Chapter B5. Evaluation of Functions

```
SUBROUTINE eulsum(sum,term,jterm)
USE nrtype; USE nrutil, ONLY : poly_term,reallocate
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: sum
REAL(SP), INTENT(IN) :: term
INTEGER(I4B), INTENT(IN) :: jterm
    Incorporates into sum the jterm'th term, with value term, of an alternating series. sum
    is input as the previous partial sum, and is output as the new partial sum. The first call
    to this routine, with the first term in the series, should be with jterm=1. On the second
    call, term should be set to the second term of the series, with sign opposite to that of the
    first call, and jterm should be 2. And so on.
REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
INTEGER(I4B), SAVE :: nterm                   Number of saved differences in wksp.
LOGICAL(LGT), SAVE :: init=.true.
if (init) then                                Initialize.
    init=.false.
    nullify(wksp)
end if
if (jterm == 1) then
    nterm=1
    wksp=>reallocate(wksp,100)
    wksp(1)=term
    sum=0.5_sp*term                           Return first estimate.
else
    if (nterm+1 > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
    wksp(2:nterm+1)=0.5_sp*wksp(1:nterm)      Update saved quantities by van Wijn-
    wksp(1)=term                                  gaarden's algorithm.
    wksp(1:nterm+1)=poly_term(wksp(1:nterm+1),0.5_sp)
    if (abs(wksp(nterm+1)) <= abs(wksp(nterm))) then    Favorable to increase p,
        sum=sum+0.5_sp*wksp(nterm+1)
        nterm=nterm+1                         and the table becomes longer.
    else                                      Favorable to increase n,
        sum=sum+wksp(nterm+1)                 the table doesn't become longer.
    end if
end if
END SUBROUTINE eulsum
```

f90 This routine uses the function `reallocate` in `nrutil` to define a temporary workspace and then, if necessary, enlarge the workspace without destroying the earlier contents. The pointer `wksp` is declared with the `SAVE` attribute. Since Fortran 90 pointers are born "in limbo," we cannot immediately test whether they are associated or not. Hence the code `if (init)...nullify(wksp)`. Then the line `wksp=>reallocate(wksp,100)` allocates an array of length 100 and points `wksp` to it. On subsequent calls to `eulsum`, if `nterm` ever gets bigger than the size of `wksp`, the call to `reallocate` doubles the size of `wksp` and copies the old contents into the new storage.

You could achieve the same effect as the code `if (init)...nullify(wksp)... wksp=>reallocate(wksp,100)` with a simple `allocate(wksp,100)`. You would then use `reallocate` only for increasing the storage if necessary. Don't! The advantage of the above scheme becomes clear if you consider what happens if `eulsum` is invoked *twice* by the calling program to evaluate two different sums. On the second invocation, when `jterm` $= 1$ again, you would be allocating an already allocated pointer. This does not generate an error — it simply leaves the original target inaccessible. Using `reallocate` instead not only allocates a new array of length 100, but also detects that `wksp` had already been associated. It dutifully (and wastefully) copies the first 100 elements of the old `wksp` into the new storage, and, more importantly, deallocates the old `wksp`, reclaiming its storage. While only two invocations of `eulsum` without intervening deallocation of memory would not cause a problem, many such invocations might well. We believe that, as a general rule, the potential for catastrophe from reckless use of `allocate` is great enough that you should *always* `deallocate` whenever storage is no longer required.

The unnecessary copying of 100 elements when `eulsum` is invoked a second time could be avoided by making `init` an argument. It hardly seems worth it to us.

For Fortran 90 neophytes, note that unlike in C you have to do nothing special to get the contents of the storage a pointer is addressing. The compiler figures out from the context whether you mean the contents, such as `wksp(1:nterm)`, or the address, such as both occurrences of `wksp` in `wksp=>reallocate(wksp,100)`.

`wksp(1:nterm+1)=poly_term(wksp(1:nterm+1),0.5_sp)` The `poly_term` function in `nrutil` tabulates the partial sums of a polynomial, or, equivalently, performs the synthetic division of a polynomial by a monomial.

Small-scale parallelism in `eulsum` is achieved straightforwardly by the use of vector constructions and `poly_term` (which parallelizes recursively). The routine is not written to take advantage of data parallelism in the (infrequent) case of wanting to sum many different series simultaneously; nor, since `wksp` is a `SAVE`d variable, can it be used in many simultaneous instances on a MIMD machine. (You can easily recode these generalizations if you need them.)

⋆ ⋆ ⋆

```
SUBROUTINE ddpoly(c,x,pd)
USE nrtype; USE nrutil, ONLY : arth,cumprod,poly_term
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(:), INTENT(OUT) :: pd
```

Given the coefficients of a polynomial of degree $N_c - 1$ as an array `c(1:`N_c`)` with `c(1)` being the constant term, and given a value `x`, this routine returns the polynomial evaluated at `x` as `pd(1)` and $N_d - 1$ derivatives as `pd(2:`N_d`)`.

```
INTEGER(I4B) :: i,nc,nd
REAL(SP), DIMENSION(size(pd)) :: fac
REAL(SP), DIMENSION(size(c)) :: d
nc=size(c)
nd=size(pd)
d(nc:1:-1)=poly_term(c(nc:1:-1),x)
do i=2,min(nd,nc)
    d(nc:i:-1)=poly_term(d(nc:i:-1),x)
end do
pd=d(1:nd)
fac=cumprod(arth(1.0_sp,1.0_sp,nd))        After the first derivative, factorial constants
pd(3:nd)=fac(2:nd-1)*pd(3:nd)                  come in.
END SUBROUTINE ddpoly
```

f90 `d(nc:1:-1)=poly_term(c(nc:1:-1),x)` The `poly_term` function in `nrutil` tabulates the partial sums of a polynomial, or, equivalently, performs synthetic division. See §22.3 for a discussion of why `ddpoly` is coded this way.

`fac=cumprod(arth(1.0_sp,1.0_sp,nd))` Here the function `arth` from `nrutil` generates the sequence $1, 2, 3 \dots$. The function `cumprod` then tabulates the cumulative products, thus making a table of factorials.

Notice that `ddpoly` doesn't need an argument to pass N_d, the number of output terms desired by the user: It gets that information from the length of the array `pd` that the user provides for it to fill. It is a minor curiosity that `pd`, declared as `INTENT(OUT)`, can thus be used, on the sly, to pass some `INTENT(IN)` information. (A Fortran 90 brain teaser could be: A subroutine with only `INTENT(OUT)` arguments can be called to print any specified integer. How is this done?)

```
SUBROUTINE poldiv(u,v,q,r)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(OUT) :: q,r
    Given the N coefficients of a polynomial in u, and the N_v coefficients of another polynomial
    in v, divide the polynomial u by the polynomial v ("u"/"v") giving a quotient polynomial
    whose coefficients are returned in q, and a remainder polynomial whose coefficients are
    returned in r. The arrays q and r are of length N, but only the first N - N_v + 1 elements
    of q and the first N_v - 1 elements of r are used. The remaining elements are returned
    as zero.
INTEGER(I4B) :: i,n,nv
n=assert_eq(size(u),size(q),size(r),'poldiv')
nv=size(v)
r(:)=u(:)
q(:)=0.0
do i=n-nv,0,-1
    q(i+1)=r(nv+i)/v(nv)
    r(i+1:nv+i-1)=r(i+1:nv+i-1)-q(i+1)*v(1:nv-1)
end do
r(nv:n)=0.0
END SUBROUTINE poldiv
```

⋆ ⋆ ⋆

```
FUNCTION ratval_s(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), INTENT(IN) :: x            Note precision! Change to REAL(SP) if desired.
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP) :: ratval_s
    Given mm, kk, and cof(1:mm+kk+1), evaluate and return the rational function (cof(1)+
    cof(2)x+···+cof(mm+1)x^mm)/(1+cof(mm+2)x+···+cof(mm+kk+1)x^kk).
ratval_s=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_s
```

f90 This simple routine uses the function `poly` from `nrutil` to evaluate the numerator and denominator polynomials. Single- and double-precision versions, `ratval_s` and `ratval_v`, are overloaded onto the name `ratval` when the module `nr` is used.

```
FUNCTION ratval_v(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP), DIMENSION(size(x)) :: ratval_v
ratval_v=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_v
```

⋆ ⋆ ⋆

The routines `recur1` and `recur2` are new in this volume, and do not have Fortran 77 counterparts. First- and second-order linear recurrences are implemented as trivial do-loops on strictly serial machines. On parallel machines, however, they pose different, and quite interesting, programming challenges. Since many calculations can be decomposed into recurrences, it is useful to have general, parallelizable routines available. The algorithms behind `recur1` and `recur2` are discussed in §22.2.

```
RECURSIVE FUNCTION recur1(a,b) RESULT(u)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B), PARAMETER :: NPAR_RECUR1=8
   Given vectors a of size $n$ and b of size $n-1$, returns a vector $u$ that satisfies the first
   order linear recurrence $u_1 = a_1$, $u_j = a_j + b_{j-1}u_{j-1}$, for $j = 2,\ldots,n$. Parallelization is
   via a recursive evaluation.
INTEGER(I4B) :: n,j
n=assert_eq(size(a),size(b)+1,'recur1')
u(1)=a(1)
if (n < NPAR_RECUR1) then          Do short vectors as a loop.
    do j=2,n
        u(j)=a(j)+b(j-1)*u(j-1)
    end do
else
      Otherwise, combine coefficients and recurse on the even components, then evaluate all
      the odd components in parallel.
    u(2:n:2)=recur1(a(2:n:2)+a(1:n-1:2)*b(1:n-1:2), &
            b(3:n-1:2)*b(2:n-2:2))
    u(3:n:2)=a(3:n:2)+b(2:n-1:2)*u(2:n-1:2)
end if
END FUNCTION recur1
```

f90 `RECURSIVE FUNCTION recur1(a,b) RESULT(u)` When a recursive function invokes itself only indirectly through a sequence of function calls, then the function name can be used for the result just as in a nonrecursive function. When the function invokes itself directly, however, as in `recur1`, then another name must be used for the result. If you are hazy on the syntax for `RESULT`, see the discussion of recursion in §21.5.

⋆ ⋆ ⋆

```
FUNCTION recur2(a,b,c)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
REAL(SP), DIMENSION(size(a)) :: recur2
```

Given vectors a of size n and b and c of size $n-2$, returns a vector u that satisfies the second order linear recurrence $u_1 = a_1$, $u_2 = a_2$, $u_j = a_j + b_{j-2}u_{j-1} + c_{j-2}u_{j-2}$, for $j = 3, \dots, n$. Parallelization is via conversion to a first order recurrence for a two-dimensional vector.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(a)-1) :: a1,a2,u1,u2
REAL(SP), DIMENSION(size(a)-2) :: b11,b12,b21,b22
n=assert_eq(size(a),size(b)+2,size(c)+2,'recur2')
a1(1)=a(1)                          Set up vector a.
a2(1)=a(2)
a1(2:n-1)=0.0
a2(2:n-1)=a(3:n)
b11(1:n-2)=0.0                      Set up matrix b.
b12(1:n-2)=1.0
b21(1:n-2)=c(1:n-2)
b22(1:n-2)=b(1:n-2)
call recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
recur2(1:n-1)=u1(1:n-1)
recur2(n)=u2(n-1)
CONTAINS

RECURSIVE SUBROUTINE recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a1,a2,b11,b12,b21,b22
REAL(SP), DIMENSION(:), INTENT(OUT) :: u1,u2
INTEGER(I4B), PARAMETER :: NPAR_RECUR2=8
```

Used by `recur2` to evaluate first order vector recurrence. Routine is a two-dimensional vector version of `recur1`, with matrix multiplication replacing scalar multiplication.

```
INTEGER(I4B) :: n,j,nn,nn1
REAL(SP), DIMENSION(size(a1)/2) :: aa1,aa2
REAL(SP), DIMENSION(size(a1)/2-1) :: bb11,bb12,bb21,bb22
n=assert_eq((/size(a1),size(a2),size(b11)+1,size(b12)+1,size(b21)+1,&
    size(b22)+1,size(u1),size(u2)/),'recur1_v')
u1(1)=a1(1)
u2(1)=a2(1)
if (n < NPAR_RECUR2) then           Do short vectors as a loop.
    do j=2,n
        u1(j)=a1(j)+b11(j-1)*u1(j-1)+b12(j-1)*u2(j-1)
        u2(j)=a2(j)+b21(j-1)*u1(j-1)+b22(j-1)*u2(j-1)
    end do
else
```

Otherwise, combine coefficients and recurse on the even components, then evaluate all the odd components in parallel.

```
    nn=n/2
    nn1=nn-1
    aa1(1:nn)=a1(2:n:2)+b11(1:n-1:2)*a1(1:n-1:2)+&
        b12(1:n-1:2)*a2(1:n-1:2)
    aa2(1:nn)=a2(2:n:2)+b21(1:n-1:2)*a1(1:n-1:2)+&
            b22(1:n-1:2)*a2(1:n-1:2)
    bb11(1:nn1)=b11(3:n-1:2)*b11(2:n-2:2)+&
            b12(3:n-1:2)*b21(2:n-2:2)
    bb12(1:nn1)=b11(3:n-1:2)*b12(2:n-2:2)+&
            b12(3:n-1:2)*b22(2:n-2:2)
    bb21(1:nn1)=b21(3:n-1:2)*b11(2:n-2:2)+&
            b22(3:n-1:2)*b21(2:n-2:2)
    bb22(1:nn1)=b21(3:n-1:2)*b12(2:n-2:2)+&
            b22(3:n-1:2)*b22(2:n-2:2)
    call recur1_v(aa1,aa2,bb11,bb12,bb21,bb22,u1(2:n:2),u2(2:n:2))
    u1(3:n:2)=a1(3:n:2)+b11(2:n-1:2)*u1(2:n-1:2)+&
```

```
           b12(2:n-1:2)*u2(2:n-1:2)
    u2(3:n:2)=a2(3:n:2)+b21(2:n-1:2)*u1(2:n-1:2)+&
           b22(2:n-1:2)*u2(2:n-1:2)
end if
END SUBROUTINE recur1_v
END FUNCTION recur2
```

⋆ ⋆ ⋆

```
FUNCTION dfridr(func,x,h,err)
USE nrtype; USE nrutil, ONLY : assert,geop,iminloc
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,h
REAL(SP), INTENT(OUT) :: err
REAL(SP) :: dfridr
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B),PARAMETER :: NTAB=10
REAL(SP), PARAMETER :: CON=1.4_sp,CON2=CON*CON,BIG=huge(x),SAFE=2.0
```

Returns the derivative of a function `func` at a point `x` by Ridders' method of polynomial extrapolation. The value `h` is input as an estimated initial stepsize; it need not be small, but rather should be an increment in `x` over which `func` changes *substantially*. An estimate of the error in the derivative is returned as `err`.

Parameters: Stepsize is decreased by `CON` at each iteration. Max size of tableau is set by `NTAB`. Return when error is `SAFE` worse than the best so far.

```
INTEGER(I4B) :: ierrmin,i,j
REAL(SP) :: hh
REAL(SP), DIMENSION(NTAB-1) :: errt,fac
REAL(SP), DIMENSION(NTAB,NTAB) :: a
call assert(h /= 0.0, 'dfridr arg')
hh=h
a(1,1)=(func(x+hh)-func(x-hh))/(2.0_sp*hh)
err=BIG
fac(1:NTAB-1)=geop(CON2,CON2,NTAB-1)
do i=2,NTAB                    Successive columns in the Neville tableau will go to smaller
    hh=hh/CON                      stepsizes and higher orders of extrapolation.
    a(1,i)=(func(x+hh)-func(x-hh))/(2.0_sp*hh)        Try new, smaller stepsize.
    do j=2,i
          Compute extrapolations of various orders, requiring no new function evaluations.
        a(j,i)=(a(j-1,i)*fac(j-1)-a(j-1,i-1))/(fac(j-1)-1.0_sp)
    end do
    errt(1:i-1)=max(abs(a(2:i,i)-a(1:i-1,i)),abs(a(2:i,i)-a(1:i-1,i-1)))
      The error strategy is to compare each new extrapolation to one order lower, both at the
      present stepsize and the previous one.
    ierrmin=iminloc(errt(1:i-1))
    if (errt(ierrmin) <= err) then         If error is decreased, save the improved an-
        err=errt(ierrmin)                      swer.
        dfridr=a(1+ierrmin,i)
    end if
    if (abs(a(i,i)-a(i-1,i-1)) >= SAFE*err) RETURN
      If higher order is worse by a significant factor SAFE, then quit early.
end do
END FUNCTION dfridr
```

f90 `ierrmin=iminloc(errt(1:i-1))` The function `iminloc` in `nrutil` is useful when you need to know the index of the smallest element in an array.

⋆ ⋆ ⋆

```
FUNCTION chebft(a,b,n,func)
USE nrtype; USE nrutil, ONLY : arth,outerprod
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: chebft
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval [a,b], and a maximum degree `n`, this routine computes the `n` coefficients c_k such that `func`$(x) \approx [\sum_{k=1}^{n} c_k T_{k-1}(y)] - c_1/2$, where y and x are related by (5.8.10). This routine is to be used with moderately large `n` (e.g., 30 or 50), the array of c's subsequently to be truncated at the smaller value m such that c_{m+1} and subsequent elements are negligible.

```
REAL(DP) :: bma,bpa
REAL(DP), DIMENSION(n) :: theta
bma=0.5_dp*(b-a)
bpa=0.5_dp*(b+a)
theta(:)=PI_D*arth(0.5_dp,1.0_dp,n)/n
chebft(:)=matmul(cos(outerprod(arth(0.0_dp,1.0_dp,n),theta)), &
    func(real(cos(theta)*bma+bpa,sp)))*2.0_dp/n
```

We evaluate the function at the `n` points required by (5.8.7). We accumulate the sum in double precision for safety.

```
END FUNCTION chebft
```

f90 `chebft(:)=matmul(...)` Here again Fortran 90 produces a very concise parallelizable formulation that requires some effort to decode. Equation (5.8.7) is a product of the matrix of cosines, where the rows are indexed by j and the columns by k, with the vector of function values indexed by k. We use the `outerprod` function in `nrutil` to form the matrix of arguments for the cosine, and rely on the element-by-element application of `cos` to produce the matrix of cosines. `matmul` then takes care of the matrix product. A subtlety is that, while the calculation is being done in double precision to minimize roundoff, the function is assumed to be supplied in single precision. Thus `real(...,sp)` is used to convert the double precision argument to single precision.

```
FUNCTION chebev_s(a,b,c,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP) :: chebev_s
```

Chebyshev evaluation: All arguments are input. `c` is an array of length M of Chebyshev coefficients, the first M elements of `c` output from `chebft` (which must have been called with the same a and b). The Chebyshev polynomial $\sum_{k=1}^{M} c_k T_{k-1}(y) - c_1/2$ is evaluated at a point $y = [x - (b + a)/2]/[(b - a)/2]$, and the result is returned as the function value.

```
INTEGER(I4B) :: j,m
REAL(SP) :: d,dd,sv,y,y2
if ((x-a)*(x-b) > 0.0) call nrerror('x not in range in chebev_s')
m=size(c)
d=0.0
dd=0.0
y=(2.0_sp*x-a-b)/(b-a)              Change of variable.
y2=2.0_sp*y
do j=m,2,-1                         Clenshaw's recurrence.
    sv=d
    d=y2*d-dd+c(j)
    dd=sv
end do
chebev_s=y*d-dd+0.5_sp*c(1)         Last step is different.
END FUNCTION chebev_s
```

```
FUNCTION chebev_v(a,b,c,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
REAL(SP), DIMENSION(size(x)) :: chebev_v
INTEGER(I4B) :: j,m
REAL(SP), DIMENSION(size(x)) :: d,dd,sv,y,y2
if (any((x-a)*(x-b) > 0.0)) call nrerror('x not in range in chebev_v')
m=size(c)
d=0.0
dd=0.0
y=(2.0_sp*x-a-b)/(b-a)
y2=2.0_sp*y
do j=m,2,-1
    sv=d
    d=y2*d-dd+c(j)
    dd=sv
end do
chebev_v=y*d-dd+0.5_sp*c(1)
END FUNCTION chebev_v
```

f90 The name chebev is overloaded with scalar and vector versions. chebev_v is essentially identical to chebev_s except for the declarations of the variables. Fortran 90 does the appropriate scalar or vector arithmetic in the body of the routine, depending on the type of the variables.

⋆ ⋆ ⋆

```
FUNCTION chder(a,b,c)
USE nrtype; USE nrutil, ONLY : arth,cumsum
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chder
```

This routine returns an array of length N containing the Chebyshev coefficients of the derivative of the function whose coefficients are in the array c. Input are a,b,c, as output from routine `chebft` §5.8. The desired degree of approximation N is equal to the length of `c` supplied.

```
INTEGER(I4B) :: n
REAL(SP) :: con
REAL(SP), DIMENSION(size(c)) :: temp
n=size(c)
temp(1)=0.0
temp(2:n)=2.0_sp*arth(n-1,-1,n-1)*c(n:2:-1)
chder(n:1:-2)=cumsum(temp(1:n:2))          Equation (5.9.2).
chder(n-1:1:-2)=cumsum(temp(2:n:2))
con=2.0_sp/(b-a)
chder=chder*con                            Normalize to the interval b-a.
END FUNCTION chder
```

```
FUNCTION chint(a,b,c)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chint
```

This routine returns an array of length N containing the Chebyshev coefficients of the integral of the function whose coefficients are in the array `c`. Input are `a,b,c`, as output from routine `chebft` §5.8. The desired degree of approximation N is equal to the length of `c` supplied. The constant of integration is set so that the integral vanishes at `a`.

```
INTEGER(I4B) :: n
REAL(SP) :: con
n=size(c)
con=0.25_sp*(b-a)                          Factor that normalizes to the interval b-a.
chint(2:n-1)=con*(c(1:n-2)-c(3:n))/arth(1,1,n-2)     Equation (5.9.1).
chint(n)=con*c(n-1)/(n-1)                            Special case of (5.9.1) for n.
chint(1)=2.0_sp*(sum(chint(2:n:2))-sum(chint(3:n:2)))       Set the constant of inte-
END FUNCTION chint                                             gration.
```

f90 If you look at equation (5.9.1) for the Chebyshev coefficients of the integral of a function, you will see c_{i-1} and c_{i+1} and be tempted to use `eoshift`. We think it is almost always better to use array sections instead, as in the code above, especially if your code will ever run on a serial machine.

⋆ ⋆ ⋆

```
FUNCTION chebpc(c)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chebpc
```

Chebyshev polynomial coefficients. Given a coefficient array `c` of length N, this routine returns a coefficient array d of length N such that $\sum_{k=1}^{N} d_k y^{k-1} = \sum_{k=1}^{N} \mathtt{c}_k T_{k-1}(y) - \mathtt{c}_1/2$. The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than arithmetically.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(c)) :: dd,sv
n=size(c)
chebpc=0.0
dd=0.0
chebpc(1)=c(n)
do j=n-1,2,-1
    sv(2:n-j+1)=chebpc(2:n-j+1)
    chebpc(2:n-j+1)=2.0_sp*chebpc(1:n-j)-dd(2:n-j+1)
    dd(2:n-j+1)=sv(2:n-j+1)
    sv(1)=chebpc(1)
    chebpc(1)=-dd(1)+c(j)
    dd(1)=sv(1)
end do
chebpc(2:n)=chebpc(1:n-1)-dd(2:n)
chebpc(1)=-dd(1)+0.5_sp*c(1)
END FUNCTION chebpc
```

⋆ ⋆ ⋆

```
SUBROUTINE pcshft(a,b,d)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
```

Polynomial coefficient shift. Given a coefficient array d of length N, this routine generates a coefficient array g of the same length such that $\sum_{k=1}^{N} \mathrm{d}_k y^{k-1} = \sum_{k=1}^{N} g_k x^{k-1}$, where x and y are related by (5.8.10), i.e., the interval $-1 < y < 1$ is mapped to the interval $\mathrm{a} < x < \mathrm{b}$. The array g is returned in d.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(d)) :: dd
REAL(SP) :: x
n=size(d)
dd=d*geop(1.0_sp,2.0_sp/(b-a),n)
x=-0.5_sp*(a+b)
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1                      We accomplish the shift by synthetic division, that miracle of
    d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)         high-school algebra.
    d(1)=d(1)*x+dd(j)
end do
END SUBROUTINE pcshft
```

There is a subtle, but major, distinction between the synthetic division algorithm used in the Fortran 77 version of `pcshft` and that used above. In the Fortran 77 version, the synthetic division (translated to Fortran 90 notation) is

```
d(1:n)=dd(1:n)
do j=1,n-1
   do k=n-1,j,-1
      d(k)=x*d(k+1)+d(k)
   end do
end do
```

while, in Fortran 90, it is

```
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1
   d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)
   d(1)=d(1)*x+dd(j)
end do
```

As explained in §22.3, these are algebraically — but not algorithmically — equivalent. The inner loop in the Fortran 77 version does not parallelize, because each `k` value uses the result of the previous one. In fact, the `k` loop is a synthetic division, which can be parallelized *recursively* (as in the `nrutil` routine `poly_term`), but not simply vectorized. In the Fortran 90 version, since not one but `n-1` successive synthetic divisions are to be performed (by the outer loop), it is possible to reorganize the calculation to allow vectorization.

⋆ ⋆ ⋆

```
FUNCTION pccheb(d)
USE nrtype; USE nrutil, ONLY : arth,cumprod,geop
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(size(d)) :: pccheb
    Inverse of routine chebpc: given an array of polynomial coefficients d, returns an equivalent
    array of Chebyshev coefficients of the same length.
INTEGER(I4B) :: k,n
REAL(SP), DIMENSION(size(d)) :: denom,numer,pow
n=size(d)
pccheb(1)=2.0_sp*d(1)
pow=geop(1.0_sp,2.0_sp,n)           Powers of 2.
numer(1)=1.0                        Combinatorial coefficients computed as numer/denom.
denom(1)=1.0
denom(2:(n+3)/2)=cumprod(arth(1.0_sp,1.0_sp,(n+1)/2))
pccheb(2:n)=0.0
do k=2,n                            Loop over orders of x in the polynomial.
    numer(2:(k+3)/2)=cumprod(arth(k-1.0_sp,-1.0_sp,(k+1)/2))
    pccheb(k:1:-2)=pccheb(k:1:-2)+&
        d(k)/pow(k-1)*numer(1:(k+1)/2)/denom(1:(k+1)/2)
end do
END FUNCTION pccheb
```

⋆ ⋆ ⋆

```
SUBROUTINE pade(cof,resid)
USE nrtype
USE nr, ONLY : lubksb,ludcmp,mprove
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(INOUT) :: cof      DP for consistency with ratval.
REAL(SP), INTENT(OUT) :: resid
```

Given $\texttt{cof}(1:2N+1)$, the leading terms in the power series expansion of a function, solve the linear Padé equations to return the coefficients of a diagonal rational function approximation to the same function, namely $(\texttt{cof}(1)+\texttt{cof}(2)x+\cdots+\texttt{cof}(N+1)x^N)/(1+\texttt{cof}(N+2)x+\cdots+\texttt{cof}(2N+1)x^N)$. The value `resid` is the norm of the residual vector; a small value indicates a well-converged solution.

```
INTEGER(I4B) :: k,n
INTEGER(I4B), DIMENSION((size(cof)-1)/2) :: indx
REAL(SP), PARAMETER :: BIG=1.0e30_sp       A big number.
REAL(SP) :: d,rr,rrold
REAL(SP), DIMENSION((size(cof)-1)/2) :: x,y,z
REAL(SP), DIMENSION((size(cof)-1)/2,(size(cof)-1)/2) :: q,qlu
n=(size(cof)-1)/2
x=cof(n+2:2*n+1)                           Set up matrix for solving.
y=x
do k=1,n
    q(:,k)=cof(n+2-k:2*n+1-k)
end do
qlu=q
call ludcmp(qlu,indx,d)                    Solve by LU decomposition and backsubsti-
call lubksb(qlu,indx,x)                        tution.
rr=BIG
do                                         Important to use iterative improvement, since
    rrold=rr                                   the Padé equations tend to be ill-conditioned.
    z=x
    call mprove(q,qlu,indx,y,x)
    rr=sum((z-x)**2)                        Calculate residual.
    if (rr >= rrold) exit                   If it is no longer improving, call it quits.
end do
resid=sqrt(rrold)
do k=1,n                                    Calculate the remaining coefficients.
    y(k)=cof(k+1)-dot_product(z(1:k),cof(k:1:-1))
end do
cof(2:n+1)=y                                Copy answers to output.
cof(n+2:2*n+1)=-z
END SUBROUTINE pade
```

⋆ ⋆ ⋆

```
SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)
USE nrtype; USE nrutil, ONLY : arth,geop
USE nr, ONLY : ratval,svbksb,svdcmp
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(:), INTENT(OUT) :: cof
REAL(DP), INTENT(OUT) :: dev
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    REAL(DP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NPFAC=8,MAXIT=5
REAL(DP), PARAMETER :: BIG=1.0e30_dp
```

Returns in cof(1:mm+kk+1) the coefficients of a rational function approximation to the function func in the interval (a, b). Input quantities mm and kk specify the order of the numerator and denominator, respectively. The maximum absolute deviation of the approximation (insofar as is known) is returned as dev. Note that double-precision versions of svdcmp and svbksb are called.

```
INTEGER(I4B) :: it,ncof,npt,npth
REAL(DP) :: devmax,e,theta
REAL(DP), DIMENSION((mm+kk+1)*NPFAC) :: bb,ee,fs,wt,xs
REAL(DP), DIMENSION(mm+kk+1) :: coff,w
REAL(DP), DIMENSION(mm+kk+1,mm+kk+1) :: v
REAL(DP), DIMENSION((mm+kk+1)*NPFAC,mm+kk+1) :: u,temp
ncof=mm+kk+1
npt=NPFAC*ncof                          Number of points where function is evaluated,
npth=npt/2                                  i.e., fineness of the mesh.
dev=BIG
theta=PIO2_D/(npt-1)
xs(1:npth-1)=a+(b-a)*sin(theta*arth(0,1,npth-1))**2
```

Now fill arrays with mesh abscissas and function values. At each end, use formula that minimizes roundoff sensitivity in xs.

```
xs(npth:npt)=b-(b-a)*sin(theta*arth(npt-npth,-1,npt-npth+1))**2
fs=func(xs)
wt=1.0                                  In later iterations we will adjust these weights to
ee=1.0                                      combat the largest deviations.
e=0.0
do it=1,MAXIT                           Loop over iterations.
    bb=wt*(fs+sign(e,ee))
```

Key idea here: Fit to fn$(x) + e$ where the deviation is positive, to fn$(x) - e$ where it is negative. Then e is supposed to become an approximation to the equal-ripple deviation.

```
    temp=geop(spread(1.0_dp,1,npt),xs,ncof)
```

```
          Note that vector form of geop (returning matrix) is being used.
        u(:,1:mm+1)=temp(:,1:mm+1)*spread(wt,2,mm+1)
          Set up the "design matrix" for the least squares fit.
        u(:,mm+2:ncof)=-temp(:,2:ncof-mm)*spread(bb,2,ncof-mm-1)
        call svdcmp(u,w,v)
          Singular Value Decomposition. In especially singular or difficult cases, one might here
          edit the singular values w(1:ncof), replacing small values by zero.
        call svbksb(u,w,v,bb,coff)
        ee=ratval(xs,coff,mm,kk)-fs          Tabulate the deviations and revise the weights.
        wt=abs(ee)                           Use weighting to emphasize most deviant points.
        devmax=maxval(wt)
        e=sum(wt)/npt                        Update e to be the mean absolute deviation.
        if (devmax <= dev) then              Save only the best coefficient set found.
            cof=coff
            dev=devmax
        end if
        write(*,10) it,devmax
    end do
10  format (' ratlsq iteration=',i2,' max error=',1p,e10.3)
    END SUBROUTINE ratlsq
```

`temp=geop(spread(1.0_dp,1,npt),xs,ncof)` The design matrix u_{ij} is defined for $i = 1, \ldots, \texttt{npts}$ by

$$u_{ij} = \begin{cases} w_i x_i^{j-1}, & j = 1, \ldots, m+1 \\ -b_i x_i^{j-m-2}, & j = m+2, \ldots, n \end{cases} \tag{B5.12}$$

The first case in equation (B5.12) is computed in parallel by constructing the matrix `temp` equal to

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots \\ 1 & x_2 & x_2^2 & \cdots \\ 1 & x_3 & x_3^2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

and then multiplying by the matrix `spread(wt,2,mm+1)`, which is just

$$\begin{bmatrix} w_1 & w_1 & w_1 & \cdots \\ w_2 & w_2 & w_2 & \cdots \\ w_3 & w_3 & w_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

(Remember that multiplication using `*` means element-by-element multiplication, not matrix multiplication.) A similar construction is used for the second part of the design matrix.