CHAPTER 4
DERIVATIVES

4.1 Introduction

Because many engineering models are continuous and differentiable, it is possible to obtain derivatives of response quantities with respect to design variables. Some of the most powerful optimization algorithms rely on derivatives to compute search directions, determine stopping criteria and examine the sensitivity of a design to small changes. Inaccurate derivatives can cause premature termination or a failure to make progress. Further, the calculation of derivatives often represents a significant proportion of the computation for an optimization. Thus we are interested in methods to compute derivatives accurately and efficiently. That is the subject of this chapter. Additional information on this subject can be found in Martins et al. [16] and Nocedal and Wright [17].

4.2 Numerical Differentiation

4.2.1 Finite Difference Methods

One of the most general methods for obtaining derivatives is the finite difference method. It is easy to implement and only requires that we are able to compute function values. Thus this method is often used when we don't have access to the analysis software and need to treat it as a "black box." However, it is not as accurate as other methods for the same level of computational expense.

This method flows from the definition of a derivative:

$$\frac{df}{dx} \equiv \lim_{\Delta x \to 0} \frac{f\left(x+\Delta x\right) - f\left(x\right)}{\Delta x}$$

only instead of an infinitesimally small Δx, we have a finite Δx.

4.2.2 Truncation Error

4.2.2.1 Derivation

The error associated with using a finite Δx can be derived from a Taylor series expanded about $x + \Delta x$. For a function of only one variable:

$$f\left(x+\Delta x\right) = f\left(x\right) + \frac{df}{dx}\Delta x + \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x^2 + \ldots \tag{4.1}$$

Solving for the derivative:

$$\frac{df}{dx} = \frac{f\left(x+\Delta x\right) - f\left(x\right)}{\Delta x} - \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x + \ldots \tag{4.2}$$

If we approximate this *forward difference* derivative as,

$$\frac{df}{dx} = \frac{f\left(x+\Delta x\right) - f\left(x\right)}{\Delta x} \tag{4.3}$$

then we see we have a *truncation error* of $\left(-\frac{1}{2}\frac{d^2 f}{dx^2}\Delta x + \ldots \right)$ This error will be dominated by the first term, $-\frac{1}{2}\frac{d^2 f}{dx^2}\Delta x$, which is proportional to Δx. Since we don't usually know the sign or magnitude of the second derivative, we will assume it to be positive and indicate it *is on the order of* Δx. Thus to reduce this error, we should make Δx small.

We can also derive a *central difference* derivative:

$$f\left(x+\Delta x\right) = f\left(x\right) + \frac{df}{dx}\Delta x + \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x^2 + \frac{1}{6}\frac{d^3 f}{dx^3}\Delta x^3 + \ldots \tag{4.4}$$

$$f\left(x-\Delta x\right) = f\left(x\right) - \frac{df}{dx}\Delta x + \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x^2 - \frac{1}{6}\frac{d^3 f}{dx^3}\Delta x^3 + \ldots \tag{4.5}$$

Subtracting the second expression from the first,

$$f\left(x+\Delta x\right) - f\left(x-\Delta x\right) = 2\frac{df}{dx}\Delta x + \frac{1}{3}\frac{d^3 f}{dx^3}\Delta x^3 + \ldots \tag{4.6}$$

Solving for the derivative,

$$\frac{df}{dx} = \frac{f\left(x+\Delta x\right) - f\left(x-\Delta x\right)}{2\Delta x} - \frac{1}{6}\frac{d^3 f}{dx^3}\Delta x^2 + \ldots \tag{4.7}$$

If we approximate the derivative as,

$$\frac{df}{dx} = \frac{f\left(x+\Delta x\right) - f\left(x-\Delta x\right)}{2\Delta x} \tag{4.8}$$

then the truncation error is on the order of Δx^2. Assuming $\Delta x < 1.0$, the central difference method should have less error than the forward difference method. For example, if $\Delta x =$

0.01, the truncation error for the central difference derivative should be on the order of $(0.01)^2 = 0.0001$.

If the error of the central difference method is better, why isn't it always used? The central difference method requires two functions calls per derivative instead of one for the forward difference method. If we have 10 design variables, this means we have to call the analysis routine 20 times (twice for each variable) to get the derivatives instead of ten times. This can be prohibitively expensive.

4.2.2.2 Example of Truncation Error

We wish to compute the derivative of the function $f(x) = x^3 + x^{1/2}$ at the point $x = 3$.

The true derivative at this point is 27.2886751. We will use a forward difference derivative with $\Delta x = 0.01$,

$$\frac{df}{dx} = \frac{f(3+0.01) - f(3)}{0.01} = \frac{29.0058362 - 28.7320508}{0.01} = 27.37385$$

The absolute value of error is 0.08512.

Now suppose we use $\Delta x = 0.01$ with a central difference derivative:

$$\frac{df}{dx} = \frac{f(3+0.01) - f(3-0.01)}{2*0.01} = \frac{29.0058362 - 28.4600606}{0.02} = 27.28878$$

The absolute error has decreased to 0.000105

4.2.3 Round-off Error

4.2.3.1 Errors in Function Values

Besides truncation error, we also have *round-off* error. This is error associated with a lack of significant figures in the function values. Although this can arise from storing a real number in binary form on the computer, we can largely overcome this problem by using double precision for all real variables. The more likely source of round-off error is lack of significant figures in the computer model, which often involves numerical approximations. Returning to our expression for a forward difference derivative, we will now consider we have some error ε in the representation of the true value of the functions. We will assume this error is approximately the same for both values and is positive.

$$\frac{df}{dx} = \frac{f(x+\Delta x) + \varepsilon - f(x) + \varepsilon}{\Delta x} - \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x + \ldots \qquad (4.9)$$

$$\frac{df}{dx} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \frac{2\varepsilon}{\Delta x} - \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x + \ldots \tag{4.10}$$

where we see we have a new error term, $\frac{2\varepsilon}{\Delta x}$. To make this term small, we should make Δx large.

4.2.3.2 Subtractive Cancellation

Besides a lack of significant figures in the functions values, we can lose a lot of significant figures in a finite difference calculation because of *subtractive cancellation.* This refers to the loss of significance caused by the subtraction of two large, nearly equal numbers in the numerator (4.3). This can easily cut our significant figures in half. We will consider subtractive cancellation to be part of round-off error and will denote this by ξ (assumed positive) and include it in our estimate of round-off error:

$$\left(\frac{2\varepsilon + \xi}{\Delta x}\right) \tag{4.11}$$

4.2.3.3 Example of Round-off Error with Subtractive Cancellation

We will illustrate the effect of a lack of significant figures and subtractive cancellation on the example in Section 4.2.2, i.e., the function $f(x) = x^3 + x^{1/2}$ at the point $x = 3$.

We will use a forward difference derivative with $\Delta x = 0.0001$ Further we will assume that because of noise in our model, the function values are limited to seven significant figures.

$$\frac{df}{dx} = \frac{f(3+0.0001) - f(3)}{0.0001} = \frac{28.73478 - 28.73205}{0.0001} = \frac{0.00273}{0.0001} = 27.3$$

We see that after subtracting the two values in the numerator, we are left with 0.00273. This difference only has three significant figures; we have lost four significant figures by this calculation.

4.2.4 Total Error

Recall that the true derivative is,

$$\frac{df}{dx} = \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x + \ldots \tag{4.12}$$

but we are estimating it as,

$$\frac{df}{dx} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \left(\frac{2\varepsilon + \xi}{\Delta x}\right) \tag{4.13}$$

where this last term is the round-off error. The total error can be approximated as,

$$\left(\frac{2\varepsilon + \xi}{\Delta x}\right) + \left|\frac{1}{2}\frac{d^2 f}{dx^2}\Delta x\right| \tag{4.14}$$

Thus we have two competing errors, truncation error and round-off error. To reduce truncation error, Δx should be small; to reduce round-off error, Δx should be large. We will have to compromise.

4.2.5 Example of Truncation and Round-off Error

We will continue our example of 4.2.3; however we will reduce the number of significant figures from seven to five to more dramatically highlight the error trade-off. We will calculate error for a whole range of Δx's:

Δx	Error
1.0	9.98
0.1	0.911
0.01	0.1113
0.001	0.2887
0.0001	2.7113
0.00001	27.728

If we plot this data,

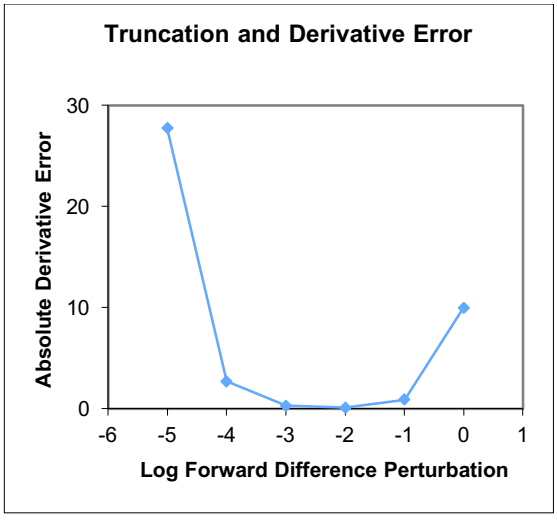

Fig. 4.1. Plot of total error for example

As we expected, we see that the data have a "U" shape. At small perturbations, round-off error dominates. At large perturbations, truncation error dominates. There is an optimal perturbation that gives us the minimum error in the numerical derivative.

Usually we don't know the true value of the derivative so we can't easily determine the optimal perturbation. However, understanding the source and control of errors in these derivatives can be very helpful. If we suspect we have a noisy model, i.e. we don't have very many significant figures in our functions, *we should use a central difference method with a large perturbation*. The large perturbation helps reduce the effects of noise, and the central difference method helps control the truncation error associated with a large perturbation.

4.2.6 Partial Derivatives

The concepts of the preceding sections on derivatives extend directly to partial derivatives. For two variables, x_1 and x_2, a finite forward difference partial derivative, $\frac{\partial f}{\partial x_1}$, would be given by,

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \Delta x_1, x_2) - f(x_1, x_2)}{\Delta x_1} \tag{4.15}$$

Note that only x_1 is perturbed to evaluate the derivative. This variable would be set back to its base value and x_2 perturbed to find $\frac{\partial f}{\partial x_2}$.

In a similar manner, a central difference partial derivative for $\frac{\partial f}{\partial x_1}$ would be given by,

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \Delta x_1, x_2) - f(x_1 - \Delta x_1, x_2)}{2\Delta x_1} \tag{4.16}$$

4.3 Complex-Step Derivative Approximation

4.3.1 Derivation

The discussion in this section closely follows that given by Martins et al. [16]. Like the finite difference approximation, the complex-step approximation can be derived using a Taylor series, only instead of expanding the series about $x + \Delta x$, we expand the series about an imaginary step $x + i\Delta x$:

$$f\left(x + i\Delta x\right) = f\left(x\right) + \frac{df}{dx} i\Delta x - \frac{1}{2}\frac{d^2 f}{dx^2}\Delta x^2 - \frac{1}{6}\frac{d^3 f}{dx^3} i\Delta x^3 \tag{4.17}$$

where we have taken advantage of the fact that $i^2 = -1$. If we group imaginary parts together and divide by Δx:

$$\frac{df}{dx} = \frac{\mathrm{Im}\left[f\left(x + i\Delta x\right)\right]}{\Delta x} + \frac{1}{6}\frac{d^3 f}{dx^3}\Delta x^2 \tag{4.18}$$

We see that the approximation involves a truncation error on the order of Δx^2. Note that unlike the regular finite difference method, we do not have the loss of significant figures associated with subtractive cancellation. This is because the base point, $f(x)$, as a real, does not show up in the imaginary formula. To counter the effects of subtractive cancellation in the regular finite difference method, we needed to keep Δx large; this results in a larger truncation error. With the complex-step, we can drive Δx small to reduce truncation error without a corresponding increase in round-off error. This allows us to get significantly better accuracy.

Because there is no subtractive cancellation, the perturbation for the complex-step can be made extremely small. At some point (~10^{-8} for the problem below) the estimate will be as accurate as the function evaluation. The step can be made almost as small as the smallest number the computer can represent and still be accurate. A typical perturbation for this method is 1e-30.

4.3.2 Example of Complex-Step

We will repeat the example of 4.2.3 which illustrated subtractive cancelling for the forward difference derivative. We will keep the same function, $f(x) = x^3 + x^{1/2}$ at the point $x = 3$, with seven significant figures. Initially, for comparison purposes, we will keep the perturbation of $\Delta x = 0.0001$. The derivative is given by,

$$\frac{df}{dx} = \frac{\mathrm{Im}\, f(3 + i0.0001)}{0.0001} = 27.28868 \tag{4.19}$$

The error has decreased from 0.011325 to 4.87e-06. If we use double precision (~15 significant figures) and reduce the perturbation to 10^{-8}, the error drops to zero, within the limits of the machine.

The complex-step provides superior accuracy for somewhat more computational cost than the forward difference derivative. For example, in a problem with 18 variables involving aerodynamic and structural analysis, Martins et al.[18] compared the accuracy and computational efficiency of finite difference to complex-step:

"The cost of the complex-step procedure is more than twice that of the finite-difference procedure since the function evaluations require complex arithmetic. We feel, however, that the complex-step calculations are worth this cost penalty since there is no need to find an acceptable step size *a priori*, as in the case of the finite-difference approximations. Again, we would like to emphasize that while there was considerable effort involved in obtaining reasonable finite-difference results by optimizing the step sizes, no such effort was necessary when using the complex-step method."

To implement this method, we need to be able to modify the analysis software. Specifically, all real type variables should be declared to be complex. Also, all functions and operators need to be defined for complex arguments.

4.4 Analytic Derivatives of Simultaneous Equations

4.4.1 Introduction

It is often the case in engineering analysis that we need to solve sets of simultaneous equations. These can be represented in the form,

$$\mathbf{Ku} = \mathbf{f} \tag{4.20}$$

Where **K** is a *p x p* coefficient matrix, **u** is a solution vector of length *p* (we will later refer to the variables **u** as the *state* variables), and **f** is a vector, also of length *p*, of right hand sides.

As an example, we could assume (4.20) represents a linear, elastic finite element structural problem, where **K** is the stiffness matrix, **u** is the vector of displacements and **f** is the vector of applied forces. **K** is a function of the design variables **x,** of length *n*, which could be the cross-sectional areas. The solution can be written for convenience as,

$$\mathbf{u} = \mathbf{K}^{-1}\mathbf{f} \tag{4.21}$$

although we would likely not actually invert **K** but use other, more efficient means to solve for the solution. In a structural design problem, we are not only interested in the deflections but also the stresses, which are related to the deflections by the equation,

$$\sigma = \mathbf{Su} \tag{4.22}$$

where σ is a vector of length *m* and **S** is a *m x p* stress-displacement matrix. We are interested to find both $\frac{\partial \mathbf{u}}{\partial x_i}$ and $\frac{\partial \sigma}{\partial x_i}$.

4.4.2 The Direct Method

4.4.2.1 Development

One way of solving for derivatives in the case of simultaneous equations is called the *direct method*.

Taking the derivative of (4.20), we have

$$\mathbf{K}\frac{\partial \mathbf{u}}{\partial x_i} + \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u} = \frac{\partial \mathbf{f}}{\partial x_i} \tag{4.23}$$

Solving for $\frac{\partial \mathbf{u}}{\partial x_i}$ we have,

$$\frac{\partial \mathbf{u}}{\partial x_i} = \mathbf{K}^{-1}\left[\frac{\partial \mathbf{f}}{\partial x_i} - \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u}\right] \tag{4.24}$$

Everything on the right hand side of (4.24) can be calculated. The matrix $\mathbf{K}^{-1}$ is known from solving (4.21) previously, and so doesn't need to be computed again—this is where we gain a lot of efficiency. If we solve (4.20) by some method that doesn't involve computing $\mathbf{K}^{-1}$, then we can employ that same method to solve (4.24). The vector **u** has previously been computed. The dependence of **f** and **K** on **x** is known or can be found (we may, however, wish to use numerical methods to compute these—as mentioned in the section which follows). We therefore have everything we need to find $\frac{\partial \mathbf{u}}{\partial x_i}$.

Equation (4.24) gives the derivatives of all displacements with respect to one variable. Assuming **S** is independent of **x**, we can find the derivatives of the stresses using,

$$\frac{\partial \sigma}{\partial x_i} = \mathbf{S}\frac{\partial \mathbf{u}}{\partial x_i} \tag{4.25}$$

We can combine (4.24) and (4.25) to give,

$$\frac{\partial \sigma}{\partial x_i} = \mathbf{S}\ \mathbf{K}^{-1}\left[\frac{\partial \mathbf{f}}{\partial x_i} - \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u}\right] \tag{4.26}$$

To find the derivatives of all stresses with respect to all variables, we would calculate (4.26) for each variable. This is like solving the system of equations represented by (4.26) for *n* right hand sides.

4.4.2.2 A Semi-Analytic Approach

It may be convenient to use finite difference to find $\frac{\partial \mathbf{f}}{\partial x_i}$ or $\frac{\partial \mathbf{K}}{\partial x_i}$. That is, we may approximate (4.26) by,

$$\frac{\partial \sigma}{\partial x_i} = \mathbf{S}\ \mathbf{K}^{-1}\left[\frac{\mathbf{f}(x_i + \Delta x_i) - \mathbf{f}(x_i)}{\Delta x_i} - \frac{\mathbf{K}(x_i + \Delta x_i) - \mathbf{K}(x_i)}{\Delta x_i}\mathbf{u}\right] \tag{4.27}$$

This may be convenient because the dependence of **K** or **f** on **x** may not be easy to determine analytically. For example, in a structural finite element program, the global stiffness matrix is assembled from the element stiffness matrices, and the dependence on **x** may be difficult to

keep track of. Computing $\frac{\partial \mathbf{K}}{\partial x_i}$ or $\frac{\partial \mathbf{f}}{\partial x_i}$ using a finite difference method is relatively inexpensive; furthermore, the finite difference method will be exact if the elements of **K** or **f** are linear functions of the design variables, which is usually the case.

4.4.2.3 Example of the Direct Method

Suppose we have the following **K** matrix and **x** vector,

$$\mathbf{K} = \begin{bmatrix} 5x_1 & 2(x_1+x_2) & 3x_2 \\ 2(x_1+x_2) & 8x_2 & 4 \\ 3x_2 & 4 & 15 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.28}$$

where the current point is $\mathbf{x}^T = \begin{bmatrix} 1 & 2 \end{bmatrix}$. The matrix **K** is therefore,

$$\mathbf{K} = \begin{bmatrix} 5 & 6 & 6 \\ 6 & 16 & 4 \\ 6 & 4 & 15 \end{bmatrix} \tag{4.29}$$

We will also define **S** and **f** to be,

$$\mathbf{S} = \begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix} \qquad \mathbf{f}^T = \begin{bmatrix} 1 & 0 & 4 \end{bmatrix} \tag{4.30}$$

We assume **S** and **f** are not functions of **x**, so $\frac{\partial \mathbf{f}}{\partial x_i} = 0$ in (4.26).

The quantities $\mathbf{K}^{-1}$ and **u** have already been computed in order to solve (4.21) (we will use the inverse for sake of simplicity). They are given by,

$$\mathbf{K}^{-1} = \begin{bmatrix} 0.767 & -0.226 & -0.247 \\ -0.226 & 0.134 & 0.055 \\ -0.247 & 0.055 & 0.151 \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} -0.219 \\ -0.007 \\ 0.356 \end{bmatrix} \tag{4.31}$$

The following can be calculated,

$$\frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u} = \begin{bmatrix} 5 & 2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.219 \\ -0.007 \\ 0.356 \end{bmatrix} = \begin{bmatrix} -1.110 \\ -0.438 \\ 0 \end{bmatrix} \tag{4.32}$$

$$\frac{\partial \mathbf{K}}{\partial x_2}\mathbf{u} = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 8 & 0 \\ 3 & 0 & 0 \end{bmatrix}\begin{bmatrix} -0.219 \\ -0.007 \\ 0.356 \end{bmatrix} = \begin{bmatrix} 1.055 \\ -0.493 \\ -0.658 \end{bmatrix} \tag{4.33}$$

The vector $\frac{\partial \sigma}{\partial x_1}$ is therefore given by,

$$\frac{\partial \sigma}{\partial x_1} = -\mathbf{S}\,\mathbf{K}^{-1}\left[\frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right]$$

$$\frac{\partial \sigma}{\partial x_1} = -\begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix}\begin{bmatrix} 0.767 & -0.226 & -0.247 \\ -0.226 & 0.134 & 0.055 \\ -0.247 & 0.055 & 0.151 \end{bmatrix}\begin{bmatrix} -1.110 \\ -0.438 \\ 0 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_1} = -\begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix}\begin{bmatrix} -0.752 \\ 0.192 \\ 0.250 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_1} = \begin{bmatrix} 3.137 \\ -4.486 \end{bmatrix} \tag{4.34}$$

In like manner,

$$\frac{\partial \sigma}{\partial x_2} = -\mathbf{S}\,\mathbf{K}^{-1}\left[\frac{\partial \mathbf{K}}{\partial x_2}\mathbf{u}\right]$$

$$\frac{\partial \sigma}{\partial x_2} = -\begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix}\begin{bmatrix} 0.767 & -0.226 & -0.247 \\ -0.226 & 0.134 & 0.055 \\ -0.247 & 0.055 & 0.151 \end{bmatrix}\begin{bmatrix} 1.055 \\ -0.493 \\ -0.658 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_2} = -\begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix}\begin{bmatrix} 1.083 \\ -0.340 \\ -0.386 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_2} = \begin{bmatrix} -2.805 \\ 9.036 \end{bmatrix} \tag{4.35}$$

4.4.3 The Adjoint Method

4.4.3.1 Development

There is another way we might approach computing derivatives of simultaneous equations. It is called the *adjoint method*. We will see that the adjoint and the direct methods are complementary to each other.

To discuss the adjoint method, we will begin with the equation,

$$\frac{\partial \sigma}{\partial x_1} = \mathbf{S}\ \mathbf{K}^{-1}\left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right] \tag{4.36}$$

For the moment, suppose we are interested in computing just the scalar, $\frac{\partial \sigma_1}{\partial x_1}$. This can be found by taking the first row of the stress-displacement matrix, **S**, which we will call $\mathbf{s}_1^T$, and using it in (4.36),

$$\frac{\partial \sigma_1}{\partial x_1} = \underbrace{\mathbf{s}_1^T\ \mathbf{K}^{-1}}\left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right] \tag{4.37}$$

Let us now consider the term with the brace underneath, i.e. $\mathbf{s}_1^T\ \mathbf{K}^{-1}$. We note that $\mathbf{s}_1^T$ is a *1 x p* vector, $\mathbf{K}^{-1}$ is a *p x p* matrix, and their product results in a *p* length row vector. We will define this vector to be the *adjoint vector* **v**:

$$\mathbf{v}_1^T = \mathbf{s}_1^T\ \mathbf{K}^{-1} \tag{4.38}$$

If we post-multiply by **K**:

$$\mathbf{v}_1^T\mathbf{K} = \mathbf{s}_1^T \tag{4.39}$$

and take the transpose of both sides, we see that **v** can be viewed as the solution to the following *adjoint* equations,

$$\mathbf{K}^T\mathbf{v}_1 = \mathbf{s}_1 \tag{4.40}$$

We need to solve the system of equations represented by (4.40) *m* times, *once for each function*. With the direct method, we solved a similar system of equations, (4.26), *n* times, *once for each variable*. If we set $\mathbf{v}_1^T = \mathbf{s}_1^T\ \mathbf{K}^{-1}$ and substitute it into (4.37) we have,

$$\frac{\partial \sigma_1}{\partial x_1} = \mathbf{v}_1^T\left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right] \tag{4.41}$$

And for the next function,

$$\frac{\partial \sigma_2}{\partial x_1} = \mathbf{v}_2^T \left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1} \mathbf{u} \right]$$

Or for all stresses with respect to x_1 :

$$\frac{\partial \sigma}{\partial x_1} = \mathbf{V} \left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1} \mathbf{u} \right]$$

where $\mathbf{V} = \mathbf{S}\mathbf{K}^{-1}$

To apply the adjoint method, we calculate the **V** matrix and then post multiply this matrix by the vector in brackets. This gives us the derivatives of all stresses with respect to one variable. We then do this for each variable in turn. As will be shown in a following section, the adjoint method results in less computation than the direct method when we have more variables than functions.

4.4.3.2 Example of the Adjoint Method

We will take our previous example and apply the adjoint method to it. Recalling,

$$\mathbf{K}^{-1} = \begin{bmatrix} 0.767 & -0.226 & -0.247 \\ -0.226 & 0.134 & 0.055 \\ -0.247 & 0.055 & 0.151 \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} -0.219 \\ -0.007 \\ 0.356 \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix}$$

We compute $\mathbf{V} = \mathbf{S}\mathbf{K}^{-1}$:

$$\mathbf{V} = \begin{bmatrix} 20 & 10 & 40 \\ 10 & 30 & 25 \end{bmatrix} \begin{bmatrix} 0.767 & -0.226 & -0.247 \\ -0.226 & 0.134 & 0.055 \\ -0.247 & 0.055 & 0.151 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 3.219 & -0.993 & 1.644 \\ -5.274 & 3.116 & 2.945 \end{bmatrix}$$

We then compute $-\left[\frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right]$. As we did this computation in the previous example, we won't show it here. Combining,

$$\frac{\partial \sigma}{\partial x_1} = -\mathbf{V}\left[\frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right]$$

$$\frac{\partial \sigma}{\partial x_1} = -\begin{bmatrix} 3.219 & -0.993 & 1.644 \\ -5.274 & 3.116 & 2.945 \end{bmatrix}\begin{bmatrix} -1.110 \\ -0.438 \\ 0 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_1} = \begin{bmatrix} 3.137 \\ -4.486 \end{bmatrix}$$

We now calculate the derivatives with respect to x_2:

$$\frac{\partial \sigma}{\partial x_2} = -\begin{bmatrix} 3.219 & -0.993 & 1.644 \\ -5.274 & 3.116 & 2.945 \end{bmatrix}\begin{bmatrix} 1.055 \\ -0.493 \\ -0.658 \end{bmatrix}$$

$$\frac{\partial \sigma}{\partial x_2} = \begin{bmatrix} -2.805 \\ 9.036 \end{bmatrix}$$

4.4.4 Comparison of Direct Method and Adjoint Method

The direct method is more efficient when we have more functions than variables; the adjoint method is more efficient when we have more variables than functions. One case where the adjoint method might be preferred is a shape optimization problem, where we may have many design variables (the shape variables) and fewer functions (objective and constraints).

To illustrate this difference in efficiency, suppose we have a problem where

number of variables (n) = 5
number of functions (m) = 10
number of state variables (p) = 20

For the direct method, we would need to compute the following n times:

$$\frac{\partial \sigma}{\partial x_i} = \mathbf{S}\ \mathbf{K}^{-1}\left[\frac{\partial \mathbf{f}}{\partial x_i} - \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u}\right]$$

If we just keep track of the number of multiplies, and we don't include the computation to construct $\left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u}\right]$, which is used by both methods, we have the following matrices being multiplied together, where just their sizes are shown. (Recall that for matrix multiplication the number of multiplies is equal to the outer dimension of the first matrix times the inner dimension of either matrix times the outer dimension of the second matrix.)

$$\frac{\partial \sigma}{\partial x_i} = [10x20]\underbrace{[20x20][20x1]}_{400\,multiplies}$$

$$\frac{\partial \sigma}{\partial x_i} = \underbrace{[10x20][20x1]}_{200\,multiplies}$$

= 600 multiplies per variable times 5 variables

= 3000 total multiplies

For the adjoint method, we would first compute,

$$\mathbf{v}_j^T = \mathbf{s}_j^T\ \mathbf{K}^{-1}$$

$$\mathbf{v}_j^T = \underbrace{[1x20][20x20]}_{400\,multiplies}$$

= 400 multiplies for each function time ten functions

= 4000 multiplies

Then for each of these **v** vectors we need to calculate the dot product below five times, once for each variable:

$$\frac{\partial \sigma_j}{\partial x_i} = \mathbf{v}_j^T\left[\frac{\partial \mathbf{f}}{\partial x_i} - \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u}\right]$$

$$\frac{\partial \sigma_j}{\partial x_i} = \underbrace{[1x20][20x1]}_{20\,multiplies}$$

= 100 multiplies per function (for all five variables) times ten functions

= 4000 + 1000

= 5000 total multiplies

Thus we see for this case, where $m > n$, the direct method is more efficient. If we now reverse the problem so we have $n = 10$, $m = 5$ (more variables than functions) and p kept the same at 20, we find the adjoint method requires 3000 multiplies, and the direct method requires 5000 multiplies.

Notice how inexpensive it is to add derivatives of additional variables to the adjoint method. Adding another variable only requires the calculation of a dot product. In the above example, this would add 20 multiplies to the 5000 we already have. Indeed, 80% of the computation of the example is associated with creating the **v** vectors and these are not a function of the number of variables.

4.4.5 The Total Derivative

4.4.5.1 Development

In this section we will extend the ideas of the previous sections a little further to introduce the concept of a total derivative.

As mentioned in Section 4.4, it is often the case that as part of computing analysis functions we need to drive a set of equations to zero. For example, in our structural analysis problem, we could write (4.20) as,

$$\mathbf{Ku} - \mathbf{f} = \mathbf{0} \tag{4.42}$$

Recall that the vector **u** represents the state variables. In general, we will refer to equations such as (4.42) as the *residual* equations and we solve for the state variables to drive the residuals to zero. We are interested in the derivatives of the independent variables, **x,** that take into account the presence of state variables.

To keep things simple, we will consider that we have only one function, *f,* of interest. We will write,

$$f = f(\mathbf{x}, \mathbf{u}(\mathbf{x})) \tag{4.43}$$

where the notation **u(x)** acknowledges that the state variables are implicit functions of the independent variables. That is, as the independent variables change, the state variables must change as well to keep the residuals at zero.

Differentiating (4.43),

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} + \sum \frac{\partial f}{\partial u_j}\frac{du_j}{dx_i} = \frac{\partial f}{\partial x_i} + \nabla f(\mathbf{u})^T \frac{d\mathbf{u}}{dx_i} \tag{4.44}$$

The vector of residuals will likewise include **u** (as before, a vector of length p) as a function of **x**:

$$\mathbf{r}(\mathbf{x}, \mathbf{u}(\mathbf{x})) = 0 \tag{4.45}$$

Taking the derivative with respect to x_i gives,

$$\frac{d\mathbf{r}}{dx_i} = \frac{\partial \mathbf{r}}{\partial x_i} + \frac{\partial \mathbf{r}}{\partial \mathbf{u}} \frac{d\mathbf{u}}{dx_i} \tag{4.46}$$

where $\frac{\partial \mathbf{r}}{\partial \mathbf{u}}$ is a square matrix, i.e.,

$$\frac{\partial \mathbf{r}}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial r_1}{\partial u_1} & \frac{\partial r_1}{\partial u_2} & \cdots & \frac{\partial r_1}{\partial u_p} \\ \vdots & \vdots & & \vdots \\ \frac{\partial r_p}{\partial u_1} & \frac{\partial r_p}{\partial u_2} & \cdots & \frac{\partial r_p}{\partial u_p} \end{bmatrix} \tag{4.47}$$

We note in (4.46) that $\frac{d\mathbf{r}}{dx_i} = 0$ because the residual equations should be zero everywhere, and this implies the derivatives are also zero. (This is perhaps not obvious, and it is certainly not the case that just because a function is zero at some point, its derivatives are zero. This condition is a result of the functions being zero *everywhere*.) Thus we can write,

$$\frac{d\mathbf{u}}{dx_i} = -\left[\frac{\partial \mathbf{r}}{\partial \mathbf{u}}\right]^{-1} \frac{\partial \mathbf{r}}{\partial x_i} \tag{4.48}$$

Substituting into (4.44) gives the *total derivative*,

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} - \nabla f(\mathbf{u})^T \left[\frac{\partial \mathbf{r}}{\partial \mathbf{u}}\right]^{-1} \frac{\partial \mathbf{r}}{\partial x_i} \tag{4.49}$$

What is the meaning of the total derivative? *The total derivative takes into account the change in the state variables required to keep the residuals at zero*.

4.4.5.2 Structural Example

Let's apply (4.49) to our structural example. We have,

$$f = \sigma_1$$
$$\mathbf{r} = \mathbf{K}\mathbf{u} - \mathbf{f} = \mathbf{0}$$

$$\sigma_1 = \mathbf{s}_1^T \mathbf{u}$$

We also note,

$$\frac{\partial \mathbf{r}}{\partial x_i} = \left[\frac{\partial \mathbf{K}}{\partial x_i}\mathbf{u} - \frac{\partial \mathbf{f}}{\partial x_i} \right]; \qquad \frac{\partial \mathbf{r}}{\partial \mathbf{u}} = \mathbf{K}; \qquad \nabla f(\mathbf{u})^T = \mathbf{s}_1^T$$

Also $\frac{\partial f}{\partial x_i} = 0$ since we assume **S** is not a function of **x**. Putting these into (4.49) gives,

$$\frac{\partial \sigma_1}{\partial x_1} = \mathbf{s}_1^T \ \mathbf{K}^{-1} \left[\frac{\partial \mathbf{f}}{\partial x_1} - \frac{\partial \mathbf{K}}{\partial x_1}\mathbf{u} \right]$$

which is the same as (4.37).

4.4.5.3 The Total Derivative: Direct vs. Adjoint

We can view the direct or adjoint methods as a different way to group terms in (4.49). If we group terms as,

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} - \nabla f(\mathbf{u})^T \overbrace{\left[\frac{\partial \mathbf{r}}{\partial \mathbf{u}} \right]^{-1} \frac{\partial \mathbf{r}}{\partial x_i}}^{\phi} \tag{4.50}$$

Then we have,

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} - \nabla f(\mathbf{u})^T \phi \tag{4.51}$$

where we can view ϕ as a solution to the following set of equations:

$$\frac{\partial \mathbf{r}}{\partial \mathbf{u}} \phi = \frac{\partial \mathbf{r}}{\partial x_i} \tag{4.52}$$

and we are using the direct method for solution. If we group terms as,

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} - \underbrace{\nabla f(\mathbf{u})^T \left[\frac{\partial \mathbf{r}}{\partial \mathbf{u}} \right]^{-1}}_{\psi} \frac{\partial \mathbf{r}}{\partial x_i} \tag{4.53}$$

Equation (4.49) becomes,

$$\frac{df}{dx_i} = \frac{\partial f}{\partial x_i} - \psi \frac{\partial \mathbf{r}}{\partial x_i} \tag{4.54}$$

where ψ is the solution to the following adjoint equations,

$$\left[\frac{\partial \mathbf{r}}{\partial \mathbf{u}}\right]^T \psi = \nabla f(\mathbf{u}) \tag{4.55}$$

and we are using the adjoint method to obtain derivatives.

4.5 Algorithmic Differentiation

4.5.1 Description

Algorithmic or *Automatic Differentiation* (AD) is based on applying the chain rule for differentiation to a computer program. This approach involves no approximation error and is relatively easy to implement in a computer language supporting object-oriented programming; however, it does require that the analysis code be modified and executed in an environment which supports AD. It can also involve significant computation.

A good overview of this subject is given by Naumann [19]. We also direct the interested reader to the website www.autodiff.org which has open source software packages for C/C++, Fortran, Python, Java, Julia, and MATLAB among others.

There are two approaches to AD, which are somewhat similar to the direct and adjoint methods in computing analytic derivatives of simultaneous equations. These are the forward and reverse methods. We will focus here on the forward method, which is the most common and easiest to understand.

In a computer environment, we often do a series of calculations where the result of one calculation feeds into another. As we calculate these intermediate values, we could also calculate derivatives that "feed forward" into other calculations.

For example, suppose we wish to evaluate,

$$f = x^2 \exp(3x) \qquad \text{at} \quad x = 2$$

using a computer program. We could compute the function in one line, but we could also break it up into several steps:

```
x = 2;
f1 = 3*x;
f2 = exp(f1);
f3 = x^2;
f4 = f3*f2;
```

The table below shows the sequence of values for both functions and derivatives ($x = 2$), where we use the notation $f' = \frac{df}{dx}$:

Function	Value	Derivative	Value
x	2	$x' = 1$	1
$f_1 = 3x$	6	$f_1' = 3x'$	3
$f_2 = \exp(f_1)$	403.43	$f_2' = \exp(f_1) f_1'$	1210.29
$f_3 = x^2$	4	$f_3' = 2xx'$	4
$f_4 = f_3 * f_2$	1613.7	$f_4' = f_3 f_2' + f_2 f_3'$	6454.9

We see that not only can the calculation of current function values be based on prior calculations, but the calculation of current derivatives can be based on prior derivative calculations. With AD, we compute derivatives automatically as we compute functions.

To reinforce the overall concept, we quote from Martins et al. [16], "AD applies the chain rule for every single line in the [computer] program….In it simplest form, each function in [a] sequence depends only on the inputs and the functions that have been computed earlier in the sequence, as expressed in the functional dependence." There is a caveat, however: "We assume that all of the loops in the program are *unrolled*, and therefore no variables are overwritten and each variable only depends on earlier variables in the sequence. Later, when we explain how AD is implemented, it will become clear that this assumption is not restrictive…"

In the next section, we will show how an AD environment can be implemented in MATLAB.

4.5.2 An AD Scheme for MATLAB

To implement a relatively simple, yet quite powerful scheme for AD in MATLAB, we will take advantage of object-oriented programming and define x to be an object that has both a value and a derivative. As we perform operations on x such as addition, division, exponentiation, etc., we will redefine these operators using operator overloading so they do both value and derivative computations. We begin with an overview of the definition of a class called "`valder`" (from value/derivative) which we will create. All of the programming which follows in this section is taken from Neidinger [20].

```
classdef valder
   properties
      val  %function value
      der  %derivative value or gradient vector
   end
   methods
      function obj = valder(a,b)...
      function vec = double(obj)...
      function h = plus(u,v)...
      function h = uminus(u)...
      function h = minus(u,v)...
```

```
      function h = mtimes(u,v)...
      function h = mrdivide(u,v)...
      function h = mpower(u,v)...
      function h = exp(u)...
      function h = log(u)...
      function h = sqrt(u)...
      function h = sin(u)...
      etc.
   end
end
```

For clarity individual functions have not been displayed. None of the functions is longer than seven lines, however, and some are only one line. For example, here is the function for `exp(u)`:

```
      function h = exp(u)
         %VALDER/EXP overloads exp of a valder object argument
         h = valder(exp(u.val), exp(u.val)*u.der);
      end
```

A complete listing of the `valder` class with functions is given in the Appendix to this chapter.

Now suppose we wish to evaluate our example function of $f = x^2 \exp(3x)$ at $x = 2$,

Below is one version of MATLAB code to do this. We begin by making x a `valder` object. The first argument is the value; the second argument is the derivative (the derivative of x is just 1). Since x is a `valder` object, all the operations after the first line are computed with the functions from the class definition.

Code:

```
>> x = valder(2,1);
>> f1 = 3*x;
>> f2 = exp(f1);
>> f3 = x^2;
>> f4 = f3*f2
```

Results:

```
f4 =

  valder with properties:

    val: 1.6137e+03
    der: 6.4549e+03
```

Or we could do everything in one line:

```
>> x =valder(2,1);
>> f = x^2 * exp(3 * x)

f =

  valder with properties:

    val: 1.6137e+03
```

```
der: 6.4549e+03
```

We can understand better what is happening relative to operator overloading by referring to MATLAB documentation. A listing of some of the basic operators gives:

```
plus          Addition
uplus         Unary plus
minus         Subtraction
uminus        Unary minus
times         Element-wise multiplication
rdivide       Right array division
ldivide       Left array division
power         Element-wise power
mtimes        Matrix Multiplication
mrdivide      Solve systems of linear equations xA = B for x
mldivide      Solve systems of linear equations Ax = B for x
mpower        Matrix power
Etc.
```

Futher explanation of one of these (we will choose "`plus`") as given in the documentation is helpful:

```
plus, +

Addition

Syntax
C = A + B
C = plus(A,B)

Description
C = A + B adds arrays A and B and returns the result in C.

C = plus(A,B) is an alternate way to execute A + B, but is rarely used. It enables
operator overloading for classes.
```

The last sentence is key: "`C = plus(A,B) is an alternate way to execute A + B, but is rarely used. It enables operator overloading for classes.`"

Thus, if MATLAB is operating on objects of class `valder`, and it encounters a plus sign ("+"), it executes the addition in accordance with the definition of "plus" as given in `valder`:

```
function h = plus(u,v)
   %VALDER/PLUS overloads addition + with at least one valder object argument
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u+v.val, v.der);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(u.val+v, u.der);
   else
      h = valder(u.val+v.val, u.der+v.der);
   end
end
```

We can see by this code that the `valder` definition of "+" not only executes addition for the function values, but also computes the derivative when two functions are added together (which results in adding the derivative values together). In this manner both function values and derivatives are carried along as the program executes.

4.5.3 Extending the Program to Multiple Variables

It may not be obvious (at least it wasn't to me!) that the above code can be extended to the situation of multiple variables *with no changes*. If we have multiple variables, meaning we have gradient vectors instead of just single derivatives, we just supply a vector of derivative values when we declare something to be the `valder` class. For example,

```
X1 = valder(2,[1,0]);
X2 = valder(3,[0,1]);
```

The first statement makes `X1` an object of `valder` class with a beginning value of 2 and a partial derivative with respect to `X1` of 1 and with respect to `X2` of 0. Similarly for variable `X2:` its starting value is 3; the partial derivatives are 0 and 1.

Let's illustrate the multi-variable case by applying AD to the two-bar truss problem, where we have height and diameter as design variables, and with the usual functions of weight, stress, buckling and deflection.

Code for this follows:

```
%This routine calls the analysis program for the two-bar truss
%which automatically computes derivatives

%Set initial values of variables
x1 = 30;
x2 = 3;

%Call the analysis routine
[Functions, Jacobian] = twobarAD(x1,x2);

%Print values
Functions
Jacobian
```

A listing of the `twobarAD` function follows:

```
function [F, J] = twobarAD(x1, x2)
%This function computes function values and derivatives
%for the two-bar truss using AD as given by Neidinger
%SIAM Review, Vol. 52, No. 3, pp. 543-563

wdth = 60.; thik = 0.15;
dens = 0.3; modu = 30000.; load = 66.;

%make height and diameter design variable objects of class valder
hght = valder(x1,[1,0]);
diam = valder(x2,[0,1]);

% compute intermediate functions
leng = ((wdth/2.)^2 + hght^2)^0.5;
area = pi * diam * thik;
iovera = (diam^2 + thik^2) / 8.;

% compute functions
wght = 2. * dens * area * leng;
strs = load * leng / (2. * area * hght);
buck = (pi^2 * modu * iovera / (leng^2));
buckcon = strs - buck;
```

```
defl = load * leng^3 / (2. * modu * area * hght^2);

F = [wght.val, strs.val, buckcon.val, defl.val];
J = [wght.der; strs.der; buckcon.der; defl.der];
```

When we execute this code (and, of course, have the `valder` class definition in the same folder), we get:

```
Functions =

   35.9874   33.0116 -152.5062    0.0660

Jacobian =

    0.5998   11.9958
   -0.5502  -11.0039
    5.6337 -134.3739
   -0.0011   -0.0220
```

where the rows of the Jacobian matrix are gradient vectors for the individual functions.

The simplicity and power of what has been done here are striking. By adding a little over one hundred lines of code, and taking advantage of object-oriented programming and operator overloading, we have been able, with the addition of a few lines to the analysis routine, to convert our analysis code from giving functions only to computing functions and gradients. And what we have done is quite general and could be applied to other analysis routines.

We should discuss a little bit about accuracy, implementation and efficiency. AD methods are highly accurate—as accurate as the function computation. They require that the analysis software support AD. In terms of efficiency, each partial derivative requires roughly the same amount of computation as computing the original function, so the analysis software can take several times longer to execute.

4.6 Summary Comparison of Methods

In the table below we summarize the pros and cons of the methods we have presented in this chapter.

Method	Generality	Accuracy	Efficiency	Comments
Finite Difference	High	Low to Medium	Medium to Low	Can be used on any differentiable model
Complex step	Medium	High	Medium	Must be able to modify the analysis code
Automatic differentiation	Medium	High	Medium	Must be able to modify the analysis code
Analytic Differentiation of Simultaneous Equations	Medium	High	High	Must be able to modify the analysis code

4.7 Appendix

4.7.1 Listing of Valder.m

The following listing, created by Richard Neidinger [20], contains many, but not all routines for basic operators in order to implement Automatic Differentiation in MATLAB using objects.

```
classdef valder
   % VALDER class implementing Automatic Differentiation by operator overloading.
   % Computes first order derivative or multivariable gradient vectors by
   % starting with a known simple valder such as x=valder(3,1) and
   % propagating it through elementary functions and operators.
   % by Richard D. Neidinger 10/23/08
   properties
      val  %function value
      der  %derivative value or gradient vector
   end
   methods
      function obj = valder(a,b)
         %VALDER class constructor; only the bottom case is needed.
         if nargin == 0 %never intended for use.
            obj.val = [];
            obj.der = [];
         elseif nargin == 1 %c=valder(a) for constant w/ derivative 0.
            obj.val = a;
            obj.der = 0;
         else
            obj.val = a; %given function value
            obj.der = b; %given derivative value or gradient vector
         end
      end
      function vec = double(obj)
         %VALDER/DOUBLE Convert valder object to vector of doubles.
         vec = [ obj.val, obj.der ];
      end
      function h = plus(u,v)
         %VALDER/PLUS overloads addition + with at least one valder object argument
```

```
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u+v.val, v.der);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(u.val+v, u.der);
   else
      h = valder(u.val+v.val, u.der+v.der);
   end
end
function h = uminus(u)
   %VALDER/UMINUS overloads negation with at least 1 valder object argument
   h = valder(-u.val, -u.der);
end
function h = minus(u,v)
   %VALDER/MINUS overloads subtraction with at least 1 valder object argument
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u-v.val, -v.der);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(u.val-v, u.der);
   else
      h = valder(u.val-v.val, u.der-v.der);
   end
end
function h = mtimes(u,v)
   %VALDER/MTIMES overloads mult.(*) with at least one valder object argument
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u*v.val, u*v.der);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(v*u.val, v*u.der);
   else
      h = valder(u.val*v.val, u.der*v.val + u.val*v.der);
   end
end
function h = mrdivide(u,v)
   %VALDER/MRDIVIDE overloads div.(/)with at least one valder object argument
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u/v.val, (-u*v.der)/(v.val)^2);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(u.val/v, u.der/v);
   else
      h = valder(u.val/v.val, (u.der*v.val-u.val*v.der)/(v.val)^2);
   end
end
function h = mpower(u,v)
   %VALDER/MPOWER overloads power ^ with at least one valder object argument
   if ~isa(u,'valder') %u is a scalar (u is not of class 'valder')
      h = valder(u^v.val, u^v.val*log(u)*v.der);
   elseif ~isa(v,'valder') %v is a scalar (v is not of class 'valder')
      h = valder(u.val^v, v*u.val^(v-1)*u.der);
   else
      h = exp(v*log(u)); %call overloaded log, * and exp
   end
end
function h = exp(u)
   %VALDER/EXP overloads exp of a valder object argument
   h = valder(exp(u.val), exp(u.val)*u.der);
end
function h = log(u)
   %VALDER/LOG overloads natural logarithm of a valder object argument
   h = valder(log(u.val), (1/u.val)*u.der);
end
function h = sqrt(u)
   %VALDER/SQRT overloads square root of a valder object argument
   h = valder(sqrt(u.val), u.der/(2*sqrt(u.val)));
end
function h = sin(u)
```

```
         %VALDER/SIN overloads sine with a valder object argument
         h = valder(sin(u.val), cos(u.val)*u.der);
      end
      function h = cos(u)
         %VALDER/COS overloads cosine of a valder object argument
         h = valder(cos(u.val), -sin(u.val)*u.der);
      end
      function h = tan(u)
         %VALDER/TAN overloads tangent of a valder object argument
         h = valder(tan(u.val), (sec(u.val))^2*u.der);
      end
      function h = asin(u)
         %VALDER/ASIN overloads arcsine of a valder object argument
         h = valder(asin(u.val), u.der/sqrt(1-u.val^2));
      end
      function h = atan(u)
         %VALDER/ATAN overloads arctangent of a valder object argument
         h = valder(atan(u.val), u.der/(1+u.val^2));
      end
   end
end
```