that automatically performs the partial differentiations? We have two principal options:

Option 1: Breaking the expression down to its scalar components and applying scalar FEED subroutines to the resulting scalar operations.
Option 2: Designing FEED subroutines that handle matrix operations such as the above mentioned, and applying these in a sequential manner.

Evaluating (12) by choosing option 1 is certainly possible. However, the derivation of the scalar components is time-consuming and especially error prone. Option 1 thus fails to satisfy the criteria chosen above with respect to

user-friendly representation, applicable to general arrays. In the rest of this paper we will investigate option 2. The algorithm obtained will be implemented in FORTRAN, but could as well be implemented in any other language with the proper modifications.

3.2.4. Subproblems. The problem of designing an algorithm involves the solution of the following subproblems:

Subproblem 1: Definition of storage locations for a general FEED array (matrix, vector, and scalar).

Subproblem 2: Representation of an array operation (how the user specifies, for example, a matrix multiplication).

Subproblem 3: Definition of the elements of the general array (as constants or as a function of the independent variables).

Subproblem 4: Communication of input and results with a calling program.

Subproblem 5: Numerical calculation of an array operation.

3.3. *Synthesis*

3.3.1. Defining Storage. In the conventional scalar FEED algorithm described in Section 2, a scalar and its derivatives are stored as a list of the following form:

```
SCALAR(list)
```

where the `list` contains the value of the scalar as its first element, and the partial derivatives with respect to the independent variables as the subsequent elements, in a well-ordered sequence as described in footnote 1 in section 1. A general well-ordered FEED array is then logically stored in the form

```
ARRAY(list,row,column)
```

the order of indices here selected to be apparent in a moment. A column vector is, according to this scheme, stored as

```
COLUMN(list,row,1)
```

In order to maintain the use of conventionally defined FEED subroutines, a scalar will be defined as

```
SCALAR(list)
```

for reasons to be clarified in the further pursuit of the example.

The actual allocation of storage can be performed at run time, or the arrays can be created large enough to hold any matrix the program will

encounter. However, run-time storage allocation is not supported by all programming languages, so for the sake of general application we will here choose to define sufficiently large arrays. In that case, some parts of the arrays may not contain information. Therefore, the dimensions of the vector or matrix saved in the array must be carried along. We choose to define an integer array with two elements, **DIM(2)**, giving the size of the matrix or vector for each array—for example,

```
integer DIMH(2)
DIMH(1) = 2
DIMH(2) = 1
```

stating that the matrix H contains a column vector with 2 rows, that is, H is a 2×1 matrix.

Also, the following variables are in this implementation chosen to be carried along in a **COMMON** statement:

(1) The number **NVAR** of independent variables.
(2) The highest order **NDERIV** of derivatives desired.
(3) The resulting length **NLIST** of the FEED list.

However, other means of transmission can be selected as well.

3.3.2. Representation of an Array Operation. In this part we will deal with criterion 2: The user-friendly representation. Continuing along the lines of the scalar FEED algorithm, we would be satisfied if we could obtain a representation of an array operation of the form

```
call ⟨subroutine-name⟩(⟨matrix1⟩[,⟨matrix2⟩],⟨resulting matrix⟩)
```

—for example,

```
call MULTM(M1,M2,M3)
```

for performing the matrix operation $M_3 = M_1 M_2$.

3.3.3. Definition of the Elements of an Array. To define an element in a column vector H, one possible representation is

```
call ⟨subroutine-name⟩(⟨operand1⟩[,⟨operand2⟩],⟨element⟩)
```

—for example,

```
call MULT(X1,X2,H1)
```

for defining the first element h_1 of the column vector H as $x_1 x_2$, using conventional scalar FEED subroutines.

We are now ready to pursue the example of Section 3.2.2 further. Firstly, we want to define the elements of the column vector H as a function of the

independent variables $(x_1, \ldots, x_n)$ as stored in the column vector X. The function $H = (h_1, h_2)$ is given by

$$h_1 = x_1 x_2,$$

$$h_2 = x_2 + x_3,$$

so we write

```
call MULT(X1,X2,H1)
call ADD(X2,X3,H2)
```

The program has to recognize X1 as the first element of the column vector X, X2 as the second element, and X3 as the third element, and H1 and H2 have to be identified as the elements of the column vector H. The user should not have to be concerned with this identification problem; it should be handled intrinsically in the program. This can be achieved by the equivalence statements

```
real*8 X(list,row,column),X1(list),X2(list),X3(list)
equivalence(X(1,1,1),X1(1)),(X(1,2,1),X2(1)),
           (X(1,3,1),X3(1))
```

for the vector X, and similarly for H. The values of list, row, and column can be defined at run time or defined large enough to hold any matrix the program will encounter.

Let us analyze the exact meaning of this equivalence statement. It states that the following variables share the same memory location, or—in other words—that they are numerically equal:

```
X(1,1,1) = X1(1)
X(2,1,1) = X1(2)
X(3,1,1) = X1(3)
   .
   .
   .
X(list,1,1) = X1(list)
```

and

```
X(1,2,1) = X2(1)
X(2,2,1) = X2(2)
X(3,2,1) = X2(3)
   .
   .
   .
X(list,2,1) = X2(list)
```

and similarly for X3 and H. It is important to remember at this point that the first index is updated before the subsequent indices. When the user then

defines the value of H1 as

```
call MULT(X1,X2,H1)
```

this is interpreted as: Multiply the first element of X by the second element of X and store the result in the first element of H. All these elements are regarded as FEED scalars in the meaning of a FEED list as described above.

3.3.4. Communication with Other Programs. Special subroutines have been written to communicate with a calling program. Assume, for example, that the values of the column vector X are stored in a simple array INPUT(n). According to our general definition of a FEED array, this has to be converted into an array

```
X(list,row,column)
```

This is done by a subroutine LIN, which is the matrix analogue to the scalar FEED subroutine of the same name. It has the following function:

(1) It stores the elements of the column vector INPUT in the elements X(1,1,1), X(1,2,1), X(1,3,1),..., X(1,n,1).

(2) It initializes the FEED list by setting

$$\frac{\partial x_i}{\partial x_i} = 1 \qquad \text{for} \quad i = 1,\ldots,n,$$

$$\frac{\partial x_i}{\partial x_j} = 0 \qquad \text{for} \quad i \neq j, \quad i = 1,\ldots,n, \quad j = 1,\ldots,n.$$

Similar subprograms have been written to convert vectors and matrices that are independent of X into matrix FEED arrays. These serve the same function as LIN, except that all derivatives are initialized to zero for obvious reasons.

3.3.5. Implementation. The full calling program to solve the example problem in Section 3.2.2 would then be:

```
      program EXAMPLE
c-----------------------------------------------------------------------
c     This program evaluates the expression
c
c     [H_X^T(X) * K * (Y - H(X))]_X, where
c
c     H(1) = X(1) * X(2)
c     H(2) = X(2) + X(3)
c
```

```
c     and K and Y are independent of X.
c-----------------------------------------------------------------------
c
c     DEFINITIONS
c-----------------------------------------------------------------------
c
      real*8 X(10,3,3),X1(10),X2(10),X3(10),INPUT(3)
      real*8 H(10,3,3),H1(10),H2(10),H3(10)
      real*8 K(10,3,3),KIN(3,3)
      real*8 Y(10,3,3),YIN(3)
      real*8 M1(10,3,3),M2(10,3,3),M3(10,3,3),
     -       M4(10,3,3),M5(10,3,3),RES(10,3,3)
      integer DIMX(2),DIMH(2),DIMK(2),DIMY(2),
     -        DIMM1(2),DIMM2(2),DIMM3(2),
     -        DIMM4(2),DIMM5(2),DIMRES(2)
c
c     The equivalence statements for X and H.
c
      equivalence
     -   (X(1,1,1),X1(1)),(X(1,2,1),X2(1)),(X(1,3,1),X3(1))
      equivalence
     -   (H(1,1,1),H1(1)),(H(1,2,1),H2(1)),(H(1,3,1),H3(1))

c
c     COMMON statement with information about the FEED list.
c
      common NLIST,NVAR,NDERIV
c
c     Input statements to determine the actual dimension of the arrays
c     X, H, Y and K. These are stored in DIMX, DIMH, DIMY and DIMK,
c     respectively. Furthermore, input statements save the values of X
c     in the array INPUT, the values of Y in the array YIN, and the values of
c     in the array KIN.
c
c     The highest order of derivatives desired is saved in the variable
c     NDERIV. The length NLIST of the FEED list can then be calculated.
c
      NVAR = DIMX(1)
c
c     Initialization of FEED arrays X, Y and K.
c
```

```
      call LIN(INPUT,X)
      call LINC1(YIN,Y)
      call LINC2(KIN,K)
c
c     SCALAR OPERATIONS.
c-----------------------------------------------------------------------
c     Definition of H(1) and H(2)
c
      call MULT(X1,X2,H1)
      call ADD(X2,X3,H2)
c
c     MATRIX OPERATIONS.
c-----------------------------------------------------------------------
c     The matrix operations are now performed by calling matrix FEED
c     subroutines.
c
      call DERM(H,DIMH,M1,DIMM1)
      call TRANM(M1,DIMM1,M2,DIMM2)
      call MULTM(M2,DIMM2,K,DIMK,M3,DIMM3)
      call SUBM(Y,DIMY,H,DIMH,M4,DIMM4)
      call MULTM(M3,DIMM3,M4,DIMM4,M5,DIMM5)
      call DERM(M5,DIMM5,RES,DIMRES)
c
c     OUTPUT
c-----------------------------------------------------------------------
c     The result is now in RES(1,row,column).
c
      do 10 i = 1,DIMRES(1)
      do 1 j = 1,DIMRES(2)
1     write(5,'(''Result row#'',il,''Column#'',il,
     -       '':'',f16.8)') i,j,RES(1,i,j)
10    continue
      end
```

3.3.6. *Implementation of Matrix* FEED *Subroutines*

The matrix FEED subroutines are written once and for all and saved in the FEED library. These subroutines are then linked with the user-written program, as the one described above. The matrix subroutines can be regarded as generalizations of the corresponding scalar FEED subroutines, applied to a matrix with a FEED list appended for each element.

Matrix addition. The matrix FEED addition subroutine has the following implementation:

```
      subroutine ADDM(MTX1,DIM1,MTX2,DIM2,MTXOUT,DIMOUT)
c
c-----------------------------------------------------------------------
c     MATRIX FEED LIBRARY: Adding two matrices.
c-----------------------------------------------------------------------
c
c     This subroutine adds MTX1 (with dimensions DIM1) and MTX2 (with
c     dimensions DIM2) and stores the result in MTXOUT (with dimensions
c     DIMOUT).
c
      real*8 MTX1(10,3,3),MTX2(10,3,3),MTXOUT(10,3,3)
      integer DIM1(2),DIM2(2),DIMOUT(2)
      common NLIST,NVAR,NDERIV
c
c     Check that the dimensions of the operand matrices are correct for
c     a matrix addition.
      if (DIM1(1).ne.DIM2(1).or.DIM1(2).ne.DIM2(2))
     -    write(5,'(''Invalid matrix addition.'')')
c
c     Addition is performed for each element (ir,ic) in the matrix and
c     for each component il of the FEED list.
c
      do 3 ir=1,DIM1(1)
         do 2 ic=1,DIM1(2)
            do 1 il=1,NLIST
1            MTXOUT(il,ir,ic)=MTX1(il,ir,ic)+MTX2(il,ir,ic)
2        continue
3     continue
c
c     Determine the dimensions of the resulting matrix.
c
      DIMOUT(1)=DIM1(1)
      DIMOUT(2)=DIM2(2)
      return
      end
```

Matrix FEED subroutines can be written for any matrix operation, although more complicated implementations may be necessary. To illustrate, a subroutine for matrix multiplication is given below:

Matrix multiplication.

```
      subroutine MULTM(MTX1,DIM1,MTX2,DIM2,MTXOUT,DIMOUT)
c
c-----------------------------------------------------------------------
c     MATRIX FEED LIBRARY: Multiplying two matrices.
c-----------------------------------------------------------------------
c     This subroutine multiplies MTX1 and MTX2 (with dimensions DIM1 and
c     DIM2, respectively) and stores the result in matrix MTXOUT (with
c     dimension DIMOUT).
c
      real*8 MTX1(10,3,3),MTX2(10,3,3),MTXOUT(10,3,3)
      integer DIM1(2),DIM2(2),DIMOUT(2)
      common NLIST,NVAR,NDERIV
c
c     Check that the dimensions of the matrix operands are consistent
c     with a matrix multiplication.
c
      if (DIM1(2).ne.DIM2(1)) then
        write(5,'(''Invalid matrix multiplication.'')')
        goto 999
        endif
c
c     Determine the dimensions of the resulting matrix.
c
      DIMOUT(1)=DIM1(1)
      DIMOUT(2)=DIM2(2)
c
c     Initialize MTXOUT.
c
      do 10 ir=1,DIMOUT(1)
        do 9 ic=1,DIMOUT(2)
          do 100 ipos=1,NLIST
100       MTXOUT(ipos,ir,ic)=0.
c
c     We are now about to determine the values of the element (ir,ic)
c     and its derivatives. According to the rules of matrix
c     multiplication, this is done by forming the sum of products:
c     MTXOUT(1,ir,ic) =   Σ   (MTX1(1,ir,ie) * MTX2(1,ie,ic))
c                       ie=1,#col
```

```
c
c     The derivatives of the element MTXOUT(1,ir,ic) are then calculated
c     and stored at the appropriate location ipos in the FEED list, as
c     MTXOUT(ipos,ir,ic).
c
          do 8 ie=1,DIM1(2)
c
c     Calculate the value of the element:
c     MTXOUT(1,ir,ic) =   Σ   (MTX1(1,ir,ie) * MTX2(1,ie,ic))
c                       ie=1,#col
c
           ipos=1
           MTXOUT(ipos,ir,ic)=MTXOUT(ipos,ir,ic)+
     -                          MTX1(ipos,ir,ie) * MTX2(ipos,ie,ic)
c
c     Calculate the first derivatives of the element. Given the expres-
c     sion for the value of the element above, we have:
c
c       d(MTXOUT(1,ir,ic))/dxi=
c
c             Σ   (MTX1(1,ir,ie) * d(MTX2(1,ie,ic))/dxi+
c          ie=1,#col
c                             d(MTX1(1,ir,ie))/dxi * MTX2(1,ie,ic))
c
c     A well-ordered FEED list has the following storage locations:
c
c     d(MTXOUT(1,ir,ic))/dxi is stored in MTXOUT(ipos,ir,ic);
c     d(MTX1(1,ir,ie))/dxi is stored in MTX1(ipos,ir,ie);
c     d(MTX2(1,ie,ic))/dxi is stored in MTX2(ipos,ie,ic),
c
c     where the counter ipos=1+i.
c
          if(NDERIV.ge.1) then
            do 1 iv=1,NVAR
              ipos=ipos+1
              MTXOUT(ipos,ir,ic)=MTXOUT(ipos,ir,ic)+
     -          MTX1(1,ir,ie) * MTX2(ipos,ie,ic)+
     -          MTX1(ipos,ir,ie) * MTX2(1,ie,ic)
1           continue
          endif
c
c     Calculate the second derivatives of the element:
```

```
c
c     d2(MTXOUT(1,ir,ic))/dxi*dxj=
c
c        Σ    (MTX1(1,ir,ie) * d2(MTX2(1,ie,ic))/dxi*dxj+
c     ie=1, col  d2(MTX1(1,ir,ie))/dxi*dxj * MTX2(1,ie,ic)+
c                d(MTX1(1,ir,ie))/dxi * d(MTX2(1,ie,ic))/dxj+
c                d(MTX1(1,ir,ie))/dxj * d(MTX2(1,ie,ic))/dxi)
c
c     A well-ordered FEED list has the following storage locations:
c
c       d2(MTXOUT(1,ir,ic))/dxi*dxj=MTXOUT(ipos,ir,ic)
c       d2(MTX1(1,ir,ie))/dxi*dxj=MTX1(ipos,ir,ie)
c       d2(MTX2(1,ie,ic))/dxi*dxj=MTX2(ipos,ie,ic)
c       d(MTX1(1,ir,ie))/dxi=MTX1(1+i ,ir,ie)
c       d(MTX2(1,ie,ic))/dxi=MTX2(1+i,ie,ic),
c
c     where the counter ipos runs through all the second derivatives in
c     the FEED list.
c
              if(NDERIV.ge.2) then
                do 3 iv1=1,NVAR
                  do 2 iv2=iv1,NVAR
                  ipos=ipos+1
                  MTXOUT(ipos,ir,ic)=MTXOUT(ipos,ir,ic)+
     -            MTX1(1,ir,ie) * MTX2(ipos,ie,ic)+
     -            MTX1(ipos,ir,ie) * MTX2(1,ie,ic)+
     -            MTX1(1+iv1,ir,ie) * MTX2(1+iv2,ie,ic)+
     -            MTX1(1+iv2,ir,ie) * MTX2(1+iv1,ie,ic)
2                 continue
3               continue
              endif
8     continue
9     continue
10    continue
999   continue
      return
      end
```

Formation of the Jacobian matrix.

```
      subroutine DERM(MTXIN,DIMIN,MTXOUT,DIMOUT)
c
```

```
c     This subroutine forms the Jacobian matrix H_x:
c
c         {dh1/dx1  ...  dh1/dxn}
c         {dh2/dx1  ...  dh2/dxn}
c         {  ...           ...  }
c         {dhm/dx1  ...  dhm/dxn}
c
c     and its derivatives from the column vector (h_1,...,h_m)^T and its
c     derivatives. Recall that the derivatives are stored in the subse-
c     quent elements of the FEED list.
c
      real*8 MTXIN(10,3,3),MTXOUT(10,3,3)
      integer DIMIN(2),DIMOUT(2)
      common NLIST,NVAR,NDERIV
c
c     The subroutine is only applicable to a column vector - this is
c     checked:
c
      if(DIMIN(2).ne.1) then
        write(5,'('' Subroutine DERM can only be called for '',
     -            ''column vectors.'')')
        goto 999
        endif
c
c     Find the dimensions of the Jacobian matrix.
c
      DIMOUT(1) = DIMIN(1)
      DIMOUT(2)=NVAR
c
c     The DO-loop scans through every element of the Jacobian matrix:
c
      do 10 ir=1,DIMOUT(1)
        do 9 ic=1,DIMOUT(2)
c
c     Initialize MTXOUT.
c
          do 1 ipos2=1,NLIST
1           MTXOUT(ipos2,ir,ic)=0.
c
c     Find the values of the element:
c
c     MTXOUT(1,ir,ic)=d(MTXIN(1,ir,1))/dxic,
```

```
c
c     where d(MTXIN(1,ir,1))/dxic is stored in the location
c     MTXIN(1+ic,ir,1).
c
            if(NDERIV.ge.1) then
              ipos2=1
               MTXOUT(ipos2,ir,ic)=MTXIN(1+ic,ir,1)
               endif
c
c     Find the first derivatives of the elements of the Jacobian matrix,
c     which are the second derivatives of the column vector MTXIN, and
c     store them in the FEED list of the Jacobian matrix:
c
c     d(MTXOUT(1,ir,ic))/dxiv=d2(MTXIN(1,ir,1))/dxic*dxiv.
c
c     Recall that the well-ordered sequence of the second derivatives in
c     a FEED list is:
c
c     d2/dx1*dx1, d2/dx1*dx2, d2/dx1*dx3,
c     d2/dx2*dx2, d2/dx2*dx3, d2/dx3*dx3.
c
c     The counters iv1 and iv2 scan through the FEED list of MTXIN. When
c     a second derivative with respect to xic is encountered, this
c     derivative will be placed in the FEED list of the Jacobian matrix.
c
         if(NDERIV.eq.2) then
           ipos1=1+NVAR
           do 8 iv1=1,NVAR
             do 7 iv2=iv1,NVAR
             ipos1=ipos1+1
             if(iv1.eq.ic.or.iv2.eq.ic) then
               ipos2=ipos2+1
               MTXOUT(ipos2,ir,ic)=MTXIN(ipos1,ir,1)
               endif
7            continue
8          continue
           endif
9        continue
10     continue
999    continue
       return
       end
```

3.4. Comments on the Example in Section 3.2.2

The filtering algorithm designed for the specific example in Section 3.2.2 is quite general, and can be dimensioned to handle arbitrary H functions. The user simply has to alter the scalar FEED subroutine calls clearly marked in the implementation of program **EXAMPLE** above. As the example demonstrates, once a matrix FEED library has been constructed, filtering problems involving numerous matrix operations with partial derivatives are easily implemented using the library subroutines. We have thus found the use of FEED matrix subroutines highly efficient and user-friendly in this application.

4. MATRIX FEED: GENERALIZATIONS

4.1. Applications

The matrix FEED algorithm as illustrated in Section 3 is of general use in any multidimensional problem involving evaluation of partial derivatives. We will here compare the use of the matrix FEED algorithm with an algorithm based on derivation of analytical expressions for the matrix expression. [If we combine equations (12) and (13), we can form an analytical expression for Z.] The matrix FEED algorithm has the following advantages:

(1) *User-friendly representation.* Each matrix operation is represented by a subroutine call with a mnemonic name.

(2) *Easy error check.* Although we have not implemented matrix operations by the expressions themselves, error check is nevertheless easily performed. One can simply write the list of operations performed by the subroutine calls, combine them, and compare the resulting expressions with the desired expressions.

(3) *Easily changed expressions.* If changes in the matrix operations are desired, the FEED matrix algorithm allows this to be done simply by changing the subroutine calls. In methods involving analytically derived expressions for partial derivatives, such changes may require considerable effort by the user.

(4) *General applications.* The matrix FEED algorithm handles both vectors and matrices of arbitrary dimensions. The defined storage allocations can easily be increased when needed.

(5) *Few hardware requirements.* The algorithm can be implemented on any microprocessor with sufficient storage to handle the arrays. The subroutines can readily be transformed to BASIC or any other general-application computer language.

Of course, additional experiments are needed to determine more fully the advantages and disadvantages of the matrix FEED approach.

4.2. *Design Considerations*

In due time new matrix operations will be needed as applications of different types are implemented. When planning the addition of new matrix operations to the FEED library, the following issues should be considered. First, matrix operations should be broken down to unary and binary operators to ensure easy representation. Second, the operations should be made as generally applicable as possible, so that the subroutines can be used in future applications. For example, the trace operation fulfills these requirements and can easily be implemented.

4.3. *General Matrix* FEED *Subroutine*

Below is given the general form of a matrix FEED subroutine.

```
      subroutine OPER(MTX1,DIM1[,MTX2,DIM2],MTXOUT,DIMOUT)
c
c-----------------------------------------------------------------------
c     MATRIX FEED LIBRARY: General Matrix FEED Subroutine.
c-----------------------------------------------------------------------
c     This subroutine performs a matrix operation on MTX1 [and MTX2 in
c     the case of a binary operator] with dimensions DIM1 [and DIM2,
c     respectively] and stores the result in matrix MTXOUT (with
c     dimension DIMOUT).
c
      real*8 MTX1 (<list>,<#row>,<#col>),
     -       MTX2 (<list>,<#row>,<#col>),
     -       MTXOUT(<list>,<#row>,<#col>)
      integer DIM1(2),DIM2(2),DIMOUT(2)
      common NLIST,NVAR,NDERIV
c
c     Check that the dimensions of the matrix operands are consis-
c     tent with the matrix operation.
c
      if (<logical expression involving DIM1 [and DIM2]>) then
         write(5,'('' Invalid matrix operation.'')')
         goto 999
         endif
c
c     Determine the dimensions of the resulting matrix.
```

```
c
      DIMOUT(1)=<expression involving DIM1 [and DIM2]>
      DIMOUT(2)=<expression involving DIM1 [and DIM2]>
c
c     This loop identifies each element of the vector/matrix.
c
      do 10 ir=1,DIMOUT(1)
        do 9 ic=1,DIMOUT(2)
c
c     Initialize MTXOUT (this may not be required).
c
            do 100 ipos=1,NLIST
100         MTXOUT(ipso,ir,ic)=0.
c
c     Determine the value of the element (ir,ic).
c
            ipos=1
            MTXOUT(ipos,ir,ic)=OPER (MTX1 [,MTX2])
c
c     Calculate the first derivatives of the element.
c
            if(NDERIV.ge.1) then
            do 1 iv=1,NVAR
              ipos=ipos+1
              MTXOUT(ipos,ir,ic)=dOPER/dxiv (MTX1 [,MTX2])
1             continue
            endif
c
c     Calculate the second derivatives of the element.
c
            if(NDERIV.ge.2) then
              do 3 iv1=1,NVAR
                do 2 iv2=iv1,NVAR
                ipos=ipos+1
                MTXOUT(ipos, ir,ic)=
                 d2OPER/dxiv1*dxiv2 (MTX1 [,MTX2])
2               continue
3             continue
            endif
c
c     Similar nth-order loops involving nth-order derivatives
c     when required.
```

```
c
9               continue
10              continue
999             continue
                return
                end
```

5. NUMERICAL TESTS

For the example given in Section 3.2.2, a program implementing two algorithms was constructed:

First algorithm. Matrix FEED subroutine calls for the evaluation of (12), with H given by (13).

Second algorithm. Evaluation of the analytical expression formed by combining (13) with (12) and analytically performing the partial differentiations involved.

The results from these two algorithms were compared for several values of the independent variables. The two algorithms gave results which were in agreement to sixteen decimal places.

The detailed numerical example is available from the authors upon request.

6. DISCUSSION

In the original presentation [1] of the FEED algorithm, three types of scalar calculus subroutines were included in the FEED library: subroutine **LIN** for independent variables, subroutines such as **LOGG** for real-valued functions of one variable, and subroutines such as **ADD** for real-valued functions of two variables. In a subsequent paper [2] a selection subroutine was added to the **FEED** library to facilitate the automatic differentiation of real-valued functions defined in terms of the derivatives of other functions.

In the present paper it is shown that the automatic differentiation of expressions involving nested matrix and derivative operations can be systematically handled by introducing *matrix* calculus subroutines into the FEED library. These matrix FEED subroutines have been thoroughly tested, and have proved to be a very useful tool of general application..

REFERENCES

1 R. Kalaba, L. Tesfatsion, and J.-L. Wang, A finite algorithm for the exact evaluation of higher-order partial derivatives of functions of many variables, *J. Math. Anal. Appl.* 12:181–191 (1983).

2 R. Kalaba and L. Tesfatsion, Automatic differentiation of functions of derivatives, *Comput. Math. Appl.* 12A:1091–1103 (1986).

3 R. Wengert, A simple automatic derivative evaluation program, *Comm. ACM* 7:463–464 (1964).