```
    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 $j = 2, \ldots, N-1$)

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

These are $N - 2$ linear equations in the N unknowns $y_i'', i = 1, \ldots, N$. 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 $\mathtt{x}_i$ and $\mathtt{y}_i$. 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 $\mathtt{xa}_i$'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