Chapter 3. Interpolation and Extrapolation

3.0 Introduction

We sometimes know the value of a function $f(x)$ at a set of points $x_1, x_2, \ldots, x_N$ (say, with $x_1 < \ldots < x_N$), but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i's are equally spaced, but not necessarily.

The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i. If the desired x is in between the largest and smallest of the x_i's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former stock-market analysts can attest).

Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions which might arise in practice. By far most common among the functional forms used are polynomials (§3.1). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.2). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapters 12 and 13.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: If we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function f at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function f at *any* desired points for the purpose of developing your approximation. We deal with function approximation in Chapter 5.

One can easily find pathological functions that make a mockery of any interpolation scheme. Consider, for example, the function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln\left[(\pi - x)^2\right] + 1 \tag{3.0.1}$$

which is well-behaved everywhere except at $x = \pi$, very mildly singular at $x = \pi$, and otherwise takes on all positive and negative values. Any interpolation based on the values $x = 3.13, 3.14, 3.15, 3.16$, will assuredly get a very wrong answer for the value $x = 3.1416$, even though a graph plotting those five points looks really quite smooth! (Try it on your calculator.)

Because pathologies can lurk anywhere, it is highly desirable that an interpolation and extrapolation routine should provide an estimate of its own error. Such an error estimate can never be foolproof, of course. We could have a function that, for reasons known only to its maker, takes off wildly and unexpectedly between two tabulated points. Interpolation always presumes some degree of smoothness for the function interpolated, but within this framework of presumption, deviations from smoothness can be detected.

Conceptually, the interpolation process has two stages: (1) Fit an interpolating function to the data points provided. (2) Evaluate that interpolating function at the target point x.

However, this two-stage method is generally not the best way to proceed in practice. Typically it is computationally less efficient, and more susceptible to roundoff error, than methods which construct a functional estimate $f(x)$ directly from the N tabulated values every time one is desired. Most practical schemes start at a nearby point $f(x_i)$, then add a sequence of (hopefully) decreasing corrections, as information from other $f(x_i)$'s is incorporated. The procedure typically takes $O(N^2)$ operations. If everything is well behaved, the last correction will be the smallest, and it can be used as an informal (though not rigorous) bound on the error.

In the case of polynomial interpolation, it sometimes does happen that the coefficients of the interpolating polynomial are of interest, even though their use in *evaluating* the interpolating function should be frowned on. We deal with this eventuality in §3.5.

Local interpolation, using a finite number of "nearest-neighbor" points, gives interpolated values $f(x)$ that do not, in general, have continuous first or higher derivatives. That happens because, as x crosses the tabulated values x_i, the interpolation scheme switches which tabulated points are the "local" ones. (If such a switch is allowed to occur anywhere *else*, then there will be a discontinuity in the interpolated function itself at that point. Bad idea!)

In situations where continuity of derivatives is a concern, one must use the "stiffer" interpolation provided by a so-called *spline* function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined "slightly" nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§3.3) are the most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

The number of points (minus one) used in an interpolation scheme is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. If the added points are distant from the point of interest x, the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. This oscillation may have no relation at all to the behavior of the "true" function (see Figure 3.0.1). Of course, adding points *close* to the desired point usually does help,

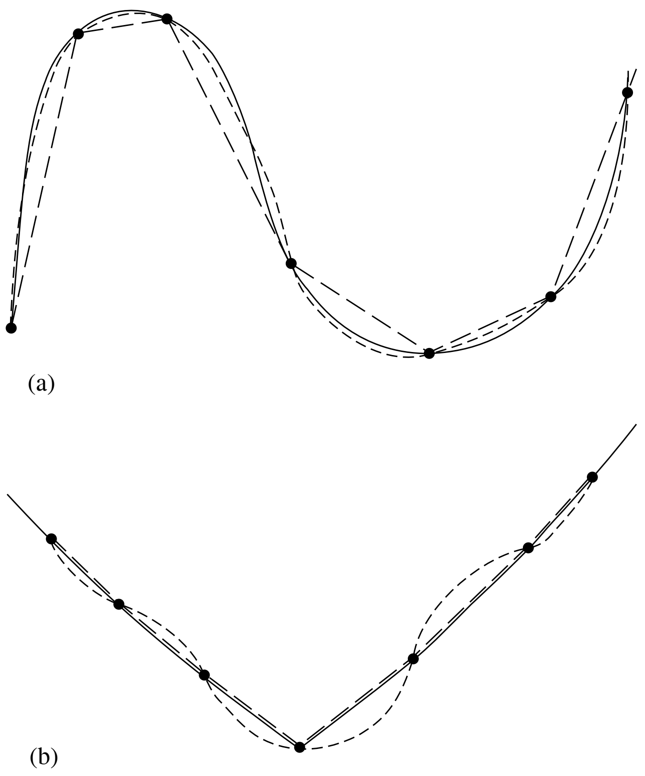


Figure 3.0.1. (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too "stiff," than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

but a finer mesh implies a larger table of values, not always available.

Unless there is solid evidence that the interpolating function is close in form to the true function f, it is a good idea to be cautious about high-order interpolation. We enthusiastically endorse interpolations with 3 or 4 points, we are perhaps tolerant of 5 or 6; but we rarely go higher than that unless there is quite rigorous monitoring of estimated errors.

When your table of values contains many more points than the desirable order of interpolation, you must begin each interpolation with a search for the right "local" place in the table. While not strictly a part of the subject of interpolation, this task is important enough (and often enough botched) that we devote §3.4 to its discussion.

The routines given for interpolation are also routines for extrapolation. An important application, in Chapter 16, is their use in the integration of ordinary differential equations. There, considerable care *is* taken with the monitoring of errors. Otherwise, the dangers of extrapolation cannot be overemphasized: An interpolating function, which is perforce an extrapolating function, will typically go berserk when the argument x is outside the range of tabulated values by more than the typical spacing of tabulated points.

Interpolation can be done in more than one dimension, e.g., for a function

$f(x, y, z)$. Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations. We discuss this in §3.6.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 2.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 3.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 4.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 3.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 6.

3.1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N - 1$ through the N points $y_1 = f(x_1), y_2 = f(x_2), \dots, y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

$$\begin{aligned} P(x) = {} & \frac{(x - x_2)(x - x_3)...(x - x_N)}{(x_1 - x_2)(x_1 - x_3)...(x_1 - x_N)} y_1 + \frac{(x - x_1)(x - x_3)...(x - x_N)}{(x_2 - x_1)(x_2 - x_3)...(x_2 - x_N)} y_2 \\ & + \cdots + \frac{(x - x_1)(x - x_2)...(x - x_{N-1})}{(x_N - x_1)(x_N - x_2)...(x_N - x_{N-1})} y_N \end{aligned} \tag{3.1.1}$$

There are N terms, each a polynomial of degree $N - 1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i.

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let P_1 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_1, y_1); so $P_1 = y_1$. Likewise define $P_2, P_3, \dots, P_N$. Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2). Likewise $P_{23}, P_{34}, \dots,$ $P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123...N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.

The various P's form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example, with $N = 4$,

$$\begin{array}{lllll} x_1: & y_1 = P_1 & & & \\ & & P_{12} & & \\ x_2: & y_2 = P_2 & & P_{123} & \\ & & P_{23} & & P_{1234} \\ x_3: & y_3 = P_3 & & P_{234} & \\ & & P_{34} & & \\ x_4: & y_4 = P_4 & & & \end{array} \tag{3.1.2}$$

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a "daughter" P and its two "parents,"

$$P_{i(i+1)\dots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\dots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\dots(i+m)}}{x_i - x_{i+m}} \tag{3.1.3}$$

This recurrence works because the two parents already agree at points $x_{i+1} \dots x_{i+m-1}$.

An improvement on the recurrence (3.1.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \dots, N - 1$),

$$\begin{aligned} C_{m,i} &\equiv P_{i\dots(i+m)} - P_{i\dots(i+m-1)} \\ D_{m,i} &\equiv P_{i\dots(i+m)} - P_{(i+1)\dots(i+m)}. \end{aligned} \tag{3.1.4}$$

Then one can easily derive from (3.1.3) the relations

$$\begin{aligned} D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\ C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \end{aligned} \tag{3.1.5}$$

At each level m, the C's and D's are the corrections that make the interpolation one order higher. The final answer $P_{1\dots N}$ is equal to the sum of *any* y_i plus a set of C's and/or D's that form a path through the family tree to the rightmost daughter.

Here is a routine for polynomial interpolation or extrapolation from N input points. Note that the input arrays are assumed to be unit-offset. If you have zero-offset arrays, remember to subtract 1 (see §1.2):

```
#include <math.h>
#include "nrutil.h"

void polint(float xa[], float ya[], int n, float x, float *y, float *dy)
```
Given arrays `xa[1..n]` and `ya[1..n]`, and given a value `x`, this routine returns a value `y`, and an error estimate `dy`. If $P(x)$ is the polynomial of degree $N-1$ such that $P(\mathtt{xa}_i) = \mathtt{ya}_i, i = 1,\dots,\mathtt{n}$, then the returned value $\mathtt{y} = P(\mathtt{x})$.
```
{
    int i,m,ns=1;
    float den,dif,dift,ho,hp,w;
    float *c,*d;

    dif=fabs(x-xa[1]);
    c=vector(1,n);
    d=vector(1,n);
    for (i=1;i<=n;i++) {            Here we find the index ns of the closest table entry,
        if ( (dift=fabs(x-xa[i])) < dif) {
            ns=i;
            dif=dift;
        }
        c[i]=ya[i];                 and initialize the tableau of c's and d's.
        d[i]=ya[i];
    }
    *y=ya[ns--];                    This is the initial approximation to y.
    for (m=1;m<n;m++) {             For each column of the tableau,
        for (i=1;i<=n-m;i++) {      we loop over the current c's and d's and update
            ho=xa[i]-x;                 them.
            hp=xa[i+m]-x;
            w=c[i+1]-d[i];
            if ( (den=ho-hp) == 0.0) nrerror("Error in routine polint");
            This error can occur only if two input xa's are (to within roundoff) identical.
            den=w/den;
            d[i]=hp*den;            Here the c's and d's are updated.
            c[i]=ho*den;
        }
        *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
        After each column in the tableau is completed, we decide which correction, c or d,
        we want to add to our accumulating value of y, i.e., which path to take through the
        tableau—forking up or down. We do this in such a way as to take the most "straight
        line" route through the tableau to its apex, updating ns accordingly to keep track of
        where we are. This route keeps the partial approximations centered (insofar as possible)
        on the target x. The last dy added is thus the error indication.
    }
    free_vector(d,1,n);
    free_vector(c,1,n);
}
```

Quite often you will want to call polint with the dummy arguments xa and ya replaced by actual arrays *with offsets*. For example, the construction polint(&xx[14],&yy[14],4,x,y,dy) performs 4-point interpolation on the tabulated values xx[15..18], yy[15..18]. For more on this, see the end of §3.4.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.1.

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

3.2 Rational Function Interpolation and Extrapolation

Some functions are not well approximated by polynomials, but *are* well approximated by rational functions, that is quotients of polynomials. We denote by $R_{i(i+1)\dots(i+m)}$ a rational function passing through the $m+1$ points $(x_i, y_i)\dots(x_{i+m}, y_{i+m})$. More explicitly, suppose

$$R_{i(i+1)\dots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{3.2.1}$$

Since there are $\mu + \nu + 1$ unknown p's and q's (q_0 being arbitrary), we must have

$$m + 1 = \mu + \nu + 1 \tag{3.2.2}$$

In specifying a rational function interpolating function, you must give the desired order of both the numerator and the denominator.

Rational functions are sometimes superior to polynomials, roughly speaking, because of their ability to model functions with poles, that is, zeros of the denominator of equation (3.2.1). These poles might occur for real values of x, if the function to be interpolated itself has poles. More often, the function $f(x)$ is finite for all finite *real* x, but has an analytic continuation with poles in the complex x-plane. Such poles can themselves ruin a polynomial approximation, even one restricted to real values of x, just as they can ruin the convergence of an infinite power series in x. If you draw a circle in the complex plane around your m tabulated points, then you should not expect polynomial interpolation to be good unless the nearest pole is rather far outside the circle. A rational function approximation, by contrast, will stay "good" as long as it has enough powers of x in its denominator to account for (cancel) any nearby poles.

For the interpolation problem, a rational function is constructed so as to go through a chosen set of tabulated functional values. However, we should also mention in passing that rational function approximations can be used in analytic work. One sometimes constructs a rational function approximation by the criterion that the rational function of equation (3.2.1) itself have a power series expansion that agrees with the first $m + 1$ terms of the power series expansion of the desired function $f(x)$. This is called *Padé approximation*, and is discussed in §5.12.

Bulirsch and Stoer found an algorithm of the Neville type which performs rational function extrapolation on tabulated data. A tableau like that of equation (3.1.2) is constructed column by column, leading to a result and an error estimate. The Bulirsch-Stoer algorithm produces the so-called *diagonal* rational function, with the degrees of numerator and denominator equal (if m is even) or with the degree of the denominator larger by one (if m is odd, cf. equation 3.2.2 above). For the derivation of the algorithm, refer to [1]. The algorithm is summarized by a recurrence relation exactly analogous to equation (3.1.3) for polynomial approximation:

$$R_{i(i+1)\dots(i+m)} = R_{(i+1)\dots(i+m)} + \frac{R_{(i+1)\dots(i+m)} - R_{i\dots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\dots(i+m)} - R_{i\dots(i+m-1)}}{R_{(i+1)\dots(i+m)} - R_{(i+1)\dots(i+m-1)}}\right) - 1} \tag{3.2.3}$$

This recurrence generates the rational functions through $m+1$ points from the ones through m and (the term $R_{(i+1)\dots(i+m-1)}$ in equation 3.2.3) $m-1$ points. It is started with

$$R_i = y_i \tag{3.2.4}$$

and with

$$R \equiv [R_{i(i+1)\dots(i+m)} \quad \text{with} \quad m = -1] = 0 \tag{3.2.5}$$

Now, exactly as in equations (3.1.4) and (3.1.5) above, we can convert the recurrence (3.2.3) to one involving only the small differences

$$\begin{aligned} C_{m,i} &\equiv R_{i\dots(i+m)} - R_{i\dots(i+m-1)} \\ D_{m,i} &\equiv R_{i\dots(i+m)} - R_{(i+1)\dots(i+m)} \end{aligned} \tag{3.2.6}$$

Note that these satisfy the relation

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \tag{3.2.7}$$

which is useful in proving the recurrences

$$\begin{aligned} D_{m+1,i} &= \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}} \\ C_{m+1,i} &= \frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}} \end{aligned} \tag{3.2.8}$$

This recurrence is implemented in the following function, whose use is analogous in every way to `polint` in §3.1. Note again that unit-offset input arrays are assumed (§1.2).

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-25            A small number.
#define FREERETURN {free_vector(d,1,n);free_vector(c,1,n);return;}

void ratint(float xa[], float ya[], int n, float x, float *y, float *dy)
```
Given arrays `xa[1..n]` and `ya[1..n]`, and given a value of `x`, this routine returns a value of `y` and an accuracy estimate `dy`. The value returned is that of the diagonal rational function, evaluated at `x`, which passes through the `n` points ($\mathtt{xa}_i, \mathtt{ya}_i$), $i = 1...\mathtt{n}$.
```
{
    int m,i,ns=1;
    float w,t,hh,h,dd,*c,*d;
    c=vector(1,n);
    d=vector(1,n);
    hh=fabs(x-xa[1]);
    for (i=1;i<=n;i++) {
        h=fabs(x-xa[i]);
        if (h == 0.0) {
            *y=ya[i];
            *dy=0.0;
            FREERETURN
        } else if (h < hh) {
            ns=i;
            hh=h;
        }
        c[i]=ya[i];
        d[i]=ya[i]+TINY;          The TINY part is needed to prevent a rare zero-over-zero
    }                                 condition.
    *y=ya[ns--];
    for (m=1;m<n;m++) {
        for (i=1;i<=n-m;i++) {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;          h will never be zero, since this was tested in the initial-
            t=(xa[i]-x)*d[i]/h;       izing loop.
            dd=t-c[i+1];
            if (dd == 0.0) nrerror("Error in routine ratint");
            This error condition indicates that the interpolating function has a pole at the
            requested value of x.
            dd=w/dd;
            d[i]=c[i+1]*dd;
            c[i]=t*dd;
        }
        *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
    }
    FREERETURN
}
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.2. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.

3.3 Cubic Spline Interpolation

Given a tabulated function $y_i = y(x_i),\ i = 1...N$, focus attention on one particular interval, between x_j and x_{j+1}. Linear interpolation in that interval gives the interpolation formula

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

where

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.3.2}$$

Equations (3.3.1) and (3.3.2) are a special case of the general Lagrange interpolation formula (3.1.1).

Since it is (piecewise) linear, equation (3.3.1) has zero second derivative in the interior of each interval, and an undefined, or infinite, second derivative at the abscissas x_j. The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative, and continuous in the second derivative, both within an interval and at its boundaries.

Suppose, contrary to fact, that in addition to the tabulated values of y_i, we also have tabulated values for the function's second derivatives, y'', that is, a set of numbers y_i''. Then, within each interval, we can add to the right-hand side of equation (3.3.1) a cubic polynomial whose second derivative varies linearly from a value y_j'' on the left to a value y_{j+1}'' on the right. Doing so, we will have the desired continuous second derivative. If we also construct the cubic polynomial to have zero *values* at x_j and x_{j+1}, then adding it in will not spoil the agreement with the tabulated functional values y_j and y_{j+1} at the endpoints x_j and x_{j+1}.

A little side calculation shows that there is only one way to arrange this construction, namely replacing (3.3.1) by

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \tag{3.3.3}$$

where A and B are defined in (3.3.2) and

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \qquad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \tag{3.3.4}$$

Notice that the dependence on the independent variable x in equations (3.3.3) and (3.3.4) is entirely through the linear x-dependence of A and B, and (through A and B) the cubic x-dependence of C and D.

We can readily check that y'' is in fact the second derivative of the new interpolating polynomial. We take derivatives of equation (3.3.3) with respect to x, using the definitions of A, B, C, D to compute $dA/dx, dB/dx, dC/dx$, and dD/dx. The result is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \tag{3.3.5}$$

for the first derivative, and

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \tag{3.3.6}$$

for the second derivative. Since $A = 1$ at x_j, $A = 0$ at x_{j+1}, while B is just the other way around, (3.3.6) shows that y'' is just the tabulated second derivative, and also that the second derivative will be continuous across (e.g.) the boundary between the two intervals (x_{j-1}, x_j) and (x_j, x_{j+1}).

The only problem now is that we supposed the y_i'''s to be known, when, actually, they are not. However, we have not yet required that the *first* derivative, computed from equation (3.3.5), be continuous across the boundary between two intervals. The key idea of a cubic spline is to require this continuity and to use it to get equations for the second derivatives y_i''.

The required equations are obtained by setting equation (3.3.5) evaluated for $x = x_j$ in the interval (x_{j-1}, x_j) equal to the same equation evaluated for $x = x_j$ but in the interval (x_j, x_{j+1}). With some rearrangement, this gives (for $j = 2, \dots, N-1$)

$$\frac{x_j - x_{j-1}}{6} y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3} y_j'' + \frac{x_{j+1} - x_j}{6} y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{3.3.7}$$

These are $N - 2$ linear equations in the N unknowns $y_i'', i = 1, \dots, N$. Therefore there is a two-parameter family of possible solutions.

For a unique solution, we need to specify two further conditions, typically taken as boundary conditions at x_1 and x_N. The most common ways of doing this are either

- set one or both of y_1'' and y_N'' equal to zero, giving the so-called *natural cubic spline*, which has zero second derivative on one or both of its boundaries, or
- set either of y_1'' and y_N'' to values calculated from equation (3.3.5) so as to make the first derivative of the interpolating function have a specified value on either or both boundaries.

One reason that cubic splines are especially practical is that the set of equations (3.3.7), along with the two additional boundary conditions, are not only linear, but also *tridiagonal*. Each y_j'' is coupled only to its nearest neighbors at $j \pm 1$. Therefore, the equations can be solved in $O(N)$ operations by the tridiagonal algorithm (§2.4). That algorithm is concise enough to build right into the spline calculational routine. This makes the routine not completely transparent as an implementation of (3.3.7), so we encourage you to study it carefully, comparing with `tridag` (§2.4). Arrays are assumed to be unit-offset. If you have zero-offset arrays, see §1.2.

```
#include "nrutil.h"

void spline(float x[], float y[], int n, float yp1, float ypn, float y2[])
```
Given arrays `x[1..n]` and `y[1..n]` containing a tabulated function, i.e., $y_i = f(x_i)$, with $x_1 < x_2 < \ldots < x_N$, and given values `yp1` and `ypn` for the first derivative of the interpolating function at points 1 and `n`, respectively, this routine returns an array `y2[1..n]` that contains the second derivatives of the interpolating function at the tabulated points x_i. If `yp1` and/or `ypn` are equal to 1×10^{30} or larger, the routine is signaled to set the corresponding boundary condition for a natural spline, with zero second derivative on that boundary.
```
{
    int i,k;
    float p,qn,sig,un,*u;

    u=vector(1,n-1);
    if (yp1 > 0.99e30)                  The lower boundary condition is set either to be "nat-
        y2[1]=u[1]=0.0;                     ural"
    else {                              or else to have a specified first derivative.
        y2[1] = -0.5;
        u[1]=(3.0/(x[2]-x[1]))*((y[2]-y[1])/(x[2]-x[1])-yp1);
    }
    for (i=2;i<=n-1;i++) {          This is the decomposition loop of the tridiagonal al-
        sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);        gorithm. y2 and u are used for tem-
        p=sig*y2[i-1]+2.0;                        porary storage of the decomposed
        y2[i]=(sig-1.0)/p;                        factors.
        u[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
        u[i]=(6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
    }
    if (ypn > 0.99e30)              The upper boundary condition is set either to be
        qn=un=0.0;                      "natural"
    else {                          or else to have a specified first derivative.
        qn=0.5;
        un=(3.0/(x[n]-x[n-1]))*(ypn-(y[n]-y[n-1])/(x[n]-x[n-1]));
    }
    y2[n]=(un-qn*u[n-1])/(qn*y2[n-1]+1.0);
    for (k=n-1;k>=1;k--)            This is the backsubstitution loop of the tridiagonal
        y2[k]=y2[k]*y2[k+1]+u[k];       algorithm.
    free_vector(u,1,n-1);
}
```

It is important to understand that the program spline is called only *once* to process an entire tabulated function in arrays x_i and y_i. Once this has been done, values of the interpolated function for any value of x are obtained by calls (as many as desired) to a separate routine splint (for "*spl*ine *int*erpolation"):

```
void splint(float xa[], float ya[], float y2a[], int n, float x, float *y)
```
Given the arrays xa[1..n] and ya[1..n], which tabulate a function (with the xa_i's in order), and given the array y2a[1..n], which is the output from spline above, and given a value of x, this routine returns a cubic-spline interpolated value y.
```
{
    void nrerror(char error_text[]);
    int klo,khi,k;
    float h,b,a;

    klo=1;                       We will find the right place in the table by means of
    khi=n;                           bisection. This is optimal if sequential calls to this
    while (khi-klo > 1) {            routine are at random values of x. If sequential calls
        k=(khi+klo) >> 1;            are in order, and closely spaced, one would do better
        if (xa[k] > x) khi=k;        to store previous values of klo and khi and test if
        else klo=k;                  they remain appropriate on the next call.
    }                            klo and khi now bracket the input value of x.
    h=xa[khi]-xa[klo];
    if (h == 0.0) nrerror("Bad xa input to routine splint");    The xa's must be dis-
    a=(xa[khi]-x)/h;                                                    tinct.
    b=(x-xa[klo])/h;             Cubic spline polynomial is now evaluated.
    *y=a*ya[klo]+b*ya[khi]+((a*a*a-a)*y2a[klo]+(b*b*b-b)*y2a[khi])*(h*h)/6.0;
}
```

CITED REFERENCES AND FURTHER READING:

De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer-Verlag).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §§4.4–4.5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.4.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §3.8.

3.4 How to Search an Ordered Table

Suppose that you have decided to use some particular interpolation scheme, such as fourth-order polynomial interpolation, to compute a function $f(x)$ from a set of tabulated x_i's and f_i's. Then you will need a fast way of finding your place in the table of x_i's, given some particular value x at which the function evaluation is desired. This problem is not properly one of numerical analysis, but it occurs so often in practice that it would be negligent of us to ignore it.

Formally, the problem is this: Given an array of abscissas `xx[j]`, `j`=1, 2, . . . ,`n`, with the elements either monotonically increasing or monotonically decreasing, and given a number `x`, find an integer `j` such that `x` lies between `xx[j]` and `xx[j+1]`. For this task, let us define fictitious array elements `xx[0]` and `xx[n+1]` equal to plus or minus infinity (in whichever order is consistent with the monotonicity of the table). Then `j` will always be between 0 and `n`, inclusive; a value of 0 indicates "off-scale" at one end of the table, `n` indicates off-scale at the other end.

In most cases, when all is said and done, it is hard to do better than *bisection*, which will find the right place in the table in about $\log_2$`n` tries. We already did use bisection in the spline evaluation routine `splint` of the preceding section, so you might glance back at that. Standing by itself, a bisection routine looks like this:

```
void locate(float xx[], unsigned long n, float x, unsigned long *j)
Given an array xx[1..n], and given a value x, returns a value j such that x is between xx[j]
and xx[j+1]. xx must be monotonic, either increasing or decreasing. j=0 or j=n is returned
to indicate that x is out of range.
{
    unsigned long ju,jm,jl;
    int ascnd;

    jl=0;                       Initialize lower
    ju=n+1;                     and upper limits.
    ascnd=(xx[n] >= xx[1]);
    while (ju-jl > 1) {         If we are not yet done,
        jm=(ju+jl) >> 1;        compute a midpoint,
        if (x >= xx[jm] == ascnd)
            jl=jm;              and replace either the lower limit
        else
            ju=jm;              or the upper limit, as appropriate.
    }                           Repeat until the test condition is satisfied.
    if (x == xx[1]) *j=1;       Then set the output
    else if(x == xx[n]) *j=n-1;
    else *j=jl;
}                               and return.
```

A unit-offset array `xx` is assumed. To use `locate` with a zero-offset array, remember to subtract 1 from the address of `xx`, and also from the returned value `j`.

Search with Correlated Values

Sometimes you will be in the situation of searching a large table many times, and with nearly identical abscissas on consecutive searches. For example, you may be generating a function that is used on the right-hand side of a differential equation: Most differential-equation integrators, as we shall see in Chapter 16, call

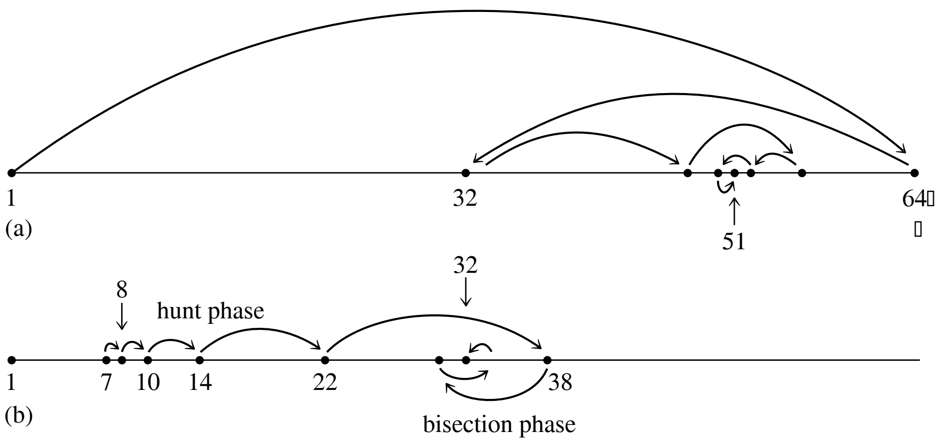


Figure 3.4.1. (a) The routine `locate` finds a table entry by bisection. Shown here is the sequence of steps that converge to element 51 in a table of length 64. (b) The routine `hunt` searches from a previous known position in the table by increasing steps, then converges by bisection. Shown here is a particularly unfavorable example, converging to element 32 from element 7. A favorable example would be convergence to an element near 7, such as 9, which would require just three "hops."

for right-hand side evaluations at points that hop back and forth a bit, but whose trend moves slowly in the direction of the integration.

In such cases it is wasteful to do a full bisection, *ab initio*, on each call. The following routine instead starts with a guessed position in the table. It first "hunts," either up or down, in increments of 1, then 2, then 4, etc., until the desired value is bracketed. Second, it then bisects in the bracketed interval. At worst, this routine is about a factor of 2 slower than `locate` above (if the hunt phase expands to include the whole table). At best, it can be a factor of $\log_2 n$ faster than `locate`, if the desired point is usually quite close to the input guess. Figure 3.4.1 compares the two routines.

```
void hunt(float xx[], unsigned long n, float x, unsigned long *jlo)
```
Given an array `xx[1..n]`, and given a value `x`, returns a value `jlo` such that `x` is between `xx[jlo]` and `xx[jlo+1]`. `xx[1..n]` must be monotonic, either increasing or decreasing. `jlo=0` or `jlo=n` is returned to indicate that `x` is out of range. `jlo` on input is taken as the initial guess for `jlo` on output.
```
{
    unsigned long jm,jhi,inc;
    int ascnd;

    ascnd=(xx[n] >= xx[1]);              True if ascending order of table, false otherwise.
    if (*jlo <= 0 || *jlo > n) {         Input guess not useful. Go immediately to bisec-
        *jlo=0;                              tion.
        jhi=n+1;
    } else {
        inc=1;                           Set the hunting increment.
        if (x >= xx[*jlo] == ascnd) {    Hunt up:
            if (*jlo == n) return;
            jhi=(*jlo)+1;
            while (x >= xx[jhi] == ascnd) {        Not done hunting,
                *jlo=jhi;
                inc += inc;              so double the increment
                jhi=(*jlo)+inc;
                if (jhi > n) {           Done hunting, since off end of table.
                    jhi=n+1;
                    break;
                }                        Try again.
            }                               Done hunting, value bracketed.
        } else {                            Hunt down:
            if (*jlo == 1) {
                *jlo=0;
                return;
            }
            jhi=(*jlo)--;
            while (x < xx[*jlo] == ascnd) {       Not done hunting,
                jhi=(*jlo);
                inc <<= 1;                  so double the increment
                if (inc >= jhi) {           Done hunting, since off end of table.
                    *jlo=0;
                    break;
                }
                else *jlo=jhi-inc;
            }                               and try again.
        }                                   Done hunting, value bracketed.
    }                                       Hunt is done, so begin the final bisection phase:
    while (jhi-(*jlo) != 1) {
        jm=(jhi+(*jlo)) >> 1;
        if (x >= xx[jm] == ascnd)
            *jlo=jm;
        else
            jhi=jm;
    }
    if (x == xx[n]) *jlo=n-1;
    if (x == xx[1]) *jlo=1;
}
```

If your array `xx` is zero-offset, read the comment following `locate`, above.

After the Hunt

The problem: Routines `locate` and `hunt` return an index `j` such that your desired value lies between table entries `xx[j]` and `xx[j+1]`, where `xx[1..n]` is the full length of the table. But, to obtain an `m`-point interpolated value using a routine like `polint` (§3.1) or `ratint` (§3.2), you need to supply much shorter `xx` and `yy` arrays, of length `m`. How do you make the connection?

The solution: Calculate

```
k = IMIN(IMAX(j-(m-1)/2,1),n+1-m)
```

(The macros `IMIN` and `IMAX` give the minimum and maximum of two integer arguments; see §1.2 and Appendix B.) This expression produces the index of the leftmost member of an `m`-point set of points centered (insofar as possible) between `j` and `j+1`, but bounded by 1 at the left and `n` at the right. `C` then lets you call the interpolation routine with array addresses offset by `k`, e.g.,

```
polint(&xx[k-1],&yy[k-1],m,...)
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.5 Coefficients of the Interpolating Polynomial

Occasionally you may wish to know not the value of the interpolating polynomial that passes through a (small!) number of points, but the coefficients of that polynomial. A valid use of the coefficients might be, for example, to compute simultaneous interpolated values of the function and of several of its derivatives (see §5.3), or to convolve a segment of the tabulated function with some other function, where the moments of that other function (i.e., its convolution with powers of x) are known analytically.

However, please be certain that the coefficients are what you need. Generally the coefficients of the interpolating polynomial can be determined much less accurately than its value at a desired abscissa. Therefore it is not a good idea to determine the coefficients only for use in calculating interpolating values. Values thus calculated will not pass exactly through the tabulated points, for example, while values computed by the routines in §3.1–§3.3 will pass exactly through such points.

Also, you should not mistake the interpolating polynomial (and its coefficients) for its cousin, the *best fit* polynomial through a data set. Fitting is a *smoothing* process, since the number of fitted coefficients is typically much less than the number of data points. Therefore, fitted coefficients can be accurately and stably determined even in the presence of statistical errors in the tabulated values. (See §14.8.) Interpolation, where the number of coefficients and number of tabulated points are equal, takes the tabulated values as perfect. If they in fact contain statistical errors, these can be magnified into oscillations of the interpolating polynomial in between the tabulated points.

As before, we take the tabulated points to be $y_i \equiv y(x_i)$. If the interpolating polynomial is written as

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_N x^N \tag{3.5.1}$$

then the c_i's are required to satisfy the linear equation

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} \tag{3.5.2}$$

This is a *Vandermonde matrix*, as described in §2.8. One could in principle solve equation (3.5.2) by standard techniques for linear equations generally (§2.3); however the special method that was derived in §2.8 is more efficient by a large factor, of order N, so it is much better.

Remember that Vandermonde systems can be quite ill-conditioned. In such a case, *no* numerical method is going to give a very accurate answer. Such cases do not, please note, imply any difficulty in finding interpolated *values* by the methods of §3.1, but only difficulty in finding *coefficients*.

Like the routine in §2.8, the following is due to G.B. Rybicki. Note that the arrays are all assumed to be zero-offset.

```
#include "nrutil.h"

void polcoe(float x[], float y[], int n, float cof[])
```
Given arrays x[0..n] and y[0..n] containing a tabulated function $y_i = f(x_i)$, this routine returns an array of coefficients cof[0..n], such that $y_i = \sum_j \text{cof}_j x_i^j$.
```
{
    int k,j,i;
    float phi,ff,b,*s;

    s=vector(0,n);
    for (i=0;i<=n;i++) s[i]=cof[i]=0.0;
    s[n] = -x[0];
    for (i=1;i<=n;i++) {                Coefficients s_i of the master polynomial P(x) are
        for (j=n-i;j<=n-1;j++)              found by recurrence.
            s[j] -= x[i]*s[j+1];
        s[n] -= x[i];
    }
    for (j=0;j<=n;j++) {
        phi=n+1;
        for (k=n;k>=1;k--)              The quantity phi = ∏_{j≠k}(x_j − x_k) is found as a
            phi=k*s[k]+x[j]*phi;            derivative of P(x_j).
        ff=y[j]/phi;
        b=1.0;                          Coefficients of polynomials in each term of the La-
        for (k=n;k>=0;k--) {                grange formula are found by synthetic division of
            cof[k] += b*ff;                 P(x) by (x − x_j). The solution c_k is accumu-
            b=s[k]+x[j]*b;                  lated.
        }
    }
    free_vector(s,0,n);
}
```

Another Method

Another technique is to make use of the function value interpolation routine already given (polint §3.1). If we interpolate (or extrapolate) to find the value of the interpolating polynomial at $x = 0$, then this value will evidently be c_0. Now we can subtract c_0 from the y_i's and divide each by its corresponding x_i. Throwing out one point (the one with smallest x_i is a good candidate), we can repeat the procedure to find c_1, and so on.

It is not instantly obvious that this procedure is stable, but we have generally found it to be somewhat *more* stable than the routine immediately preceding. This method is of order N^3, while the preceding one was of order N^2. You will find, however, that neither works very well for large N, because of the intrinsic ill-condition of the Vandermonde problem. In single precision, N up to 8 or 10 is satisfactory; about double this in double precision.

```
#include <math.h>
#include "nrutil.h"

void polcof(float xa[], float ya[], int n, float cof[])
```
Given arrays xa[0..n] and ya[0..n] containing a tabulated function $\text{ya}_i = f(\text{xa}_i)$, this routine returns an array of coefficients cof[0..n] such that $\text{ya}_i = \sum_j \text{cof}_j \text{xa}_i^j$.
```
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    int k,j,i;
    float xmin,dy,*x,*y;
    x=vector(0,n);
    y=vector(0,n);
    for (j=0;j<=n;j++) {
        x[j]=xa[j];
        y[j]=ya[j];
    }
    for (j=0;j<=n;j++) {
        polint(x-1,y-1,n+1-j,0.0,&cof[j],&dy);
        Subtract 1 from the pointers to x and y because polint uses dimensions [1..n]. We
        extrapolate to x = 0.
        xmin=1.0e38;
        k = -1;
        for (i=0;i<=n-j;i++) {                  Find the remaining x_i of smallest
            if (fabs(x[i]) < xmin) {                absolute value,
                xmin=fabs(x[i]);
                k=i;
            }
            if (x[i]) y[i]=(y[i]-cof[j])/x[i];  (meanwhile reducing all the terms)
        }
        for (i=k+1;i<=n-j;i++) {                and eliminate it.
            y[i-1]=y[i];
            x[i-1]=x[i];
        }
    }
    free_vector(y,0,n);
    free_vector(x,0,n);
}
```

If the point $x = 0$ is not in (or at least close to) the range of the tabulated x_i's, then the coefficients of the interpolating polynomial will in general become very large. However, the real "information content" of the coefficients is in small differences from the "translation-induced" large values. This is one cause of ill-conditioning, resulting in loss of significance and poorly determined coefficients. You should consider redefining the origin of the problem, to put $x = 0$ in a sensible place.

Another pathology is that, if too high a degree of interpolation is attempted on a smooth function, the interpolating polynomial will attempt to use its high-degree coefficients, in combinations with large and almost precisely canceling combinations, to match the tabulated values down to the last possible epsilon of accuracy. This effect is the same as the intrinsic tendency of the interpolating polynomial values to oscillate (wildly) between its constrained points, and would be present even if the machine's floating precision were infinitely good. The above routines polcoe and polcof have slightly different sensitivities to the pathologies that can occur.

Are you still quite certain that using the *coefficients* is a good idea?

CITED REFERENCES AND FURTHER READING:

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §5.2.

3.6 Interpolation in Two or More Dimensions

In multidimensional interpolation, we seek an estimate of $y(x_1, x_2, \ldots, x_n)$ from an n-dimensional grid of tabulated values y and n one-dimensional vectors giving the tabulated values of each of the independent variables $x_1, x_2, \ldots, x_n$. We will not here consider the problem of interpolating on a mesh that is not Cartesian, i.e., has tabulated function values at "random" points in n-dimensional space rather than at the vertices of a rectangular array. For clarity, we will consider explicitly only the case of two dimensions, the cases of three or more dimensions being analogous in every way.

In two dimensions, we imagine that we are given a matrix of functional values `ya[1..m][1..n]`. We are also given an array `x1a[1..m]`, and an array `x2a[1..n]`. The relation of these input quantities to an underlying function $y(x_1, x_2)$ is

$$\texttt{ya[j][k]} = y(\texttt{x1a[j]}, \texttt{x2a[k]}) \tag{3.6.1}$$

We want to estimate, by interpolation, the function y at some untabulated point (x_1, x_2).

An important concept is that of the *grid square* in which the point (x_1, x_2) falls, that is, the four tabulated points that surround the desired interior point. For convenience, we will number these points from 1 to 4, counterclockwise starting from the lower left (see Figure 3.6.1). More precisely, if

$$\begin{aligned} \texttt{x1a[j]} &\le x_1 \le \texttt{x1a[j+1]} \\ \texttt{x2a[k]} &\le x_2 \le \texttt{x2a[k+1]} \end{aligned} \tag{3.6.2}$$

defines `j` and `k`, then

$$\begin{aligned} y_1 &\equiv \texttt{ya[j][k]} \\ y_2 &\equiv \texttt{ya[j+1][k]} \\ y_3 &\equiv \texttt{ya[j+1][k+1]} \\ y_4 &\equiv \texttt{ya[j][k+1]} \end{aligned} \tag{3.6.3}$$

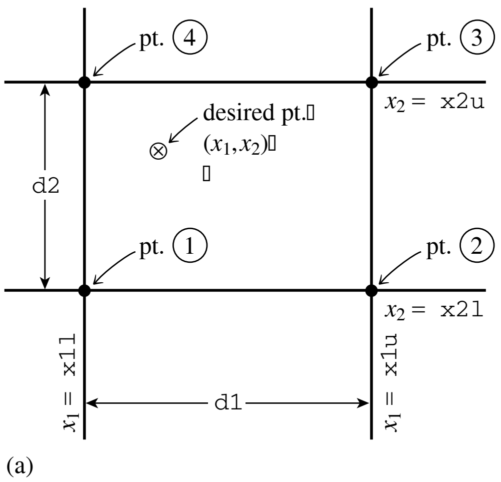
The simplest interpolation in two dimensions is *bilinear interpolation* on the grid square. Its formulas are:

$$\begin{aligned} t &\equiv (x_1 - \texttt{x1a[j]})/(\texttt{x1a[j+1]} - \texttt{x1a[j]}) \\ u &\equiv (x_2 - \texttt{x2a[k]})/(\texttt{x2a[k+1]} - \texttt{x2a[k]}) \end{aligned} \tag{3.6.4}$$

(so that t and u each lie between 0 and 1), and

$$y(x_1, x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4 \tag{3.6.5}$$

Bilinear interpolation is frequently "close enough for government work." As the interpolating point wanders from grid square to grid square, the interpolated



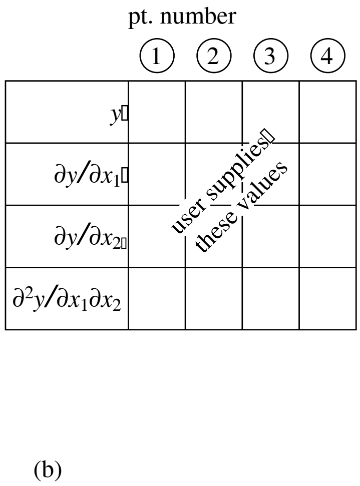

pt. number
1 2 3 4
y
$\partial y/\partial x_1$
$\partial y/\partial x_2$
$\partial^2 y/\partial x_1 \partial x_2$
user supplies these values
(b)

Figure 3.6.1. (a) Labeling of points used in the two-dimensional interpolation routines `bcuint` and `bcucof`. (b) For each of the four points in (a), the user supplies one function value, two first derivatives, and one cross-derivative, a total of 16 numbers.

function value changes continuously. However, the gradient of the interpolated function changes discontinuously at the boundaries of each grid square.

There are two distinctly different directions that one can take in going beyond bilinear interpolation to higher-order methods: One can use higher order to obtain increased accuracy for the interpolated function (for sufficiently smooth functions!), without necessarily trying to fix up the continuity of the gradient and higher derivatives. Or, one can make use of higher order to enforce smoothness of some of these derivatives as the interpolating point crosses grid-square boundaries. We will now consider each of these two directions in turn.

Higher Order for Accuracy

The basic idea is to break up the problem into a succession of one-dimensional interpolations. If we want to do `m-1` order interpolation in the x_1 direction, and `n-1` order in the x_2 direction, we first locate an `m` $\times$ `n` sub-block of the tabulated function matrix that contains our desired point (x_1, x_2). We then do `m` one-dimensional interpolations in the x_2 direction, i.e., on the rows of the sub-block, to get function values at the points (`x1a[j]`, x_2), `j` $= 1, \ldots,$ `m`. Finally, we do a last interpolation in the x_1 direction to get the answer. If we use the polynomial interpolation routine `polint` of §3.1, and a sub-block which is presumed to be already located (and addressed through the pointer `float **ya`, see §1.2), the procedure looks like this:

```
#include "nrutil.h"

void polin2(float x1a[], float x2a[], float **ya, int m, int n, float x1,
    float x2, float *y, float *dy)
```
Given arrays `x1a[1..m]` and `x2a[1..n]` of independent variables, and a submatrix of function values `ya[1..m][1..n]`, tabulated at the grid points defined by `x1a` and `x2a`; and given values `x1` and `x2` of the independent variables; this routine returns an interpolated function value `y`, and an accuracy indication `dy` (based only on the interpolation in the `x1` direction, however).
```
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    int j;
    float *ymtmp;

    ymtmp=vector(1,m);
    for (j=1;j<=m;j++) {                            Loop over rows.
        polint(x2a,ya[j],n,x2,&ymtmp[j],dy);        Interpolate answer into temporary stor-
    }                                                   age.
    polint(x1a,ymtmp,m,x1,y,dy);                    Do the final interpolation.
    free_vector(ymtmp,1,m);
}
```

Higher Order for Smoothness: Bicubic Interpolation

We will give two methods that are in common use, and which are themselves not unrelated. The first is usually called *bicubic interpolation*.

Bicubic interpolation requires the user to specify at each grid point not just the function $y(x_1, x_2)$, but also the gradients $\partial y/\partial x_1 \equiv y_{,1}$, $\partial y/\partial x_2 \equiv y_{,2}$ and the cross derivative $\partial^2 y/\partial x_1 \partial x_2 \equiv y_{,12}$. Then an interpolating function that is *cubic* in the scaled coordinates t and u (equation 3.6.4) can be found, with the following properties: (i) The values of the function and the specified derivatives are reproduced exactly on the grid points, and (ii) the values of the function and the specified derivatives change continuously as the interpolating point crosses from one grid square to another.

It is important to understand that nothing in the equations of bicubic interpolation requires you to specify the extra derivatives *correctly*! The smoothness properties are tautologically "forced," and have nothing to do with the "accuracy" of the specified derivatives. It is a separate problem for you to decide how to obtain the values that are specified. The better you do, the more *accurate* the interpolation will be. But it will be *smooth* no matter what you do.

Best of all is to know the derivatives analytically, or to be able to compute them accurately by numerical means, at the grid points. Next best is to determine them by numerical differencing from the functional values already tabulated on the grid. The relevant code would be something like this (using centered differencing):

```
    y1a[j][k]=(ya[j+1][k]-ya[j-1][k])/(x1a[j+1]-x1a[j-1]);
    y2a[j][k]=(ya[j][k+1]-ya[j][k-1])/(x2a[k+1]-x2a[k-1]);
    y12a[j][k]=(ya[j+1][k+1]-ya[j+1][k-1]-ya[j-1][k+1]+ya[j-1][k-1])
            /((x1a[j+1]-x1a[j-1])*(x2a[k+1]-x2a[k-1]));
```

To do a bicubic interpolation within a grid square, given the function `y` and the derivatives `y1`, `y2`, `y12` at each of the four corners of the square, there are two steps: First obtain the sixteen quantities c_{ij}, $i, j = 1, \ldots, 4$ using the routine `bcucof` below. (The formulas that obtain the c's from the function and derivative values are just a complicated linear transformation, with coefficients which, having been determined once in the mists of numerical history, can be tabulated and forgotten.) Next, substitute the c's into any or all of the following bicubic formulas for function and derivatives, as desired:

$$
\begin{aligned}
y(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} c_{ij} t^{i-1} u^{j-1} \\
y_{,1}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (i-1) c_{ij} t^{i-2} u^{j-1} (dt/dx_1) \\
y_{,2}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (j-1) c_{ij} t^{i-1} u^{j-2} (du/dx_2) \\
y_{,12}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (i-1)(j-1) c_{ij} t^{i-2} u^{j-2} (dt/dx_1)(du/dx_2)
\end{aligned}
\tag{3.6.6}
$$

where t and u are again given by equation (3.6.4).

```
void bcucof(float y[], float y1[], float y2[], float y12[], float d1, float d2,
    float **c)
```
Given arrays `y[1..4]`, `y1[1..4]`, `y2[1..4]`, and `y12[1..4]`, containing the function, gradients, and cross derivative at the four grid points of a rectangular grid cell (numbered counterclockwise from the lower left), and given `d1` and `d2`, the length of the grid cell in the 1- and 2-directions, this routine returns the table `c[1..4][1..4]` that is used by routine `bcuint` for bicubic interpolation.
```
{
    static int wt[16][16]=
        { 1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
        -3,0,0,3,0,0,0,0,-2,0,0,-1,0,0,0,0,
        2,0,0,-2,0,0,0,0,1,0,0,1,0,0,0,0,
        0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,
        0,0,0,0,-3,0,0,3,0,0,0,0,-2,0,0,-1,
        0,0,0,0,2,0,0,-2,0,0,0,0,1,0,0,1,
        -3,3,0,0,-2,-1,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,-3,3,0,0,-2,-1,0,0,
        9,-9,9,-9,6,3,-3,-6,6,-6,-3,3,4,2,1,2,
        -6,6,-6,6,-4,-2,2,4,-3,3,3,-3,-2,-1,-1,-2,
        2,-2,0,0,1,1,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,2,-2,0,0,1,1,0,0,
        -6,6,-6,6,-3,-3,3,3,-4,4,2,-2,-2,-2,-1,-1,
        4,-4,4,-4,2,2,-2,-2,2,-2,-2,2,1,1,1,1};
    int l,k,j,i;
    float xx,d1d2,cl[16],x[16];

    d1d2=d1*d2;
    for (i=1;i<=4;i++) {          Pack a temporary vector x.
        x[i-1]=y[i];
        x[i+3]=y1[i]*d1;
        x[i+7]=y2[i]*d2;
        x[i+11]=y12[i]*d1d2;
    }
    for (i=0;i<=15;i++) {         Matrix multiply by the stored table.
        xx=0.0;
        for (k=0;k<=15;k++) xx += wt[i][k]*x[k];
        cl[i]=xx;
    }
    l=0;
    for (i=1;i<=4;i++)            Unpack the result into the output table.
        for (j=1;j<=4;j++) c[i][j]=cl[l++];
}
```

The implementation of equation (3.6.6), which performs a bicubic interpolation, gives back the interpolated function value and the two gradient values, and uses the above routine `bcucof`, is simply:

```
#include "nrutil.h"

void bcuint(float y[], float y1[], float y2[], float y12[], float x1l,
    float x1u, float x2l, float x2u, float x1, float x2, float *ansy,
    float *ansy1, float *ansy2)
```
Bicubic interpolation within a grid square. Input quantities are `y,y1,y2,y12` (as described in `bcucof`); `x1l` and `x1u`, the lower and upper coordinates of the grid square in the 1-direction; `x2l` and `x2u` likewise for the 2-direction; and `x1,x2`, the coordinates of the desired point for the interpolation. The interpolated function value is returned as `ansy`, and the interpolated gradient values as `ansy1` and `ansy2`. This routine calls `bcucof`.
```
{
    void bcucof(float y[], float y1[], float y2[], float y12[], float d1,
        float d2, float **c);
    int i;
    float t,u,d1,d2,**c;

    c=matrix(1,4,1,4);
    d1=x1u-x1l;
    d2=x2u-x2l;
    bcucof(y,y1,y2,y12,d1,d2,c);          Get the c's.
    if (x1u == x1l || x2u == x2l) nrerror("Bad input in routine bcuint");
    t=(x1-x1l)/d1;                        Equation (3.6.4).
    u=(x2-x2l)/d2;
    *ansy=(*ansy2)=(*ansy1)=0.0;
    for (i=4;i>=1;i--) {                  Equation (3.6.6).
        *ansy=t*(*ansy)+((c[i][4]*u+c[i][3])*u+c[i][2])*u+c[i][1];
        *ansy2=t*(*ansy2)+(3.0*c[i][4]*u+2.0*c[i][3])*u+c[i][2];
        *ansy1=u*(*ansy1)+(3.0*c[4][i]*t+2.0*c[3][i])*t+c[2][i];
    }
    *ansy1 /= d1;
    *ansy2 /= d2;
    free_matrix(c,1,4,1,4);
}
```

Higher Order for Smoothness: Bicubic Spline

The other common technique for obtaining smoothness in two-dimensional interpolation is the *bicubic spline*. Actually, this is equivalent to a special case of bicubic interpolation: The interpolating function is of the same functional form as equation (3.6.6); the values of the derivatives at the grid points are, however, determined "globally" by one-dimensional splines. However, bicubic splines are usually implemented in a form that looks rather different from the above bicubic interpolation routines, instead looking much closer in form to the routine `polin2` above: To interpolate one functional value, one performs `m` one-dimensional splines across the rows of the table, followed by one additional one-dimensional spline down the newly created column. It is a matter of taste (and trade-off between time and memory) as to how much of this process one wants to precompute and store. Instead of precomputing and storing all the derivative information (as in bicubic interpolation), spline users typically precompute and store only one auxiliary table, of second derivatives in one direction only. Then one need only do spline *evaluations* (not constructions) for the `m` row splines; one must still do a construction *and* an evaluation for the final column spline. (Recall that a spline construction is a process of order N, while a spline evaluation is only of order $\log N$ — and that is just to find the place in the table!)

Here is a routine to precompute the auxiliary second-derivative table:

```
void splie2(float x1a[], float x2a[], float **ya, int m, int n, float **y2a)
```
Given an m by n tabulated function ya[1..m][1..n], and tabulated independent variables x2a[1..n], this routine constructs one-dimensional natural cubic splines of the rows of ya and returns the second-derivatives in the array y2a[1..m][1..n]. (The array x1a[1..m] is included in the argument list merely for consistency with routine splin2.)
```
{
    void spline(float x[], float y[], int n, float yp1, float ypn, float y2[]);
    int j;

    for (j=1;j<=m;j++)
        spline(x2a,ya[j],n,1.0e30,1.0e30,y2a[j]);        Values 1×10^30 signal a nat-
}                                                          ural spline.
```

(If you want to interpolate on a sub-block of a bigger matrix, see §1.2.)

After the above routine has been executed once, any number of bicubic spline interpolations can be performed by successive calls of the following routine:

```
#include "nrutil.h"

void splin2(float x1a[], float x2a[], float **ya, float **y2a, int m, int n,
    float x1, float x2, float *y)
```
Given x1a, x2a, ya, m, n as described in splie2 and y2a as produced by that routine; and given a desired interpolating point x1,x2; this routine returns an interpolated function value y by bicubic spline interpolation.
```
{
    void spline(float x[], float y[], int n, float yp1, float ypn, float y2[]);
    void splint(float xa[], float ya[], float y2a[], int n, float x, float *y);
    int j;
    float *ytmp,*yytmp;

    ytmp=vector(1,m);
    yytmp=vector(1,m);             Perform m evaluations of the row splines constructed by
    for (j=1;j<=m;j++)                 splie2, using the one-dimensional spline evaluator
        splint(x2a,ya[j],y2a[j],n,x2,&yytmp[j]);     splint.
    spline(x1a,yytmp,m,1.0e30,1.0e30,ytmp);      Construct the one-dimensional col-
    splint(x1a,yytmp,ytmp,m,x1,y);                   umn spline and evaluate it.
    free_vector(yytmp,1,m);
    free_vector(ytmp,1,m);
}
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Kinahan, B.F., and Harm, R. 1975, *Astrophysical Journal*, vol. 200, pp. 330–335.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §5.2.7.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7.