CHAPTER 8

Calculating Derivatives

Most algorithms for nonlinear optimization and nonlinear equations require knowledge of derivatives. Sometimes the derivatives are easy to calculate by hand, and it is reasonable to expect the user to provide code to compute them. In other cases, the functions are too complicated, so we look for ways to calculate or approximate the derivatives automatically. A number of interesting approaches are available, of which the most important are probably the following.

Finite Differencing. This technique has its roots in Taylor's theorem (see Chapter 2). By observing the change in function values in response to small perturbations of the unknowns near a given point x, we can estimate the response to *infintesimal* perturbations, that is, the derivatives. For instance, the partial derivative of a smooth function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ with respect to the ith variable x_i can be approximated by the central-difference formula

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + \epsilon e_i) - f(x - \epsilon e_i)}{2\epsilon},$$

where ϵ is a small positive scalar and e_i is the ith unit vector, that is, the vector whose elements are all 0 except for a 1 in the ith position.

Automatic Differentiation. This technique takes the view that the computer code for evaluating the function can be broken down into a composition of elementary arithmetic operations, to which the chain rule (one of the basic rules of calculus) can be applied. Some software tools for automatic differentiation (such as ADIFOR [25]) produce new code that calculates both function and derivative values. Other tools (such as ADOL-C [154]) keep a record of the elementary computations that take place while the function evaluation code for a given point x is executing on the computer. This information is processed to produce the derivatives at the same point x.

Symbolic Differentiation. In this technique, the algebraic specification for the function f is manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient. Commonly used symbolic manipulation tools can be found in the packages Mathematica [311], Maple [304], and Macsyma [197].

In this chapter we discuss the first two approaches: finite differencing and automatic differentiation.

The usefulness of derivatives is not restricted to *algorithms* for optimization. Modelers in areas such as design optimization and economics are often interested in performing post-optimal *sensitivity analysis*, in which they determine the sensitivity of the optimum to small perturbations in the parameter or constraint values. Derivatives are also important in other areas such as nonlinear differential equations and simulation.

8.1 FINITE-DIFFERENCE DERIVATIVE APPROXIMATIONS

Finite differencing is an approach to the calculation of approximate derivatives whose motivation (like that of so many algorithms in optimization) comes from Taylor's theorem. Many software packages perform automatic calculation of finite differences whenever the user is unable or unwilling to supply code to calculate exact derivatives. Although they yield only approximate values for the derivatives, the results are adequate in many situations.

By definition, derivatives are a measure of the sensitivity of the function to infinitesimal changes in the values of the variables. Our approach in this section is to make small, *finite* perturbations in the values of x and examine the resulting *differences* in the function values.

By taking ratios of the function difference to variable difference, we obtain approximations to the derivatives.

APPROXIMATING THE GRADIENT

An approximation to the gradient vector $\nabla f(x)$ can be obtained by evaluating the function f at $(n+1)$ points and performing some elementary arithmetic. We describe this technique, along with a more accurate variant that requires additional function evaluations.

A popular formula for approximating the partial derivative $\partial f/\partial x_i$ at a given point x is the *forward-difference*, or *one-sided-difference*, approximation, defined as

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+\epsilon e_i) - f(x)}{\epsilon}. \tag{8.1}$$

The gradient can be built up by simply applying this formula for $i = 1, 2, \ldots, n$. This process requires evaluation of f at the point x as well as the n perturbed points $x + \epsilon e_i$, $i = 1, 2, \ldots, n$: a total of $(n+1)$ points.

The basis for the formula (8.1) is Taylor's theorem, Theorem 2.1 in Chapter 2. When f is twice continuously differentiable, we have

$$f(x+p) = f(x) + \nabla f(x)^T p + \tfrac{1}{2} p^T \nabla^2 f(x+tp) p, \qquad \text{some } t \in (0,1) \tag{8.2}$$

(see (2.6)). If we choose L to be a bound on the size of $\|\nabla^2 f(\cdot)\|$ in the region of interest, it follows directly from this formula that the last term in this expression is bounded by $(L/2)\|p\|^2$, so that

$$\left\| f(x+p) - f(x) - \nabla f(x)^T p \right\| \le (L/2)\|p\|^2. \tag{8.3}$$

We now choose the vector p to be ϵe_i, so that it represents a small change in the value of a single component of x (the ith component). For this p, we have that $\nabla f(x)^T p = \nabla f(x)^T e_i = \partial f/\partial x_i$, so by rearranging (8.3), we conclude that

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x+\epsilon e_i) - f(x)}{\epsilon} + \delta_\epsilon, \qquad \text{where } |\delta_\epsilon| \le (L/2)\epsilon. \tag{8.4}$$

We derive the forward-difference formula (8.1) by simply ignoring the error term δ_ϵ in this expression, which becomes smaller and smaller as ϵ approaches zero.

An important issue in implementing the formula (8.1) is the choice of the parameter ϵ. The error expression (8.4) suggests that we should choose ϵ as small as possible. Unfortunately, this expression ignores the roundoff errors that are introduced when the function f is evaluated on a real computer, in floating-point arithmetic. From our discussion in the Appendix (see (A.30) and (A.31)), we know that the quantity $\mathbf{u}$ known as *unit roundoff* is crucial: It is a bound on the relative error that is introduced whenever an arithmetic operation is performed on two floating-point numbers. ($\mathbf{u}$ is about 1.1×10^{-16} in double-precision IEEE floating-point arithmetic.) The effect of these errors on the final computed value of f depends on the way in which f is computed. It could come from an arithmetic formula, or from a differential equation solver, with or without refinement.

As a rough estimate, let us assume simply that the relative error in the computed f is bounded by $\mathbf{u}$, so that the computed values of $f(x)$ and $f(x+\epsilon e_i)$ are related to the exact values in the following way:

$$\begin{aligned} |\text{comp}(f(x)) - f(x)| &\le \mathbf{u}L_f, \\ |\text{comp}(f(x+\epsilon e_i)) - f(x+\epsilon e_i)| &\le \mathbf{u}L_f, \end{aligned}$$

where comp$(\cdot)$ denotes the computed value, and L_f is a bound on the value of $|f(\cdot)|$ in the region of interest. If we use these computed values of f in place of the exact values in (8.4) and (8.1), we obtain an error that is bounded by

$$(L/2)\epsilon + 2\mathbf{u}L_f/\epsilon. \tag{8.5}$$

Naturally, we would like to choose ϵ to make this error as small as possible; it is easy to see that the minimizing value is

$$\epsilon^2 = \frac{4L_f\mathbf{u}}{L}.$$

If we assume that the problem is well scaled, then the ratio L_f/L (the ratio of function values to second derivative values) does not exceed a modest size. We can conclude that the following choice of ϵ is fairly close to optimal:

$$\epsilon = \sqrt{\mathbf{u}}. \tag{8.6}$$

(In fact, this value is used in many of the optimization software packages that use finite differencing as an option for estimating derivatives.) For this value of ϵ, we have from (8.5) that the total error in the forward-difference approximation is fairly close to $\sqrt{\mathbf{u}}$.

A more accurate approximation to the derivative can be obtained by using the *central-difference* formula, defined as

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+\epsilon e_i) - f(x-\epsilon e_i)}{2\epsilon}. \tag{8.7}$$

As we show below, this approximation is more accurate than the forward-difference approximation (8.1). It is also about twice as expensive, since we need to evaluate f at the points x and $x \pm \epsilon e_i$, $i = 1, 2, \dots, n$: a total of $2n+1$ points.

The basis for the central difference approximation is again Taylor's theorem. When the second derivatives of f exist and are Lipschitz continuous, we have from (8.2) that

$$\begin{aligned} f(x+p) &= f(x) + \nabla f(x)^T p + \tfrac{1}{2} p^T \nabla^2 f(x+tp)p \quad \text{for some } t \in (0,1) \\ &= f(x) + \nabla f(x)^T p + \tfrac{1}{2} p^T \nabla^2 f(x)p + O\left(\|p\|^3\right). \end{aligned} \tag{8.8}$$

By setting $p = \epsilon e_i$ and $p = -\epsilon e_i$, respectively, we obtain

$$\begin{aligned} f(x+\epsilon e_i) &= f(x) + \epsilon \frac{\partial f}{\partial x_i} + \frac{1}{2}\epsilon^2 \frac{\partial^2 f}{\partial x_i^2} + O\left(\epsilon^3\right), \\ f(x-\epsilon e_i) &= f(x) - \epsilon \frac{\partial f}{\partial x_i} + \frac{1}{2}\epsilon^2 \frac{\partial^2 f}{\partial x_i^2} + O\left(\epsilon^3\right). \end{aligned}$$

(Note that the final error terms in these two expressions are generally not the same, but they are both bounded by some multiple of ϵ^3.) By subtracting the second equation from the first and dividing by 2ϵ, we obtain the expression

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x+\epsilon e_i) - f(x-\epsilon e_i)}{2\epsilon} + O\left(\epsilon^2\right).$$

We see from this expression that the error is $O\left(\epsilon^2\right)$, as compared to the $O(\epsilon)$ error in the forward-difference formula (8.1). However, when we take evaluation error in f into account, the accuracy that can be achieved in practice is less impressive; the same assumptions that were used to derive (8.6) lead to an optimal choice of ϵ of about $\mathbf{u}^{1/3}$ and an error of about $\mathbf{u}^{2/3}$. In some situations, the extra few digits of accuracy may improve the performance of the algorithm enough to make the extra expense worthwhile.

APPROXIMATING A SPARSE JACOBIAN

Consider now the case of a vector function $r : \mathbb{R}^n \to \mathbb{R}^m$, such as the residual vector that we consider in Chapter 10 or the system of nonlinear equations from Chapter 11. The matrix $J(x)$ of first derivatives for this function is defined as follows:

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right]_{\substack{j=1,2,\ldots,m \\ i=1,2,\ldots,n}} = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_m(x)^T \end{bmatrix}, \tag{8.9}$$

where r_j, $j = 1, 2, \ldots, m$ are the components of r. The techniques described in the previous section can be used to evaluate the full Jacobian $J(x)$ one column at a time. When r is twice continuously differentiable, we can use Taylor's theorem to deduce that

$$\|r(x+p) - r(x) - J(x)p\| \le (L/2)\|p\|^2, \tag{8.10}$$

where L is a Lipschitz constant for J in the region of interest. If we require an approximation to the Jacobian–vector product $J(x)p$ for a given vector p (as is the case with inexact Newton methods for nonlinear systems of equations; see Section 11.1), this expression immediately suggests choosing a small nonzero ϵ and setting

$$J(x)p \approx \frac{r(x+\epsilon p) - r(x)}{\epsilon}, \tag{8.11}$$

an approximation that is accurate to $O(\epsilon)$. A two-sided approximation can be derived from the formula (8.7).

If an approximation to the full Jacobian $J(x)$ is required, we can compute it a column at a time, analogously to (8.1), by setting set $p = \epsilon e_i$ in (8.10) to derive the following estimate of the ith column:

$$\frac{\partial r}{\partial x_i}(x) \approx \frac{r(x+\epsilon e_i) - r(x)}{\epsilon}. \tag{8.12}$$

A full Jacobian estimate can be obtained at a cost of $n+1$ evaluations of the function r. When the Jacobian is sparse, however, we can often obtain the estimate at a much lower cost, sometimes just three or four evaluations of r. The key is to estimate a number of different columns of the Jacobian simultaneously, by judicious choices of the perturbation vector p in (8.10).

We illustrate the technique with a simple example. Consider the function $r : \mathbb{R}^n \to \mathbb{R}^n$ defined by

$$r(x) = \begin{bmatrix} 2(x_2^3 - x_1^2) \\ 3(x_2^3 - x_1^2) + 2(x_3^3 - x_2^2) \\ 3(x_3^3 - x_2^2) + 2(x_4^3 - x_3^2) \\ \vdots \\ 3(x_n^3 - x_{n-1}^2) \end{bmatrix}. \tag{8.13}$$

Each component of r depends on just two or three components of x, so that each row of the Jacobian contains only two or three nonzero elements. For the case of $n = 6$, the Jacobian has the following structure:

$$\begin{bmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}, \tag{8.14}$$

where each cross represents a nonzero element, with zeros represented by a blank space.

Staying for the moment with the case $n = 6$, suppose that we wish to compute a finite-difference approximation to the Jacobian. (Of course, it is easy to calculate this particular Jacobian by hand, but there are complicated functions with similar structure for which hand calculation is more difficult.) A perturbation $p = \epsilon e_1$ to the first component of x will affect only the first and second components of r. The remaining components will be unchanged, so that the right-hand-side of formula (8.12) will correctly evaluate to zero in the components 3, 4, 5, 6. It is wasteful, however, to reevaluate these components of r when we know in advance that their values are not affected by the perturbation. Instead, we look for a way to modify the perturbation vector so that it does not have any further effect on components 1 and 2, but *does* produce a change in some of the components 3, 4, 5, 6, which we can then use as the basis of a finite-difference estimate for some *other* column of the Jacobian. It is not hard to see that the additional perturbation ϵe_4 has the desired property: It alters the 3rd, 4th, and 5th elements of r, but leaves the 1st and 2nd elements unchanged. The changes in r as a result of the perturbations ϵe_1 and ϵe_4 do not interfere with each other.

To express this discussion in mathematical terms, we set

$$p = \epsilon(e_1 + e_4),$$

and note that

$$r(x+p)_{1,2} = r(x + \epsilon(e_1 + e_4))_{1,2} = r(x + \epsilon e_1)_{1,2} \tag{8.15}$$

(where the notation $[\cdot]_{1,2}$ denotes the subvector consisting of the first and second elements), while

$$r(x+p)_{3,4,5} = r(x + \epsilon(e_1 + e_4))_{3,4,5} = r(x + \epsilon e_4)_{3,4,5}. \tag{8.16}$$

By substituting (8.15) into (8.10), we obtain

$$r(x+p)_{1,2} = r(x)_{1,2} + \epsilon[J(x)e_1]_{1,2} + O(\epsilon^2).$$

By rearranging this expression, we obtain the following difference formula for estimating the (1, 1) and (2, 1) elements of the Jacobian matrix:

$$\begin{bmatrix} \dfrac{\partial r_1}{\partial x_1}(x) \\ \dfrac{\partial r_2}{\partial x_1}(x) \end{bmatrix} = [J(x)e_1]_{1,2} \approx \frac{r(x+p)_{1,2} - r(x)_{1,2}}{\epsilon}. \tag{8.17}$$

A similar argument shows that the nonzero elements of the fourth column of the Jacobian can be estimated by substituting (8.16) into (8.10); we obtain

$$\begin{bmatrix} \dfrac{\partial r_4}{\partial x_3}(x) \\ \dfrac{\partial r_4}{\partial x_4}(x) \\ \dfrac{\partial r_4}{\partial x_5}(x) \end{bmatrix} = [J(x)e_4]_{3,4,5} \approx \frac{r(x+p)_{3,4,5} - r(x)_{3,4,5}}{\epsilon}. \tag{8.18}$$

To summarize: We have been able to estimate *two* columns of the Jacobian $J(x)$ by evaluating the function r at the single extra point $x + \epsilon(e_1 + e_4)$.

We can approximate the remainder of $J(x)$ in an economical manner as well. Columns 2 and 5 can be approximated by choosing $p = \epsilon(e_2 + e_5)$, while we can use $p = \epsilon(e_3 + e_6)$ to approximate columns 3 and 6. In total, we need 3 evaluations of the function r (after the initial evaluation at x) to estimate the entire Jacobian matrix.

In fact, for *any* choice of n in (8.13) (no matter how large), three extra evaluations of r are sufficient to approximate the entire Jacobian. The corresponding choices of perturbation vectors p are

$$\begin{aligned} p &= \epsilon(e_1 + e_4 + e_7 + e_{10} + \cdots), \\ p &= \epsilon(e_2 + e_5 + e_8 + e_{11} + \cdots), \\ p &= \epsilon(e_3 + e_6 + e_9 + e_{12} + \cdots). \end{aligned}$$

In the first of these vectors, the nonzero components are chosen so that no two of the columns 1, 4, 7, ... have a nonzero element in the same row. The same property holds for the other two vectors and, in fact, points the way to the criterion that we can apply to general problems to decide on a valid set of perturbation vectors.

Algorithms for choosing the perturbation vectors can be expressed conveniently in the language of graphs and graph coloring. For any function $r : \mathbb{R}^n \to \mathbb{R}^m$, we can construct a *column incidence graph* G with n nodes by drawing an arc between nodes i and k if there is some component of r that depends on both x_i and x_k. In other words, the ith and kth columns of the Jacobian $J(x)$ each have a nonzero element in some row j, for some $j = 1, 2, \ldots, m$ and some value of x. (The intersection graph for the function defined in

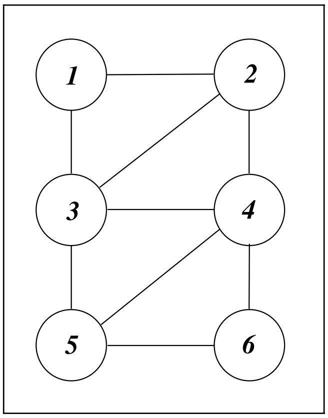

Figure 8.1
Column incidence graph for $r(x)$ defined in (8.13).

(8.13), with $n = 6$, is shown in Figure 8.1.) We now assign each node a "color" according to the following rule: Two nodes can have the same color if there is no arc that connects them. Finally, we choose one perturbation vector corresponding to each color: If nodes $i_1, i_2, \ldots, i_\ell$ have the same color, the corresponding p is $\epsilon(e_{i_1} + e_{i_2} + \cdots + e_{i_\ell})$.

Usually, there are many ways to assign colors to the n nodes in the graph in a way that satisfies the required condition. The simplest way is just to assign each node a different color, but since that scheme produces n perturbation vectors, it is usually not the most efficient approach. It is generally very difficult to find the coloring scheme that uses the fewest possible colors, but there are simple algorithms that do a good job of finding a near-optimal coloring at low cost. Curtis, Powell, and Reid [83] and Coleman and Moré [68] provide descriptions of some methods and performance comparisons. Newsam and Ramsdell [227] show that by considering a more general class of perturbation vectors p, it is possible to evaluate the full Jacobian using no more than n_z evaluations of r (in addition to the evaluation at the point x), where n_z is the maximum number of nonzeros in each row of $J(x)$.

For some functions r with well-studied structures (those that arise from discretizations of differential operators, or those that give rise to banded Jacobians, as in the example above), optimal coloring schemes are known. For the tridiagonal Jacobian of (8.14) and its associated graph in Figure 8.1, the scheme with three colors is optimal.

APPROXIMATING THE HESSIAN

In some situations, the user may be able to provide a routine to calculate the gradient $\nabla f(x)$ but not the Hessian $\nabla^2 f(x)$. We can obtain the Hessian by applying the techniques described above for the vector function r to the gradient ∇f. By using the graph coloring techniques discussed above, sparse Hessians often can be approximated in this manner by using considerably fewer than n perturbation vectors. This approach ignores symmetry of the Hessian, and will usually produce a nonsymmetric approximation. We can recover symmetry by adding the approximation to its transpose and dividing the result by 2. Alternative differencing approaches that take symmetry of $\nabla^2 f(x)$ explicitly into account are discussed below.

Some important algorithms—most notably the Newton–CG methods described in Chapter 7—do not require knowledge of the full Hessian. Instead, each iteration requires us to supply the Hessian–vector product $\nabla^2 f(x)p$, for a given vector p. We can obtain an approximation to this matrix-vector product by appealing once again to Taylor's theorem. When second derivatives of f exist and are Lipschitz continuous near x, we have

$$\nabla f(x + \epsilon p) = \nabla f(x) + \epsilon \nabla^2 f(x)p + O(\epsilon^2), \tag{8.19}$$

so that

$$\nabla^2 f(x)p \approx \frac{\nabla f(x + \epsilon p) - \nabla f(x)}{\epsilon} \tag{8.20}$$

(see also (7.10)). The approximation error is $O(\epsilon)$, and the cost of obtaining the approximation is a single gradient evaluation at the point $x + \epsilon p$. The formula (8.20) corresponds to the forward-difference approximation (8.1). A central-difference formula like (8.7) can be derived by evaluating $\nabla f(x - \epsilon p)$ as well.

For the case in which even gradients are not available, we can use Taylor's theorem once again to derive formulae for approximating the Hessian that use only function values. The main tool is the formula (8.8): By substituting the vectors $p = \epsilon e_i$, $p = \epsilon e_j$, and $p = \epsilon(e_i + e_j)$ into this formula and combining the results appropriately, we obtain

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{f(x + \epsilon e_i + \epsilon e_j) - f(x + \epsilon e_i) - f(x + \epsilon e_j) + f(x)}{\epsilon^2} + O(\epsilon). \tag{8.21}$$

If we wished to approximate every element of the Hessian with this formula, then we would need to evaluate f at $x + \epsilon(e_i + e_j)$ for all possible i and j (a total of $n(n+1)/2$ points) as well as at the n points $x + \epsilon e_i$, $i = 1, 2, \dots, n$. If the Hessian is sparse, we can, of course, reduce this operation count by skipping the evaluation whenever we know the element $\partial^2 f/\partial x_i \partial x_j$ to be zero.

APPROXIMATING A SPARSE HESSIAN

We noted above that a Hessian approximation can be obtained by applying finite-difference Jacobian estimation techniques to the gradient ∇f, treated as a vector function. We now show how symmetry of the Hessian $\nabla^2 f$ can be used to reduce the number of perturbation vectors p needed to obtain a complete approximation, when the Hessian is sparse. The key observation is that, because of symmetry, any estimate of the element $[\nabla^2 f(x)]_{i,j} = \partial^2 f(x)/\partial x_i \partial x_j$ is also an estimate of its symmetric counterpart $[\nabla^2 f(x)]_{j,i}$.

We illustrate the point with the simple function $f : \mathbb{R}^n \to \mathbb{R}$ defined by

$$f(x) = x_1 \sum_{i=1}^{n} i^2 x_i^2. \tag{8.22}$$

It is easy to show that the Hessian $\nabla^2 f$ has the "arrowhead" structure depicted below, for the case of $n = 6$:

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & & & & \\ \times & & \times & & & \\ \times & & & \times & & \\ \times & & & & \times & \\ \times & & & & & \times \end{bmatrix}. \tag{8.23}$$

If we were to construct the intersection graph for the function ∇f (analogous to Figure 8.1), we would find that every node is connected to every other node, for the simple reason that row 1 has a nonzero in every column. According to the rule for coloring the graph, then, we would have to assign a different color to every node, which implies that we would need to evaluate ∇f at the $n + 1$ points x and $x + \epsilon e_i$ for $i = 1, 2, \ldots, n$.

We can construct a much more efficient scheme by taking the symmetry into account. Suppose we first use the perturbation vector $p = \epsilon e_1$ to estimate the first column of $\nabla^2 f(x)$. Because of symmetry, the same estimates apply to the first *row* of $\nabla^2 f$. From (8.23), we see that all that remains is to find the diagonal elements $\nabla^2 f(x)_{22}, \nabla^2 f(x)_{33}, \ldots, \nabla^2 f(x)_{66}$. The intersection graph for these remaining elements is completely disconnected, so we can assign them all the same color and choose the corresponding perturbation vector to be

$$p = \epsilon(e_2 + e_3 + \cdots + e_6) = \epsilon(0, 1, 1, 1, 1, 1)^T. \tag{8.24}$$

Note that the second component of ∇f is not affected by the perturbations in components 3, 4, 5, 6 of the unknown vector, while the third component of ∇f is not affected by perturbations in components 2, 4, 5, 6 of x, and so on. As in (8.15) and (8.16), we have for each component i that

$$\nabla f(x + p)_i = \nabla f(x + \epsilon(e_2 + e_3 + \cdots + e_6))_i = \nabla f(x + \epsilon e_i)_i.$$

By applying the forward-difference formula (8.1) to each of these individual components, we then obtain

$$\frac{\partial^2 f}{\partial x_i^2}(x) \approx \frac{\nabla f(x + \epsilon e_i)_i - \nabla f(x)_i}{\epsilon} = \frac{\nabla f(x + \epsilon p)_i - \nabla f(x)_i}{\epsilon}, \qquad i = 2, 3, \ldots, 6.$$

By exploiting symmetry, we have been able to estimate the entire Hessian by evaluating ∇f only at x and two other points.

Again, graph-coloring techniques can be used to choose the perturbation vectors p economically. We use the *adjacency graph* in place of the intersection graph described earlier. The adjacency graph has n nodes, with arcs connecting nodes i and k whenever $i \neq k$ and $\partial^2 f(x)/(\partial x_i \partial x_k) \neq 0$ for some x. The requirements on the coloring scheme are a little more complicated than before, however. We require not only that connected nodes have different colors, but also that any path of length 3 through the graph contain at least three colors. In other words, if there exist nodes i_1, i_2, i_3, i_4 in the graph that are connected by arcs (i_1, i_2), (i_2, i_3), and (i_3, i_4), then at least three different colors must be used in coloring these four nodes. See Coleman and Moré [69] for an explanation of this rule and for algorithms to compute valid colorings. The perturbation vectors are constructed as before: Whenever the nodes $i_1, i_2, \ldots, i_\ell$ have the same color, we set the corresponding perturbation vector to be $p = \epsilon(e_{i_1} + e_{i_2} + \cdots + e_{i_\ell})$.

8.2 AUTOMATIC DIFFERENTIATION

Automatic differentiation is the generic name for techniques that use the computational representation of a function to produce analytic values for the derivatives. Some techniques produce code for the derivatives at a general point x by manipulating the function code directly. Other techniques keep a record of the computations made during the evaluation of the function at a specific point x and then review this information to produce a set of derivatives at x.

Automatic differentiation techniques are founded on the observation that any function, no matter how complicated, is evaluated by performing a sequence of simple elementary operations involving just one or two arguments at a time. Two-argument operations include addition, multiplication, division, and the power operation a^b. Examples of single-argument operations include the trigonometric, exponential, and logarithmic functions. Another common ingredient of the various automatic differentiation tools is their use of the *chain rule.* This is the well-known rule from elementary calculus that says that if h is a function of the vector $y \in \mathbb{R}^m$, which is in turn a function of the vector $x \in \mathbb{R}^n$, we can write the derivative of h with respect to x as follows:

$$\nabla_x h(y(x)) = \sum_{i=1}^{m} \frac{\partial h}{\partial y_i} \nabla y_i(x). \tag{8.25}$$

See Appendix A for further details.

There are two basic modes of automatic differentiation: the *forward* and *reverse* modes. The difference between them can be illustrated by a simple example. We work through such an example below, and indicate how the techniques can be extended to general functions, including vector functions.

AN EXAMPLE

Consider the following function of 3 variables:

$$f(x) = (x_1 x_2 \sin x_3 + e^{x_1 x_2})/x_3. \tag{8.26}$$

Figure 8.2 shows how the evaluation of this function can be broken down into its elementary operations and also indicates the partial ordering associated with these operations. For instance, the multiplication $x_1 * x_2$ must take place prior to the exponentiation $e^{x_1 x_2}$, or else we would obtain the incorrect result $(e^{x_1})x_2$. This graph introduces the *intermediate variables* $x_4, x_5, \ldots$ that contain the results of intermediate computations; they are distinguished from the *independent variables* x_1, x_2, x_3 that appear at the left of the graph. We can express the evaluation of f in arithmetic terms as follows:

$$\begin{aligned} x_4 &= x_1 * x_2, \\ x_5 &= \sin x_3, \\ x_6 &= e^{x_4}, \\ x_7 &= x_4 * x_5, \\ x_8 &= x_6 + x_7, \\ x_9 &= x_8/x_3. \end{aligned} \tag{8.27}$$

The final node x_9 in Figure 8.2 contains the function value $f(x)$. In the terminology of graph theory, node i is the *parent* of node j, and node j the *child* of node i, whenever there is a directed arc from i to j. Any node can be evaluated when the values of all its parents are known, so computation flows through the graph from left to right. Flow of

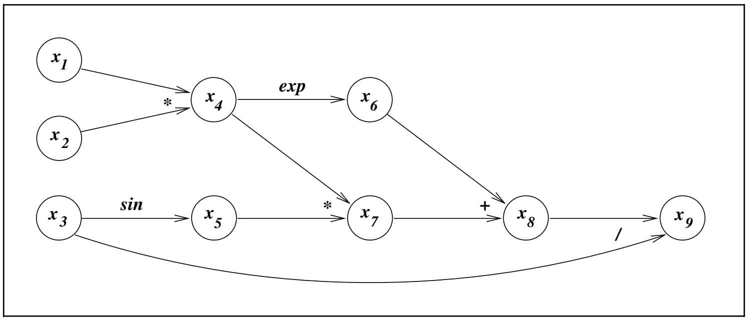

Figure 8.2 Computational graph for $f(x)$ defined in (8.26).

computation in this direction is known as a *forward sweep.* It is important to emphasize that software tools for automatic differentiation do not require the user to break down the code for evaluating the function into its elements, as in (8.27). Identification of intermediate quantities and construction of the computational graph is carried out, explicitly or implicitly, by the software tool itself.

THE FORWARD MODE

In the forward mode of automatic differentiation, we evaluate and carry forward a directional derivative of each intermediate variable x_i in a given direction $p \in \mathbb{R}^n$, simultaneously with the evaluation of x_i itself. For the three-variable example above, we use the following notation for the directional derivative for p associated with each variable:

$$D_p x_i \overset{\text{def}}{=} (\nabla x_i)^T p = \sum_{j=1}^{3} \frac{\partial x_i}{\partial x_j} p_j, \qquad i = 1, 2, \ldots, 9, \tag{8.28}$$

where ∇ indicates the gradient with respect to the three independent variables. Our goal is to evaluate $D_p x_9$, which is the same as the directional derivative $\nabla f(x)^T p$. We note immediately that initial values $D_p x_i$ for the independent variables x_i, $i = 1, 2, 3$, are simply the components p_1, p_2, p_3 of p. The direction p is referred to as the *seed vector.*

As soon as the value of x_i at any node is known, we can find the corresponding value of $D_p x_i$ from the chain rule. For instance, suppose we know the values of x_4, $D_p x_4$, x_5, and $D_p x_5$, and we are about to calculate x_7 in Figure 8.2. We have that $x_7 = x_4 x_5$; that is, x_7 is a function of the two variables x_4 and x_5, which in turn are functions of x_1, x_2, x_3. By applying the rule (8.25), we have that

$$\nabla x_7 = \frac{\partial x_7}{\partial x_4} \nabla x_4 + \frac{\partial x_7}{\partial x_5} \nabla x_5 = x_5 \nabla x_4 + x_4 \nabla x_5.$$

By taking the inner product of both sides of this expression with p and applying the definition (8.28), we obtain

$$D_p x_7 = \frac{\partial x_7}{\partial x_4} D_p x_4 + \frac{\partial x_7}{\partial x_5} D_p x_5 = x_5 D_p x_4 + x_4 D_p x_5. \tag{8.29}$$

The directional derivatives $D_p x_i$ are therefore evaluated side by side with the intermediate results x_i, and at the end of the process we obtain $D_p x_9 = D_p f = \nabla f(x)^T p$.

The principle of the forward mode is straightforward enough, but what of its practical implementation and computational requirements? First, we repeat that the user does *not* need to construct the computational graph, break the computation down into elementary operations as in (8.27), or identify intermediate variables. The automatic differentiation software should perform these tasks implicitly and automatically. Nor is it necessary to store the information x_i and $D_p x_i$ for *every* node of the computation graph at once (which is just as well, since this graph can be very large for complicated functions). Once all the children of any node have been evaluated, its associated values x_i and $D_p x_i$ are not needed further and may be overwritten in storage.

The key to practical implementation is the side-by-side evaluation of x_i and $D_p x_i$. The automatic differentiation software associates a scalar $D_p w$ with any scalar w that appears in the evaluation code. Whenever w is used in an arithmetic computation, the software performs an associated operation (based on the chain rule) with the gradient vector $D_p w$. For instance, if w is combined in a division operation with another value y to produce a new value z, that is,

$$z \leftarrow \frac{w}{y},$$

we use w, z, $D_p w$, and $D_p y$ to evaluate the directional derivative $D_p z$ as follows:

$$D_p z \leftarrow \frac{1}{y} D_p w - \frac{w}{y^2} D_p y. \tag{8.30}$$

To obtain the complete gradient vector, we can carry out this procedure simultaneously for the n seed vectors $p = e_1, e_2, \ldots, e_n$. By the definition (8.28), we see that $p = e_j$ implies that $D_p f = \partial f / \partial x_j$, $j = 1, 2, \ldots, n$. We note from the example (8.30) that the additional cost of evaluating f and ∇f (over the cost of evaluating f alone) may be significant. In this example, the single division operation on w and y needed to calculate z gives rise to approximately $2n$ multiplications and n additions in the computation of the gradient elements $D_{e_j} z$, $j = 1, 2, \ldots, n$. It is difficult to obtain an exact bound on the increase in computation, since the costs of retrieving and storing the data should also be taken into account. The storage requirements may also increase by a factor as large as n, since we now have to store n additional scalars $D_{e_j} x_i$, $j = 1, 2, \ldots, n$, alongside each intermediate variable x_i. It is usually possible to make savings by observing that many of these quantities are zero, particularly in the early stages of the computation (that is, toward the left of the computational graph), so sparse data structures can be used to store the vectors $D_{e_j} x_i$, $j = 1, 2, \ldots, n$ (see [27]).

The forward mode of automatic differentiation can be implemented by means of a precompiler, which transforms function evaluation code into extended code that evaluates the derivative vectors as well. An alternative approach is to use the operator-overloading facilities available in languages such as C++ to transparently extend the data structures and operations in the manner described above.

THE REVERSE MODE

The reverse mode of automatic differentiation does not perform function and gradient evaluations concurrently. Instead, after the evaluation of f is complete, it recovers the partial derivatives of f with respect to each variable x_i—independent and intermediate variables alike—by performing a *reverse sweep* of the computational graph. At the conclusion of this process, the gradient vector ∇f can be assembled from the partial derivatives $\partial f/\partial x_i$ with respect to the independent variables x_i, $i = 1, 2, \ldots, n$.

Instead of the gradient vectors $D_p x_i$ used in the forward mode, the reverse mode associates a scalar variable $\bar{x}_i$ with each node in the graph; information about the partial derivative $\partial f/\partial x_i$ is accumulated in $\bar{x}_i$ during the reverse sweep. The $\bar{x}_i$ are sometimes called the *adjoint variables*, and we initialize their values to zero, with the exception of the rightmost node in the graph (node N, say), for which we set $\bar{x}_N = 1$. This choice makes sense because x_N contains the final function value f, so we have $\partial f/\partial x_N = 1$.

The reverse sweep makes use of the following observation, which is again based on the chain rule (8.25): For any node i, the partial derivative $\partial f/\partial x_i$ can be built up from the partial derivatives $\partial f/\partial x_j$ corresponding to its child nodes j according to the following formula:

$$\frac{\partial f}{\partial x_i} = \sum_{j \text{ a child of } i} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i}. \tag{8.31}$$

For each node i, we add the right-hand-side term in (8.31) to $\bar{x}_i$ as soon as it becomes known; that is, we perform the operation

$$\bar{x}_i \mathrel{+}= \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i}. \tag{8.32}$$

(In this expression and the ones below, we use the arithmetic notation of the programming language C, in which x+=a means $x \leftarrow x + a$.) Once contributions have been received from all the child nodes of i, we have $\bar{x}_i = \partial f/\partial x_i$, so we declare node i to be "finalized." At this point, node i is ready to contribute a term to the summation for each of its parent nodes according to the formula (8.31). The process continues in this fashion until all nodes are finalized. Note that for derivative evaluation, the flow of computation in the graph is from children to parents—the opposite direction to the computation flow for function evaluation.

During the reverse sweep, we work with *numerical values*, not with formulae or computer code involving the variables x_i or the partial derivatives $\partial f/\partial x_i$. During the forward sweep—the evaluation of f—we not only calculate the values of each variable x_i, but we also calculate and store the numerical values of each partial derivative $\partial x_j/\partial x_i$. Each of these partial derivatives is associated with a particular arc of the computational graph. The numerical values of $\partial x_j/\partial x_i$ computed during the forward sweep are then used in the formula (8.32) during the reverse sweep.

We illustrate the reverse mode for the example function (8.26). In Figure 8.3 we fill in the graph of Figure 8.2 for a specific evaluation point $x = (1, 2, \pi/2)^T$, indicating the

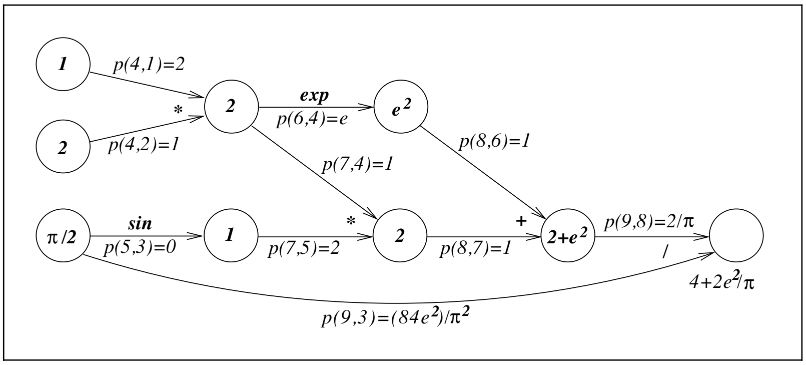

Figure 8.3 Computational graph for $f(x)$ defined in (8.26) showing numerical values of intermediate values and partial derivatives for the point $x = (1, 2, \pi/2)^T$. Notation: $p(j, i) = \partial x_j / \partial x_i$.

numerical values of the intermediate variables $x_4, x_5, \ldots, x_9$ associated with each node and the partial derivatives $\partial x_j / \partial x_i$ associated with each arc.

As mentioned above, we initialize the reverse sweep by setting all the adjoint variables $\bar{x}_i$ to zero, except for the rightmost node, for which we have $\bar{x}_9 = 1$. Since $f(x) = x_9$ and since node 9 has no children, we have $\bar{x}_9 = \partial f / \partial x_9$, and so we can immediately declare node 9 to be finalized.

Node 9 is the child of nodes 3 and 8, so we use formula (8.32) to update the values of $\bar{x}_3$ and $\bar{x}_8$ as follows:

$$\bar{x}_3 {+}{=} \frac{\partial f}{\partial x_9} \frac{\partial x_9}{\partial x_3} = -\frac{2 + e^2}{(\pi/2)^2} = \frac{-8 - 4e^2}{\pi^2}, \tag{8.33a}$$

$$\bar{x}_8 {+}{=} \frac{\partial f}{\partial x_9} \frac{\partial x_9}{\partial x_8} = \frac{1}{\pi/2} = \frac{2}{\pi}. \tag{8.33b}$$

Node 3 is not finalized after this operation; it still awaits a contribution from its other child, node 5. On the other hand, node 9 is the only child of node 8, so we can declare node 8 to be finalized with the value $\frac{\partial f}{\partial x_8} = 2/\pi$. We can now update the values of $\bar{x}_i$ at the two parent nodes of node 8 by applying the formula (8.32) once again; that is,

$$\bar{x}_6 {+}{=} \frac{\partial f}{\partial x_8} \frac{\partial x_8}{\partial x_6} = \frac{2}{\pi};$$

$$\bar{x}_7 {+}{=} \frac{\partial f}{\partial x_8} \frac{\partial x_8}{\partial x_7} = \frac{2}{\pi}.$$

At this point, nodes 6 and 7 are finalized, so we can use them to update nodes 4 and 5. At the end of this process, when all nodes are finalized, nodes 1, 2, and 3 contain

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \end{bmatrix} = \nabla f(x) = \begin{bmatrix} (4+4e^2)/\pi \\ (2+2e^2)/\pi \\ (-8-4e^2)/\pi^2 \end{bmatrix},$$

and the derivative computation is complete.

The main appeal of the reverse mode is that its computational complexity is low for the scalar functions $f : \mathbb{R}^n \to \mathbb{R}$ discussed here. The extra arithmetic associated with the gradient computation is at most four or five times the arithmetic needed to evaluate the function alone. Taking the division operation in (8.33) as an example, we see that two multiplications, a division, and an addition are required for (8.33a), while a division and an addition are required for (8.33b). This is about five times as much work as the single division involving these nodes that was performed during the forward sweep.

As we noted above, the forward mode may require up to n times more arithmetic to compute the gradient ∇f than to compute the function f alone, making it appear uncompetitive with the reverse mode. When we consider vector functions $r : \mathbb{R}^n \to \mathbb{R}^m$, the relative costs of the forward and reverse modes become more similar as m increases, as we describe in the next section.

An apparent drawback of the reverse mode is the need to store the entire computational graph, which is needed for the reverse sweep. In principle, storage of this graph is not too difficult to implement. Whenever an elementary operation is performed, we can form and store a new node containing the intermediate result, pointers to the (one or two) parent nodes, and the partial derivatives associated with these arcs. During the reverse sweep, the nodes can be read in the reverse order to that in which they were written, giving a particularly simple access pattern. The process of forming and writing the graph can be implemented as a straightforward extension to the elementary operations via operator overloading (as in ADOL-C [154]). The reverse sweep/gradient evaluation can be invoked as a simple function call.

Unfortunately, the computational graph may require a huge amount of storage. If each node can be stored in 20 bytes, then a function that requires one second of evaluation time on a 100 megaflop computer may produce a graph of up to 2 gigabytes in size. The storage requirements can be reduced, at the cost of some extra arithmetic, by performing partial forward and reverse sweeps on pieces of the computational graph, reevaluating portions of the graph as needed rather than storing the whole structure. Descriptions of this approach, sometimes known as *checkpointing*, can be found in Griewank [150] and Grimm, Pottier, and Rostaing-Schmidt [157]. An implementation of checkpointing in the context of variational data assimilation can be found in Restrepo, Leaf, and Griewank [264] .

VECTOR FUNCTIONS AND PARTIAL SEPARABILITY

So far, we have looked at automatic differentiation of general scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$. In nonlinear least-squares problems (Chapter 10) and nonlinear equations

(Chapter 11), we have to deal with vector functions $r : \mathbb{R}^n \to \mathbb{R}^m$ with m components r_j, $j = 1, 2, \dots, m$. The rightmost column of the computational graph then consists of m nodes, none of which has any children, in place of the single node described above. The forward and reverse modes can be adapted in straightforward ways to find the Jacobian $J(x)$, the $m \times n$ matrix defined in (8.9).

Besides their applications to least-squares and nonlinear-equations problems, automatic differentiation of vector functions is a useful technique for dealing with partially separable functions. We recall that partial separability is commonly observed in large-scale optimization, and we saw in Chapter 7 that there exist efficient quasi-Newton procedures for the minimization of objective functions with this property. Since an automatic procedure for detecting the decomposition of a given function f into its partially separable representation was developed recently by Gay [118], it has become possible to exploit the efficiencies that accrue from this property without asking much information from the user.

In the simplest sense, a function f is partially separable if we can express it in the form

$$f(x) = \sum_{i=1}^{ne} f_i(x), \tag{8.34}$$

where each *element function* $f_i(\cdot)$ depends on just a few components of x. If we construct the vector function r from the partially separable components, that is,

$$r(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_{ne}(x) \end{bmatrix},$$

it follows from (8.34) that

$$\nabla f(x) = J(x)^T e, \tag{8.35}$$

where, as usual, $e = (1, 1, \dots, 1)^T$. Because of the partial separability property, most columns of $J(x)$ contain just a few nonzeros. This structure makes it possible to calculate $J(x)$ efficiently by applying graph-coloring techniques, as we discuss below. The gradient $\nabla f(x)$ can then be recovered from the formula (8.35).

In constrained optimization, it is often beneficial to evaluate the objective function f and the constraint functions c_i, $i \in \mathcal{I} \cup \mathcal{E}$, simultaneously. By doing so, we can take advantage of common expressions (which show up as shared intermediate nodes in the computation graph) and thus can reduce the total workload. In this case, the vector function r can be defined as

$$r(x) = \begin{bmatrix} f(x) \\ [c_j(x)]_{j \in \mathcal{I} \cup \mathcal{E}} \end{bmatrix}.$$

An example of shared intermediate nodes was seen in Figure 8.2, where x_4 is shared during the computation of x_6 and x_7.

CALCULATING JACOBIANS OF VECTOR FUNCTIONS

The forward mode is the same for vector functions as for scalar functions. Given a seed vector p, we continue to associate quantities $D_p x_i$ with the node that calculates each intermediate variable x_i. At each of the rightmost nodes (containing r_j, $j = 1, 2, \ldots, m$), this variable contains the quantity $D_p r_j = (\nabla r_j)^T p$, $j = 1, 2, \ldots, m$. By assembling these m quantities, we obtain $J(x)p$, the product of the Jacobian and our chosen vector p. As in the case of scalar functions ($m = 1$), we can evaluate the complete Jacobian by setting $p = e_1, e_2, \ldots, e_n$ and evaluating the n quantities $D_{e_j} x_i$ simultaneously. For sparse Jacobians, we can use the coloring techniques outlined above in the context of finite-difference methods to make more intelligent and economical choices of the seed vectors p. The factor of increase in cost of arithmetic, when compared to a single evaluation of r, is about equal to the number of seed vectors used.

The key to applying the reverse mode to a vector function $r(x)$ is to choose seed vectors $q \in \mathbb{R}^m$ and apply the reverse mode to the scalar functions $r(x)^T q$. The result of this process is the vector

$$\nabla[r(x)^T q] = \nabla \left[\sum_{j=1}^{m} q_j r_j(x) \right] = J(x)^T q.$$

Instead of the Jacobian–vector product that we obtain with the forward mode, the reverse mode yields a Jacobian-transpose–vector product. The technique can be implemented by seeding the variables $\bar{x}_i$ in the m dependent nodes that contain $r_1, r_2, \ldots, r_m$, with the components $q_1, q_2, \ldots, q_m$ of the vector q. At the end of the reverse sweep, the node for independent variables $x_1, x_2, \ldots, x_n$ will contain

$$\frac{d}{dx_i}\left[r(x)^T q\right], \qquad i = 1, 2, \ldots, n,$$

which are simply the components of $J(x)^T q$.

As usual, we can obtain the full Jacobian by carrying out the process above for the m unit vectors $q = e_1, e_2, \ldots, e_m$. Alternatively, for sparse Jacobians, we can apply the usual coloring techniques to find a smaller number of seed vectors q—the only difference being that the graphs and coloring strategies are defined with reference to the transpose $J(x)^T$ rather than to $J(x)$ itself. The factor of increase in the number of arithmetic operations required, in comparison to an evaluation of r alone, is no more than 5 times the number of seed vectors. (The factor of 5 is the usual overhead from the reverse mode for a scalar function.) The space required for storage of the computational graph is no greater than in the scalar case. As before, we need only store the graph topology information together with the partial derivative associated with each arc.

The forward- and reverse-mode techniques can be combined to cumulatively reveal all the elements of $J(x)$. We can choose a set of seed vectors p for the forward mode to reveal some columns of J, then perform the reverse mode with another set of seed vectors q to reveal the rows that contain the remaining elements.

Finally, we note that for some algorithms, we do not need full knowledge of the Jacobian $J(x)$. For instance, iterative methods such as the inexact Newton method for nonlinear equations (see Section 11.1) require repeated calculation of $J(x)p$ for a succession of vectors p. Each such matrix–vector product can be computed using the forward mode by using a single forward sweep, at a similar cost to evaluation of the function alone.

CALCULATING HESSIANS: FORWARD MODE

So far, we have described how the forward and reverse modes can be applied to obtain first derivatives of scalar and vector functions. We now outline extensions of these techniques to the computation of the Hessian $\nabla^2 f$ of a scalar function f, and evaluation of the Hessian–vector product $\nabla^2 f(x)p$ for a given vector p.

Recall that the forward mode makes use of the quantities $D_p x_i$, each of which stores $(\nabla x_i)^T p$ for each node i in the computational graph and a given vector p. For a given *pair* of seed vectors p and q (both in $\mathbb{R}^n$) we now define another scalar quantity by

$$D_{pq} x_i = p^T (\nabla^2 x_i) q, \tag{8.36}$$

for each node i in the computational graph. We can evaluate these quantities during the forward sweep through the graph, alongside the function values x_i and the first-derivative values $D_p x_i$. The initial values of D_{pq} at the independent variable nodes x_i, $i = 1, 2 \ldots, n$, will be 0, since the second derivatives of x_i are zero at each of these nodes. When the forward sweep is complete, the value of $D_{pq} x_i$ in the rightmost node of the graph will be $p^T \nabla^2 f(x) q$.

The formulae for transformation of the $D_{pq} x_i$ variables during the forward sweep can once again be derived from the chain rule. For instance, if x_i is obtained by adding the values at its two parent nodes, $x_i = x_j + x_k$, the corresponding accumulation operations on $D_p x_i$ and $D_{pq} x_i$ are as follows:

$$D_p x_i = D_p x_j + D_p x_k, \qquad D_{pq} x_i = D_{pq} x_j + D_{pq} x_k. \tag{8.37}$$

The other binary operations $-$, $\times$, $/$ are handled similarly. If x_i is obtained by applying the unitary transformation L to x_j, we have

$$x_i = L(x_j), \tag{8.38a}$$

$$D_p x_i = L'(x_j)(D_p x_j), \tag{8.38b}$$

$$D_{pq} x_i = L''(x_j)(D_p x_j)(D_q x_j) + L'(x_j) D_{pq} x_j. \tag{8.38c}$$

We see in (8.38c) that computation of $D_{pq}x_i$ can rely on the first-derivative quantities $D_p x_i$ and $D_q x_i$, so both these quantities must be accumulated during the forward sweep as well.

We could compute a general dense Hessian by choosing the pairs (p, q) to be all possible pairs of unit vectors (e_j, e_k), for $j = 1, 2, \dots, n$ and $k = 1, 2, \dots, j$, a total of $n(n+1)/2$ vector pairs. (Note that we need only evaluate the lower triangle of $\nabla^2 f(x)$, because of symmetry.) When we know the sparsity structure of $\nabla^2 f(x)$, we need evaluate $D_{e_j e_k} x_i$ only for the pairs (e_j, e_k) for which the (j, k) component of $\nabla^2 f(x)$ is possibly nonzero.

The total increase factor for the number of arithmetic operations, compared with the amount of arithmetic to evaluate f alone, is a small multiple of $1 + n + N_z(\nabla^2 f)$, where $N_z(\nabla^2 f)$ is the number of elements of $\nabla^2 f$ that we choose to evaluate. This number reflects the evaluation of the quantities x_i, $D_{e_j} x_i$ $(j = 1, 2, \dots, n)$, and $D_{e_j e_k} x_i$ for the $N_z(\nabla^2 f)$ vector pairs (e_j, e_k). The "small multiple" results from the fact that the update operations for $D_p x_i$ and $D_{pq} x_i$ may require a few times more operations than the update operation for x_i alone; see, for example, (8.38). One storage location per node of the graph is required for each of the $1 + n + N_z(\nabla^2 f)$ quantities that are accumulated, but recall that storage of node i can be overwritten once all its children have been evaluated.

When we do not need the complete Hessian, but only a matrix–vector product involving the Hessian (as in the Newton–CG algorithm of Chapter 7), the amount of arithmetic is, of course, smaller. Given a vector $q \in \mathbb{R}^n$, we use the techniques above to compute the first-derivative quantities $D_{e_1} x_i, \dots D_{e_n} x_i$ and $D_q x_i$, as well as the second-derivative quantities $D_{e_1 q} x_i, \dots, D_{e_n q} x_i$, during the forward sweep. The final node will contain the quantities

$$e_j^T \left(\nabla^2 f(x)\right) q = \left[\nabla^2 f(x) q\right]_j, \quad j = 1, 2, \dots, n,$$

which are the components of the vector $\nabla^2 f(x) q$. Since $2n + 1$ quantities in addition to x_i are being accumulated during the forward sweep, the increase factor in the number of arithmetic operations increases by a small multiple of $2n$.

An alternative technique for evaluating sparse Hessians is based on the forward-mode propagation of first and second derivatives of *univariate* functions. To motivate this approach, note that the (i, j) element of the Hessian can be expressed as follows:

$$\begin{aligned}[\nabla^2 f(x)]_{ij} &= e_i^T \nabla^2 f(x) e_j \\ &= \frac{1}{2}\left[(e_i + e_j)^T \nabla^2 f(x)(e_i + e_j) - e_i^T \nabla^2 f(x) e_i - e_j^T \nabla^2 f(x) e_j\right].\end{aligned} \tag{8.39}$$

We can use this interpolation formula to evaluate $[\nabla^2 f(x)]_{ij}$, provided that the second derivatives $D_{pp}x_k$, for $p = e_i$, $p = e_j$, $p = e_i + e_j$, and all nodes x_k, have been evaluated during the forward sweep through the computational graph. In fact, we can evaluate all the nonzero elements of the Hessian, provided that we use the forward mode to evaluate $D_p x_k$ and $D_{pp}x_k$ for a selection of vectors p of the form $e_i + e_j$, where i and j are both indices in $\{1, 2, \dots, n\}$, possibly with $i = j$.

One advantage of this approach is that it is no longer necessary to propagate "cross terms" of the form $D_{pq}x_k$ for $p \neq q$ (see, for example, (8.37) and (8.38c)). The propagation formulae therefore simplify somewhat. Each $D_{pp}x_k$ is a function of x_ℓ, $D_p x_\ell$, and $D_{pp}x_\ell$ for all parent nodes ℓ of node k.

Note, too, that if we define the univariate function ψ by

$$\psi(t) = f(x + tp), \tag{8.40}$$

then the values of $D_p f$ and $D_{pp} f$, which emerge at the completion of the forward sweep, are simply the first two derivatives of ψ evaluated at $t = 0$; that is,

$$D_p f = p^T \nabla f(x) = \psi'(t)|_{t=0}, \qquad D_{pp} f = p^T \nabla^2 f(x) p = \psi''(t)|_{t=0}.$$

Extension of this technique to third, fourth, and higher derivatives is possible. Interpolation formulae analogous to (8.39) can be used in conjunction with higher derivatives of the univariate functions ψ defined in (8.40), again for a suitably chosen set of vectors p, where each p is made up of a sum of unit vectors e_i. For details, see Bischof, Corliss, and Griewank [26].

CALCULATING HESSIANS: REVERSE MODE

We can also devise schemes based on the reverse mode for calculating Hessian–vector products $\nabla^2 f(x)q$, or the full Hessian $\nabla^2 f(x)$. A scheme for obtaining $\nabla^2 f(x)q$ proceeds as follows. We start by using the forward mode to evaluate both f and $\nabla f(x)^T q$, by accumulating the two variables x_i and $D_q x_i$ during the forward sweep in the manner described above. We then apply the reverse mode in the normal fashion to the computed function $\nabla f(x)^T q$. At the end of the reverse sweep, the nodes $i = 1, 2, \dots, n$ of the computational graph that correspond to the independent variables will contain

$$\frac{\partial}{\partial x_i}(\nabla f(x)^T q) = \left[\nabla^2 f(x) q\right]_i, \qquad i = 1, 2, \dots, n.$$

The number of arithmetic operations required to obtain $\nabla^2 f(x)q$ by this procedure increases by only a modest factor, independent of n, over the evaluation of f alone. By the usual analysis for the forward mode, we see that the computation of f and $\nabla f(x)^T q$ jointly requires a small multiple of the operation count for f alone, while the reverse sweep introduces a further factor of at most 5. The total increase factor is approximately 12 over the evaluation of f alone. If the entire Hessian $\nabla^2 f(x)$ is required, we could apply the procedure just described with $q = e_1, e_2, \ldots, e_n$. This approach would introduce an additional factor of n into the operation count, leading to an increase of at most $12n$ over the cost of f alone.

Once again, when the Hessian is sparse with known structure, we may be able to use graph-coloring techniques to evaluate this entire matrix using many fewer than n seed vectors. The choices of q are similar to those used for finite-difference evaluation of the Hessian, described above. The increase in operation count over evaluating f alone is a multiple of up to $12N_c(\nabla^2 f)$, where N_c is the number of seed vectors q used in calculating $\nabla^2 f$.

CURRENT LIMITATIONS

The current generation of automatic differentiation tools has proved its worth through successful application to some large and difficult design optimization problems. However, these tools can run into difficulties with some commonly used programming constructs and some implementations of computer arithmetic. As an example, if the evaluation of $f(x)$ depends on the solution of a partial differential equation (PDE), then the computed value of f may contain truncation error arising from the finite-difference or the finite-element technique that is used to solve the PDE numerically. That is, we have $\hat{f}(x) = f(x) + \tau(x)$, where $\hat{f}(\cdot)$ is the computed value of $f(\cdot)$ and $\tau(\cdot)$ is the truncation error. Though $|\tau(x)|$ is usually small, its derivative $\tau'(x)$ may not be, so the error in the computed derivative $\hat{f}'(x)$ is potentially large. (The finite-difference approximation techniques discussed in Section 8.1 experience the same difficulty.) Similar problems arise when the computer uses piecewise rational functions to approximate trigonometric functions.

Another source of potential difficulty is the presence of branching in the code to improve the speed or accuracy of function evaluation in certain domains. A pathological example is provided by the linear function $f(x) = x - 1$. If we used the following (perverse, but valid) piece of code to evaluate this function,

```
if (x = 1.0) then f = 0.0 else f = x − 1.0,
```

then by applying automatic differentiation to this procedure we would obtain the derivative value $f'(1) = 0$. For a discussion of such issues and an approach to dealing with them, see Griewank [151, 152].

In conclusion, automatic differentiation should be regarded as a set of increasingly sophisticated techniques that enhances optimization algorithms, allowing them to be applied more widely to practical problems involving complicated functions. By providing sensitivity information, it helps the modeler to extract more information from the results of the computation. Automatic differentiation should not be regarded as a panacea that absolves the user altogether from the responsibility of thinking about derivative calculations.

NOTES AND REFERENCES

A comprehensive and authoritative reference on automatic differentiation is the book of Griewank [152]. The web site `www.autodiff.org` contains a wealth of current information about theory, software, and applications. A number of edited collections of papers on automatic differentiation have appeared since 1991; see Griewank and Corliss [153], Berz et al. [20], and Bücker et al. [40]. An historical paper of note is Corliss and Rall [78], which includes an extensive bibliography. Software tool development in automatic differentiation makes use not only of forward and reverse modes but also includes "mixed modes" and "cross-country algorithms" that combine the two approaches; see for example Naumann [222].

The field of automatic differentiation grew considerably during the 1990s, and and a number of good software tools appeared. These included ADIFOR [25] and ADIC [28], and ADOL-C [154]. Tools developed in more recent years include TAPENADE, which accepts Fortran code through a web server and returns differentiated code; TAF, a commercial tool that also performs source-to-source automatic differentiation of Fortran codes; OpenAD, which works with Fortran, C, and C++; and TOMLAB/MAD, which works with MATLAB code.

The technique for calculating the gradient of a partially separable function was described by Bischof et al. [24], whereas the computation of the Hessian matrix has been considered by several authors; see, for example, Gay [118].

The work of Coleman and Moré [69] on efficient estimation of Hessians was predated by Powell and Toint [261], who did not use the language of graph coloring but nevertheless devised highly effective schemes. Software for estimating sparse Hessians and Jacobians is described by Coleman, Garbow, and Moré [66, 67]. The recent paper of Gebremedhin, Manne, and Pothen [120] contains a comprehensive discussion of the application of graph coloring to both finite difference and automatic differentiation techniques.

✐ Exercises

✐ **8.1** Show that a suitable value for the perturbation ϵ in the central-difference formula is $\epsilon = \mathbf{u}^{1/3}$, and that the accuracy achievable by this formula when the values of f contain roundoff errors of size $\mathbf{u}$ is approximately $\mathbf{u}^{2/3}$. (Use similar assumptions to the ones used to derive the estimate (8.6) for the forward-difference formula.)

✐ **8.2** Derive a central-difference analogue of the Hessian–vector approximation formula (8.20).

8.3 Verify the formula (8.21) for approximating an element of the Hessian using only function values.

8.4 Verify that if the Hessian of a function f has nonzero diagonal elements, then its adjacency graph is a subgraph of the intersection graph for ∇f. In other words, show that any arc in the adjacency graph also belongs to the intersection graph.

8.5 Draw the adjacency graph for the function f defined by (8.22). Show that the coloring scheme in which node 1 has one color while nodes $2, 3, \ldots, n$ have another color is valid. Draw the intersection graph for ∇f.

8.6 Construct the adjacency graph for the function whose Hessian has the nonzero structure

$$\begin{bmatrix} \times & \times & \times & \times & & \\ \times & \times & \times & & \times & \\ \times & \times & \times & & & \times \\ \times & & & \times & & \\ & \times & & & \times & \\ & & \times & & & \times \end{bmatrix},$$

and find a valid coloring scheme with just four colors.

8.7 Trace the computations performed in the forward mode for the function $f(x)$ in (8.26), expressing the intermediate derivatives ∇x_i, $i = 4, 5, \ldots, 9$ in terms of quantities available at their parent nodes and then in terms of the independent variables x_1, x_2, x_3.

8.8 Formula (8.30) showed the gradient operations associated with scalar division. Derive similar formulae for the following operations:

$$\begin{aligned} (s, t) &\to s + t && \text{addition;} \\ t &\to e^t && \text{exponentiation;} \\ t &\to \tan(t) && \text{tangent;} \\ (s, t) &\to s^t. \end{aligned}$$

8.9 By calculating the partial derivatives $\partial x_j / \partial x_i$ for the function (8.26) from the expressions (8.27), verify the numerical values for the arcs in Figure 8.3 for the evaluation point $x = (1, 2, \pi/2)^T$. Work through the remaining details of the reverse sweep process, indicating the order in which the nodes become finalized.

✐ **8.10** Using (8.33) as a guide, describe the reverse sweep operations corresponding to the following elementary operations in the forward sweep:

$$\begin{aligned} x_k &\leftarrow x_i x_j \qquad \text{multiplication;} \\ x_k &\leftarrow \cos(x_i) \qquad \text{cosine.} \end{aligned}$$

In each case, compare the arithmetic workload in the reverse sweep to the workload required for the forward sweep.

✐ **8.11** Define formulae similar to (8.37) for accumulating the first derivatives $D_p x_i$ and the second derivatives $D_{pq} x_i$ when x_i is obtained from the following three binary operations: $x_i = x_j - x_k$, $x_i = x_j x_k$, and $x_i = x_j / x_k$.

✐ **8.12** By using the definitions (8.28) of $D_p x_i$ and (8.36) of $D_{pq} x_i$, verify the differentiation formulae (8.38) for the unitary operation $x_i = L(x_j)$.

✐ **8.13** Let $a \in \mathbb{R}^n$ be a fixed vector and define f as $f(x) = \frac{1}{2}\left(x^T x + \left(a^T x\right)^2\right)$. Count the number of operations needed to evaluate f, ∇f, $\nabla^2 f$, and the Hessian–vector product $\nabla^2 f(x) p$ for an arbitrary vector p.