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 $\mathtt{y}_i = f(\mathtt{x}_i)$, this routine returns an array of coefficients `cof[0..n]`, such that $\mathtt{y}_i = \sum_j \mathtt{cof}_j \mathtt{x}_i^j$.
```
{
    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 = \prod_{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 $\mathtt{ya}_i = f(\mathtt{xa}_i)$, this routine returns an array of coefficients `cof[0..n]` such that $\mathtt{ya}_i = \sum_j \mathtt{cof}_j \mathtt{xa}_i^j$.
```
{
    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