Doubly Recursive Multivariate Automatic Differentiation

DAN KALMAN
American University
Washington, D.C. 20016
kalman@american.edu

Automatic differentiation is a way to find the derivative of an expression without finding an expression for the derivative. More specifically, in a computing environment with automatic differentiation, you can obtain a numerical value for $f'(x)$ by entering an expression for $f(x)$. The resulting computation is accurate to the precision of the computer system—it does not depend on the approximation of derivatives by difference quotients. Indeed, the computation is equivalent to evaluating a symbolic expression for $f'(x)$, but no one has to find that expression—not even the computer system.

That's right. The automatic differentiation system never formulates a symbolic expression for the derivative. Automatically calling on something like *Mathematica* to produce a symbolic derivative, and then plugging in a value for x is the wrong image entirely. Automatic differentiation is something completely different.

Well OK, but so what? Symbolic algebra systems are so prevalent and powerful today, why should we be concerned with avoiding symbolic methods? There are two answers. The first is practical. Symbolic generation of derivatives can lead to exponential growth in the length of expressions. That causes computational problems in real applications. Accordingly, there is a practical applied side to the subject of automatic differentiation, as witnessed by the serious attention of computer scientists and numerical analysts [**3, 4**].

The second answer is more mathematical. It is a relatively easy task to create a single variable automatic differentiation system capable of evaluating first derivatives. In fact, writing in this MAGAZINE in 1986, Rall [**10**] gives a beautiful presentation of just such a system. What is mathematically interesting is an amazingly elegant extension of the one-variable/one-derivative system that handles essentially any number of variables and derivatives. The extension is recursively defined, employing an induction on both the number of variables and the number of derivatives, and using fundamental definitions that are virtually identical to the ones used in Rall's system.

The purpose of this paper is to present the recursive automatic differentiation system. To set the stage, we will begin with a brief review of Rall's one-variable/one-derivative system, followed by an example of the recursive system in action. Then the mathematical formulation of the recursive system will be presented. The paper will end with a brief discussion of practical issues related to the recursive system.

Rall's system

Because automatic differentiation is a computational technique, it is best understood in the context of a computer language. In particular, recall that in a scientific computer language such as Basic, or FORTRAN, variables correspond to memory locations. For example, consider the statements

$$x = 3$$
$$f = x^2 - 5.$$

The first causes a value of 3 to be stored in the memory location for x, while the second reads the value of x, squares it, subtracts 5, and stores the result in the memory location for f. We can think of this as a procedure for evaluating the function $f(x) = x^2 - 3$.

In Rall's system, the idea is to simultaneously evaluate both $f(x)$ and $f'(x)$. In this system, each variable corresponds to an ordered pair of memory locations, one for the value of a function, and one for the value of the derivative. Now the goal is for the statements above to produce the pair $(4, 6)$, incorporating the values of both $f(3)$ and $f'(3)$.

This is accomplished as follows. First, when a variable is assigned a value in a statement such as $x = 3$ the automatic differentiation system stores in the memory for x the pair $(3, 1)$. This corresponds to the value of the identity function $I(x) = x$, and its derivative, at $x = 3$. Second, any numerical constant that appears in an expression is represented by a pair corresponding to the value and derivative of a constant function. For the example above, the constant 5 is represented by $(5, 0)$—the value of the constant function $C(x) \equiv 5$, and its derivative. Finally, each operation appearing in the expression is carried out in an extended sense, operating on pairs. The rule for pair addition or subtraction is just the usual componentwise operation. The rule for pair multiplication is

$$(a_1, a_2) \times (b_1, b_2) = (a_1 b_1, a_2 b_1 + a_1 b_2). \tag{1}$$

Using these definitions, we can anticipate what the automatic differentiation system will do in response to the pair of statements

$$\begin{aligned} x &= 3 \\ f &= x^2 - 5. \end{aligned}$$

The first statement leads to the creation of the pair (3,1). The second statement translates into a sequence of operations on pairs:

$$\begin{aligned} f &= (3, 1) \times (3, 1) - (5, 0) \\ &= (3 \cdot 3, 1 \cdot 3 + 3 \cdot 1) - (5, 0) \\ &= (9, 6) - (5, 0) \\ &= (4, 6). \end{aligned}$$

As easily verified, this result correctly represents the value of both $x^2 - 5$ and its derivative at $x = 3$. Notice that there is no symbolic computation here. However, the equivalent of symbolic differentiation rules are built into the definitions of pair addition and multiplication. Thus, the expression for f is evaluated to produce both the value of the expression and of its derivative.

It should be stressed that the operations on pairs can be formulated without any reference to functions and derivatives. We adopt an abstract framework with objects (ordered pairs) and operations. As defined above, ordered pairs can be added, subtracted, and multiplied. In fact, extended operations for pairs can be defined for all the usual elementary functions. For example, the sine of a pair is defined according to

$$\sin(a_1, a_2) = (\sin a_1, a_2 \cos a_1). \tag{2}$$

Of course, these abstract definitions are inspired by the idea that each ordered pair will contain values of a function and its derivative. To make the connection explicit, we will use the notation $f^{[1,1]}(x) = (f(x), f'(x))$, where the $[1, 1]$ indicates the presence of one variable, and the inclusion of one derivative. Thus, in the original computation, we found $f^{[1,1]}(3) = (4, 6)$. Similarly, using the sine operation for pairs, the statements

$$\begin{aligned} x &= 3 \\ g &= \sin(x^2 - 5) \end{aligned}$$

result in the computation of $\sin(4, 6) = (\sin 4, 6\cos 4)$. The elements of this ordered pair are the correct values of $\sin(x^2 - 5)$ and its derivative at $x = 3$. That is, with $g(x)$ defined as $\sin(x^2 - 5)$, the lines above compute $g^{[1,1]}(3)$.

What makes the system work is that each operation correctly propagates derivative values. For the arithmetic operations, that means

$$\begin{aligned} f^{[1,1]}(3) + g^{[1,1]}(3) &= (f + g)^{[1,1]}(3) \\ f^{[1,1]}(3) - g^{[1,1]}(3) &= (f - g)^{[1,1]}(3) \\ f^{[1,1]}(3) \times g^{[1,1]}(3) &= (fg)^{[1,1]}(3). \end{aligned} \tag{3}$$

Observe that the rules for addition, subtraction, and products of pairs are based on the sum and product rules for derivatives. Similarly, (2) is really nothing more than the chain rule, since the derivative of $\sin(f(x))$ is given by $\cos(f(x))f'(x)$. With a_1 in place of $f(x)$ and a_2 in place of $f'(x)$, this becomes $\cos(a_1)a_2$. That shows that in (2), if $(a_1, a_2) = f^{[1,1]}(3)$, then $\sin(a_1, a_2) = \sin(f^{[1,1]}(3)) = (\sin \circ f)^{[1,1]}(3)$. In a similar way, any differentiable function ϕ can be extended to pairs by the formula

$$\phi(a_1, a_2) = \left(\phi(a_1), \phi'(a_1)a_2\right). \tag{4}$$

With this definition, we have

$$\phi(f^{[1,1]}) = (\phi \circ f)^{[1,1]}. \tag{5}$$

Although these examples pertain to a function of a single variable, and involve only a single derivative, it is easy to envision extensions involving several variables and partial derivatives of various orders. Throughout, we will restrict our attention to functions sufficiently smooth so that order of differentiation does not matter.

In the recursive system that we will present below, the idea is to compute all of the partial derivatives up to some specified order. In this system, evaluating a function f at a point in its domain means determining an object $f^{[n,m]}$ that contains the function value as well as the values of all partial derivatives through order m with respect to n variables. These objects are referred to as *derivative structures*. Since m defines the maximum number of derivatives, it is called the derivative index. Similarly, n is the variable index. As in the discussion above, we can proceed abstractly by defining derivative structures and appropriate operations without any mention of functions and derivatives. However, given a function f, we do need some way to construct $f^{[n,m]}$ as one of our abstract derivative structures, and equations analogous to (3) and (5) must hold.

The recursive system in action

Before describing the abstract system, let's take a look at how the system operates. Consider the function

$$f(x, y, z) = \frac{\sqrt{x + y}}{\sqrt{z - y}},$$

and suppose we wish to evaluate f and all partial derivatives through second order at the point (4, 5, 14). The recursive automatic differentiation system can be given this problem with the following commands (with slightly modified syntax for readability):

```
x = DS-Make-Var(3,2,1,4)
y = DS-Make-Var(3,2,2,5)
z = DS-Make-Var(3,2,3,14)
u = DS-Sqrt(DS-Add(x,y))
v = DS-Sqrt(DS-Sub(z,y))
Print DS-Divide(u,v)
```

These commands involve applications of several different functions within the automatic differentiation system. First, there are three invocations of `DS-Make-Var`. This function creates the derivative structures corresponding to the independent variables x, y, and z. For example, `x = DS-Make-Var(3,2,1,4)` creates a derivative structure for **3** variables, and for partial derivatives through order **2**, corresponding to variable number **1** (x), and assigning that variable a value of **4**. This command is the equivalent of $x = 4$ in the one-variable/one-derivative system. Similarly, the next two statements create the derivative structures corresponding to variables y and z, assigning values of 5 and 14, respectively. The other commands are the derivative structure versions of standard operations; `DS-Add` is addition of derivative structures, `DS-Sqrt` applies the square root for derivative structures, and so on. So the fourth statement adds the derivatives structures for `x` and `y` and takes the square root of the result. That defines a new derivative structure, `u`. Similarly, the next line defines `v` by subtracting `y` from `z`, and applying the derivative structure for square roots. The final command applies derivative structure division to `u` and `v`, and prints the result.

As in Rall's system, the computations above are completely numerical. For example, the derivative structure for the variable x stores the value of x, 4, as well as all the partial derivatives through second order with respect to x, y, and z. These values are, of course, trivially determined. The partial derivative with respect to x is 1, and all the other partial derivatives are 0. But the point is that the derivative structure called `x` is just some sort of array with entries of 4, 1, and many zeroes. In the same way, `y` and `z` are arrays of numbers as well. When these are combined according to the commands listed above, the final result is printed out as

```
0.01235
0.11111  0.00000                 −0.01235
1.00000  0.05556  −0.00309       −0.05556  −0.00309       0.00926.
```

These are the values of f and its derivatives, in the following arrangement:

$$\begin{array}{llllll}
f_{yy} & & & & & \\
f_y & f_{xy} & & f_{yz} & & \\
f & f_x & f_{xx} & f_z & f_{xz} & f_{zz}.
\end{array}$$

The subscripts indicate partial differentiation: f_x for $\frac{\partial f}{\partial x}$, f_{xy} for $\frac{\partial^2 f}{\partial y \partial x}$, and so on. The rationale for laying out the derivatives in this way will become clear when the general system is defined. For this example, it is enough to see how the system operates, and to observe that all the desired partial derivatives are correctly computed.

At this point, I hope that the basic idea of the automatic differentiation system is clear. Numerical values for a function and its derivatives are arranged in some sort of data structure, and operations on these structures are defined according to the rules of differentiation so that derivatives are correctly propagated. The structures for the simplest functions, namely the constant functions (like $c(x, y, z) \equiv 5$) and variables (like

$I_1(x, y, z) \equiv x$) are easy to specify directly. By operating on these simple derivative structures, we can formulate derivative structures for essentially arbitrary expressions involving the variables and elementary functions.

Although these ideas are feasible in principle, I also hope the reader has some sense of the difficulty of handling all the details in practice. At first glance, the idea of defining appropriate structures to contain all the partial derivatives through second order relative to three variables, and then specifying the proper operations of arithmetic, as well as proper definitions for functions like sine and cosine, should seem fairly intimidating, or at least unpleasantly tedious. Happily, and surprisingly, there is a remarkably simple recursive formulation that is no more complicated than Rall's one-variable/one-derivative system. Indeed, considered formally, the operations within this recursive formulation are virtually identical to the operations in Rall's system. With that in mind, let us turn now to the recursive development of an automatic differentiation system.

The objects

The first step in constructing the recursive system is to define the objects, or derivative structures, on which we will operate. Let us consider a few motivating examples. First, for functions of a single variable, automatic calculation of m derivatives can be provided by operating on $(m+1)$-tuples. A typical object in the system, $a = (a_0, a_1, \cdots, a_m)$, includes the value of a function and its first m derivatives. For example, with $m = 3$, we can write

$$a = f^{[1,3]} = (f, f_x, f_{xx}, f_{xxx}).$$

For a function of two variables, assuming equality of mixed partials, the partial derivatives through order m are conveniently arranged in a triangular array. This is illustrated in FIGURE 1 for $m = 3$. It is important to note that the entry in the lower left-hand corner has a special significance. In the derivative structure $f^{[2,m]}$, the lower left-hand corner is the value of the original function f.

$$\begin{array}{llll}
f_{yyy} & & & \\
f_{yy} & f_{yyx} & & \\
f_y & f_{yx} & f_{yxx} & \\
f & f_x & f_{xx} & f_{xxx}
\end{array}$$

Figure 1 Layout of $f^{[2,3]}$

Observe that the array in FIGURE 1 can be decomposed into two parts. The bottom row is a vector of derivatives with respect to a single variable, as described in the preceding paragraph. That is, the bottom row is just $f^{[1,3]}$. The second part, all of the triangle *except* the bottom row, is also a derivative structure, namely ${f_y}^{[2,2]}$; it contains the value of f_y, and all of its first and second order partial derivatives with respect to x and y. This gives $f^{[2,3]}$ as a combination of $f^{[1,3]}$ and ${f_y}^{[2,2]}$.

In a similar way, we can lay out the entries of $f^{[3,3]}$, that is, the partial derivatives through third order with respect to three variables (see FIGURE 2). The partial derivatives are arranged in a pyramid composed of several triangular layers. Each layer has the same form as the triangular array in FIGURE 1. As before, there is a distinguished entry identifying the function f, at the lower left-hand corner of the lowest level. Also, as before, there is a natural decomposition into two parts. The first part is the bottom triangular array, which is recognizable as $f^{[2,3]}$. It contains all partial derivatives through order $m = 3$ with respect to x and y. The complementary part is the sub-pyramid made up of levels 2, 3, and 4. This can be recognized as $f_z^{[3,2]}$. It contains all partial derivatives relative to the three variables x, y, and z, through order 2 of the function f_z. The decomposition gives $f^{[3,3]}$ as a combination of $f^{[2,3]}$ and $f_z^{[3,2]}$.

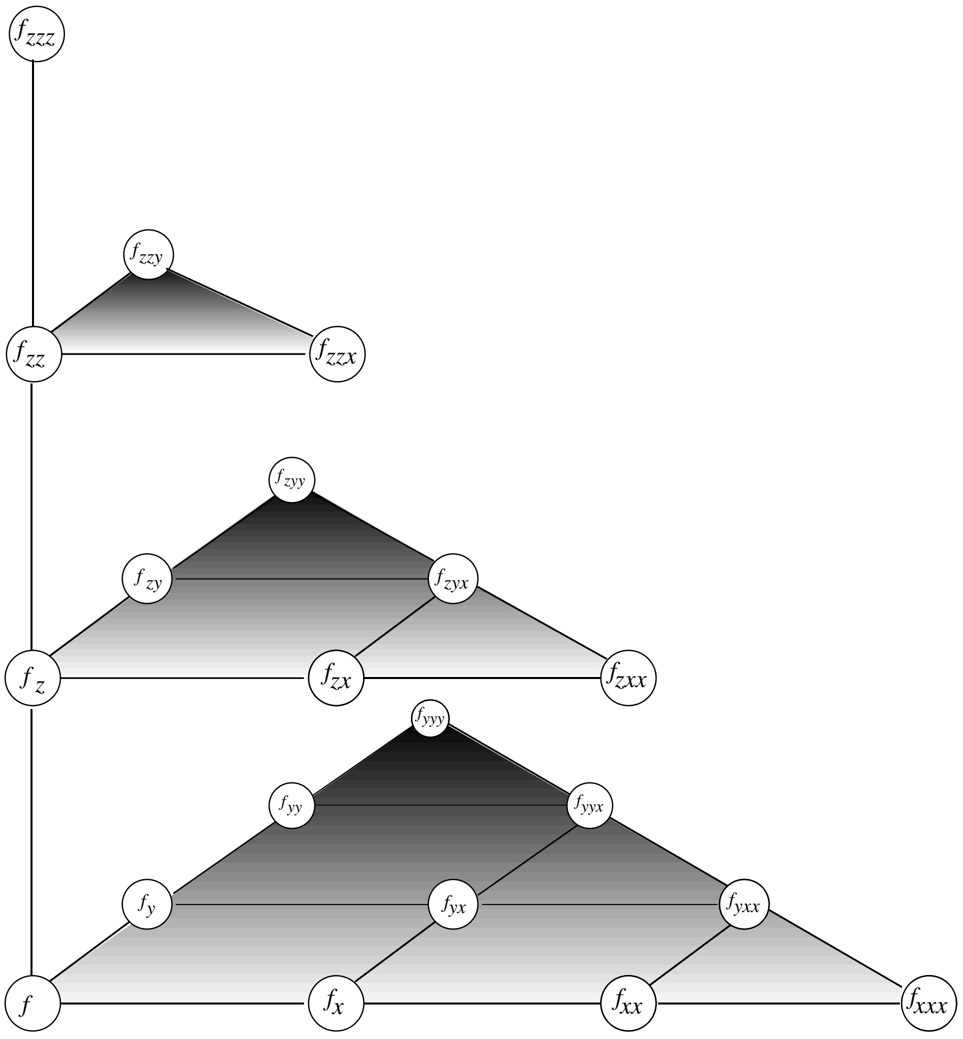

Figure 2 Layout of $f^{[3,3]}$

These examples suggest a hierarchy of automatic differentiation objects. For any n and m, we can imagine a set of objects that contain all partial derivatives through order m with respect to n variables. These will be our *derivative structures*. Thus, for a single variable we have derivative vectors; for two variables, derivative triangles; for three variables, derivative pyramids; and in general, derivative structures.

The decomposition discussed in the examples above can be described in general using the terminology of derivative structures. For each example we considered, a derivative structure of partial derivatives through order m with respect to n variables was partitioned into two smaller derivative structures. The first part had the same number of derivatives (m) and one fewer variables ($n - 1$) than the original structure, while the second part had one fewer derivatives ($m - 1$) and the same number of variables as the original. These observations inspire the following recursive definition of derivative structures.

DEFINITION 1. *For $m, n \geq 0$, we define $DS(n, m)$, the set of derivative structures with derivative index m and variable index n, as follows. If $m = 0$ or $n = 0$, $DS(n, m)$ is just $\mathbb{R}$, the real numbers. Otherwise*

$$DS(n, m) = DS(n - 1, m) \times DS(n, m - 1)$$

(where $\times$ denotes the Cartesian product).

It should be emphasized here that this definition makes no mention of functions or derivatives. It abstractly defines a class of objects, built up recursively, and reducing to real numbers at the lowest level of the recursion. In this context, a derivative structure is understood most simply as a binary tree, with real numbers as the leaves. An element $a \in DS(4, 7)$, for example, has two components, one in $DS(3, 7)$ and the other in $DS(4, 6)$. Each of these components likewise has two components, as shown in FIGURE 3. Each branch of the tree ends when one of the two indices reaches zero, indicating that the corresponding component is a real number. For $a = f^{[n,m]}$, the real numbers at the leaves are simply the values of partial derivatives of f. However, this visualization turns out to be of limited value. Instead, the best approach is to retain the recursive image of an element of $DS(n, m)$ as an ordered pair, each of whose components is a lower order derivative structure.

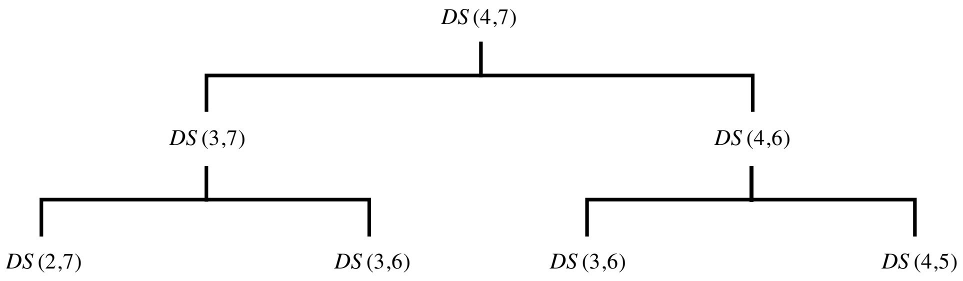

Figure 3 Partial Tree for $a \in DS(4, 7)$.

The idea of a derivative structure as an ordered pair hints at the connection to Rall's automatic differentiation system. Shortly we will see that the definitions for operations on derivative structures make this connection into a perfect analogy. But there is one final prerequisite needed. In terms of the triangular arrays and pyramids considered earlier, the two components of a derivative structure are particular substructures. For example, if $a = (a_1, a_2)$ is a derivative pyramid, then a_1 is a derivative triangle, and a_2 is a smaller derivative pyramid. We also need a third substructure, denoted a_1^*. Later an abstract recursive definition of a_1^* will be provided. But conceptually, think of a_1^* as follows: If the derivative structure $a = f^{[n,m]}$, then it contains within it $f^{[n,m-1]}$, the derivatives up to order $m - 1$. That substructure is a_1^*. Thus, in FIGURE 1, a_1 is the bottom row, a_2 is the sub-triangle consisting of everything but the bottom row, and a_1^* is the triangle that contains everything except the third order derivatives lying along the hypotenuse. Notice that a_2 and a_1^* have the same size and shape, but are derivative structures for different functions. Similarly, in FIGURE 2, the triangle on the lowest level is a_1, the remaining levels form the sub-pyramid a_2, and a_1^* is the sub-pyramid consisting of everything except the highest order derivatives lying on the slanting outer face of the pyramid.

This completes the background we need to define derivative structure operations. We know that a derivative structure a is an ordered pair (a_1, a_2), that the components are derivative substructures of lower order, and that a_1^* is another sub-structure with the same size and shape as a_2. The operations on derivative structures are defined in terms of these substructures.

Operations on derivative structures

To build expressions out of derivative structures, we need to be able to apply arithmetic operations and elementary functions. By considering the reciprocal function $r(x) = 1/x$ as one of our elementary functions, we eliminate the need to define derivative structure division. To divide a/b we simply multiply $a \times r(b)$. Accordingly, the only arithmetic operations that we need are addition, subtraction, and multiplication. As a convenience we will also include scalar multiplication.

The definitions of all the arithmetic operations are recursive. The case of addition, subtraction, and scalar multiplication will make this clear.

DEFINITION 2. *For $DS(0, m)$ and $DS(n, 0)$, the elements are real numbers and addition, subtraction, and multiplication are the usual real number operations. For $n, m > 0$, let $a = (a_1, a_2)$ and $b = (b_1, b_2)$ be elements of $DS(n, m)$, and let r be a real number. Then addition, subtraction, and scalar multiplication are defined by*

$$\begin{aligned} a + b &= (a_1 + b_1, a_2 + b_2) \\ a - b &= (a_1 - b_1, a_2 - b_2) \\ ra &= (ra_1, ra_2). \end{aligned}$$

Formally, these are identical to the componentwise definitions in Rall's system. But they have a slightly different meaning in the present context. To add a and b we must add their components, which are themselves derivative structures. The computer implementation of the addition is thus recursive. To add two elements of $DS(3, 4)$, for example, we recall the addition operation for components in $DS(3, 3)$ and in $DS(2, 4)$. Those additions, in turn, spawn additions of more derivative structures. At each recursion, though, one of the two indices is reduced. Eventually, an index becomes zero, and the recursion terminates with an addition of real numbers. Subtraction and scalar multiplication operate similarly.

The definition of multiplication is again an analog of what we saw in Rall's system.

DEFINITION 3. *For $DS(0, m)$ and $DS(n, 0)$ multiplication is defined to be the usual real number operation. For $n, m > 0$, if $a = (a_1, a_2)$ and $b = (b_1, b_2)$ are elements of $DS(n, m)$, define*

$$a \times b = (a_1 \times b_1, a_2 \times b_1^* + a_1^* \times b_2).$$

Formally, this is virtually identical to the one-variable/one-derivative multiplication rule defined by (1). The only difference is that there are no asterisks in (1). Indeed, the ordered pairs in Rall's systems are elements of $DS(1, 1)$, and in that setting, a_1 and a_1^* are identical. However, while there are clear formal similarities between multiplication in Rall's system and in $DS(n, m)$, it must be remembered that in the latter system the definition is recursive. As for the operations of addition, subtraction, and scalar multiplication, the multiplication of derivative structures requires multiplying their components, and hence a recursive use of multiplication. And as we saw earlier, the recursive process keeps generating more and more multiplications, finally reaching a point at which the derivative objects reduce to real numbers. So, while the multiplication definition seems to have the same simplicity as in Rall's system, under the surface there is a complex sequence of operations implicitly defined.

Finally we come to the elementary functions. Given a derivative structure a and an elementary function ϕ, we wish to define $\phi(a)$. Once again, the definition is almost identical to what appeared in Rall's system.

DEFINITION 4. *Let ϕ be an m-times differentiable function of a real variable. If $m = 0$ or $n = 0$, $DS(n, m)$ is just $\mathbb{R}$ and ϕ is applied to the elements in the usual way. For $n, m > 0$, if $a = (a_1, a_2) \in DS(n, m)$, define*

$$\phi(a) = \big(\phi(a_1), \phi'(a_1^*) \times a_2\big).$$

This definition is a direct analog of (4), to which it reduces in the case that $n = m = 1$. As we saw with multiplication, the only formal difference is the appearance of an asterisk in the general derivative structure definition. Here again, the actual computation of $\phi(a)$ is recursive, and the recursion terminates when ϕ or one of its derivatives is finally called upon to operate on a real number.

That's it. That is all you need to construct arbitrary elementary function expressions involving general derivative structures. As promised, the definitions are virtually the same as those in Rall's system, and yet they provide for the automatic generation of partial derivatives to essentially arbitrary order with respect to an essentially arbitrary number of variables. But the presentation is not quite complete. We still have to see how to create the fundamental derivative structures that correspond to constants and variables. And at some point we need to see why the definitions just given really work.

Fundamental derivative structures

So far, we have defined derivative structures and their operations abstractly, without mention of functions and partial derivatives. To make the connection with automatic differentiation clear, we must have a definition of $f^{[n,m]}$ as an element of $DS(n, m)$.

DEFINITION 5. *Let f be a function of at least n variables with continuous partial derivatives through order m, and let x be an element of the domain of f. Then the derivative structure for f with derivatives through order m with respect to the first n variables is given at x by*

$$f^{[n,m]}(x) = \begin{cases} f(x) & \textit{if } n = 0 \textit{ or } m = 0 \\ \big(f^{[n-1,m]}(x), (\partial_n f)^{[n,m-1]}(x)\big) & \textit{otherwise} \end{cases}$$

where ∂_n denotes partial differentiation with respect to the nth variable of f.

This definition is a formalization of the pattern we saw in special cases, but some caution is needed. How do we know that $f^{[n,m]}$, as defined here, really does contain all the partial derivatives it is supposed to? For now the reader is asked to accept the validity of the definition. We will return to the justification in the next section.

Given the preceding definition, we can construct derivative structures for constants recursively. For example, to create the derivative structure for the constant 5, we consider the constant function $f(x, y, z, \ldots) \equiv 5$. Now $f^{[n,m]}$ has two components. The first is $f^{[n-1,m]}$, and that can be constructed recursively. The second is $\partial_n f^{[n-1,m]}$, and since f is constant, the partial derivative is 0. But that is again a constant function. Thus, a recursive construction algorithm can operate similarly to the operation algorithms. To construct a constant in $DS(n, m)$, we must first construct constants in

$DS(n-1, m)$ and $DS(n, m-1)$. The recursion proceeds until one index becomes 0, and at that point the value of the constant is returned. That constant is 5 just once, corresponding to tracing the left branch all the way down the tree to a leaf. In any path that involves a right branch, the function will be differentiated at least once, and it will be a zero function that is finally evaluated. On some level, however, this image is irrelevant. All that really matters is that a simple recursive construction algorithm for constants exists in the automatic differentiation system.

To illustrate the situation for the independent variables, let's consider the function $I_2(x, y, z) = y$. How do we construct ${I_2}^{[3,2]}$ at $y = 8$, for example? At the top level, ${I_2}^{[3,2]}$ is an ordered pair. The first component is ${I_2}^{[2,2]}$, which will be constructed recursively. The second component is $\partial_z {I_2}^{[3,1]}$, and since $\partial_z y = 0$ that is just the derivative structure of the constant 0. It can be constructed using the algorithm for a constant. At the next level, ${I_2}^{[2,2]}$ is decomposed into ${I_2}^{[1,2]}$ and $\partial_y {I_2}^{[2,1]}$. For the first of these, notice that the first index is 1. This is a derivative structure that does not involve any derivatives with respect to y, and for its construction we can treat y as the constant 8. For the second component, $\partial_y I_2 = \partial_y y = 1$. Again we need only construct a derivative structure for a constant. In a similar way, the derivative structure for any of the independent variables can be constructed recursively. Indeed, ${x_j}^{[n,m]} = (a_1, a_2)$ is defined as follows: If $j < m$, then a_1 is defined by a recursive construction of ${x_j}^{[n,m-1]}$ and a_2 is a derivative structure for the constant 0. If $j = m$, then a_1 is constructed as a constant derivative structure, with whatever value was assigned to x_j, and a_2 is the derivative structure for the constant 1. And if $j > m$, a_1 is again a constant derivative structure with the value of x_j, but a_2 is the derivative structure of the constant 0.

This is the construction used to define `DS-Make-Var` in the sample computation presented earlier. In fact, if you review that computation, you will see that we have now defined every operation that appears there. The automatic differentiation system is complete. With algorithms for constructing derivative structures for independent variables and constants, and definitions of derivative structure operations and elementary functions, nothing more is needed. However, we have yet to see any verification that the system actually works. How do we know, for example, that the arithmetic definitions propagate derivatives correctly? How do we know that applying an elementary function to a derivative structure as in Definition 4 produces the desired derivative information at the end? For that matter, how do we even know that the recursive definition for $f^{[n,m]}$ is correct? The next section will address these questions.

Validation of the system

There are two aspects of the system that require validation. First, we have to verify that the recursive definition of $f^{[n,m]}$ properly represents the intuition suggested by the triangle and pyramid examples. Second, it must be established that the definitions of derivative structure operations correctly propagate derivative information. That is, we must see that

$$\begin{aligned} f^{[n,m]} + g^{[n,m]} &= (f+g)^{[n,m]} \\ f^{[n,m]} - g^{[n,m]} &= (f-g)^{[n,m]} \\ f^{[n,m]} \times g^{[n,m]} &= (fg)^{[n,m]} \end{aligned} \tag{6}$$

and

$$\phi(f^{[n,m]}) = (\phi \circ f)^{[n,m]}. \tag{7}$$

For both of these ends, expressing a derivative structure a as an ordered pair (a_1, a_2) and referring to the components and to a_1^* will be of central importance. It simplifies the presentation to express these substructures using an operator notation. Thus, if $a = (a_1, a_2)$ is a derivative structure, we define $V(a) = a_1$ and $D(a) = a_2$. The names of these operators reflect the meaning of the components in the one-variable/one-derivative system, where a_1 is the *value* of the function, and a_2 is the *derivative*. Recall that a_1^* is obtained from a by removing all the highest order derivatives, so that a_1^* is a *lower order* version of a. Accordingly, we use the notation $L(a) = a_1^*$.

Although the conceptual meaning of the operators is clear, formal definitions will be given for completeness. For L, this is particularly important as there has not yet been given an abstract definition in terms of derivative structures.

DEFINITION 6. *Let $a \in DS(n, m)$. If $n = 0$ or $m = 0$, a is a real number and $V(a)$, $D(a)$, and $L(a)$ are all defined to equal a. Otherwise, $a = (a_1, a_2)$. In this case, we define $V(a) = a_1$, $D(a) = a_2$, and $L(a)$ according to*

$$L(a) = \begin{cases} L(a_1) & \text{if } m = 1 \\ (L(a_1), L(a_2)) & \text{if } m > 1. \end{cases}$$

It may not be immediately apparent that this definition of L is consistent with the earlier explanation of a_1^*. The reader may wish to verify that the definition works correctly for triangles and pyramids. However, for the arguments that will follow, it is not logically necessary to connect the definition of L with the conceptual image of $f^{[n,m]}$. Instead, we will be content to take $L(a)$ as the *definition* of a_1^*, and show that this definition has the properties we need for automatic differentiation.

The three operators provide the means to connect the abstract definition of $DS(n, m)$ to the ideas illustrated by the derivative vectors, triangles, and pyramids. As a first instance of this, we have the following result.

THEOREM 1. *Derivative structures for functions are related to the operations V, D, and L as follows:*

$$\begin{aligned} V(f^{[n,m]}) &= f^{[n-1,m]} \\ D(f^{[n,m]}) &= (\partial_n f)^{[n,m-1]} \\ L(f^{[n,m]}) &= f^{[n,m-1]}. \end{aligned}$$

If the derivatives in $f^{[n,m]}$ are laid out as in the examples of triangles and pyramids, these identities are obvious. However, it is possible to prove the identities using only the abstract definitions of the operators and of $f^{[n,m]}$. In fact, the first two identities are immediate consequences of the abstract definition of $f^{[n,m]}$. The third identity can be proved by a straightforward induction argument that exploits the recursive definitions of both $f^{[n,m]}$ and L. This same style of proof is effective for a number of the results to follow, and while a detailed proof for the third identity above will not be given, a sample proof will be given for a later theorem. In any case, it is important to note that the induction proof uses only the abstract definitions of L and $f^{[n,m]}$, and so makes no direct use of the full image of how partial derivatives are laid out in $f^{[n,m]}$. Thus, the fact that the third identity can be established by an abstract proof confirms that, at least in this regard, L and $f^{[n,m]}$ operate according to expectation.

Theorem 1 lends itself to a simple formal algorithm for applying V, D, or L to $f^{[n,m]}$: V decrements the variable index by 1; L decrements the derivative index by 1; and D both decrements the derivative index and differentiates f once with respect to the nth variable. Using just the first two of these rules we can prove the next result.

THEOREM 2. *Suppose $f^{[n,m]}$ is defined at x. Let e_j be a nonnegative integer for $1 \le j \le n$ with $\sum e_j \le m$. Then the partial derivative $\partial_1^{e_1} \cdots \partial_n^{e_n} f(x)$ can be obtained from $f^{[n,m]}(x)$ as follows: If $\sum e_j = m$ then*

$$\partial_1^{e_1} \cdots \partial_n^{e_n} f(x) = D^{e_1} V D^{e_2} V \cdots V D^{e_n} f^{[n,m]}(x);$$

otherwise

$$\partial_1^{e_1} \cdots \partial_n^{e_n} f(x) = V D^{e_1} V D^{e_2} V \cdots V D^{e_n} f^{[n,m]}(x).$$

The proof is simply a matter of applying the identities in Theorem 1. Rather than present the details in a formal way, it will be more illuminating to work through an example. Consider the derivative structure $f^{[3,6]}$ and suppose we want to obtain $\partial_1^2 \partial_2 \partial_3^2 f(x)$. Since this is a fifth derivative and $m = 6$, the theorem says to compute $V D^2 V D V D^2 f^{[3,6]}$. We can verify that the desired result is obtained by applying the identities in Theorem 1 as follows:

$$\begin{aligned}
V D^2 V D V D^2 f^{[3,6]}(x) &= V D^2 V D V (\partial_3^2 f)^{[3,4]}(x) \\
&= V D^2 V D (\partial_3^2 f)^{[2,4]}(x) \\
&= V D^2 V (\partial_2 \partial_3^2 f)^{[2,3]}(x) \\
&= V D^2 (\partial_2 \partial_3^2 f)^{[1,3]}(x) \\
&= V (\partial_1^2 \partial_2 \partial_3^2 f)^{[1,1]}(x) \\
&= (\partial_1^2 \partial_2 \partial_3^2 f)^{[0,1]}(x) \\
&= \partial_1^2 \partial_2 \partial_3^2 f(x).
\end{aligned}$$

This example reveals the general nature of the algorithm for extracting a particular derivative from $f^{[n,m]}$. Notice that the D operator only performs differentiation of $f^{[n,m]}$ with respect to x_n. But each time we apply V, we reduce the value of n, and hence change the variable that D differentiates. If we want a certain number of derivatives with respect to x_n, we apply D that many times. Then we apply V, in effect, shifting the focus to x_{n-1}. If we want one or more derivatives with respect to x_{n-1}, we apply D that many times again. So we continue, alternately applying D to differentiate and V to shift to a new variable, until all the desired derivatives have been applied. For an mth derivative, there will be m applications of D, reducing the derivative index to 0, and so reducing the derivative structure to a real number. Otherwise, there will be exactly n applications of V. This will reduce the variable index to 0, and so again result in a real number.

It should be stressed again that the operators V and D were defined completely abstractly, with no reference to derivatives. In a computational system, a particular derivative structure is simply an organized network of memory locations which store real values. The algorithm above navigates through such a network to a particular entry. Theorems 1 and 2 show that when a derivative structure is constructed according to the abstract definition of $f^{[n,m]}$, the desired derivative values can all be located and extracted. More specifically, visualizing the network as a binary tree, each application of V selects a left branch from a node, each application of D selects a right branch, and after either m applications of D or n applications of V a terminal node is reached. Thus, Theorem 2 can be understood as a prescription for finding the appropriate terminal node for a particular partial derivative.

To complete the validation of the system, we must see that derivative structure operations really do succeed in constructing $f^{[n,m]}$. That is, we must verify (6) and (7). The formal statement is given in the following theorem.

THEOREM 3. *Let f and g be real valued functions of n or more variables, with continuous partial derivatives through order m, let x be in the domain of f and g, let r be a real number, and let ϕ be a real function m times differentiable at $f(x)$. Then the following identities hold:*

$$\begin{aligned} f^{[n,m]}(x) + g^{[n,m]}(x) &= (f+g)^{[n,m]}(x) \\ f^{[n,m]}(x) - g^{[n,m]}(x) &= (f-g)^{[n,m]}(x) \\ rf^{[n,m]}(x) &= (rf)^{[n,m]}(x) \\ f^{[n,m]}(x) \times g^{[n,m]}(x) &= (fg)^{[n,m]}(x) \\ \phi(f^{[n,m]}(x)) &= (\phi \circ f)^{[n,m]}(x). \end{aligned}$$

As mentioned earlier, the recursive nature of the definitions makes induction a natural approach to proving results like these. To illustrate, here is a proof of the final identity above. It assumes that the preceding identities have already been established.

Proof. The proof is by induction on $n+m$. If either n or m is zero, the conclusion holds trivially. So assume that both n and m are positive, and that the conclusion holds for $f^{[n',m']}$ whenever $n'+m' < n+m$. From the definition of ϕ for derivative structures, if $f^{[n,m]}(x)$ is expressed as the pair (a_1, a_2), then

$$\phi\left(f^{[n,m]}(x)\right) = \left(\phi(a_1), \phi'(a_1^*) \times a_2\right).$$

In terms of the V, D, and L operators, this becomes

$$\phi\left(f^{[n,m]}(x)\right) = \left(\phi\left(Vf^{[n,m]}(x)\right), \phi'\left(Lf^{[n,m]}(x)\right) \times Df^{[n,m]}(x)\right).$$

Applying Theorem 1 we obtain

$$\phi\left(f^{[n,m]}(x)\right) = \left(\phi\left(f^{[n-1,m]}(x)\right), \phi'\left(f^{[n,m-1]}(x)\right) \times (\partial_n f)^{[n,m-1]}(x)\right).$$

Now we are ready to use the induction hypothesis. On the right side of the preceding equation, the real functions ϕ and ϕ' are applied to derivative structures with lower order than $f^{[n,m]}(x)$. By induction, we can *bring ϕ and ϕ' inside their respective parentheses*, leading to

$$\phi\left(f^{[n,m]}(x)\right) = \Big((\phi \circ f)^{[n-1,m]}(x), (\phi' \circ f)^{[n,m-1]}(x) \times (\partial_n f)^{[n,m-1]}(x)\Big).$$

Similarly, the identity for derivative structure multiplication allows us to bring the product on the right side of the equation inside the parentheses. Performing that reduction and recognizing the normal real function chain rule then produces

$$\begin{aligned} \phi(f^{[n,m]}(x)) &= \Big((\phi \circ f)^{[n-1,m]}(x), (\phi' \circ f \cdot \partial_n f)^{[n,m-1]}(x)\Big) \\ &= \left((\phi \circ f)^{[n-1,m]}(x), [\partial_n(\phi \circ f)]^{[n,m-1]}(x)\right) \\ &= (\phi \circ f)^{[n,m]}(x). \end{aligned}$$

This shows that the identity holds for $f^{[n,m]}$, completing the induction argument. ∎

This concludes the validation of the recursively defined automatic differentiation system. It has been demonstrated that the simple recursive definitions for derivative structure operations properly propagate partial derivatives. To put it more simply, we have seen that the recursive automatic differentiation system *works*. In a final section, we discuss a few ideas connected with implementation and computational efficiency.

Implementation and efficiency

The recursive automatic differentiation system presented here can be implemented in any computer programming language that supports recursion. A working version is described in [**6**]. There, the interested reader will find LISP code for the entire system, amounting to about 150 lines. Although the presentation in [**6**] is from a different point of view than the double recursion described here, the LISP code can be considered an implementation of either point of view. In fact, the double recursion described here was discovered as a direct result of studying the implementation in [**6**]. It should also be mentioned that the original idea for treating the number of derivatives recursively is due to Neidinger [**8**]. His work provided a critical inspiration for both the approach of [**6**] and the double recursion presented here.

It is beyond the scope of this paper to discuss the LISP implementation in detail. However, there is one aspect that is worth considering. The programming for the automatic differentiation system must include derivative structure formulations for all the familiar elementary functions: exponential, sine, cosine, etc. Each of these is programmed according to Definition 4. Interestingly, this definition can be implemented quite generally, and then used to create the procedures for all the desired elementary functions. The basic idea is to define a procedure that will combine the original function ϕ, the derivative ϕ', and the derivative structure a to compute $\phi(a)$. For the sake of discussion, let us call the procedure `Compose`. It will take as arguments procedures `phi` and `phi-prime`, and a derivative structure `a`. If `a` is actually just a real value, `Compose` applies `phi` to `a` and returns the result. Otherwise, `Compose` uses the V, D, and L operators to compute `a1`, `a2`, and `a1*`, respectively. Then it applies `phi` to `a1`, `phi-prime` to `a1*`, and returns the ordered pair `(phi(a1), phi-prime(a1*) * a2)`.

All of the elementary functions are defined in terms of the procedure `Compose`. For example, here is what the definition of the derivative structure exponential function might look like:

```
Function DS-Exp(a)
  if a is real
     return exp(a)
  else
     return Compose(DS-Exp, DS-Exp, a)
  end
```

Note that `DS-Exp` plays the role of both `phi` and `phi-prime` in the call to `Compose`. Thus, the computation of `DS-Exp(a)` requires evaluations of `DS-Exp(a1)` and `DS-Exp(a1*)`. This is simply a direct implementation of the recursive nature of Definition 4. In a similar way, the reciprocal function is defined as follows:

```
Function DS-Recip(a)
  if a is real
     return 1/a
  else
```

```
    return Compose(DS-Recip, DS-DRecip, a)
  end
```

Here, DS-DRecip is a derivative structure function that plays the role of the derivative of the reciprocal function. That is, with $\phi(x) = 1/x$, the derivative is $\phi'(x) = -1/x^2$. This can be defined by

```
Function DS-DRecip(a)
   recip-a = DS-Recip(a)
   return -1 * recip-a * recip-a
```

And now that we have defined the reciprocal function, it is no problem to add the natural logarithm.

```
Function DS-Ln(a)
  if a is real
    return ln(a)
  else
    return Compose(DS-Ln, DS-Recip, a)
  end
```

As these examples suggest, the development of a complete automatic differentiation system requires very little programming, once the derivative structure operations are in place. For each elementary function that is included, the developer does have to explicitly specify the derivative. However, that is a small price to pay for the automatic generation of derivatives to essentially arbitrary order. And in any case, one cannot reasonably hope to avoid defining derivatives altogether in a system that is supposed to compute derivatives automatically. In comparison to other approaches to automatic differentiation for higher derivatives [**2, 7**], the development presented here is remarkably simple.

This simplicity streamlines the task of implementing an automatic differentiation system. How the system performs is quite another issue, and it turns out that the elegance of the recursive approach is accompanied by some significant sources of inefficiency. While we will not take up this issue in any significant way here, a few brief comments are in order.

A little reflection reveals that a naive implementation of the doubly recursive approach involves widespread recomputation of previously obtained results. To illustrate this idea, consider the third derivative of the product fg. We know by Leibniz' rule that

$$(fg)''' = f''' + 3f''g' + 3f'g'' + g'''.$$

This can be derived by repeatedly applying the product rule, and then algebraically simplifying the result. In particular, three different terms, each equal to $f''g'$, would appear, giving rise to the single term $3f''g'$ in Leibniz' rule. The recursive automatic differentiation system is similar to repeatedly applying the product rule without algebraic simplification. That would entail three separate evaluations of $f''g'$.

In contrast, Neidinger [**9**] has developed a multivariate automatic differentiation system that uses explicit looping and subscripting. This system avoids the recomputation that can arise in the recursion process, and should be expected to outperform a direct implementation of the design presented here.

Inspired by Neidinger's approach, there are obvious strategies for reducing some of the recursive approach's inefficiency. In particular, a carefully optimized multiplication procedure, based on Leibniz' rule rather than simple recursion, might make a significant impact. Another attractive idea is to identify and exploit redundant calculations in the recursion process. Yet another improvement would be to take advantage of sparseness, eliminating computations that ultimately lead to multiplication by zero. Whether a modified version of the recursive system would be competitive with Neidinger's system is a question for further study.

However, no matter what formulation is used, direct computation of all partial derivatives of an expression is simply not the fastest approach. A more efficient alternative is to use systems of univariate automatic differentiation computations and an interpolation scheme [**1**]. Although this does increase memory requirements, it is easily shown to produce huge reductions in execution for large scale systems. Thus, for example, in a system with several hundred variables and a need for third order partial derivatives, any direct computation of all partial derivatives would be much slower than the alternative using interpolation.

On the other hand, computational speed is not always an issue. An automatic differentiation system of the type described here has been used successfully in an interactive application for analyzing systems of constraints arising in the design of satellite systems. In that context, automatic differentiation was used to perform sensitivity analyses among dozens of variables. For this application, computation was limited by the speed of user input, not by the speed with which the automatic differentiation system operated. In that situation, the speed of the automatic differentiation system was of no concern at all.

More generally, as computational speed continues to increase, the importance of execution efficiency will continue to decline, particularly for problems with small numbers of variables. In these cases, the directness and simplicity of the current development offers an attractive paradigm for implementing an automatic differentiation system.

Acknowledgment. This paper is based on an invited address at the January 1997 AMS/MAA meeting [**5**].

REFERENCES

1. C. Bischof, G. Corliss, and A. Griewank, Structured second- and higher-order derivatives through univariate Taylor series, preprint MCS-P296-0392, Argonne National Laboratory, Argonne, Illinois, May 1992.
2. H. Flanders, Automatic differentiation of composite functions, in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 95–99.
3. A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2000.
4. A. Griewank and G. F. Corliss, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.
5. D. Kalman, Automatic differentiation: computing derivative values without derivative formulas, Invited address, Joint Meetings of the American Mathematical Society and the Mathematical Association of America, San Diego, January 1997.
6. D. Kalman and R. Lindell, Recursive multivariate automatic differentiation, *Optimization Methods and Software* **6** (1995), 161–192.
7. C. L. Lawson, Automatic differentiation of inverse functions, in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 87–94.
8. R. D. Neidinger, Automatic differentiation and APL, *College Math. J.* **20** (1989), 238–251.
9. R. D. Neidinger, An efficient method for the numerical evaluation of partial derivatives of arbitrary order, *ACM Transactions on Mathematical Software* **18** (1992), 159–173.
10. L. B. Rall, The arithmetic of differentiation, this MAGAZINE **59** (1986), 275–282.