ALGORITHMIC DIFFERENTIATION AND DIFFERENCING

Values of derivatives and Taylor coefficients of functions are required in various computational applications of mathematics to engineering and science. The traditional method for evaluation of derivatives is to use *symbolic differentiation,* in which the rules of differentiation are applied to transform formulas for functions into formulas for their derivatives. Then derivative values are calculated by evaluating these formulas.

Algorithmic differentiation (AD) is an alternative method for evaluation of derivatives. AD is based on the sequence of basic operations, that is, the algorithm used to evaluate the function to be differentiated. Each step in such a sequence consists of an arithmetic operation or the evaluation of some intrinsic function such as the sine or square root. The rules of differentiation are then applied to transform this sequence into a sequence of operations for evaluation of the desired derivative. Thus, AD transforms the *algorithm* for evaluation of a function into an algorithm for evaluation of its derivatives. Since evaluation of functions on digital computers is carried out by means of algorithms in the form of subroutines or programs, AD is particularly suitable in this case. Hence, the historical designation "automatic differentiation" as the process was intended for use on computers. These terms for AD are equivalent.

The processes of algorithmic and symbolic differentiation are based on the same definitions and theorems of differential calculus. They differ in their goals. The purpose of symbolic differentiation being production of formulas for derivatives, while the purpose of AD is computation of values of derivatives. Hence, AD is also referred to as "computational differentiation." Their starting points also differ. Symbolic differentiation begins with formulas, and AD begins with algorithms. If the function to be differentiated is expressed as a formula, then an equivalent algorithm for its evaluation must be derived before AD can be applied. Automatic methods for conversion of formulas to algorithms are well known (1) and are used to produce internal algorithms by calculators and computer programs which accept formulas as input. On the other hand, AD is applicable to functions which are *only* defined algorithmically, as by computer subroutines or programs. For many computational purposes, such as the solution of linear systems of equations, efficient algorithms are preferred to formulas. AD generally requires less computational effort than symbolic differentiation followed by formula evaluation even for functions defined by formulas. The algorithmic approach to derivatives also applies to accurate evaluation of divided differences as described in the final section of this article.

In this article the basic idea of AD is first illustrated by a simple example. This is followed by sections on automatic generation of Taylor series and its application to the computational solution of initial value problems for ordinary differential equations. Subsequent sections deal with evaluating partial derivatives, including gradients, Jacobians, and Hessians, along with various applications, including estimation of sensitivities, solution of nonlinear systems of equations, and optimization. See Ref. 2 for an introduction to AD and its applications.

AN EXAMPLE

To begin on familiar ground, first consider the application of AD to a function defined by a formula. Suppose a circuit, the details of which are unimportant, produces the amplitude-modulated current $I(t)$ given by

$$I(t) = A\sin\Omega t\,\sin\omega t \qquad (1)$$

as a function of time t, where the amplitude A and the frequencies Ω, ω are known constants pertaining to the circuit. If this current flows through a device with inductance L, then the corresponding voltage drop is given by

$$E(t) = L\dot{I}(t) = L\frac{d}{dt}(A\sin\Omega t\,\sin\omega t) \qquad (2)$$

Suppose we want to construct a graph of $I(t)$ and $E(t)$ by evaluating $I(t)$, $E(t) = L\overset{2}{I}(t)$ for a number of values of t and connecting the resulting points to obtain smooth curves. First, consider the evaluation of $I(t)$ itself. Although formula (1) defines $I(t)$, it does not give an explicit step-by-step procedure to compute its value for given t. A straightforward method is to compute the quantities $s_1, \ldots, s_7$ given by

$$\begin{aligned} s_1 &= t \\ s_2 &= \Omega s_1 \\ s_3 &= \sin s_2 \\ s_4 &= As_3 \\ s_5 &= \omega s_1 \\ s_6 &= \sin s_5 \\ s_7 &= s_4 s_6 \end{aligned} \qquad (3)$$

For a given value of t, it is evident that $s_7 = I(t)$. It follows that Eq. (1) and Eq. (3) are equivalent but different representations of the same function. In fact, given Eq. (3), Eq. (1) is obtained by literal substitution for the values of s_1, ..., s_7, starting with $s_1 = t$. The algorithmic representation in Eq. (3) of the function is called a *code list* (3), because early computers and programmable calculators required this kind of explicit list of operations to evaluate a function. Computers and calculators that accept formulas as input convert them internally to a sequence similar to Eqs. (3) to carry out the evaluation process.

Now the value of the derivative $\overset{2}{I}(t)$ is computed by applying the rules of differentiation to the code list (3) rather than to Eq. (1). This is implemented in several ways. The earliest is *interpretation* of the code list, introduced by Moore (4) and later by Wengert (5). In this method, the code list is used to construct a corresponding sequence of calls to subroutines that compute the appropriate derivative values. For example, if $s = uv$ appears as an entry in the code list, then the values of $u, v, \overset{2}{u}, \overset{2}{v}$ are sent to a subroutine that returns the value $\overset{2}{s} = \overset{2}{u}v + u\overset{2}{v}$. In terms of the usual differentiation formulas, the result of this process is the sequence $\overset{2}{s}_1, \ldots, \overset{2}{s}_7$ given by

$$\begin{aligned}
\dot{s}_1 &= 1 \\
\dot{s}_2 &= \Omega\dot{s}_1 \\
\dot{s}_3 &= \dot{s}_2 \cos s_2 \\
\dot{s}_4 &= A\dot{s}_3 \\
\dot{s}_5 &= \omega\dot{s}_1 \\
\dot{s}_6 &= \dot{s}_5 \cos s_5 \\
\dot{s}_7 &= s_4\dot{s}_6 + \dot{s}_4 s_6
\end{aligned} \tag{4}$$

As in the case of Eq. (3) for evaluating $I(t)$, it is evident that the result of Eq. (4) is $\overset{2}{s}_7 = \overset{2}{I}(t)$. Thus, it is possible to compute the value of the derivative of a function directly from a code list for evaluating the function. Furthermore, literally evaluating the sequence in Eq. (4) gives the formula

$$\dot{I}(t) = A(\omega \sin \Omega t \cos \omega t + \Omega \cos \Omega t \sin \omega t)$$

So in this sense, automatic and symbolic differentiation of a function are equivalent. However, it is important to note that AD is used to compute *numerical* values of $s_1, \ldots, s_7$ and $\overset{2}{s}_1, \ldots, \overset{2}{s}_7$ rather than literal values. Certain values from the code list for evaluating the function are required for evaluating its derivative, in this case s_2, s_4, s_5, s_6.

Another method for automatic differentiation uses the fact that the formulas for derivatives as used in Eq. (4) can themselves be represented by code lists. For example, the derivative of $s = uv$ would be computed in the three steps $d_1 = \overset{2}{u}v$, $d_2 = u\overset{2}{v}$, $d_3 = d_1 + d_2$ from the previously obtained values of $u, v, \overset{2}{u}, \overset{2}{v}$. Then these sublists are inserted at the appropriate place to obtain a code list for the derivative of the original function. Application of this method to Eq. (3) gives

$$\begin{aligned}
d_1 &= 1 && (= \dot{s}_1) \\
d_2 &= \Omega d_1 && (= \dot{s}_2) \\
d_3 &= \cos s_2 && \\
d_4 &= d_2 d_3 && (= \dot{s}_3) \\
d_5 &= A d_4 && (= \dot{s}_4) \\
d_6 &= \omega d_1 && (= \dot{s}_5) \\
d_7 &= \cos s_5 && \\
d_8 &= d_6 d_7 && (= \dot{s}_6) \\
d_9 &= s_4 d_8 && \\
d_{10} &= d_5 s_6 && \\
d_{11} &= d_9 + d_{10} && (= \dot{s}_7)
\end{aligned} \tag{5}$$

This process is called *code transformation* because it transforms a code list for the function into a code list for its derivative. The same method is used to transform Eq. (6) into a code list for the second derivative $\overset{2}{I}(t)$, if desired, and so on. An early example of code transformation appears in Ref. 3; see also Ref. 6. A third way to implement automatic differentiation, *operator overloading,* is described later.

However implemented, it follows from the chain rule for derivatives that AD succeeds if and only if the function represented by the code list is differentiable at the given value of t. Nondifferentiability causes the step-by-step evaluation of the derivative to break down at some stage, for example, because of attempted division by zero or evaluating a standard function for an invalid argument.

Before leaving this simple example, it is also important to note that AD can be used to compute values of *differentials* as well as derivatives (6). If the value of d_1 in the code list in Eq. (6) is taken to be $d_1 = \tau$ instead of $d_1 = 1$, then the result is $d_{11} = \overset{2}{I}(t)\tau$. By definition, this is the value of the differential $dI = \overset{2}{I}(t)dt$ for dt equal to the given value τ. In some applications, dI is used as an approximation to the increment $\Delta I = I(t + \tau) - I(t)$ for $\tau = dt$ small. If the values of $I(t_0)$ and $\overset{2}{I}(t_0)\tau$ are computed with $\tau = t - t_0$, then the results are also the values of the first two terms of the Taylor series expansion of $I(t)$ at $t = t_0$,

$$I(t) = I(t_0) + \dot{I}(t_0)(t - t_0) + \cdots$$

Next, we show how AD is used to obtain values of as many subsequent terms of the Taylor series as desired for a sufficiently differentiable function, in particular, for series solution of initial-value problems for ordinary differential equations.

ALGORITHMIC GENERATION OF TAYLOR COEFFICIENTS

Suppose that the function $x(t)$ has a convergent Taylor series expansion at $t = t_0$, at least for $|t - t_0|$ sufficiently small. This expansion is written

$$x(t) = a_0 + a_1 + \cdots + a_n + \cdots = \sum_{n=0}^{\infty} a_n \tag{6}$$

where

$$a_n = a_n(t) = \frac{1}{n!} x^{(n)}(t_0)(t - t_0)^n, \quad n = 0, 1, 2, \ldots \tag{7}$$

The numbers $a_0, a_1, \ldots$ are called the *normalized* Taylor coefficients of $x(t)$ expanded at $t = t_0$ with increment $\tau = t - t_0$. For computational purposes, it is convenient to identify the function value $x(t)$ with the vector of its normalized Taylor coefficients, $x(t) \leftrightarrow (a_0, a_1, \ldots, a_n, \ldots)$. If C is a constant, then $C \leftrightarrow (C, 0, 0, \ldots)$, and for the independent variable t, $t \leftrightarrow (t_0, t - t_0, 0, \ldots)$.

Normalized Taylor coefficients can be evaluated by AD for functions defined algorithmically. This method is also called *recursive generation* of normalized Taylor coefficients, and a few special applications prior to the computer era are known (6). By 1959, this technique was incorporated in computer programs created by R. E. Moore and his coworkers at Lockheed Aviation (7).

For simplicity, assume that the function to be expanded is defined by a code list. This reduces the problem to calculating normalized Taylor coefficients of the results of arithmetic operations and standard functions, given the coefficients of their arguments. Typical formulas for these are given later, and more details are in Refs. 6 and 7. The computations involved are *numerical* and are carried out very rapidly on a digital computer. This permits carrying out the Taylor expansion to a much higher degree than ordinarily possible by symbolic methods, which are of course limited to functions defined by formulas.

Suppose that the normalized Taylor coefficients for the functions $x(t)$ and $y(t)$ expanded at $t = t_0$ with increment $\tau = t - t_0$ are given. The coefficients of the result $z(t) = x(t) \circ y(t)$, where $\circ$ denotes an arithmetic operation, are obtained directly by power series arithmetic. Let $a_0, a_1, \ldots, a_n, \ldots$ and $b_0, b_1, \ldots, b_n, \ldots$ be the coefficients of the series for the operands $x(t)$, $y(t)$, respectively. The coefficients $c_0, c_1, \ldots, c_n, \ldots$ of the result $z(t) = x(t) \pm y(t)$ are given by

$$c_n = a_n \pm b_n, \quad n = 0, 1, \ldots \tag{8}$$

for addition and subtraction, respectively, and by the convolutional formula

$$c_n = \sum_{k=0}^{n} a_k b_{n-k} = \sum_{k=0}^{n} a_{n-k} b_k \tag{9}$$

for multiplication, $z(t) = x(t)y(t)$. The formula for division, $z(t) = x(t)/y(t)$, is obtained by using Eq. (11) for the product $y(t)z(t) = x(t)$, which gives a system of linear equations to be solved for $c_0, c_1, \ldots$. These equations are $b_0c_0 = a_0$, $b_0c_1 + b_1c_0 = a_1$, and so on. If $b_0 \neq 0$, then they are solved in turn for the general coefficient of the result,

$$c_n = \left(a_n - \sum_{k=0}^{n-1} b_{n-k} c_k\right) \frac{1}{b_0}, \quad n = 0, 1, \ldots \tag{10}$$

This formula is *recursive* because the previously computed values of $c_0, \ldots, c_{n-1}$ are needed for calculating c_n, whereas the formulas for addition, subtraction, and multiplication depend only on the coefficients of their arguments.

Generating normalized Taylor coefficients is also required for standard functions, for example, for $z(t) = \sin x(t)$ given the coefficients of $x(t)$. In principle, this is done by substituting of the power series for $x(t)$ in the power series for the sine function and collecting coefficients of like powers of $\tau = t - t_0$, but the algebra is extremely cumbersome. An efficient method formulated by Moore (4) (see Refs. 6 and 7), uses differential equations satisfied by the standard functions. This technique is described in general in the next section. Here, the basic idea is illustrated by the exponential function

$$y(t) = \exp[x(t)] = e^{x(t)}$$

which satisfies the first-order linear differential equation

$$\dot{y}(t) = e^{x(t)}\dot{x}(t) = y(t)\dot{x}(t) \tag{11}$$

Given the normalized Taylor coefficients $a_0, a_1, \ldots$ of $x(t)$ expanded at $t = t_0$ with increment $\tau = t - t_0$, the corresponding coefficients $b_0, b_1, \ldots$ of $y(t)$ are found as follows: First, note that at $t = t_0$, the initial condition $y(t_0) = \exp[x(t_0)]$, that is, $b_0 = \exp(a_0)$. Next, formal term-by-term differentiation of the series for $x(t)$ gives $\overset{2}{x}(t) \leftrightarrow [a_1/\tau, 2a_2/\tau, \ldots, (n+1)a_{n+1}/\tau, \ldots]$ and a similar vector of coefficients for $\overset{2}{y}(t)$. It follows from the differential equation in Eq. (14) that the coefficient $(n+1)b_{n+1}/\tau$ of $\overset{2}{y}(t)$ is equal to the corresponding coefficient in the series for the product $y(t)\overset{2}{x}(t)$ given by Eq. (11).

After multiplying by τ and dividing by $(n+1)$, the result is

$$b_{n+1} = \frac{1}{n+1}\sum_{k=0}^{n}(n-k+1)a_{n-k+1}b_k, \quad n = 0, 1, 2, \ldots \tag{12}$$

As with division, this formula gives b_{n+1} in terms of $b_0, \ldots, b_n$ and the known coefficients of $x(t)$ and hence is recursive. Starting with the initial value $b_0 = \exp(a_0)$, Eq. (15) gives $b_1 = a_1b_0$, and so on. Note that the exponential function is evaluated only *once* to obtain b_0. Calculating subsequent coefficients is purely arithmetical.

Automatically generating Taylor coefficients is easily implemented by using the algorithmic representation of the function (such as by a code list) to construct calls to subroutines for arithmetic operations and standard functions. Another method is operator overloading, which is described later.

Because computation is inherently finite, the results actually obtained are the coefficients $a_0, \ldots, a_d$ of the Taylor polynomial

$$T_d x(t) = \sum_{n=0}^{d} a_n = \sum_{n=0}^{d} \frac{1}{n!} x^{(n)}(t_0)(t - t_0)^n \tag{13}$$

of degree d rather than the complete Taylor series for a typical function $x(t)$. As before, it is convenient to use the correspondence $T_d x(t) \leftrightarrow (a_0, a_1, \ldots, a_d)$ between a Taylor polynomial and the $(d+1)$-dimensional vector of its coefficients. For given values of t_0, t, AD is used to generate Taylor polynomials of high degree d with a reasonable amount of effort, compared with symbolic differentiation when the latter is applicable. For example, consider calculating $I_{100}(t)$ by AD from the code list in Eq. (3) compared with applying symbolic differentiation 100 times to Eq. (1).

From calculus, the goodness of the approximation of $x(t)$ by $T_d x(t)$ is given by the *remainder term,*

$$\begin{aligned} R_d(t_0)x(t) = x(t) - T_d x(t) &= \frac{1}{d!}\int_{t_0}^{t} x^{(d+1)}(\theta)\left(\frac{t-\theta}{t-t_0}\right)^d d\theta \\ &= x^{(d+1)}(\vartheta)\frac{(t-t_0)^{d+1}}{(d+1)!}, \quad t_0 < \vartheta < t \end{aligned} \tag{14}$$

Moore has shown that automatically generating the Taylor coefficient by using interval arithmetic is a computational procedure that yields guaranteed bounds for the remainder term. For details, see Ref. 7.

SOLUTION OF INITIAL-VALUE PROBLEMS

The principal application of automatically generating Taylor coefficients is not to known functions but rather to unknown functions defined by initial-value problems for ordinary differential equations. The simplest example is for a single, first-order equation

$$\dot{x}(t) = f[t, x(t)], \quad x(t_0) = a_0 \tag{15}$$

where the known function $f(t, x)$ is defined by an algorithm, such as a code list, and the initial value a_0 is given. The method works as follows: The Taylor coefficients $(t_0, t - t_0, 0, \ldots, 0)$ of t are known, and suppose that the coefficients $(a_0, a_1, \ldots, a_d)$ of $T_d x(t)$ have been computed. Then, AD is used to obtain the coefficients $(b_0, b_1, \ldots, b_d)$ of the Taylor polynomial $T_d f[t, T_d x(t)]$. From the Taylor series for $\overset{2}{x}(t)$ and the differential, Eq. (18) it follows that

$$a_{d+1} = \frac{1}{d+1} b_d(t - t_0), \quad d = 0, 1, 2, \ldots \tag{16}$$

so the series for $x(t)$ is extended as long as the coefficients b_d can be calculated. This process starts with the initial value a_0. Then because $b_0 = f(t_0, a_0)$, $a_1 = b_0(t - t_0)$, and so on. Generally speaking, the value of $t - t_0$ is small, so multiplication by it and division by $(d+1)$ reduce the effect of any error in calculating of b_d on the value of the subsequent Taylor coefficient a_{d+1} of $x(t)$.

Initial-value problems for higher order equations

$$x^{(n)}(t) = f[t, x(t), \dot{x}(t), \ldots, x^{(n-1)}(t)] \tag{17}$$

with $x^{(k)}(t_0)$ given for $k = 0, \ldots, n-1$, are handled in much the same way as the first-order problem. Here,

$$a_k = \frac{1}{k!} x^{(k)}(t_0)(t - t_0)^k, \quad k = 0, 1, \ldots, n-1$$

so the coefficients of the Taylor polynomial $x_{n-1}(t)$ are known. From these, the coefficients $b_0, \ldots, b_{n-1}$ of f_{n-1} $[t, x(t), \ldots, x^{(n-1)}(t)]$ are calculated. Then Eq. (19) gives a_n and likewise subsequent coefficients of $x(t)$. Another method is to transform higher order differential equations into a system of first-order equations. This is done by the substitutions $x_k(t) = x^{(k)}(t)$, $k = 1, \ldots, n-1$ that give

$$\dot{x}_1(t) = x_2(t)$$

and

$$\dot{x}_2(t) = x_3(t)$$
$$\cdots$$
$$\dot{x}_{n-1}(t) = f[t, x_1(t), \ldots, x_{n-1}(t)]$$

This is a special case of the general first-order system

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}[t, \boldsymbol{x}(t)] \tag{18}$$

where $\boldsymbol{x}(t) = [x_1(t), \ldots, x_m(t)]$ and $\boldsymbol{f}(t) = [f_1(t, \boldsymbol{x}(t)], \ldots, f_m[t, \boldsymbol{x}(t)]$ are m-dimensional vectors of functions of t. It is assumed that $\boldsymbol{f}[t, \boldsymbol{x}(t)]$ is a known function of its arguments. Given the initial condition

$$\boldsymbol{x}(t_0) = \boldsymbol{a}_0 = (a_{10}, a_{20}, \ldots, a_{m0})$$

the Taylor expansion of $\boldsymbol{x}(t)$ is carried out similarly as for a single equation, but of course more arithmetic is involved (7). It is assumed as before that the functions $f_1(t), \ldots, f_m(t)$ have representations suitable for automatic differentiation.

For example, recurrence relationships for the standard functions $c(t) = \cos x(t)$ and $s(t) = \sin x(t)$, are obtained from the first-order system

$$\dot{c}(t) = -s(t)\dot{x}(t)$$

and

$$\dot{s}(t) = c(t)\dot{x}(t)$$

which these functions satisfy, together with the initial conditions $c(t_0) = \cos x(t_0)$, $s(t_0) = \sin x(t_0)$, that is, $c_0 = \cos a_0$, $s_0 = \sin a_0$, where $c(t) \leftrightarrow (c_0, c_1, \ldots)$, $s(t) \leftrightarrow (s_0, s_1, \ldots)$, and $x(t) \leftrightarrow (a_0, a_1, \ldots)$. The results are

$$c_{n+1} = -\frac{1}{n+1} \sum_{k=0}^{n} (n - k + 1) a_{n-k+1} s_k$$

and (19)

$$s_{n+1} = \frac{1}{n+1} \sum_{k=0}^{n} (n - k + 1) a_{n-k+1} c_k$$

$n = 0, 1, 2, \ldots$ (see Refs. 6 and 7). Equations (11) and (19) are sufficient to compute an arbitrary number of Taylor coefficients of the function defined by the code list in Eq. (3), given the values of t_0 and t.

Further work on solving differential equations by automatically generating Taylor series has been done by Chang and Corliss (8) and the computer program ATOMFT (9) developed for this purpose. Using interval arithmetic for bounding solutions of initial-value problems, as originated by Moore (see Ref. 7), runs into a technical problem called the "wrapping effect," when applied to systems. This causes interval error bounds to increase unrealistically rapidly. Moore (10) proposed automatic coordinate transformations to alleviate this problem. Further work by Lohner (11) produced an efficient method for minimizing the wrapping effect that is implemented in the computer program AWA.

FIRST-ORDER PARTIAL DERIVATIVES

Automatic differentiation is also effective for evaluating partial derivatives and Taylor coefficients of functions of several variables. For example, suppose that the resonant frequency $f = f(R, L, C)$ of a certain circuit is defined by the formula

$$f = \frac{1}{2\pi}\sqrt{\frac{1}{LC} - \left(\frac{R}{L}\right)^2} \tag{20}$$

AD requires algorithmic representations of functions, in this case, the code list

$$\begin{aligned} s_1 &= R \\ s_2 &= L \\ s_3 &= C \\ s_4 &= s_2 s_3 \\ s_5 &= 1/s_4 \\ s_6 &= s_1/s_2 \\ s_7 &= \mathrm{sqr}(s_6) \\ s_8 &= s_5 - s_7 \\ s_9 &= \mathrm{sqrt}(s_8) \\ s_{10} &= (1/2\pi)s_9 \end{aligned} \tag{21}$$

for evaluating $f(R, L, C)$ at given values of R, L, and C. In Eq. (30), the standard functions $\mathrm{sqr}(s) = s^2$ and $\mathrm{sqrt}(s) = \sqrt{s}$ have been introduced for convenience, and it is assumed that the value of the constant $1/2\pi$ is available as a single quantity.

Values of the partial derivatives $\partial f/\partial R$, $\partial f/\partial L$, and $\partial f/\partial R$ are useful in a number of applications. For example, $\partial f/\partial R$ can be taken as a measure of the *sensitivity* of the value of f to a change in R with L and C held constant because $\Delta f = f(R+\Delta R, L, C) - f(R, L, C)$ is approximately equal to $(\partial f/\partial R)\,\Delta R$ in this case, and similarly for the other variables. Partial derivatives are also used to estimate the impact of round-off error on final results of calculations (12), and the *gradient* of f, which is the vector

$$\nabla f = \left(\frac{\partial f}{\partial R}, \frac{\partial f}{\partial L}, \frac{\partial f}{\partial C}\right)$$

appears in optimization and other problems.

The obvious, but usually not the most efficient way to evaluate partial derivatives is to apply the rules for differentiation to the code list for the function, as in the case of ordinary derivatives and Taylor coefficients. In terms of differentials, this gives

$$\begin{aligned}
ds_1 &= dR\\
ds_2 &= dL\\
ds_3 &= dC\\
ds_4 &= s_2 ds_3 + s_3 ds_2\\
ds_5 &= -ds_4/\mathrm{sqr}(s_4)\\
ds_6 &= (ds_1 - s_6 ds_2)/s_2\\
ds_7 &= 2s_6 ds_6\\
ds_8 &= ds_5 - ds_7\\
ds_9 &= ds_8/(2s_9)\\
ds_{10} &= (1/2\pi) ds_9
\end{aligned} \tag{22}$$

The result of evaluating Eq. (32) along with Eq. (30) is the *total differential* $df = ds_{10}$ of f,

$$df = \frac{\partial f}{\partial R}dR + \frac{\partial f}{\partial L}dL + \frac{\partial f}{\partial C}dC$$

(see Refs. 5 and 6). This result is the same as the normalized Taylor coefficient f_1 of f computed from the Taylor polynomials of degree one with coefficients (R, dR), (L, dL), and (C, dC), respectively. It is evident that the value of $\partial f/\partial R$ is obtained from Eq. (32) for $dR = 1$, $dL = 0$, $dC = 0$, and similarly for the other two partial derivatives of f. Thus, the computational sequence Eq. (32) has to be repeated three times to obtain the components of the gradient ∇f of f. This method is called the *forward* mode of AD because the intermediate calculations are done in the same order as in the code list in Eq. (30) for evaluating the function. An often more efficient method is the *reverse* mode described in the next section.

Note that if $(LC)^{-1} < (R/L)^2$, then evaluating f in real arithmetic by Eq. (30) breaks down at s_9 because of a negative argument for the square root, whereas for $(LC)^{-1} = (R/L)^2, f = 0$ but differentiation breaks down at ds_9 because of the indicated division by $s_9 = 0$.

As pointed out by Wengert (5), higher partial derivatives can be recovered from Taylor coefficients. If the Taylor polynomials with coefficients $(R, dR, 0)$, $(L, dL, 0)$, and $(C, dC, 0)$ are substituted for the respective variables, then the value of the second normalized Taylor coefficient $f_2 = f_2(dR, dL, dC)$ of the function is given by

$$f_2 = \frac{1}{2}\left(\frac{\partial^2}{\partial R^2}dR^2 + \frac{\partial^2 f}{\partial L^2}dL^2 + \frac{\partial^2 f}{\partial C^2}dC^2\right) + \frac{\partial^2 f}{\partial R \partial L}dRdL + \frac{\partial^2 f}{\partial R \partial C}dRdC + \frac{\partial^2 f}{\partial L \partial C}dLdC$$

Thus, for $dR = 1$, $dL = dC = 0$, the value of the second partial derivative with respect to R is given by $\partial^2 f/\partial R^2 = 2f_2(1, 0, 0)$, and similarly for $\partial^2 f/\partial L$ and $\partial^2 f/\partial^C$. Then the values of the mixed, second partial derivatives are computed by solving linear equations. For example, the choice $dR = dL = 1$, $dC = 0$ gives

$$\frac{\partial^2}{\partial R \partial L} = f_2(1, 1, 0) - 2f_2(1, 0, 0) - 2f_2(0, 1, 0)$$

The method of code transformation (6) is likewise applicable to evaluating individual partial derivatives or gradient vectors. The idea is to obtain code lists from Eq. (32) that contain the needed entries. For example, the code list

$$\begin{aligned}
dr_1 &= dR\\
dr_2 &= dr_1/s_2\\
dr_3 &= 2s_6\\
dr_4 &= dr_2 dr_3\\
dr_5 &= -dr_4\\
dr_6 &= 2s_9\\
dr_7 &= dr_5/dr_6\\
dr_8 &= (1/2\pi) dr_7
\end{aligned}$$

produces the differential coefficient $dr_8 = (\partial f/\partial R)dR$. Similar code lists for $(\partial f/\partial L)dL$ and $(\partial f/\partial C)dC$ can be adjoined to the code list in Eq. (30) for $f(R, L, C)$. This increases the length of the code list by approximately a factor of three and results in the corresponding increase in computational effort for evaluating the gradient, compared to the value of the function alone. Once the code lists for the first partial derivatives have been formed, they are used to construct code lists for second partial derivatives, and so on. This leads eventually to a large amount of code compared to the repetitive generation of Taylor coefficients described previously.

GRADIENTS IN REVERSE MODE

The reverse mode of automatic differentiation appears in the 1976 paper by Linnainmaa (13), the Ph.D. thesis of Speelpennig (14), and later in the paper (12) by Iri. As the name implies, this computation proceeds in the reverse order of the sequence of operations used to evaluate the function. For example, consider the function $f(R, L, C)$ defined by the code list in Eq. (30). The first partial derivatives of this function are

$$\begin{aligned}
\frac{\partial f}{\partial R} &= \frac{\partial s_{10}}{\partial s_1}\\
\frac{\partial f}{\partial L} &= \frac{\partial s_{10}}{\partial s_2}
\end{aligned}$$

and

$$\frac{\partial f}{\partial C} = \frac{\partial s_{10}}{\partial s_3}$$

These quantities are obtained by differentiating s_{10} beginning with s_{10}, then working backward through the code list:

$$\begin{aligned}
&\frac{\partial s_{10}}{\partial s_{10}} = 1\\
&\frac{\partial s_{10}}{\partial s_9} = \frac{1}{2\pi}\\
&\frac{\partial s_{10}}{\partial s_3} = \frac{\partial s_{10}}{\partial s_9}\frac{\partial s_9}{\partial s_3} = \frac{\partial s_{10}}{\partial s_9}\frac{1}{2s_9}\\
&\frac{\partial s_{10}}{\partial s_7} = \frac{\partial s_{10}}{\partial s_3}\frac{\partial s_3}{\partial s_7} = -\frac{\partial s_{10}}{\partial s_3}\\
&\frac{\partial s_{10}}{\partial s_6} = \frac{\partial s_{10}}{\partial s_7}\frac{\partial s_7}{\partial s_6} = \frac{\partial s_{10}}{\partial s_7}2s_6\\
&\frac{\partial s_{10}}{\partial s_5} = \frac{\partial s_{10}}{\partial s_3}\frac{\partial s_3}{\partial s_5} = \frac{\partial s_{10}}{\partial s_3}\\
&\frac{\partial s_{10}}{\partial s_4} = \frac{\partial s_{10}}{\partial s_5}\frac{\partial s_5}{\partial s_4} = -\frac{\partial s_{10}}{\partial s_5}\frac{1}{\mathrm{sqr}(s_4)}\\
&\frac{\partial s_{10}}{\partial s_3} = \frac{\partial s_{10}}{\partial s_4}\frac{\partial s_4}{\partial s_3} = \frac{\partial s_{10}}{\partial s_4}s_2\\
&\frac{\partial s_{10}}{\partial s_2} = \frac{\partial s_{10}}{\partial s_6}\frac{\partial s_6}{\partial s_2} + \frac{\partial s_{10}}{\partial s_4}\frac{\partial s_4}{\partial s_2} = -\frac{\partial s_{10}}{\partial s_6}\frac{s_1}{\mathrm{sqr}(s_2)} + \frac{\partial s_{10}}{\partial s_4}s_3\\
&\frac{\partial s_{10}}{\partial s_1} = \frac{\partial s_{10}}{\partial s_6}\frac{\partial s_6}{\partial s_1} = \frac{\partial s_{10}}{\partial s_6}\frac{1}{s_2}
\end{aligned} \tag{23}$$

In this simple example, 19 arithmetic operations are required to evaluate the partial derivatives of f in reverse mode, whereas the forward mode based on Eq. (32) takes 24 operations. The following analysis indicates that the savings are generally greater as the number of independent variables increases.

In general, suppose that the function $f = f(x_1, \ldots, x_m)$ of m variables is represented by the code list $s = (s_1, \ldots, s_n)$, where the values of $x_1, \ldots, x_m$ are assigned to $s_1, \ldots, s_m$, respectively. The forward and reverse modes of AD result from applying the chain rule to s in different ways. In the forward mode,

$$\frac{\partial t_i}{\partial x_j} = \sum_{k\in K_i} \frac{\partial t_i}{\partial t_k}\frac{\partial t_k}{\partial x_j}, \quad i = 1, \ldots, n; \quad j = 1, \ldots, m \tag{24}$$

where K_i denotes the set of indices $k < i$ such that s_i depends explicitly on s_k. Consequently, there are mn quantities to evaluate in this case. For the reverse mode, let I_j denote the set of indices $i > j$ such that s_i depends explicitly on s_j. Then,

$$\frac{\partial s_n}{\partial s_n} = 1$$

and

$$\frac{\partial s_n}{\partial s_j} = \sum_{i\in I_j} \frac{\partial s_n}{\partial s_i}\frac{\partial s_i}{\partial s_j}, \quad j = n-1, \ldots, 1 \tag{25}$$

giving a total of $(n-1)$ quantities to evaluate. Thus the computational effort in the reverse mode is independent of the number of variables m instead of increasing proportionally as in the forward mode. A more detailed analysis of complexity takes into account that the sets K_i contains at most two indices, whereas I_j may contain as many as $(n-j)$ (15).

The method of code transformation applied to the computation in Eq. (39) gives a code list for the gradient in the reverse mode;

$$\begin{aligned}
g_1 &= 1/2\pi && (= \partial s_{10}/\partial s_9)\\
g_2 &= 2s_9\\
g_3 &= 1/g_2\\
g_4 &= g_1 g_3 && (= \partial s_{10}/\partial s_3)\\
g_5 &= -g_4 && (= \partial s_{10}/\partial s_7)\\
g_6 &= 2s_6\\
g_7 &= g_5 g_6 && (= \partial s_{10}/\partial s_6)\\
g_8 &= \mathrm{sqr}(s_4)\\
g_9 &= 1/g_8\\
g_{10} &= g_4 g_9\\
g_{11} &= -g_{10} && (= \partial s_{10}/\partial s_4)\\
g_{12} &= s_2 g_{11} && (= \partial s_{10}/\partial s_3 = \partial f/\partial C)\\
g_{13} &= \mathrm{sqr}(s_2)\\
g_{15} &= g_7 g_{14}\\
g_{16} &= -g_{15}\\
g_{17} &= g_{11} s_3\\
g_{18} &= g_{16} + g_{17} && (= \partial s_{10}/\partial s_2 = \partial f/\partial L)\\
g_{19} &= 1/s_2\\
g_{20} &= g_7 g_{19} && (= \partial s_{10}/\partial s_1 = \partial f/\partial R)
\end{aligned} \tag{26}$$

where the trivialities $\partial s_{10}/\partial s_{10} = 1$ and $\partial s_{10}/\partial s_5 = \partial s_{10}/\partial s_8$ have been omitted from the computation. Now if desired, reverse mode AD is applied to the code list in Eq. (43) to obtain higher partial derivatives.

CODE LISTS, PROGRAMS, AND COMPUTATIONAL GRAPHS

In early papers on AD, it was simply assumed implicitly that the function of interest is expressed as a composition of elementary operations to which the rules of differentiation are applied. Then the chain rule guarantees that this composite function is correctly differentiated. Explicit formulation of code lists followed a little later (3). Precise definitions were given by Kedem (16), who also extended the idea of AD from code lists to computer subroutines and entire programs.

Technically speaking, a *code list* is a sequence $s = \{s_1, \ldots, s_n\}$ in which each *entry* s_i is (1) an *assignment* of the form $s_i = t$, where the value of t is known, (2) arithmetic operation $s_i = s_j \circ s_k, j, k < i$ involving previous entries, where $\circ$ denotes addition, subtraction, multiplication, or division, or (3) $s_i = \phi(s_j), j < i$, where s_j is a previous entry and the function ϕ is one of a known set of *standard functions* (or *library functions*), such as sine, cosine, and square root, available as subroutines or built into computer hardware. Before the advent of electronic calculators and computers, functions were also evaluated in this way with tabulated or easily computed functions comprising the set of standard functions but without much attention to the actual process. AD depends on an explicit formulation of the sequence of steps in the evaluation process and, of course, differentiability of the individual operations and standard functions. These steps consist of specifying one or more input variables, followed by the calculating of intermediate variables and finally the output variables, giving the desired function values.

Because computers require exact specification of the sequence of operations to be performed, one of the first advances in computer science was formula translation, that is, conversion of formulas to equivalent code lists for functional evaluation (1). In addition, the advent of computers focused attention on algorithms, that is, step-by-step methods for functional evaluation rather than formulas. For example, the solutions of linear systems of equations are functions of the coefficients of the system matrix and the components of the right-hand side. Cramer's rule gives formulas for these solutions in terms of determinants, but these are essentially useless for actual computation. Instead, linear systems are generally solved by an elimination algorithm (17). If the data of the problem depend on one or more variables, then AD is applied to this process to obtain values of derivatives of the solutions. The same applies to other functions defined by algorithms, as embodied in computer subroutines or entire programs.

In the previous sections, the principles of AD were developed for functions defined by code lists, sometimes called "straight-line code." Generally, computer subroutines and programs contain loops and branches in addition to straight-line code. Although these do not affect AD, in principle, certain practical problems arise [see Ref. 16 and the paper by Fischer (18)].

A loop is simply a set of instructions repeated a fixed or variable number of times. Thus, a loop can be "unrolled" into a segment of straight-line code which is longer than the original by the same factor. If the number of repetitions is fixed, this presents no essential difficulty other than the usual ones of computational time and storage required.

A branch occurs in a computational routine if different sets of instructions are executed under different conditions. For example, the value of $\mathrm{abs}(x) = |x|$ for real x is calculated to be x if $x \geq 0$, or $-x$ otherwise. If a branch occurs, then AD yields the value of the derivative of the function computed by the branch actually taken, provided, of course, that this function is differentiable. For the standard function $\mathrm{abs}(x)$, $\mathrm{abs}'(x) = -1$ for $x < 0$, $\mathrm{abs}'(x) = 1$ for $x > 0$, whereas AD terminates for $x = 0$ if properly implemented.

A useful tool for analyzing computer programs is the *computational graph,* introduced by L. V. Kantorovich (19). For example, Fig. 1 shows diagrammatically how to evaluate the function given by Eq. (29) according to the equivalent code list in Eq. (30). The nodes of this graph indicate the operations to be performed on the input variables. Now the automatic differentiation process is visualized as transformation of the computational graph corresponding to the code transformation described before. This transformation is carried out in forward mode by Eq. (8) or reverse mode by Eq. (15). Because the input variables are conventionally placed at the bottom of the computational graph, the reverse mode is referred to as "top down" whereas the forward mode is "bottom up" in this terminology.

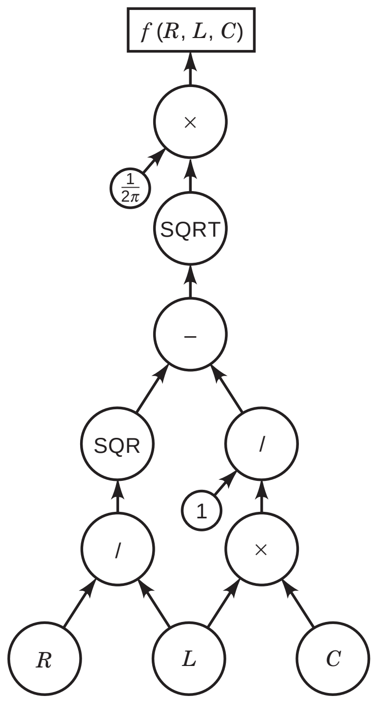

Figure 1. Computational graph of $f(R, L, C)$.

Computational graphs form what are called *directed acyclic graphs* (20). Known results from the theory of these graphs are used to analyze the automatic differentiation process and its complexity (12, 15). To automate the results of graph theory, the nodes of a computational graph are numbered, and the edge from node i to node j is designated by the ordered pair (i, j). Then the operation performed at node i determines the result of differentiation. This leads to a matrix representation of the process of automatic differentiation. See Ref. 2 for a matrix-vector formulation of the forward and reverse modes of gradient computation.

DIFFERENTIATION ARITHMETICS

The process of automatic differentiation has an equivalent formulation as a mathematical system in which the operations yield values of derivatives in addition to values of functions (22). It is evident that a code list such as Eq. (3) can be evaluated if arithmetic operations and standard functions are defined for the quantities involved. For example, complex or interval arithmetic (7) could be used to evaluate Eq. (3) instead of real arithmetic. Instead, consider the set of ordered pairs $U = (u, \overset{2}{u})$ where with addition and subtraction are defined by $U \pm V = (u \pm v, \overset{2}{u} \pm \overset{2}{v})$, respectively, multiplication by $UV = (uv, u\overset{2}{v} + v\overset{2}{u})$, and division by $U/V = \{u/v, [\overset{2}{u} - (u/v)\,\overset{2}{v}]/v\}$ for $v \neq 0$. In this system, the sine function is defined as $\sin U = (\sin u, \overset{2}{u} \cos u)$. Now if the evaluation of Eq. (3) begins with $s_1 = (t, 1)$ and constants, such as Ω represented by $(\Omega, 0)$, then the result is $s_7 = (I(t), \overset{2}{I}(t))$ which gives the values of the function and its derivative. Here, the rules of differentiation are built into the definitions of the arithmetic operations and standard functions.

A direct generalization of the previous example is *Taylor arithmetic.* Here, the elements are $(d+1)$-dimensional vectors $U=(u_0, u_1, \ldots, u_d)$ corresponding to the coefficients of a Taylor polynomial of degree d. In this arithmetic, addition and subtraction are defined by Eq. (10), and multiplication and division are given by Eqs. (11) and (10), respectively. Representations of standard functions are derived as previously, for example, $\exp(u_0, \ldots, u_d)=(v_0, \ldots, v_d)$, and $v_0=\exp(u_0)$ and $v_1, \ldots, v_d$ are given by Eq. (15). In this arithmetic, the independent variable is represented by $(t_0, t-t_0, 0, \ldots, 0)$ for Taylor expansion at t_0, and constants, such as Ω, by $(\Omega, 0, \ldots, 0)$. With these as inputs, evaluating Eq. (3) in Taylor arithmetic gives the coefficients of the Taylor polynomial of degree d of $I(t)$ expanded at t_0. More generally, if an arbitrary Taylor polynomial is given as the input variable, then the result of the evaluation process is the corresponding Taylor polynomial of the output.

Differentiation arithmetics are also available for automatically evaluating functions of several variables and their partial derivatives. The simplest is *gradient* arithmetic with elements $(f, \nabla f)$, where $\nabla f=(f_1, \ldots, f_m)$ is an m-dimensional vector. Arithmetic operations in this arithmetic are defined by

$$\begin{aligned}(f, \nabla f) \pm (g, \nabla g) &= (f \pm g, \nabla f \pm \nabla g)\\ (f, \nabla f)(g, \nabla g) &= (fg, f\nabla g + g\nabla f)\\ (f, \nabla f)/(g, \nabla g) &= \{f/g, [\nabla f - (f/g)\nabla g]/g\}, g \neq 0\end{aligned}$$

as before. If $\phi(x)$ is a differentiable standard function of the single variable x, then the definition of this function in gradient arithmetic is $\phi(f, \nabla f)=[\phi(f), \phi'(f)\ \nabla f]$ by the chain rule. The independent variables $x_1, \ldots, x_m$ are represented by $(x_i, \boldsymbol{e}_i)$, where $\boldsymbol{e}_i$ is the ith unit vector, $i=1, \ldots, m$, and constants c by $(c, \mathbf{0})$ because the gradient of a constant is the zero vector $\mathbf{0}=(0, \ldots, 0)$. Thus evaluating Eq. (30) in gradient arithmetic with the inputs $s_1=[R, (1, 0, 0)]$, $s_2=[L, (0, 1, 0)]$, $s_3=[C, (0, 0, 1)]$ gives the output $(f, \nabla f)$, the values of the function $f=f(R, L, C)$, and its gradient vector. Gradient arithmetic also applies if the input variables are functions of other variables. As long as the values and gradients of the input variables are known, the values and gradients of the output variables are computed correctly by gradient arithmetic.

Straightforward evaluation of a code list, such as Eq. (30) in gradient arithmetic is a forward-mode computation, often less efficient than reverse mode. This comparison applies to serial computation. If a parallel computer with sufficient capacity to compute the components of $(s, \nabla s)$ simultaneously is available, then only one evaluation of the code list in gradient arithmetic is required. When it is simpler to program just the parallel evaluation of ∇s, then two passes through the code list are required, one for the function value and the next for its gradient.

Differentiation arithmetics for higher partial derivatives are constructed according to the same pattern. The $(m \times m)$ symmetrical matrix

$$\boldsymbol{H}f = \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]$$

of second partial derivatives is called the *Hessian* of the function $f=f(x_1, \ldots, x_m)$ of m variables. The corresponding *Hessian arithmetic* is based on the definition of arithmetic operations and standard functions for the triples $(f, \nabla f, Hf)$ representing the value, gradient vector, and Hessian matrix of a function. For details, see Refs. 2 and 22.

When based on real or complex arithmetic, differentiation arithmetics form a mathematical system called a commutative ring with identity. Performed in interval arithmetic (7), differentiation arithmetics give lower and upper bounds for the Taylor coefficients or partial derivatives to take into account the possibilities of inexact data and round-off error in the computation. Bounds on Taylor coefficients are useful for determining the accuracy of Taylor polynomial approximations to solutions of differential equations and other functions.

SOME APPLICATIONS OF AUTOMATIC DIFFERENTIATION

Automatic differentiation is applicable to the wide variety of computational problems that require evaluation of derivatives. The solution of initial-value problems has been described previously. Other applications of AD to scientific and engineering problems are in the conference proceedings Refs. 23 and 24. Here, brief descriptions of applying AD to solving nonlinear systems of equations, optimization, implicit differentiation, and differentiation of inverse functions are given.

Computational solution of nonlinear equations is usually carried out by iterative algorithms that yield a sequence of improved approximations to solutions, if successful. For a single equation $f(x)=0$, *Newton's method,*

$$x_{n+1} = x_n - [f'(x_n)]^{-1} f(x_n), \quad n = 0, 1, 2, \ldots$$

yields a sequence that converges rapidly to a solution $x=x^*$ of the equation, if the initial approximation x_0 is good enough and $f'(x^*) \neq 0$ (3). This method generalizes immediately to the m-dimensional problem $\boldsymbol{f}(\boldsymbol{x})=\mathbf{0}$, where $\boldsymbol{x}=(x_1, \ldots, x_m)$ and $bf\,f(\boldsymbol{x})=[f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})]$. The derivative of the transformation $\boldsymbol{f}$ is represented by the $(m \times m)$ Jacobian matrix

$$\boldsymbol{f}'(\boldsymbol{x}) = \left[\frac{\partial f_i(\boldsymbol{x})}{\partial x_j}\right]$$

and the Newton step $\Delta\boldsymbol{x}_n = \boldsymbol{x}_{n+1} - \boldsymbol{x}_n$ is obtained by solving the linear system of equations

$$\boldsymbol{f}'(\boldsymbol{x}_n)\Delta\boldsymbol{x}_n = -\boldsymbol{f}(\boldsymbol{x}_n)$$

The rows of the Jacobian matrix $\boldsymbol{f}'(\boldsymbol{x}_n)$are the gradients of the component functions $f_i(\boldsymbol{x}_n)$ and are computable by AD in forward or reverse mode. Conditions for the convergence of Newton's method and bounds for the error $\boldsymbol{x}^* - \boldsymbol{x}_n$ are verified on the basis of evaluation of the Hessians $Hf_i(\boldsymbol{x}_n)$ by AD in interval arithmetic (3). It is also possible to compute Newton steps by solving a large, sparse, linear system of equations based on differentiating the code list for values of the component functions (25).

A simple optimization problem is to find maximum or minimum values of a real function $\phi(x) = \phi(x_1, \ldots, x_m)$ of m variables. If this function is differentiable, then these extremal values are found at one or more of the critical points of the function, which are the solutions of the generally nonlinear system of equations $\nabla\phi(\boldsymbol{x}) = \boldsymbol{0}$. Once the values of the function $\boldsymbol{f}(\boldsymbol{x}) = \nabla\phi(\boldsymbol{x})$ and its Jacobian matrix $\boldsymbol{f}'(\boldsymbol{x}) = H\phi(\boldsymbol{x})$ are obtained by AD, calculating critical points proceeds by Newton's method or some other optimization method based on evaluating the Newton step. Optimization involving constraint functions is handled similarly by AD, after introducing Lagrange multipliers (22).

In addition to functions defined explicitly in terms of the input variables, AD is also applicable to functions defined implicitly. For example, suppose that the relationship $f(x, t) = 0$ defines $x = x(t)$. From the calculation of $\nabla f(x, t)$ by AD, the value of $\overset{2}{x}(t)$ is given by the usual formula $\overset{2}{x}(t) = -(\partial f/\partial t)/(\partial f/\partial x)$. Similarly, for relationships of the form $f(x_1, \ldots, x_m) = 0$, the gradient $\nabla f(\boldsymbol{x})$ obtained by AD furnishes the coefficients of linear systems of $(m - 1)$ equations for the various partial derivatives $\partial x_i/\partial x_j$.

A special case of implicit differentiation is the differentiation of inverse functions, which are usually not known explicitly, but are obtained by solving equations. In the case of one variable, this means solving the equation $f(y) = x$ for $y = f^{-1}(x) = g(x)$ by Newton's method or some other iterative procedure (3). The iteration is continued until the solution y is considered satisfactorily accurate according to some criterion giving a stopping condition. In principle, AD can be applied to the iterative procedure to obtain corresponding approximations to derivatives and values of the inverse function. However, a more efficient and likely more accurate method is to obtain the value of $f'(x)$ by AD from the known algorithm for $f(x)$, from which $g'(x) = 1/f'(x)$ gives the derivative of the inverse function. It is interesting to note that the value of $g'(x)$ is obtained in this case without the need to evaluate $g(x)$. This applies also to vector-valued functions of several variables in m dimensions. If $\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{f}\boldsymbol{-1}(\boldsymbol{x})$ and if the Jacobian matrix $\boldsymbol{f}'(\boldsymbol{x})$ calculated by AD is invertible, then $\boldsymbol{g}'(\boldsymbol{x}) = [\boldsymbol{f}(\boldsymbol{x})]^{-1}$.

IMPLEMENTATION OF AUTOMATIC DIFFERENTIATION

Methods for implementing AD are interpretation, code transformation, and operator overloading. Interpretive procedures take the code list for a function as input, analyze each entry, and then use the appropriate subroutine to compute the result. This approach is useful, for example, in interactive programs that accept functions entered from the keyboard of a computer. The corresponding code lists are generated and the desired derivatives computed by interpretation of the code list. Interpretation was also used in early AD programs in which the functions to be differentiated were provided by subroutines (6). In noninteractive programs, interpretation is generally less efficient than code transformation.

Code transformation is generally carried out by precompilation. A program written for function evaluation is analyzed and code for the desired derivatives is inserted at the appropriate places. Then the resulting program is compiled for efficient execution. Current examples of precompilers for AD are PADRE2 (26), ADOL-F (27), and ADIFOR 2.0 (28) for programs written in FORTRAN, and ADOL-C (29) for C programs.

The use of precompilers requires some caution. This is particularly true when dealing with what is called "legacy" code which was written previously by someone else without differentiation in mind. Functions are often approximated very accurately by piecewise rational or highly oscillatory functions, but AD applied to these algorithms can yield nonsensical derivative values. See Refs. 16 and 18 for a discussion of problems which arise in the use of AD.

The idea of *operator overloading* is a natural reflection of a common practice in mathematics. For example, the symbol "+" is used to denote the addition of diverse quantities, such as integers, real or complex numbers, vectors, matrices, functions, and so on. The idea is essentially the same in each instance, but the actual operation to be performed differs. Without thinking much about it, a person uses the meaning of the addition symbol aappropriate to the *type* of addends considered. However, computers are ordinarily built to add only integers or floating-point numbers, and early computer languages reflected this limitation of the meaning of + and the types of addends permitted. Later, languages, such as C++ (30), were developed which allow extending the meaning of operator symbols to types of operands defined by the programmer. This is called "overloading" the symbol. The overloaded operations and functions are carried out by appropriate subroutines. The compiler checks types of operands and constructs calls to these subroutines. See Refs. 31 and 32 for examples of automating differentiation arithmetics by operator overloading. In C++, AD is implemented by "class libraries" containing the appropriate definitions, operators, and functions for various differentiation arithmetics (30).

Use of operator overloading to implement AD simplifies programming because functions and subroutines are written in the usual way and the compiled program produces derivative and functional values. Here, differentiation is done at compile time because the compiler generates the sequences of calls to subroutines for evaluating functions and their derivatives. The price for this convenience is that the final computation is carried out in forward mode with the corresponding possible loss of efficiency. As mentioned before, this may not be a drawback in parallel computation.

ALGORITHMIC DIFFERENCING

The algorithmic method also facilitates accurate computation of divided differences

$$f[x+h, x] = \frac{f(x+h) - f(x)}{h},$$

see Refs. 33 and 34. Direct computation of $f\,[x + h, x]$ by subtraction followed by division is problematical in finite-precision arithmetic as h approaches zero due to the fact that $f(x + h)$ and $f(x)$ will agree to more significant digits, and the difference will eventually consist of only round-off error which is then multiplied by the large number

$1/h$. This difficulty is avoided in algorithmic differencing (AΔ) by the use of expressions for divided differences of the arithmetic operations which are numerically stable for $|h|$ small, and approach the values of their derivatives as $h \to 0$ as required by mathematical theory. The postfix operator $[x + h, x]$ is defined for differentiable functions $f(x)$ by

$$f[x+h, x] = \{ \begin{matrix} (f(x+h) - f(x))/h & \text{if } h \neq 0, \\ f'(x) & \text{if } h = 0. \end{matrix} \tag{27}$$

For composite functions $(f \circ g)(x) = f(g(x))$, the *chain rule*

$$(f \circ g)[x+h, x] = f[g(x+h), g(x)] \cdot g[x+h, x] \tag{28}$$

holds as for derivatives. This guarantees that starting the algorithm with the divided difference of the input (for example, $x[x + h, x] = 1$) will yield the divided difference of the output. The AΔ rules for arithmetic operations and intrinsic functions are obtained as in elementary calculus as expressions which give derivatives as $h \to 0$. For example, the divided-difference expression for the quotient is

$$(f/g)[x+h, x] = (f[x, x+h] - (f/g)g[x, x+h])/(g+h),$$

which gives the derivative formula $(f/g)' = (f' - (f/g)g')/g$ for $h = 0$. Similarly, if $f(x) = \arctan g(x)$, one has

$$f[x, x+h] = \frac{1}{h} \arctan(\frac{hg[x, x+h]}{1 + g(u + g[x, x+h])}),$$

for $h \neq 0$, and the derivative

$$f[x, x] = f'(x) = \frac{u'(x)}{1 + g^2(x)}$$

for $h = 0$, and so on.

The arithmetic operations and standard functions for AD are the special cases of those for AΔ with $h = 0$. Thus, a computer program to implement AΔ can be used to provide values of derivatives, divided differences (or differences $f(x+h) - f(x) = h\, f[x+h, x]$) of equivalent accuracy for a wide range of values of h.

BIBLIOGRAPHY

1. A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1988.
2. L. B. Rall, G. F. Corliss, Introduction to automatic differentiation, inM. Berz *et al.* (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, Philadelphia: SIAM, 1996.
3. L. B. Rall, *Computational Solution of Nonlinear Operator Equations*, New York: Wiley, 1969. Reprint, Huntington, NY: Krieger, 1979.
4. R. E. Moore, *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. Thesis, Stanford University, 1962.
5. R. E. Wengert, A simple automatic derivative evaluation program, *Commun. ACM*, **7** (8): 463–464, 1964.
6. L. B. Rall, *Automatic Differentiation: Techniques and Applications*, New York: Springer, 1981.
7. R. E. Moore, *Methods and Applications of Interval Analysis*, Philadelphia: SIAM, 1979.
8. Y. F. Chang, G. F. Corliss, Solving ordinary differential equations using Taylor series, *ACM Trans. Math. Softw.*, **8**: 114–144, 1982.
9. Y. F. Chang, G. F. Corliss, ATOMFT: Solving ODE's and DAE's using Taylor series, *Comput. Math. Appl.*, **28**: 209–233, 1994.
10. R. E. Moore, Automatic local coordinate transformations to reduce the growth of error bounds in interval computation of solutions of ordinary differential equations, inL. B. Rall (ed.), *Error in Digital Computation*, Vol. 2, New York: Wiley, 1965.
11. R J. Lohner, Enclosing the solutions of ordinary initial and boundary value problems, inE. W. Kaucher, U. W. Kulisch, andC. Ullrich (eds.), *Computer Arithmetic: Scientific Computation and Programming Languages*, Stuttgart: Wiley-Teubner, 1987.
12. M. Iri, Simultaneous computation of function, partial derivatives and estimates of rounding errors—Complexity and practicality, *Jpn. J. Appl. Math.*, **1**: 223–252, 1984.
13. S. Linnainmaa, Taylor expansion of the accumulated rounding error, *BIT*, **16**: 146–160, 1976.
14. B. Speelpennig, *Computing Fast Partial Derivatives of Functions Given by Algorithms*, Ph.D. Thesis, University of Illinois, 1980.
15. A. Griewank, Some bounds on the complexity of gradients, Jacobians, and Hessians, inP. M. Pardalos (ed.), *Complexity in Nonlinear Optimization*, Singapore: World Scientific, 1993.
16. G. Kedem, Automatic differentiation of computer programs, *ACM Trans. Math. Softw.*, **6**: 150–165, 1980.
17. G. Forsythe, C. B. Moler, *Computer Solutions of Linear Algebraic Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1967.
18. H. Fischer, Special problems in automatic differentiation, inA. Griewank andG. F. Corliss (eds.), *Automatic Differentiation of Algorithms, Theory, Implementation, and Application*, Philadelphia: SIAM, 1992.
19. L. V. Kantorovich, On a mathematical symbolism convenient for performing mathematical calculations, Russian, *Dokl. Akad. Nauk USSR*, **113**: 738–741, 1957.
20. C. W. Marshall, *Applied Graph Theory*, New York: Wiley, 1971.
21. L. B. Rall, Gradient computation by matrix multiplication, inH. Fischer, B. Riedmüller, andS. Schäffler (eds.), *Applied Mathematics and Parallel Computing*, Heidelberg: Physica-Verlag, 1996.
22. L. B. Rall, Differentiation arithmetics, inC. Ullrich (ed.), *Computer Arithmetic and Self-Validating Numerical Methods*, New York: Academic Press, 1990.
23. A. Griewank, G. F. Corliss (eds.), *Automatic Differentiation of Algorithms, Theory, Implementation, and Applications*, Philadelphia: SIAM, 1992.
24. M. Berz *et al.* (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, Philadelphia: SIAM, 1996.
25. A. Griewank, Direct calculation of Newton steps without accumulating Jacobians, inT. F. Coleman andY. Li (eds.), *Large-Scale Numerical Optimization*, Philadelphia: SIAM, 1990.
26. K. Kubota, PADRE2—Fortran precompiler for automatic differentiation and estimates of rounding errors, inM. Berz *et al.* (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, Philadelphia: SIAM, 1996.
27. D. Shiriaev, A. Griewank, ADOL-F: Automatic differentiation of Fortran codes, inM. Berz *et al.* (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, Philadelphia: SIAM, 1996.

28. C. Bischof, A. Carle, Users' experience with ADIFOR 2.0, inM. Berz *et al.* (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, Philadelphia: SIAM, 1996.
29. D. W. Juedes, A taxonomy of automatic differentiation tools, inA. Griewank andG. F. Corliss (eds.), *Automatic Differentiation of Algorithms, Theory, Implementation, and Applications*, Philadelphia: SIAM, 1991.
30. B. Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1987.
31. L. B. Rall, Differentiation and generation of Taylor coefficients in Pascal-SC, inU. W. Kulisch andW. L. Miranker (eds.), *A New Approach to Scientific Computation*, New York: Academic Press, 1983.
32. L. B. Rall, Differentiation in Pascal-SC, type GRADIENT,*ACM Trans. Math. Softw.*, **10**: 161–184, 1984.
33. L. B. Rall and T. W. Reps, Algorithmic differencing, In U. Kulisch, R. Lohner, and A. Facius (eds.), *Perspectives on Enclosure Methods*, Springer-Verlag, Vienna, 2001.
34. T. W. Reps and L. B. Rall, Computational divided differencing and divided-difference arithmetics, *Higher-Order and Symbolic Computation*, **16**, 93–149, 2003.

LOUIS B. RALL
GEORGE F. CORLISS
Department of Mathematics University of Wisconsin-Madison, 480 Lincoln Drive, Madison, WI
Department of Electrical and Computer Engineering Marquette University, Milwaukee, WI