Chapter B3. Interpolation and Extrapolation

```
SUBROUTINE polint(xa,ya,x,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays xa and ya of length N, and given a value x, this routine returns a value y, and an error estimate dy. If $P(x)$ is the polynomial of degree $N-1$ such that $P(\mathtt{xa}_i) = \mathtt{ya}_i, i = 1, \ldots, N$, then the returned value $\mathtt{y} = P(\mathtt{x})$.

```
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,den,ho
n=assert_eq(size(xa),size(ya),'polint')
c=ya                                       Initialize the tableau of c's and d's.
d=ya
ho=xa-x
ns=iminloc(abs(x-xa))                      Find index ns of closest table entry.
y=ya(ns)                                   This is the initial approximation to y.
ns=ns-1
do m=1,n-1                                 For each column of the tableau,
    den(1:n-m)=ho(1:n-m)-ho(1+m:n)         we loop over the current c's and d's and up-
    if (any(den(1:n-m) == 0.0)) &              date them.
        call nrerror('polint: calculation failure')
          This error can occur only if two input xa's are (to within roundoff) identical.
    den(1:n-m)=(c(2:n-m+1)-d(1:n-m))/den(1:n-m)
    d(1:n-m)=ho(1+m:n)*den(1:n-m)          Here the c's and d's are updated.
    c(1:n-m)=ho(1:n-m)*den(1:n-m)
    if (2*ns < n-m) then     After each column in the tableau is completed, we decide
        dy=c(ns+1)               which correction, c or d, we want to add to our accu-
    else                         mulating value of y, i.e., which path to take through
        dy=d(ns)                 the tableau—forking up or down. We do this in such a
        ns=ns-1                  way as to take the most "straight line" route through the
    end if                       tableau to its apex, updating ns accordingly to keep track
    y=y+dy                       of where we are. This route keeps the partial approxima-
end do                           tions centered (insofar as possible) on the target x. The
END SUBROUTINE polint            last dy added is thus the error indication.
```

```
SUBROUTINE ratint(xa,ya,x,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays xa and ya of length N, and given a value of x, this routine returns a value of y and an accuracy estimate dy. The value returned is that of the diagonal rational function, evaluated at x, that passes through the N points $(\mathtt{xa}_i, \mathtt{ya}_i)$, $i = 1 \ldots N$.

```
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,dd,h,t
```

```
REAL(SP), PARAMETER :: TINY=1.0e-25_sp          A small number.
n=assert_eq(size(xa),size(ya),'ratint')
h=xa-x
ns=iminloc(abs(h))
y=ya(ns)
if (x == xa(ns)) then
    dy=0.0
    RETURN
end if
c=ya
d=ya+TINY                                       The TINY part is needed to prevent
ns=ns-1                                             a rare zero-over-zero condition.
do m=1,n-1
    t(1:n-m)=(xa(1:n-m)-x)*d(1:n-m)/h(1+m:n)    h will never be zero, since this was
    dd(1:n-m)=t(1:n-m)-c(2:n-m+1)                   tested in the initializing loop.
    if (any(dd(1:n-m) == 0.0)) &
        call nrerror('failure in ratint')       This error condition indicates that
    dd(1:n-m)=(c(2:n-m+1)-d(1:n-m))/dd(1:n-m)       the interpolating function has a
    d(1:n-m)=c(2:n-m+1)*dd(1:n-m)                   pole at the requested value of
    c(1:n-m)=t(1:n-m)*dd(1:n-m)                     x.
    if (2*ns < n-m) then
        dy=c(ns+1)
    else
        dy=d(ns)
        ns=ns-1
    end if
    y=y+dy
end do
END SUBROUTINE ratint
```

⋆ ⋆ ⋆

```
SUBROUTINE spline(x,y,yp1,ypn,y2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: yp1,ypn
REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
```

Given arrays x and y of length N containing a tabulated function, i.e., $y_i = f(x_i)$, with $x_1 < x_2 < \ldots < x_N$, and given values yp1 and ypn for the first derivative of the interpolating function at points 1 and N, respectively, this routine returns an array y2 of length N that contains the second derivatives of the interpolating function at the tabulated points x_i. If yp1 and/or ypn are equal to 1×10^{30} or larger, the routine is signaled to set the corresponding boundary condition for a natural spline, with zero second derivative on that boundary.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(x)) :: a,b,c,r
n=assert_eq(size(x),size(y),size(y2),'spline')
c(1:n-1)=x(2:n)-x(1:n-1)              Set up the tridiagonal equations.
r(1:n-1)=6.0_sp*((y(2:n)-y(1:n-1))/c(1:n-1))
r(2:n-1)=r(2:n-1)-r(1:n-2)
a(2:n-1)=c(1:n-2)
b(2:n-1)=2.0_sp*(c(2:n-1)+a(2:n-1))
b(1)=1.0
b(n)=1.0
if (yp1 > 0.99e30_sp) then            The lower boundary condition is set either to be "nat-
    r(1)=0.0                              ural"
    c(1)=0.0
else                                  or else to have a specified first derivative.
    r(1)=(3.0_sp/(x(2)-x(1)))*((y(2)-y(1))/(x(2)-x(1))-yp1)
```

```
    c(1)=0.5
end if                              The upper boundary condition is set either to be
if (ypn > 0.99e30_sp) then              "natural"
    r(n)=0.0
    a(n)=0.0
else                                or else to have a specified first derivative.
    r(n)=(-3.0_sp/(x(n)-x(n-1)))*((y(n)-y(n-1))/(x(n)-x(n-1))-ypn)
    a(n)=0.5
end if
call tridag(a(2:n),b(1:n),c(1:n-1),r(1:n),y2(1:n))
END SUBROUTINE spline
```

```
FUNCTION splint(xa,ya,y2a,x)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY: locate
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
REAL(SP), INTENT(IN) :: x
REAL(SP) :: splint
```

Given the arrays xa and ya, which tabulate a function (with the xa_i's in increasing or decreasing order), and given the array y2a, which is the output from spline above, and given a value of x, this routine returns a cubic-spline interpolated value. The arrays xa, ya and y2a are all of the same size.

```
INTEGER(I4B) :: khi,klo,n
REAL(SP) :: a,b,h
n=assert_eq(size(xa),size(ya),size(y2a),'splint')
klo=max(min(locate(xa,x),n-1),1)
```

We will find the right place in the table by means of locate's bisection algorithm. This is optimal if sequential calls to this routine are at random values of x. If sequential calls are in order, and closely spaced, one would do better to store previous values of klo and khi and test if they remain appropriate on the next call.

```
khi=klo+1                       klo and khi now bracket the input value of x.
h=xa(khi)-xa(klo)
if (h == 0.0) call nrerror('bad xa input in splint')    The xa's must be distinct.
a=(xa(khi)-x)/h                 Cubic spline polynomial is now evaluated.
b=(x-xa(klo))/h
splint=a*ya(klo)+b*ya(khi)+((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.0_sp
END FUNCTION splint
```

f90 `klo=max(min(locate(xa,x),n-1),1)` In the Fortran 77 version of `splint`, there is in-line code to find the location in the table by bisection. Here we prefer an explicit call to `locate`, which performs the bisection. On some massively multiprocessor (MMP) machines, one might substitute a different, more parallel algorithm (see next note).

⋆ ⋆ ⋆

```
FUNCTION locate(xx,x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), INTENT(IN) :: x
INTEGER(I4B) :: locate
```

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.

```
INTEGER(I4B) :: n,jl,jm,ju
LOGICAL :: ascnd
```

```
n=size(xx)
ascnd = (xx(n) >= xx(1))          True if ascending order of table, false otherwise.
jl=0                              Initialize lower
ju=n+1                            and upper limits.
do
    if (ju-jl <= 1) exit          Repeat until this condition is satisfied.
    jm=(ju+jl)/2                  Compute a midpoint,
    if (ascnd .eqv. (x >= xx(jm))) then
        jl=jm                     and replace either the lower limit
    else
        ju=jm                     or the upper limit, as appropriate.
    end if
end do
if (x == xx(1)) then              Then set the output, being careful with the endpoints.
    locate=1
else if (x == xx(n)) then
    locate=n-1
else
    locate=jl
end if
END FUNCTION locate
```

The use of bisection is perhaps a sin on a genuinely parallel machine, but (since the process takes only logarithmically many sequential steps) it is at most a *small* sin. One can imagine a "fully parallel" implementation like,

```
k=iminloc(abs(x-xx))
if ((x < xx(k)) .eqv. (xx(1) < xx(n))) then
        locate=k-1
else
        locate=k
end if
```

Problem is, unless the number of *physical* (not logical) processors participating in the `iminloc` is larger than N, the length of the array, this "parallel" code turns a $\log N$ algorithm into one scaling as N, quite an unacceptable inefficiency. So we prefer to be small sinners and bisect.

```
SUBROUTINE hunt(xx,x,jlo)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: jlo
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
   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 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(I4B) :: n,inc,jhi,jm
LOGICAL :: ascnd
n=size(xx)
ascnd = (xx(n) >= xx(1))                True if ascending order of table, false otherwise.
if (jlo <= 0 .or. jlo > n) then         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) .eqv. ascnd) then      Hunt up:
        do
            jhi=jlo+inc
            if (jhi > n) then           Done hunting, since off end of table.
```

```
            jhi=n+1
            exit
        else
            if (x < xx(jhi) .eqv. ascnd) exit
            jlo=jhi                 Not done hunting,
            inc=inc+inc             so double the increment
        end if
    end do                          and try again.
else                                Hunt down:
    jhi=jlo
    do
        jlo=jhi-inc
        if (jlo < 1) then           Done hunting, since off end of table.
            jlo=0
            exit
        else
            if (x >= xx(jlo) .eqv. ascnd) exit
            jhi=jlo                 Not done hunting,
            inc=inc+inc             so double the increment
        end if
    end do                          and try again.
end if
end if                              Done hunting, value bracketed.
do                                  Hunt is done, so begin the final bisection phase:
    if (jhi-jlo <= 1) then
        if (x == xx(n)) jlo=n-1
        if (x == xx(1)) jlo=1
        exit
    else
        jm=(jhi+jlo)/2
        if (x >= xx(jm) .eqv. ascnd) then
            jlo=jm
        else
            jhi=jm
        end if
    end if
end do
END SUBROUTINE hunt
```

⋆ ⋆ ⋆

```
FUNCTION polcoe(x,y)
USE nrtype; USE nrutil, ONLY : assert_eq,outerdiff
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: polcoe
```

Given same-size arrays x and y containing a tabulated function $y_i = f(x_i)$, this routine returns a same-size array of coefficients c_j, such that $y_i = \sum_j c_j x_i^{j-1}$.

```
INTEGER(I4B) :: i,k,n
REAL(SP), DIMENSION(size(x)) :: s
REAL(SP), DIMENSION(size(x),size(x)) :: a
n=assert_eq(size(x),size(y),'polcoe')
s=0.0                         Coefficients s_i of the master polynomial P(x) are found by
s(n)=-x(1)                       recurrence.
do i=2,n
    s(n+1-i:n-1)=s(n+1-i:n-1)-x(i)*s(n+2-i:n)
    s(n)=s(n)-x(i)
end do
a=outerdiff(x,x)              Make vector w_j = prod_{j/=n}(x_j - x_n), using polcoe for tempo-
polcoe=product(a,dim=2,mask=a /= 0.0)        rary storage.
```

```
   Now do synthetic division by x − x_j. The division for all x_j can be done in parallel (on a
   parallel machine), since the : in the loop below is over j.
a(:,1)=-s(1)/x(:)
do k=2,n
    a(:,k)=-(s(k)-a(:,k-1))/x(:)
end do
s=y/polcoe
polcoe=matmul(s,a)             Solve linear system.
END FUNCTION polcoe
```

For a description of the coding here, see §22.3, especially equation (22.3.9). You might also want to compare the coding here with the Fortran 77 version, and also look at the description of the method on p. 84 in Volume 1. The Fortran 90 implementation here is in fact much closer to that description than is the Fortran 77 method, which goes through some acrobatics to roll the synthetic division and matrix multiplication into a single set of two nested loops. The price we pay, here, is storage for the matrix a. Since the degree of any useful polynomial is not a very large number, this is essentially no penalty.

Also worth noting is the way that parallelism is brought to the required synthetic division. For a *single* such synthetic division (e.g., as accomplished by the nrutil routine poly_term), parallelism can be obtained only by recursion. Here things are much simpler, because we need a whole bunch of simultaneous and independent synthetic divisions; so we can just do them in the obvious, data-parallel, way.

```
FUNCTION polcof(xa,ya)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), DIMENSION(size(xa)) :: polcof
   Given same-size arrays xa and ya containing a tabulated function ya_i = f(xa_i), this routine
   returns a same-size array of coefficients c_j such that ya_i = Σ_j c_j xa_i^(j−1).
INTEGER(I4B) :: j,k,m,n
REAL(SP) :: dy
REAL(SP), DIMENSION(size(xa)) :: x,y
n=assert_eq(size(xa),size(ya),'polcof')
x=xa
y=ya
do j=1,n
    m=n+1-j
    call polint(x(1:m),y(1:m),0.0_sp,polcof(j),dy)
      Use the polynomial interpolation routine of §3.1 to extrapolate to x = 0.
    k=iminloc(abs(x(1:m)))              Find the remaining x_k of smallest absolute value,
    where (x(1:m) /= 0.0) y(1:m)=(y(1:m)-polcof(j))/x(1:m)       reduce all the terms,
    y(k:m-1)=y(k+1:m)                   and eliminate x_k.
    x(k:m-1)=x(k+1:m)
end do
END FUNCTION polcof
```

⋆ ⋆ ⋆

```
SUBROUTINE polin2(x1a,x2a,ya,x1,x2,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays x1a of length M and x2a of length N of independent variables, and an $M \times N$ array of function values ya, tabulated at the grid points defined by x1a and x2a, and given values x1 and x2 of the independent variables, this routine returns an interpolated function value y, and an accuracy indication dy (based only on the interpolation in the x1 direction, however).

```
INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: ymtmp
REAL(SP), DIMENSION(size(x2a)) :: yntmp
m=assert_eq(size(x1a),size(ya,1),'polin2: m')
ndum=assert_eq(size(x2a),size(ya,2),'polin2: ndum')
do j=1,m                                      Loop over rows.
    yntmp=ya(j,:)                             Copy row into temporary storage.
    call polint(x2a,yntmp,x2,ymtmp(j),dy)     Interpolate answer into temporary stor-
end do                                            age.
call polint(x1a,ymtmp,x1,y,dy)                Do the final interpolation.
END SUBROUTINE polin2
```

⋆ ⋆ ⋆

```
SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: d1,d2
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c
```

Given arrays y, y1, y2, and y12, each of length 4, 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 4×4 table c that is used by routine bcuint for bicubic interpolation.

```
REAL(SP), DIMENSION(16) :: x
REAL(SP), DIMENSION(16,16) :: wt
DATA wt /1,0,-3,2,4*0,-3,0,9,-6,2,0,-6,4,&
    8*0,3,0,-9,6,-2,0,6,-4,10*0,9,-6,2*0,-6,4,2*0,3,-2,6*0,-9,6,&
    2*0,6,-4,4*0,1,0,-3,2,-2,0,6,-4,1,0,-3,2,8*0,-1,0,3,-2,1,0,-3,&
    2,10*0,-3,2,2*0,3,-2,6*0,3,-2,2*0,-6,4,2*0,3,-2,0,1,-2,1,5*0,&
    -3,6,-3,0,2,-4,2,9*0,3,-6,3,0,-2,4,-2,10*0,-3,3,2*0,2,-2,2*0,&
    -1,1,6*0,3,-3,2*0,-2,2,5*0,1,-2,1,0,-2,4,-2,0,1,-2,1,9*0,-1,2,&
    -1,0,1,-2,1,10*0,1,-1,2*0,-1,1,6*0,-1,1,2*0,2,-2,2*0,-1,1/
x(1:4)=y                                Pack a temporary vector x.
x(5:8)=y1*d1
x(9:12)=y2*d2
x(13:16)=y12*d1*d2
x=matmul(wt,x)                          Matrix multiply by the stored table.
c=reshape(x,(/4,4/),order=(/2,1/))      Unpack the result into the output table.
END SUBROUTINE bcucof
```

f90 `x=matmul(wt,x) ... c=reshape(x,(/4,4/),order=(/2,1/))` It is a powerful technique to combine the `matmul` intrinsic with `reshape`'s of the input or output. The idea is to use `matmul` whenever the calculation can be cast into the form of a linear mapping between input and output objects. Here the `order=(/2,1/)` parameter specifies that we want the packing to be by rows, not by Fortran's default of columns. (In this two-dimensional case, it's the equivalent of applying `transpose`.)

```
SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,ansy1,ansy2)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : bcucof
IMPLICIT NONE
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2
REAL(SP), INTENT(OUT) :: ansy,ansy1,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`.

```
INTEGER(I4B) :: i
REAL(SP) :: t,u
REAL(SP), DIMENSION(4,4) :: c
call bcucof(y,y1,y2,y12,x1u-x1l,x2u-x2l,c)       Get the c's.
if (x1u == x1l .or. x2u == x2l) call &
    nrerror('bcuint: problem with input values - boundary pair equal?')
t=(x1-x1l)/(x1u-x1l)                              Equation (3.6.4).
u=(x2-x2l)/(x2u-x2l)
ansy=0.0
ansy2=0.0
ansy1=0.0
do i=4,1,-1                                       Equation (3.6.6).
    ansy=t*ansy+((c(i,4)*u+c(i,3))*u+c(i,2))*u+c(i,1)
    ansy2=t*ansy2+(3.0_sp*c(i,4)*u+2.0_sp*c(i,3))*u+c(i,2)
    ansy1=u*ansy1+(3.0_sp*c(4,i)*t+2.0_sp*c(3,i))*t+c(2,i)
end do
ansy1=ansy1/(x1u-x1l)
ansy2=ansy2/(x2u-x2l)
END SUBROUTINE bcuint
```

⋆ ⋆ ⋆

```
SUBROUTINE splie2(x1a,x2a,ya,y2a)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : spline
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: y2a
```

Given an $M \times N$ tabulated function `ya`, and N tabulated independent variables `x2a`, this routine constructs one-dimensional natural cubic splines of the rows of `ya` and returns the second derivatives in the $M \times N$ array `y2a`. (The array `x1a` is included in the argument list merely for consistency with routine `splin2`.)

```
INTEGER(I4B) :: j,m,ndum
m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splie2: m')
ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splie2: ndum')
do j=1,m
    call spline(x2a,ya(j,:),1.0e30_sp,1.0e30_sp,y2a(j,:))
```

Values 1×10^{30} signal a natural spline.

```
end do
END SUBROUTINE splie2
```

```
FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : spline,splint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya,y2a
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP) :: splin2
```

Given x1a, x2a, ya as described in splie2 and y2a as produced by that routine; and given a desired interpolating point x1,x2; this routine returns an interpolated function value by bicubic spline interpolation.

```
INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: yytmp,y2tmp2
m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splin2: m')
ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splin2: ndum')
do j=1,m
    yytmp(j)=splint(x2a,ya(j,:),y2a(j,:),x2)
```

Perform m evaluations of the row splines constructed by splie2, using the one-dimensional spline evaluator splint.

```
end do
call spline(x1a,yytmp,1.0e30_sp,1.0e30_sp,y2tmp2)
```

Construct the one-dimensional column spline and evaluate it.

```
splin2=splint(x1a,yytmp,y2tmp2,x1)
END FUNCTION splin2
```