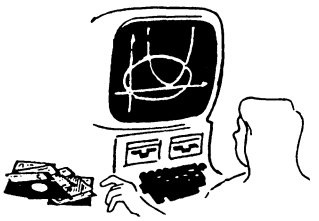


COMPUTER CORNER

Edited by
Eugene A. Herman

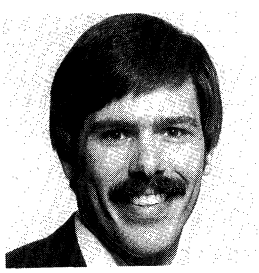
In this column, readers are encouraged to share their expertise and experiences with computers as they relate to college mathematics. Articles may illustrate how computers can be used to enhance pedagogy, solve problems, or model real-life situations. Readers are also invited to submit interesting (not necessarily original) algorithms in a structured language or pseudocode, with explanatory text to make the purpose and validity clear.

All manuscripts for this column should be prepared according to the guidelines on the inside front cover and sent to:

Eugene A. Herman
Department of Mathematics
Grinnell College
Grinnell, IA 50112

Automatic Differentiation and APL

Richard D. Neidinger



Richard Neidinger is an Assistant Professor of Mathematics at Davidson College in North Carolina. He received a B.A. from Trinity University in San Antonio and an M.A. and Ph.D. from the University of Texas at Austin. He came to Davidson in 1984 after completing the Ph.D. under the direction of Professor Haskell Rosenthal. His research interests focus on functional analysis, although he also enjoys applications of the computer.

Automatic differentiation is an unorthodox approach to differentiation, and APL is an unorthodox approach to computer programming; their combination yields a beautiful and powerful tool. By combining the formal rules of differentiation with numerical evaluations of successive derivatives as they are produced, automatic differentiation makes it possible to evaluate high-order derivatives of a function at a point with accuracy equal to that provided by symbolic differentiation, but at a fraction of the cost in computer time and space. The ability of APL to "think" in

terms of vector operators makes it an ideal language for the implementation of automatic differentiation. We shall develop several small programs that enable the user to evaluate (not just approximate) the derivatives up to order n of any elementary function (i.e., any function obtained by algebraic means from polynomials, logarithms, exponentials, trigonometric functions, and inverse trigonometric functions).

No previous experience with APL or automatic differentiation is required; in fact, each topic enhances the exposition of the other. The language APL will be explicated through solving this problem, for which it is so well-suited. The ideas of automatic differentiation will be discussed while a remarkably simple (and original) implementation is developed. The method of recursive evaluation of higher-order derivatives by using a vector operation that embodies Leibniz's rule is an extension of previous work. A good exposition of the mathematics of automatic differentiation (without actual implementation) is [5], which inspired this paper. To compare implementations of automatic differentiation in more conventional languages, see [6], [7], and [11]. All of the computer work was originally done on an IBM PC using the inexpensive Pocket APL system by STSC.

Let us introduce the idea of automatic differentiation with a problem. Suppose that we want to find the values of the function f and its first ten derivatives at the point $a = 1$, where $f(x) = (x \cdot \csc(x))/\ln(\arctan(\exp(x)))$. The idea of computing the symbolic derivatives by hand is absurd and, in fact, the task overwhelmed a standard commercial program designed for the purpose. Numerical methods (based on difference quotients) could be used to obtain approximate answers rather quickly, but the difficulty of obtaining reasonable accuracy (cf. [7]) grows rapidly with the order of the derivative. Anyway, our desire is to evaluate these derivatives with accuracy limited only by computer representation (roundoff) error. Automatic differentiation is a very different approach that is theoretically exact and computationally practical; indeed the above problem is solved in Section 2.

While symbolic differentiation uses "the rules" on symbols, automatic differentiation uses "the rules" with values. Calculations are performed in a system of vectors. An example illustrates the considerations that arise. Suppose $h(x) = \sin(u(x))$ and the vector $(u(a), u'(a), u''(a), \ldots, u^{(n)}(a))$ is known. We wish to compute $(h(a), h'(a), h''(a), \ldots, h^{(n)}(a))$. A typical calculus student could produce a formula for $h''(a)$ in terms of $u(a)$, $u'(a)$, and $u''(a)$. However, a general algorithm, to produce $h^{(n)}$ for an arbitrary n, involves recursive formulas and iterative programming. Section 2 develops such algorithms. Section 1 considers the case $n = 1$, where the overall structure of automatic differentiation is clear and an APL implementation is straightforward.

1. Calculating First Derivatives; An Introduction to APL

We wish to calculate the value of the derivative of an arbitrary function at an arbitrary point a. Every function is associated with a numerical vector as indicated in Table 1, where s denotes a constant scalar.

Table 1

function	vector at a
$h(x)$	$H = (h(a), h'(a))$
$u(x)$	$U = (u(a), u'(a))$
$v(x)$	$V = (v(a), v'(a))$
$\mathrm{id}(x) = x$	$X = (a, 1)$
$c(x) = s$	$C_s = (s, 0)$

The goal is to calculate H when given the formula for $h(x)$. Suppose that $h(x)$ is given as an arithmetic operation combining $u(x)$ and $v(x)$, and that U and V are known. Then, we need only define a corresponding vector operation on U and V that yields H. But this operation is determined by a differentiation rule! For example, if $h(x) = u(x) + v(x)$, then $H = U + V$. Similarly, if $h(x) = r \cdot u(x)$, then $H = rU$. If $h(x) = u(x) \cdot v(x)$, then $H = U$ TIMES V, where we define

$$U \text{ TIMES } V = \big(u(a) \cdot v(a), u'(a) \cdot v(a) + u(a) \cdot v'(a)\big).$$

This idea also applies to composition with transcendental functions. As an example, suppose that $h(x) = \sin(u(x))$ and that U is known. Then $H = \text{SIN}\, U$, where we define

$$\text{SIN}\, U = \big(\sin(u(a)), \cos(u(a)) \cdot u'(a)\big).$$

It is now obvious that such vector operators, with built-in chain rule, could be defined for all of the usual operations and functions of calculus.

Ultimately, the hypothesis that U and V are known rests on the last vectors in Table 1; X and C_s are known. The expression for any elementary function $f(x)$ can be translated into the corresponding vector operations ultimately applied to X or C_s. The result will be $F = (f(a), f'(a))$. Consider, for example, $f(x) = 7x + 3$. Using automatic differentiation,

$$F = 7X + C_3 = 7(a, 1) + (3, 0) = (7a + 3, 7),$$

which yields $f(a) = 7a + 3$ and $f'(a) = 7$. Or, take any typical formula, say $f(x) = \sin(x^2)$. Then $F = \text{SIN}(X \text{ TIMES } X)$. Indeed,

$$F = \text{SIN}(X \text{ TIMES } X) = \text{SIN}((a, 1) \text{ TIMES } (a, 1)) = \text{SIN}(a \cdot a, 1a + 1a)$$
$$= \text{SIN}(a^2, 2a) = \big(\sin(a^2), \cos(a^2) \cdot 2a\big).$$

This unusual method of evaluation would be practical in a computing environment featuring numerical vector calculations and the ability to define new vector operations. APL is A Programming Language characterized by easy vector and matrix calculations enabled by using a large and expandable library of functions or dyadic operators. In fact, all programming is thought of as adding functions or operators to the library. Work may then be done in the immediate execution mode.

Let's go to the APL immediate execution mode and observe vector addition and scalar multiplication. Our entries will be indented, and the computer's response will be at the left margin. A space separates elements in a vector.

```
      8 7 3 ¯2 + 4 1 9 7
12 8 12 5
      2 × 3 6 2
6 12 4
      X ← 3 1
      X + (2×X) + 7 0
16 3
      X[0]
3
      X[1]
1
```

The left arrow assigns the vector 3 1 to the global variable X. The next line calculates $(f(3), f'(3))$ for the function $f(x) = x + 2x + 7$. Elements of a vector are

referenced with an index starting with zero. This indexing was chosen (by APL command $\square IO \leftarrow 0$) so that $X[k]$ would refer to the kth derivative.

All of the many functions and operators built into the APL language are called by special character symbols. Most of the unusual symbols used in this paper are summarized in the appendix. Glance over this now and use it for reference. Because there are so many operators, absolutely no order priority is specified. All lines are executed from *right to left* with parentheses necessary to specify otherwise. This is a common point for errors, since it is fairly unusual.

Any new vector operator is defined in the function definition mode. The first del (an upside down delta) starts the definition and the second del ends the definition, returning to the immediate execution mode.

```
      ∇ H ← U TIMES V
[1]   H ← (U[0]×V[0]), (U[0]×V[1])+(U[1]×V[0])
[2]   ∇
      X ← 3 1
      X TIMES X
9 6


      ∇ H ← SIN U
[1]   H ← (1○U[0]), (2○U[0])×U[1]
[2]   ∇
      X ← 0 1
      SIN X
0 1
```

(The circle functions $1\circ$ and $2\circ$ represent sin and cos, respectively.)

To differentiate $f(x) = x^2$ at 3 by automatic differentiation, define the vector $X \leftarrow 3\ 1$ and perform the calculation $F = X$ TIMES X. (Alternatively, x^2 could be handled by defining a vector operator U POWER r.) Thus the above computation XTIMES X yields $(f(3), f'(3))$. Similarly, the above SIN X produces $(f(0), f'(0))$ for the function $f(x) = \sin(x)$.

In general, given a function $f(x)$, we define a function FDF(a) that returns the vector $F = (f(a), f'(a))$. The point of evaluation is variable. For example, consider $f(x) = x^2 - 7x$.

```
      ∇ F ← FDF a
[1]   X ← a,1
[2]   F ← (X TIMES X) - 7×X
[3]   ∇
      FDF 5
¯10 3
      FDF ¯2
18 ¯11
```

In order to change the function, we use "∇FDF[2]" which means "redefine line 2 of the previous FDF definition." Line 1 remains the same. For example, we evaluate f and f' for $f(x) = \sin(x^2)$.

```
      ∇ FDF[2]
[2]   F ← SIN (X TIMES X)
[3]   ∇
      FDF 0
0 0
      FDF ¯1
0.8414709848 ¯1.080604612
```

We need a dozen more one-line programs, similar to TIMES and SIN, to embody the vector operations corresponding to the typical derivative rules of elementary calculus. These programs will be left for the reader. Because of the strange symbols for the primitive functions in the appendix, Table 3, the corresponding vector operators may be given standard names such as COS and EXP without any confusion. Together, the fourteen one-line programs and the two-line program FDF are saved as a workspace, named AUTODIFF, in the APL environment. (In APL, we do not save and retrieve individual programs; we save and retrieve libraries of functions, called workspaces. Typically, a user will have several different workspaces, each designed for a particular type of work.) The AUTODIFF workspace enables the user to evaluate the derivative of any elementary function. The user simply edits line 2 of FDF, entering the desired formula (written in vector operators). Dyadic operators, TIMES and DIV, allow the expression to be written in typical "infix" form.

The AUTODIFF workspace generalizes, with very little change, to enable the user to evaluate the gradient of any elementary multivariate function. Although this development follows a different track from Section 2, the implementation of automatic differentiation for gradients is a significant and beautiful application of APL. Since the inception of automatic differentiation in [10], applications have been to multivariate analysis. (cf. [1]). Automatic differentiation for gradients uses a generalization of Table 1. At a point $\mathbf{p} = (a, b, c)$, associate each function $u(x, y, z)$ with the numerical vector $U = (u(\mathbf{p}), u_x(\mathbf{p}), u_y(\mathbf{p}), u_z(\mathbf{p}))$ or $(u(\mathbf{p}), \text{grad}\, u(\mathbf{p}))$. Then $X = (a, 1, 0, 0)$, $Y = (b, 0, 1, 0)$ and $Z = (c, 0, 0, 1)$. The automatic differentiation operators on these vectors are the same as above except that the first derivative is replaced by the gradient. In APL, this is accomplished by replacing $U[1]$ and $V[1]$ by $1 \downarrow U$ and $1 \downarrow V$ respectively. Moreover, this will work for any number of variables since $1 \downarrow U$ always returns the gradient vector. The revised workspace, consisting of fourteen one-line programs (no loops or conditionals) and a driver FGRADF, enables the evaluation of the gradient of any elementary function in any number of variables! An exposition of this is available from the author.

2. Calculating Higher-Order Derivatives

In order to evaluate derivatives up to order n at the point a, we generalize Table 1.

Table 2

function	vector at a
$h(x)$	$H = (h(a), h'(a), h''(a), \ldots, h^{(n)}(a))$
$u(x)$	$U = (u(a), u'(a), u''(a), \ldots, u^{(n)}(a))$
$v(x)$	$V = (v(a), v'(a), v''(a), \ldots, v^{(n)}(a))$
$\text{id}(x) = x$	$X = (a, 1, 0, 0, \ldots, 0)$
$c(x) = s$	$C_s = (s, 0, 0, 0, \ldots, 0)$

We will simply write $U = (u, u', u'', \ldots, u^{(n)})$ where the dependence on a is understood. Again, if $h(x)$ is a combination of $u(x)$ and $v(x)$ where U and V are known, then we will define a corresponding vector operation on U and V that yields H. All of the APL functions of Section 1 will require new, generalized versions (stored in a new workspace).

In a blank APL workspace, we define the global variable "order" to store the value of n. All programs will refer to order. The following function returns C_s. (See the replicate function in the appendix, Table 4.)

```
      order ← 4
      ∇ VECTOR ← C s
[1]   VECTOR ← s,order/0
[2]   ∇
      C 2
2 0 0 0 0
```

The space of vectors of the form $U = (u, u', u'', \ldots, u^{(n)})$ is linear in the sense that if $h(x) = s \cdot u(x) + t \cdot v(x)$ then $H = sU + tV$. Since X and C_s are known, linear functions are ready for automatic differentiation. Let's try $f(x) = 7x + 3$ and evaluate $F = (f(a), f'(a), \ldots, f^{(n)}(a))$ for different values of a and n.

```
      ∇ F ← FDF a
[1]   X ← a,1,(order-1)/0
[2]   F ← (7×X) + C 3
[3]   ∇
      FDF 2
17 7 0 0 0
      order ← 10
      FDF ¯4
¯25 7 0 0 0 0 0 0 0 0 0
```

For $h(x) = u(x)v(x)$, we need to define the vector operator TIMES such that $H = U$ TIMES V. We can easily calculate that $h = uv$, $h' = u'v + uv'$, $h'' = u''v + 2u'v' + uv''$, and $h^{(3)} = u^{(3)}v + 3u''v + 3u'v'' + uv^{(3)}$. Indeed, the higher derivatives are formed as in the binomial theorem, a fact that is known as Leibniz's rule:

$$h^{(k)} = (0!k)\,u^{(k)}v + (1!k)\,u^{(k-1)}v' + \cdots + (k!k)\,uv^{(k)},$$

where we have used the APL notation for the binomial coefficients (see appendix).

To implement Leibniz's rule, we shall define a vector operator BDOT (for "binomial dot product"). Specifically, if $P = (p_0, p_1, \ldots, p_k)$ and $Q = (q_0, q_1, \ldots, q_k)$ then we define

$$P \text{ BDOT } Q = (0!k)\,p_kq_0 + (1!k)\,p_{k-1}q_1 + \cdots + (k!k)\,p_0q_k.$$

(The program defining BDOT will be given presently.) For the remainder of this paper, the importance of Leibniz's rule, reformulated as follows, cannot be overemphasized:

$$h(x) = u(x)v(x) \Rightarrow h^{(k)} = (u, u', \ldots, u^{(k)}) \text{ BDOT } (v, v', \ldots, v^{(k)}).$$

The vector $H = (h, h', \ldots, h^{(n)})$ will be built by catenating one entry at a time, using BDOT to perform Leibniz's rule. Thus the operator TIMES is described by the following pseudo-APL program. (Observe that $((k+1)\uparrow U)$ returns $(u, u', \ldots, u^{(k)})$, the $(k+1)$-length initial segment of U.)

```
      ∇ H ← U TIMES V
[1]   H ← U[0]×V[0]
[2]   FOR k ← 1 TO order
[3]      H ← H, ((k+1)↑U) BDOT ((k+1)↑V)
[4]   NEXT k
[5]   ∇
```

Unfortunately, such a simple looping structure does not exist in APL. The actual program shows the best (line 5) and the worst of APL. The new symbols are explained immediately following the program.

```
      ∇ H ← U TIMES V ;k
[1]   H ← U[0]×V[0]
[2]   →(order=0)/0  ⍝Stop if order=0
[3]   k←0
[4] Loop: k←k+1
[5]   H ← H, ((k+1)↑U) BDOT ((k+1)↑V)
[6]   →(k<order)/Loop
[7]   ∇
```

The variable k is declared to be local by listing it preceded by a semicolon in the function header. The APL symbol ⍝ indicates that the remainder of the line is a comment. The line $\rightarrow(k < \text{order})/\text{Loop}$ means if $k <$ order then branch to the line labeled Loop. (Specifically, $k <$ order evaluates as 1 for true or 0 for false; then the replicate function creates 1 or 0 copies of the word Loop. The result is $\rightarrow$ Loop which means goto Loop, or $\rightarrow$ "empty vector" which means goto the next line. Also, $\rightarrow 0$ means exit the program. Other APL programmers may implement this loop with different branches written with different symbols, though we've covered all the reader will need to know.)

We still have to define the operator BDOT. Let P and Q be any vectors of length $k+1$, for any $k \geq 0$. Let BINOMCOEFS be the vector $(0!k),(1!k),\ldots,(k!k)$. Now P BDOT Q is given by multiplying corresponding elements of BINOMCOEFS, Q, and the reverse of P, followed by summing across the resulting vector. In APL notation: $+/$ BINOMCOEFS $\times (\phi P) \times Q$.

There are several ways to generate the vector BINOMCOEFS. To use $i!k$ directly on each call to BDOT is very inefficient. These coefficients will be stored in a global variable that is essentially Pascal's triangle. In fact, the familiar algorithm for constructing Pascal's triangle is imitated using vector addition in the program PASCAL.

```
      ∇ PASCAL n ;ROW
[1]   ROW ← 1
[2]   ROW  ⍝This line prints the variable ROW
[3]   TRIANGLE ← 1
[4] Loop: ROW ← (ROW,0) + (0,ROW)
[5]   ROW
[6]   TRIANGLE ← TRIANGLE,ROW
[7]   →((⍴ROW)≤n)/Loop
[8]   ∇
```

```
      PASCAL 10
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

As long as order ≤ 10, we will not need to run PASCAL again. For simplicity and efficiency, the global variable TRIANGLE is simply a long vector. Thus BDOT must extract the "row" of length $m = k + 1$. Finally, we have the actual BDOT program.

```
      ∇ scalar ← P BDOT Q ;m;BINOMCOEFS
[1]   m ← ρ,P
[2]   BINOMCOEFS ← m↑ ((m-1)×m÷2)↓ TRIANGLE
[3]   scalar ← +/ BINOMCOEFS × (ΦP) × Q
[4]   ∇
```

(One technical footnote: the comma in line 1 of BDOT converts the argument into a vector. It is only necessary in case P is a scalar as opposed to a vector of length one.)

Let us test the TIMES operator by calculating the first four derivatives of $f(x) = x^3$ at -2.

```
      ∇ FDF[2]
[2]   F ← X TIMES X TIMES X
[3]   ∇
      order ← 4
      FDF ¯2
¯8 12 ¯12 6 0
```

Now we can concentrate on finding recursive mathematical formulas for other kth-order differentiation rules. The function BDOT will prove to be extremely useful. Consider first the quotient rule.

Suppose $h(x) = u(x) \div v(x)$. We wish to find $h^{(k)}$ in terms of the "known" vectors U and V. Straightforward iteration of the quotient rule gets very messy and appears difficult to reproduce without using symbolic (literal string) manipulation. Instead, observe that $u(x) = h(x)v(x)$ and use Leibniz's rule. The kth derivative is

$$\begin{aligned} u^{(k)} &= (h, h', h'', \ldots, h^{(k)})\,\mathrm{BDOT}\,(v, v', v'', \ldots, v^{(k)}) \\ &= h^{(k)}v + (h, h', h'', \ldots, h^{(k-1)}, 0)\,\mathrm{BDOT}\,(v, v', v'', \ldots, v^{(k)}). \end{aligned}$$

This equation can be solved for $h^{(k)}$:

$$h^{(k)} = \left[u^{(k)} - (h, h', h'', \ldots, h^{(k-1)}, 0)\,\mathrm{BDOT}\,(v, v', v'', \ldots, v^{(k)})\right] \div v.$$

This is a relatively simple recursive formula. If H is built one entry at a time as in TIMES, H will be $(h, h', h'', \ldots, h^{(k-1)})$ when it is time to evaluate $h^{(k)}$. Thus DIV is defined as follows.

```
      ∇ H ← U DIV V ;k
[1]   H ← U[0]÷V[0]
[2]   →(order=0)/0
[3]   k←0
[4] Loop: k←k+1
[5]   H ← H, ( U[k] - ((H,0) BDOT (k+1)↑V) ) ÷ V[0]
[6]   →(k<order)/Loop
[7]   ∇
```

We test DIV on $f(x) = 1 \div x$.

```
      ∇ FDF[2]
[2]   F ← (C 1) DIV X
[3]   ∇
      FDF 10
0.1 ¯0.01 2E¯3 ¯6E¯4 2.4E¯4
```

Next, suppose that $h(x) = e^{u(x)}$. We seek to define the EXP operator so that $H = \text{EXP}\, U$, or $(h, h', h'', \ldots, h^{(n)}) = \text{EXP}(u, u', u'', \ldots, u^{(n)})$. Thinking recursively, we observe that $h'(x) = h(x)u'(x)$. Then Leibniz's rule yields $h'' = (h, h')\,\text{BDOT}\,(u', u'')$ and, in general,

$$h^{(k)} = (h, h', \ldots, h^{(k-1)})\,\text{BDOT}\,(u', u'', \ldots, u^{(k)}).$$

Thus we arrive at the following definition of EXP. (Observe that $1 \downarrow U$ returns the vector $(u', u'', \ldots, u^{(n)})$.)

```
      ∇ H ← EXP U ;k;UPRIME
[1]   H ← *U[0]
[2]   →(order=0)/0
[3]   UPRIME ← 1↓U
[4]   k←0
[5] Loop: k←k+1
[6]   H ← H, H BDOT k↑UPRIME
[7]   →(k<order)/Loop
[8]   ∇
```

Whereas the definition of EXP used BDOT (and iteration) directly, the LN operator will use DIV. Suppose that $h(x) = \ln(u(x))$. Then $h'(x) = u'(x) \div u(x)$. Assuming that U is known, the vector UPRIME ($1 \downarrow U$, as in the EXP program) corresponding to $u'(x)$ is also known (up to order -1 derivatives of u'). Therefore, we may evaluate the vector corresponding to $h'(x)$ by calling the DIV operator, as is done in the next program. (The APL function $^{-}k \downarrow U$ drops the last k elements of U; thus $(^{-}1) \downarrow U$ returns $U = (u, u', u'', \ldots, u^{(n-1)})$.)

```
      ∇ H ← LN U
[1]   H ← ⊛U[0]
[2]   →(order=0)/0
[3]   order ← order-1
[4]   H ← H, (1↓U) DIV (¯1↓U)
[5]   order ← order+1
[6]   ∇
```

As we shall see, most other functions or their derivatives (as in LN) can be given in terms of previously defined operations. The last real challenge of typical calculus functions is SIN and COS.

Suppose that $h(x) = \sin(u(x))$. Let $g(x) = \cos(u(x))$. Then $h'(x) = g(x)u'(x)$ and $g'(x) = -h(x)u'(x)$. Using Leibniz's rule, $h'' = (g, g')\,\text{BDOT}\,(u', u'')$ and $g'' = -(h, h')$ BDOT (u', u''). In general,

$$h^{(k)} = \left(g, g', \ldots, g^{(k-1)}\right)\text{BDOT}\left(u', u'', \ldots, u^{(k)}\right) \quad \text{and}$$

$$g^{(k)} = -\left(h, h', \ldots, h^{(k-1)}\right)\text{BDOT}\left(u', u'', \ldots, u^{(k)}\right).$$

This jointly recursive definition can be used to generate the vectors H and G simultaneously. The vectors H and G will be respectively named SINU and COSU in the following APL program. Rather than include this entire program in two functions SIN and COS, we define one function SINCOS which adds (or replaces) the global variables SINU and COSU to the workspace library of variables. (There is no provision in APL for output *parameters* except for one function value.) The functions SIN, COS, TAN, and COT may all call SINCOS and return the appropriate value.

```
      ∇ SINCOS U ;k;NEWSIN;UPRIME
[1]   ⍝ The results are returned in the global
      variables SINU and COSU
[2]   SINU ← 1○U[0]
[3]   COSU ← 2○U[0]
[4]   →(order=0)/0
[5]   UPRIME ← 1↓U
[6]   k←0
[7]  Loop: k←k+1
[8]   NEWSIN ← SINU, COSU BDOT k↑UPRIME
[9]   COSU ← COSU, -(SINU BDOT k↑UPRIME)
[10]  SINU ← NEWSIN
[11]  →(k<order)/Loop
[12]  ∇
      ∇ H ← SIN U
[1]   SINCOS U
[2]   H ← SINU
[3]   ∇
      ∇ H ← COS U
[1]   SINCOS U
[2]   H ← COSU
[3]   ∇
      ∇ H ← TAN U
[1]   SINCOS U
[2]   H ← SINU DIV COSU
[3]   ∇
      ∇ H ← COT U
[1]   SINCOS U
[2]   H ← COSU DIV SINU
[3]   ∇
```

For convenience, we add RECIP, SEC, and CSC.

```
      ∇ H ← RECIP U
[1]   H ← (C 1) DIV U
[2]   ∇
      ∇ H ← SEC U
[1]   H ← RECIP COS U
[2]   ∇
      ∇ H ← CSC U
[1]   H ← RECIP SIN U
[2]   ∇
```

ARCTAN and ARCSIN are like LN in that their derivatives are given in terms of previously defined operations. (The variable U in the programs below is a value parameter; that is, changes to U within the program do not affect variables (used as actual arguments) outside the program.)

```
      ∇ H ← ARCTAN U ;UPRIME
[1]   H ← ¯3○U[0]
[2]   →(order=0)/0
[3]   UPRIME ← 1↓U
[4]   U ← ¯1↓U
[5]   order ← order-1
[6]   H ← H, UPRIME DIV ((C 1) + U TIMES U)
[7]   order ← order+1
[8]   ∇
```

```
      ∇ H ← ARCSIN U ;UPRIME
[1]   H ← ¯1○U[0]
[2]   →(order=0)/0
[3]   UPRIME ← 1↓U
[4]   U ← ¯1↓U
[5]   order ← order-1
[6]   H ← H, UPRIME DIV SQRT ((C 1) - U TIMES U)
[7]   order ← order+1
[8]   ∇
```

The vector operator SQRT has not yet been defined, and it is a fitting finale. Suppose that $h(x) = \sqrt{u(x)}$. Observe that $h'(x) = u'(x) \div (2h(x))$. Thus $2h(x)h'(x) = u'(x)$. By Leibniz's rule,

$$u^{(k)} = 2 \times (h, h', \ldots, h^{(k-1)}) \,\mathrm{BDOT}\, (h', h'', \ldots, h^{(k)}).$$

We may find a recursive formula giving $h^{(k)}$ in terms of one BDOT operation. The details are left to the reader.

Almost all functions commonly used, such as hyperbolic functions, could be defined in terms of the preceding operations, as were RECIP, SEC, and CSC. One could define U POWER r to be EXP($r \times$ LN(U)). However, in many special cases it is more efficient, less restrictive, and more accurate to use repeated TIMES, the RECIP function, or SQRT. (It is difficult to directly implement a general real power function.) POWER is left undefined, as in the language Pascal, so that the user must choose the expression appropriate to the application.

We may now enter any elementary function $f(x)$ in FDF, any a, and any order and then evaluate $f(a), f'(a), \ldots, f^{(\text{order})}(a)$. We are limited only by computer representation (roundoff) error and computation time. (The limitation of TRIANGLE can be raised.) Let's try the example from the introduction.

```
      ∇ FDF[2]
[2]   F ← (X TIMES CSC X) DIV LN(ARCTAN(EXP X))
[3]   ∇
      order ← 10
      FDF 1
6.018945428 ¯5.953764719 27.62437643 ¯139.6701361
1021.683358 ¯9127.489017 98448.35779 ¯1236873.597
17767053.52 ¯287085222.7 5154373690
```

The above calculation took less than 4 seconds on an IBM PS/2 Model 50 without a math coprocessor. Finding some other method to check the above answers is a real challenge. Symbolic differentiation using the software "CALCULUS" [2] verified $f(1)$ and the first three derivatives. However, the third derivative was 90 lines long, with 36 characters (no spaces) in each line. Producing the third derivative from the second took over 15 seconds. There was not enough room for the fourth derivative. Numerical approximations of the higher derivatives verified the first few digits of each entry but that was all the accuracy obtained by approximation. Automatic differentiation appears more reliable than any other available method.

Our implementation is very accurate through about 20 derivatives, but there are limitations. Even though our method is theoretically exact, we are dealing with the finite number system of a computer. For high values of order, the massive number of calculations (using the huge numbers in TRIANGLE) may result in significant roundoff error. An abrupt loss of significance has been observed in some examples around the 25th derivative.

One application of FDF is to the calculation of Taylor polynomial coefficients for $f(x)$. Indeed, this problem is essentially equivalent to calculating derivatives. In [6] these Taylor coefficients are recursively generated and derivatives are found from the coefficients (cf. [4]).

3. Conclusion

Automatic differentiation is interesting mathematics and, when implemented in APL, it is a powerful, accurate, clear, and practical tool for finding the numerical value of derivatives at a point. For first derivatives, automatic differentiation is simply an implementation of the set of differentiation rules at a point as presented in most calculus books. The idea of calculating in a system of vectors (originating implicitly in [10]) is the unusual twist that distinguishes the present method. Cumbersome computer code in BASIC and FORTRAN may have been a deterrent to the awareness of automatic differentiation as an alternative to the old established methods. In contrast, the APL code of Section 1 is very simple. Mathematical symbolism is imitated in APL by the large and expandable library of operators and functions and the vector orientation of the language. These features fit automatic differentiation very well. Indeed, for higher-order derivatives, the symbolism of APL aids in the mathematical derivations. Specifically, the encapsulation of Leibniz's rule as the dyadic operator BDOT leads to recursive formulas for the kth derivatives of other mathematical operations and functions. These recursive formulas are interesting from a purely mathematical viewpoint. From a practical viewpoint, the method

is unmatched in its ability to calculate the value of high-order derivatives at a point. Were it not for the evident lack of clear programming structures in APL (and thus its unpopularity), our implementation would be ideal.

Concerning the language APL, we observe that APL is heavily dependent on the use of an interpreter as opposed to a compiler. We've seen the usefulness of the immediate execution mode in a workspace of user-defined functions and global variables. Even the lack of clear programming structures encourages good use of the interpreter. Loops are very inefficient in interpreted languages since each line must be retranslated into machine code on each pass through the loop. In APL, a *line of code* is powerful enough to perform tasks usually relegated to a loop (or even nested loops) in other languages. The program BDOT is a typical example of what can be accomplished without any loops or conditional branches. Another characteristic of an interpreter is that it requires less advance notice of data types, array dimensions, and function calls. This ability is thoroughly exploited in APL, and simplifies this application where we constantly deal with vectors of different lengths.

There is further work to be done in automatic differentiation. Given the ease with which the methods for differentiating a function of one variable generalize to multivariate functions, the next obvious step is to design an APL workspace that calculates all partial derivatives up to any given order for any elementary multivariate function. For a fixed order, differentiation rules could easily be directly programmed. However, arbitrary order would require recursion formulas as in Section 2. Considering the ability of APL to handle arrays of arbitrary dimensions as easily as vectors, this task is within reach, and the author is preparing such a sequel. Another area that needs to be pursued is the theoretical and empirical analysis of the numerical limitations of this method as indicated at the end of Section 2.

Appendix: Selected APL Symbols

Table 3. Primitive scalar functions.

`X×Y`	multiplication	the circle functions	
`X÷Y`	division	`1○Y`	$\sin(Y)$
`X*Y`	X to the power of Y	`2○Y`	$\cos(Y)$
`*Y`	e to the power of Y	`3○Y`	$\tan(Y)$
`⍟Y`	natural logarithm of Y	`¯1○Y`	$\arcsin(Y)$
`X!Y`	binomial coefficient, Y choose X	`¯3○Y`	$\arctan(Y)$

A scalar function operates on each element of a vector argument, as in scalar multiplication. If a dyadic scalar function has two vector arguments of equal length, the operation is applied to corresponding elements, for example 5 3 2 × 4 9 1 = 20 27 2.

Table 4. Vector manipulation functions.

`⍴Y`	length of Y, returns the number of elements in vector Y
`+/Y`	sum across Y, returns the sum of all elements in vector Y
`X,Y`	catenate vectors X and Y
`k↑Y`	take the first k elements of Y, returns this k element vector
`k↓Y`	drop the first k elements of Y, returns the remaining vector
`⌽Y`	reverse Y, returns the elements of Y in reverse order (a vector)
`k/x`	replicate x, returns a vector consisting of k copies of x

REFERENCES

1. H. Kagiwada, R. Kalaba, N. Rasakhoo, and K. Spingarn, *Numerical Derivatives and Nonlinear Analysis*, Plenum, New York-London, 1986.
2. J. G. Kemeny, *CALCULUS for the IBM PC*, True BASIC Inc., Hanover, NH, 1985.
3. E. J. LeCuyer Jr., Teaching mathematics using APL, *College Mathematics Journal* 17 (1986) 344–358.
4. R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM Studies in Applied Mathematics, 2, SIAM, Philadelphia, 1979, 24–31.
5. L. B. Rall, The arithmetic of differentiation, *Mathematics Magazine* 59 (1986) 275–282.
6. _____, Automatic differentiation: Techniques and applications, *Lecture Notes in Computer Science 120*, Springer-Verlag, Berlin-Heidelberg-New York, 1981.
7. _____, Differentiation in Pascal-SC: Type GRADIENT, *ACM Trans. Math. Software* 10 (1984) 161–184.
8. M. Rubinstein and S. D. Lewis, APL: A language for modern times, *PC: The Independent Guide to IBM Personal Computers* (*PC Magazine*) 3 (April 3, 1984).
9. J. R. Turner, *Pocket APL Reference Guide*, STSC Inc., Rockville, Maryland, 1985.
10. R. E. Wengert, A simple automatic derivative evaluation program, *Communications ACM* 7 (1964) 463–464.
11. A. S. Wexler, Automatic evaluation of derivatives, *Applied Mathematics & Computation* 24 (1987) 19–46.

Classroom Computer Capsules

Authors are invited to submit articles for a new section, Classroom Computer Capsules, which will feature illuminating examples of using the computer to enhance teaching. These short articles will demonstrate the use of readily available computing resources to present or elucidate familiar topics in ways that can have an immediate and beneficial effect in the classroom.