```
        for (i=0;i<mm-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) throw("Poly_interp error");
            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+1) < (mm-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.
    }
    return y;
}
```

The user interface to Poly_interp is virtually the same as for Linear_interp (end of §3.1), except that an additional argument in the constructor sets M, the number of points used (the order plus one). A cubic interpolator looks like this:

```
    Int n=...;
    VecDoub xx(n), yy(n);
    ...
    Poly_interp myfunc(xx,yy,4);
```

Poly_interp stores an error estimate dy for the most recent call to its interp function:

```
    Doub x,y,err;
    ...
    y = myfunc.interp(x);
    err = myfunc.dy;
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at http://www.nr.com/aands, §25.2.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.1.

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

3.3 Cubic Spline Interpolation

Given a tabulated function $y_i = y(x_i)$, $i = 0...N-1$, 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 (3.2.1).

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

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

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, and 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 $j = 1, \dots, N-2$)

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

These are $N-2$ linear equations in the N unknowns y_i'', $i = 0, \dots, N-1$. 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_0 and x_{N-1}. The most common ways of doing this are either

- set one or both of y_0'' and y_{N-1}'' 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_0'' and y_{N-1}'' 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.

Although the boundary condition for natural splines is commonly used, another possibility is to estimate the first derivatives at the endpoints from the first and last few tabulated points. For details of how to do this, see the end of §3.7. Best, of course, is if you can compute the endpoint first derivatives analytically.

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 function called by the constructor.

The object for cubic spline interpolation looks like this:

interp_1d.h
```
struct Spline_interp : Base_interp
```
Cubic spline interpolation object. Construct with **x** and **y** vectors, and (optionally) values of the first derivative at the endpoints, then call `interp` for interpolated values.
```
{
    VecDoub y2;

    Spline_interp(VecDoub_I &xv, VecDoub_I &yv, Doub yp1=1.e99, Doub ypn=1.e99)
    : Base_interp(xv,&yv[0],2), y2(xv.size())
    {sety2(&xv[0],&yv[0],yp1,ypn);}

    Spline_interp(VecDoub_I &xv, const Doub *yv, Doub yp1=1.e99, Doub ypn=1.e99)
    : Base_interp(xv,yv,2), y2(xv.size())
    {sety2(&xv[0],yv,yp1,ypn);}

    void sety2(const Doub *xv, const Doub *yv, Doub yp1, Doub ypn);
    Doub rawinterp(Int jl, Doub xv);
};
```

For now, you can ignore the second constructor; it will be used later for two-dimensional spline interpolation.