A Recursive Approach to Multivariate Automatic Differentiation

Dan Kalman
The American University
Washington, D.C. 20016

Robert Lindell
The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009-2957

April 16, 1997

Abstract

In one approach to automatic differentiation, the range of a function is generalized from a single real value to an aggregate representing the values of the function and one or more derivatives. The operations and functions of elementary analysis are extended to these aggregates so as to preserve the validity of the derivatives. In this paper we develop a recursive approach to defining the necessary operations in the context of functions of several variables. Formally, the definitions are essentially the same as those needed in the single variable case. The resulting system provides automatic propagation of values of all partial derivatives up to a prespecified order for a function of several variables.

1 Introduction

Automatic Differentiation refers to a family of techniques for automatically computing derivatives as a byproduct of function evaluation. A survey of different approaches can be found in [4]. In this paper we shall restrict our attention to what is called the *forward mode* of automatic differentiation, and in particular, the approach described in [17]. In this approach, to provide automatic calculation of the first m derivatives of real valued expressions of a single variable x, we define an algebraic system consisting of real $m+1$ tuples, and extend to these tuples the

familiar binary operations and elementary functions generally defined on real variables. The idea is that each tuple represents the value of a function and its first m derivatives, and that the operations preserve this interpretation. Thus, if $a = (a_0, a_1, \cdots, a_m)$ consists of the first m derivatives of f, and $b = (b_0, b_1, \cdots, b_m)$ consists of the first m derivatives of g, then the product ab will consist of the first m derivatives of fg. Similarly, the extension of the square-root function to tuples is so contrived that $\sqrt{a}$ will consist of the first m derivatives of $\sqrt{f}$. In an automatic differentiation system built along these lines, there must be some functions that are evaluated directly to produce tuples. For example, the constant function with value c can be evaluated directly to produce the tuple $(c, 0, \cdots, 0)$, and the identity function $I(x) = x$ can be evaluated directly to produce $(x, 1, 0, \cdots, 0)$. It may be convenient to directly evaluate other functions as well, for example, solutions to differential equations. Then, these evaluations may be combined with all of the operations on tuples to obtain values and derivatives of composite functions. As a simple example, consider the function $h(x) = (x^2 + 7)e^x$. In order to compute $(h(4), h'(4), h''(4))$, we substitute $(4, 1, 0)$ for x and $(7, 0, 0)$ for the constant 7 in the expression defining h. Evaluating the resulting tuple expression produces the desired result, $(h(4), h'(4), h''(4))$.

In this paper, we develop this approach in the context of real valued functions of several variables to automatically generate all partial derivatives up to a specified degree. As described above, it will be necessary to define an algebraic system whose elements represent the desired derivative values. But where the preceding discussion used tuples of reals, we will use more complicated objects called *derivative structures*. These are defined recursively: the derivative structures suitable for functions of n variables are tuples whose entries are derivative structures for $n - 1$ variables. The operations are also defined recursively. Interestingly, the definitions of these operations are formally identical to the operations used in the single variable case. This results in extremely simple and elegant definitions of the operations on derivative structures.

There is a strong analogy between the approach we will follow and the recursive construction of multivariate polynomial rings. There is a standard construction for adjoining an indeterminate to a ring, resulting in a polynomial ring. The same construction applied again yields a polynomial ring in two indeterminates over the original ring of coefficients. By way of analogy, the construction sketched above for automatic differentation relative to a single variable is well known. We will show that essentially this same construction can be iterated to provide automatic generation of partial derivatives relative to several variables.

Neidinger [11] develops an alternate approach to multivariate automatic differentiation. His structures are defined using multidimensional arrays and explicit subscript manipulation. One very attractive feature of this development is a recursion based on the differential order. In contrast, our approach as outlined above uses recursion based on the number of variables.

The recursion on the number of variables makes the addition of the n^{th} independent variable the same as the first. In a similar way, Neidinger's recursion on differential order makes the generation of the m^{th} derivative the same as the generation of the first derivative.

We have adapted Neidinger's style of recursion on differential order to our system of derivative structures. The result is extremely concise and elegant definitions of structures and operations in a system for multivariate automatic differentiation. The system supports an arbitrary number of variables and derivatives of arbitrary order. We implemented a sample system in LISP that provided automatic differentiation of arbitrary formulas composed from the operations of arithmetic, as well as the sine, cosine, exponential and squareroot functions. This implementation consists of fewer than 170 lines of code.

There are three points of emphasis in the presentation to follow. Of first emphasis is a formal mathematical system for automatic differentiation. This system is defined recursively with respect to the number of independent variables. Although the formal system can be implemented directly to automate the computation of all partial derivatives through a fixed differential order, it is mainly of theoretical interest. The second point of emphasis is the derivation of automatic differentiation algorithms which are recursive with respect to differential order. The validation of the algorithms depends on the theoretical foundation provided by the formal system. The final point of emphasis is the simplicity and elegance of a sample implementation of these algorithms.

Mention has been made of the relation of our work to that of Neidinger. Other related work includes [3] and [8]. The first of these considers the differentiation of composite functions, and the second considers the differentiation of inverse functions. We will remark on these related papers in somewhat greater detail after developing our approach below.

Here is an outline of the remainder of the paper. In the next section we will develop the structures that are used to store the partial derivatives in our system, and which are the operands in computations within the system. Section 3 describes how partial derivatives are arranged within the structures. Next, Sections 4 and 5 define the arithmetic operations and elementary functions of analysis for these structures. This will complete the presentation of the formal system that is mentioned as the first point of emphasis above. The second point of emphasis concerns recursion on differential order; that topic is considered in Section 6. The sample implementation, the third point of emphasis, is discussed in Section 7. The concluding section is a general discussion.

2 Recursively Defined Derivative Structures

Before giving the formal definition of derivative structures, we present an example to illustrate the idea. We begin with a single variable. If we are interested in the first two derivatives, the derivative structure is simply an ordered triple. Thus, we generalize from the mapping $x \to f(x)$ to $x \to (f(x), f'(x), f''(x))$. We will use the notation $f^{[1,2]}(x) = (f(x), f'(x), f''(x))$. The superscripts indicate a single variable and two derivatives, respectively.

Using the single variable case as a foundation, we proceed to the case of several variables. Suppose f is a function of three variables, x, y, z, and we wish to compute partial derivatives through the second order. To begin, we take the partials relative to x and arrange the results in a vector, just as in the single variable case: $f^{[1,2]} = (f, f_x, f_{xx})$ (where we indicate partial differentiation with respect to x by affixing a subscript of x). Now, let us perform the same operation on $f^{[1,2]}$, but this time differentiating with respect to y. The result is

$$(f^{[1,2]}, (f^{[1,2]})_y, (f^{[1,2]})_{yy}) = ((f, f_x, f_{xx}), (f_y, f_{xy}, f_{xxy}), (f_{yy}, f_{xyy}, f_{xxyy}))$$

Although this does generate the required derivatives, there are some of higher order than 2. A slight modification rectifies the situation: in each successive derivative relative to y, we truncate one position on the right. This produces

$$\begin{aligned}(f^{[1,2]}, (f^{[1,1]})_y, (f^{[1,0]})_{yy}) &= ((f, f_x, f_{xx}), (f, f_x)_y, (f)_{yy}) \\ &= ((f, f_x, f_{xx}), (f_y, f_{xy}), (f_{yy}))\end{aligned}$$

which stores all the partial derivatives up to order 2 with respect to x and y. This construction generalizes to arbitrary order in the obvious way. For example, if we are interested in derivatives up to order 3, we would construct at the first stage $(f, f_x, f_{xx}, f_{xxx})$, and at the second $((f, f_x, f_{xx}, f_{xxx}), (f_y, f_{xy}, f_{xxy}), (f_{yy}, f_{xyy}), (f_{yyy}))$. Because we performed this construction in two stages, let us refer to this as a second stage structure of order 3, and denote it by $f^{[2,3]}$. Consistent with this terminology, we refer to $f^{[1,3]}$ as a first stage structure of order 3. For a second stage structure of order 1, we construct at the first stage (f, f_x), and at the second $((f, f_x), (f_y))$. This is $f^{[2,1]}$.

To conclude the example, we will apply the same process once again, but differentiating this time with respect to z. At the preceding step, notice that with each successive derivative relative to y, we reduced by one the order of the first stage structure being differentiated. Now in analogous fashion, each derivative relative to z will operate on a stage 2 structure, and the orders of these structures will be reduced by 1 with each successive differentiation. Assuming that we are interested in derivatives through second order, we will begin

with the second order stage 2 structure derived above. Applying the construction just outlined, we construct $(((f, f_x, f_{xx}), (f_y, f_{xy}), (f_{yy})), \quad ((f, f_x), (f_y))_z, \quad ((f))_{zz})$ or, more simply, $(((f, f_x, f_{xx}), (f_y, f_{xy}), (f_{yy})), \ ((f_z, f_{xz}), (f_{yz})), \ ((f_{zz})))$ This is a stage 3 derivative structure. Notice that it does contain all partial derivatives through the second order relative to the three variables.

It should be noted here that there are two obvious approaches for defining an automatic differentiation system. In the example, and in the development to follow, we define the structures and operations abstractly. For example, we define the exponential function for the triple (a, b, c) rather than for $f^{[1,2]}$. There is no use made of the fact that a, b, and c are actually derivative values for some function. In this approach, it must be shown that the operations we defined abstractly have the right properties for performing automatic differentiation. In particular, identities that show that the abstract operations preserve the meaning of the components as derivatives must be satisfied. The alternative is to adopt these identities as the definitions of the operations. Returning to the example of the exponential function, we could define $e^{f^{[2]}}$ to be $(e^f)^{[2]}$. In this approach it is known *a priori* that the operations perform automatic differentiation correctly. However, one must establish that it is possible to realize the definitions, and derive computational formulas for the operations that depend only on the values stored in the structures, not on the underlying functions. It seems evident that these approaches are roughly equivalent. In one case we define an operation according to the computational formula and show it has the necessary properties, while in the other we characterize the operation in terms of the properties, and derive the computational formula as a consequence. We have chosen the first approach to emphasize that ultimately the operations are performed on structures with no knowledge of the functions those structures represent.

With the preceding example as motivation, we proceed to define the structures for our automatic differentiation system. We call them derivative structures. As in the example, each structure has a differential order m, and the structures are defined recursively in stages. From this point on we abandon the term *stage*, which is evocative of a recursive process, in favor of *level*, which is more suggestive of a hierarchical organization. Indeed, the derivative structures should be thought of as populating a hierarchy: the first stage structures occupy the first level, the second stage structures occupy the second level, and so on. In the definition, and throughout the paper, $\mathbf{R}$ is the set of real numbers. The symbol Π represents the cartesian product.

Definition 1 *For $m \geq 0$, we define $DS(n, m)$, the derivative structure at level n of order m, as follows.*

$$DS(n, m) = \begin{cases} \mathbf{R} & \text{if } n = 0 \\ \prod_{j=0}^{m} DS(n-1, m-j) & \text{if } n > 0 \end{cases}$$

The definition says that an element of $DS(n,m)$ is an ordered $m+1$-tuple. The first entry is an element of $DS(n-1,m)$, the second of $DS(n-1,m-1)$, and so on. We will follow the usual conventions of subscripting to denote the elements of a tuple. Thus, for $a \in DS(n,m)$ we will write $a = (a_0, a_1, \cdots, a_m)$ with the understanding that $a_j \in DS(n-1, m-j)$ holds. In a similar fashion, $a_{j,k}$ is the k^{th} element of a_j, $a_{j,k,l}$ is the l^{th} element of $a_{j,k}$, and so on.

$$\left[\begin{array}{llll} (& a_{00} & a_{01} & a_{02} \quad a_{03} \ ) \\ (& a_{10} & a_{11} & a_{12} \ ) \\ (& a_{20} & a_{21} \ ) & \\ (& a_{30} \ ) & & \end{array}\right]$$

Figure 1: An element of $DS(2,3)$.

Observe that we have as special cases, $DS(1,m) = \mathbf{R}^{m+1}$ ($m+1$ dimensional real space), and $DS(n,0)$ is $\mathbf{R}$. These are consistent with the example above, and represent extreme values of level and order. At level 1 we are interested in the derivatives with respect to a single variable, and hence to a vector of reals. With order 0 we care only about a function value, that is, a single real. The case $n = 2$ is more interesting. If we envision $a \in DS(2,m)$ as a column, with its j^{th} entry (which is a vector in $\mathbf{R}^{m-j}$) laid out as a row, then we obtain a triangular array. For $m = 3$ it appears as illustrated in Fig. 1. Similarly, an element of $DS(3,m)$ can be viewed as a pyramid. For the general case, observe that in an element of $DS(n,m)$, each numerical entry is identified by n subscripts $(s_1, s_2, \cdots, s_n)$, which we may visualize as a lattice point in $\mathbf{R}^n$. The complete set of entries of the derivative structure corresponds to the lattice points for which $s_j \geq 0$ and $0 \leq s_1 + s_2 + \cdots + s_n \leq m$. In the cases considered above, with $n = 1, 2, 3$, it is the index sets that confer the shapes of linear, triangular, and pyramidal arrays. So in general we visualize an element of $DS(n,m)$ as an n dimensional pyramid; the n edges which lie along the coordinate axes of $\mathbf{R}^n$ each have length m. In this view, n specifies the dimension and m the *size* of a derivative structure.

The focus of this discussion has been on individual numerical entries of derivative structures, at what might be called the lowest level of detail. This contrasts with the recursive point of view that will be emphasized throughout the paper, and which heeds only the highest level of detail, namely, the representation of a level n derivative structure as a tuple of level $n-1$ structures. Viewed at the lowest level of detail, it is clear that our derivative structures are essentially the same as those described in [11]. There, operations are formulated in terms of the individual elements which are identified by subscript vectors. In our approach, operations at one level are defined in terms of the components in the preceding level, allowing us to conceptually manipulate objects of one dimension, rather than n. The connections between

these two approaches will be further elaborated below.

Derivative structures have also been studied in the context of differential equations, referred to as *k-jets* (see [1]) or *prolongations* ([13]). Those developments refer to the algebraic and topological properties of derivative structures, but are not concerned with procedures for computation with the structures.

The definition of $DS(n,m)$ is explicitly recursive with respect to the level parameter n. For most of the development below, that is the recursive approach that will be used. There will be one case, however, where recursion relative to the order parameter m is used. This development depends on recognizing within an element of $DS(n,m)$ a substructure that belongs to $DS(n,m-1)$. In fact, there are $n+1$ simple ways to project $DS(n,m)$ into $DS(n,m-1)$. While they are not all needed for the recursion mentioned above, there is a nice connection between these projections and differentiation. Accordingly, we define the complete set of projections next, and will discuss the connection with differentiation in the sequel.

Definition 2 *Let* $a = (a_0, a_1, \cdots, a_{m-1}, a_m) \in DS(n,m)$. *For* $1 \leq r \leq n$ *the mapping* $\pi_r : DS(n,m) \rightarrow DS(n,m-1)$ *is defined as follows. If* $r = n$ *then* $\pi_r(a) = (a_1, a_2, \cdots, a_m)$, *otherwise,* $\pi_r(a) = (\pi_r(a_0), \pi_r(a_1), \cdots, \pi_r(a_{m-1}))$. *Similarly,* $\pi_0 : DS(n,m) \rightarrow DS(n,m-1)$ *is defined as* $(a_0, a_1, \cdots, a_{m-1})$ *if* $n = 1$, *and as* $(\pi_0(a_0), \pi_0(a_1), \cdots, \pi_0(a_{m-1}))$ *otherwise.*

There is a geometric interpretation of these projections. As mentioned earlier, we may visualize an element of $DS(n,m)$ as an array of lattice points comprising an n dimensional pyramid. In this view, each projection defined above corresponds to excising one $n-1$ dimensional face of the pyramid. This preserves the dimension n while reducing the size m by one. For example, an element of $DS(2,m)$ may be depicted as in Fig. 1. The projection π_2 has the effect of removing the top line of the triangle (all entries with first subscript equal to 0). Similarly, π_1 removes the left side of the triangle (all entries with second subscript 0). The remaining projection, π_0 removes all the entries on the hypotenuse of the triangle (the entries whose subscripts sum to $m = 3$). Similarly, visualize an element of $DS(3,m)$ as a tetrahedron. Three of the faces of the tetrahedron correspond to entries with one subscript equal to 0, while on the remaining face the subscripts of an element sum to m. The projections π_r for $r > 0$ each remove one of the first three faces, and the last face is removed by π_0.

As noted previously, the motivation for defining $DS(n,m)$ is to provide objects which can encapsulate the partial derivatives of a function of several variables. To indicate that f has p variables we will write $f : \mathbf{R}^p \rightarrow \mathbf{R}$, as a shorthand for the more correct $f : D \rightarrow \mathbf{R}$, $D \subset \mathbf{R}^p$. In general we do not assume $D = \mathbf{R}^p$. Unless stated otherwise it is assumed that $n \leq p$ and that f is m times continuously differentiable. Then we will use an element of $DS(n,m)$ to

store all of the partial derivatives of f up to order m relative to the first n variables. We also want to define arithmetic operations and functions of elementary analysis on $DS(n,m)$ in a way that is consistent with the storage of derivatives. Thus, if $*$ is one of the arithmetic operations that have been extended from $\mathbf{R}$ to $DS(n,m)$, and if f and g are functions, we would like to be able to obtain all the partial derivatives of $f*g$ by applying $*$ directly to the $DS(n,m)$ elements associated with f and g. Accordingly, there are three tasks before us. First, for each sufficiently differentiable function $f:\mathbf{R}^p \to \mathbf{R}$ we will define a corresponding function $f^{[n,m]}:\mathbf{R}^p \to DS(n,m)$. Second, we will define binary operations and elementary functions on $DS(n,m)$. Finally, we will demonstrate the compatibility of these definitions by showing for each operation $*$ and elementary function ϕ that $(f*g)^{[n,m]} = f^{[n,m]} * g^{[n,m]}$ and $\phi \circ f^{[n,m]} = (\phi \circ f)^{[n,m]}$. (As usual, $\phi \circ f$ indicates the composition of the functions ϕ and f.)

3 Storing Derivatives in Elements of $DS(n,m)$

Before proceeding, we introduce the notation ∂_n for the partial differentiation operator relative to the n^{th} variable. Thus, if $f(x_1, x_2, \cdots, x_p)$ is a differentiable real valued function, then $\partial_n f$ represents the partial derivative of f with respect to x_n. As mentioned earlier, we assume that f has continuous partial derivatives through order m, so that the order of application of the partial differentiation operators ∂_j is immaterial. If f is such a function, we define a function induced by f that takes its values in $DS(n,m)$. As for $DS(n,m)$ itself, the definition is recursive.

Definition 3 *For any f, and for $n \le p$, the function $f^{[n,m]}:\mathbf{R}^p \to DS(n,m)$ is defined as follows. For any $x \in \mathbf{R}^p$*

$$f^{[n,m]}(x) = \begin{cases} f(x) & \text{if } n = 0 \\ (f^{[n-1,m]}(x), \partial_n f^{[n-1,m-1]}(x), \cdots, {\partial_n}^m f^{[n-1,0]}(x)) & \text{if } n > 0 \end{cases}$$

Implicit in this definition is the understanding that ∂_r operates on $DS(n,m)$ componentwise.

A few examples will illustrate that this definition formalizes the construction presented at the start of Section 2. Consider first the case $n = 1$, and recall that $DS(1,m) = \mathbf{R}^{m+1}$. In this case we have

$$f^{[1,m]}(x) = (f(x), \partial_1 f(x), {\partial_1}^2 f(x), \cdots, {\partial_1}^m f(x))$$

If $p = 1$ as well, this is simply the vector of derivatives of a function of a single variable. Vectors of this type are quite common in discussions of forward mode automatic differentiation, (see

for example [7, 9, 17]). It is clear that $DS(n,m)$ is a generalization of this type of structure. As we shall see below, the operations we define on $DS(n,m)$ generalize those defined on one variable derivative vectors in just the same way.

For the next example, take $n = p = 2$ and $m = 2$. We will represent the two variables by x and y, with ∂_x and ∂_y defined accordingly. By definition, $f^{[2,2]}(x,y)$ is an element of $DS(2,2)$, and hence is a triple. Let us display that triple as a column vector:

$$f^{[2,2]} = \begin{bmatrix} f^{[1,2]} \\ \partial_y f^{[1,1]} \\ {\partial_y}^2 f^{[1,0]} \end{bmatrix}$$

Using our previous representation for $f^{[1,m]}$, we expand the entries of the column with this result:

$$f^{[2,2]} = \begin{bmatrix} (f, \partial_x f, {\partial_x}^2 f) \\ \partial_y (f, \partial_x f) \\ {\partial_y}^2 (f) \end{bmatrix} = \begin{bmatrix} (f & \partial_x f & {\partial_x}^2 f) \\ (\partial_y f & \partial_y \partial_x f) & \\ ({\partial_y}^2 f) & & \end{bmatrix} \tag{1}$$

It is apparent from these examples that the definition of $f^{[n,m]}$ formalizes the construction presented in Section 2.

We remarked earlier that we define operations on derivative structures recursively, without explicitly referring to the numerical entries present at the lowest level of the structures. However, once the derivative structure for a particular function has been computed, we do require the ability to select individual derivatives at will. Every derivative can be represented as a composition of the ∂_j with j in decreasing order. Thus, given $f^{[n,m]}$ we would like to access the entry corresponding to ${\partial_n}^{r_n} {\partial_{n-1}}^{r_{n-1}} \cdots {\partial_1}^{r_1} f$. From the definition, it is clear that component r_n of $f^{[n,m]}$ is the only place ${\partial_n}^{r_n}$ appears. This component is itself a derivative structure, and it is only in *its* r_{n-1}^{th} component that ${\partial_{n-1}}^{r_{n-1}}$ appears. The pattern that this suggests is formalized in our first theorem.

Theorem 1 *The relationship between partial derivatives of f and entries of $f^{[n,m]}$ is given by the following identity.*

$${\partial_n}^{r_n} {\partial_{n-1}}^{r_{n-1}} \cdots {\partial_1}^{r_1} f = (f^{[n,m]})_{r_n, r_{n-1}, \cdots, r_1}$$

Proof:

For $n = 1$, the required identity is just the definition of $f^{[1,m]}$. Arguing inductively, assume

the identity for $n-1$. Using first the definition of $f^{[n,m]}$, and then the induction hypothesis, we derive the following string of identities:

$$\begin{aligned}
(f^{[n,m]})_{r_n,r_{n-1},\cdots,r_1} &= (f^{[n,m]}{}_{r_n})_{r_{n-1},\cdots,r_1} \\
&= \partial_n{}^{r_n}(f^{[n-1,m-r_n]})_{r_{n-1},\cdots,r_1} \\
&= \partial_n{}^{r_n}(\partial_{n-1}{}^{r_{n-1}}\cdots\partial_1{}^{r_1}f) \\
&= \partial_n{}^{r_n}\partial_{n-1}{}^{r_{n-1}}\cdots\partial_1{}^{r_1}f
\end{aligned}$$

□

As an application, we describe an algorithm to construct the derivative structure associated for the j^{th} component function $I_j(x_1, x_2, \cdots, x_n) = x_j$. Clearly, $\partial_j I_j = 1$, and all other derivatives are 0. Thus, given the value of x_j, we construct $I_j{}^{[n,m]}$ as follows. Begin with an element of $DS(n,m)$ with all entries equal to 0, and make just two assignments:

$$\begin{aligned}
(I_j{}^{[n,m]})_{0,0,\cdots,0} &= x_j \\
(I_j{}^{[n,m]})_{0,0,\cdots,1,\cdots,0} &= 1
\end{aligned}$$

In the latter equation, the subscript vector has a one in position j and all other entries 0. Applying these operations for the case $(n,m) = (2,2)$ we obtain

$$I_x{}^{[2,2]}(x,y) = \begin{bmatrix} (x & 1 & 0) \\ (0 & 0) & \\ (0) & & \end{bmatrix}$$

and

$$I_y{}^{[2,2]}(x,y) = \begin{bmatrix} (y & 0 & 0) \\ (1 & 0) & \\ (0) & & \end{bmatrix}$$

These are consistent with Eq. (1).

In the preceding exposition, we have shown how the the derivatives of f and the entries of $f^{[n,m]}$ are related. The next theorem provides several algebraic rules relating differentiation, derivative structures, and projections.

Theorem 2 *The following identities hold for any f and for $1 \le r \le n$:*

$$\partial_r(f^{[n,m]}) = (\partial_r f)^{[n,m]} \tag{2}$$

$$\pi_r f^{[n,m]} = \partial_r(f^{[n,m-1]}) \tag{3}$$

$$\pi_0 f^{[n,m]} = f^{[n,m-1]} \tag{4}$$

The proof of each identity is a straightforward induction argument, very similar to the proof of the previous theorem. The interested reader will find the proofs in the appendix.

In automatic differentiation, derivative structures are used in place of numerical values as the operands appearing in equations. In particular, we will replace each constant c with the derivative structure of the corresponding constant function, and each variable x_j with the derivative structure of the corresponding component function $I_j(x_1, x_2, \cdots, x_p) = x_j$. Let us adopt the notation $c^{[n,m]}$ for the derivative structure of the constant function with value c. Naturally, we will need to compute $c^{[n,m]}$ and ${I_j}^{[n,m]}$ before they can be used as operands. Although we previously described ${I_j}^{[n,m]}$ using explicit subscripting, it is also of interest to provide a recursive definition. For this purpose, and for the constant functions as well, we may specialize Definition 3, using Eq. (2) for simplifying the terms of the form ${\partial_n}^j f^{[n-1,m-j]}$. For constants, we obtain

$$c^{[n,m]} = \begin{cases} c & \text{if } n = 0 \\ (c^{[n-1,m]}, 0^{[n-1,m-1]}, \cdots, 0^{[n-1,0]}) & \text{if } n > 0 \end{cases}$$

For I_j Definition 3 specializes to

$${I_j}^{[n,m]}(x) = \begin{cases} x_j & \text{if } n = 0 \\ ({I_j}^{[n-1,m]}(x), 1^{[n-1,m-1]}, 0^{[n-1,m-2]}, \cdots 0^{[n-1,0]}) & \text{if } j = n \\ ({I_j}^{[n-1,m]}(x), 0^{[n-1,m-1]}, 0^{[n-1,m-2]}, \cdots 0^{[n-1,0]}) & \text{otherwise} \end{cases} \tag{5}$$

So far, we have defined what derivative structures are, and shown how a real valued differentiable function induces a corresponding map with derivative structures as values. The next section will carry on this development by defining operations on derivative structures.

4 Arithmetic Operations on Derivative Structures

In this section we will extend the binary arithmetic operations to derivative structures. In a subsequent section, we will show how to define elementary functions on $DS(n, m)$. In particular, we will show how to extend the real function $f(x) = 1/x$ to operate on derivative structures. This will permit the division of derivative structures to be defined in terms of multiplication and inversion: $a/b = a \times (1/b)$. Therefore, we will not define division of derivative structures at this time.

At this point it may be helpful to emphasize again the approach that is being followed. The definitions which will be given for arithmetic operations are contrived to commute with the

mapping $f \rightarrow f^{[n,m]}$. That is, each operation $*$ is to satisfy $f^{[n,m]} * g^{[n,m]} = (f * g)^{[n,m]}$. In fact, these operations can be defined in a way that is independent of the choice of the functions f and g. We highlight this fact by formulating the operations as computational rules in terms of the components of derivative structures, rather than in terms of the entries of $f^{[n,m]}$ and $g^{[n,m]}$. Given this approach, it is necessary to verify that the definitions are contrived properly, and that $f^{[n,m]} * g^{[n,m]}$ truly reproduces $(f * g)^{[n,m]}$.

Addition and subtraction are defined componentwise, while multiplication amounts to a convolution. This naturally entails multiplying each component of one derivative structure by every component of another and, in particular, leads to multiplying and adding derivative structures of unequal orders. There is no conceptual difficulty in such operations, we simply ignore extraneous components as necessary. The resulting definitions permit arithmetic operations on derivative structures with a common level index n, irrespective of their orders.

All of the definitions are recursive on level: operations on elements at level n are defined in terms of components which are themselves derivative structures of level $n-1$. We implicitly exploit the fact that the operations are already defined on $DS(0,m) = \mathbf{R}$. Here are the definitions for addition and subtraction.

Definition 4 *For $n > 0$, let $a = (a_0, a_1, \cdots, a_p) \in DS(n,p)$, $b = (b_0, b_1, \cdots, b_q) \in DS(n,q)$, and $m = \min(p,q)$. Then*

$$a + b = (a_0 + b_0, a_1 + b_1, \cdots, a_m + b_m) \tag{6}$$

$$a - b = (a_0 - b_0, a_1 - b_1, \cdots, a_m - b_m) \tag{7}$$

As asserted, this definition permits us to add or subtract any two derivative structures with a common value of n. The result is in $DS(n,m)$ where m is the smaller of the orders of the two derivative structures. Next we define multiplication.

Definition 5 *For $n > 0$, let $a = (a_0, a_1, \cdots, a_p) \in DS(n,p)$, $b = (b_0, b_1, \cdots, b_q) \in DS(n,q)$, and $m = \min(p,q)$. Then, the product $a \times b$ is the element of $DS(n,m)$ defined by*

$$(a \times b)_j = \sum_{k=0}^{j} \binom{j}{k} a_k \times b_{j-k} \; ; \quad 0 \leq j \leq m \tag{8}$$

As before the definition is recursive. Evidently inspired by Leibniz' rule, Eq. (8) is formally identical to the definition of multiplication for derivative vectors used in the single variable case,

as in [7, 11]. There, the components of a and b are real numbers, and it is a simple matter to verify that $f^{[1,m]} \times g^{[1,m]} = (fg)^{[1,m]}$. In the general case with $n > 1$, the symbols have a slightly different meaning. The components a_k and b_{j-k} are not reals, they are derivative structures, though at a lower level than a and b. Thus multiplication for $DS(n, m)$ is defined in terms of multiplication in $DS(n-1, m)$. Notice that we have again formulated the definition to apply whenever two derivative structures share a common n.

It is instructive to verify that the orders of the components of $a \times b$ are defined properly. For a particular j, with $0 \leq j \leq m$, the component $(a \times b)_j$ is supposed to have order $m - j$. Now the definition gives $(a \times b)_j$ as a sum of products. The order of the result will clearly equal the minimum of the orders of the operands. As k runs from 0 to j, the order of a_k (which is $p - k$) decreases from p to $p - j$. Similarly, the order of b_{j-k} increases from $q - j$ to q. The order of the sum is thus $\min(p - j, q - j) = \min(p, q) - j = m - j$. This shows that each component of the product is a derivative structure of the correct order.

As a final operation, define scalar multiplication componentwise. To be more precise,

Definition 6 *If $n > 0$, $a \in DS(n, m)$ and s is a real number, we define the product of a and s by*

$$sa = (sa_0, sa_1, \cdots, sa_m) \tag{9}$$

At this point, one might proceed to demonstrate that the operations just defined have the usual algebraic properties (associativity, commutativity, etc.). However, as we shall explain shortly, these properties are inherited from $\mathbf{R}$ by virtue of the mapping $f \to f^{[n,m]}$. First, we observe that the operations are compatible with the mapping, in the sense that the following identities hold:

$$f^{[n,m]} + g^{[n,m]} = (f + g)^{[n,m]} \tag{10}$$

$$f^{[n,m]} - g^{[n,m]} = (f - g)^{[n,m]} \tag{11}$$

$$f^{[n,m]} \times g^{[n,m]} = (fg)^{[n,m]} \tag{12}$$

$$sg^{[n,m]} = (sg)^{[n,m]} \tag{13}$$

In the appendix we include a proof for Eq. (12), the other proofs being similar. It is also worth mentioning that the usual rules of differentiation extend to derivative structures. That is, for derivative structure valued functions a and b we have the product rule

$$\partial_r(a \times b) = (\partial_r a) \times b + a \times (\partial_r b)$$

and Leibniz' rule

$$\partial_r{}^j(a \times b) = \sum_{k=0}^{j} \binom{j}{k} \partial_r{}^k a \times \partial_r{}^{j-k} b$$

These, too, are proved in the appendix.

The four identities Eqs. $(10-13)$ show that we may operate on functions and their derivative structures interchangeably. One consequence of this is that the operations on derivative structures inherit all the usual properties of the corresponding operations on real numbers. For example, to show that derivative structure multiplication is commutative, let a and b be arbitrary elements of $DS(n,m)$. There are functions f and g such that $f^{[n,m]}(0) = a$ and $g^{[n,m]}(0) = b$. Then we have

$$\begin{aligned} a \times b &= f^{[n,m]}(0) \times g^{[n,m]}(0) \\ &= (fg)^{[n,m]}(0) \\ &= (gf)^{[n,m]}(0) \\ &= g^{[n,m]}(0) \times f^{[n,m]}(0) \\ &= b \times a \end{aligned}$$

As another application, we observe that the Eqs. $(10-13)$ imply that polynomial functions extend to derivative objects in the natural way. For future reference, we state this as

Theorem 3 *If $p(w_1, w_2, \cdots, w_t)$ is a polynomial in t variables, then p is defined on derivative structures, provided all the arguments are at the same level. In addition, for any functions $f_1, f_2, \cdots, f_t$, we have*

$$p({f_1}^{[n,m_1]}, {f_2}^{[n,m_2]}, \cdots, {f_t}^{[n,m_t]}) = [p(f_1, f_2, \cdots, f_t)]^{[n,m]} \tag{14}$$

where $m = \min(m_1, m_2, \cdots, m_t)$.

A special case of this theorem is that any single variable polynomial p extends to derivative structures such that $(p \circ f)^{[n,m]} = p \circ f^{[n,m]}$ for all f. In the next section we generalize this result from polynomials to elementary functions of analysis.

5 Elementary Functions on Derivative Structures

This section will complete the development of derivative structures. Its goal is to define elementary functions ϕ on $DS(n,m)$ in such a way that the identity

$$\phi \circ f^{[n,m]} = (\phi \circ f)^{[n,m]} \tag{15}$$

holds. Then, if we are interested in the derivative structure for the composite function $\phi \circ f$, we can begin by computing the derivative structure for f and then apply ϕ directly to the result. In normal scientific computation, formulas are evaluated by performing sequences of operations on numerical values stored in memory. Our automatic differentiation system works similarly, although the operands are derivative structures rather than individual numerical values. Think of $f^{[n,m]}$ as being an intermediate result that has occurred in the evaluation of a complicated expression, with ϕ the next operation to be applied. That is the motivation for defining the operation of ϕ on derivative structures.

The recursive approach developed so far follows a simple pattern: definitions at one level are formulated in terms of objects at the next lower level using relations that are formally identical to the definitions at the lowest level, single variable automatic differentiation. The extension of elementary functions to the derivative structures follows the same pattern. Before presenting the general formulation, we consider a special case that illustrates the main idea.

In single variable automatic differentiation, the exponential function is defined on vectors, which exist in our development as elements of $DS(1,m)$. Consider for example taking $m = 2$. Then the exponential function is defined for real triples (a,b,c) according to the equation

$$e^{(a,b,c)} = (e^a, be^a, (b^2 + c)e^a) \tag{16}$$

For any function f, $f^{[1,2]} = (f, f', f'')$, and it is easy to verify that $e^{f^{[1,2]}} = (e^f)^{[1,2]}$. Thus, Eq. (16) provides the proper definition for the exponential function on $DS(1,2)$. The interesting thing is that we can use this same equation to define the exponential function on $DS(2,2)$. That is, if $(a,b,c) \in DS(2,2)$, then the components are derivative structures at level 1, and all of the operations that appear on the right side of Eq. (16) make sense. Moreover, this is the *right* definition for the exponential on $DS(2,2)$, because it implies that $e^{f^{[2,2]}} = (e^f)^{[2,2]}$. And now that the exponential is defined on $DS(2,2)$, we can use Eq. (16) again to define it on $DS(3,2)$. Continuing in this way, the exponential can be defined on $DS(n,2)$ for any n, in each case preserving the identity $e^{f^{[n,2]}} = (e^f)^{[n,2]}$. That is, Eq. (16) defines the exponential function on $DS(n,2)$ recursively in terms of functions defined on $DS(n-1,2)$. This idea generalizes. Informally, once we have an equation that defines a function ϕ on $DS(1,m)$ so

that $\phi(f^{[1,m]}) = (\phi \circ f)^{[1,m]}$, the same equation can be used to define ϕ on $DS(n,m)$, and with that definition, $\phi(f^{[n,m]}) = (\phi \circ f)^{[n,m]}$. To formalize this idea, we present a general definition for the extension of a function ϕ to $DS(n,m)$, and prove as a theorem that this definition satisfies $\phi(f^{[n,m]}) = (\phi \circ f)^{[n,m]}$. However, the general definition is not proposed for actual computation. We shall see that in particular examples, such as for the exponential function, the general definition can be vastly simplified. Indeed, for all of the elementary functions of analysis, simple extensions to $DS(1,m)$ are easily formulated for specific low values of m. The point of the general development is to validate the recursive use of these simple extensions to operate on $DS(n,m)$. In what follows, we will assume that ϕ is an m times continuously differentiable real function defined on some domain $D \subseteq \mathbf{R} = DS(0,m)$. When we speak of extending ϕ to $DS(n,m)$, we really mean that ϕ will extend to the subset of $DS(n,m)$ corresponding to D.

To generalize the procedure used with the exponential function, we begin with a simple observation about derivatives of composite functions, stated without proof.

Lemma 1 *For each nonnegative integer j, there is a polynomial $P_j(u_0, u_1, \cdots, u_j, v_1, \cdots, v_j)$ such that for any sufficiently differentiable functions $\phi : \mathbf{R} \to \mathbf{R}$ and $f : \mathbf{R}^p \to \mathbf{R}$, we have for $1 \leq r \leq p$*

$$\partial_r{}^j(\phi \circ f) = P_j(\phi \circ f, \phi' \circ f, \cdots, \phi^{(j)} \circ f, \partial_r f, \partial_r{}^2 f, \cdots, \partial_r{}^j f)$$

For example, since $\partial_r(\phi \circ f) = (\phi' \circ f)\partial_r f$, we see that $P_1(u_0, u_1, v_1) = u_1 v_1$. Similarly, $\partial_r{}^2(\phi \circ f) = (\phi'' \circ f)(\partial_r f)^2 + (\phi' \circ f)\partial_r{}^2 f$. This shows that $P_2(u_0, u_1, u_2, v_1, v_2) = u_2 v_1^2 + u_1 v_2$. Note that we include u_0 only so that the lemma makes sense for $j = 0$, in which case, $P_0(u_0) = u_0$. It is clear in these examples, and true in general, that P_j does not depend on the choice of functions ϕ and f.

We are now in a position to extend the function ϕ to $DS(n,m)$. As before, the definition is recursive on level. That is, for a derivative structure at level n, the definition of $\phi(a)$ depends on the application of ϕ (as well as some of its derivatives) to derivative structures at level $n-1$.

Definition 7 *Suppose $\phi, \phi', \cdots, \phi^{(m)}$ have been extended to $DS(n-1,m)$, and are defined at $a_0 \in DS(n,m)$. Then, for any $a = (a_0, a_1, \cdots, a_m) \in DS(n,m)$, $\phi(a) = (\phi_0, \phi_1, \cdots, \phi_m)$ where $\phi_j = P_j(\phi(a_0), \phi'(a_0), \cdots, \phi^{(j)}(a_0), a_1, a_2, \cdots, a_j)$*

Notice that every argument of the polynomial P_j is a derivative structure at level $n-1$ so the definition does in fact result in such a derivative structure. Moreover, the minimal order of

all the arguments is $m-j$, the order of a_j. Thus, ϕ_j does have order $m-j$, as it should. Since P_j is a polynomial, each ϕ_j is certainly defined.

Let us reexamine the example of the exponential function in the light of this definition. We require only three of the P_j: $P_0(u) = u$, $P_1(u_0, u_1, v_1) = u_1 v_1$, and $P_2(u_0, u_1, u_2, v_1, v_2) = u_2 v_1^2 + u_1 v_2$. Then the definition says that $e^{(a_0,a_1,a_2)} = (P_0(e^{a_0}), P_1(e^{a_0}, e^{a_0}, a_1), P_2(e^{a_0}, e^{a_0}, e^{a_0}, a_1, a_2))$. Using the definitions of the P_i, we derive $e^{(a_0,a_1,a_2)} = (e^{a_0}, e^{a_0}a_1, e^{a_0}a_1^2 + e^{a_0}a_2)$, as before. In exactly the same way, we can extend the reciprocal function $\phi(x) = 1/x$ to $DS(2,m)$. Note that $\phi'(x) = -1/x^2 = -\phi(x)^2$ and $\phi''(x) = 2/x^3 = 2\phi(x)^3$. Thus, if ϕ is defined at level $n-1$, so are its first two derivatives, so that Definition (7) gives

$$\frac{1}{(a_0, a_1, a_2)} = (\frac{1}{a_0}, -\frac{a_1}{a_0^2}, \frac{2a_1^2}{a_0^3} - \frac{a_2}{a_0^2})$$

This provides for division of derivative structures, at least through order 2. As a final example, take $\phi(x) = \sqrt{x}$ so that $\phi'(x) = \frac{1}{2\sqrt{x}}$ and $\phi''(x) = \frac{-1}{4x\sqrt{x}}$. Then

$$\sqrt{(a_0, a_1, a_2)} = (\sqrt{a_0}, \frac{a_1}{2\sqrt{a_0}}, \frac{a_2}{2\sqrt{a_0}} - \frac{a_1^2}{4a_0\sqrt{a_0}})$$

The point of these examples is to illustrate the meaning of the definition, and to demonstrate that the definition does agree with the ideas that were presented informally at the start of this section. The examples also illustrate that simplification can greatly reduce the complexity that is suggested by the use of the polynomial P_j. It still remains to verify that the definition is generally the *right* one from the standpoint of automatic differentiation. This is one of the main results of the paper. We present it as our next theorem.

Theorem 4 *If ϕ is extended to $DS(n,m)$ as in Definition (7) (as are all derivatives of ϕ that are recursively referred to by Definition (7)), and if ϕ is defined at $f^{[n,m]}(x)$, then $\phi(f^{[n,m]}(x)) = (\phi \circ f)^{[n,m]}(x.)$*

Proof:

We proceed by induction, and observe that when $n = 0$ there is nothing to prove. We will show that the j^{th} components of the two sides are equal. To simplify the notation, we suppress the variable x. By definition, the j^{th} component of $\phi(f^{[n,m]})$ is given by

$$\begin{aligned}(\phi(f^{[n,m]}))_j &= [\phi(f^{[n-1,m]}, \partial_n f^{[n-1,m-1]}, \cdots, \partial_n{}^m f^{[n-1,0]})]_j \\ &= P_j(\phi(f^{[n-1,m]}), \phi'(f^{[n-1,m]}), \cdots, \phi^{(j)}(f^{[n-1,m]}), \\ &\qquad \partial_n f^{[n-1,m-1]}, \cdots, \partial_n{}^j f^{[n-1,m-j]})\end{aligned}$$

By the induction hypothesis, every one of the expressions $\phi^{(k)}(f^{[n-1,m]})$ can be replaced by $(\phi^{(k)} \circ f)^{[n-1,m]}$. Also, by Eq. (2), the term ${\partial_n}^k f^{[n-1,m-k]}$ can be replaced by $({\partial_n}^k f)^{[n-1,m-k]}$. Making these substitutions we obtain

$$\begin{aligned}(\phi(f^{[n,m]}))_j &= P_j((\phi \circ f)^{[n-1,m]}, (\phi' \circ f)^{[n-1,m]}, \cdots, (\phi^{(j)} \circ f)^{[n-1,m]}, \\ &\qquad ({\partial_n} f)^{[n-1,m-1]}, \cdots, ({\partial_n}^j f)^{[n-1,m-j]})\end{aligned}$$

Now every argument of the polynomial P_j is of the form $h^{[n-1,t]}$ for some function h and some integer t between $m-j$ and m. Therefore, we may apply Eq. (14), with the result

$$(\phi(f^{[n,m]}))_j = [P_j(\phi \circ f, \phi' \circ f, \cdots, \phi^{(j)} \circ f, \partial_n f, \cdots, {\partial_n}^j f)]^{[n-1,m-j]}$$

By definition of P_j, the value of the polynomial is ${\partial_n}^j(\phi \circ f)$. This produces the final set of equations

$$\begin{aligned}(\phi(f^{[n,m]}))_j &= [{\partial_n}^j(\phi \circ f)]^{[n-1,m-j]} \\ &= [(\phi \circ f)^{[n,m]}]_j\end{aligned}$$

Therefore, $\phi(f^{[n,m]}) = (\phi \circ f)^{[n,m]}$. □

This theorem is a crucial justification for automatic differentiation. Observe that the proof assumes the elementary functions of interest act on $DS(n,m)$ according to Definition 7. However, most of the elementary functions we are interested in will be defined by composing other functions. This suggests a possible ambiguity: if ϕ_1 and ϕ_2 are elementary functions, then their composition $\phi_3 = \phi_1 \circ \phi_2$ can be extended to $DS(n,m)$ in two ways. First, we may use Definition 7 directly on the function ϕ_3. The proof of the theorem assumes this approach. In the second method, the definition is used to extend ϕ_1 and ϕ_2 to $DS(n,m)$, and then these extensions are composed. Many of the operations applied in automatic differentiation will be computed using this approach. The theorem shows that both approaches result in the same extension of ϕ_3. Indeed, any $a \in DS(n,m)$ can be represented as $f^{[n,m]}(0)$. Thus, $\phi_3(a) = \phi_3(f^{[n,m]}(0)) = (\phi_3 \circ f)^{[n,m]}(0) = (\phi_1 \circ \phi_2 \circ f)^{[n,m]}(0) = \phi_1[(\phi_2 \circ f)^{[n,m]}(0)] = \phi_1[\phi_2(a)]$. This justifies us in using Definition 7 only for a few basic functions, deriving the other elementary functions (including the derivatives of some of the basic functions) through arithmetic operations and composition. Put another way, it shows that the preceding theorem applies whether we think of ϕ and its derivatives as defined by Definition 7 or as compositions of basic functions.

Another Example

As a final topic in this section, we apply all of the foregoing in an example illustrating the use of automatic differentiation. Suppose we wish to calculate all partial derivatives through the second order of the function $f(x, y, z) = (e^x - y \sin z)/(x^2 + y)$ at some point, say (2,3,.5). That is, we wish to determine $f^{[3,2]}(2,3,.5)$. Observe that we can express f as a combination of simpler functions:

$$f = \frac{e^{I_x} - I_y \times \sin(I_z)}{I_x^2 + I_y}$$

Thus, by repeatedly applying Theorem 4 and Eqs. $(10 - 12)$, we derive

$$f^{[3,2]} = \frac{e^{{I_x}^{[3,2]}} - {I_y}^{[3,2]} \times \sin({I_z}^{[3,2]})}{({I_x}^{[3,2]})^2 + {I_y}^{[3,2]}} \tag{17}$$

Now the numerical entries of the derivative structures for the functions I_x, I_y, and I_z can be explicitly formulated according to Eq. (5), using 2 for x, 3 for y, and .5 for z. Then the computation of the right-hand side of Eq. (17) can be completed using our derivative structure operations and elementary functions. The ultimate result will be $f^{[3,2]}(2, 3, .5)$. This would be a suitable exercise to program on a computer. It is not recommended for computation by hand.

This example suggests an architecture for an automatic differentiation system up to a fixed order, say 2. Procedures are coded to perform addition, subtraction, multiplication, and scalar multiplication according to Eqs. $(6 - 9)$. A few basic functions are explictly coded such as $e^x, 1/x, \sqrt{x}, \sin x$, and $\cos x$. The system is coded so that assignments of the form $x_j = 7$ produce a derivative structure in accord with Eq. (5). Finally, formulas composed of variables, operations, and basic functions are evaluated using the corresponding derivative structure definitions. In this process, each invocation of a basic derivative function will trigger recursive calls to other functions with operands at the next lower level. Care must be taken to assure that each recursively called function is defined. That is, the basic functions must be closed in the sense that their evaluation involves only other basic functions. Thus, including $\sin x$ as a basic function requires including $\cos x$ also. Some obvious examples of sets of basic functions that have the required closure property are $\{1/x\}$, $\{1/x, \ln x\}$, $\{1/x, \arctan x\}$, $\{1/x, \sqrt{x}\}$, $\{e^x\}$, $\{\cos x, \sin x\}$, and unions of these sets. Several of the basic functions have domains that are proper subsets of the reals, and the computational procedures should be designed to flag exceptions when they occur. Theorem 4 shows that when all the computations can be carried through without meeting such exceptions, the derivatives will be properly calculated.

6 Recursion on Order

As just described, direct application of Definition (7) provides a simple foundation for an automatic differentiation system involving an arbitrary number of variables, but with a fixed limit on the number of derivatives. A more flexible system would permit whatever order a computation specified, providing essentially arbitrary order. In [11] this is achieved by recursively defining higher order derivatives for the elementary functions. In this section we apply the approach of [11] to derivative structures. More specifically, we show how to compute the value of a function ϕ on a derivative structure using recursion on order. Theorem 4 provides a useful theoretical tool to verify that the algorithm is correct.

Theorem 5 *The extension of the elementary function ϕ to $DS(n,m)$ can be formulated recursively in terms of both n and m as follows. For $a = (a_0, \cdots, a_m) \in DS(n,m)$, $\phi(a) = (\phi_0, \phi_1, \cdots, \phi_m)$ where*

$$\phi_j = \begin{cases} \phi(a_0) & \text{if } j = 0 \\ [\phi'(\pi_0(a)) \times \pi_n(a)]_{j-1} & \text{if } j > 0 \end{cases}$$

Proof:

The identity that is to be proven has two parts. The first part, when $j = 0$, follows directly from Definition (7). The second part of the identity amounts to the equality of $\pi_n(\phi(a))$ with $\phi'(\pi_0(a)) \times \pi_n(a)$. We will establish this for $a = f^{[n,m]}$ using the properties we have derived for derivative structures. As commented in an earlier discussion, this will suffice to demonstrate the theorem for arbitrary a.

Beginning with $\pi_n(\phi(f^{[n,m]}))$, we will make a series of transformations as follows:

$$\begin{aligned} \pi_n(\phi(f^{[n,m]})) &= \pi_n[(\phi \circ f)^{[n,m]}] && \text{(Theorem 4)} \\ &= \partial_n(\phi \circ f)^{[n,m-1]} && \text{(Eq. (3))} \\ &= [\partial_n(\phi \circ f)]^{[n,m-1]} && \text{(Eq. (2))} \\ &= [(\phi' \circ f)\partial_n f]^{[n,m-1]} \\ &= (\phi' \circ f)^{[n,m-1]} \times \partial_n f^{[n,m-1]} && \text{(Eq. (12))} \end{aligned}$$

To conclude the proof, we rewrite each factor of this last expression. The first factor becomes $\phi'(f^{[n,m-1]})$, hence $\phi'(\pi_0 f^{[n,m]})$ by applying Theorem 4 to ϕ', and then Eq. (4). The second factor becomes first $\partial_n(f^{[n,m-1]})$ via Eq. (2), and then $\pi_n(f^{[n,m]})$ via Eq. (3). This shows that $\pi_n(\phi(f^{[n,m]})) = \phi'(\pi_0 f^{[n,m]}) \times \pi_n(f^{[n,m]})$. Replacing $f^{[n,m]}$ by a gives us the required identity,

completing the proof. □

As a simple example, consider $\phi(x) = e^x$. The theorem says that we can calculate e^a as $(e^{a_0}, e^{\pi_0 a} \times \pi_n a)$ for any $a \in DS(n,m)$. Here, we have abused the notation slightly by writing $(b_0,(b_1,\cdots,b_m))$ where $(b_0,b_1,\cdots,b_m)$ is what is really intended. This should cause no confusion. A similar example is provided by $\phi(x) = \sin x$. This time we have $\sin(a) = (\sin(a_0), \cos(\pi_0 a) \times \pi_n a)$. As a final example, let $\phi(x) = 1/x$. Then we have

$$\frac{1}{a} = (\frac{1}{a_0}, \frac{-1}{(\pi_0 a)^2} \times \pi_n a)$$

This makes sense because a_0 is at a lower level than a, and $\pi_0 a$ is of lower degree, so that the reciprocal of each can be evaluated recursively.

These examples illustrate the application of Theorem 5. The method of the proof can also be applied to obtain a recursive form for the multiplication of derivative structures. The first step is to derive the identity

$$\pi_n[(fg)^{[n,m]}] = (\pi_n f^{[n,m]}) \times (\pi_0 g^{[n,m]}) + (\pi_0 f^{[n,m]}) \times (\pi_n g^{[n,m]})$$

This generalizes to

$$\pi_n(a \times b) = (\pi_n a) \times (\pi_0 b) + (\pi_0 a) \times (\pi_n b) \tag{18}$$

for any elements a and b of $DS(n,m)$. Thus, multiplication is defined by substituting Eq. (18) into

$$a \times b = (a_0 \times b_0, \pi_n(a \times b))$$

Note that Eq. (18) represents a generalization of the product rule that eliminates the need for Leibniz' rule. This is analogous to the use of the chain rule. In each case, the rule governing the first derivative is recursively applied to obtain higher derivatives. Note also that with this definition, we have a uniform recursive formulation of multivariate automatic differentiation. However, the recursive form of multiplication would obviously be much less efficient than Eq. (8). For each term ct with a nontrivial binomial coefficient c in the earlier definition, the recursive definition would have to sum c identical terms t.

7 Implementation

We built a trial implementation of our automatic differentiation system in Lisp. The language was selected for its ease of development and because it provides features that support list

manipulation and recursion in a natural way. In this section we will describe some of the aspects of the trial implementation. For completeness, we will first briefly review lisp.

Lisp is an interactive language. The fundamental operation is evaluation of expressions. Each expression to be evaluated is a list, of which the first element is the name of a function, and the remaining elements are the arguments to the function. Thus, for example, the expression `(+ 3 5)` is evaluated to obtain 8.

Derivative structures are defined to be lists. A level one derivative structure is simply a list of numbers, in the form (1 4 0 9), for example. A level two structure has lists as its elements. Here is an example: ((1 4 0 9) (1 3 7) (-2 1) (9)). Naturally, Lisp provides a number of functions for manipulating lists. Three that are particularly relevant here are `first`, `rest`, and `cons`. The `first` operation returns a copy of the first entry in a list, `rest` returns a copy of the list with the first entry omitted, and `cons` creates a new list by adding a new first element to a copy of an existing list. As an example, if `x` is the list `(6 2 9)`, then the function call `(first x)` returns 6, `(rest x)` returns `(2 9)`, and `(cons (first x) (rest x))` returns a copy of `x`.

Our automatic differentiation system is actually a suite of function definitions that augment the standard lisp functions. To use the system, one enters expressions to be evaluated. Here is an example:

```
(setq x (DS-make-var 3 2 1 17.2))
(setq y (DS-make-var 3 2 2 12.5))
(setq z (DS-make-var 3 2 3 11.8))
(DS* (DSexp x) (DS+ y z))))
```

This shows only what the user would type, not what the lisp system would respond. The three invocations of `setq` are assignments of variable names. The function `DS-make-var` is one of our automatic differentiation functions. It creates a derivative structure for an independent variable using four arguments: the level index, the order index, the number of the variable being assigned a value, and the assigned value. This is implemented essentially as in Eq. (5). Thus, the first three expressions above create derivative structures for three independent variables and derivatives through the second order, assigns variable names of `x`, `y`, and `z`, and assigns numerical values to those variables of 17.2, 12.5, and 11.8. The last statement above calls for a calculation. The function names are standard functions prefixed with `DS`. Thus `DS*` is derivative structure multiplication, `DSexp` is the derivative structure exponential function, and `DS+` is derivative structure addition. The complete calculation results in the derivative structure for $e^x(y+z)$. The result of evaluating this last expression will contain the numerical values of all partial derivatives through the second order at the point $(x, y, z) = (17.2, 12.5, 11.8)$.

As an alternative to interactively executing each step, we can combine the steps into a function. The lisp function `defun` is provided for that purpose. Evaluation of the expression

```
(defun f (x-val y-val z-val)
  (let (
                (x (DS-make-var 3 2 1 x-val))
                (y (DS-make-var 3 2 2 y-val))
                (z (DS-make-var 3 2 3 z-val))
               )
            (DS* (DSexp x) (DS+ y z))))
```

produces a function called `f`. It takes three numerical arguments referred to in the expression as `x-val`, `y-val`, and `z-val`. The function uses these as the values for the derivative structures of `x`, `y`, and `z`, and then calculates and returns the derivative structure for $e^x(y+z)$. Once the `defun` expression has been evaluated, the earlier computation can be performed by evaluating the expression `(f 17.2 12.5 11.8)`. This permits functions to be defined and then used repeatedly in combination with other functions. For example, we might also evaluate the expression `(DSsqrt (f 17.2 12.5 11.8) )` to obtain all the partial derivatives of $\sqrt{e^x(y+z)}$.

The preceding remarks should give some insight into how the prototype automatic differentiation system is used. To conclude this section, we make a few remarks regarding the implementation of the system. The code itself is included in an appendix.

As mentioned, the prototype system is a suite of functions. We will consider three classes of those functions, concerned with three types of derivative structure operation:

- initialization
- arithmetic
- elementary functions

The function `DS-make-var` is an example of an initialization function. It creates a derivative structure for one of the functions I_j. The only other initialization function is `DS-make-const` which creates a derivative structure for a constant. The definitions of these functions are direct implementations of the recursive definitions given in the discussion surrounding Eq. (5).

The arithmetic functions are addition, subtraction, multiplication, and scalar multiplication. These are coded just as defined in Eqs. $(6-9)$.

The elementary functions are defined using a general function based on Theorem 5. The general function is called `DSrecur`. It calls the projections π_0 and π_n, implemented as the lisp functions `DSpi-0` and `DSpi-n`. These are defined recursively, essentially as given in Definition (2). Actually, `DSpi-n` is just the `rest` function.

The function `DSrecur` takes four arguments, a derivative structure and pointers to three functions, and evaluates the identity given in Theorem 5. Here is a version of the code that has been edited slightly for improved readability. The actual code can be found in the appendix.

```
(defun DSrecur (a real-f f df)
  (if
   (number? a) (apply real-f a)
   (else  (let* (
             (a0   (first a))
             (pi0a (DSpi-0 a))
             (pina (DSpi-n a))
             (fa0  (apply f a0))
             (dfa  (apply df pi0a))
            )
        (cons fa0 (DS* dfa pina))))))
```

The function pointers are `real-f`, `f`, and `df`. The function operates as follows. If the argument `a` is a number (as opposed to a derivative structure), `(number? a)` evaluates to be true. In this case, the function `real-f` is applied to the number `a`, the result is returned, and that concludes `DSrecur`. Thus, `real-f` is the real variable function that is to be used at the last stage of the recursion. In the case `a` is not a number, the formula in Theorem 5 is evaluated. The `let*` function assigns local variable names to several intermediate results: the first entry of `a`, the projections π_0(`a`) and π_m(`a`), the result of applying the function `f` to `a0`, and the result of applying the derivative function `df` to π_0(`a`). What is finally returned is the result from the `cons` function in the last line. This will just be a list whose first element is $f(a_0)$ and whose remaining entries are computed as the derivative structure product of $f'(\pi_0(\texttt{a}))$ with $\pi_m(\texttt{a})$.

To illustrate how `DSrecur` is used, consider the following definition:

```
(defun DSexp (a) (DSrecur a 'exp 'DSexp 'DSexp))
```

This defines the derivative structure exponential function. What happens when the expression `(DSexp a)` is evaluated? It calls `DSrecur` with the argument `a`, and supplying a pointer to

the standard exponential function, as well as two pointers to itself. If `a` is a number, `DSrecur` simply applies the exponential function and returns the result. That is the value then returned by `DSexp`. In any other case, `DSrecur` performs the calculation indicated in Theorem 5. Note that this involves two recursive calls to `DSexp` – one with an argument of `a0` which is at a lower level than `a`, and one with an argument of `pi0a` which is of lower order. We consider a few more examples to further illustrate how simple it is to add basic functions to the prototype system using the `DSrecur` function.

The reciprocal function is defined in three lines. The first defines the base level reciprocal of numbers, the second defines the derivative of the reciprocal, and the third uses `DSrecur` to define the reciprocal. As before, the code has been edited for readability, and the actual code is in the appendix

```
(defun real-recip (a) (if (not= a 0) (divide 1 a) else nil))
(defun DSdrecip (a) (DS* -1 (DSrecip (DS* a a))))
(defun DSrecip (a) (DSrecur a 'real-recip 'DSrecip 'DSdrecip))
```

As a final example, the reciprocal function is used in the definition of the squareroot function.

```
(defun real-sqrt (a) (if (> a 0) (sqrt a) else nil))
(defun DSdsqrt (a) (DSrecip (DS* 2 (DSsqrt a))))
(defun DSsqrt (a) (DSrecur a 'real-sqrt 'DSsqrt 'DSdsqrt))
```

In the trial implementation we have defined all the arithmetic operations, together with the exponential, sine, cosine, and squareroot functions. Including the projection operations, the constructors for constants and variables, and functions for extracting particular partial derivatives from a derivative structure, the package is expressed in about 170 lines of code. This package automatically computes partial derivatives up to any order for functions of any number of variables composed of arithmetic operations and applications of the exponential, sine, cosine, and squareroot functions. Implementation and testing required less than three days.

8 Discussion

Throughout this paper, the point of the exposition has been to derive an extremely simple recursive foundation for automatic differentiation involving several variables. While we are not

aware of previous work in which recursion is applied to the number of variables, our use of recursion on order is but a minor modification of the approach presented by Neidinger [11]. In contrast, Flanders [3] and Lawson [8] use explicit representations of the polynomials P_j. These approaches require elaborate notation and an involved formulation. Moreover, they appear to impose limitations. For example, Flanders mentions the memory problems associated with storing the polynomials for high order derivatives, and Lawson defers the application of his method to third and higher order derivatives pending further design efforts. Our development provides an alternate solution to that of Flanders in the composite function problem. It would be interesting to try applying our methods in the inverse function problem considered by Lawson. Other related problems where our approach might be applicable are considered in Chapters 15 and 16 of [4].

We have not emphasized issues related to implementation and performance, and a detailed consideration of these issues is beyond the scope of this work. Nevertheless, the issue of performance has obvious importance. The very notion of propagating all the partial derivatives throughout the computation is subject to examination. Many applications may not require all the partial derivatives, and it is obviously inefficient in such cases to compute them all. Moreover, there are ways to reconstruct the full set of partial derivatives of a function from univariate taylor expansions for the restrictions of the function to a set of lines [2]. That scheme requires storing more data than the set of partial derivatives, but the computational complexity of propagating a series of univariate expansions provides a vast savings over the convolutions inherent in the derivative structure product defined here.

The compactness of our approach, and the simplicity of its implementation make it an attractive alternative for systems that are not driven by performance constraints. It is a natural question whether a recursive implementation can perform competitively with previous approaches, for example, that of Neidinger. As mentioned earlier, Neidinger's system is essentially identical to our own in so far as the structures and computations are concerned. However, the operations in Neidinger's approach are defined in terms of explicit iteration and subscripting, and his implementation is consequently more complicated. It would be interesting to see what performance penalty, if any, is imposed by the concise simplicity of the recursive implementation. In the following discussion, we will suggest some directions for a future investigation of efficiency.

One obvious way to improve the performance of the system is to store Taylor coefficients rather than partial derivatives, as this eliminates the need for binomial coefficients and saves a few operations in derivative structure multiplication. This approach has been discussed by several authors, see for instance [4, chapters 2 and 14], [10]. Examination of our recursive definitions reveals that multiplication is of central importance, and is therefore a first target for

optimization. Thus, the use of Taylor coefficients seems highly desirable. However, the relationships between differentiation and the projections π_r would be disturbed by this modification, so some care may be indicated.

The recursions on both order and level generally lead to unnecessary recomputation of intermediate results. Here are very simple examples related to each type of recursion. For the recursion on order, consider the lisp definition for `DSexp` in the preceding section. It generates a series of expansions according to the following scheme

$$\begin{aligned} e^{(a_0,a_1,\cdots,a_m)} &= (e^{a_0}, (e^{(a_0,a_1,\cdots,a_{m-1})} \times (a_1,\cdots,a_m)) \\ &= (e^{a_0}, e^{a_0} \times a_1, e^{a_0} \times (a_1^2 + a_2), \cdots) \end{aligned}$$

Each appearance of e^{a_0} is a recalculation of the same result. The redundancy illustrated in this example reflects a simple algebraic relationship between the exponential function and its derivative. Because the elementary functions all have derivatives which can be expressed in terms of other elementary functions, it is inevitable that this same kind of redundancy will occur for any elementary function if the order is sufficiently large.

As an example of redundant calculation in the recursion on level, consider the natural logarithm. The recursive formulation will take the form

$$\log(a) = (\log(a_0), \frac{1}{\pi_0 a} \times \pi_n a)$$

As the recursion proceeds, a reciprocal will be computed independently for π_0 of one component of a at every level. In each case, the reciprocal calculation will recompute the reciprocals from each lower order. This type of redundancy reflects the commutativity of the projection π_0 with the reciprocal operation. If $1/\pi_0(a)$ is computed once at the highest level, the result will contain each reciprocal required at lower levels. In general, the projection π_0 commutes with all the elementary functions: $\phi(\pi_0 a) = \pi_0 \phi(a)$. Therefore, a similar computational redundancy must occur in the level recursion for all the elementary functions.

Some of the effects of this inefficiency might be offset using a memoization scheme [12]. Without modification, this technique could be used to eliminate redundant calls to a function that occur in the course of the recursion. However, the technique would not combine computations that are common to substructures in several different parts of the recursion. It seems likely that some modification of memoization might be able to improve efficiency for those computations as well. It might also be interesting to implement automatic differentiation in an environment that provides so called *lazy* evaluation. For example, this is a common feature of functional programming [5] languages. Indeed, a specialization of this idea to automatic differentiation is one of the features of an efficient approach called *reverse mode* [4, chapter

1]. Our interest here would be to take advantage of lazy evaluation as a service provided by the programming language, without special treatment by the programmer. Perhaps this will lead to computational savings without compromising the simplicity of the recursive formulation of automatic differentiation. Both memoization and lazy evaluation are areas for future investigation.

Another approach for improving efficiency is to explicitly provide mechanisms for reusing results of intermediate calculations. For example, by defining a recursion on order that uses two derivatives (rather than one), some of the redundant calculation might be avoided. It also might be possible to propagate some of the intermediate results forward in the recursion to be reused at higher levels. The challenge in these approaches will be to develop efficient methods without making too great a sacrifice in simplicity.

There is something a little curious about the way the ideas developed in this paper were first conceived. Therefore, as a final topic, we briefly describe the circumstances leading up to this paper. We have been developing software in C++ for univariate automatic differentiation, and designed a hierarchy of classes that implement first arrays (that is, subscripted structures), then vectors (arrays with vector operations defined), and finally derivative vectors (vectors with the multiplication and elementary functions described above, specialized to the case $n = 1$). This development is very general, permitting a single section of code to be used to define arrays of a variety of entry types, and similarly for vectors and derivative vectors. In particular, having defined arrays of numbers, it is possible to use these as the elements of arrays to obtain two dimensional arrays. In the same way, having defined vectors of numbers, one can use these as the elements of vectors to generate matrices. The operations of addition and scalar multiplication are then obtained at no additional expense. It will be seen from this background that the question would naturally arise, what does one obtain if derivative vectors are formed using derivative vectors as the entries, and using derivative vector operations defined on these entries? This paper is the result of answering that question. It is quite common that theoretical insights contribute to the formulation of efficient coding practices. But in this case, the reverse happened. The efficient coding practice of striving for maximal generality inspired the theoretical insight that nesting derivative structures provides a recursive foundation for multivariate automatic differentiation.

A Appendix: Proofs of Selected Theorems

Theorem 2 consists of three identities. We will repeat each identity followed by its proof.

$$\partial_r(f^{[n,m]}) = (\partial_r f)^{[n,m]} \tag{19}$$

Proof:

When $n = 0$, the conclusion of the theorem is $\partial_r f = \partial_r f$. Proceeding by induction, assume the result for all levels up to n. Then we have for component j of $\partial_r(f^{[n,m]})$

$$\begin{aligned}
\partial_r(f^{[n,m]})_j &= \partial_r {\partial_n}^j f^{[n-1,m-j]} \\
&= {\partial_n}^j \partial_r f^{[n-1,m-j]} \\
&= {\partial_n}^j (\partial_r f)^{[n-1,m-j]} \\
&= [(\partial_r f)^{[n,m]}]_j
\end{aligned}$$

where the next to the last expression was obtained using the induction hypothesis. This shows that $\partial_r(f^{[n,m]}) = (\partial_r f)^{[n,m]}$ as desired. □

$$\pi_r f^{[n,m]} = \partial_r(f^{[n,m-1]}) \tag{20}$$

Proof:

When $n = r = 1$, both $\pi_r f^{[n,m]}$ and $\partial_r(f^{[n,m-1]})$ equal $(\partial_1 f, {\partial_1}^2 f, \cdots, {\partial_1}^m f)$. For $n > 1$ we consider two cases. First, suppose $r = n$. By definition,

$$\begin{aligned}
\pi_n(f^{[n,m]}) &= \pi_n((f^{[n-1,m]}, \partial_n f^{[n-1,m-1]}, \cdots, {\partial_n}^m f^{[n-1,0]})) \\
&= (\partial_n f^{[n-1,m-1]}, \cdots, {\partial_n}^m f^{[n-1,0]}) \\
&= \partial_n(f^{[n-1,m-1]}, \cdots, {\partial_n}^{m-1} f^{[n-1,0]}) \\
&= \partial_n f^{[n,m-1]}
\end{aligned}$$

In the second case, $r < n$, the argument proceeds by induction. We start again with the definition of π_r, apply Eq. (19), and then the induction hypothesis.

$$\begin{aligned}
\pi_r(f^{[n,m]}) &= (\pi_r(f^{[n-1,m]}), \pi_r(\partial_n f^{[n-1,m-1]}), \cdots, \pi_r({\partial_n}^{m-1} f^{[n-1,1]})) \\
&= (\pi_r(f^{[n-1,m]}), \pi_r((\partial_n f)^{[n-1,m-1]}), \cdots, \pi_r((\partial_n{}^{m-1} f)^{[n-1,1]})) \\
&= (\partial_r f^{[n-1,m-1]}, \partial_r(\partial_n f)^{[n-1,m-2]}, \cdots, \partial_r(\partial_n{}^{m-1} f)^{[n-1,0]})
\end{aligned}$$

Now change the order of differentiation using Eq. (19) again.

$$\begin{aligned}
\pi_r(f^{[n,m]}) &= ((\partial_r f)^{[n-1,m-1]}, \partial_n(\partial_r f)^{[n-1,m-2]}, \cdots, {\partial_n}^{m-1}(\partial_r f)^{[n-1,0]}) \\
&= (\partial_r f)^{[n,m-1]}
\end{aligned}$$

A final application of Eq. (19) yields the desired identity

$$\pi_r(f^{[n,m]}) = \partial_r f^{[n,m-1]}$$

completing the proof. □

$$\pi_0(f^{[n,m]}) = f^{[n,m-1]} \tag{21}$$

Proof:

For $n = 1$, the conclusion follows directly from the definitions of $f^{[1,m]}$ and π_0. Therefore, assume the result for levels less than n, and consider $f^{[n,m]}$. As in the preceding proof, we may express this as

$$f^{[n,m]} = (f^{[n-1,m]}, (\partial_n f)^{[n-1,m-1]}, \cdots, (\partial_n{}^n f)^{[n-1,0]})$$

Thus, applying the definition of π_0 and the induction hypothesis,

$$\begin{aligned} \pi_0(f^{[n,m]}) &= (\pi_0(f^{[n-1,m]}), \pi_0(\partial_n f)^{[n-1,m-1]}, \cdots, \pi_0(\partial_n{}^{m-1} f)^{[n-1,1]}) \\ &= (f^{[n-1,m-1]}, (\partial_n f)^{[n-1,m-2]}, \cdots, (\partial_n{}^{m-1} f)^{[n-1,0]}) \\ &= f^{[n,m-1]} \end{aligned}$$

This completes the proof. □

Next, we establish

$$f^{[n,m]} \times g^{[n,m]} = (fg)^{[n,m]} \tag{22}$$

A preliminary lemma will simplify the proof of Eq. (22), and is interesting itself as an extension of the product rule.

Lemma 2 *Suppose that* $a : \mathbf{R}^p \to DS(n, m_1)$ *and* $b : \mathbf{R}^p \to DS(n, m_2)$ *are differentiable functions. Then, for any integer* r *with* $1 \le r \le p$,

$$\partial_r(a \times b) = (\partial_r a) \times b + a \times (\partial_r b)$$

Proof:

For $n = 0$, the proposed condition is simply the product rule. For a fixed n assume the

result for $DS(n-1,m)$ for all m, and, with a and b defined as in the statement of the lemma, consider $\partial_r(a \times b)$. The j^{th} component of this expression is given by

$$\begin{aligned}
\partial_r(a \times b)_j &= \partial_r \sum_{k=0}^{j} \binom{j}{k} a_k \times b_{j-k} \\
&= \sum_{k=0}^{j} \binom{j}{k} \partial_r(a_k \times b_{j-k}) \\
&= \sum_{k=0}^{j} \binom{j}{k} [(\partial_r a_k) \times b_{j-k} + a_k \times (\partial_r b_{j-k})]
\end{aligned}$$

The last expression reflects the application of the induction hypothesis to a_k and b_{j-k}. These take values in $DS(n-1, m_1-k)$ and $DS(n-1, m_2-j+k)$, respectively. Since these are at level $n-1$, the induction hypothesis applies. To complete the proof, separate the previous expression into two sums, and use the definition of multiplication once again:

$$\begin{aligned}
\partial_r(a \times b)_j &= [\sum_{k=0}^{j} \binom{j}{k} (\partial_r a_k) \times b_{j-k}] + [\sum_{k=0}^{j} \binom{j}{k} a_k \times (\partial_r b_{j-k})] \\
&= [(\partial_r a) \times b]_j + [a \times (\partial_r b)]_j
\end{aligned}$$

This shows that $\partial_r(a \times b) = (\partial_r a) \times b + a \times (\partial_r b)$, and completes the proof. □

Observe that once we have the product rule for functions into $DS(n,m)$, we obtain Leibniz' rule as well. That is, for any j,

$$\partial_r{}^j(a \times b) = \sum_{k=0}^{j} \binom{j}{k} \partial_r{}^k a \times \partial_r{}^{j-k} b$$

This is a general property of derivations and can be proved by induction on j in the usual way. We will make use of it in the following proof of Eq. (22).

Proof:

As before, we proceed by induction. For $n = 0$, the identity reduces to $fg = fg$. Assume the result holds for levels less than n. We begin with $f^{[n,m]} \times g^{[n,m]}$ and use the definitions of $f^{[n,m]}$, $g^{[n,m]}$, and multiplication in $DS(n,m)$ to obtain

$$\begin{aligned}
(f^{[n,m]} \times g^{[n,m]})_j &= \sum_{k=0}^{j} \binom{j}{k} (f^{[n,m]})_k \times (g^{[n,m]})_{j-k} \\
&= \sum_{k=0}^{j} \binom{j}{k} (\partial_n{}^k f^{[n-1,m-k]}) \times (\partial_n{}^{j-k} g^{[n-1,m-j+k]})
\end{aligned}$$

Now reduce the order of each term in the sum to the minimum order that appears, that is, to $m - j$. By the version of Leibniz' rule given above, the resulting expression is equal to ${\partial_n}^j(f^{[n-1,m-j]} \times g^{[n-1,m-j]})$. The induction hypothesis transforms this last expression into ${\partial_n}^j(fg)^{[n-1,m]}$, which is the definition of the j^{th} component of $(fg)^{[n,m]}$. □

B Appendix: Lisp Code for Prototype System

```
(defun DS-make-const (n m &optional (value 0))
  "  Creates a constant element of DS(N,M), zero by default
  or equal to the optional VALUE "
  (cond
   ((or (< n 0) (< m 0)) nil)
   ((and (= m 0) (= n 0)) value)
   ((= m 0) (list (DS-make-const (- n 1) m value)))
   ((= 1 n) (append (DS-make-const n (- m 1) value) (list 0)))
   (t (cons (DS-make-const (- n 1) m value) (DS-make-const n (- m 1))))))

(defun DS-make-var (n m var-num &optional (value 0))
  "Make an element of DS(n,m) for the VAR-NUM th variable"
  (cond
   ((< m 1) nil)
   ((< n var-num) (DS-make-const n m))
   ((> n var-num) (cons (DS-make-var (- n 1) m var-num value)
                        (DS-make-const n (- m 1))))
   ((= n 1) (cons value (cons 1 (DS-make-const 1 (- m 2)))))
   ( t (let* ((a0 (DS-make-const (- n 1) m value))
              (a1 (DS-make-const (- n 1) (- m 1) 1)))
         (cons a0 (cons a1 (DS-make-const n (- m 2))))))))

(defun DSpi-m (a)
  "Projection that deletes car of the list"
  (cdr a))

(defun DSpi-0 (a)
  "Projection pi 0 removes outermost elements of A"
  (cond
```

```
   ((not a) nil)
   ((not (listp a)) nil)
   (t (let* ((a0 (car a))
             (len (length a))
             (len-1 (- len 1))
             (pi0a0 (DSpi-0 a0)))
        (cond
         ((not (listp a0)) (sublist a len-1))
         (pi0a0 (cons (DSpi-0 a0) (DSpi-0 (cdr a))))
         (t nil))))))

(defun DS-extract (a dlist)
  "Assuming first argument A is a derivative structure, return the
entry specified by second argument DLIST.  The car of DLIST specifies
one element of the list A, the next element of DLIST specifies which element
of that element of A, etc.  Or put another way, if DLIST is (d1 d2 d3 ...)
then the value returned is the partial derivative resulting from d1
differentiations relative to the last variable, d2 differentiations relative
to the next to the last variable, etc."
  (cond
   ((not dlist) a)
   ((not (and a (listp a))) nil)
   (t (DS-extract (nth (car dlist) a) (cdr dlist)))))

(defun DS+ (a b &rest args)
  (if args (DS+ a (apply 'DS+ b args))
    (let* ((a-is-number (not (listp a)))
           (b-is-number (not (listp b))))
      (cond
       ((not (and a b)) nil)
       ((and a-is-number b-is-number) (+ a b))
       (a-is-number (DS-add-value b a))
       (b-is-number (DS-add-value a b))
       (t (cons (DS+ (car a) (car b)) (DS+ (cdr a) (cdr b))))))))

(defun DS- (a b)
  (let* ((a-is-number (not (listp a)))
         (b-is-number (not (listp b))))
    (cond
```

```
     ((not (and a b)) nil)
     ((and a-is-number b-is-number) (- a b))
     (a-is-number (DS-add-value (DS* -1 b) a))
     (b-is-number (DS-sub-value a b))
     (t (cons (DS- (car a) (car b)) (DS- (cdr a) (cdr b)))))))

(defun DS* (a b &rest args)
  (if args (DS* a (apply 'DS* b args))
    (let* ((a-is-number (not (listp a)))
           (b-is-number (not (listp b))))
      (cond
       ((not (and a b)) nil)
       ((and a-is-number b-is-number) (* a b))
       ( a-is-number (cons (DS* a (car b)) (DS* a (cdr b))))
       ( b-is-number (cons (DS* b (car a)) (DS* b (cdr a))))
       (t (let* ((a-order (- (length a) 1))
                 (b-order (- (length b) 1))
                 (result-order (min a-order b-order))
                 (result nil)
                 (counter result-order))
            (loop while (>= counter 0)
                  do (progn (setq result (cons (bdot a b counter) result))
                            (setq counter (- counter 1))))
            result))))))

(defun bdot (a b n)
  (if (= n 0) (DS* (car a) (car b))
    (let* ((a-sub (sublist a (+ n 1)))
           (b-rev (reverse (sublist b (+ n 1))))
           (k 0))
      (accum-bdot a-sub b-rev k n))))

(defun accum-bdot (a b k n)
  "accumulates a sum of list elements of A and B multiplied by
   binomial coefficients"
  (if (= k n) (DS* (car a) (car b))
    (DS+
     (DS* (nCm n k) (DS* (car a) (car b)))
     (accum-bdot (cdr a) (cdr b) (+ k 1) n))))
```

```
(defun DS/ (a b)
  (DS* a (DSrecip b)))

(defun DSrecur (a real-f f df &rest args)
  "General purpose recursion shell for auto diff for the function f.
First arg is A, the DS which is being operated on.  Second arg REAL-F
is the procedure to perform if A is real.   Args F and DF are
the function and its derivative. The remaining args list ARGS is provided
to permit parameters in function definitions.  For example, an integer
power function can be defined with the power as a parameter."
  (cond
   ((not a) nil)
   ((not (listp a)) (apply real-f a args))
   (t (let* ((a0   (car a))
             (pi0a (DSpi-0 a))
             (pima (DSpi-m a))
             (fa0  (apply f a0 args))
             (dfa  (apply df pi0a args)))
        (cons fa0 (DS* dfa pima))))))

(defun real-recip (a) (if (/= a 0) (/ 1 a) nil))
(defun DSdrecip (a) (DS* -1 (DSrecip (DS* a a))))
(defun DSrecip (a) (DSrecur a 'real-recip 'DSrecip 'DSdrecip))

(defun real-sqrt (a) (if (>= a 0) (sqrt a) nil))
(defun DSdsqrt (a) (DSrecip (DS* 2 (DSsqrt a))))
(defun DSsqrt (a) (DSrecur a 'real-sqrt 'DSsqrt 'DSdsqrt))

(defun DSexp (a) (DSrecur a 'exp 'DSexp 'DSexp))
(defun DSsin (a) (DSrecur a 'sin 'DSsin 'DScos))
(defun DSdcos (a) (DS* -1 (DSsin a)))
(defun DScos (a) (DSrecur a 'cos 'DScos 'DSdcos))

(defun real-log (a) (if (< 0 a) (log a) nil))
(defun DSlog (a) (DSrecur a 'real-log 'DSlog 'DSrecip))


(defun sublist (list n)
```

```
  " Create new list from LIST using first N elements."
  (cond
   ((<= (length list) n) list)
   ((= 0 n) nil)
   (t (cons (car list) (sublist (cdr list) (- n 1))))))

(defun nCm (n m)
  "Binomial coefficient"
  (cond
   ((= m 0) 1)
   ((= n m) 1)
   (t (+ (nCm (- n 1) (- m 1)) (nCm (- n 1) m)))))

(defun DS-update-value (op ds val)
  "Use operation OP to update the value entry of the DS by VAL"
  (let ((ds0 (car ds)))
    (if (listp ds0) (DS-update-value op ds0 val)
      (rplaca ds (apply op ds0 val nil))))
  ds)

(defun DS-add-value (ds val) (DS-update-value '+ ds val))
(defun DS-sub-value (ds val) (DS-update-value '- ds val))
```

References

[1] V. I. Arnold. *Geometrical Methods in the Theory of Ordinary Differential Equations, 2nd Ed.*, Springer-Verlag, New York, 1988.

[2] Christian Bischof, George Corliss, and Andreas Griewank. Structured Second- and Higher-Order Derivatives through Univariate Taylor Series. Preprint MCS-P296-0392, Argonne National Laboratory, Argonne, Illinois, May 1992.

[3] Harley Flanders. Automatic Differentiation of Composite Functions. in Andreas Griewank and George F. Corliss, Eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, 95 – 99, SIAM, Philadelphia, 1991.

[4] Andreas Griewank and George F. Corliss, Eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[5] John Hughes. Why Functional Programming Matters. in David A. Turner, Ed. *Research Topics in Functional Programming*, 17 – 942, Addison-Wesley, Reading, Massachusetts, 1990.

[6] Ronald E. Huss. An Ada library for automatic evaluation of derivatives. *Applied Mathematics and Computation*, 35:103 – 123, 1990.

[7] Dan Kalman and Robert Lindell. Automatic Differentiation in Astrodynamical Modeling. in Andreas Griewank and George F. Corliss, Eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, 228 – 241, SIAM, Philadelphia, 1991.

[8] Charles L. Lawson. Automatic Differentiation of Inverse Functions. in Andreas Griewank and George F. Corliss, Eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, 87 – 94, SIAM, Philadelphia, 1991.

[9] Richard D. Neidinger. Automatic differentiation and apl. *College Mathematics Journal*, 20(3):238 – 251, May 1989.

[10] Richard D. Neidinger. Computing Multivariate Taylor Series Coefficients to Arbitrary Order. Presentation: *International Congress for Industrial and Applied Mathematics*, Washington, D.C., July 8, 1991.

[11] Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. *ACM Transactions on Mathematical Software*, 18(2):159 – 173, June 1992.

[12] Peter Norvig. *Paradigms of Artificial Inteligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann, San Mateo, California, 1992.

[13] Peter J. Olver. *Applications of Lie Groups to Differential Equations*, Springer-Verlag, New York, 1986.

[14] Louis B. Rall. Applications of software for automatic differentiation in numerical computation. Technical Report 1976, Mathematics Research Center, Madison, Wisconsin, July 1979.

[15] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science No. 120. Springer–Verlag, Berlin, 1981.

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

[17] Louis B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59(5):275 – 282, December 1986.