Chapter B4. Integration of Functions

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

This routine computes the nth stage of refinement of an extended trapezoidal rule. `func` is input as the name of the function to be integrated between limits `a` and `b`, also input. When called with `n`=1, the routine returns as `s` the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls with `n`=2,3,... (in that sequential order) will improve the accuracy of `s` by adding 2^{n-2} additional interior points. `s` should not be modified between sequential calls.

```
REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
    s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
    it=2**(n-2)
    del=(b-a)/it                        This is the spacing of the points to be added.
    fsum=sum(func(arth(a+0.5_sp*del,del,it)))
    s=0.5_sp*(s+del*fsum)               This replaces s by its refined value.
end if
END SUBROUTINE trapzd
```

f90 While most of the quadrature routines in this chapter are coded as functions, `trapzd` is a subroutine because the argument `s` that returns the function value must also be supplied as an input parameter. We could change the subroutine into a function by declaring `s` to be a local variable with the `SAVE` attribute. However, this would prevent us from being able to use the routine recursively to do multidimensional quadrature (see `quad3d` on p. 1065). When `s` is left as an argument, a fresh copy is created on each recursive call. As a `SAVE`'d variable, by contrast, its value would get overwritten on each call, and the code would not be properly "re-entrant."

`s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))` Note how we use the `(/.../)` construct to supply a set of scalar arguments to a vector function.

⋆ ⋆ ⋆

```
FUNCTION qtrap(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qtrap
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp, UNLIKELY=-1.0e30_sp
    Returns the integral of the function func from a to b. The parameter EPS should be set to
    the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
    allowed number of steps. Integration is performed by the trapezoidal rule.
REAL(SP) :: olds
INTEGER(I4B) :: j
olds=UNLIKELY                          Any number that is unlikely to be the average of the
do j=1,JMAX                                function at its endpoints will do here.
    call trapzd(func,a,b,qtrap,j)
    if (j > 5) then                    Avoid spurious early convergence.
        if (abs(qtrap-olds) < EPS*abs(olds) .or. &
            (qtrap == 0.0 .and. olds == 0.0)) RETURN
    end if
    olds=qtrap
end do
call nrerror('qtrap: too many steps')
END FUNCTION qtrap
```

★ ★ ★

```
FUNCTION qsimp(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qsimp
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp, UNLIKELY=-1.0e30_sp
    Returns the integral of the function func from a to b. The parameter EPS should be set to
    the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
    allowed number of steps. Integration is performed by Simpson's rule.
INTEGER(I4B) :: j
REAL(SP) :: os,ost,st
ost=UNLIKELY
os= UNLIKELY
do j=1,JMAX
    call trapzd(func,a,b,st,j)
    qsimp=(4.0_sp*st-ost)/3.0_sp       Compare equation (4.2.4).
    if (j > 5) then                    Avoid spurious early convergence.
        if (abs(qsimp-os) < EPS*abs(os) .or. &
            (qsimp == 0.0 .and. os == 0.0)) RETURN
    end if
    os=qsimp
```

```
    ost=st
end do
call nrerror('qsimp: too many steps')
END FUNCTION qsimp
```

* * *

```
FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint,trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Returns the integral of the function func from a to b. Integration is performed by Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.

Parameters: EPS is the fractional accuracy desired, as determined by the extrapolation error estimate; JMAX limits the total number of steps; K is the number of points used in the extrapolation.

```
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromb
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call trapzd(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
        if (abs(dqromb) <= EPS*abs(qromb)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=0.25_sp*h(j)
end do
call nrerror('qromb: too many steps')
END FUNCTION qromb
```

These store the successive trapezoidal approximations and their relative stepsizes.

This is a key step: The factor is 0.25 even though the stepsize is decreased by only 0.5. This makes the extrapolation a polynomial in h^2 as allowed by equation (4.2.1), not just a polynomial in h.

* * *

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

This routine computes the nth stage of refinement of an extended midpoint rule. func is input as the name of the function to be integrated between limits a and b, also input. When

called with n=1, the routine returns as s the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding $(2/3) \times 3^{n-1}$ additional interior points. s should not be modified between sequential calls.

```
REAL(SP) :: del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
if (n == 1) then
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)              The added points alternate in spacing between
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)      del and 2*del.
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))        The new sum is combined with the old integral
end if                                     to give a refined integral.
END SUBROUTINE midpnt
```

f90 midpnt is a subroutine and not a function for the same reasons as trapzd. This is also true for the other mid... routines below.

```
s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
```
Here we use (/.../) to pass a single scalar argument to a vector function.

⋆ ⋆ ⋆

```
FUNCTION qromo(func,a,b,choose)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromo
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func

    SUBROUTINE choose(funk,aa,bb,s,n)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE choose
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=14,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6
```

Romberg integration on an open interval. Returns the integral of the function func from a to b, using any specified integrating subroutine choose and Romberg's method. Normally choose will be an open formula, not evaluating the function at the endpoints. It is assumed that choose triples the number of steps on each call, and that its error series contains only

even powers of the number of steps. The routines midpnt, midinf, midsql, midsqu, and midexp are possible choices for choose. The parameters have the same meaning as in qromb.

```
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromo
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call choose(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromo,dqromo)
        if (abs(dqromo) <= EPS*abs(qromo)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=h(j)/9.0_sp          This is where the assumption of step tripling and an even
end do                              error series is used.
call nrerror('qromo: too many steps')
END FUNCTION qromo
```

⋆ ⋆ ⋆

```
SUBROUTINE midinf(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that the function is evaluated at evenly spaced points in $1/x$ rather than in x. This allows the upper limit bb to be as large and positive as the computer allows, or the lower limit aa to be as large and negative, but not both. aa and bb must have the same sign.

```
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
call assert(aa*bb > 0.0, 'midinf args')
b=1.0_sp/aa                 These two statements change the limits of integration ac-
a=1.0_sp/bb                     cordingly.
if (n == 1) then            From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)        This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=funk(1.0_sp/x)/x**2
    END FUNCTION func
END SUBROUTINE midinf
```

f90 FUNCTION func(x) The change of variable could have been effected by a statement function in midinf itself. However, the statement function is a Fortran 77 feature that is deprecated in Fortran 90 because it does not allow the benefits of having an explicit interface, i.e., a complete set of specification statements. Statement functions can always be coded as internal subprograms instead.

```
SUBROUTINE midsql(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that it allows for an inverse square-root singularity in the integrand at the lower limit aa.

```
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)                  These two statements change the limits of integration ac-
a=0.0                              cordingly.
if (n == 1) then               From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)           This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=2.0_sp*x*funk(aa+x**2)
    END FUNCTION func
END SUBROUTINE midsql
```

```
SUBROUTINE midsqu(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that it allows for an inverse square-root singularity in the integrand at the upper limit bb.

```
REAL(SP) :: a,b,del
```

```
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)                These two statements change the limits of integration ac-
a=0.0                            cordingly.
if (n == 1) then             From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)         This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=2.0_sp*x*funk(bb-x**2)
    END FUNCTION func
END SUBROUTINE midsqu


SUBROUTINE midexp(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
    This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement
    of the integral of funk from aa to bb, except that bb is assumed to be infinite (value passed
    not actually used). It is assumed that the function funk decreases exponentially rapidly at
    infinity.
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=exp(-aa)                   These two statements change the limits of integration ac-
a=0.0                            cordingly.
if (n == 1) then             From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)         This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=funk(-log(x))/x
    END FUNCTION func
END SUBROUTINE midexp
```

⋆ ⋆ ⋆

```
SUBROUTINE gauleg(x1,x2,x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp
```

Given the lower and upper limits of integration x1 and x2, this routine returns arrays x and w of length N containing the abscissas and weights of the Gauss-Legendre N-point quadrature formula. The parameter EPS is the relative precision. Note that internal computations are done in double precision.

```
INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: xl,xm
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauleg')
m=(n+1)/2                                   The roots are symmetric in the interval,
xm=0.5_dp*(x2+x1)                              so we only have to find half of them.
xl=0.5_dp*(x2-x1)
z=cos(PI_D*(arth(1,1,m)-0.25_dp)/(n+0.5_dp))   Initial approximations to the roots.
unfinished=.true.
do its=1,MAXIT                              Newton's method carried out simultane-
    where (unfinished)                         ously on the roots.
        p1=1.0
        p2=0.0
    end where
    do j=1,n                                Loop up the recurrence relation to get
        where (unfinished)                     the Legendre polynomials evaluated
            p3=p2                              at z.
            p2=p1
            p1=((2.0_dp*j-1.0_dp)*z*p2-(j-1.0_dp)*p3)/j
        end where
    end do
      p1 now contains the desired Legendre polynomials. We next compute pp, the derivatives,
      by a standard relation involving also p2, the polynomials of one lower order.
    where (unfinished)
        pp=n*(z*p1-p2)/(z*z-1.0_dp)
        z1=z
        z=z1-p1/pp                          Newton's method.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gauleg')
x(1:m)=xm-xl*z                              Scale the root to the desired interval,
x(n:n-m+1:-1)=xm+xl*z                          and put in its symmetric counterpart.
w(1:m)=2.0_dp*xl/((1.0_dp-z**2)*pp**2)      Compute the weight
w(n:n-m+1:-1)=w(1:m)                           and its symmetric counterpart.
END SUBROUTINE gauleg
```

f90 Often we have an iterative procedure that has to be applied until all components of a vector have satisfied a convergence criterion. Some components of the vector might converge sooner than others, and it is inefficient on a small-scale parallel (SSP) machine to continue iterating on those components. The general structure we use for such an iteration is exemplified by the following lines from gauleg:

```
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
   ...
unfinished=.true.
do its=1,MAXIT
```

```
        where (unfinished)
              ...
           unfinished=(abs(z-z1) > EPS)
        end where
        if (.not. any(unfinished)) exit
    end do
    if (its == MAXIT+1) call nrerror('too many iterations in gauleg')
```

We use the logical mask `unfinished` to control which vector components are processed inside the `where`. The mask gets updated on each iteration by testing whether any further vector components have converged. When all have converged, we exit the iteration loop. Finally, we check the value of `its` to see whether the maximum allowed number of iterations was exceeded before all components converged.

The logical expression controlling the `where` block (in this case `unfinished`) gets evaluated completely on entry into the `where`, and it is then perfectly fine to modify it inside the block. The modification affects only the *next* execution of the `where`.

On a strictly *serial* machine, there is of course some penalty associated with the above scheme: after a vector component converges, its corresponding component in `unfinished` is redundantly tested on each further iteration, until the slowest-converging component is done. If the number of iterations required does not vary too greatly from component to component, this is a minor, often negligible, penalty. However, one should be on the alert against algorithms whose worst-case convergence could differ from typical convergence by orders of magnitude. For these, one would need to implement a more complicated packing-unpacking scheme. (See discussion in Chapter B6, especially introduction, p. 1083, and notes for `factrl`, p. 1087.)

```
SUBROUTINE gaulag(x,w,alf)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp
```

Given `alf`, the parameter α of the Laguerre polynomials, this routine returns arrays `x` and `w` of length N containing the abscissas and weights of the N-point Gauss-Laguerre quadrature formula. The abscissas are returned in ascending order. The parameter EPS is the relative precision. Note that internal computations are done in double precision.

```
INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP) :: anu
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
    C3=2.579930e-03_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION(size(x)) :: rhs,r2,r3,theta
REAL(DP), DIMENSION(size(x)) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished
n=assert_eq(size(x),size(w),'gaulag')
anu=4.0_sp*n+2.0_sp*alf+2.0_sp
rhs=arth(4*n-1,-4,n)*PI/anu
r3=rhs**(1.0_sp/3.0_sp)
r2=r3**2
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))
z=anu*cos(theta)**2
unfinished=.true.
do its=1,MAXIT
    where (unfinished)
```

Initial approximations to the roots go into `z`.

Newton's method carried out simultaneously on the roots.

```
        p1=1.0
        p2=0.0
    end where
    do j=1,n                              Loop up the recurrence relation to get the La-
        where (unfinished)                    guerre polynomials evaluated at z.
            p3=p2
            p2=p1
            p1=((2.0_dp*j-1.0_dp+alf-z)*p2-(j-1.0_dp+alf)*p3)/j
        end where
    end do
      p1 now contains the desired Laguerre polynomials. We next compute pp, the derivatives,
      by a standard relation involving also p2, the polynomials of one lower order.
    where (unfinished)
        pp=(n*p1-(n+alf)*p2)/z
        z1=z
        z=z1-p1/pp                        Newton's formula.
        unfinished=(abs(z-z1) > EPS*z)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gaulag')
x=z                                       Store the root and the weight.
w=-exp(gammln(alf+n)-gammln(real(n,sp)))/(pp*n*p2)
END SUBROUTINE gaulag
```

The key difficulty in parallelizing this routine starting from the Fortran 77 version is that the initial guesses for the roots of the Laguerre polynomials were given in terms of previously determined roots. This prevents one from finding all the roots simultaneously. The solution is to come up with a new approximation to the roots that is a simple explicit formula, like the formula we used for the Legendre roots in `gauleg`.

We start with the approximation to $L_n^\alpha(x)$ given in equation (10.15.8) of [1]. We keep only the first term and ask when it is zero. This gives the following prescription for the kth root x_k of $L_n^\alpha(x)$: Solve for θ the equation

$$2\theta - \sin 2\theta = \frac{4n - 4k + 3}{4n + 2\alpha + 2}\pi \tag{B4.1}$$

Since $1 \le k \le n$ and $\alpha > -1$, we can always find a value such that $0 < \theta < \pi/2$. Then the approximation to the root is

$$x_k = (4n + 2\alpha + 2)\cos^2\theta \tag{B4.2}$$

This typically gives 3-digit accuracy, more than enough for the Newton iteration to be able to refine the root. Unfortunately equation (B4.1) is not an explicit formula for θ. (You may recognize it as being of the same form as Kepler's equation in mechanics.) If we call the right-hand side of (B4.1) y, then we can get an explicit formula by working out the power series for $y^{1/3}$ near $\theta = 0$ (using a computer algebra program). Next invert the series to give θ as a function of $y^{1/3}$. Finally, economize the series (see §5.11). The result is the concise approximation

$$\begin{aligned}\theta = {} & 0.9084064y^{1/3} + 5.214976 \times 10^{-2}y + 2.579930 \times 10^{-3}y^{5/3} \\ & + 3.986126 \times 10^{-3}y^{7/3}\end{aligned} \tag{B4.3}$$

```
SUBROUTINE gauher(x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp,PIM4=0.7511255444649425_dp
```

This routine returns arrays x and w of length N containing the abscissas and weights of the N-point Gauss-Hermite quadrature formula. The abscissas are returned in descending order. Note that internal computations are done in double precision.

Parameters: EPS is the relative precision, PIM4 $= 1/\pi^{1/4}$.

```
INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP) :: anu
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
    C3=2.579930e-03_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION((size(x)+1)/2) :: rhs,r2,r3,theta
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauher')
m=(n+1)/2                          The roots are symmetric about the origin, so we have to
anu=2.0_sp*n+1.0_sp                    find only half of them.
rhs=arth(3,4,m)*PI/anu
r3=rhs**(1.0_sp/3.0_sp)
r2=r3**2
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))
z=sqrt(anu)*cos(theta)             Initial approximations to the roots.
unfinished=.true.
do its=1,MAXIT                     Newton's method carried out simultaneously on the roots.
    where (unfinished)
        p1=PIM4
        p2=0.0
    end where
    do j=1,n                       Loop up the recurrence relation to get the Hermite poly-
        where (unfinished)             nomials evaluated at z.
            p3=p2
            p2=p1
            p1=z*sqrt(2.0_dp/j)*p2-sqrt(real(j-1,dp)/real(j,dp))*p3
        end where
    end do
      p1 now contains the desired Hermite polynomials. We next compute pp, the derivatives,
      by the relation (4.5.21) using p2, the polynomials of one lower order.
    where (unfinished)
        pp=sqrt(2.0_dp*n)*p2
        z1=z
        z=z1-p1/pp                 Newton's formula.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gauher')
x(1:m)=z                           Store the root
x(n:n-m+1:-1)=-z                   and its symmetric counterpart.
w(1:m)=2.0_dp/pp**2                Compute the weight
w(n:n-m+1:-1)=w(1:m)               and its symmetric counterpart.
END SUBROUTINE gauher
```

Once again we need an explicit approximation for the polynomial roots, this time for $H_n(x)$. We can use the same approximation scheme as for $L_n^\alpha(x)$, since

$$H_{2m}(x) \propto L_m^{-1/2}(x^2), \qquad H_{2m+1}(x) \propto x L_m^{1/2}(x^2) \tag{B4.4}$$

Equations (B4.1) and (B4.2) become

$$2\theta - \sin 2\theta = \frac{4k-1}{2n+1}\pi$$
$$x_k = \sqrt{2n+1}\cos\theta \tag{B4.5}$$

Here $k = 1, 2, \ldots, m$ where $m = [(n+1)/2]$, and $k = 1$ is the largest root. The negative roots follow from symmetry. The root at $x = 0$ for odd n is included in this approximation.

```
SUBROUTINE gaujac(x,w,alf,bet)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf,bet
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp
    Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns
    arrays x and w of length N containing the abscissas and weights of the N-point Gauss-
    Jacobi quadrature formula. The abscissas are returned in descending order. The parameter
    EPS is the relative precision. Note that internal computations are done in double precision.
INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: alfbet,a,c,temp
REAL(DP), DIMENSION(size(x)) :: b,p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished
n=assert_eq(size(x),size(w),'gaujac')
alfbet=alf+bet                  Initial approximations to the roots go into z.
z=cos(PI*(arth(1,1,n)-0.25_dp+0.5_dp*alf)/(n+0.5_dp*(alfbet+1.0_dp)))
unfinished=.true.
do its=1,MAXIT                  Newton's method carried out simultaneously on the roots.
    temp=2.0_dp+alfbet
    where (unfinished)          Start the recurrence with P_0 and P_1 to avoid a division
        p1=(alf-bet+temp*z)/2.0_dp      by zero when α + β = 0 or −1.
        p2=1.0
    end where
    do j=2,n                    Loop up the recurrence relation to get the Jacobi poly-
        a=2*j*(j+alfbet)*temp       nomials evaluated at z.
        temp=temp+2.0_dp
        c=2.0_dp*(j-1.0_dp+alf)*(j-1.0_dp+bet)*temp
        where (unfinished)
            p3=p2
            p2=p1
            b=(temp-1.0_dp)*(alf*alf-bet*bet+temp*&
                (temp-2.0_dp)*z)
            p1=(b*p2-c*p3)/a
        end where
    end do
      p1 now contains the desired Jacobi polynomials. We next compute pp, the derivatives,
      by a standard relation involving also p2, the polynomials of one lower order.
    where (unfinished)
        pp=(n*(alf-bet-temp*z)*p1+2.0_dp*(n+alf)*&
            (n+bet)*p2)/(temp*(1.0_dp-z*z))
        z1=z
        z=z1-p1/pp              Newton's formula.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gaujac')
x=z                             Store the root and the weight.
```

```
w=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0_sp)-&
    gammln(n+alf+bet+1.0_sp))*temp*2.0_sp**alfbet/(pp*p2)
END SUBROUTINE gaujac
```

Now we need an explicit approximation for the roots of the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$. We start with the asymptotic expansion (10.14.10) of [1]. Setting this to zero gives the formula

$$x = \cos\left[\frac{k - 1/4 + \alpha/2}{n + (\alpha + \beta + 1)/2}\pi\right] \tag{B4.6}$$

This is better than the formula (22.16.1) in [2], especially at small and moderate n.

⋆ ⋆ ⋆

```
SUBROUTINE gaucof(a,b,amu0,x,w)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : eigsrt,tqli
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), INTENT(IN) :: amu0
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
```

Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix. On input, a and b of length N are the coefficients of the recurrence relation for the set of monic orthogonal polynomials. The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as amu0. The abscissas are returned in descending order in array x of length N, with the corresponding weights in w, also of length N. The arrays a and b are modified. Execution can be speeded up by modifying tqli and eigsrt to compute only the first component of each eigenvector.

```
REAL(SP), DIMENSION(size(a),size(a)) :: z
INTEGER(I4B) :: n
n=assert_eq(size(a),size(b),size(x),size(w),'gaucof')
b(2:n)=sqrt(b(2:n))          Set up superdiagonal of Jacobi matrix.
call unit_matrix(z)          Set up identity matrix for tqli to compute eigenvectors.
call tqli(a,b,z)
call eigsrt(a,z)             Sort eigenvalues into descending order.
x=a
w=amu0*z(1,:)**2             Equation (4.5.12).
END SUBROUTINE gaucof
```

⋆ ⋆ ⋆

```
SUBROUTINE orthog(anu,alpha,beta,a,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: anu,alpha,beta
REAL(SP), DIMENSION(:), INTENT(OUT) :: a,b
```

Computes the coefficients a_j and b_j, $j = 0, \ldots N-1$, of the recurrence relation for monic orthogonal polynomials with weight function $W(x)$ by Wheeler's algorithm. On input, alpha and beta contain the $2N-1$ coefficients α_j and β_j, $j = 0, \ldots 2N-2$, of the recurrence

```
    relation for the chosen basis of orthogonal polynomials. The 2N modified moments ν_j are
    input in anu for j = 0,...2N − 1. The first N coefficients are returned in a and b.
INTEGER(I4B) :: k,n,ndum
REAL(SP), DIMENSION(2*size(a)+1,2*size(a)+1) :: sig
n=assert_eq(size(a),size(b),'orthog: n')
ndum=assert_eq(2*n,size(alpha)+1,size(anu),size(beta)+1,'orthog: ndum')
sig(1,3:2*n)=0.0                Initialization, Equation (4.5.33).
sig(2,2:2*n+1)=anu(1:2*n)
a(1)=alpha(1)+anu(2)/anu(1)
b(1)=0.0
do k=3,n+1                      Equation (4.5.34).
    sig(k,k:2*n-k+3)=sig(k-1,k+1:2*n-k+4)+(alpha(k-1:2*n-k+2) &
        -a(k-2))*sig(k-1,k:2*n-k+3)-b(k-2)*sig(k-2,k:2*n-k+3) &
        +beta(k-1:2*n-k+2)*sig(k-1,k-1:2*n-k+2)
    a(k-1)=alpha(k-1)+sig(k,k+1)/sig(k,k)-sig(k-1,k)/sig(k-1,k-1)
    b(k-1)=sig(k,k)/sig(k-1,k-1)
end do
END SUBROUTINE orthog
```

⋆ ⋆ ⋆

f90 As discussed in Volume 1, multidimensional quadrature can be performed by calling a one-dimensional quadrature routine along each dimension. If the same routine is used for all such calls, then the calls are recursive. The file quad3d.f90 contains two modules, quad3d_qgaus_mod and quad3d_qromb_mod. In the first, the basic one-dimensional quadrature routine is a 10-point Gaussian quadrature routine called qgaus and three-dimensional quadrature is performed by calling quad3d_qgaus. In the second, the basic one-dimensional routine is qromb of §4.3 and the three-dimensional routine is quad3d_qromb. The Gaussian quadrature is simpler but its accuracy is not controllable. The Romberg integration lets you specify an accuracy, but is apt to be very slow if you try for too much accuracy. The only difference between the stand-alone version of trapzd and the version included here is that we have to add the keyword RECURSIVE. The only changes from the stand-alone version of qromb are: We have to add RECURSIVE; we remove trapzd from the list of routines in USE nr; we increase EPS to 3×10^{-6}. Even this value could be too ambitious for difficult functions. You may want to set JMAX to a smaller value than 20 to avoid burning up a lot of computer time. Some people advocate using a smaller EPS on the inner quadrature (over z in our routine) than on the outer quadratures (over x or y). That strategy would require separate copies of qromb.

```
MODULE quad3d_qgaus_mod
USE nrtype
PRIVATE                         Hide all names from the outside,
PUBLIC quad3d_qgaus             except quad3d itself.
REAL(SP) :: xsav,ysav
INTERFACE                       User-supplied functions.
    FUNCTION func(x,y,z)        The three-dimensional function to be integrated.
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: z
    REAL(SP), DIMENSION(size(z)) :: func
    END FUNCTION func

    FUNCTION y1(x)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y1
    END FUNCTION y1
    FUNCTION y2(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y2
    END FUNCTION y2
    FUNCTION z1(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z1
    END FUNCTION z1
    FUNCTION z2(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z2
    END FUNCTION z2
END INTERFACE
```

The routine quad3d_qgaus returns as ss the integral of a user-supplied function func over a three-dimensional region specified by the limits x1, x2, and by the user-supplied functions y1, y2, z1, and z2, as defined in (4.6.2). Integration is performed by calling qgaus recursively.

```
CONTAINS
FUNCTION h(x)                    This is H of eq. (4.6.5).
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
do i=1,size(x)
    xsav=x(i)
    h(i)=qgaus(g,y1(xsav),y2(xsav))
end do
END FUNCTION h
FUNCTION g(y)                    This is G of eq. (4.6.4).
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
do j=1,size(y)
    ysav=y(j)
    g(j)=qgaus(f,z1(xsav,ysav),z2(xsav,ysav))
end do
END FUNCTION g
FUNCTION f(z)                    The integrand f(x, y, z) evaluated at fixed x and y.
REAL(SP), DIMENSION(:), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f
RECURSIVE FUNCTION qgaus(func,a,b)
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qgaus
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
REAL(SP) :: xm,xr
REAL(SP), DIMENSION(5) :: dx, w = (/ 0.2955242247_sp,0.2692667193_sp,&
    0.2190863625_sp,0.1494513491_sp,0.0666713443_sp /),&
    x = (/ 0.1488743389_sp,0.4333953941_sp,0.6794095682_sp,&
```

```
    0.8650633666_sp,0.9739065285_sp /)
xm=0.5_sp*(b+a)
xr=0.5_sp*(b-a)
dx(:)=xr*x(:)
qgaus=xr*sum(w(:)*(func(xm+dx)+func(xm-dx)))
END FUNCTION qgaus

SUBROUTINE quad3d_qgaus(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qgaus(h,x1,x2)
END SUBROUTINE quad3d_qgaus
END MODULE quad3d_qgaus_mod
```

f90 PRIVATE...PUBLIC quad3d_qgaus By default, all module entities are accessible by a routine that uses the module (unless we restrict the USE statement with ONLY). In this module, the user needs access only to the routine quad3d_qgaus; the variables xsav, ysav and the procedures f, g, h, and qgaus are purely internal. It is good programming practice to prevent duplicate name conflicts or data overwriting by limiting access to only the desired entities. Here the PRIVATE statement with no variable names resets the default from PUBLIC. Then we include in the PUBLIC statement only the function name we want to be accessible.

REAL(SP) :: xsav,ysav In Fortran 90, we generally avoid declaring global variables in COMMON blocks. Instead, we give them complete specifications in a module. A deficiency of Fortran 90 is that it does not allow pointers to functions. So here we have to use the fixed-name function func for the function to be integrated over. If we could have a pointer to a function as a global variable, then we would just set the pointer to point to the user function (of any name) in the calling program. Similarly the functions y1, y2, z1, and z2 could also have any name.

CONTAINS Here follow the internal subprograms f, g, h, qgaus, and quad3d_qgaus. Note that such internal subprograms are all "visible" to each other, i.e., their interfaces are mutually explicit, and do not require INTERFACE statements.

RECURSIVE SUBROUTINE qgaus(func,a,b,ss) The RECURSIVE keyword is required for the compiler to process correctly any procedure that is invoked again in its body before the return from the first call has been completed. While some compilers may let you get away without explicitly informing them that a routine is recursive, don't count on it!

```
MODULE quad3d_qromb_mod
    Alternative to quad3d_qgaus_mod that uses qromb to perform each one-dimensional in-
    tegration.
USE nrtype
PRIVATE
PUBLIC quad3d_qromb
REAL(SP) :: xsav,ysav
INTERFACE
    FUNCTION func(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: z
    REAL(SP), DIMENSION(size(z)) :: func
    END FUNCTION func

    FUNCTION y1(x)
```

```
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y1
    END FUNCTION y1
    FUNCTION y2(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y2
    END FUNCTION y2
    FUNCTION z1(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z1
    END FUNCTION z1
    FUNCTION z2(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z2
    END FUNCTION z2
END INTERFACE
CONTAINS
FUNCTION h(x)
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
do i=1,size(x)
    xsav=x(i)
    h(i)=qromb(g,y1(xsav),y2(xsav))
end do
END FUNCTION h
FUNCTION g(y)
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
do j=1,size(y)
    ysav=y(j)
    g(j)=qromb(f,z1(xsav,ysav),z2(xsav,ysav))
end do
END FUNCTION g
FUNCTION f(z)
REAL(SP), DIMENSION(:), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f
RECURSIVE FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=3.0e-6_sp
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromb
```

```
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call trapzd(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
        if (abs(dqromb) <= EPS*abs(qromb)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=0.25_sp*h(j)
end do
call nrerror('qromb: too many steps')
END FUNCTION qromb

RECURSIVE SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
    s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
    it=2**(n-2)
    del=(b-a)/it
    fsum=sum(func(arth(a+0.5_sp*del,del,it)))
    s=0.5_sp*(s+del*fsum)
end if
END SUBROUTINE trapzd

SUBROUTINE quad3d_qromb(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qromb(h,x1,x2)
END SUBROUTINE quad3d_qromb
END MODULE quad3d_qromb_mod
```

`MODULE quad3d_qromb_mod` The only difference between this module and the previous one is that all calls to `qgaus` are replaced by calls to `qromb` and that the routine `qgaus` is replaced by `qromb` and `trapzd`.

CITED REFERENCES AND FURTHER READING:

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Volume II (New York: McGraw-Hill). [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [2]