CHAPTER 3

Interpolation and Extrapolation

3.0 Introduction

We sometimes know the value of a function $f(x)$ at a set of points $x_0, x_1, \ldots, x_{N-1}$ (say, with $x_0 < \ldots < x_{N-1}$) but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i's are equally spaced, but not necessarily.

The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i. If the desired x is in between the largest and smallest of the x_i's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former investment analysts can attest).

Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions that might arise in practice. By far most common among the functional forms used are polynomials (§3.2). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.4). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapters 12 and 13.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: If we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function f at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function f at *any* desired points for the purpose of developing

your approximation. We deal with function approximation in Chapter 5.

One can easily find pathological functions that make a mockery of any interpolation scheme. Consider, for example, the function

$$f(x) = 3x^2 + \frac{1}{\pi^4}\ln\left[(\pi - x)^2\right] + 1 \tag{3.0.1}$$

which is well-behaved everywhere except at $x = \pi$, very mildly singular at $x = \pi$, and otherwise takes on all positive and negative values. Any interpolation based on the values $x = 3.13, 3.14, 3.15, 3.16$, will assuredly get a very wrong answer for the value $x = 3.1416$, even though a graph plotting those five points looks really quite smooth! (Try it.)

Because pathologies can lurk anywhere, it is highly desirable that an interpolation and extrapolation routine should provide an estimate of its own error. Such an error estimate can never be foolproof, of course. We could have a function that, for reasons known only to its maker, takes off wildly and unexpectedly between two tabulated points. Interpolation always presumes some degree of smoothness for the function interpolated, but within this framework of presumption, deviations from smoothness can be detected.

Conceptually, the interpolation process has two stages: (1) Fit (once) an interpolating function to the data points provided. (2) Evaluate (as many times as you wish) that interpolating function at a target point x.

However, this two-stage method is usually not the best way to proceed in practice. Typically it is computationally less efficient, and more susceptible to roundoff error, than methods that construct a functional estimate $f(x)$ directly from the N tabulated values every time one is desired. Many practical schemes start at a nearby point $f(x_i)$, and then add a sequence of (hopefully) decreasing corrections, as information from other nearby $f(x_i)$'s is incorporated. The procedure typically takes $O(M^2)$ operations, where $M \ll N$ is the number of local points used. If everything is well behaved, the last correction will be the smallest, and it can be used as an informal (though not rigorous) bound on the error. In schemes like this, we might also say that there are two stages, but now they are: (1) Find the right starting position in the table (x_i or i). (2) Perform the interpolation using M nearby values (for example, centered on x_i).

In the case of polynomial interpolation, it sometimes does happen that the coefficients of the interpolating polynomial are of interest, even though their use in *evaluating* the interpolating function should be frowned on. We deal with this possibility in §3.5.

Local interpolation, using M nearest-neighbor points, gives interpolated values $f(x)$ that do not, in general, have continuous first or higher derivatives. That happens because, as x crosses the tabulated values x_i, the interpolation scheme switches which tabulated points are the "local" ones. (If such a switch is allowed to occur anywhere *else*, then there will be a discontinuity in the interpolated function itself at that point. Bad idea!)

In situations where continuity of derivatives is a concern, one must use the "stiffer" interpolation provided by a so-called *spline* function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined "slightly" nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§3.3) are the

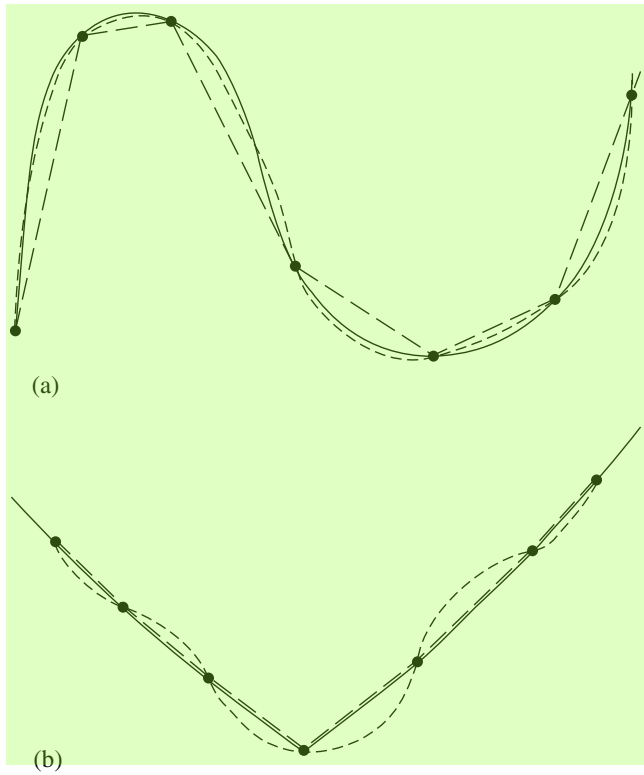

Figure 3.0.1. (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too "stiff," than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

The number M of points used in an interpolation scheme, minus 1, is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. If the added points are distant from the point of interest x, the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. This oscillation may have no relation at all to the behavior of the "true" function (see Figure 3.0.1). Of course, adding points *close* to the desired point usually does help, but a finer mesh implies a larger table of values, which is not always available.

For polynomial interpolation, it turns out that the *worst* possible arrangement of the x_i's is for them to be equally spaced. Unfortunately, this is by far the most common way that tabulated data are gathered or presented. High-order polynomial interpolation on equally spaced data is *ill-conditioned*: small changes in the data can give large differences in the oscillations between the points. The disease is particularly bad if you are interpolating on values of an analytic function that has poles in

the complex plane lying inside a certain oval region whose major axis is the M-point interval. But even if you have a function with no nearby poles, roundoff error can, in effect, create nearby poles and cause big interpolation errors. In §5.8 we will see that these issues go away if you are allowed to choose an optimal set of x_i's. But when you are handed a table of function values, that option is not available.

As the order is increased, it is typical for interpolation error to decrease at first, but only up to a certain point. Larger orders result in the error exploding.

For the reasons mentioned, it is a good idea to be cautious about high-order interpolation. We can enthusiastically endorse polynomial interpolation with 3 or 4 points; we are perhaps tolerant of 5 or 6; but we rarely go higher than that unless there is quite rigorous monitoring of estimated errors. Most of the interpolation methods in this chapter are applied *piecewise* using only M points at a time, so that the order is a fixed value $M - 1$ no matter how large N is. As mentioned, *splines* (§3.3) are a special case where the function and various derivatives are required to be continuous from one interval to the next, but the order is nevertheless held fixed a a small value (usually 3).

In §3.4 we discuss *rational function interpolation*. In many, but not all, cases, rational function interpolation is more robust, allowing higher orders to give higher accuracy. The standard algorithm, however, allows poles on the real axis or nearby in the complex plane. (This is not necessarily bad: You may be trying to approximate a function with such poles.) A newer method, *barycentric rational interpolation* (§3.4.1) suppresses all nearby poles. This is the only method in this chapter for which we might actually encourage experimentation with high order (say, > 6). Barycentric rational interpolation competes very favorably with splines: its error is often smaller, and the resulting approximation is infinitely smooth (unlike splines).

The interpolation methods below are also methods for extrapolation. An important application, in Chapter 17, is their use in the integration of ordinary differential equations. There, considerable care *is* taken with the monitoring of errors. Otherwise, the dangers of extrapolation cannot be overemphasized: An interpolating function, which is perforce an extrapolating function, will typically go berserk when the argument x is outside the range of tabulated values by more (and often significantly less) than the typical spacing of tabulated points.

Interpolation can be done in more than one dimension, e.g., for a function $f(x, y, z)$. Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations, but there are also other techniques applicable to scattered data. We discuss multidimensional methods in §3.6 – §3.8.

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.

Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, vol. 1 (Berlin: Springer), Chapter 9.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 2.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 3.

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.; reprinted 2001 (New York: Dover), Chapter 3.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), Chapter 6.

3.1 Preliminaries: Searching an Ordered Table

We want to define an interpolation object that knows everything about interpolation except one thing — how to actually interpolate! Then we can plug mathematically different interpolation methods into the object to get different objects sharing a common user interface. A key task common to all objects in this framework is finding your place in the table of x_i's, given some particular value x at which the function evaluation is desired. It is worth some effort to do this efficiently; otherwise you can easily spend more time searching the table than doing the actual interpolation.

Our highest-level object for one-dimensional interpolation is an abstract base class containing just one function intended to be called by the user: `interp(x)` returns the interpolated function value at x. The base class "promises," by declaring a virtual function `rawinterp(jlo,x)`, that every derived interpolation class will provide a method for local interpolation when given an appropriate local starting point in the table, an offset `jlo`. Interfacing between `interp` and `rawinterp` must thus be a method for calculating `jlo` from x, that is, for searching the table. In fact, we will use two such methods.

interp_1d.h

`struct Base_interp`

Abstract base class used by all interpolation routines in this chapter. Only the routine `interp` is called directly by the user.

```
{
    Int n, mm, jsav, cor, dj;
    const Doub *xx, *yy;
    Base_interp(VecDoub_I &x, const Doub *y, Int m)
    Constructor: Set up for interpolating on a table of x's and y's of length m. Normally called
    by a derived class, not by the user.
        : n(x.size()), mm(m), jsav(0), cor(0), xx(&x[0]), yy(y) {
        dj = MIN(1,(int)pow((Doub)n,0.25));
    }

    Doub interp(Doub x) {
    Given a value x, return an interpolated value, using data pointed to by xx and yy.
        Int jlo = cor ? hunt(x) : locate(x);
        return rawinterp(jlo,x);
    }

    Int locate(const Doub x);          See definitions below.
    Int hunt(const Doub x);

    Doub virtual rawinterp(Int jlo, Doub x) = 0;
    Derived classes provide this as the actual interpolation method.

};
```

Formally, the problem is this: Given an array of abscissas x_j, $j = 0, \ldots, N-1$, with the abscissas either monotonically increasing or monotonically decreasing, and given an integer $M \leq N$ and a number x, find an integer $j_{\rm lo}$ such that x is centered

among the M abscissas $x_{j_{\rm lo}}, \ldots, x_{j_{\rm lo}+M-1}$. By centered we mean that x lies between x_m and x_{m+1} insofar as possible, where

$$m = j_{\rm lo} + \left\lfloor \frac{M-2}{2} \right\rfloor \tag{3.1.1}$$

By "insofar as possible" we mean that $j_{\rm lo}$ should never be less than zero, nor should $j_{\rm lo}+M-1$ be greater than $N-1$.

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.

interp_1d.h

```
Int Base_interp::locate(const Doub x)
Given a value x, return a value j such that x is (insofar as possible) centered in the subrange
xx[j..j+mm-1], where xx is the stored pointer. The values in xx must be monotonic, either
increasing or decreasing. The returned value is not less than 0, nor greater than n-1.
{
    Int ju,jm,jl;
    if (n < 2 || mm < 2 || mm > n) throw("locate size error");
    Bool ascnd=(xx[n-1] >= xx[0]);      True if ascending order of table, false otherwise.
    jl=0;                               Initialize lower
    ju=n-1;                             and upper limits.
    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.
    cor = abs(jl-jsav) > dj ? 0 : 1;    Decide whether to use hunt or locate next time.
    jsav = jl;
    return MAX(0,MIN(n-mm,jl-((mm-2)>>1)));
}
```

The above `locate` routine accesses the array of values `xx[]` via a pointer stored by the base class. This rather primitive method of access, avoiding the use of a higher-level vector class like `VecDoub`, is here preferable for two reasons: (1) It's usually faster; and (2) for two-dimensional interpolation, we will later need to point directly into a row of a matrix. The peril of this design choice is that it assumes that consecutive values of a vector are stored consecutively, and similarly for consecutive values of a single row of a matrix. See discussion in §1.4.2.

3.1.1 Search with Correlated Values

Experience shows that in many, perhaps even most, applications, interpolation routines are called 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 17, call for right-hand side evaluations at points that hop back and forth a bit, but whose trend moves slowly in the direction of the integration.

In such cases it is wasteful to do a full bisection, *ab initio*, on each call. Much more desirable is to give our base class a tiny bit of intelligence: If it sees two calls that are "close," it anticipates that the next call will also be. Of course, there must not be too big a penalty if it anticipates wrongly.

The `hunt` method starts with a guessed position in the table. It first "hunts," either up or down, in increments of 1, then 2, then 4, etc., until the desired value is bracketed. It then bisects in the bracketed interval. At worst, this routine is about a

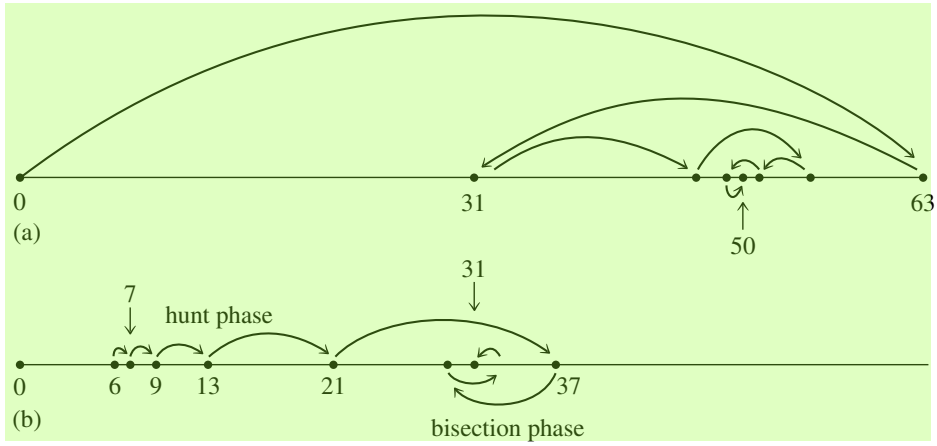

Figure 3.1.1. Finding a table entry by bisection. Shown here is the sequence of steps that converge to element 50 in a table of length 64. (b) The routine `hunt` searches from a previous known position in the table by increasing steps and then converges by bisection. Shown here is a particularly unfavorable example, converging to element 31 from element 6. A favorable example would be convergence to an element near 6, such as 8, which would require just three "hops."

factor of 2 slower than `locate` above (if the hunt phase expands to include the whole table). At best, it can be a factor of $\log_2 n$ faster than `locate`, if the desired point is usually quite close to the input guess. Figure 3.1.1 compares the two routines.

interp_1d.h

```
Int Base_interp::hunt(const Doub x)
```

Given a value `x`, return a value `j` such that `x` is (insofar as possible) centered in the subrange `xx[j..j+mm-1]`, where `xx` is the stored pointer. The values in `xx` must be monotonic, either increasing or decreasing. The returned value is not less than 0, nor greater than `n-1`.

```
{
    Int jl=jsav, jm, ju, inc=1;
    if (n < 2 || mm < 2 || mm > n) throw("hunt size error");
    Bool ascnd=(xx[n-1] >= xx[0]);          True if ascending order of table, false otherwise.
    if (jl < 0 || jl > n-1) {               Input guess not useful. Go immediately to bisec-
        jl=0;                                   tion.
        ju=n-1;
    } else {
        if (x >= xx[jl] == ascnd) {         Hunt up:
            for (;;) {
                ju = jl + inc;
                if (ju >= n-1) { ju = n-1; break;}          Off end of table.
                else if (x < xx[ju] == ascnd) break;        Found bracket.
                else {                      Not done, so double the increment and try again.
                    jl = ju;
                    inc += inc;
                }
            }
        } else {                            Hunt down:
            ju = jl;
            for (;;) {
                jl = jl - inc;
                if (jl <= 0) { jl = 0; break;}              Off end of table.
                else if (x >= xx[jl] == ascnd) break;       Found bracket.
                else {                      Not done, so double the increment and try again.
                    ju = jl;
                    inc += inc;
```

```
            }
        }
    }
}
while (ju-jl > 1) {                    Hunt is done, so begin the final bisection phase:
    jm = (ju+jl) >> 1;
    if (x >= xx[jm] == ascnd)
        jl=jm;
    else
        ju=jm;
}
cor = abs(jl-jsav) > dj ? 0 : 1;       Decide whether to use hunt or locate next
jsav = jl;                                 time.
return MAX(0,MIN(n-mm,jl-((mm-2)>>1)));
}
```

The methods `locate` and `hunt` each update the boolean variable `cor` in the base class, indicating whether consecutive calls seem correlated. That variable is then used by `interp` to decide whether to use `locate` or `hunt` on the next call. This is all invisible to the user, of course.

3.1.2 Example: Linear Interpolation

You may think that, at this point, we have wandered far from the subject of interpolation methods. To show that we are actually on track, here is a class that efficiently implements piecewise linear interpolation.

interp_linear.h

```
struct Linear_interp : Base_interp
Piecewise linear interpolation object. Construct with x and y vectors, then call interp for
interpolated values.
{
    Linear_interp(VecDoub_I &xv, VecDoub_I &yv)
        : Base_interp(xv,&yv[0],2)  {}
    Doub rawinterp(Int j, Doub x) {
        if (xx[j]==xx[j+1]) return yy[j];      Table is defective, but we can recover.
        else return yy[j] + ((x-xx[j])/(xx[j+1]-xx[j]))*(yy[j+1]-yy[j]);
    }
};
```

You construct a linear interpolation object by declaring an instance with your filled vectors of abscissas x_i and function values $y_i = f(x_i)$.

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

Behind the scenes, the base class constructor is called with $M = 2$ because linear interpolation uses just the two points bracketing a value. Also, pointers to the data are saved. (You must ensure that the vectors `xx` and `yy` don't go out of scope while `myfunc` is in use.)

When you want an interpolated value, it's as simple as

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

If you have several functions that you want to interpolate, you declare a separate instance of `Linear_interp` for each one.

We will now use the same interface for more advanced interpolation methods.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.2 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points there is a unique quadratic. *Et cetera.* The interpolating polynomial of degree $M-1$ through the M points $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_{M-1} = f(x_{M-1})$ is given explicitly by Lagrange's classical formula,

$$\begin{aligned} P(x) = {} & \frac{(x-x_1)(x-x_2)\ldots(x-x_{M-1})}{(x_0-x_1)(x_0-x_2)\ldots(x_0-x_{M-1})} y_0 \\ & + \frac{(x-x_0)(x-x_2)\ldots(x-x_{M-1})}{(x_1-x_0)(x_1-x_2)\ldots(x_1-x_{M-1})} y_1 + \cdots \\ & + \frac{(x-x_0)(x-x_1)\ldots(x-x_{M-2})}{(x_{M-1}-x_0)(x_{M-1}-x_1)\ldots(x_{M-1}-x_{M-2})} y_{M-1} \end{aligned} \tag{3.2.1}$$

There are M terms, each a polynomial of degree $M-1$ and each constructed to be zero at all of the x_i's 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_0 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_0, y_0); so $P_0 = y_0$. Likewise define $P_1, P_2, \ldots, P_{M-1}$. Now let P_{01} be the value at x of the unique polynomial of degree one passing through both (x_0, y_0) and (x_1, y_1). Likewise $P_{12}, P_{23}, \ldots, P_{(M-2)(M-1)}$. Similarly, for higher-order polynomials, up to $P_{012\ldots(M-1)}$, which is the value of the unique interpolating polynomial through all M 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 $M = 4$,

$$\begin{array}{lllll} x_0: & y_0 = P_0 & & & \\ & & P_{01} & & \\ x_1: & y_1 = P_1 & & P_{012} & \\ & & P_{12} & & P_{0123} \\ x_2: & y_2 = P_2 & & P_{123} & \\ & & P_{23} & & \\ x_3: & y_3 = P_3 & & & \end{array} \tag{3.2.2}$$

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,"

$$P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}} \tag{3.2.3}$$

This recurrence works because the two parents already agree at points $x_{i+1} \ldots x_{i+m-1}$.

An improvement on the recurrence (3.2.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \ldots, M-1$)

$$\begin{aligned} C_{m,i} &\equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)} \end{aligned} \tag{3.2.4}$$

Then one can easily derive from (3.2.3) the relations

$$\begin{aligned} D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\ C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \end{aligned} \tag{3.2.5}$$

At each level m, the C's and D's are the corrections that make the interpolation one order higher. The final answer $P_{0\ldots(M-1)}$ 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 the class implementing polynomial interpolation or extrapolation. All of its "support infrastructure" is in the base class `Base_interp`. It needs only to provide a `rawinterp` method that contains Neville's algorithm.

interp_1d.h

```
struct Poly_interp : Base_interp
Polynomial interpolation object. Construct with x and y vectors, and the number M of points
to be used locally (polynomial order plus one), then call interp for interpolated values.
{
    Doub dy;
    Poly_interp(VecDoub_I &xv, VecDoub_I &yv, Int m)
        : Base_interp(xv,&yv[0],m), dy(0.) {}
    Doub rawinterp(Int jl, Doub x);
};

Doub Poly_interp::rawinterp(Int jl, Doub x)
Given a value x, and using pointers to data xx and yy, this routine returns an interpolated
value y, and stores an error estimate dy. The returned value is obtained by mm-point polynomial
interpolation on the subrange xx[jl..jl+mm-1].
{
    Int i,m,ns=0;
    Doub y,den,dif,dift,ho,hp,w;
    const Doub *xa = &xx[jl], *ya = &yy[jl];
    VecDoub c(mm),d(mm);
    dif=abs(x-xa[0]);
    for (i=0;i<mm;i++) {            Here we find the index ns of the closest table entry,
        if ((dift=abs(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<mm;m++) {            For each column of the tableau,
```

```
        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 $y_i = y(x_i)$, $i = 0 \ldots N-1$, focus attention on one particular interval, between x_j and x_{j+1}. Linear interpolation in that interval gives the interpolation formula

$$y = Ay_j + By_{j+1} \qquad (3.3.1)$$

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

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

These are $N-2$ linear equations in the N unknowns $y_i'', i = 0, \ldots, N-1$. 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.

The user interface differs from previous ones only in the addition of two constructor arguments, used to set the values of the first derivatives at the endpoints, y'_0 and y'_{N-1}. These are coded with default values that signal that you want a natural spline, so they can be omitted in most situations. Both constructors invoke `sety2` to do the actual work of computing, and storing, the second derivatives.

interp_1d.h

```
void Spline_interp::sety2(const Doub *xv, const Doub *yv, Doub yp1, Doub ypn)
```

This routine stores an array `y2[0..n-1]` with second derivatives of the interpolating function at the tabulated points pointed to by `xv`, using function values pointed to by `yv`. If `yp1` and/or `ypn` are equal to 1×10^{99} or larger, the routine is signaled to set the corresponding boundary condition for a natural spline, with zero second derivative on that boundary; otherwise, they are the values of the first derivatives at the endpoints.

```
{
    Int i,k;
    Doub p,qn,sig,un;
    Int n=y2.size();
    VecDoub u(n-1);
    if (yp1 > 0.99e99)                  The lower boundary condition is set either to be "natural"
        y2[0]=u[0]=0.0;
    else {                              or else to have a specified first derivative.
        y2[0] = -0.5;
        u[0]=(3.0/(xv[1]-xv[0]))*((yv[1]-yv[0])/(xv[1]-xv[0])-yp1);
    }
    for (i=1;i<n-1;i++) {               This is the decomposition loop of the tridiagonal algorithm. y2 and u are used for temporary storage of the decomposed factors.
        sig=(xv[i]-xv[i-1])/(xv[i+1]-xv[i-1]);
        p=sig*y2[i-1]+2.0;
        y2[i]=(sig-1.0)/p;
        u[i]=(yv[i+1]-yv[i])/(xv[i+1]-xv[i]) - (yv[i]-yv[i-1])/(xv[i]-xv[i-1]);
        u[i]=(6.0*u[i]/(xv[i+1]-xv[i-1])-sig*u[i-1])/p;
    }
    if (ypn > 0.99e99)                  The upper boundary condition is set either to be "natural"
        qn=un=0.0;
    else {                              or else to have a specified first derivative.
        qn=0.5;
        un=(3.0/(xv[n-1]-xv[n-2]))*(ypn-(yv[n-1]-yv[n-2])/(xv[n-1]-xv[n-2]));
    }
    y2[n-1]=(un-qn*u[n-2])/(qn*y2[n-2]+1.0);
    for (k=n-2;k>=0;k--)                This is the backsubstitution loop of the tridiagonal algorithm.
        y2[k]=y2[k]*y2[k+1]+u[k];
}
```

Note that, unlike the previous object `Poly_interp`, `Spline_interp` stores data that depend on the contents of your array of y_i's at its time of creation — a whole vector `y2`. Although we didn't point it out, the previous interpolation object actually allowed the misuse of altering the contents of their x and y arrays on the fly (as long as the lengths didn't change). If you do that with `Spline_interp`, you'll get definitely wrong answers!

The required `rawinterp` method, never called directly by the users, uses the stored `y2` and implements equation (3.3.3):

interp_1d.h

```
Doub Spline_interp::rawinterp(Int jl, Doub x)
```

Given a value `x`, and using pointers to data `xx` and `yy`, and the stored vector of second derivatives `y2`, this routine returns the cubic spline interpolated value `y`.

```
{
    Int klo=jl,khi=jl+1;
    Doub y,h,b,a;
    h=xx[khi]-xx[klo];
    if (h == 0.0) throw("Bad input to routine splint");     The xa's must be distinct.
    a=(xx[khi]-x)/h;
```

```
    b=(x-xx[klo])/h;                    Cubic spline polynomial is now evaluated.
    y=a*yy[klo]+b*yy[khi]+((a*a*a-a)*y2[klo]
        +(b*b*b-b)*y2[khi])*(h*h)/6.0;
    return y;
}
```

Typical use looks like this:

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

and then, as often as you like,

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

Note that no error estimate is available.

CITED REFERENCES AND FURTHER READING:

De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer).

Ueberhuber, C.W. 1997, *Numerical Computation: Methods, Software, and Analysis*, vol. 1 (Berlin: Springer), Chapter 9.

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. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.4.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §3.8.

3.4 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 $R_{i(i+1)\ldots(i+m)}$ a rational function passing through the $m+1$ points $(x_i, y_i), \ldots, (x_{i+m}, y_{i+m})$. More explicitly, suppose

$$R_{i(i+1)\ldots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{3.4.1}$$

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

$$m + 1 = \mu + \nu + 1 \tag{3.4.2}$$

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 (3.4.1). 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 (3.4.1) 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 that performs rational function extrapolation on tabulated data. A tableau like that of equation (3.2.2) 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 the numerator and denominator equal (if m is even) or with the degree of the denominator larger by one (if m is odd; cf. equation 3.4.2 above). For the derivation of the algorithm, refer to [1]. The algorithm is summarized by a recurrence relation exactly analogous to equation (3.2.3) for polynomial approximation:

$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} + \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1} \tag{3.4.3}$$

This recurrence generates the rational functions through $m+1$ points from the ones through m and (the term $R_{(i+1)\ldots(i+m-1)}$ in equation 3.4.3) $m-1$ points. It is started with

$$R_i = y_i \tag{3.4.4}$$

and with

$$R \equiv [R_{i(i+1)\ldots(i+m)} \quad \text{with} \quad m = -1] = 0 \tag{3.4.5}$$

Now, exactly as in equations (3.2.4) and (3.2.5) above, we can convert the recurrence (3.4.3) to one involving only the small differences

$$\begin{aligned} C_{m,i} &\equiv R_{i\ldots(i+m)} - R_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv R_{i\ldots(i+m)} - R_{(i+1)\ldots(i+m)} \end{aligned} \tag{3.4.6}$$

Note that these satisfy the relation

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \tag{3.4.7}$$

which is useful in proving the recurrences

$$D_{m+1,i} = \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}}$$
$$C_{m+1,i} = \frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}} \qquad (3.4.8)$$

The class for rational function interpolation is identical to that for polynomial interpolation in every way, except, of course, for the different method implemented in `rawinterp`. See the end of §3.2 for usage. Plausible values for M are in the range 4 to 7.

interp_1d.h

```
struct Rat_interp : Base_interp
```

Diagonal rational function interpolation object. Construct with **x** and **y** vectors, and the number `m` of points to be used locally, then call `interp` for interpolated values.

```
{
    Doub dy;
    Rat_interp(VecDoub_I &xv, VecDoub_I &yv, Int m)
        : Base_interp(xv,&yv[0],m), dy(0.) {}
    Doub rawinterp(Int jl, Doub x);
};

Doub Rat_interp::rawinterp(Int jl, Doub x)
```

Given a value `x`, and using pointers to data `xx` and `yy`, this routine returns an interpolated value `y`, and stores an error estimate `dy`. The returned value is obtained by `mm`-point diagonal rational function interpolation on the subrange `xx[jl..jl+mm-1]`.

```
{
    const Doub TINY=1.0e-99;          A small number.
    Int m,i,ns=0;
    Doub y,w,t,hh,h,dd;
    const Doub *xa = &xx[jl], *ya = &yy[jl];
    VecDoub c(mm),d(mm);
    hh=abs(x-xa[0]);
    for (i=0;i<mm;i++) {
        h=abs(x-xa[i]);
        if (h == 0.0) {
            dy=0.0;
            return ya[i];
        } 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<mm;m++) {
        for (i=0;i<mm-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) throw("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+1) < (mm-m) ? c[ns+1] : d[ns--]));
    }
    return y;
}
```

3.4.1 Barycentric Rational Interpolation

Suppose one tries to use the above algorithm to construct a global approximation on the entire table of values using all the given nodes $x_0, x_1, \ldots, x_{N-1}$. One potential drawback is that the approximation can have poles inside the interpolation interval where the denominator in (3.4.1) vanishes, even if the original function has no poles there. An even greater (related) hazard is that we have allowed the order of the approximation to grow to $N-1$, probably much too large.

An alternative algorithm can be derived [2] that has no poles anywhere on the real axis, and that allows the actual order of the approximation to be specified to be any integer $d < N$. The trick is to make the degree of both the numerator and the denominator in equation (3.4.1) be $N-1$. This requires that the p's and the q's not be independent, so that equation (3.4.2) no longer holds.

The algorithm utilizes the *barycentric form* of the rational interpolant

$$R(x) = \frac{\displaystyle\sum_{i=0}^{N-1} \frac{w_i}{x - x_i} y_i}{\displaystyle\sum_{i=0}^{N-1} \frac{w_i}{x - x_i}} \tag{3.4.9}$$

One can show that by a suitable choice of the weights w_i, *every* rational interpolant can be written in this form, and that, as a special case, so can polynomial interpolants [3]. It turns out that this form has many nice numerical properties. Barycentric rational interpolation competes very favorably with splines: its error is often smaller, and the resulting approximation is infinitely smooth (unlike splines).

Suppose we want our rational interpolant to have approximation order d, i.e., if the spacing of the points is $O(h)$, the error is $O(h^{d+1})$ as $h \to 0$. Then the formula for the weights is

$$w_k = \sum_{\substack{i=k-d \\ 0 \le i < N-d}}^{k} (-1)^k \prod_{\substack{j=i \\ j \ne k}}^{i+d} \frac{1}{x_k - x_j} \tag{3.4.10}$$

For example,

$$\begin{aligned} w_k &= (-1)^k, && d = 0 \\ w_k &= (-1)^{k-1} \left[\frac{1}{x_k - x_{k-1}} + \frac{1}{x_{k+1} - x_k} \right], && d = 1 \end{aligned} \tag{3.4.11}$$

In the last equation, you omit the terms in w_0 and w_{N-1} that refer to out-of-range values of x_k.

Here is a routine that implements barycentric rational interpolation. Given a set of N nodes and a desired order d, with $d < N$, it first computes the weights w_k. Then subsequent calls to `interp` evaluate the interpolant using equation (3.4.9). Note that the parameter `jl` of `rawinterp` is not used, since the algorithm is designed to construct an approximation on the entire interval at once.

The workload to construct the weights is of order $O(Nd)$ operations. For small d, this is not too different from splines. Note, however, that the workload for each subsequent interpolated value is $O(N)$, not $O(d)$ as for splines.

interp_1d.h

```
struct BaryRat_interp : Base_interp
```

Barycentric rational interpolation object. After constructing the object, call `interp` for interpolated values. Note that no error estimate `dy` is calculated.

```
{
    VecDoub w;
    Int d;
    BaryRat_interp(VecDoub_I &xv, VecDoub_I &yv, Int dd);
    Doub rawinterp(Int jl, Doub x);
    Doub interp(Doub x);
};

BaryRat_interp::BaryRat_interp(VecDoub_I &xv, VecDoub_I &yv, Int dd)
        : Base_interp(xv,&yv[0],xv.size()), w(n), d(dd)
```

Constructor arguments are $\mathbf{x}$ and $\mathbf{y}$ vectors of length `n`, and order d of desired approximation.

```
{
    if (n<=d) throw("d too large for number of points in BaryRat_interp");
    for (Int k=0;k<n;k++) {
```

Compute weights from equation (3.4.10).

```
        Int imin=MAX(k-d,0);
        Int imax = k >= n-d ? n-d-1 : k;
        Doub temp = imin & 1 ? -1.0 : 1.0;
        Doub sum=0.0;
        for (Int i=imin;i<=imax;i++) {
            Int jmax=MIN(i+d,n-1);
            Doub term=1.0;
            for (Int j=i;j<=jmax;j++) {
                if (j==k) continue;
                term *= (xx[k]-xx[j]);
            }
            term=temp/term;
            temp=-temp;
            sum += term;
        }
        w[k]=sum;
    }
}
Doub BaryRat_interp::rawinterp(Int jl, Doub x)
```

Use equation (3.4.9) to compute the barycentric rational interpolant. Note that `jl` is not used since the approximation is global; it is included only for compatibility with `Base_interp`.

```
{
    Doub num=0,den=0;
    for (Int i=0;i<n;i++) {
        Doub h=x-xx[i];
        if (h == 0.0) {
            return yy[i];
        } else {
            Doub temp=w[i]/h;
            num += temp*yy[i];
            den += temp;
        }
    }
    return num/den;
}
Doub BaryRat_interp::interp(Doub x) {
```

No need to invoke `hunt` or `locate` since the interpolation is global, so override `interp` to simply call `rawinterp` directly with a dummy value of `jl`.

```
    return rawinterp(1,x);
}
```

It is wise to start with small values of d before trying larger values.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §2.2.[1]

Floater, M.S., and Hormann, K. 2006+, "Barycentric Rational Interpolation with No Poles and High Rates of Approximation," at `http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0606hormann.pdf`.[2]

Berrut, J.-P., and Trefethen, L.N. 2004, "Barycentric Lagrange Interpolation," *SIAM Review*, vol. 46, pp. 501–517.[3]

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.

3.5 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 §5.1), 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.

Please be certain, however, 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 §14.9.) 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

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_{N-1} x^{N-1} \tag{3.5.1}$$

then the c_i's are required to satisfy the linear equation

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{3.5.2}$$

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 §3.2, but only difficulty in finding *coefficients*.

Like the routine in §2.8, the following is due to G.B. Rybicki.

polcoef.h

```
void polcoe(VecDoub_I &x, VecDoub_I &y, VecDoub_O &cof)
```

Given arrays `x[0..n-1]` and `y[0..n-1]` containing a tabulated function $y_i = f(x_i)$, this routine returns an array of coefficients `cof[0..n-1]`, such that $y_i = \sum_{j=0}^{n-1} \mathtt{cof}_j x_i^j$

```
{
    Int k,j,i,n=x.size();
    Doub phi,ff,b;
    VecDoub s(n);
    for (i=0;i<n;i++) s[i]=cof[i]=0.0;
    s[n-1]= -x[0];
    for (i=1;i<n;i++) {
        for (j=n-1-i;j<n-1;j++)
            s[j] -= x[i]*s[j+1];
        s[n-1] -= x[i];
    }
    for (j=0;j<n;j++) {
        phi=n;
        for (k=n-1;k>0;k--)
            phi=k*s[k]+x[j]*phi;
        ff=y[j]/phi;
        b=1.0;
        for (k=n-1;k>=0;k--) {
            cof[k] += b*ff;
            b=s[k]+x[j]*b;
        }
    }
}
```

Coefficients s_i of the master polynomial $P(x)$ are found by recurrence.

The quantity `phi` $= \prod_{j \neq k}(x_j - x_k)$ is found as a derivative of $P(x_j)$.

Coefficients of polynomials in each term of the Lagrange formula are found by synthetic division of $P(x)$ by $(x - x_j)$. The solution c_k is accumulated.

3.5.1 Another Method

Another technique is to make use of the function value interpolation routine already given (`polint`; §3.2). 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.

polcoef.h

```
void polcof(VecDoub_I &xa, VecDoub_I &ya, VecDoub_O &cof)
```

Given arrays `xa[0..n-1]` and `ya[0..n-1]` containing a tabulated function $\mathtt{ya}_i = f(\mathtt{xa}_i)$, this routine returns an array of coefficients `cof[0..n-1]`, such that $\mathtt{ya}_i = \sum_{j=0}^{n-1} \mathtt{cof}_j \, \mathtt{xa}_i^j$.

```
{
    Int k,j,i,n=xa.size();
    Doub xmin;
    VecDoub x(n),y(n);
    for (j=0;j<n;j++) {
        x[j]=xa[j];
        y[j]=ya[j];
    }
    for (j=0;j<n;j++) {                          Fill a temporary vector whose size
        VecDoub x_t(n-j),y_t(n-j);                   decreases as each coefficient is
        for (k=0;k<n-j;k++) {                        found.
            x_t[k]=x[k];
            y_t[k]=y[k];
        }
        Poly_interp interp(x,y,n-j);
        cof[j] = interp.rawinterp(0,0.);         Extrapolate to x = 0.
        xmin=1.0e99;
        k = -1;
        for (i=0;i<n-j;i++) {                    Find the remaining x_i of smallest
            if (abs(x[i]) < xmin) {                  absolute value
                xmin=abs(x[i]);
                k=i;
            }
            if (x[i] != 0.0)                     (meanwhile reducing all the terms)
                y[i]=(y[i]-cof[j])/x[i];
        }
        for (i=k+1;i<n-j;i++) {                  and eliminate it.
            y[i-1]=y[i];
            x[i-1]=x[i];
        }
    }
}
```

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. In this case, 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*; reprinted 1994 (New York: Dover), §5.2.

3.6 Interpolation on a Grid in Multidimensions

In multidimensional interpolation, we seek an estimate of a function of more than one independent variable, $y(x_1, x_2, \ldots, x_n)$. The Great Divide is, Are we given a complete set of tabulated values on an n-dimensional grid? Or, do we know function values only on some scattered set of points in the n-dimensional space? In one dimension, the question never arose, because any set of x_i's could once sorted into ascending order, could be viewed as a valid one-dimensional grid (regular spacing not being a requirement).

As the number of dimensions n gets large, maintaining a full grid becomes rapidly impractical, because of the explosion in the number of gridpoints. Methods that work with scattered data, to be considered in §3.7, then become the methods of choice. Don't, however, make the mistake of thinking that such methods are intrinsically more accurate than grid methods. In general they are less accurate. Like the proverbial three-legged dog, they are remarkable because they work at all, not because they work, necessarily, well!

Both kinds of methods are practical in two dimensions, and some other kinds as well. For example, *finite element methods*, of which *triangulation* is the most common, find ways to impose some kind of geometrically regular structure on scattered points, and then use that structure for interpolation. We will treat two-dimensional interpolation by triangulation in detail in §21.6; that section should be considered as a continuation of the discussion here.

In the remainder of this section, we consider only the case of interpolating on a grid, and we implement in code only the (most common) case of two dimensions. All of the methods given generalize to three dimensions in an obvious way. When we implement methods for scattered data, in §3.7, the treatment will be for general n.

In two dimensions, we imagine that we are given a matrix of functional values y_{ij} with $i = 0, \ldots, M-1$ and $j = 0, \ldots, N-1$. We are also given an array of x_1 values x_{1i}, and an array of x_2 values x_{2j}, with i and j as just stated. The relation of these input quantities to an underlying function $y(x_1, x_2)$ is just

$$y_{ij} = y(x_{1i}, x_{2j}) \tag{3.6.1}$$

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 0 to 3, counterclockwise starting from the lower left. More precisely, if

$$\begin{aligned} x_{1i} &\le x_1 \le x_{1(i+1)} \\ x_{2j} &\le x_2 \le x_{2(j+1)} \end{aligned} \tag{3.6.2}$$

defines values of i and j, then

$$\begin{aligned} y_0 &\equiv y_{ij} \\ y_1 &\equiv y_{(i+1)j} \\ y_2 &\equiv y_{(i+1)(j+1)} \\ y_3 &\equiv y_{i(j+1)} \end{aligned} \tag{3.6.3}$$

The simplest interpolation in two dimensions is *bilinear interpolation* on the grid square. Its formulas are

$$t \equiv (x_1 - x_{1i})/(x_{1(i+1)} - x_{1i})$$
$$u \equiv (x_2 - x_{2j})/(x_{2(j+1)} - x_{2j}) \tag{3.6.4}$$

(so that t and u each lie between 0 and 1) and

$$y(x_1, x_2) = (1-t)(1-u)y_0 + t(1-u)y_1 + tuy_2 + (1-t)uy_3 \tag{3.6.5}$$

Bilinear interpolation is frequently "close enough for government work." As the interpolating point wanders from grid square to grid square, the interpolated function value changes continuously. However, the gradient of the interpolated function changes discontinuously at the boundaries of each grid square.

We can easily implement an object for bilinear interpolation by grabbing pieces of "machinery" from our one-dimensional interpolation classes:

interp_2d.h

```
struct Bilin_interp {
```
Object for bilinear interpolation on a matrix. Construct with a vector of x_1 values, a vector of x_2 values, and a matrix of tabulated function values y_{ij}. Then call `interp` for interpolated values.
```
    Int m,n;
    const MatDoub &y;
    Linear_interp x1terp, x2terp;

    Bilin_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym)
        : m(x1v.size()), n(x2v.size()), y(ym),
        x1terp(x1v,x1v), x2terp(x2v,x2v) {}
```
Construct dummy 1-dim interpolations for their `locate` and `hunt` functions.
```
    Doub interp(Doub x1p, Doub x2p) {
        Int i,j;
        Doub yy, t, u;
        i = x1terp.cor ? x1terp.hunt(x1p) : x1terp.locate(x1p);
        j = x2terp.cor ? x2terp.hunt(x2p) : x2terp.locate(x2p);
```
Find the grid square.
```
        t = (x1p-x1terp.xx[i])/(x1terp.xx[i+1]-x1terp.xx[i]);
        u = (x2p-x2terp.xx[j])/(x2terp.xx[j+1]-x2terp.xx[j]);
```
Interpolate.
```
        yy = (1.-t)*(1.-u)*y[i][j] + t*(1.-u)*y[i+1][j]
            + (1.-t)*u*y[i][j+1] + t*u*y[i+1][j+1];
        return yy;
    }
};
```

Here we declare two instances of `Linear_interp`, one for each direction, and use them merely to do the bookkeeping on the arrays x_{1i} and x_{2_j} — in particular, to provide the "intelligent" table-searching mechanisms that we have come to rely on. (The second occurrence of `x1v` and `x2v` in the constructors is just a placeholder; there are not really any one-dimensional "y" arrays.)

Usage of `Bilin_interp` is just what you'd expect:

```
    Int m=..., n=...;
    MatDoub yy(m,n);
    VecDoub x1(m), x2(n);
    ...
    Bilin_interp myfunc(x1,x2,yy);
```

followed (any number of times) by

```
Doub x1,x2,y;
...
y = myfunc.interp(x1,x2);
```

Bilinear interpolation is a good place to start, in two dimensions, unless you positively know that you need something fancier.

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.

3.6.1 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 $m \times n$ 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 (x_{1i}, x_2), with m values of i. Finally, we do a last interpolation in the x_1 direction to get the answer.

Again using the previous one-dimensional machinery, this can all be coded very concisely as

interp_2d.h

```
struct Poly2D_interp {
```

Object for two-dimensional polynomial interpolation on a matrix. Construct with a vector of x_1 values, a vector of x_2 values, a matrix of tabulated function values y_{ij}, and integers to specify the number of points to use locally in each direction. Then call `interp` for interpolated values.

```
    Int m,n,mm,nn;
    const MatDoub &y;
    VecDoub yv;
    Poly_interp x1terp, x2terp;

    Poly2D_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym,
        Int mp, Int np) : m(x1v.size()), n(x2v.size()),
        mm(mp), nn(np), y(ym), yv(m),
        x1terp(x1v,yv,mm), x2terp(x2v,x2v,nn) {}
```

Dummy 1-dim interpolations for their `locate` and `hunt` functions.

```
    Doub interp(Doub x1p, Doub x2p) {
        Int i,j,k;
        i = x1terp.cor ? x1terp.hunt(x1p) : x1terp.locate(x1p);
        j = x2terp.cor ? x2terp.hunt(x2p) : x2terp.locate(x2p);
        Find grid block.
        for (k=i;k<i+mm;k++) {
            x2terp.yy = &y[k][0];
            yv[k] = x2terp.rawinterp(j,x2p);
        }
        return x1terp.rawinterp(i,x1p);
    }
};
```

`mm` interpolations in the x_2 direction.

A final interpolation in the x_1 direction.

The user interface is the same as for `Bilin_interp`, except that the constructor has two additional arguments that specify the number of points (order plus one) to be used locally in, respectively, the x_1 and x_2 interpolations. Typical values will be in the range 3 to 7.

Code stylists won't like some of the details in `Poly2D_interp` (see discussion in §3.1 immediately following `Base_interp`). As we loop over the rows of the sub-block, we reach into the guts of `x2terp` and repoint its `yy` array to a row of our y matrix. Further, we alter the contents of the array `yv`, for which `x1terp` has stored a pointer, on the fly. None of this is particularly dangerous as long as we control the implementations in both `Base_interp` and `Poly2D_interp`; and it makes for a very efficient implementation. You should view these two classes as not just (implicitly) `friend` classes, but as *really intimate* friends.

3.6.2 Higher Order for Smoothness: Bicubic Spline

A favorite technique for obtaining smoothness in two-dimensional interpolation is the *bicubic spline*. To set up a bicubic spline, you (one time) construct M one-dimensional splines across the rows of the two-dimensional matrix of function values. Then, for each desired interpolated value you proceed as follows: (1) Perform M spline interpolations to get a vector of values $y(x_{1i}, x_2), i = 0, \ldots, M-1$ (2) Construct a one-dimensional spline through those values. (3) Finally, spline-interpolate to the desired value $y(x_1, x_2)$.

If this sounds like a lot of work, well, yes, it is. The one-time setup work scales as the table size $M \times N$, while the work per interpolated value scales roughly as $M \log M + N$, both with pretty hefty constants in front. This is the price that you pay for the desirable characteristics of splines that derive from their nonlocality. For tables with modest M and N, less than a few hundred, say, the cost is usually tolerable. If it's not, then fall back to the previous local methods.

Again a very concise implementation is possible:

interp_2d.h

```
struct Spline2D_interp {
```
Object for two-dimensional cubic spline interpolation on a matrix. Construct with a vector of x_1 values, a vector of x_2 values, and a matrix of tabulated function values y_{ij}. Then call `interp` for interpolated values.
```
    Int m,n;
    const MatDoub &y;
    const VecDoub &x1;
    VecDoub yv;
    NRvector<Spline_interp*> srp;

    Spline2D_interp(VecDoub_I &x1v, VecDoub_I &x2v, MatDoub_I &ym)
        : m(x1v.size()), n(x2v.size()), y(ym), yv(m), x1(x1v), srp(m) {
        for (Int i=0;i<m;i++) srp[i] = new Spline_interp(x2v,&y[i][0]);
        Save an array of pointers to 1-dim row splines.
    }

    ~Spline2D_interp(){
        for (Int i=0;i<m;i++) delete srp[i];        We need a destructor to clean up.
    }

    Doub interp(Doub x1p, Doub x2p) {
        for (Int i=0;i<m;i++) yv[i] = (*srp[i]).interp(x2p);
        Interpolate on each row.
        Spline_interp scol(x1,yv);                  Construct the column spline,
        return scol.interp(x1p);                    and evaluate it.
    }
};
```

The reason for that ugly vector of pointers to `Spline_interp` objects is that we need to initialize each row spline separately, with data from the appropriate row. The user interface is the same as `Bilin_interp`, above.

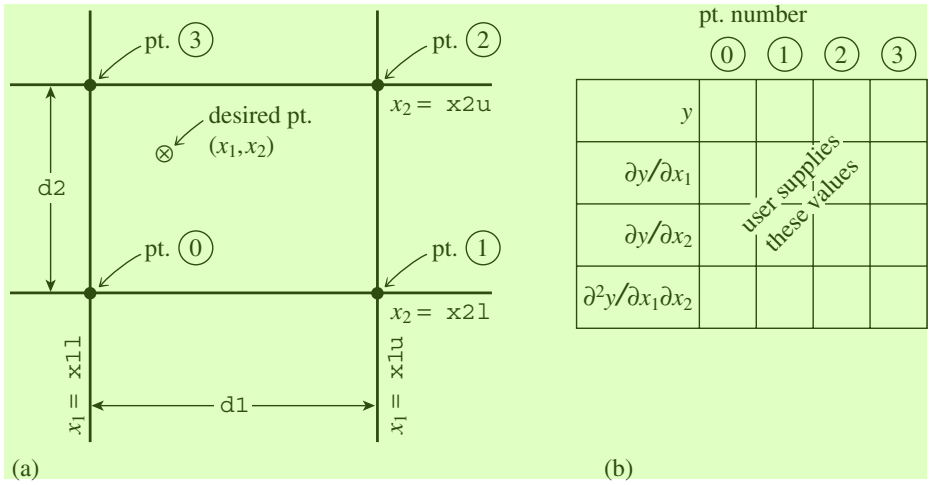

Figure 3.6.1. (a) Labeling of points used in the two-dimensional interpolation routines `bcuint` and `bcucof`. (b) For each of the four points in (a), the user supplies one function value, two first derivatives, and one cross-derivative, a total of 16 numbers.

3.6.3 Higher Order for Smoothness: Bicubic Interpolation

Bicubic interpolation gives the same degree of smoothness as bicubic spline interpolation, but it has the advantage of being a local method. Thus, after you set it up, a function interpolation costs only a constant, plus $\log M + \log N$, to find your place in the table. Unfortunately, this advantage comes with a lot of complexity in coding. Here, we will give only some building blocks for the method, not a complete user interface.

Bicubic splines are in fact a special case of bicubic interpolation. In the general case, however, we leave the values of all derivatives at the grid points as freely specifiable. You, the user, can specify them *any way you want*. In other words, you 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 $\partial^2 y/\partial x_1 \partial x_2 \equiv y_{,12}$ (see Figure 3.6.1). 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 16 quantities c_{ij}, $i, j = 0, \ldots, 3$ 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 that, 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:

$$
\begin{aligned}
y(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} c_{ij} t^i u^j \\
y_{,1}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} i c_{ij} t^{i-1} u^j (dt/dx_1) \\
y_{,2}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} j c_{ij} t^i u^{j-1} (du/dx_2) \\
y_{,12}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} ij c_{ij} t^{i-1} u^{j-1} (dt/dx_1)(du/dx_2)
\end{aligned}
\tag{3.6.6}
$$

where t and u are again given by equation (3.6.4).

interp_2d.h

```
void bcucof(VecDoub_I &y, VecDoub_I &y1, VecDoub_I &y2, VecDoub_I &y12,
    const Doub d1, const Doub d2, MatDoub_O &c) {
```

Given arrays y[0..3], y1[0..3], y2[0..3], and y12[0..3], 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[0..3][0..3] that is used by routine bcuint for bicubic interpolation.

```
static Int wt_d[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;
Doub xx,d1d2=d1*d2;
VecDoub cl(16),x(16);
static MatInt wt(16,16,wt_d);
for (i=0;i<4;i++) {
```

Pack a temporary vector x.

```
        x[i]=y[i];
        x[i+4]=y1[i]*d1;
        x[i+8]=y2[i]*d2;
        x[i+12]=y12[i]*d1d2;
    }
    for (i=0;i<16;i++) {          Matrix-multiply by the stored table.
        xx=0.0;
        for (k=0;k<16;k++) xx += wt[i][k]*x[k];
        cl[i]=xx;
    }
    l=0;
    for (i=0;i<4;i++)             Unpack the result into the output table.
        for (j=0;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:

interp_2d.h

```
void bcuint(VecDoub_I &y, VecDoub_I &y1, VecDoub_I &y2, VecDoub_I &y12,
    const Doub x1l, const Doub x1u, const Doub x2l, const Doub x2u,
    const Doub x1, const Doub x2, Doub &ansy, Doub &ansy1, Doub &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`.

```
    Int i;
    Doub t,u,d1=x1u-x1l,d2=x2u-x2l;
    MatDoub c(4,4);
    bcucof(y,y1,y2,y12,d1,d2,c);        Get the c's.
    if (x1u == x1l || x2u == x2l)
        throw("Bad input in routine bcuint");
    t=(x1-x1l)/d1;                      Equation (3.6.4).
    u=(x2-x2l)/d2;
    ansy=ansy2=ansy1=0.0;
    for (i=3;i>=0;i--) {                Equation (3.6.6).
        ansy=t*ansy+((c[i][3]*u+c[i][2])*u+c[i][1])*u+c[i][0];
        ansy2=t*ansy2+(3.0*c[i][3]*u+2.0*c[i][2])*u+c[i][1];
        ansy1=u*ansy1+(3.0*c[3][i]*t+2.0*c[2][i])*t+c[1][i];
    }
    ansy1 /= d1;
    ansy2 /= d2;
}
```

You can combine the best features of bicubic interpolation and bicubic splines by using splines to compute values for the necessary derivatives at the grid points, storing these values, and then using bicubic interpolation, with an efficient table-searching method, for the actual function interpolations. Unfortunately this is beyond our scope here.

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.

Kinahan, B.F., and Harm, R. 1975, "Chemical Composition and the Hertzsprung Gap," *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); reprinted 2003 (New York: Dover), §7.7.

3.7 Interpolation on Scattered Data in Multidimensions

We now leave behind, if with some trepidation, the orderly world of regular grids. Courage is required. We are given an arbitrarily scattered set of N data points $(\mathbf{x}_i, y_i), i = 0, \ldots, N-1$. Here $\mathbf{x}_i$ denotes an n-dimensional vector of independent variables, $(x_{1i}, x_{2i}, \ldots, x_{ni})$, and y_i is the value of the function at that point.

In this section we discuss two of the most widely used *general* methods for this problem, *radial basis function (RBF)* interpolation, and *kriging*. Both of these methods are expensive. By that we mean that they require $O(N^3)$ operations to initially digest a set of data points, followed by $O(N)$ operations for each interpolated value. Kriging is also able to supply an error estimate — but at the rather high cost of $O(N^2)$ per value. Shepard interpolation, discussed below, is a variant of RBF that at least avoids the $O(N^3)$ initial work; otherwise these workloads effectively limit the usefulness of these general methods to values of $N \lesssim 10^4$. It is therefore worthwhile for you to consider whether you have any other options. Two of these are

- If n is not too large (meaning, usually, $n = 2$), and if the data points are fairly dense, then consider triangulation, discussed in §21.6. Triangulation is an example of a finite element method. Such methods construct some semblance of geometric regularity and then exploit that construction to advantage. Mesh generation is a closely related subject.

- If your accuracy goals will tolerate it, consider moving each data point to the nearest point on a regular Cartesian grid and then using Laplace interpolation (§3.8) to fill in the rest of the grid points. After that, you can interpolate on the grid by the methods of §3.6. You will need to compromise between making the grid very fine (to minimize the error introduced when you move the points) and the compute time workload of the Laplace method.

If neither of these options seem attractive, and you can't think of another one that is, then try one or both of the two methods that we now discuss. RBF interpolation is probably the more widely used of the two, but kriging is our personal favorite. Which works better will depend on the details of your problem.

The related, but easier, problem of *curve interpolation* in multidimensions is discussed at the end of this section.

3.7.1 Radial Basis Function Interpolation

The idea behind RBF interpolation is very simple: Imagine that every known point j "influences" its surroundings the same way in all directions, according to

some assumed functional form $\phi(r)$ the radial basis function — that is a function only of radial distance $r = |\mathbf{x} - \mathbf{x}_j|$ from the point. Let us try to approximate the interpolating function everywhere by a linear combination of the ϕ's, centered on all the known points,

$$y(\mathbf{x}) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|) \tag{3.7.1}$$

where the w_i's are some unknown set of weights. How do we find these weights? Well, we haven't used the function values y_i yet. The weights are determined by requiring that the interpolation be exact at all the known data points. That is equivalent to solving a set of N linear equations in N unknowns for the w_i's:

$$y_j = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \tag{3.7.2}$$

For many functional forms ϕ, it can be proved, under various general assumptions, that this set of equations is nondegenerate and can be readily solved by, e.g., LU decomposition (§2.3). References [1,2] provide entry to the literature.

A variant on RBF interpolation is *normalized radial basis function (NRBF)* interpolation, in which we require the sum of the basis functions to be unity or, equivalently, replace equations (3.7.1) and (3.7.2) by

$$y(\mathbf{x}) = \frac{\sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|)}{\sum_{i=0}^{N-1} \phi(|\mathbf{x} - \mathbf{x}_i|)} \tag{3.7.3}$$

and

$$y_j \sum_{i=0}^{N-1} \phi(|\mathbf{x}_j - \mathbf{x}_i|) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \tag{3.7.4}$$

Equations (3.7.3) and (3.7.4) arise more naturally from a Bayesian statistical perspective [3]. However, there is no evidence that either the NRBF method is consistently superior to the RBF method, or vice versa. It is easy to implement both methods in the same code, leaving the choice to the user.

As we already mentioned, for N data points the one-time work to solve for the weights by LU decomposition is $O(N^3)$. After that, the cost is $O(N)$ for each interpolation. Thus $N \sim 10^3$ is a rough dividing line (at 2007 desktop speeds) between "easy" and "difficult." If your N is larger, however, don't despair: There are *fast multipole methods*, beyond our scope here, with much more favorable scaling [1,4,5]. Another, much lower-tech, option is to use *Shepard interpolation* discussed later in this section.

Here are a couple of objects that implement everything discussed thus far. `RBF_fn` is a virtual base class whose derived classes will embody different functional forms for `rbf`$(r) \equiv \phi(r)$. `RBF_interp`, via its constructor, digests your data and solves the equations for the weights. The data points $\mathbf{x}_i$ are input as an $N \times n$ matrix, and the code works for any dimension n. A boolean argument `nrbf` inputs whether NRBF is to be used instead of RBF. You call `interp` to get an interpolated function value at a new point $\mathbf{x}$.

interp_rbf.h

```
struct RBF_fn {
Abstract base class template for any particular radial basis function. See specific examples below.
    virtual Doub rbf(Doub r) = 0;
};

struct RBF_interp {
Object for radial basis function interpolation using n points in dim dimensions. Call constructor once, then interp as many times as desired.
    Int dim, n;
    const MatDoub &pts;
    const VecDoub &vals;
    VecDoub w;
    RBF_fn &fn;
    Bool norm;

    RBF_interp(MatDoub_I &ptss, VecDoub_I &valss, RBF_fn &func, Bool nrbf=false)
    : dim(ptss.ncols()), n(ptss.nrows()) , pts(ptss), vals(valss),
    w(n), fn(func), norm(nrbf) {
    Constructor. The n × dim matrix ptss inputs the data points, the vector valss the function values. func contains the chosen radial basis function, derived from the class RBF_fn. The default value of nrbf gives RBF interpolation; set it to 1 for NRBF.
        Int i,j;
        Doub sum;
        MatDoub rbf(n,n);
        VecDoub rhs(n);
        for (i=0;i<n;i++) {                Fill the matrix φ(|r_i − r_j|) and the r.h.s. vector.
            sum = 0.;
            for (j=0;j<n;j++) {
                sum += (rbf[i][j] = fn.rbf(rad(&pts[i][0],&pts[j][0])));
            }
            if (norm) rhs[i] = sum*vals[i];
            else rhs[i] = vals[i];
        }
        LUdcmp lu(rbf);                    Solve the set of linear equations.
        lu.solve(rhs,w);
    }

    Doub interp(VecDoub_I &pt) {
    Return the interpolated function value at a dim-dimensional point pt.
        Doub fval, sum=0., sumw=0.;
        if (pt.size() != dim) throw("RBF_interp bad pt size");
        for (Int i=0;i<n;i++) {            Sum over all tabulated points.
            fval = fn.rbf(rad(&pt[0],&pts[i][0]));
            sumw += w[i]*fval;
            sum += fval;
        }
        return norm ? sumw/sum : sumw;
    }

    Doub rad(const Doub *p1, const Doub *p2) {
    Euclidean distance.
        Doub sum = 0.;
        for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
        return sqrt(sum);
    }
};
```

3.7.2 Radial Basis Functions in General Use

The most often used radial basis function is the *multiquadric* first used by Hardy, circa 1970. The functional form is

$$\phi(r) = (r^2 + r_0^2)^{1/2} \tag{3.7.5}$$

where r_0 is a scale factor that you get to choose. Multiquadrics are said to be less sensitive to the choice of r_0 than some other functional forms.

In general, both for multiquadrics and for other functions, below, r_0 should be larger than the typical separation of points but smaller than the "outer scale" or feature size of the function that you are interpolating. There can be several orders of magnitude difference between the interpolation accuracy with a good choice for r_0, versus a poor choice, so it is definitely worth some experimentation. One way to experiment is to construct an RBF interpolator omitting one data point at a time and measuring the interpolation error at the omitted point.

The *inverse multiquadric*

$$\phi(r) = (r^2 + r_0^2)^{-1/2} \tag{3.7.6}$$

gives results that are comparable to the multiquadric, sometimes better.

It might seem odd that a function and its inverse (actually, reciprocal) work about equally well. The explanation is that what really matters is smoothness, and certain properties of the function's Fourier transform that are not very different between the multiquadric and its reciprocal. The fact that one increases monotonically and the other decreases turns out to be almost irrelevant. However, if you want the extrapolated function to go to zero far from all the data (where an accurate value is impossible anyway), then the inverse multiquadric is a good choice.

The *thin-plate spline* radial basis function is

$$\phi(r) = r^2 \log(r/r_0) \tag{3.7.7}$$

with the limiting value $\phi(0) = 0$ assumed. This function has some physical justification in the energy minimization problem associated with warping a thin elastic plate. There is no indication that it is generally better than either of the above forms, however.

The *Gaussian* radial basis function is just what you'd expect,

$$\phi(r) = \exp\left(-\tfrac{1}{2} r^2 / r_0^2\right) \tag{3.7.8}$$

The interpolation accuracy using Gaussian basis functions can be very sensitive to r_0, and they are often avoided for this reason. However, for smooth functions and with an optimal r_0, very high accuracy can be achieved. The Gaussian also will extrapolate any function to zero far from the data, and it gets to zero quickly.

Other functions are also in use, for example those of Wendland [6]. There is a large literature in which the above choices for basis functions are tested against specific functional forms or experimental data sets [1,2,7]. Few, if any, general recommendations emerge. We suggest that you try the alternatives in the order listed above, starting with multiquadrics, and that you not omit experimenting with different choices of the scale parameters r_0.

The functions discussed are implemented in code as:

interp_rbf.h

```
struct RBF_multiquadric : RBF_fn {
Instantiate this and send to RBF_interp to get multiquadric interpolation.
    Doub r02;
    RBF_multiquadric(Doub scale=1.) : r02(SQR(scale)) {}
    Constructor argument is the scale factor. See text.
    Doub rbf(Doub r) { return sqrt(SQR(r)+r02); }
};

struct RBF_thinplate : RBF_fn {
Same as above, but for thin-plate spline.
    Doub r0;
    RBF_thinplate(Doub scale=1.) : r0(scale) {}
    Doub rbf(Doub r) { return r <= 0. ? 0. : SQR(r)*log(r/r0); }
};

struct RBF_gauss : RBF_fn {
Same as above, but for Gaussian.
    Doub r0;
    RBF_gauss(Doub scale=1.) : r0(scale) {}
    Doub rbf(Doub r) { return exp(-0.5*SQR(r/r0)); }
};

struct RBF_inversemultiquadric : RBF_fn {
Same as above, but for inverse multiquadric.
    Doub r02;
    RBF_inversemultiquadric(Doub scale=1.) : r02(SQR(scale)) {}
    Doub rbf(Doub r) { return 1./sqrt(SQR(r)+r02); }
};
```

Typical use of the objects in this section should look something like this:

```
Int npts=...,ndim=...;
Doub r0=...;
MatDoub pts(npts,ndim);
VecDoub y(npts);
...
RBF_multiquadric multiquadric(r0);
RBF_interp myfunc(pts,y,multiquadric,0);
```

followed by any number of interpolation calls,

```
VecDoub pt(ndim);
Doub val;
...
val = myfunc.interp(pt);
```

3.7.3 Shepard Interpolation

An interesting special case of normalized radial basis function interpolation (equations 3.7.3 and 3.7.4) occurs if the function $\phi(r)$ goes to infinity as $r \to 0$, and is finite (e.g., decreasing) for $r > 0$. In that case it is easy to see that the weights w_i are just equal to the respective function values y_i, and the interpolation formula is simply

$$y(\mathbf{x}) = \frac{\sum_{i=0}^{N-1} y_i \phi(|\mathbf{x} - \mathbf{x}_i|)}{\sum_{i=0}^{N-1} \phi(|\mathbf{x} - \mathbf{x}_i|)} \tag{3.7.9}$$

(with appropriate provision for the limiting case where $\mathbf{x}$ is equal to one of the $\mathbf{x}_i$'s). Note that no solution of linear equations is required. The one-time work is negligible, while the work for each interpolation is $O(N)$, tolerable even for very large N.

Shepard proposed the simple power-law function

$$\phi(r) = r^{-p} \tag{3.7.10}$$

with (typically) $1 < p \lesssim 3$, as well as some more complicated functions with different exponents in an inner and outer region (see [8]). You can see that what is going on is basically interpolation by a nearness-weighted average, with nearby points contributing more strongly than distant ones.

Shepard interpolation is rarely as accurate as the well-tuned application of one of the other radial basis functions, above. On the other hand, it is simple, fast, and often just the thing for quick and dirty applications. It, and variants, are thus widely used.

An implementing object is

interp_rbf.h

```
struct Shep_interp {
Object for Shepard interpolation using n points in dim dimensions. Call constructor once, then interp as many times as desired.
    Int dim, n;
    const MatDoub &pts;
    const VecDoub &vals;
    Doub pneg;

    Shep_interp(MatDoub_I &ptss, VecDoub_I &valss, Doub p=2.)
    : dim(ptss.ncols()), n(ptss.nrows()) , pts(ptss),
    vals(valss), pneg(-p) {}
    Constructor. The n × dim matrix ptss inputs the data points, the vector valss the function values. Set p to the desired exponent. The default value is typical.

    Doub interp(VecDoub_I &pt) {
    Return the interpolated function value at a dim-dimensional point pt.
        Doub r, w, sum=0., sumw=0.;
        if (pt.size() != dim) throw("RBF_interp bad pt size");
        for (Int i=0;i<n;i++) {
            if ((r=rad(&pt[0],&pts[i][0])) == 0.) return vals[i];
            sum += (w = pow(r,pneg));
            sumw += w*vals[i];
        }
        return sumw/sum;
    }

    Doub rad(const Doub *p1, const Doub *p2) {
    Euclidean distance.
        Doub sum = 0.;
        for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
        return sqrt(sum);
    }
};
```

3.7.4 Interpolation by Kriging

Kriging is a technique named for South African mining engineer D.G. Krige. It is basically a form of linear prediction (§13.6), also known in different communities as *Gauss-Markov estimation* or *Gaussian process regression*.

Kriging can be either an interpolation method or a fitting method. The distinction between the two is whether the fitted/interpolated function goes exactly through all the input data points (interpolation), or whether it allows measurement errors to be specified and then "smooths" to get a statistically better predictor that does not

generally go through the data points (does not "honor the data"). In this section we consider only the former case, that is, interpolation. We will return to the latter case in §15.9.

At this point in the book, it is beyond our scope to derive the equations for kriging. You can turn to §13.6 to get a flavor, and look to references [9,10,11] for details. To use kriging, you must be able to estimate the mean square variation of your function $y(\mathbf{x})$ as a function of offset distance $\mathbf{r}$, a so-called *variogram*,

$$v(\mathbf{r}) \sim \frac{1}{2}\left\langle [y(\mathbf{x}+\mathbf{r}) - y(\mathbf{x})]^2 \right\rangle \tag{3.7.11}$$

where the average is over all $\mathbf{x}$ with fixed $\mathbf{r}$. If this seems daunting, don't worry. For interpolation, even very crude variogram estimates work fine, and we will give below a routine to estimate $v(\mathbf{r})$ from your input data points $\mathbf{x}_i$ and $y_i = y(\mathbf{x}_i)$, $i = 0, \ldots, N-1$, automatically. One usually takes $v(\mathbf{r})$ to be a function only of the magnitude $r = |\mathbf{r}|$ and writes it as $v(r)$.

Let v_{ij} denote $v(|\mathbf{x}_i - \mathbf{x}_j|)$, where i and j are input points, and let v_{*j} denote $v(|\mathbf{x}_* - \mathbf{x}_j|)$, $\mathbf{x}_*$ being a point at which we want an interpolated value $y(\mathbf{x}_*)$. Now define two vectors of length $N+1$,

$$\begin{aligned} \mathbf{Y} &= (y_0, y_1, \ldots, y_{N-1}, 0) \\ \mathbf{V}_* &= (v_{*1}, v_{*2}, \ldots, v_{*,N-1}, 1) \end{aligned} \tag{3.7.12}$$

and an $(N+1) \times (N+1)$ symmetric matrix,

$$\mathbf{V} = \begin{pmatrix} v_{00} & v_{01} & \ldots & v_{0,N-1} & 1 \\ v_{10} & v_{11} & \ldots & v_{1,N-1} & 1 \\ & & \ldots & & \ldots \\ v_{N-1,0} & v_{N-1,1} & \ldots & v_{N-1,N-1} & 1 \\ 1 & 1 & \ldots & 1 & 0 \end{pmatrix} \tag{3.7.13}$$

Then the kriging interpolation estimate $\hat{y}_* \approx y(\mathbf{x}_*)$ is given by

$$\hat{y}_* = \mathbf{V}_* \cdot \mathbf{V}^{-1} \cdot \mathbf{Y} \tag{3.7.14}$$

and its variance is given by

$$\mathrm{Var}(\hat{y}_*) = \mathbf{V}_* \cdot \mathbf{V}^{-1} \cdot \mathbf{V}_* \tag{3.7.15}$$

Notice that if we compute, once, the LU decomposition of $\mathbf{V}$, and then backsubstitute, once, to get the vector $\mathbf{V}^{-1} \cdot \mathbf{Y}$, then the individual interpolations cost only $O(N)$ to compute the vector $\mathbf{V}_*$ and take a vector dot product. On the other hand, every computation of a variance, equation (3.7.15), requires an $O(N^2)$ backsubstitution.

As an aside (if you have looked ahead to §13.6) the purpose of the extra row and column in $\mathbf{V}$, and extra last components in $\mathbf{V}_*$ and $\mathbf{Y}$, is to automatically calculate, and correct for, an appropriately weighted average of the data, and thus to make equation (3.7.14) an unbiased estimator.

Here is an implementation of equations (3.7.12) – (3.7.15). The constructor does the one-time work, while the two overloaded `interp` methods calculate either an interpolated value or else a value and a standard deviation (square root of the variance). You should leave the optional argument `err` set to the default value of `NULL` until you read §15.9.

krig.h

```
template<class T>
struct Krig {
```
Object for interpolation by kriging, using npt points in ndim dimensions. Call constructor once, then interp as many times as desired.
```
    const MatDoub &x;
    const T &vgram;
    Int ndim, npt;
    Doub lastval, lasterr;
    VecDoub y,dstar,vstar,yvi;
    MatDoub v;
    LUdcmp *vi;
```
Most recently computed value and (if computed) error.
```
    Krig(MatDoub_I &xx, VecDoub_I &yy, T &vargram, const Doub *err=NULL)
    : x(xx),vgram(vargram),npt(xx.nrows()),ndim(xx.ncols()),dstar(npt+1),
    vstar(npt+1),v(npt+1,npt+1),y(npt+1),yvi(npt+1) {
```
Constructor. The npt × ndim matrix xx inputs the data points, the vector yy the function values. vargram is the variogram function or functor. The argument err is not used for interpolation; see §15.9.
```
        Int i,j;
        for (i=0;i<npt;i++) {
            y[i] = yy[i];
            for (j=i;j<npt;j++) {
                v[i][j] = v[j][i] = vgram(rdist(&x[i][0],&x[j][0]));
            }
            v[i][npt] = v[npt][i] = 1.;
        }
        v[npt][npt] = y[npt] = 0.;
        if (err) for (i=0;i<npt;i++) v[i][i] -= SQR(err[i]);
        vi = new LUdcmp(v);
        vi->solve(y,yvi);
    }
    ~Krig() { delete vi; }
```
Fill $\mathbf{Y}$ and $\mathbf{V}$.

§15.9.
```
    Doub interp(VecDoub_I &xstar) {
```
Return an interpolated value at the point xstar.
```
        Int i;
        for (i=0;i<npt;i++) vstar[i] = vgram(rdist(&xstar[0],&x[i][0]));
        vstar[npt] = 1.;
        lastval = 0.;
        for (i=0;i<=npt;i++) lastval += yvi[i]*vstar[i];
        return lastval;
    }

    Doub interp(VecDoub_I &xstar, Doub &esterr) {
```
Return an interpolated value at the point xstar, and return its estimated error as esterr.
```
        lastval = interp(xstar);
        vi->solve(vstar,dstar);
        lasterr = 0;
        for (Int i=0;i<=npt;i++) lasterr += dstar[i]*vstar[i];
        esterr = lasterr = sqrt(MAX(0.,lasterr));
        return lastval;
    }

    Doub rdist(const Doub *x1, const Doub *x2) {
```
Utility used internally. Cartesian distance between two points.
```
        Doub d=0.;
        for (Int i=0;i<ndim;i++) d += SQR(x1[i]-x2[i]);
        return sqrt(d);
    }
};
```

The constructor argument vgram, the variogram function, can be either a func-

tion or functor (§1.3.3). For interpolation, you can use a `Powvargram` object that fits a simple model

$$v(r) = \alpha r^{\beta} \tag{3.7.16}$$

where β is considered fixed and α is fitted by unweighted least squares over all pairs of data points i and j. We'll get more sophisticated about variograms in §15.9; but for interpolation, excellent results can be obtained with this simple choice. The value of β should be in the range $1 \le \beta < 2$. A good general choice is 1.5, but for functions with a strong linear trend, you may want to experiment with values as large as 1.99. (The value 2 gives a degenerate matrix and meaningless results.) The optional argument nug will be explained in §15.9.

krig.h

```
struct Powvargram {
Functor for variogram v(r) = αr^β, where β is specified, α is fitted from the data.
    Doub alph, bet, nugsq;

    Powvargram(MatDoub_I &x, VecDoub_I &y, const Doub beta=1.5, const Doub nug=0.)
    : bet(beta), nugsq(nug*nug) {
    Constructor. The npt × ndim matrix x inputs the data points, the vector y the function
    values, beta the value of β. For interpolation, the default value of beta is usually adequate.
    For the (rare) use of nug see §15.9.
        Int i,j,k,npt=x.nrows(),ndim=x.ncols();
        Doub rb,num=0.,denom=0.;
        for (i=0;i<npt;i++) for (j=i+1;j<npt;j++) {
            rb = 0.;
            for (k=0;k<ndim;k++) rb += SQR(x[i][k]-x[j][k]);
            rb = pow(rb,0.5*beta);
            num += rb*(0.5*SQR(y[i]-y[j]) - nugsq);
            denom += SQR(rb);
        }
        alph = num/denom;
    }

    Doub operator() (const Doub r) const {return nugsq+alph*pow(r,bet);}
};
```

Sample code for interpolating on a set of data points is

```
MatDoub x(npts,ndim);
VecDoub y(npts), xstar(ndim);
...
Powvargram vgram(x,y);
Krig<Powvargram> krig(x,y,vgram);
```

followed by any number of interpolations of the form

```
ystar = krig.interp(xstar);
```

Be aware that while the interpolated values are quite insensitive to the variogram model, the estimated errors are rather sensitive to it. You should thus consider the error estimates as being order of magnitude only. Since they are also relatively expensive to compute, their value in this application is not great. They will be much more useful in §15.9, when our model includes measurement errors.

3.7.5 Curve Interpolation in Multidimensions

A different kind of interpolation, worth a brief mention here, is when you have an ordered set of N tabulated points in n dimensions that lie on a one-dimensional curve, $\mathbf{x}_0, \ldots \mathbf{x}_{N-1}$, and you want to interpolate other values along the curve. Two

cases worth distinguishing are: (i) The curve is an open curve, so that $\mathbf{x}_0$ and $\mathbf{x}_{N-1}$ represent endpoints. (ii) The curve is a closed curve, so that there is an implied curve segment connecting $\mathbf{x}_{N-1}$ back to $\mathbf{x}_0$.

A straightforward solution, using methods already at hand, is first to approximate distance along the curve by the sum of chord lengths between the tabulated points, and then to construct spline interpolations for each of the coordinates, $0, \ldots, n-1$, as a function of that parameter. Since the derivative of any single coordinate with respect to arc length can be no greater than 1, it is guaranteed that the spline interpolations will be well-behaved.

Probably 90% of applications require nothing more complicated than the above. If you are in the unhappy 10%, then you will need to learn about *Bézier curves*, *B-splines*, and *interpolating splines* more generally [12,13,14]. For the happy majority, an implementation is

interp_curve.h

```
struct Curve_interp {
```
Object for interpolating a curve specified by n points in dim dimensions.
```
    Int dim, n, in;
    Bool cls;                                   Set if a closed curve.
    MatDoub pts;
    VecDoub s;
    VecDoub ans;
    NRvector<Spline_interp*> srp;

    Curve_interp(MatDoub &ptsin, Bool close=0)
    : n(ptsin.nrows()), dim(ptsin.ncols()), in(close ? 2*n : n),
    cls(close), pts(dim,in), s(in), ans(dim), srp(dim) {
```
    Constructor. The n × dim matrix ptsin inputs the data points. Input close as 0 for an open curve, 1 for a closed curve. (For a closed curve, the last data point should *not* duplicate the first — the algorithm will connect them.)
```
        Int i,ii,im,j,ofs;
        Doub ss,soff,db,de;
        ofs = close ? n/2 : 0;                  The trick for closed curves is to duplicate half a
        s[0] = 0.;                                  period at the beginning and end, and then
        for (i=0;i<in;i++) {                        use the middle half of the resulting spline.
            ii = (i-ofs+n) % n;
            im = (ii-1+n) % n;
            for (j=0;j<dim;j++) pts[j][i] = ptsin[ii][j];       Store transpose.
            if (i>0) {                          Accumulate arc length.
                s[i] = s[i-1] + rad(&ptsin[ii][0],&ptsin[im][0]);
                if (s[i] == s[i-1]) throw("error in Curve_interp");
```
                Consecutive points may not be identical. For a closed curve, the last data point should *not* duplicate the first.
```
            }
        }
        ss = close ? s[ofs+n]-s[ofs] : s[n-1]-s[0];  Rescale parameter so that the
        soff = s[ofs];                              interval [0,1] is the whole curve (or one period).
        for (i=0;i<in;i++) s[i] = (s[i]-soff)/ss;
        for (j=0;j<dim;j++) {                   Construct the splines using endpoint derivatives.
            db = in < 4 ? 1.e99 : fprime(&s[0],&pts[j][0],1);
            de = in < 4 ? 1.e99 : fprime(&s[in-1],&pts[j][in-1],-1);
            srp[j] = new Spline_interp(s,&pts[j][0],db,de);
        }
    }
    ~Curve_interp() {for (Int j=0;j<dim;j++) delete srp[j];}

    VecDoub &interp(Doub t) {
```
    Interpolate a point on the stored curve. The point is parameterized by t, in the range [0,1]. For open curves, values of t outside this range will return extrapolations (dangerous!). For closed curves, t is periodic with period 1.
```
```

```
        if (cls) t = t - floor(t);
        for (Int j=0;j<dim;j++) ans[j] = (*srp[j]).interp(t);
        return ans;
    }

    Doub fprime(Doub *x, Doub *y, Int pm) {
    Utility for estimating the derivatives at the endpoints. x and y point to the abscissa and
    ordinate of the endpoint. If pm is +1, points to the right will be used (left endpoint); if it
    is −1, points to the left will be used (right endpoint). See text, below.
        Doub s1 = x[0]-x[pm*1], s2 = x[0]-x[pm*2], s3 = x[0]-x[pm*3],
            s12 = s1-s2, s13 = s1-s3, s23 = s2-s3;
        return -(s1*s2/(s13*s23*s3))*y[pm*3]+(s1*s3/(s12*s2*s23))*y[pm*2]
            -(s2*s3/(s1*s12*s13))*y[pm*1]+(1./s1+1./s2+1./s3)*y[0];
    }

    Doub rad(const Doub *p1, const Doub *p2) {
    Euclidean distance.
        Doub sum = 0.;
        for (Int i=0;i<dim;i++) sum += SQR(p1[i]-p2[i]);
        return sqrt(sum);
    }

};
```

The utility routine `fprime` estimates the derivative of a function at a tabulated abscissa x_0, using four consecutive tabulated abscissa-ordinate pairs, $(x_0, y_0), \ldots, (x_3, y_3)$. The formula for this, readily derived by power-series expansion, is

$$y_0' = -C_0 y_0 + C_1 y_1 - C_2 y_2 + C_3 y_3 \tag{3.7.17}$$

where

$$\begin{aligned} C_0 &= \frac{1}{s_1} + \frac{1}{s_2} + \frac{1}{s_3} \\ C_1 &= \frac{s_2 s_3}{s_1(s_2 - s_1)(s_3 - s_1)} \\ C_2 &= \frac{s_1 s_3}{(s_2 - s_1)s_2(s_3 - s_2)} \\ C_3 &= \frac{s_1 s_2}{(s_3 - s_1)(s_3 - s_2)s_3} \end{aligned} \tag{3.7.18}$$

with

$$\begin{aligned} s_1 &\equiv x_1 - x_0 \\ s_2 &\equiv x_2 - x_0 \\ s_3 &\equiv x_3 - x_0 \end{aligned} \tag{3.7.19}$$

CITED REFERENCES AND FURTHER READING:

Buhmann, M.D. 2003, *Radial Basis Functions: Theory and Implementations* (Cambridge, UK: Cambridge University Press).[1]

Powell, M.J.D. 1992, "The Theory of Radial Basis Function Approximation" in *Advances in Numerical Analysis II: Wavelets, Subdivision Algorithms and Radial Functions*, ed. W. A. Light (Oxford: Oxford University Press), pp. 105–210.[2]

Wikipedia. 2007+, "Radial Basis Functions," at `http://en.wikipedia.org/`.[3]

Beatson, R.K. and Greengard, L. 1997, "A Short Course on Fast Multipole Methods", in *Wavelets, Multilevel Methods and Elliptic PDEs*, eds. M. Ainsworth, J. Levesley, W. Light, and M. Marletta (Oxford: Oxford University Press), pp. 1–37.[4]

Beatson, R.K. and Newsam, G.N. 1998, "Fast Evaluation of Radial Basis Functions: Moment-Based Methods" in *SIAM Journal on Scientific Computing*, vol. 19, pp. 1428-1449.[5]

Wendland, H. 2005, *Scattered Data Approximation* (Cambridge, UK: Cambridge University Press).[6]

Franke, R. 1982, "Scattered Data Interpolation: Tests of Some Methods," *Mathematics of Computation*, vol. 38, pp. 181–200.[7]

Shepard, D. 1968, "A Two-dimensional Interpolation Function for Irregularly-spaced Data," in *Proceedings of the 1968 23rd ACM National Conference* (New York: ACM Press), pp. 517–524.[8]

Cressie, N. 1991, *Statistics for Spatial Data* (New York: Wiley).[9]

Wackernagel, H. 1998, *Multivariate Geostatistics*, 2nd ed. (Berlin: Springer).[10]

Rybicki, G.B., and Press, W.H. 1992, "Interpolation, Realization, and Reconstruction of Noisy, Irregularly Sampled Data," *Astrophysical Journal*, vol. 398, pp. 169–176.[11]

Isaaks, E.H., and Srivastava, R.M. 1989, *Applied Geostatistics* (New York: Oxford University Press).

Deutsch, C.V., and Journel, A.G. 1992, *GSLIB: Geostatistical Software Library and User's Guide* (New York: Oxford University Press).

Knott, G.D. 1999, *Interpolating Cubic Splines* (Boston: Birkhäuser).[12]

De Boor, C. 2001, *A Practical Guide to Splines* (Berlin: Springer).[13]

Prautzsch, H., Boehm, W., and Paluszny, M. 2002, *Bézier and B-Spline Techniques* (Berlin: Springer).[14]

3.8 Laplace Interpolation 1

In this section we look at a *missing data* or *gridding* problem, namely, how to restore missing or unmeasured values on a regular grid. Evidently some kind of interpolation from the not-missing values is required, but how shall we do this in a principled way?

One good method, already in use at the dawn of the computer age [1,2], is *Laplace interpolation*, sometimes called *Laplace/Poisson interpolation*. The general idea is to find an interpolating function y that satisfies Laplace's equation in n dimensions,

$$\nabla^2 y = 0 \tag{3.8.1}$$

wherever there is no data, and which satisfies

$$y(\mathbf{x}_i) = y_i \tag{3.8.2}$$

at all measured data points. Generically, such a function does exist. The reason for choosing Laplace's equation (among all possible partial differential equations, say) is that the solution to Laplace's equation selects, in some sense, the smoothest possible interpolant. In particular, its solution minimizes the integrated square of the gradient,

$$\int_\Omega |\nabla y|^2 \, d\Omega \tag{3.8.3}$$

where Ω denotes the n-dimensional domain of interest. This is a very general idea, and it can be applied to irregular meshes as well as to regular grids. Here, however, we consider only the latter.

For purposes of illustration (and because it is the most useful example) we further specialize to the case of two dimensions, and to the case of a Cartesian grid whose x_1 and x_2 values are evenly spaced — like a checkerboard.

In this geometry, the finite difference approximation to Laplace's equation has a particularly simple form, one that echos the *mean value theorem* for continuous solutions of the Laplace equation: The value of the solution at any free gridpoint (i.e., not a point with a measured value) equals the average of its four Cartesian neighbors. (See §20.0.) Indeed, this already sounds a lot like interpolation.

If y_0 denotes the value at a free point, while y_u, y_d, y_l, and y_r denote the values at its up, down, left, and right neighbors, respectively, then the equation satisfied is

$$y_0 - \tfrac{1}{4}y_u - \tfrac{1}{4}y_d - \tfrac{1}{4}y_l - \tfrac{1}{4}y_r = 0 \tag{3.8.4}$$

For gridpoints with measured values, on the other hand, a different (simple) equation is satisfied,

$$y_0 = y_{0(\text{measured})} \tag{3.8.5}$$

Note that these nonzero right-hand sides are what make an inhomogeneous, and therefore generally solvable, set of linear equations.

We are not quite done, since we must provide special forms for the top, bottom, left, and right boundaries, and for the four corners. Homogeneous choices that embody "natural" boundary conditions (with no preferred function values) are

$$\begin{aligned}
y_0 - \tfrac{1}{2}y_u - \tfrac{1}{2}y_d &= 0 \qquad \text{(left and right boundaries)} \\
y_0 - \tfrac{1}{2}y_l - \tfrac{1}{2}y_r &= 0 \qquad \text{(top and bottom boundaries)} \\
y_0 - \tfrac{1}{2}y_r - \tfrac{1}{2}y_d &= 0 \qquad \text{(top-left corner)} \\
y_0 - \tfrac{1}{2}y_l - \tfrac{1}{2}y_d &= 0 \qquad \text{(top-right corner)} \\
y_0 - \tfrac{1}{2}y_r - \tfrac{1}{2}y_u &= 0 \qquad \text{(bottom-left corner)} \\
y_0 - \tfrac{1}{2}y_l - \tfrac{1}{2}y_u &= 0 \qquad \text{(bottom-right corner)}
\end{aligned} \tag{3.8.6}$$

Since every gridpoint corresponds to exactly one of the equations in (3.8.4), (3.8.5), or (3.8.4), we have exactly as many equations as there are unknowns. If the grid is M by N, then there are MN of each. This can be quite a large number; but the equations are evidently very sparse. We solve them by defining a derived class from §2.7's `Linbcg` base class. You can readily identify all the cases of equations (3.8.4) – (3.8.6) in the code for `atimes`, below.

interp_laplace.h

```
struct Laplace_interp : Linbcg {
Object for interpolating missing data in a matrix by solving Laplace's equation. Call constructor
once, then solve one or more times (see text).
    MatDoub &mat;
    Int ii,jj;
    Int nn,iter;
    VecDoub b,y,mask;

    Laplace_interp(MatDoub_IO &matrix) : mat(matrix), ii(mat.nrows()),
    jj(mat.ncols()), nn(ii*jj), iter(0), b(nn), y(nn), mask(nn) {
    Constructor. Values greater than 1.e99 in the input matrix mat are deemed to be missing
    data. The matrix is not altered until solve is called.
        Int i,j,k;
        Doub vl = 0.;
```

```
        for (k=0;k<nn;k++) {
            i = k/jj;
            j = k - i*jj;
            if (mat[i][j] < 1.e99) {
                b[k] =  y[k] = vl = mat[i][j];
                mask[k] = 1;
            } else {
                b[k] = 0.;
                y[k] = vl;
                mask[k] = 0;
            }
        }
    }
```

Fill the r.h.s. vector, the initial guess, and a mask of the missing data.

```
    void asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp);
    void atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp);
```

See definitions below. These are the real algorithmic content.

```
    Doub solve(Doub tol=1.e-6, Int itmax=-1) {
```

Invoke `Linbcg::solve` with appropriate arguments. The default argument values will usually work, in which case this routine need be called only once. The original matrix `mat` is refilled with the interpolated solution. 1

```
        Int i,j,k;
        Doub err;
        if (itmax <= 0) itmax = 2*MAX(ii,jj);
        Linbcg::solve(b,y,1,tol,itmax,iter,err);
        for (k=0,i=0;i<ii;i++) for (j=0;j<jj;j++) mat[i][j] = y[k++];
        return err;
    }
};
```

4

```
void Laplace_interp::asolve(VecDoub_I &b, VecDoub_O &x, const Int itrnsp) {
```

Diagonal preconditioner. (Diagonal elements all unity.)

```
    Int i,n=b.size();
    for (i=0;i<n;i++) x[i] = b[i];
}
```

```
void Laplace_interp::atimes(VecDoub_I &x, VecDoub_O &r, const Int itrnsp) {
```

2

Sparse matrix, and matrix transpose, multiply. This routine embodies eqs. (3.8.4), (3.8.5), and (3.8.6).

```
    Int i,j,k,n=r.size(),jjt,it;
    Doub del;
    for (k=0;k<n;k++) r[k] = 0.;
    for (k=0;k<n;k++) {
        i = k/jj;
        j = k - i*jj;
        if (mask[k]) {                                  Measured point, eq. (3.8.5).
            r[k] += x[k];
        } else if (i>0 && i<ii-1 && j>0 && j<jj-1) {    Interior point, eq. (3.8.4).
            if (itrnsp) {
                r[k] += x[k];
                del = -0.25*x[k];
                r[k-1] += del;
                r[k+1] += del;
                r[k-jj] += del;
                r[k+jj] += del;
            } else {
                r[k] = x[k] - 0.25*(x[k-1]+x[k+1]+x[k+jj]+x[k-jj]);
            }
        } else if (i>0 && i<ii-1) {                     Left or right edge, eq. (3.8.6).
            if (itrnsp) {
                r[k] += x[k];
                del = -0.5*x[k];
                r[k-jj] += del;
```

3

```
                r[k+jj] += del;
            } else {
                r[k] = x[k] - 0.5*(x[k+jj]+x[k-jj]);
            }
        } else if (j>0 && j<jj-1) {                    Top or bottom edge, eq. (3.8.6).
            if (itrnsp) {
                r[k] += x[k];
                del = -0.5*x[k];
                r[k-1] += del;
                r[k+1] += del;
            } else {
                r[k] = x[k] - 0.5*(x[k+1]+x[k-1]);
            }
        } else {                                       Corners, eq. (3.8.6).
            jjt = i==0 ? jj : -jj;
            it = j==0 ? 1 : -1;
            if (itrnsp) {
                r[k] += x[k];
                del = -0.5*x[k];
                r[k+jjt] += del;
                r[k+it] += del;
            } else {
                r[k] = x[k] - 0.5*(x[k+jjt]+x[k+it]);
            }
        }
    }
}
```

Usage is quite simple. Just fill a matrix with function values where they are known, and with `1.e99` where they are not; send the matrix to the constructor; and call the `solve` routine. The missing values will be interpolated. The default arguments should serve for most cases.

```
Int m=...,n=...;
MatDoub mat(m,n);
...
Laplace_interp mylaplace(mat);
mylaplace.solve();
```

Quite decent results are obtained for smooth functions on 300×300 matrices in which a random 10% of gridpoints have known function values, with 90% interpolated. However, since compute time scales as $MN \max(M, N)$ (that is, as the cube), this is not a method to use for much larger matrices, unless you break them up into overlapping tiles. If you experience convergence difficulties, then you should call `solve`, with appropriate nondefault arguments, several times in succession, and look at the returned error estimate after each call returns.

3.8.1 Minimum Curvature Methods

Laplace interpolation has a tendency to yield cone-like cusps around any small islands of known data points that are surrounded by a sea of unknowns. The reason is that, in two dimensions, the solution of Laplace's equation near a point source is logarithmically singular. When the known data is spread fairly evenly (if randomly) across the grid, this is not generally a problem. *Minimum curvature methods* deal with the problem at a more fundamental level by being based on the biharmonic equation

$$\nabla(\nabla y) = 0 \tag{3.8.7}$$

instead of Laplace's equation. Solutions of the biharmonic equation minimize the integrated square of the curvature,

$$\int_\Omega |\nabla^2 y|^2 \, d\Omega \tag{3.8.8}$$

Minimum curvature methods are widely used in the earth-science community [3,4].

The references give a variety of other methods that can be used for missing data interpolation and gridding.

CITED REFERENCES AND FURTHER READING:

Noma, A.A. and Misulia, M.G. 1959, "Programming Topographic Maps for Automatic Terrain Model Construction," *Surveying and Mapping*, vol. 19, pp. 355–366.[1]

Crain, I.K. 1970, "Computer Interpolation and Contouring of Two-dimensional Data: a Review," *Geoexploration*, vol. 8, pp. 71–86.[2]

Burrough, P.A. 1998, *Principles of Geographical Information Systems*, 2nd ed. (Oxford, UK: Clarendon Press)

Watson, D.F. 1982, *Contouring: A Guide to the Analysis and Display of Spatial Data* (Oxford, UK: Pergamon).

Briggs, I.C. 1974, "Machine Contouring Using Minimum Curvature," *Geophysics*, vol. 39, pp. 39–48.[3]

Smith, W.H.F. and Wessel, P. 1990, "Gridding With Continuous Curvature Splines in Tension," *Geophysics*, vol. 55, pp. 293–305.[4]