Chapter 1

Introduction

The basic idea behind the differentiation methodology explored in this book is by no means new. It has been rediscovered and implemented many times, yet its application still has not reached its full potential. It is our aim to communicate the basic ideas and describe the fundamental principles in a way that clarifies the current state of the art and stimulates further research and development. We also believe that algorithmic differentiation (AD) provides many good exercises that require and develop skills in both mathematics and computer science. One of the obstacles in this area, which involves "symbolic" and "numerical" methods, has been a confusion in terminology, as discussed in the prologue. There is not even general agreement on the best name for the field, which is frequently referred to as *automatic* or *computational differentiation* in the literature. For this book the adjective *algorithmic* seemed preferable, because much of the material emphasizes algorithmic structure, sometimes glossing over the details and pitfalls of actual implementations. Readers who are primarily seeking guidance on using AD software should consult the various software sites linked to the Web page `http://www.autodiff.org/`.

Frequently we have a program that calculates numerical values for a function, and we would like to obtain accurate values for derivatives of the function as well. Derivatives play a central role in sensitivity analysis (model validation), inverse problems (data assimilation), and design optimization (simulation parameter choice). At a more elementary level they are used for solving algebraic and differential equations, curve fitting, and many other applications.

Sometimes we seek first-order derivatives, for example, gradient vectors, for a single-target (cost) function, or a Jacobian matrix corresponding to the normals of a set of constraint functions. Sometimes we seek higher-order derivatives, for example, a Hessian times direction vector product or a truncated Taylor series. Sometimes we wish to nest derivatives, for example, getting gradients of the form $\nabla_x F(x, f(x), f'(x))$ from programs for f and F.

The techniques described and analyzed in this book are applied to programs that calculate numerical values in order to produce transformed programs that calculate various derivatives of these values with comparable accuracy and efficiency.

Friends and Relations

At this point, it is worth pausing to distinguish AD from some of the things that it is not. A rather crude way of obtaining approximate numerical derivatives of a function f is the divided difference approach that is sketched in Table 1.1.

$D_{+h}\,f(x) \equiv \dfrac{f(x+h)-f(x)}{h}$ or $D_{\pm h}\,f(x) \equiv \dfrac{f(x+h)-f(x-h)}{2h}$

Table 1.1: How we Do Not compute derivatives

It is easy to see why these difference quotients do not provide accurate values: If h is small, then cancellation error reduces the number of significant figures in $D_{+h}f$, but if h is not small, then truncation errors (terms such as $h^2 f'''(x)$) become significant. Even if h is optimally chosen, the values of D_{+h} and $D_{\pm h}$ will be accurate to only about $\frac{1}{2}$ or $\frac{2}{3}$ of the significant digits of f, respectively. For second- (and higher-)order derivatives these accuracy problems become acute.

In contrast, AD methods incur no truncation errors at all and usually yield derivatives with working accuracy.

Rule 0
Algorithmic differentiation does not incur truncation errors.

To illustrate the difference, let us consider the squared Euclidean norm

$$f(x) = \sum_{i=1}^{n} x_i^2 \quad \text{at} \quad x_i = i \quad \text{for} \quad i = 1 \ldots n \,.$$

Suppose we try to approximate the first gradient component of $f(x)$ at this point by the difference quotient

$$\frac{1}{h}\left[f(x + h\,e_1) - f(x)\right] = \frac{\partial}{\partial x_1} f(x) + h = 2x_1 + h = 2 + h \,,$$

where $e_1 \in \mathbb{R}^n$ is the first Cartesian basis vector and the truncation error term h happens to be exact. No matter how we perform the calculation of $f(x + h\,e_1)$, its value must be represented as a floating point number, which may require a roundoff error of size $f(x + h\,e_1)\varepsilon \approx n^3 \frac{\varepsilon}{3}$, with $\varepsilon = 2^{-54} \backsimeq 10^{-16}$ denoting the machine accuracy (see Exercise 2.2). For $h = \sqrt{\varepsilon}$, as is often recommended, the difference quotient comes with a rounding error of size $\frac{1}{3}n^3\sqrt{\varepsilon} \approx \frac{1}{3}n^3 10^{-8}$. Hence, not even the sign needs to be correct when n is of order 1,000, a moderate dimension in scientific computing. Things are not much better if one selects h somewhat larger to minimize the sum of the roundoff and truncation errors, whose form is normally not known, of course.

In contrast, we will see in Exercise 3.2 that AD yields the ith gradient component of the squared norm as $2\,x_i$ in both its forward and reverse modes.

The last statement is prone to create the impression that somehow derivatives are generated "symbolically." While this is in some sense true, there is in fact a big difference between the techniques advocated here and the ones implemented in current computer algebra packages. In AD tools the algebraic expression $2\,x_i$ is never generated; instead the numerical value of x_i is multiplied by 2 and then returned as the value of the gradient component. This distinction between AD and fully symbolic computing may seem almost intangible on the squared Euclidean norm, but it should become clear in the following example.

Consider the product example due to Speelpenning [Spe80]:

$$f(x) = \prod_{i=1}^{n} x_i = x_1 * x_2 * \cdots * x_n \ .$$

The corresponding gradient has the symbolic form

$$\begin{aligned}
\nabla f(x) &= \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right) = \left(\prod_{j\neq i} x_j\right)_{i=1\ldots n} \\
&= \big(x_2 * x_3 * \cdots \quad \cdots * \ x_i \ * x_{i+1} * \cdots \quad \cdots * x_{n-1} * \ x_n \ , \\
&\quad\ x_1 * x_3 * \cdots \quad \cdots * \ x_i \ * x_{i+1} * \cdots \quad \cdots * x_{n-1} * \ x_n \ , \\
&\quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\quad\ x_1 * x_2 * \cdots \quad \cdots * x_{i-1} * x_{i+1} * \cdots \quad \cdots * x_{n-1} * \ x_n \ , \\
&\quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\quad\ x_1 * x_2 * \cdots \quad \cdots * x_{i-1} * \ x_i \ * \cdots \quad \cdots * x_{n-2} * \ x_n \ , \\
&\quad\ x_1 * x_2 * \cdots \quad \cdots * x_{i-1} * \ x_i \ * \cdots \quad \cdots * x_{n-2} * x_{n-1}\big) \ .
\end{aligned}$$

As we have emphasized by the generous allocation of space to this display, symbolic expressions can take up a lot of memory or screen without necessarily being all that useful to look at. Obviously, there are a lot of common subexpressions that can be used repeatedly for evaluating ∇f at any particular point. This is exactly what AD achieves directly, i.e., without first generating a mess of symbolic expressions and then simplifying them again. A key role is played by intermediate values, such as the initial partial products $x_1 * x_2 * \cdots * x_i$, which are computed during the evaluation of $f(x)$ itself (see Exercise 3.6). They get their own identifiers and may be saved for subsequent calculations, especially the evaluation of ∇f and higher derivatives. In short, the result of applying AD to a function is a program for evaluating its derivative, but not a formula.

Like symbolic differentiation, AD operates by systematic application of the chain rule, which should be familiar from elementary differential calculus. However, in the case of AD, the chain rule is applied not to symbolic expressions but to actual numerical values. As demonstrated by Monagan and Neuenschwander one can of course implement AD within a computer algebra system [MN93].

In summary, we may view symbolic differentiation as a fairly close relative with whom we ought to interact more than we usually do. Differencing, on the other hand, is a completely different technique, which is often extremely useful as a consistency check [Wol82] and certainly must be considered a baseline in

terms of accuracy and efficiency. Probably infuriated by their undiminished popularity, AD enthusiasts sometimes abuse difference quotients as easy targets and punching bags. To save us from this unprofessional transgression, we note right away:

RULE 1
DIFFERENCE QUOTIENTS MAY SOMETIMES BE USEFUL TOO.

Computer algebra packages have very elaborate facilities for simplifying expressions on the basis of algebraic identities. In contrast, current AD packages assume that the given program calculates the underlying function efficiently, and under this assumption they produce a transformed program that calculates the derivative(s) with the same level of efficiency. Although large parts of this translation process may in practice be automated (or at least assisted by software tools), the best results will be obtained when AD takes advantage of the user's insight into the structure underlying the program, rather than by the blind application of AD to existing code.

Insight gained from AD can also provide valuable performance improvements in handwritten derivative code (particularly in the case of the reverse accumulation AD technique that we shall meet shortly) and in symbolic manipulation packages. In this book, therefore, we shall view AD not primarily as a set of recipes for writing software but rather as a conceptual framework for considering a number of issues in program and algorithm design.

A Baby Example

Let us look at some of the basic ideas of AD. In this part of the introduction we shall give a plain vanilla account that sweeps all the difficult (and interesting) issues under the carpet.

Suppose we have a program that calculates some floating point outputs y from some floating point inputs x. In doing so, the program will calculate a number of intermediate floating point values. The program control flow may jump around as a result of if-statements, do-loops, procedure calls and returns, and maybe the occasional unreconstructed go-to. Some of these floating point values will be stored in various memory locations corresponding to program variables, possibly overwriting a previous value of the same program variable or reusing a storage location previously occupied by a different program variable that is now out of scope. Other values will be held for a few operations in a temporary register before being overwritten or discarded. Although we will be concerned with many aspects of these computer-related issues later, for the moment we wish to remain more abstract.

Our first abstraction is that of an *evaluation trace.* An evaluation trace is basically a record of a particular run of a given program, with particular specified values for the input variables, showing the sequence of floating point values calculated by a (slightly idealized) processor and the operations that

computed them. Here is our baby example: Suppose $y = f(x_1, x_2)$ has the formula

$$y = [\sin(x_1/x_2) + x_1/x_2 - \exp(x_2)] * [x_1/x_2 - \exp(x_2)]$$

and we wish to compute the value of y corresponding to $x_1 = 1.5, x_2 = 0.5$. Then a compiled computer program may execute the sequence of operations listed in Table 1.1.

Table 1.2: An Evaluation Trace of Our Baby Example

v_{-1}	=	x_1	=	1.5000		
v_0	=	x_2	=	0.5000		
v_1	=	v_{-1}/v_0	=	1.5000/0.5000	=	3.0000
v_2	=	$\sin(v_1)$	=	$\sin(3.0000)$	=	0.1411
v_3	=	$\exp(v_0)$	=	$\exp(0.5000)$	=	1.6487
v_4	=	$v_1 - v_3$	=	$3.0000 - 1.6487$	=	1.3513
v_5	=	$v_2 + v_4$	=	$0.1411 + 1.3513$	=	1.4924
v_6	=	$v_5 * v_4$	=	$1.4924 * 1.3513$	=	2.0167
y	=	v_6	=	2.0167		

The evaluation trace, shown in Table 1.2, contains a sequence of mathematical variable definitions. Unlike program variables, mathematical variables can normally not be assigned a value more than once, so the list actually defines a sequence of mathematical functions as well as a sequence of numerical values. We stress that this is just a baby example to illustrate what we mean by applying the chain rule to numerical values. While we are doing this, we wish you to imagine a much bigger example with hundreds of variables and loops repeated thousands of times. The first variables in the list are the input variables; then the intermediate variables (denoted by v_i with $i > 0$) follow, and finally, so does the output variable (just one in this example). Each variable is calculated from previously defined variables by applying one of a number of simple operations: plus, minus, times, sin, exp, and so forth. We can imagine obtaining a low-level evaluation trace by putting a tiny spy camera into the computer's CPU and recording everything that goes past. The evaluation trace does not correspond directly to the expression for f as it was originally written, but to a more efficient evaluation with repeated subexpressions evaluated once and then reused.

It is common for programmers to rearrange mathematical equations so as to improve performance or numerical stability, and this is something that we do not wish to abstract away. The evaluation trace therefore deliberately captures the precise sequence of operations and arguments actually used by a particular implementation of an algorithm or function. Reused subexpressions will be algorithmically exploited by the derivative program, as we shall see.

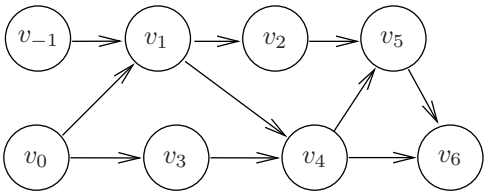

Figure 1.1: Computational Graph of Table 1.2

However, numerical algorithms are increasingly being programmed for parallel execution, and the evaluation trace is (apparently) a serial model. Although a scheduler would spot that any left-hand side can be evaluated as soon as all the corresponding right-hand sides are available, and the evaluation trace could be annotated with an allocation of operations to processors, it is usually more convenient to use a slightly different representation of the evaluation trace called the "computational graph." Figure 1.1 displays the computational graph for our example.

We shall formalize the concepts of evaluation procedures and computational graphs in Chapter 2. The evaluation trace is a linearization of the computational graph. Our baby example lacks many of the features of a real program. Any real modeling function will also contain nested loops, lots of procedure calls to solve systems of equations, and so on. Consequently, the evaluation trace recording the execution of a self-respecting program may be billions of lines long, even if the text of the program itself is quite short. Then there is no prospect of the entire evaluation trace actually being written down (or even fitting into a computer memory) at all.

Forward Mode by Example

Suppose we want to differentiate the output variable y with respect to x_1. (We probably want the derivative with respect to x_2 as well, but let us keep things simple for the moment.) In effect, we are regarding x_1 as the only independent variable and y as a dependent variable.

One way of getting what we want is to work out the numerical value of the derivative of every one of the variables on the list with respect to the independent variable x_1. So, we associate with each variable v_i another variable, $\dot{v}_i = \partial v_i/\partial x_1$. Applying the chain rule mechanically to each line in the evaluation trace, in order, tells us how to assign the correct numerical value to each $\dot{v}_i$. Clearly, $\dot{v}_{-1} = 1.0, \dot{v}_0 = 0.0$ and (for example) since $v_1 = v_{-1}/v_0$, we must have

$$\begin{aligned}\dot{v}_1 &= (\partial v_1/\partial v_{-1})\dot{v}_{-1} + (\partial v_1/\partial v_0)\dot{v}_0 = (\dot{v}_{-1} - v_1 * \dot{v}_0)/v_0 \\ &= (1.0000 - 3.0000 * 0.0000)/0.5000 = 2.0000\,.\end{aligned}$$

Similarly, $v_2 = \sin(v_1)$, so

$$\dot{v}_2 = (\partial v_2/\partial v_1)\dot{v}_1 = \cos(v_1) * \dot{v}_1 = -0.9900 * 2.0000 = -1.9800\,.$$

Augmenting the evaluation trace for our example gives the derived trace in Table 1.3. Imagining a longer calculation with lots of outputs, at the end we would still have that $\dot{y}_i = \partial y_i/\partial x_1$ for each y_i. The total floating point operation

Table 1.3: Forward-Derived Evaluation Trace of Baby Example

$v_{-1} = x_1$	$= 1.5000$		
$\dot{v}_{-1} = \dot{x}_1$	$= 1.0000$		
$v_0 = x_2$	$= 0.5000$		
$\dot{v}_0 = \dot{x}_2$	$= 0.0000$		
$v_1 = v_{-1}/v_0$	$= 1.5000/0.5000$	$=$	3.0000
$\dot{v}_1 = (\dot{v}_{-1} - v_1 * \dot{v}_0)/v_0$	$= 1.0000/0.5000$	$=$	2.0000
$v_2 = \sin(v_1)$	$= \sin(3.0000)$	$=$	0.1411
$\dot{v}_2 = \cos(v_1) * \dot{v}_1$	$= -0.9900 * 2.0000$	$=$	-1.9800
$v_3 = \exp(v_0)$	$= \exp(0.5000)$	$=$	1.6487
$\dot{v}_3 = v_3 * \dot{v}_0$	$= 1.6487 * 0.0000$	$=$	0.0000
$v_4 = v_1 - v_3$	$= 3.0000 - 1.6487$	$=$	1.3513
$\dot{v}_4 = \dot{v}_1 - \dot{v}_3$	$= 2.0000 - 0.0000$	$=$	2.0000
$v_5 = v_2 + v_4$	$= 0.1411 + 1.3513$	$=$	1.4924
$\dot{v}_5 = \dot{v}_2 + \dot{v}_4$	$= -1.9800 + 2.0000$	$=$	0.0200
$v_6 = v_5 * v_4$	$= 1.4924 * 1.3513$	$=$	2.0167
$\dot{v}_6 = \dot{v}_5 * v_4 + v_5 * \dot{v}_4$	$= 0.0200 * 1.3513 + 1.4924 * 2.0000$	$=$	3.0118
$y = v_6$	$= 2.0100$		
$\dot{y} = \dot{v}_6$	$= 3.0110$		

count of the added lines to evaluate $\partial y/\partial x_1$ is a small multiple of that for the underlying code to evaluate y.

This is the basic forward mode of AD, which we shall examine in sections 3.1 and 4.5. It is called "forward" because the derivatives values $\dot{v}_i$ are carried along simultaneously with the values v_i themselves. The problem that remains to be addressed is how to transform a program with a particular evaluation trace into an augmented program whose evaluation trace also contains exactly the same extra variables and additional lines as in the derived evaluation trace. This transformation can be done by a preprocessor or by systematic use of operator overloading. Chapter 6 examines this task in some detail.

We can use exactly the same code to evaluate $\partial y/\partial x_2$ as well: the only change is to set $\dot{x}_1 = 0.0, \dot{x}_2 = 1.0$ at the beginning, instead of the other way round, obtaining $\dot{y} = -13.7239$. However, it may be more efficient to redefine the $\dot{v}_i$ to be vectors (rather than scalars) and to evaluate several partial derivatives at once. This approach is particularly efficient when, because of the structure of the problem, the vectors of partial derivatives are sparse. We shall examine this vector-mode approach in Chapters 7 and 8.

Reverse Mode by Example

An alternative to the forward approach is the *reverse* or *adjoint* mode, which we shall study in detail in sections 3.2 and 4.6 as well as in Chapter 5. Rather than

choosing an input variable and calculating the sensitivity of every intermediate variable with respect to that input, we choose instead an output variable and calculate the sensitivity of that output with respect to each of the intermediate variables. As it turns out adjoint sensitivities must naturally be computed backward, i.e., starting from the output variables. We use the term "reverse mode" for this technique because the label "backward differentiation" is well established for certain methods to integrate stiff ODEs [HNW96].

For our example there is only one output variable, namely, y. Hence we associate with each variable v_i another variable $\bar{v}_i = \partial y/\partial v_i$, called the *adjoint variable*. (Strictly speaking, this is an abuse of derivative notation. What we mean is that $\bar{v}_i = \partial y/\partial \delta_i$, where δ_i is a new independent variable added to the right-hand side of the equation defining v_i.) Adding a small numerical value δ_i to v_i will change the calculated numerical value of y by $\bar{v}_i\,\delta_i$ to first order in δ_i.

Once again, the evaluation procedures can be mechanically transformed to provide a procedure for the computation of the adjoint derivatives $\bar{v}_i = \partial y/\partial v_i$. This time, in order to apply the chain rule, we must work through the original evaluation trace backwards. By definition $\bar{y} = 1.0$, and (for example) since the only ways in which v_1 can affect y are via the definitions $v_4 = v_1 - v_3$ and $v_2 = \sin(v_1)$ in which v_1 appears on the right-hand side, we must have

$$\bar{v}_1 = \bar{v}_4 + \bar{v}_2 * \cos(v_1) = 2.8437 + 1.3513 * (-0.9900) = 1.5059 \; .$$

If you are troubled by this handwaving derivation, simply accept the results for the time being or consult section 3.2 directly. Although we could keep all the contributions in a single assignment to $\bar{v}_1$, it is convenient to split the assignment into two parts, one involving $\bar{v}_4$ and one involving $\bar{v}_2$:

$$\begin{aligned} \bar{v}_1 &= \bar{v}_4 &&= 2.8437 \; , \\ \bar{v}_1 &= \bar{v}_1 + \bar{v}_2 * \cos(v_1) &&= 2.8437 + 1.3513 * (-0.9900) = 1.5059 \; . \end{aligned}$$

This splitting enables us to group each part of the expression for $\bar{v}_1$ with the corresponding line involving v_1 in the original evaluation trace: $v_4 = v_1 - v_3$ and $v_2 = \sin(v_1)$, respectively. The split does mean (in contrast to what has been the case so far) that some of the adjoint variables are incremented as well as being initially assigned.

The complete set of additions to the evaluation trace (corresponding to all the lines of the original) is given in Table 1.4. Once again, the augmented evaluation trace is generated by a program for calculating derivatives. This time the augmented trace is called the reverse trace. Note that the adjoint statements are lined up vertically underneath the original statements that spawned them.

As with the forward propagation method, the total floating point operation count of the added lines is a small multiple of that for the underlying code to evaluate y. Note that the value 3.0118 obtained for $\bar{x}_1 = dy/dx_1$ agrees to four digits with the value obtained for $\partial y/\partial x_1 = \dot{y}$ with $\dot{x}_1 = 1$ in the forward mode. However, this time we have obtained accurate values (to within roundoff error) for both $\bar{x}_1 = \partial y/\partial x_1$ and $\bar{x}_2 = \partial y/\partial x_2$ together. This situation remains true when we imagine a larger example with many more independent variables:

Table 1.4: Reverse-Derived Trace of Baby Example

$$
\begin{aligned}
v_{-1} &= x_1 = 1.5000 \\
v_0 &= x_2 = 0.5000 \\
v_1 &= v_{-1}/v_0 = 1.5000/0.5000 = 3.0000 \\
v_2 &= \sin(v_1) = \sin(3.0000) = 0.1411 \\
v_3 &= \exp(v_0) = \exp(0.5000) = 1.6487 \\
v_4 &= v_1 - v_3 = 3.0000 - 1.6487 = 1.3513 \\
v_5 &= v_2 + v_4 = 0.1411 + 1.3513 = 1.4924 \\
v_6 &= v_5 * v_4 = 1.4924 * 1.3513 = 2.0167 \\
y &= v_6 = 2.0167 \\
\bar{v}_6 &= \bar{y} = 1.0000 \\
\bar{v}_5 &= \bar{v}_6 * v_4 = 1.0000 * 1.3513 = 1.3513 \\
\bar{v}_4 &= \bar{v}_6 * v_5 = 1.0000 * 1.4924 = 1.4924 \\
\bar{v}_4 &= \bar{v}_4 + \bar{v}_5 = 1.4924 + 1.3513 = 2.8437 \\
\bar{v}_2 &= \bar{v}_5 = 1.3513 \\
\bar{v}_3 &= -\bar{v}_4 = -2.8437 \\
\bar{v}_1 &= \bar{v}_4 = 2.8437 \\
\bar{v}_0 &= \bar{v}_3 * v_3 = -2.8437 * 1.6487 = -4.6884 \\
\bar{v}_1 &= \bar{v}_1 + \bar{v}_2 * \cos(v_1) = 2.8437 + 1.3513 * (-0.9900) = 1.5059 \\
\bar{v}_0 &= \bar{v}_0 - \bar{v}_1 * v_1/v_0 = -4.6884 - 1.5059 * 3.000/0.5000 = -13.7239 \\
\bar{v}_{-1} &= \bar{v}_1/v_0 = 1.5059/0.5000 = 3.0118 \\
\bar{x}_2 &= \bar{v}_0 = -13.7239 \\
\bar{x}_1 &= \bar{v}_{-1} = 3.0118
\end{aligned}
$$

Even if our output y depended upon a million inputs x_i, we could still use this reverse or adjoint method of AD to obtain all 1 million components of $\nabla_x y$ with an additional floating point operation count of between one and four times that required for a single evaluation of y. For vector-valued functions one obtains the product of its Jacobian transpose with a vector $\bar{y} = (\bar{y}_i)$, provided the adjoints of the y_i are initialized to $\bar{y}_i$.

Attractions of the Reverse Mode

Although this "something-for-nothing" result seems at first too good to be true, a moment's reflection reassures us that it is a simple consequence of the fact that there is only one dependent variable. If there were several outputs, we would need to rerun the adjoint code (or redefine the $\bar{v}_i$ to be vectors) in order to obtain the complete Jacobian. However, in many applications the number of output variables of interest is several orders of magnitude smaller than the number of inputs. Indeed, problems that contain a single objective function depending upon thousands or millions of inputs are quite common, and in

such cases the reverse method does indeed provide a complete, accurate set of first-order derivatives (sensitivities) for a cost of between one and four function evaluations. So much for gradients being computationally expensive!

Of course, we still have to find a way to transform the original program (which produced the initial evaluation trace) into a program that will produce the required additional lines and allocate space for the adjoint variables. Since we need to process the original evaluation trace "backward," we need (in some sense) to be able to transform the original program so that it runs backward. This program reversal is in fact the main challenge for efficient gradient calculation.

We conclude this subsection with a quick example of the way in which AD may produce useful by-products. The adjoint variables $\bar{v}_i$ allow us to make an estimate of the effect of rounding errors on the calculated value of evaluation-trace-dependent variable y. Since the rounding error δ_i introduced by the operation that computes the value of v_i is generally bounded by $\varepsilon|v_i|$ for some machine-dependent constant ε, we have (to first order) that the calculated value of y differs from the true value by at most

$$\varepsilon \sum_i |\bar{v}_i \, v_i| \; .$$

Techniques of this kind can be combined with techniques from interval analysis and generalized to produce self-validating algorithms [Chr92].

Further Developments and Book Organization

The book is organized into this introduction and three separate parts comprising Chapters 2–6, Chapters 7–11, and Chapters 12–15, respectively. Compared to the first edition, which appeared in 2000 we have made an effort to bring the material up-to-date and make especially Part I more readable to the novice. Chapter 2 describes our model of computer-evaluated functions. In the fundamental Chapter 3 we develop the forward and reverse modes for computing dense derivative vectors of first order. A detailed analysis of memory issues and the computational complexity of the forward and reverse mode differentiation is contained in Chapter 4. Apart from some refinements and extensions of the reverse mode, Chapter 5 describes how a combination of reverse and forward differentiation yields second-order adjoint vectors. In the context of optimization calculations these vectors might be thought of as products of the Hessian of the Lagrangian function with feasible directions $\dot{x}$. The weight vector $\bar{y}$ plays the role of the Lagrangian multipliers. The resulting second-order derivative vectors can again be obtained at a cost that is a small multiple of the underlying vector function. It might be used, for example, in truncated Newton methods for (unconstrained) optimization. The sixth chapter concludes Part I, which could be used as a stand-alone introduction to AD. Here we discuss how AD software can be implemented and used, giving pointers to a few existing packages and also discussing the handling of parallelism.

Part II contains some newer material, especially concerning the NP completeness issue and has also been substantially reorganized and extended. The point of departure for Part II is the observation that, in general, Jacobians and Hessians cannot always be obtained significantly more cheaply than by computing a family of matrix-vector or vector-matrix products, with the vector ranging over a complete basis of the domain or range space. Consequently, we will find that (in contrast to the situation for individual adjoint vectors) the cost ratio for square Jacobians or Hessians relative to that for evaluating the underlying function grows in general like the number of variables. Besides demonstrating this worst-case bound in its final Chapter 11, Part II examines the more typical situation, where Jacobians or Hessians can be obtained more cheaply, because they are sparse or otherwise structured. Apart from dynamically sparse approaches (Chapter 7) we discuss matrix compression in considerable detail (Chapter 8). Chapter 9 opens the door to generalizations of the forward and reverse modes, whose optimal application is provably NP hard by reduction to ensemble computation. The actual accumulation of Jacobians and Hessians is discussed in Chapter 10. Chapter 11 considers reformulations of a given problem based on partial separability and provides some general advice to users about problem preparation.

Part III, called "Advances and Reversals," contains more advanced material that will be of interest mostly to AD researchers and users with special needs (very large problems, third and higher derivatives, generalized gradients), in addition to the mathematically curious. Note that in AD, "reversals" are not a bad thing—quite the opposite. While the basic reverse mode for adjoint calculations has an amazingly low operations count, its memory consumption can be excessive. In Chapter 12 we consider various ways of trading space and time in running the reverse mode. Chapter 13 details methods for evaluating higher derivatives, and Chapter 14 concerns the differentiation of codes with nondifferentiabilities (i.e., strictly speaking, most of them). The differentiation of implicit functions and iterative processes is discussed in Chapter 15.

Most chapters contain a final section with some exercises. None of these are very difficult once the curious terminology for this mixture of mathematics and computing has been understood.

There are some slight changes in the notation of this second edition compared to the first. The most important one is that the adjoint vectors overlined with a bar are no longer considered as row vectors but as column vectors and thus frequently appear transposed by the appropriate superscript. The other change, which might cause confusion is that in the final Chapter 15 on implicit and iterative differentiation the implicit adjoint vector is now defined with a positive sign in equation (15.8).

Historical Notes

AD has a long history, with many checkpoints. Arguably, the subject began with Newton and Leibniz, whose approach to differential equations appears to envisage applying the derivative calculations to numbers rather than symbolic

formulas. Since then the subject has been rediscovered and reinvented several times, and any account of its history is necessarily partisan.

Something very like the reverse method of AD was being used to design electrical circuits in the 1950s [DR69, HBG71], and Wilkinson's backward error analysis dates to the beginning of the 1960s. Shortly thereafter the availability of compilers led to pressure for an algorithmic transformation of programs to compute derivatives. Early steps in this direction, using forward mode, were taken by Beda et al. [BK$^+$59], Wengert [Wen64], Wanner [Wan69], and Warner [War75] with developments by Kedem [Ked80] and Rall [Ral84]. For more application-oriented presentations see Sargent and Sullivan [SS77], Pfeiffer [Pfe80], and Ponton [Pon82]. Subsequently, the potential of the reverse mode for automation was realized independently by a number of researchers, among them Ostrovskii, Volin, and Borisov [OVB71], Linnainmaa [Lin76], Speelpenning [Spe80], Cacuci et al. [CW$^+$80], Werbos [Wer82], Baur and Strassen [BS83], Kim et al. [KN$^+$84], Iri, Tsuchiya, and Hoshi [ITH88], and Griewank [Gri89]. There is also a close connection to back propagation in neural networks [Wer88]. Checkpointing and preaccumulation were discussed by Volin and Ostrovskii as early as 1985 [VO85]. Computer technology and software have continued to develop, and the ability to build algorithmic tools to perform differentiation is now within the reach of everyone in the field of scientific computing.