$f(x, y, z)$. 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,

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

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 $P_{123\dots N}$, 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

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

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

$$m + 1 = \mu + \nu + 1 \qquad (3.2.2)$$

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