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

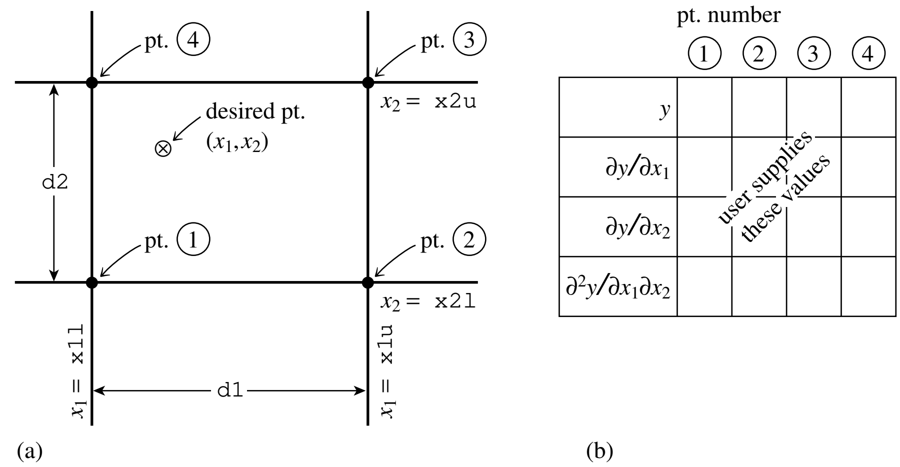

Figure 3.6.1. (a) Labeling of points used in the two-dimensional interpolation routines `bcuint` and `bcucof`. (b) For each of the four points in (a), the user supplies one function value, two first derivatives, and one cross-derivative, a total of 16 numbers.

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` × `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.