values of the interpolated function for any value of $x$ are obtained by calls (as many as desired) to a separate routine \texttt{splint} (for “spline interpolation”):

\begin{verbatim}
SUBROUTINE splint(xa,ya,y2a,n,x,y)
INTEGER n
REAL x,y,xa(n),ya(n),y2a(n)
Given the arrays xa(1:n) and ya(1:n) of length n, which tabulate a function (with the xa,’s in order), and given the array y2a(1:n), which is the output from \texttt{spline} above, and given a value of x, this routine returns a cubic-spline interpolated value y.
INTEGER k,khi,klo
REAL a,b,h
klo=1
khi=n
1 if (khi-klo.gt.1) then
  k=(khi+klo)/2
  if (xa(k).gt.x) then 
    khi=k
  else
    klo=k
  endif
  goto 1
endif
klo and khi now bracket the input value of x.
h=xa(khi)-xa(klo)
if (h.eq.0.) pause ‘bad xa input in splint’ The xa’s must be distinct.
a=(xa(khi)-x)/h
b=(x-xa(klo))/h
y=a*ya(klo)+b*ya(khi)
*+((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.
return
END
\end{verbatim}

CITED REFERENCES AND FURTHER READING:

### 3.4 How to Search an Ordered Table

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

Formally, the problem is this: Given an array of abscissas $xx(j)$, $j=1,2,...,n$, with the elements either monotonically increasing or monotonically decreasing, and given a number $x$, find an integer $j$ such that $x$ lies between $xx(j)$ and $xx(j+1)$. 

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press.
Programs Copyright (C) 1986-1992 by Numerical Recipes Software. Reproduction without written permission is strictly prohibited.
Send queries to the webmaster, Cambridge University Press, via http://www.u16.umd.edu/press/
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 returned 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:

```fortran
SUBROUTINE locate(xx,n,x,j)
    INTEGER j,n
    REAL x,xx(n)
    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(1:n) must be monotonic, either increasing or decreasing. j=0
    or j=n is returned to indicate that x is out of range.
    INTEGER jl,jm,ju
    jl=0
    Initialize lower
    ju=n+1
    and upper limits.
    10 if(ju-jl.gt.1)then
    If we are not yet done,
    jm=(ju+jl)/2
    compute a midpoint,
    if((xx(n).ge.xx(1)).eqv.(x.ge.xx(jm)))then
    jl=jm
    and replace either the lower limit
    else
    ju=jm
    or the upper limit, as appropriate.
    endif
    goto 10
    Repeat until
    endif
    if(x.eq.xx(1))then
    Then set the output
    j=1
    else if(x.eq.xx(n))then
    j=n-1
    else
    endif
    return
    and return.
END
```

Note the use of the logical equality relation .eqv., which is true when its two logical operands are either both true or both false. This relation allows the routine to work for both monotonically increasing and monotonically decreasing orders of $xx(1:n)$.

**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 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.
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.”

SUBROUTINE hunt(xx,n,x,jlo)
INTEGER jlo,n
REAL x,xx(n)
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.
INTEGER inc,jhi,jm
LOGICAL ascnd
ascnd=xx(n).ge.xx(1)
True if ascending order of table, false otherwise.
if(jlo.le.0.or.jlo.gt.n)then
Input guess not useful. Go immediately to bisection.
endif
inc=1
Set the hunting increment.
if(x.ge.xx(jlo).eqv.ascnd)then
Hunt up:
1 jhi=jlo+inc
if(jhi.gt.n)then
Done hunting, since off end of table.
jhi=n+1
else if(x.ge.xx(jhi).eqv.ascnd)then
Not done hunting,
jlo=jhi
inc=inc+inc
goto 1
endif
endif
else
Hunt down:
2 jlo=jlo-inc
if(jlo.lt.1)then
Done hunting, since off end of table.
jlo=0
else if(x.ge.xx(jlo).eqv.ascnd)then
Not done hunting,
jhi=jlo
inc=inc+inc
goto 2
endif
endif
3 if(jhi-jlo.eq.1)then
Hunt is done, so begin the final bisection phase:
if(x.eq.xx(n))jlo=n-1
if(x.eq.xx(1))jlo=1

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 \( \text{polint} \) (§3.1) or \( \text{ratint} \) (§3.2), you need to supply much shorter \( xx \) and \( yy \) arrays, of length \( m \). How do you make the connection?

The solution: Calculate

\[
k = \min(\max(j-(m-1)/2,1),n+1-m)
\]

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. FORTRAN then lets you call the interpolation routine with array addresses offset by \( k \), e.g.,

\[
\text{call polint}(xx(k),yy(k),m,...)
\]

CITED REFERENCES AND FURTHER READING:

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.