An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order

RICHARD D. NEIDINGER
Davidson College

For any typical multivariable expression f, point $\vec{\mathbf{a}}$ in the domain of f, and positive integer maxorder, this method produces the numerical values of all partial derivatives at $\vec{\mathbf{a}}$ up through order maxorder. By the technique known as automatic differentiation, theoretically exact results are obtained using numerical (as opposed to symbolic) manipulation. The key ideas are a hyperpyramid data structure and a generalized Leibniz's rule. Any expression in n variables corresponds to a hyperpyramid array, in n-dimensional space, containing the numerical values of all unique partial derivatives (not wasting space on different permutations of derivatives). The arrays for simple expressions are combined by hyperpyramid operators to form the arrays for more complicated expressions. These operators are facilitated by a generalized Leibniz's rule which, given a product of multivariable functions, produces any partial derivative by forming the minimum number of products (between two lower partials) together with a product of binomial coefficients. The algorithms are described in abstract pseudo-code. A section on implementation shows how these ideas can be converted into practical and efficient programs in a typical computing environment. For any specific problem, only the expression itself would require recoding.

Categories and Subject Descriptors: G.1 [**Mathematics of Computing**]: Numerical Analysis; G.1.0 [**Numerical Analysis**]: General—*numerical algorithms*; G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis*, *efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Automatic differentiation, high order derivatives, partial derivatives, data structure

1. INTRODUCTION

Automatic differentiation is distinguished as a numerical, yet theoretically exact, method (see Rall [12] for expository introduction). We seek an automatic differentiation algorithm such that given any positive integers n and maxorder, any typical expression f in n variables, and any point $\vec{\mathbf{a}}$ in the domain of f, the algorithm produces the numerical values of all unique partial derivatives at $\vec{\mathbf{a}}$ up through order maxorder. To change n or maxorder

Author's address: Department of Mathematics, PO Box 1719, Davidson College, Davidson, NC 28036.

will involve simply changing constants; only the expression f will require recoding.

The numerical algorithm of Wexler [16] accomplishes this task but distinguishes different permutations of partial derivative operators and performs repetitive calculations. The data structure used Wexler [16] is a tree, and tree operators are defined by explicit recursion on levels of the tree. In contrast, our hyperpyramid data structure stores only unique partial derivatives (regardless of permutation), and hyperpyramid operators sequentially perform the minimum number of multiplications. For example, suppose $h = (xy)\exp(x + y)$ and the numerical values of all partials of $u = xy$ and $v = \exp(x + y)$ have been computed. Consider how (the numerical value of) $h_{xxxyy} = (xy + 2x + 3y + 6)\exp(x + y)$ is formed. In our method, the last term comes from the two multiplications in $6u_{xy}v_{xxy}$, whereas, in Wexler [16] (and Kalaba et al. [8]), this same term comes from multiplying u_{xy} and v_{xxy} (or equivalent permutations) together six times and adding. In this sense, our method is efficient. Ideas for even more efficient methods are discussed in Section 6. While the space and time requirements are still significant, any automatic differentiation method usually beats symbolic algebra systems by orders of magnitude [4, 11].

The coefficient in the example, $6u_{xy}v_{xxy}$, is given by a generalized Leibniz's rule for multivariable functions. The classical Leibniz's rule (nth order product rule for single variable calculus) is applied to automatic differentiation by Kalaba et al. [8] and Neidinger [11]. Moore [10] and Rall [13] use an elegant (and simpler) version of this rule for single variable Taylor series coefficients (see Section 6). Neidinger [11] shows how Leibniz's rule is encapsulated in an operator "BDOT" and then this operation is used in all derivations. Insight from preparing [11] led to the method described here.

It is important to distinguish that our program works for any variable (i.e., simply change the constant) order, as opposed to the claim that for any order, the method produces a program specific to that order. Many papers fulfill this latter claim, including Wengert's original [15]. More recently, Kalaba et al. [8] and Jackson and McCormick [6] present algorithms which generate arbitrary order differentiation rules. These resulting rules are used to manipulate numerical arrays (as opposed to symbolic formulas) of partial derivatives, i.e., automatic differentiation. Nevertheless, the algorithms which generate these rules are not in pseudo-code form; Kabala et al. [8] uses unique partitions of index sets into a variable number of subsets; Jackson and McCormick [6, p. 72] depend on the "form" of scalar coefficients. Thus Kabala et al. [8] and Jackson and McCormick [6] fulfill the latter claim above and not the objective of this paper. It should be noted, however, that specific applications use a fixed order of derivatives (often, the gradient and Hessian are sufficient) and the very special form of the derivatives given by Jackson and McCormick [6] can be beneficially exploited in applications.

2. NOTATION AND DATA STRUCTURE

Fix n and maxorder as arbitrary constants. We restrict attention to real-valued functions on (an open domain in) $\mathbb{R}^n$ that are C^∞ (i.e., all partial

derivatives exist and are continuous on the domain). Hence the mixed partials agree (i.e., $f_{xy} = f_{yx}$). In particular, we are concerned with expressions in n variables obtained by algebraic (including composition) means from polynomial, logarithmic, exponential, trigonometric and inverse trigonometric functions.

To condense the notation for partial derivatives of $f(x_1, x_2, \ldots, x_n)$, define an index set of n-tuples of nonnegative integers, $J = \{(j_1, j_2, \ldots, j_n) \mid j_1 + j_2 + \cdots + j_n \leq \text{maxorder}\}$. For each $\vec{\mathbf{j}}$ in J, let

$$D_{\vec{\mathbf{j}}} f = \left(\frac{\partial}{\partial x_1}\right)^{j_1} \left(\frac{\partial}{\partial x_2}\right)^{j_2} \cdots \left(\frac{\partial}{\partial x_n}\right)^{j_n} f.$$

Define a partial order on J by $\vec{\mathbf{j}} \leq \vec{\mathbf{k}}$ if and only if $j_i \leq k_i$ for all $i = 1, 2, \ldots, n$. Thus $D_{\vec{\mathbf{k}}} f$ could be obtained from $D_{\vec{\mathbf{j}}} f$ by taking further derivatives. Vectors in J are added or subtracted coordinatewise, so that $D_{\vec{\mathbf{j}}} D_{\vec{\mathbf{m}}} f = D_{\vec{\mathbf{j}}+\vec{\mathbf{m}}} f$.

To evaluate partial derivatives of expressions at $\vec{\mathbf{a}}$, every expression $h(\vec{\mathbf{x}})$ is associated with a numerical array H, indexed by the set J, where entry $H[\vec{\mathbf{j}}] = (D_{\vec{\mathbf{j}}} h)(\vec{\mathbf{a}})$. For example, if $n = 2$, using variables x and y, and maxorder = 2, then

$$H = \begin{bmatrix} h(\vec{\mathbf{a}}) & h_y(\vec{\mathbf{a}}) & h_{yy}(\vec{\mathbf{a}}) \\ h_x(\vec{\mathbf{a}}) & h_{xy}(\vec{\mathbf{a}}) & \\ h_{xx}(\vec{\mathbf{a}}) & & \end{bmatrix}.$$

If $n = 3$, H is a pyramid (a tetrahedron) of nodes in 3-dimensional space. In general, H is a hyperpyramid of nodes in n-dimensional space, which we refer to as a PYRAMID. In the text, lower case is used for scalars (real or integer), **boldface** is used for vectors indexed by $1 \ldots n$ (in $\mathbb{R}^n$ or J), and UPPER CASE is used for PYRAMIDs.

The goal is to calculate H when given the expression for $h(\vec{\mathbf{x}})$ and $\vec{\mathbf{a}} = (a_1, a_2, \ldots, a_n)$. For the simplest expressions, the associated PYRAMIDs are known. Specifically, for the constant s, we associate the PYRAMID CS where CS[**0**] = s and all other entries are zero. For each variable x_i, we associated the PYRAMID XI which contains all zeros except that XI[**0**] = a_i and XI[$\vec{\mathbf{e}}_i$] = 1, where $\vec{\mathbf{e}}_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ with the 1 in the ith position. More complicated PYRAMIDs can be built from these basic components.

Suppose that $h(\vec{\mathbf{x}})$ is given as an arithmetic operation combining $u(\vec{\mathbf{x}})$ and $v(\vec{\mathbf{x}})$, and that PYRAMIDs U and V are known (i.e., $U[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} u(\vec{\mathbf{a}})$ and $V[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} v(\vec{\mathbf{a}})$ for all $\vec{\mathbf{j}} \in J$). Then, the central idea of automatic differentiation is to define a corresponding PYRAMID operation on U and V which yields H. This operation is determined by a differentiation rule! For example, if $h(\vec{\mathbf{x}}) = u(\vec{\mathbf{x}}) + v(\vec{\mathbf{x}})$, then $H = U + V$, where addition of arrays is assumed to be coordinatewise (as in matrix addition). Similarly, if $h(\vec{\mathbf{x}}) = r \cdot u(\vec{\mathbf{x}})$, then $H = rU$, where a scalar multiplication applies to each entry (as in scalar multiplication of a matrix).

The plan is to translate any expression $f(\vec{\mathbf{x}})$, into the corresponding PYRAMID operators ultimately applied to some XI's and CS's. For example, suppose $n = 2$ and maxorder = 2. If $f(x_1, x_2) = 7x_1 - 4x_2 + 5$ and $\vec{\mathbf{a}} = (3, -2)$, then

$$\begin{aligned} F &= 7 \cdot X1 - 4 \cdot X2 + C5 \\ &= 7\begin{bmatrix} 3 & 0 & 0 \\ 1 & 0 & \\ 0 & & \end{bmatrix} - 4\begin{bmatrix} -2 & 1 & 0 \\ 0 & 0 & \\ 0 & & \end{bmatrix} + \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & \\ 0 & & \end{bmatrix} \\ &= \begin{bmatrix} 34 & -4 & 0 \\ 7 & 0 & \\ 0 & & \end{bmatrix} \end{aligned}$$

containing the values, at $\vec{\mathbf{a}} = (3, -2)$, of f and all its partials up through order maxorder.

Other PYRAMID operations, corresponding to more complicated real operations, will be established in Section 3, the heart of this paper. We will define an operator TIMES such that if $h(\vec{\mathbf{x}}) = u(\vec{\mathbf{x}}) \cdot v(\vec{\mathbf{x}})$ and U and V are known, then $H = U$ TIMES V. Similarly, DIV will be defined. For transcendental functions, we define PYRAMID operators: EXP, LN, SQRT, ARCTAN, ARCSIN, SIN, COS, TAN, etc. These work as in the following example: if $h(\vec{\mathbf{x}}) = \sin(u(\vec{\mathbf{x}}))$, then $H = \mathrm{SIN}(U)$ where, for all $\vec{\mathbf{j}} \in J$, input U satisfies $U[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} u(\vec{\mathbf{a}})$ and output H satisfies $H[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} h(\vec{\mathbf{a}})$. Note that PYRAMID operators are distinguished from the corresponding real operators by upper or lower case, respectively. Any multivariable expression can be translated into the corresponding PYRAMID operations ultimately applied to some XI's and CS's. The result will be the exact (assuming no roundoff) values of all partial derivatives up through any desired maxorder.

3. THE PYRAMID OPERATOR ALGORITHMS

The PYRAMID operator definitions depend heavily on the rule for partial derivatives of a product of functions. This is classical analysis and can be proved by straight-forward induction on the order of the partial derivative operator. The expressions in parentheses are binomial coefficients.

THEOREM (Generalized Leibniz's Rule). *If u and v are real-valued C^∞ functions on (an open domain in) $\mathbb{R}^n$ and $\vec{\mathbf{m}} \in J$, then*

$$D_{\vec{\mathbf{m}}}(uv) = \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{m}}} \binom{m_1}{j_1}\binom{m_2}{j_2} \cdots \binom{m_n}{j_n} \cdot D_{\vec{\mathbf{j}}} u \cdot D_{\vec{\mathbf{m}}-\vec{\mathbf{j}}} v$$

where the sum is taken over all $\vec{\mathbf{j}} \in J$ such that $\vec{\mathbf{0}} \le \vec{\mathbf{j}} \le \vec{\mathbf{m}}$ (coordinatewise).

We can interpret this rule as a real-valued operation on PYRAMIDs. First, define

$$\mathrm{coef}(\vec{\mathbf{m}}, \vec{\mathbf{j}}) = \binom{m_1}{j_1}\binom{m_2}{j_2} \cdots \binom{m_n}{j_n}.$$

Let P and Q be PYRAMIDs and $\vec{\mathbf{m}} \le \vec{\mathbf{k}}$ be elements of J (consider $\vec{\mathbf{m}} = \vec{\mathbf{k}}$ for now). Define

$$\mathrm{bdot}(P, \vec{\mathbf{m}}, Q, \vec{\mathbf{k}}) = \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{m}}} \mathrm{coef}(\vec{\mathbf{m}}, \vec{\mathbf{j}}) * P[\vec{\mathbf{j}}] * Q[\vec{\mathbf{k}} - \vec{\mathbf{j}}].$$

This can be visualized as an operation on n-dimensional subarrays. Let PBOX = $\{P[\mathbf{j}] \mid \mathbf{0} \le \mathbf{j} \le \mathbf{m}\}$, a "hyper-box" of entries between origin $\mathbf{0}$ and opposite corner $\mathbf{m}$. Let QBOX = $\{Q[\mathbf{k} - \mathbf{j}] \mid \mathbf{0} \le \mathbf{j} \le \mathbf{m}\}$, the hyper-box of Q between $\mathbf{k} - \mathbf{m}$ and $\mathbf{k}$, with entries reversed in order along every axis. Finally let CBOX = $\{\mathrm{coef}(\mathbf{m}, \mathbf{j}) \mid \mathbf{0} \le \mathbf{j} \le \mathbf{m}\}$, a hyper-box of entries with a row of Pascal's Triangle along each edge and every other entry is the product of the corresponding edge entries. (In APL terminology, COBX is an outer product of n rows (lengths $m_1 + 1, \ldots, m_n + 1$) of Pascal's Triangle.) To find $\mathrm{bdot}(P, \mathbf{m}, Q, \mathbf{k})$, take the coordinate-wise product of PBOX, QBOX, and CBOX and then sum all the entries; the name bdot reflects that this is somewhat like a dot product of PBOX and QBOX with these binomial coefficients thrown in.

All of our derivations will be based on the following

COROLLARY. *Suppose h, u, and v are real-valued C^∞ functions on (an open domain in) $\mathbb{R}^n$ such that $D_{\vec{\mathbf{e}}} h = u \cdot D_{\vec{\mathbf{e}}} v$, for some $\vec{\mathbf{e}} \in J$. If $\mathbf{k} \in J$ such that $\vec{\mathbf{e}} \le \mathbf{k}$ and U and V are* PYRAMIDs *such that $U[\mathbf{j}] = D_{\vec{\mathbf{j}}} u(\vec{\mathbf{a}})$ for all $\mathbf{0} \le \mathbf{j} \le \mathbf{k} - \vec{\mathbf{e}}$ and $V[\mathbf{j}] = D_{\vec{\mathbf{j}}} v(\vec{\mathbf{a}})$ for all $\vec{\mathbf{e}} \le \mathbf{j} \le \mathbf{k}$, then $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \mathrm{bdot}(U, \mathbf{k} - \vec{\mathbf{e}}, V, \mathbf{k})$.*

PROOF. Let $h' = D_{\vec{\mathbf{e}}} h$ and $v' = D_{\vec{\mathbf{e}}} v$. Let $\vec{\mathbf{m}} = \vec{\mathbf{k}} - \vec{\mathbf{e}}$. By the Generalized Leibniz's Rule on $h' = u \cdot v'$,

$$D_{\vec{\mathbf{m}}} h' = \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{m}}} \binom{m_1}{j_1} \binom{m_2}{j_2} \cdots \binom{m_n}{j_n} \cdot D_{\vec{\mathbf{j}}} u \cdot D_{\vec{\mathbf{m}}-\vec{\mathbf{j}}} v'.$$

Therefore,

$$D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{m}}} \mathrm{coef}(\vec{\mathbf{m}}, \vec{\mathbf{j}}) \cdot D_{\vec{\mathbf{j}}} u(\vec{\mathbf{a}}) \cdot D_{\vec{\mathbf{k}}-\vec{\mathbf{j}}} v(\vec{\mathbf{a}}) = \mathrm{bdot}(U, \vec{\mathbf{m}}, V, \vec{\mathbf{k}}). \quad \square$$

This corollary is used to derive all of the PYRAMID operators. In each of the following derivations, we consider $h(\vec{\mathbf{x}})$ to be an expression composed of one operator or function applied to the expression $u(\vec{\mathbf{x}})$ (and possibly $v(\vec{\mathbf{x}})$) and seek a recursive formula for $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}})$ for $\mathbf{k} \ne \mathbf{0}$. This leads to an algorithm which builds H iteratively.

In each derivation, assume that $U[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} u(\vec{\mathbf{a}})$ and $V[\vec{\mathbf{j}}] = D_{\vec{\mathbf{j}}} v(\vec{\mathbf{a}})$ for all $\vec{\mathbf{j}} \in J$. We choose and arbitrary $\mathbf{k} \in J$ and seek the value of $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}})$, for assignment to $H[\mathbf{k}]$. For derivation of recursive formulas, assume that $H[\mathbf{j}] = D_{\vec{\mathbf{j}}} h(\vec{\mathbf{a}})$ has been calculated for all $\mathbf{j} \le \mathbf{k}$, $\mathbf{j} \ne \mathbf{k}$ and that $H[\mathbf{k}] = 0$. It is helpful to let H^+ denote the PYRAMID such that $H^+[\mathbf{j}] = D_{\vec{\mathbf{j}}} h(\vec{\mathbf{a}})$ for all $\mathbf{j} \le \mathbf{k}$, including $\mathbf{k}$.

If $h(\vec{\mathbf{x}}) = u(\vec{\mathbf{x}}) \cdot v(\vec{\mathbf{x}})$, then $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \text{bdot}(U, \vec{\mathbf{k}}, V, \vec{\mathbf{k}})$, by the Corollary with $\vec{\mathbf{e}} = \vec{\mathbf{0}}$. Thus the PYRAMID operator TIMES is defined as follows.

```
FUNCTION H ← U TIMES V
BEGIN
  FOR each k ∈ J DO H[k] ← bdot(U, k, V, k)
END
```

We begin a new derivation for $h(\vec{\mathbf{x}}) = u(\vec{\mathbf{x}})/v(\vec{\mathbf{x}})$ by observing that $u(\vec{\mathbf{x}}) = h(\vec{\mathbf{x}}) \cdot v(\vec{\mathbf{x}})$. By the Corollary (with $\vec{\mathbf{e}} = \vec{\mathbf{0}}$), $D_{\vec{\mathbf{k}}} u(\vec{\mathbf{a}}) = \text{bdot}(H^+, \vec{\mathbf{k}}, V, \vec{\mathbf{k}})$, where the last term in the sum from bdot is $1 * D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) * V[\vec{\mathbf{0}}]$. Using H, with $H[\vec{\mathbf{k}}] = 0$, we see that

$$D_{\vec{\mathbf{k}}} u(\vec{\mathbf{a}}) = \text{bdot}(H, \vec{\mathbf{k}}, V, \vec{\mathbf{k}}) + D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) * V[\vec{\mathbf{0}}].$$

Thus

$$D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \big(U[\vec{\mathbf{k}}] - \text{bdot}(H, \vec{\mathbf{k}}, V, \vec{\mathbf{k}})\big)/V[\vec{\mathbf{0}}].$$

To use this recursive formula to iteratively build PYRAMID H, we define the loop "FOR each $\vec{\mathbf{k}} \in J$ with respect to partial order." This means that $\vec{\mathbf{k}}$ is incremented through the index set J so that, on iteration $\vec{\mathbf{k}}$, the loop has been completed for all index vectors $\leq \vec{\mathbf{k}}$ (coordinatewise), except $\vec{\mathbf{k}}$ itself. Though this may sound tricky, almost any natural ordering of J starting with $\vec{\mathbf{0}} = (0, 0, \ldots, 0)$ will have this property, including nested loops for each coordinate. (Specifically, J may be ordered by the value of $\vec{\mathbf{k}}$ when this vector is interpreted as a representation in base (maxorder + 1).)

```
FUNCTION H ← U DIV V
BEGIN
  FOR each k ∈ J with respect to partial order DO BEGIN
    H[k] ← 0
    H[k] ← (U[k] − bdot(H, k, V, k)) / V[0]
  END
END
```

Given $\vec{\mathbf{k}}$, the following derivations apply the Corollary with an order-one vector $\vec{\mathbf{e}} \leq \vec{\mathbf{k}}$. (An order-one vector in J has some entry equal to one and all other entries equal to zero.)

Suppose $h(\vec{\mathbf{x}}) = \exp(u(\vec{\mathbf{x}}))$, then $D_{\vec{\mathbf{e}}} h = h \cdot D_{\vec{\mathbf{e}}} u$ for any order-one vector $\vec{\mathbf{e}}$. Given $\vec{\mathbf{k}} \neq \vec{\mathbf{0}}$, let $\vec{\mathbf{e}}$ be an order-one vector $\leq \vec{\mathbf{k}}$. By the Corollary, $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}})$. This formula is straightforward, since the summation only uses entries in H through $\vec{\mathbf{k}} - \vec{\mathbf{e}}$.

```
FUNCTION H ← EXP(U)
BEGIN
  H[0] ← exp(U[0])
  FOR each k ∈ J, k ≠ 0, with respect to partial order DO BEGIN
    e ← an order-one vector ≤ k
    H[k] ← bdot(H, k − e, U, k)
  END
END
```

If $h(\vec{\mathbf{x}}) = \ln(u(\vec{\mathbf{x}}))$, then $D_{\vec{\mathbf{e}}} h = D_{\vec{\mathbf{e}}} u / u$ or $D_{\vec{\mathbf{e}}} u = u \cdot D_{\vec{\mathbf{e}}} h$ for any order-one

vector $\vec{\mathbf{e}}$. Given $\vec{\mathbf{k}} \neq \vec{\mathbf{0}}$, let $\vec{\mathbf{e}}$ be an order-one vector $\leq \vec{\mathbf{k}}$. By the Corollary, $D_{\vec{\mathbf{k}}} u(\vec{\mathbf{a}}) = \text{bdot}(U, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H^+, \vec{\mathbf{k}})$, where the first term in the summation is $1 * U[\vec{\mathbf{0}}] * D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}})$. With $H[\vec{\mathbf{k}}] = 0$,

$$D_{\vec{\mathbf{k}}} u(\vec{\mathbf{a}}) = U[\vec{\mathbf{0}}] * D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) + \text{bdot}(U, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}).$$

Therefore, $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = (U[\vec{\mathbf{k}}] - \text{bdot}(U, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}))/U[\vec{\mathbf{0}}]$.

```
FUNCTION H ← LN(U)
BEGIN
  H[0] ← ln(U[0])
  FOR each k ∈ J, k ≠ 0, with respect to partial order DO BEGIN
    e ← an order-one vector ≤ k
    H[k] ← 0
    H[k] ← (U[k] − bdot(U, k − e, H, k)) / U[0]
  END
END
```

If $h(\vec{\mathbf{x}}) = \sqrt{u(\vec{\mathbf{x}})}$, then $D_{\vec{\mathbf{e}}} h = 1/(2h) \cdot D_{\vec{\mathbf{e}}} u$ or $D_{\vec{\mathbf{e}}} u = 2h \cdot D_{\vec{\mathbf{e}}} h$ for any order-one vector $\vec{\mathbf{e}}$. Given $\vec{\mathbf{k}} \neq \vec{\mathbf{0}}$, let $\vec{\mathbf{e}}$ be an order-one vector $\leq \vec{\mathbf{k}}$. By the Corollary,

$$\begin{aligned} D_{\vec{\mathbf{k}}} u(\vec{\mathbf{a}}) &= \text{bdot}(2H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H^+, \vec{\mathbf{k}}) = 2 * \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H^+, \vec{\mathbf{k}}) \\ &= 2\{H[\vec{\mathbf{0}}] * D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) + \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}})\}. \end{aligned}$$

Thus $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = (0.5 * U[\vec{\mathbf{k}}] - \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}))/H[\vec{\mathbf{0}}]$. Define

$$H \leftarrow \text{SQRT}(U) \quad \text{by } H[\vec{\mathbf{k}}] \leftarrow (0.5 * U[\vec{\mathbf{k}}] - \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}))/H[\vec{\mathbf{0}}]$$

using the same outline as LN.

If $h(\vec{\mathbf{x}}) = \arctan(u(\vec{\mathbf{x}}))$, then $D_{\vec{\mathbf{e}}} h = v \cdot D_{\vec{\mathbf{e}}} u$ where $v = 1/(1 + u^2)$ and $\vec{\mathbf{e}}$ is any order-one vector. Given $\vec{\mathbf{k}} \neq \vec{\mathbf{0}}$, let $\vec{\mathbf{e}}$ be an order-one vector $\leq \vec{\mathbf{k}}$. Thus $D_{\vec{\mathbf{k}}} h(\vec{\mathbf{a}}) = \text{bdot}(V, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}})$. Since U is known, V can be calculated by automatic differentiation. The PYRAMID-valued function C(s) returns the simple PYRAMID associated with the constant s, as described in Section 2.

```
FUNCTION H ← ARCTAN(U)
BEGIN
  H[0] ← arctan(U[0])
  V ← C(1) DIV (C(1) + U TIMES U)
  FOR each k ∈ J, k ≠ 0, with respect to partial order DO BEGIN

    e ← an order-one vector ≤ k
    H[k] ← bdot(V, k − e, U, k)
  END
END
```

Similarly, ARCSIN is defined by the assignments

$$H[\vec{\mathbf{0}}] \leftarrow \arcsin(U[\vec{\mathbf{0}}])$$

$$V \leftarrow C(1) \text{ DIV SQRT } (C(1) - U \text{ TIMES } U)$$

in place of the corresponding lines in ARCTAN.

If $h(\vec{\mathbf{x}}) = \sin(u(\vec{\mathbf{x}}))$ and $g(\vec{\mathbf{x}}) = \cos(u(\vec{\mathbf{x}}))$, then $D_{\vec{\mathbf{e}}}h = g \cdot D_{\vec{\mathbf{e}}}u$ and $D_{\vec{\mathbf{e}}}g = -h \cdot D_{\vec{\mathbf{e}}}u$ for any order-one vector $\vec{\mathbf{e}}$. Given $\mathbf{k} \neq \mathbf{0}$, let $\vec{\mathbf{e}}$ be an order-one vector $\leq \vec{\mathbf{k}}$. Assuming $H[\mathbf{j}] = D_{\vec{\mathbf{j}}}h(\vec{\mathbf{a}})$ and $G[\mathbf{j}] = D_{\vec{\mathbf{j}}}g(\vec{\mathbf{a}})$ for all $\mathbf{0} \leq \mathbf{j} \leq \mathbf{k} - \vec{\mathbf{e}}$, the Corollary yields:

$$D_{\vec{\mathbf{k}}}h(\vec{\mathbf{a}}) = \text{bdot}(G, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}}) \quad \text{and}$$

$$D_{\vec{\mathbf{k}}}g(\vec{\mathbf{a}}) = \text{bdot}(-H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}}) = -\text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}})$$

We define a procedure with input U and output $SINU = H$ and $COSU = G$.

```
PROCEDURE SINCOS(U, SINU, COSU)
BEGIN
  SINU[0] ← sin(U[0])
  COSU[0] ← cos(U[0])
  FOR each k ∈ J, k ≠ 0, with respect to partial order DO BEGIN
    e ← an order-one vector ≤ k
    SINU[k] ← bdot(COSU, k − e, U, k)
    COSU[k] ← −bdot(SINU, k − e, U, k)
  END
END
```

For use in forming larger expressions, it is important to have PYRAMID operators corresponding to sin, cos, and tan.

```
FUNCTION H ← SIN(U)
BEGIN
  SINCOS(U, SINU, COSU)
  H ← SINU
END
```

Similarly define:

$H \leftarrow \text{COS}(U)$ by $\text{SINCOS}(U, SINU, COSU)$; $H \leftarrow COSU$

$H \leftarrow \text{TAN}(U)$ by $\text{SINCOS}(U, SINU, COSU)$; $H \leftarrow SINU$ DIV $COSU$

$H \leftarrow \text{COT}(U)$ by $\text{SINCOS}(U, SINU, COSU)$; $H \leftarrow COSU$ DIV $SINU$

All other typical operators can be defined in terms of preceding PYRAMID operators. For example, define:

$H \leftarrow \text{RECIP}(U)$ by $H \leftarrow C(1)$ DIV U

$H \leftarrow \text{SEC}(U)$ by $H \leftarrow \text{RECIP}(\text{COS}(U))$

$H \leftarrow \text{CSC}(U)$ by $H \leftarrow \text{RECIP}(\text{SIN}(U))$

$H \leftarrow U$ POWER r by $H \leftarrow \text{EXP}(r * \text{LN}(U))$

$H \leftarrow \text{SINH}(U)$ by $H \leftarrow 0.5 * (\text{EXP}(U) - \text{EXP}(-U))$

Indeed, any typical expression in any number of variables can be automatically differentiated by translating the expression into the corresponding PYRAMID operators.

Operator efficiency may be improved for the above "composite" operators, i.e., those that call previously defined operators. Instead of using calls to

operators such as DIV and EXP, use a direct rule for $H[\vec{\mathbf{k}}]$. Ideally, one seeks a rule that results in fewer calls to bdot. For example, consider an algorithm for TAN based on an idea supplied by Rall. Let $h(\vec{\mathbf{x}}) = \tan(u(\vec{\mathbf{x}}))$ and define $v(\vec{\mathbf{x}}) = \sec^2(u(\vec{\mathbf{x}})) = 1 + (h(\vec{\mathbf{x}}))^2$. Then $D_{\vec{\mathbf{e}}}h = v \cdot D_{\vec{\mathbf{e}}}u$ and $D_{\vec{\mathbf{e}}}v = 2h \cdot D_{\vec{\mathbf{e}}}h$, so that,

$$H[\vec{\mathbf{k}}] \leftarrow \text{bdot}(V, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}}) \quad \text{and}$$
$$V[\vec{\mathbf{k}}] \leftarrow 2 * \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}).$$

The cost of iterating these definitions is roughly equivalent to SINCOS and eliminates the cost of the third operation DIV. For a second example, consider a less successful direct algorithm for POWER. Let $h(\mathbf{x}) = (u(\vec{\mathbf{x}}))^r$. Although the first derivative rule is simple, an arbitrary order recursive rule is not. Since $u \cdot D_{\vec{\mathbf{e}}}h = r \cdot h \cdot D_{\mathbf{e}}u$, $\text{bdot}(U, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H^+, \vec{\mathbf{k}}) = r * \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}})$. Thus, $H[\vec{\mathbf{k}}] = (r * \text{bdot}(H, \vec{\mathbf{k}} - \vec{\mathbf{e}}, U, \vec{\mathbf{k}}) - \text{bdot}(U, \vec{\mathbf{k}} - \vec{\mathbf{e}}, H, \vec{\mathbf{k}}))/U[\vec{\mathbf{0}}]$. There are two calls to bdot for every PYRAMID entry; roughly equivalent to the cost of $\text{EXP}(r * \text{LN}(U))$. Some efficiency may be gained by even lower-level programming. In the POWER example, the summations performed by the bdot calls can be combined and simplified. The result is

$$H[\vec{\mathbf{k}}] = \frac{1}{U[\vec{\mathbf{0}}]k_i} \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{k}}} \left(rk_i - j_i(r+1)\right) * \text{coef}(\vec{\mathbf{k}}, \vec{\mathbf{j}}) * H[\vec{\mathbf{j}}] * U[\vec{\mathbf{k}} - \vec{\mathbf{j}}]$$

where k_i is a nonzero entry of $\vec{\mathbf{k}}$ (corresponding to the nonzero entry of $\vec{\mathbf{e}}$).

4. IMPLEMENTATION

The author has implemented the algorithms of the previous section in standard APL and in VAX Pascal (allowing array-valued functions), however several hurdles separate the abstract ideas and the actual code. After considering the key to efficient implementation of this method, we focus on implementation of the PYRAMID data structure and the index set J. This is followed by a few remarks about the largely cosmetic problems in forming the expressions to be automatically differentiated by this method.

The efficiency of this package rests squarely in the function bdot, since bdot is a loop which is nested inside every PYRAMID operator loop. Implementing $\text{coef}(\vec{\mathbf{m}}, \vec{\mathbf{j}})$ as a direct calculation is very expensive, and unnecessarily repeats the same calculations for each PYRAMID operator. All of these coefficients can be calculated once (after n and maxorder are specified) and stored in a global array COEFTRI. The array COEFTRI is also used to determine which indices $\vec{\mathbf{j}}$ satisfy $\vec{\mathbf{0}} \le \vec{\mathbf{j}} \le \vec{\mathbf{m}}$ by letting the $(\vec{\mathbf{m}}, \vec{\mathbf{j}})$ entry be zero if $\vec{\mathbf{j}}$ is not $\le \vec{\mathbf{m}}$. (This is a concern since the hyperbox of indices, $\vec{\mathbf{0}} \le \vec{\mathbf{j}} \le \vec{\mathbf{m}}$, will usually not be a subinterval in a linear ordering of all indices.) In general, efficient index vector manipulation, such as $\vec{\mathbf{k}} - \vec{\mathbf{j}}$, is helpful. It's interesting to observe that directly defined PYRAMID operators (those not calling other PYRAMID operators) are essentially all equivalent in terms of complexity; for large n and maxorder, the difference between TIMES and EXP is insignificant.

The central problem is how to implement the PYRAMID structure and effficiently handle the index set manipulation. Since the dimension n is an arbitrary constant, it is difficult to actually use an n-dimensional array. For instance, $H[\vec{\mathbf{k}}]$ (directly indexing one array with another array) is impossible in most languages and $H[k_1, k_2, \ldots, k_n]$ is clearly a syntactic problem. (Direct coding of Sections 2 and 3 is probably only possible in a fully implemented APL2, which supports nonrectangular arrays of arbitrary size and allows selection and replacement of an entry specified by an index array. Actually, replacement is not yet implemented in APL2 and an extension such as proposed by Gerth and Orth [3] is necessary and convenient.)

A practical alternative is to ravel a PYRAMID into a long one-dimensional array, ordering entries by the value of the vector index when interpreted as a representation in base (maxorder + 1). Standard combinatorics shows that the number of entries in a PYRAMID (the number of distinct partials in n variables of order $\leq$ maxorder) is the binomial coefficient ((maxorder + n) choose n). We use three different indices to refer to a position in the array:

$\vec{\mathbf{k}}$	index vector as discussed in Sections 2 and 3,
kvalue	value of $\vec{\mathbf{k}}$ as a representation in base (maxorder + 1),
k	direct index in the one-dimensional array.

For simplicity and efficiency, the direct index may be used in most instances and the value may be used to perform the index arithmetic. The latter idea follows from a simple algebraic property for $\vec{\mathbf{j}} \leq \vec{\mathbf{k}}$:

$$\text{the value of } \left(\vec{\mathbf{k}} - \vec{\mathbf{j}}\right) \text{ is } (\text{kvalue} - \text{jvalue}).$$

Index vectors are not needed within any PYRAMID operator (including bdot); vectors are used at the beginning to determine the values and COEFTRI, and at the end to label the output in a PyramidPrint procedure.

Three global arrays facilitate the conversion between different index forms: VALUE maps k to kvalue, INDEX maps kvalue to k, and VECTOR maps k to $\vec{\mathbf{k}}$. Also COEFTRI maps (m, j) to coef$(\vec{\mathbf{m}}, \vec{\mathbf{j}})$ or zero.

```
Global Constants:
n
maxorder
topindex ← ((maxorder + n) choose n) − 1
topvalue ← maxorder * (maxorder + 1)^(n − 1)

Global Arrays:
j ← 0
FOR jvalue ← 0 to topvalue DO BEGIN
  j⃗ ← represent j with n digits in base (maxorder + 1)
  order ← sum of the digits in j⃗
  IF order ≤ maxorder THEN BEGIN
    INDEX[jvalue] ← j
    VALUE[j] ← jvalue
    VECTOR[j] ← j⃗
    j ← j + 1
  END
END
```

```
COEFTRI rows 0 through maxorder ← Pascal's Triangle
  (with zeros above the diagonal)
  [requires only one addition per entry]
FOR {row} m ← (maxorder + 1) TO topindex DO BEGIN
  FOR {column} j ← 0 TO m DO
    product ← 1
    FOR i ← 1 TO n DO BEGIN
      {miji is (m_i choose j_i) or zero if j_i > m_i}
      miji ← COEFTRI[ VECTOR[m][i], VECTOR[j][i] ]
      product ← product * miji
    END
    COEFTRI[m, j] ← product
  END
END
```

A PYRAMID is an array [0 . . . topindex] of reals. Now all of the algorithms of Section 3 have simple implementations in terms of direct indices.

```
FUNCTION sum ← bdot(P, m, Q, k)
BEGIN
  kvalue ← VALUE[k]
  sum ← 0
  FOR j ← 0 TO m DO BEGIN
    kminusj ← INDEX[ kvalue − VALUE[j] ]
    sum ← sum + COEFTRI[m, j] *P[j] * Q[kminusj]
  END
END
```

For many indices j (whenver $\vec{\mathbf{j}}$ is not $\leq \vec{\mathbf{m}}$), COEFTRI[m, j] will be zero. Also, most entries in P and Q are zero when these PYRAMIDS are associated with constants, variables (the basic building blocks), or any expression which does not use all variables. Hence, many unnecessary operations are avoided by adding tests for nonzero multiplicands in the above loop.

In every PYRAMID operator the loop "FOR $\vec{\mathbf{k}} \in J \ldots$" may be implemented by "FOR $k \leftarrow 0$ (or 1) TO topindex DO." References to $\vec{\mathbf{k}}$ may be replaced by the scalar index k. The only remaining problem is $\vec{\mathbf{e}}$ and $\vec{\mathbf{k}} - \vec{\mathbf{e}}$. Let $\vec{\mathbf{e}}(i)$ be the ith order-one vector, so that the value of $\vec{\mathbf{e}}(i)$ is evalue(i) = (maxorder + 1)$^{i-1}$. For a given k, choose the largest i such that evalue(i) ≤ kvalue < evalue(i + 1). For this value of i, VECTOR[k] has a nonzero ith coordinate (i.e., $\vec{\mathbf{e}}(i) \leq \vec{\mathbf{k}}$). Moreover, of all order-one vectors $\leq \vec{\mathbf{k}}$, this value yields the smallest index m = INDEX[VALUE[k] − evalue(i)] and, hence, the shortest possible loop in bdot. All the PYRAMID operators could be implemented as in the following example, LN.

```
FUNCTION H ← LN(U)
BEGIN
  H[0] ← ln(U[0])
  evalue ← 1
  nextevalue ← maxorder + 1
  FOR k ← 1 TO topindex DO BEGIN
    kvalue ← VALUE[k]
    IF kvalue = nextevalue THEN BEGIN
      evalue ← nextevalue
      nextevalue ← evalue * (maxorder + 1)
```

```
      END
      m ← INDEX[ kvalue − evalue ]
      H[k] ← 0
      H[k] ← ( U[k] − bdot(U, m, H, k) ) / U[0]
    END
  END
```

This paragraph considers an alternative hypercube data structure that is somewhat simpler to implement than the hyperpyramid described thus far. The hypercube method changes very little in Sections 2 and 3 except the index set $J = \{(j_1, j_2, \ldots, j_n) \mid j_i \leq \text{maxorder, for each } i\}$. However, when implemented, each direct index k is equal to kvalue. Thus the arrays INDEX and VALUE, together with the corresponding conversions, are totally unnecessary. Also, the array COEFTRI forms a pattern which earns the name the "Nested Pascal's Triangle." If the desired result is all partial derivatives up to maxorder derivatives in each variable, the hypercube is preferable. In fact, if space-efficiency is not a concern, then one can use the hypercube method and simply skip indices k where the sum of entries in VECTOR[k] exceeds maxorder.

5. USING PYRAMID OPERATORS TO DIFFERENTIATE FUNCTIONS

Finally, we consider the "driver program" which would perform automatic differentiation on given functions. For example, the following function would differentiate $f(x, y, z) = x \cdot \ln(y/z) + 5x + 3$ at a supplied point $\vec{\mathbf{a}} \in \mathbb{R}^n$. The function COORD($i, \vec{\mathbf{a}}$) returns the PYRAMID XI associated with the ith variable at $\vec{\mathbf{a}}$, as described in Section 2 and implemented using INDEX[evalue(i)].

```
FUNCTION F ← FDF(a⃗)
BEGIN
  X ← COORD(1, a⃗)
  Y ← COORD(2, a⃗)
  Z ← COORD(3, a⃗)
  F ← (X TIMES LN(Y DIV Z)) + (5 * X) + C(3)
END
```

In APL, the expression for F could be written as shown above (technically, $\times$ instead of $*$). This takes advantage of many features of APL: direct array addition and scalar multiplication; user-defined dyadic (binary "infix") operators like TIMES and DIV; and the fact that real functions such as ln, exp, and sin are given by special APL symbols, thus there is no confusion in using the names LN, EXP, and SIN for PYRAMID operators. In a language such as VAX Pascal, the PYRAMID operator translation of this function would be more awkward.

```
F := PSUM(PSUM(PTIMES(X, PLN(PDIV(Y, Z)))  , CTIMES (5, X)), C (3)).
```

In Pascal-SC and other languages which allow operator overloading, the PYRAMID expression would be even nicer than the above APL version because the definition of arithmetic operators (such as $*$) can be customized according to the type of the arguments (such as PYRAMID); see Rall [14].

Of course, the function does not have to be given by a one-line expression. At the other extreme, each operation could be a separate line; forming what is known as a "factorable sequence" for the factorable function [6]. A time-efficient compromise is to write separate lines for any expression that occurs more than once in the function. Even functions given by subroutines (loops, conditionals, etc.) can be easily differentiated by converting real operations to the corresponding PYRAMID operations. If space is a constraint, the expression should be written so as to minimize pending operations which require storing temporary PYRAMIDs.

The entire program consists of an initialization section which sets the global constants and arrays as described above, a library section including all of the PYRAMID operators, and an expression section including any number of functions like FDF (all with $\leq n$ variables). The main program could call these functions with as many evaluation points as desired. To increase the number of variables or maxorder would require rerunning the initialization section.

6. IDEAS FOR IMPROVEMENT

Two avenues hold much promise for the design of even more efficient methods: the reverse mode (applied to first-order partials of multivariable functions by Griewank [4]) and the use of Taylor series recursion formulas (applied to arbitrary order derivatives of single variable functions by Rall [13]).

The reverse mode of Griewank [4] discards the assumption that we need to calculate all partial derivatives for each subexpression in our function. In our "forward mode" method, we waste time computing partials (necessarily zero) with respect to variables that are not even in the subexpression. The proposed method only calculates partials of every operation with respect to its immediate argument(s) (organized by listing the function as a factorable sequence). Accumulating these partials in reverse order will result in the desired derivatives. Application of the reverse mode to Hessians is discussed by Iri [5, p. 245–247] and implemented by Christianson [2]. The design of an arbitrary order reverse method may be difficult and would be drastically different from the method presented here. The apparent advantages are tempered by the fact that space requirements for this method could grow arbitrarily large, corresponding to the complexity of the function; whereas, our method requires only the fixed PYRAMID space for immediately pending operations. Limits on the computational efficiency of a high-order reverse method are discussed in Section 3.6 of Griewank [4].

Another alternative is to carry arrays of Taylor series coefficients, since these differ from derivatives only by appropriate factorial factors. For single variable functions, such Taylor series methods have been in use for several decades. The advantage is that the rule, corresponding to Leibniz's rule, does not involve binomial coefficients (see summary by Moore [10]). Generalization of this method to multivariable Taylor series can eliminate references to binomial coefficients. Briefly, if $U[\mathbf{j}] = D_{\mathbf{j}}u(\vec{\mathbf{a}})/(j_1!j_2!\cdots j_n!)$, then TIMES

and DIV operators use the simplified definition

$$\text{bdot}\left(P, \vec{\mathbf{k}}, Q, \vec{\mathbf{k}}\right) = \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{k}}} P\left[\vec{\mathbf{j}}\right] * Q\left[\vec{\mathbf{k}} - \vec{\mathbf{j}}\right].$$

In other operators, where $\vec{\mathbf{e}}$ has a nonzero ith coordinate, use the definition

$$\text{bdot}\left(P, \vec{\mathbf{k}} - \vec{\mathbf{e}}, Q, \vec{\mathbf{k}}\right) = \frac{1}{k_i} \sum_{\vec{\mathbf{j}}=\vec{\mathbf{0}}}^{\vec{\mathbf{k}}-\vec{\mathbf{e}}} (k_i - j_i) * P\left[\vec{\mathbf{j}}\right] * Q\left[\vec{\mathbf{k}} - \vec{\mathbf{j}}\right].$$

Using these definitions, the operator algorithms of Section 3 remain essentially unchanged, often recursively calculating $H[\vec{\mathbf{k}}]$ in terms of bdot with H as an argument. The result is the PYRAMID of multivariable Taylor series coefficients.

Even in this modified multivariable Taylor series method, the large array COEFTRI can't be simply discarded since it also plays a role in managing the data structure (see second paragraph of Section 4). The challenge then becomes to more efficiently handle the data structure. Before final revision of this paper, the author learned of Berz's work [1] which describes an efficient approach to the management of a data structure for multivariable Taylor series coefficients. The formulas presented by Berz [1] for operator algorithms differ from the recursive forms just described above.

7. CONCLUSION

The hyperpyramid method is a natural culmination, to the most general case, of the method of automatic differentiation begun by Wengert in 1964. The method calculates the theoretically exact value of all unique partial derivatives, up to any order, of any typical expression in any number of variables. The algorithms are truly general in that the number of variables n and maximum order of partial derivatives can be changed by simply changing constants. The hyperpyramid structure contains the minimum number of entries possible to complete this task. The generalized Leibniz's rule eliminates repetitive calculations; in fact, for each $\vec{\mathbf{j}}$ and $\vec{\mathbf{k}}$ in J with the sum $\vec{\mathbf{j}} + \vec{\mathbf{k}}$ in J, the pair $D_{\vec{\mathbf{j}}}u(\vec{\mathbf{a}})$ and $D_{\vec{\mathbf{k}}}v(\vec{\mathbf{a}})$ is multiplied exactly once in the operation U TIMES V. Sections 4 and 6 show that the method is practical and present ideas for efficiency in real-time. Of course, roundoff error will limit the accuracy of the results but not nearly as much as in difference approximations.

REFERENCES

1. BERZ, M. Differential algebraic description of beam dynamics to very high orders. *Particle Accelerators 24*, 2 (1989), 109–124.
2. CHRISTIANSON, B. Automatic hessians by reverse accumulation. *IMA J Numer. Anal.* To appear.
3. GERTH, J. A., AND ORTH, D. L. Indexing and merging in APL *APL88—APL Quote Quad. 18*, 2 (Dec. 1987), 156–161.

4. GRIEWANK, A. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, M. Iri, and K. Tanabe, Eds. Kluwer, Amsterdam, 1989, 83-108.
5. IRI, M. Simultaneous computation of functions, partial derivatives and estimates of rounding errors—Complexity and practicality. *Japan J. Appl. Math. 1*, 2 (1984), 223-252.
6. JACKSON, R. H. F., AND MCCORMICK, G. P. The polyadic structure of factorable function tensors with applications to high-order minimization techniques. *J. Optim. Theor. Appl. 51*, 1 (Oct. 1986), 63-94.
7. KAGIWADA, H., KALABA, R., RASAKHOO, N., AND SPINGARN, K. *Numerical Derivatives and Nonlinear Analysis*. Plenum Press, New York, 1986.
8. KALABA, R., TESFATSION, L., AND WANG, J.-L. A finite algorithm for the exact evaluation of higher order partial derivatives of functions of many variables. *J. Math. Anal. Appl. 92*, 2 (1983), 552-563.
9. KEDEM, G. Automatic differentiation of computer programs. *ACM Trans. Math. Softw. 6*, 2 (June 1980), 150-165.
10. MOORE, R. E. *Methods and Applications of Interval Analysis*. SIAM Studies in Applied Mathematics 2, SIAM, Philadelphia, Pa., 1979, 24-31.
11. NEIDINGER, R. D. Automatic differentiation and APL. *College Math. J. 20*, 3 (May 1989), 238-251.
12. RALL, L. B. The arithmetic of differentiation. *Math. Mag. 59*, 5 (Dec. 1986), 275-282.
13. RALL, L. B. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science 120, Springer-Verlag, New York, 1981.
14. RALL, L. B. Differentiation in Pascal-SC: type GRADIENT. *ACM Trans. Math. Softw. 10*, 2 (June 1984), 161-184.
15. WENGERT, R. E. A simple automatic derivative evaluation program. *Commun. ACM 7*, 8 (Aug. 1964), 463-464.
16. WEXLER, A. S. An algorithm for exact evaluation of multivariate functions and their derivatives to any order. *Comput. Stat. Data Anal. 6*, 1 (1988), 1-6.
17. WEXLER, A. S. Automatic evaluation of derivatives. *Appl. Math. Comput. 24*, 1 (1987), 19-46.

Received March 1989; accepted June 1991