

(the single real root when  $a, b, c$  are real) and

$$\begin{aligned}x_2 &= -\frac{1}{2}(A + B) - \frac{a}{3} + i\frac{\sqrt{3}}{2}(A - B) \\x_3 &= -\frac{1}{2}(A + B) - \frac{a}{3} - i\frac{\sqrt{3}}{2}(A - B)\end{aligned}\tag{5.6.18}$$

(in that same case, a complex conjugate pair). Equations (5.6.13)–(5.6.16) are arranged both to minimize roundoff error, and also (as pointed out by A.J. Glassman) to ensure that no choice of branch for the complex cube root can result in the spurious loss of a distinct root.

If you need to solve many cubic equations with only slightly different coefficients, it is more efficient to use Newton's method (§9.4).

#### CITED REFERENCES AND FURTHER READING:

- Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.
- Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.
- McKelvey, J.P. 1984, *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

## 5.7 Numerical Derivatives

Imagine that you have a procedure which computes a function  $f(x)$ , and now you want to compute its derivative  $f'(x)$ . Easy, right? The definition of the derivative, the limit as  $h \rightarrow 0$  of

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}\tag{5.7.1}$$

practically suggests the program: Pick a small value  $h$ ; evaluate  $f(x+h)$ ; you probably have  $f(x)$  already evaluated, but if not, do it too; finally apply equation (5.7.1). What more needs to be said?

Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. Applied properly, it can be the right way to compute a derivative only when the function  $f$  is *fiercely* expensive to compute, when you already have invested in computing  $f(x)$ , and when, therefore, you want to get the derivative in no more than a single additional function evaluation. In such a situation, the remaining issue is to choose  $h$  properly, an issue we now discuss:

There are two sources of error in equation (5.7.1), truncation error and roundoff error. The truncation error comes from higher terms in the Taylor series expansion,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots\tag{5.7.2}$$

whence

$$\frac{f(x+h) - f(x)}{h} = f' + \frac{1}{2}hf'' + \dots\tag{5.7.3}$$

The roundoff error has various contributions. First there is roundoff error in  $h$ : Suppose, by way of an example, that you are at a point  $x = 10.3$  and you blindly choose  $h = 0.0001$ . Neither  $x = 10.3$  nor  $x + h = 10.30001$  is a number with an exact representation in binary; each is therefore represented with some fractional error characteristic of the machine's floating-point format,  $\epsilon_m$ , whose value in single precision may be  $\sim 10^{-7}$ . The error in the *effective* value of  $h$ , namely the difference between  $x + h$  and  $x$  as represented in the machine, is therefore on the order of  $\epsilon_m x$ , which implies a fractional error in  $h$  of order  $\sim \epsilon_m x/h \sim 10^{-2}$ ! By equation (5.7.1) this immediately implies at least the same large fractional error in the derivative.

We arrive at Lesson 1: Always choose  $h$  so that  $x + h$  and  $x$  differ by an exactly representable number. This can usually be accomplished by the program steps

$$\begin{aligned} \text{temp} &= x + h \\ h &= \text{temp} - x \end{aligned} \quad (5.7.4)$$

Some optimizing compilers, and some computers whose floating-point chips have higher internal accuracy than is stored externally, can foil this trick; if so, it is usually enough to call a dummy subroutine `donothing(temp)` between the two equations (5.7.4). This forces `temp` into and out of addressable memory.

With  $h$  an “exact” number, the roundoff error in equation (5.7.1) is  $e_r \sim \epsilon_f |f(x)/h|$ . Here  $\epsilon_f$  is the fractional accuracy with which  $f$  is computed; for a simple function this may be comparable to the machine accuracy,  $\epsilon_f \approx \epsilon_m$ , but for a complicated calculation with additional sources of inaccuracy it may be larger. The truncation error in equation (5.7.3) is on the order of  $e_t \sim |h f''(x)|$ . Varying  $h$  to minimize the sum  $e_r + e_t$  gives the optimal choice of  $h$ ,

$$h \sim \sqrt{\frac{\epsilon_f f}{f''}} \approx \sqrt{\epsilon_f} x_c \quad (5.7.5)$$

where  $x_c \equiv (f/f'')^{1/2}$  is the “curvature scale” of the function  $f$ , or “characteristic scale” over which it changes. In the absence of any other information, one often assumes  $x_c = x$  (except near  $x = 0$  where some other estimate of the typical  $x$  scale should be used).

With the choice of equation (5.7.5), the fractional accuracy of the computed derivative is

$$(e_r + e_t)/|f'| \sim \sqrt{\epsilon_f} (f f'' / f'^2)^{1/2} \sim \sqrt{\epsilon_f} \quad (5.7.6)$$

Here the last order-of-magnitude equality assumes that  $f$ ,  $f'$ , and  $f''$  all share the same characteristic length scale, usually the case. One sees that the simple finite-difference equation (5.7.1) gives *at best* only the square root of the machine accuracy  $\epsilon_m$ .

If you can afford two function evaluations for each derivative calculation, then it is significantly better to use the symmetrized form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (5.7.7)$$

In this case, by equation (5.7.2), the truncation error is  $e_t \sim h^2 f'''$ . The roundoff error  $e_r$  is about the same as before. The optimal choice of  $h$ , by a short calculation analogous to the one above, is now

$$h \sim \left( \frac{\epsilon_f f}{f'''} \right)^{1/3} \sim (\epsilon_f)^{1/3} x_c \quad (5.7.8)$$

and the fractional error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3} / f' \sim (\epsilon_f)^{2/3} \quad (5.7.9)$$

which will typically be an order of magnitude (single precision) or two orders of magnitude (double precision) *better* than equation (5.7.6). We have arrived at Lesson 2: Choose  $h$  to be *the correct* power of  $\epsilon_f$  or  $\epsilon_m$  times a characteristic scale  $x_c$ .

You can easily derive the correct powers for other cases [1]. For a function of two dimensions, for example, and the mixed derivative formula

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{[f(x+h, y+h) - f(x+h, y-h)] - [f(x-h, y+h) - f(x-h, y-h)]}{4h^2} \quad (5.7.10)$$

the correct scaling is  $h \sim \epsilon_f^{1/3} x_c$ .

It is disappointing, certainly, that no simple finite-difference formula like equation (5.7.1) or (5.7.7) gives an accuracy comparable to the machine accuracy  $\epsilon_m$ , or even the lower accuracy to which  $f$  is evaluated,  $\epsilon_f$ . Are there no better methods?

Yes, there are. All, however, involve exploration of the function's behavior over scales comparable to  $x_c$ , plus some assumption of smoothness, or analyticity, so that the high-order terms in a Taylor expansion like equation (5.7.2) have some meaning. Such methods also involve multiple evaluations of the function  $f$ , so their increased accuracy must be weighed against increased cost.

The general idea of "Richardson's deferred approach to the limit" is particularly attractive. For numerical integrals, that idea leads to so-called Romberg integration (for review, see §4.3). For derivatives, one seeks to extrapolate, to  $h \rightarrow 0$ , the result of finite-difference calculations with smaller and smaller finite values of  $h$ . By the use of Neville's algorithm (§3.1), one uses each new finite-difference calculation to produce both an extrapolation of higher order, and also extrapolations of previous, lower, orders but with smaller scales  $h$ . Ridders [2] has given a nice implementation of this idea; the following program, `dfridr`, is based on his algorithm, modified by an improved termination criterion. Input to the routine is a function  $f$  (called `func`), a position  $x$ , and a *largest* stepsize `h` (more analogous to what we have called  $x_c$  above than to what we have called  $h$ ). Output is the returned value of the derivative, and an estimate of its error, `err`.

```
FUNCTION dfridr(func,x,h,err)
  INTEGER NTAB
  REAL dfridr,err,h,x,func,CON,CON2,BIG,SAFE
  PARAMETER (CON=1.4,CON2=CON*CON,BIG=1.E30,NTAB=10,SAFE=2.)
  EXTERNAL func
  C USES func
```

Returns the derivative of a function `func` at a point `x` by Ridders' method of polynomial extrapolation. The value `h` is input as an estimated initial stepsize; it need not be small,

but rather should be an increment in  $x$  over which `func` changes *substantially*. An estimate of the error in the derivative is returned as `err`.

Parameters: `Stepsize` is decreased by `CON` at each iteration. Max size of tableau is set by `NTAB`. Return when error is `SAFE` worse than the best so far.

```

INTEGER i,j
REAL errt,fac,hh,a(NTAB,NTAB)
if(h.eq.0.) pause 'h must be nonzero in dfridr'
hh=h
a(1,1)=(func(x+hh)-func(x-hh))/(2.0*hh)
err=BIG
do 12 i=2,NTAB      Successive columns in the Neville tableau will go to smaller
                   hh=hh/CON      stepsizes and higher orders of extrapolation.
                   a(1,i)=(func(x+hh)-func(x-hh))/(2.0*hh)      Try new, smaller stepsize.
                   fac=CON2
                   do 11 j=2,i      Compute extrapolations of various orders, requiring no new
                   a(j,i)=(a(j-1,i)*fac-a(j-1,i-1))/(fac-1.)      function evaluations.
                   fac=CON2*fac
                   errt=max(abs(a(j,i)-a(j-1,i)),abs(a(j,i)-a(j-1,i-1)))
                   The error strategy is to compare each new extrapolation to one order lower, both at
                   the present stepsize and the previous one.
                   if (errt.le.err) then      If error is decreased, save the improved answer.
                       err=errt
                       dfridr=a(j,i)
                   endif
                   enddo 11
                   if(abs(a(i,i)-a(i-1,i-1)).ge.SAFE*err)return
                   If higher order is worse by a significant factor SAFE, then quit early.
enddo 12
return
END

```

In `dfridr`, the number of evaluations of `func` is typically 6 to 12, but is allowed to be as great as  $2 \times \text{NTAB}$ . As a function of input  $h$ , it is typical for the accuracy to get *better* as  $h$  is made larger, until a sudden point is reached where nonsensical extrapolation produces early return with a large error. You should therefore choose a fairly large value for  $h$ , but monitor the returned value `err`, decreasing  $h$  if it is not small. For functions whose characteristic  $x$  scale is of order unity, we typically take  $h$  to be a few tenths.

Besides Ridders' method, there are other possible techniques. If your function is fairly smooth, and you know that you will want to evaluate its derivative many times at arbitrary points in some interval, then it makes sense to construct a Chebyshev polynomial approximation to the function in that interval, and to evaluate the derivative directly from the resulting Chebyshev coefficients. This method is described in §§5.8–5.9, following.

Another technique applies when the function consists of data that is tabulated at equally spaced intervals, and perhaps also noisy. One might then want, at each point, to least-squares *fit* a polynomial of some degree  $M$ , using an additional number  $n_L$  of points to the left and some number  $n_R$  of points to the right of each desired  $x$  value. The estimated derivative is then the derivative of the resulting fitted polynomial. A very efficient way to do this construction is via Savitzky-Golay smoothing filters, which will be discussed later, in §14.8. There we will give a routine for getting filter coefficients that not only construct the fitting polynomial but, in the accumulation of a single sum of data points times filter coefficients, evaluate it as well. In fact, the routine given, `savgol`, has an argument `ld` that determines which derivative of the fitted polynomial is evaluated. For the first derivative, the

appropriate setting is  $1d=1$ , and the value of the derivative is the accumulated sum divided by the sampling interval  $h$ .

CITED REFERENCES AND FURTHER READING:

- Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall), §§5.4–5.6. [1]  
 Ridders, C.J.F. 1982, *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76. [2]

## 5.8 Chebyshev Approximation

The Chebyshev polynomial of degree  $n$  is denoted  $T_n(x)$ , and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \quad (5.8.1)$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however (5.8.1) can be combined with trigonometric identities to yield explicit expressions for  $T_n(x)$  (see Figure 5.8.1),

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\dots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \quad (5.8.2)$$

(There also exist inverse formulas for the powers of  $x$  in terms of the  $T_n$ 's — see equations 5.11.2–5.11.3.)

The Chebyshev polynomials are orthogonal in the interval  $[-1, 1]$  over a weight  $(1 - x^2)^{-1/2}$ . In particular,

$$\int_{-1}^1 \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \quad (5.8.3)$$

The polynomial  $T_n(x)$  has  $n$  zeros in the interval  $[-1, 1]$ , and they are located at the points

$$x = \cos\left(\frac{\pi(k - \frac{1}{2})}{n}\right) \quad k = 1, 2, \dots, n \quad (5.8.4)$$