3.4 How to Search an Ordered Table

Suppose that you have decided to use some particular interpolation scheme, such as fourth-order polynomial interpolation, to compute a function $f(x)$ from a set of tabulated x_i's and f_i's. Then you will need a fast way of finding your place in the table of x_i's, given some particular value x at which the function evaluation is desired. This problem is not properly one of numerical analysis, but it occurs so often in practice that it would be negligent of us to ignore it.

Formally, the problem is this: Given an array of abscissas `xx[j]`, `j`$=1, 2, \dots,$`n`, with the elements either monotonically increasing or monotonically decreasing, and given a number `x`, find an integer `j` such that `x` lies between `xx[j]` and `xx[j+1]`. For this task, let us define fictitious array elements `xx[0]` and `xx[n+1]` equal to plus or minus infinity (in whichever order is consistent with the monotonicity of the table). Then `j` will always be between 0 and `n`, inclusive; a value of 0 indicates "off-scale" at one end of the table, `n` indicates off-scale at the other end.

In most cases, when all is said and done, it is hard to do better than *bisection*, which will find the right place in the table in about $\log_2$`n` tries. We already did use bisection in the spline evaluation routine `splint` of the preceding section, so you might glance back at that. Standing by itself, a bisection routine looks like this:

```
void locate(float xx[], unsigned long n, float x, unsigned long *j)
Given an array xx[1..n], and given a value x, returns a value j such that x is between xx[j]
and xx[j+1]. xx must be monotonic, either increasing or decreasing. j=0 or j=n is returned
to indicate that x is out of range.
{
    unsigned long ju,jm,jl;
    int ascnd;

    jl=0;                           Initialize lower
    ju=n+1;                         and upper limits.
    ascnd=(xx[n] >= xx[1]);
    while (ju-jl > 1) {             If we are not yet done,
        jm=(ju+jl) >> 1;            compute a midpoint,
        if (x >= xx[jm] == ascnd)
            jl=jm;                  and replace either the lower limit
        else
            ju=jm;                  or the upper limit, as appropriate.
    }                               Repeat until the test condition is satisfied.
    if (x == xx[1]) *j=1;           Then set the output
    else if(x == xx[n]) *j=n-1;
    else *j=jl;
}                                   and return.
```

A unit-offset array `xx` is assumed. To use `locate` with a zero-offset array, remember to subtract 1 from the address of `xx`, and also from the returned value `j`.

Search with Correlated Values

Sometimes you will be in the situation of searching a large table many times, and with nearly identical abscissas on consecutive searches. For example, you may be generating a function that is used on the right-hand side of a differential equation: Most differential-equation integrators, as we shall see in Chapter 16, call

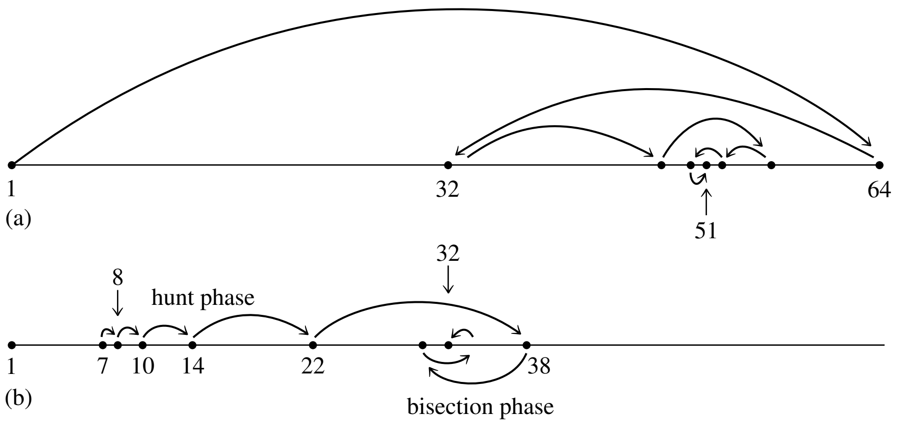

Figure 3.4.1. (a) The routine `locate` finds a table entry by bisection. Shown here is the sequence of steps that converge to element 51 in a table of length 64. (b) The routine `hunt` searches from a previous known position in the table by increasing steps, then converges by bisection. Shown here is a particularly unfavorable example, converging to element 32 from element 7. A favorable example would be convergence to an element near 7, such as 9, which would require just three "hops."

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. The following routine instead 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. Second, it then bisects in the bracketed interval. At worst, this routine is about a 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 3.4.1 compares the two routines.

```
void hunt(float xx[], unsigned long n, float x, unsigned long *jlo)
```
Given an array `xx[1..n]`, and given a value `x`, returns a value `jlo` such that `x` is between `xx[jlo]` and `xx[jlo+1]`. `xx[1..n]` must be monotonic, either increasing or decreasing. `jlo=0` or `jlo=n` is returned to indicate that `x` is out of range. `jlo` on input is taken as the initial guess for `jlo` on output.
```
{
    unsigned long jm,jhi,inc;
    int ascnd;

    ascnd=(xx[n] >= xx[1]);              True if ascending order of table, false otherwise.
    if (*jlo <= 0 || *jlo > n) {         Input guess not useful. Go immediately to bisec-
        *jlo=0;                              tion.
        jhi=n+1;
    } else {
        inc=1;                           Set the hunting increment.
        if (x >= xx[*jlo] == ascnd) {    Hunt up:
            if (*jlo == n) return;
            jhi=(*jlo)+1;
            while (x >= xx[jhi] == ascnd) {      Not done hunting,
                *jlo=jhi;
                inc += inc;              so double the increment
                jhi=(*jlo)+inc;
                if (jhi > n) {           Done hunting, since off end of table.
                    jhi=n+1;
                    break;
                }                        Try again.
```

```
            }                               Done hunting, value bracketed.
        } else {                            Hunt down:
            if (*jlo == 1) {
                *jlo=0;
                return;
            }
            jhi=(*jlo)--;
            while (x < xx[*jlo] == ascnd) {         Not done hunting,
                jhi=(*jlo);
                inc <<= 1;                  so double the increment
                if (inc >= jhi) {           Done hunting, since off end of table.
                    *jlo=0;
                    break;
                }
                else *jlo=jhi-inc;
            }                               and try again.
        }                                   Done hunting, value bracketed.
    }                                       Hunt is done, so begin the final bisection phase:
    while (jhi-(*jlo) != 1) {
        jm=(jhi+(*jlo)) >> 1;
        if (x >= xx[jm] == ascnd)
            *jlo=jm;
        else
            jhi=jm;
    }
    if (x == xx[n]) *jlo=n-1;
    if (x == xx[1]) *jlo=1;
}
```

If your array `xx` is zero-offset, read the comment following `locate`, above.

After the Hunt

The problem: Routines `locate` and `hunt` return an index `j` such that your desired value lies between table entries `xx[j]` and `xx[j+1]`, where `xx[1..n]` is the full length of the table. But, to obtain an `m`-point interpolated value using a routine like `polint` (§3.1) or `ratint` (§3.2), you need to supply much shorter `xx` and `yy` arrays, of length `m`. How do you make the connection?

The solution: Calculate

$$k = \texttt{IMIN(IMAX(j-(m-1)/2,1),n+1-m)}$$

(The macros `IMIN` and `IMAX` give the minimum and maximum of two integer arguments; see §1.2 and Appendix B.) This expression produces the index of the leftmost member of an `m`-point set of points centered (insofar as possible) between `j` and `j+1`, but bounded by 1 at the left and `n` at the right. C then lets you call the interpolation routine with array addresses offset by `k`, e.g.,

```
polint(&xx[k-1],&yy[k-1],m,...)
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.5 Coefficients of the Interpolating Polynomial

Occasionally you may wish to know not the value of the interpolating polynomial that passes through a (small!) number of points, but the coefficients of that polynomial. A valid use of the coefficients might be, for example, to compute simultaneous interpolated values of the function and of several of its derivatives (see §5.3), or to convolve a segment of the tabulated function with some other function, where the moments of that other function (i.e., its convolution with powers of x) are known analytically.

However, please be certain that the coefficients are what you need. Generally the coefficients of the interpolating polynomial can be determined much less accurately than its value at a desired abscissa. Therefore it is not a good idea to determine the coefficients only for use in calculating interpolating values. Values thus calculated will not pass exactly through the tabulated points, for example, while values computed by the routines in §3.1–§3.3 will pass exactly through such points.

Also, you should not mistake the interpolating polynomial (and its coefficients) for its cousin, the *best fit* polynomial through a data set. Fitting is a *smoothing* process, since the number of fitted coefficients is typically much less than the number of data points. Therefore, fitted coefficients can be accurately and stably determined even in the presence of statistical errors in the tabulated values. (See §14.8.) Interpolation, where the number of coefficients and number of tabulated points are equal, takes the tabulated values as perfect. If they in fact contain statistical errors, these can be magnified into oscillations of the interpolating polynomial in between the tabulated points.

As before, we take the tabulated points to be $y_i \equiv y(x_i)$. If the interpolating polynomial is written as

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

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

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} \tag{3.5.2}$$

This is a *Vandermonde matrix*, as described in §2.8. One could in principle solve equation (3.5.2) by standard techniques for linear equations generally (§2.3); however the special method that was derived in §2.8 is more efficient by a large factor, of order N, so it is much better.

Remember that Vandermonde systems can be quite ill-conditioned. In such a case, *no* numerical method is going to give a very accurate answer. Such cases do not, please note, imply any difficulty in finding interpolated *values* by the methods of §3.1, but only difficulty in finding *coefficients*.

Like the routine in §2.8, the following is due to G.B. Rybicki. Note that the arrays are all assumed to be zero-offset.