```
!  Version 1.4

!  This file collects all the various FM routines from the "More sample programs" page on the
!  FM web site.  See that page for the programs that call these routines and illustrate their
!  use.  Here is a list of the user-callable routines in this file (see the documentation at
!  the top of each routine for explanation of each of the arguments to the routines).

!  1.  Find a minimum or maximum function value of a real function of one variable.

!                   subroutine fm_find_min(min_or_max, ax, bx, tol, xval, fval, f, nf, kprt, ku)

!  2.  n-th derivative of a real function of one variable.

!                   function fm_fprime(n, a, f, nf)

!  3.  n-th derivative of a complex function of one variable.

!                   function zm_fprime(n, a, f, nf)

!  4.  Definite integral for a real function of one variable.

!                   subroutine fm_integrate(f, n, a, b, tol, result, kprt, nw)

!  5.  Inverse matrix for a real n x n matrix.

!                   subroutine fm_inverse(a, n, b, det)

!  6.  Inverse matrix for a complex n x n matrix.

!                   subroutine zm_inverse(a, n, b, det)

!  7.  Generate the real linear system of normal equations for a least square fit.

!                   subroutine fm_geneq(f, a, b, k, x, y, n)

!  8.  Solve a real n x n linear system of equations.

!                   subroutine fm_lin_solve(a, x, b, n, det)

!  9.  Solve a complex n x n linear system of equations.

!                   subroutine zm_lin_solve(a, x, b, n, det)

! 10.  Solve a real differential equation (initial value problem).

!                   subroutine fm_rk14(a, b, n_order, n_function, s, tol, s1)

! 11.  Find a root of a real function of one variable.

!                   subroutine fm_secant(ax, bx, f, nf, root, kprt, ku)

! 12.  Find a root of a complex function of one variable.

!                   subroutine zm_secant(ax, bx, f, nf, root, kprt, ku)

! 13.  Find nr roots of a complex function of one variable.
```

```
!                  subroutine zm_roots(nr, f, nf, n_found, list_of_roots, kprt, ku)



      subroutine fm_find_min(min_or_max, ax, bx, tol, xval, fval, f, nf, kprt, ku)

!  min_or_max having value 1 means minimize the function, otherwise maximize.
!  ax, bx     define the endpoints of an interval in which the search takes place.
!             Note that the extreme point returned can be an endpoint, ax or bx.
!             For example, to find a relative minimum inside the interval, make sure the
!             function values at ax and bx are not smaller than the relative minimum.
!  tol        is the tolerance for the minimum.  Usually tol should be no less than epsilon(ax),
!             meaning the x-coordinate xval of the extreme point will be accurate to about all
!             the digits carried.  The y-coordinate fval should also be accurate to nearly
!             full precision.
!             The typical graph is nearly parabolic near the minimum, and within sqrt(epsilon(ax))
!             of the minimum all the function values are essentially identical at the user's
!             level of precision.
!             This routine raises precision above the user's level in order to deal with the
!             cancellation error caused by a parabolic-type extreme point (one for which the
!             derivative of f has a simple root).
!             For cases where derivative of f has a multiple root, the cancellation is more
!             severe, and the location of the max or min may not achieve full precision.
!  xval       is returned as the value of x that minimizes (or maximizes) function f(x,nf).
!             It is a relative extreme point, and may not be the global extreme point if the
!             function has more than one extremum on the interval.
!  fval       is returned as the function value at xval.
!  f(x,nf)    is the function to be minimized.  x is the argument and nf is the function
!             number, in case several functions are defined within f.
!  kprt       controls printing within the routine:
!             kprt = 0 for no output
!             kprt = 1 for the approximation to the root and the function
!                      value to be printed once at the end of the routine.
!             kprt = 2 for the approximation to the root and the function
!                      value to be printed each iteration.
!  ku         is the unit number for output if kprt > 0.

!  The method used is a combination of golden section search and successive parabolic interpolation.
!  Convergence is never much slower than that for a fibonacci search.  If f has a continuous second
!  derivative which is positive at the minimum (which is not at ax or bx), then convergence is
!  superlinear, and usually of the order of about 1.324....

!  This routine is a slightly modified translation of function fval from netlib, which was adapted
!  from the algol 60 procedure localmin given by Richard Brent in Algorithms For Minimization
!  Without Derivatives, Prentice-Hall (1973).

      use fmvals
      use fmzm
      implicit none
      integer :: min_or_max, nf, kprt, ku
      type (fm) :: ax, bx, tol, xval, fval
      type (fm), external :: f
      intent (in) :: min_or_max, ax, bx, tol, nf, kprt, ku
      intent (inout) :: xval, fval

      character(80) :: st1, st2
      character(10) str_format
      integer :: j, kl, minv, ndsave
```

```
      type (fm), save :: a, ax2, b, bx2, c, d, e, eps, xm, p, q, r, t2, u, v, w, fu, fv, fw, fx,  &
                         x, tol1, tol2, tol3

!             Raise precision.

      ndsave = ndig
      ndig = 3*ndig
      call fm_equ(ax, ax2, ndsave, ndig)
      call fm_equ(bx, bx2, ndsave, ndig)
      call fm_equ(tol, tol2, ndsave, ndig)

      minv = 1
      if (min_or_max /= 1) minv = -1

!             str_format is the format used for tracing output if kprt > 0.

      j = min(log10(dble(mbase))*(ndsave-1)+1, dble(50))
      write (str_format, "(a, i2, a, i2)") 'es', j+15, '.', j

!             c is the squared inverse of the golden ratio.

      c = (3-sqrt(to_fm('5.0d0')))/2

!             eps is approximately the square root of the relative machine precision.

      eps = epsilon(ax2)
      tol1 = eps + 1
      eps = sqrt(eps)

      a = min(ax2, bx2)
      b = max(ax2, bx2)
      v = a + c*(b-a)
      w = v
      x = v
      e = 0
      fx = f(x, nf)*minv
      fv = fx
      fw = fx
      tol3 = tol2/3
      j = 1

      if (kprt == 2) then
          write (ku,*) ' '
          if (min_or_max == 1) then
              write (ku,*) ' fm_find_min.  Begin trace of all iterations.'
              write (ku,*) '               Search for a relative minimum on the interval'
              write (ku, "(13x, es20.10, '    to ', es20.10/)") to_dp(ax2), to_dp(bx2)
          else
              write (ku,*) ' fm_find_min.  Begin trace of all iterations.'
              write (ku,*) '               Search for a relative maximum on the interval'
              write (ku, "(13x, es20.10, '    to ', es20.10/)") to_dp(ax2), to_dp(bx2)
          endif
          st1 = fm_format(str_format, x)
          st2 = fm_format(str_format, fx*minv)
          write (ku, "('      j =', i3, 4x, '  x  = ', a)") j, trim(st1)
          write (ku, "('         ', 3x, 4x, 'f(x) = ', a/)") trim(st2)
      endif
```

```
!            The main loop starts here.

      kl = 1
      do while (kl == 1)
         kl = 0
         xm = (a+b)/2
         tol1 = eps*abs(x) + tol3
         t2 = 2*tol1

!            Check the stopping criterion.

         if (abs(x-xm) <= (t2-(b-a)/2)) exit
         p = 0
         q = 0
         r = 0
         if (abs(e) > tol1) then

!                Fit a parabola.

            r = (x-w)*(fx-fv)
            q = (x-v)*(fx-fw)
            p = (x-v)*q-(x-w)*r
            q = 2*(q-r)
            if (q > 0) then
                p = -p
            else
                q = -q
            endif
            r = e
            e = d
         endif

         if ((abs(p) >= abs(q*r/2)) .or. (p <= q*(a-x)) .or. (p >= q*(b-x))) then

!                Make a golden-section step.

            if (x < xm) then
                e = b - x
            else
                e = a - x
            endif
            d = c*e
         else

!                Make a parabolic-interpolation step.

            d = p/q
            u = x + d

!                f must not be evaluated too close to ax or bx.

            if (.not. (u-a >= t2 .and. b-u >= t2)) then
                d = tol1
                if (x >= xm) d = -d
            endif
         endif

!             f must not be evaluated too close to x.
```

```
        if (abs(d) >= tol1) then
            u = x + d
        else
            if (d > 0) then
                u = x + tol1
            else
                u = x - tol1
            endif
        endif
        fu = f(u, nf)*minv

        j = j + 1
        if (kprt == 2) then
            st1 = fm_format(str_format, u)
            st2 = fm_format(str_format, fu*minv)
            write (ku, "('      j =', i3, 4x, '  x  = ', a)") j, trim(st1)
            write (ku, "('         ', 3x, 4x, 'f(x) = ', a/)") trim(st2)
        endif

!            update  a, b, v, w, and x.

        if (fx <= fu) then
            if (u < x) then
                a = u
            else
                b = u
            endif
        endif
        if (fu <= fx) then
            if (u < x) then
                b = x
            else
                a = x
            endif
            v = w
            fv = fw
            w = x
            fw = fx
            x = u
            fx = fu
            kl = 1
            cycle
        endif

        if (.not. (fu > fw .and. w /= x)) then
            v = w
            fv = fw
            w = u
            fw = fu
            kl = 1
            cycle
        endif

        if ((fu > fv) .and. (v /= x) .and. (v /= w)) then
            kl = 1
            cycle
        endif
```

```
         v = u
         fv = fu
         kl = 1
      enddo

!             end of main loop.  Round the results back to the user's precision.

      call fm_equ(x, xval, ndig, ndsave)
      call fm_equ(fx*minv, fval, ndig, ndsave)
      ndig = ndsave

      if (kprt >= 1) then
          if (kprt == 1) write (ku,*) ' '
          if (min_or_max == 1) then
              write (ku, "('  fm_find_min.   Function ', i3, i6, ' iterations.  A relative " //  &
                         "minimum on interval'/13x, es20.10, '    to ', es20.10, '  is')")       &
                     nf, j, to_dp(ax), to_dp(bx)
              st1 = fm_format(str_format, xval)
              st2 = fm_format(str_format, fval)
              write (ku, "(15x, '   x  = ', a)") trim(st1)
              write (ku, "(15x, ' f(x) = ', a)") trim(st2)
          else
              write (ku, "('  fm_find_min.   Function ', i3, i6, ' iterations.  A relative " //  &
                         "maximum on interval'/13x, es20.10, '    to ', es20.10, '  is')")       &
                     nf, j, to_dp(ax), to_dp(bx)
              st1 = fm_format(str_format, xval)
              st2 = fm_format(str_format, fval)
              write (ku, "(15x, '   x  = ', a)") trim(st1)
              write (ku, "(15x, ' f(x) = ', a)") trim(st2)
          endif
          write (ku,*) ' '
      endif

      return
      end subroutine fm_find_min

      function fm_fprime(n, a, f, nf)     result (return_value)
      use fmvals
      use fmzm
      implicit none

!  This routine finds the n-th derivative of f(x,nf), evaluated at a.
!  nf is passed on to function f to indicate which function to use in cases where several
!  different functions may be defined there.

!  f must be defined in an interval containing a, so that f can be sampled on both sides of a.

!  n may be zero, so that in cases where f suffers cancellation error at a, an accurate
!  function value is returned.

!  fm_fprime tries to return full accuracy for the derivative, by raising precision above
!  the user's level and using difference formulas.

      type (fm) :: a, return_value
      integer :: n, nf
      type (fm), external :: f
      intent (in) :: n, a, nf
```

```
      integer :: j, k, kwarn_save, ndsave
      type (fm), save :: d1, d2, f1, f2, fmh, fph, h, tol, tol2, x1, xmh, xph

!             Raise precision slightly.

      ndsave = ndig
      ndig = ndig + ngrd52
      call fm_equ(a, x1, ndsave, ndig)
      kwarn_save = kwarn
      kwarn = 0

      d2 = 0
      f1 = f(x1, nf)
      if (f1 /= 0) then
          call fm_ulp(f1, tol)
      else
          tol = epsilon(to_fm(1))
      endif
      tol = abs(tol)

!             Check for a legal function value.

      if (is_unknown(f1) .or. is_overflow(f1) .or. is_underflow(f1) .or. n < 0) then
          d2 = to_fm(' unknown ')
          call fm_equ(d2, return_value, ndig, ndsave)
          ndig = ndsave
          kwarn = kwarn_save
          return
      endif
      f2 = f1

!             Loop at increasing precision until the difference formula is accurate.

      do j = 1, 100

         ndig = 2*ndig

!             Define the variables used below at the new higher precision.

         call fm_equ(d2, d1, ndig/2, ndig)
         call fm_equ(f2, f1, ndig/2, ndig)
         call fm_equ(tol, tol2, ndsave, ndig)
         call fm_equ(a, x1, ndsave, ndig)

!             Special case for n = 0.

         if (n == 0) then
             f2 = f(x1, nf)
             d2 = f2
             if (abs(f2-f1) < tol2) then
                 call fm_equ(d2, return_value, ndig, ndsave)
                 ndig = ndsave
                 kwarn = kwarn_save
                 return
             endif
             cycle
         endif
         f2 = f1
```

```
!             Special case for n = 1.

          if (n == 1) then
              if (x1 /= 0) then
                  call fm_ulp(x1, h)
              else
                  h = epsilon(to_fm(1))
              endif
              h = sqrt(abs(h))
              xph = x1 + h
              xmh = x1 - h
              fph = f(xph, nf)
              fmh = f(xmh, nf)
              d2 = ( fph - fmh ) / (2*h)
              if (abs(d2-d1) < tol2 .and. j > 1) then
                  call fm_equ(d2, return_value, ndig, ndsave)
                  ndig = ndsave
                  kwarn = kwarn_save
                  return
              endif
              cycle
          endif

!             General case for even n > 1.

          if (mod(n, 2) == 0) then
              if (x1 /= 0) then
                  call fm_ulp(x1, h)
              else
                  h = epsilon(to_fm(1))
              endif
              h = abs(h)**(to_fm(1)/(n+2))
              fph = f(x1, nf)
              d2 = (-1)**(n/2) * binomial(to_fm(n), to_fm(n/2)) * fph
              do k = 0, n/2-1
                 xph = x1 + (n/2-k)*h
                 fph = f(xph, nf)
                 xmh = x1 - (n/2-k)*h
                 fmh = f(xmh, nf)
                 d2 = d2 + (-1)**k * binomial(to_fm(n), to_fm(k)) * (fph + fmh)
              enddo
              d2 = d2 / h**n
              if (abs(d2-d1) < tol2 .and. j > 1) then
                  call fm_equ(d2, return_value, ndig, ndsave)
                  ndig = ndsave
                  kwarn = kwarn_save
                  return
              endif
              cycle
          endif

!             General case for odd n > 1.

          if (mod(n, 2) == 1) then
              if (x1 /= 0) then
                  call fm_ulp(x1, h)
              else
```

```
                h = epsilon(to_fm(1))
            endif
            h = abs(h)**(to_fm(1)/(n+2))
            d2 = 0
            do k = 0, n/2
               xph = x1 + (n/2-k+1)*h
               fph = f(xph, nf)
               xmh = x1 - (n/2-k+1)*h
               fmh = f(xmh, nf)
               d2 = d2 + (-1)**k * binomial(to_fm(n-1), to_fm(k)) * ( fph - fmh ) *  &
                    to_fm(n*(n+1-2*k)) / ((n-k)*(n+1-k))
            enddo
            d2 = d2 / (2*h**n)
            if (abs(d2-d1) < tol2 .and. j > 1) exit
            cycle
         endif
      enddo

!             Round and return.

      call fm_equ(d2, return_value, ndig, ndsave)
      ndig = ndsave
      kwarn = kwarn_save
      end function fm_fprime

      subroutine fm_geneq(f, a, b, k, x, y, n)
      use fmvals
      use fmzm
      implicit none

!  Generate the k x k matrix a and k x 1 vector b of normal equations for the least square
!  fit of the k-parameter model

!     y = c(1)*f(1,x) + ... + c(k)*f(k,x)

!  to the data points (x(j),y(j)), j = 1, 2, ..., n.

!  a and b are returned, and then the coefficients c can be found by solving the
!  linear system  a * c = b.

!  Function l in the model evaluated at x is referenced by f(l,x) in this routine,
!  and f should be supplied as an external function subprogram by the user.

      integer :: k, n
      type (fm) :: a(k, k), b(k), x(n), y(n)
      type (fm), external :: f
      intent (in) :: k, x, y, n
      intent (inout) :: a, b

      type (fm), allocatable :: fxi(:)
      integer :: i, j, l, ndsave
      type (fm) :: xi, yi, fxil

      if (n <= 0 .or. k <= 0) then
          write (*, "(/' Error in fm_geneq.  k, n=', 2i8/)") k, n
          stop
      endif
```

```
      allocate(fxi(k), stat=j)
      if (j /= 0) then
          write (*, "(/' Error in fm_geneq.  Unable to allocate fxi with size ', i8/)") k
          stop
      endif

!             Raise precision.

      ndsave = ndig
      ndig = 2 * ndig

!             Initialize the upper triangle of a.

      do i = 1, k
         do j = i, k
            a(i, j) = 0
         enddo
         b(i) = 0
      enddo

!             Loop over the data points.

      do i = 1, n
         call fm_equ(x(i), xi, ndsave, ndig)
         call fm_equ(y(i), yi, ndsave, ndig)

!             Compute the k function values at x(i).

         do j = 1, k
            fxi(j) = f(j, xi)
         enddo

!             Multiply the function values and add the products to the matrix.

         do l = 1, k
            fxil = fxi(l)
            do j = l, k
               a(l, j) = a(l, j) + fxil*fxi(j)
            enddo

!             Sum the right-hand-side term.

            b(l) = b(l) + yi*fxil
         enddo
      enddo

!             Round back to the user's precision.

      do i = 1, k
         do j = i, k
            call fm_equ_r1(a(i, j), ndig, ndsave)
         enddo
         call fm_equ_r1(b(i), ndig, ndsave)
      enddo
      ndig = ndsave

!             Fill the lower triangle of the a matrix using symmetry.
```

```
      if (k >= 2) then
          do i = 2, k
             do j = 1, i-1
                a(i, j) = a(j, i)
             enddo
          enddo
      endif

      deallocate(fxi)
      end subroutine fm_geneq

      recursive subroutine fm_integrate(f, n, a, b, tol, result, kprt, nw)
      use fmvals
      use fmzm
      implicit none

!  High-precision numerical integration.

!  Integrate f(x,n) from a to b.  n is passed on to function f to indicate which function to use in
!  cases where several different functions may be defined there.


!  warning:  If the function f being integrated or one of its derivatives does not exist at one or
!            both of the endpoints (a,b), the endpoints should be exactly representable in fm's
!            number system.  For non-exact numbers like 1/3, sqrt(2), or pi/2, when fm_integrate
!            raises precision to evaluate the integration formula the endpoints are not accurate
!            enough at the higher precision.

!  Example:  Integrate  sqrt( tan( x ) )  from  0  to  pi/2.
!            First, pi/2 is not exact as an FM number.  At some precisions it may have rounded up,
!            making tan(x) negative and causing an error in sqrt.  Using sqrt( abs( tan( x ) ) )
!            is safer.
!            Second, b = pi/2 has been computed at the user's precision, so when fm_integrate
!            raises precision, the value of b is just zero-padded on the end, which does not give
!            enough information about how f(x) behaves near the singularity at pi/2.
!
!            Make the endpoints exact by changing variables.  Change the interval to [ 0 , 1 ]:
!
!                                    u = ( 2/pi ) * x   =>   du = ( 2/pi ) * dx
!            so
!                                    x = ( pi/2 ) * u   =>   dx = ( pi/2 ) * du
!            new limits
!                                    x = 0  =>  u = 0  and  x = pi/2  =>  u = 1
!
!            New integral:   Integrate  (pi/2) * sqrt( abs( tan( pi*u/2 ) ) )  from  0  to  1.
!
!            Now the function f should declare a local saved type(fm) variable pi, and then use
!            call fm_pi(pi) each time f is called to make sure the value of pi is correct at the
!            higher precision being used by fm_integrate when f is called.

      type (fm) :: a, b, result, tol
      integer :: n, kprt, nw
      type (fm), external :: f
      intent (in) :: n, a, b, tol, kprt, nw
      intent (inout) :: result

!  a,b,tol, and result are all type (fm) variables, and function f returns a type (fm) result.
```

```
!  result is returned as the value of the integral, with abs((result-true)/true) less than tol
!         (i.e., tol is a relative error tolerance).
!         For example, to get 30 significant digits correct for the integral, set tol = 1.0e-30.

!  FM precision must be set high enough in the calling program (using fm_set) so that 1+tol > 1
!  at that precision.  Using tol = epsilon(to_fm(1)) will usually get a full precision result,
!  but for some functions this might fail.  A better strategy is to set precision higher than
!  the accuracy required for the integral.  For example, to get the integral to 50 significant
!  digits, call fmset(60) and then set tol = to_fm(' 1.0e-50 ') before the call to fm_integrate.

!  kprt can be used to show intermediate results on unit nw.
!  kprt = 0 for no output
!       = 1 prints a summary for each call to fm_integrate
!       = 2 prints a trace of all iterations.

!  nw is the unit number used for kprt output and any error or warning messages.

!  No method for numerical integration is foolproof.  Since it samples only a finite number of
!  function values, any numerical integration algorithm can be made to fail by choosing a
!  sufficiently badly-behaved function.  Such functions often vary by many orders of magnitude
!  over relatively small fractions of the interval (a,b).

!  f should be well-behaved in the interior of the interval (a,b).
!  The routine tries to handle any singularities of f or f' at a and/or b, so cases with interior
!  singularities should be done as separate calls having the singularities as endpoints.
!  The routine will try to handle cases where f or f' has singularities inside (a,b), but then
!  the time will be much slower and the routine might fail.

!  For a function with a removable singularity in the interior of the interval, such as
!  f(x) = 1/ln(x) - 1/(x-1), define f(x,n) to check for x = 1 and return the correct limiting
!  value, 0.5 in this case, when x is 1.

!  Among functions with no singularities, examples of badly behaved functions are those with one
!  or more extremely narrow tall spikes in their graphs. If possible, identify the peaks of any
!  such spikes first, then make separate calls with the peaks as endpoints of the intervals of
!  integration.

!  If the value of the integral is zero or very close to zero, relative error may be undefined, so
!  this routine may fail to converge and then return unknown.  For these cases, try breaking the
!  integral into two pieces and calling twice to get two non-zero results.  These two results can
!  then be added, often giving the original integral with sufficiently small absolute error even
!  though small relative error could not be attained.

!  If the function values are too extreme, it can cause problems.  For example, if an exponential
!  in f underflows and then is multiplied by something bigger than one, then f will return unknown.
!  If the result of the integral is much larger than the underflow threshold (tiny(to_fm(1))), then
!  it is safe to set the underflowed results in f to zero to avoid getting unknown.

!  If the function is nearly divergent fm_integrate may fail.  1/x from 0 to b is divergent.
!  1/x**0.99 converges, but so slowly that fm_integrate may run a long time and then might fail.
!  1/x**0.9999 converges even more slowly and fm_integrate may fail by declaring that the integral
!  seems divergent.

!  When the integrand is highly (or infinitely) oscillatory, fm_integrate may fail.
!  If f has more than about 100 oscillations on the interval (a,b), it may be necessary to break
!  the interval into smaller intervals and call fm_integrate several times.
!  For infinitely many oscillations, like sin(1/x) from 0 to 1, first turn the integral into an
!  infinite series by calling fm_integrate to integrate each separate loop between roots of
```

```
!  sin(1/x).  The function is well-behaved for each call, so fm_integrate can get high precision
!  quickly for each.  Next form a sequence of k partial sums for this series.  The series converges
!  slowly, with 50 or 100 terms giving only 3 or 4 significant digits of the sum, so an
!  extrapolation method can be used to get a more accurate value of the sum of this series from
!  its first k terms. For an alternating series like this, the extrapolation method of Cohen,
!  Villegas, and Zagier often works very well.
!  Repeated Aitken extrapolation could be used instead -- it is a more widely known method.
!  Sample program Oscillate.f95 computes this integral.


!             m  is the maximum level for the integration algorithm.  The number of function
!                evaluations roughly doubles for each successive level until the tolerance is met.
!                Using m = 12 allows up to about 5,000 digits for most integrals, but the upper
!                limit for a given m depends on the function.
!                Raising m further will approximately double the maximum precision for each
!                extra level, but will also double the memory usage for each extra level.

      integer, parameter :: m = 12
      integer, parameter :: nt = 20*2**m
      type (fm), save :: st_save(0:nt)
      integer, save :: precision_of(0:nt) = 0
      integer :: absign, i, istep, k, kl, kwsave, nds, ndsave, nretry
      integer, save :: r_level = 0, num_f = 0

      type (fm) :: a1, ab2, b1, c1, c2, ci, ct, d, eps, err1, err2, f1, f2, fmax, h,       &
                   last_h, pi, prior_hs, s, s1, s2, si, st, t, tol1, tol2, x, x1, x2, xf,  &
                   hf, xmax, v, w

      character(80) :: st1, st2
      real :: time1, time2
      logical :: spike_found, st_is_saved

!             Iterative tanh-sinh integration is used, increasing the order until convergence
!             is obtained, or m levels have been done.

      result = 0

      ncall = ncall + 1
      namest(ncall) = 'integrate'
      kwsave = kw
      kw = nw
      r_level = r_level + 1
      nretry = 0
      if (kprt >= 2) then
          write (nw, "(a)") ' '
          write (nw, "(a, i9, a)") ' Input to fm_integrate.    Function n = ', n, '.    a, b ='
          call fm_print(a)
          call fm_print(b)
          call fm_form('es20.8', tol, st1)
          write (nw, "(a, a)") ' tol =', trim(st1)
      endif
      call cpu_time(time1)

!             Check for special cases.

      if (a == b) then
          a1 = f(a, n)
          if (r_level <= 1) then
```

```
            num_f = 1
        else
            num_f = num_f + 1
        endif
        if (a1%mfm%mp(2) == munkno) then
            call fm_f_fail(a, n, nw)
            prior_hs = a1
            call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                            num_f, r_level, kprt, nw, kwsave)
            return
        else
            prior_hs = 0
            call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                            num_f, r_level, kprt, nw, kwsave)
            return
        endif
    endif

!           Check to make sure the user has set precision high enough for the value of tol chosen.

    tol1 = tol
    if (tol < abs(epsilon(a))) then
        write (nw, "(a)") ' '
        write (nw, "(a)") ' Error in fm_integrate.  tol is '
        call fm_print(tol)
        write (nw, "(a)") ' This is too small for the current precision level.  Current epsilon ='
        tol1 = abs(epsilon(a))
        call fm_print(tol1)
        write (nw, "(a)") ' This larger value will be used.  tol ='
        call fm_print(tol1)
        write (nw, "(a)") ' Use fm_set to set a higher precision before the call to'
        write (nw, "(a)") ' fm_integrate if the smaller tol is needed.'
    endif

!           Raise the precision.
!           Check for an integrable singularity at either endpoint, and increase precision
!           if it seems that a retry would be needed at the first precision.

    ndsave = ndig
    x1 = a + (b-a)*to_fm(' 1.0e-10 ')
    x2 = a + (b-a)*to_fm(' 1.0e-20 ')
    f1 = f(x1, n)
    f2 = f(x2, n)
    a1 = log10(f1/f2)/10
    x1 = b - (b-a)*to_fm(' 1.0e-10 ')
    x2 = b - (b-a)*to_fm(' 1.0e-20 ')
    f1 = f(x1, n)
    f2 = f(x2, n)
    b1 = log10(f1/f2)/10
    if (r_level <= 1) then
        num_f = 4
    else
        num_f = num_f + 4
    endif
    if (a1 < -0.999 .or. b1 < -0.999) then
        prior_hs = to_fm(' unknown ')
        write (nw, "(a)") ' '
        write (nw, "(a, i9, a)") ' fm_integrate failed -- f(x, n) for n = ', n,  &
```

```
                              ' seems to have a non-integrable singularity'
         write (nw, "(a)")     '                          at an endpoint.  a, b ='
         call fm_print(a)
         call fm_print(b)
         write (nw, "(a)") ' Check the limits of integration, function number (n), and' // &
                           ' function definition.'
         write (nw, "(a)") ' '
         call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                         num_f, r_level, kprt, nw, kwsave)
         return
      endif
      ndig = ndig+int(30/alogmt)
      call fm_equ_r1(a1, ndsave, ndig)
      call fm_equ_r1(b1, ndsave, ndig)
      if (a1 < -0.2 .or. b1 < -0.2) ndig = 2*ndig

      call cpu_time(time1)

!             Start here when doing a retry.

      kl = 1
      do while (kl == 1)
         kl = 0
         nretry = nretry + 1
         call fm_equ(a, a1, ndsave, ndig)
         call fm_equ(b, b1, ndsave, ndig)
         absign = 1
         if (a1 > b1) then
             call fm_equ(b, a1, ndsave, ndig)
             call fm_equ(a, b1, ndsave, ndig)
             absign = -1
         else if (a1 == b1) then
             prior_hs = 0
             call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                             num_f, r_level, kprt, nw, kwsave)
             return
         endif
         call fm_equ(tol1, tol2, ndsave, ndig)

         if (kprt >= 2) then
             write (nw, "(a)") ' '
             write (nw, "(a, i9, a, i5)") ' Begin fm_integrate.  ndig = ', ndig,  &
                                          '   Recursion level = ', r_level
         endif

         s = 0
         prior_hs = 0
         err2 = 1
         eps = epsilon(tol2)
         d = abs(b1-a1)/100
         fmax = 0
         xmax = a1
         h = 1
         last_h = h/2**m
         call fm_pi(pi)

         hf = (b1-a1)/2
         ab2 = a1 + hf
```

```
      do k = 1, m
         h = h/2
         istep = 2**(m-k)
         if (k > 1) then
             t = istep*last_h
             call fm_chsh(t, c1, s1)
             t = 2*s1*c1
             c2 = c1**2 + s1**2
             s2 = t
         endif
         do i = 0, nt, istep
            if (mod(i, 2*istep) /= 0 .or. k == 1) then

!              The + or -x values are the abscissas for interval (-1,1).
!              xf translates these to the interval (a,b).

               if (i == 0) then
                   x = 0
                   w = pi/2
                   xf = hf*x + ab2
                   t = f(xf, n)
                   num_f = num_f + 1
                   if (t%mfm%mp(2) == munkno) then
                       call fm_f_fail(xf, n, nw)
                       prior_hs = t
                       call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                                       num_f, r_level, kprt, nw, kwsave)
                       return
                   endif
                   if (abs(t) > fmax .and. xf > a1+d .and. xf < b1-d) then
                       fmax = abs(t)
                       xmax = xf
                   endif
                   s = s + w*hf*t
               else
                   if (k == 1) then
                       t = i*last_h
                       call fm_chsh(t, ci, si)
                   else

!                       Use the hyperbolic addition formulas to get the next cosh and sinh
!                       quickly when evaluated at i*last_h.

                      if (i == istep) then
                          ci = c1
                          si = s1
                      else
                          t  = si*c2 + ci*s2
                          ci = ci*c2 + si*s2
                          si = t
                          c1 = ci
                          s1 = si
                      endif
                   endif
                   st_is_saved = .false.
                   if (allocated(st_save(i)%mfm%mp)) then
                       if (precision_of(i) >= ndig) st_is_saved = .true.
                   endif
```

```
            if (st_is_saved) then
                st = st_save(i)
                ct = sqrt(1+st**2)
            else
                t = pi*si/2
                call fm_chsh(t, ct, st)
                st_save(i) = st
                precision_of(i) = ndig
            endif
            w = (pi/2)*ci/ct**2
            if (w < eps) exit
            x = st/ct
            xf = hf*(-x) + ab2
            if (xf > a1) then
                t = f(xf, n)
                num_f = num_f + 1
                if (t%mfm%mp(2) == munkno) then
                    call fm_f_fail(xf, n, nw)
                    prior_hs = t
                    call fm_int_end(prior_hs, result, ndsave, time1, time2, n,  &
                                    a, b, tol, num_f, r_level, kprt, nw, kwsave)
                    return
                endif
                if (abs(t) > fmax .and. xf > a1+d .and. xf < b1-d) then
                    fmax = abs(t)
                    xmax = xf
                endif
                s = s + w*hf*t
            endif
            xf = hf*(x) + ab2
            if (xf < b1) then
                t = f(xf, n)
                num_f = num_f + 1
                if (t%mfm%mp(2) == munkno) then
                    call fm_f_fail(xf, n, nw)
                    prior_hs = t
                    call fm_int_end(prior_hs, result, ndsave, time1, time2, n,  &
                                    a, b, tol, num_f, r_level, kprt, nw, kwsave)
                    return
                endif
                if (abs(t) > fmax .and. xf > a1+d .and. xf < b1-d) then
                    fmax = abs(t)
                    xmax = xf
                endif
                s = s + w*hf*t
            endif
        endif
    endif
enddo
if (kprt >= 2) then
    write (nw, "(a)") ' '
    write (nw, "(a, i9, a, i9, a)") ' k = ', k, '    ', num_f,  &
                                ' function calls so far.   Integral approximation ='
    v = h*s
    call fm_print(v)
endif
if (k > 1) then
    err1 = err2
```

```
            if (s /= 0) then
                err2 = abs( (prior_hs - h*s)/(h*s) )
            else
                err2 = abs( (prior_hs - h*s) )
            endif
            if (kprt >= 2) then
                call fm_form('es15.3', err2, st1)
                write (nw, "(a, a)") '      relative error of the last two ' //  &
                                     'approximations = ', trim(st1)
            endif

!           Check for convergence.

            if (k > 3 .and. err2 > 0 .and. err2 < tol2/10.0) exit
            if (k > 5 .and. err2 == 0) exit

!           If the errors do not decrease fast enough, raise precision and try again.

            if (k > 3*nretry .and. err1 > 0 .and. err2 > 0) then
                if (log(err2)/log(err1) < 1.2 .and. err1 < 1.0d-6) then
                    ndig = 2*ndig
                    if (kprt >= 2) then
                        write (nw, "(a, i9, a, i9)") ' fm_integrate Retry.  So far,' //  &
                                                     ' num_f = ', num_f, '   New ndig = ', ndig
                    endif
                    if (nretry <= 3) then
                        kl = 1
                        exit
                    endif
                    ndig = ndig/2
                endif
            endif
        endif
        prior_hs = h*s

!       No convergence in m iterations.
!       Before giving up, look for an interior singularity or tall spike.  If one is found,
!       split (a,b) into two intervals with the interior singularity as an endpoint, and try
!       again as two integrals.

        if (k == m .or. (k >= 9 .and. err2 > 1.0d-7 .and. abs(tol2) < 1.0d-16)) then
            if (kprt >= 2) then
                write (nw, "(a)") ' '
                write (nw, "(a, i6, a)") ' No convergence in ', m,  &
                             ' iterations.  Look for an interior singularity.'
                call fm_form('es25.6', xmax, st1)
                call fm_form('es25.6', fmax, st2)
                write (nw, "(i9, a, a, a)") num_f, ' function calls so far.   xmax, fmax =',  &
                                            trim(st1), trim(st2)
            endif
            call fm_spike(f, n, a1, b1, xmax, fmax, num_f, spike_found, kprt, nw)
            call fmequ_r1(a1%mfm, ndig, ndsave)
            call fmequ_r1(b1%mfm, ndig, ndsave)
            call fmequ_r1(xmax%mfm, ndig, ndsave)
            nds = ndig
            ndig = ndsave
            if (spike_found) then
                if (min(abs(a-xmax), abs(b-xmax)) < 1.01*d) then
```

```
                    ndig = 2*nds
                    if (nretry <= 5) then
                        kl = 1
                        exit
                    endif
                    ndig = ndsave
                endif
                if (kprt >= 2) then
                    write (nw, "(a)") ' '
                    write (nw, "(a)") ' Split the integral.  First half:  a, b = '
                    call fm_print(a1)
                    call fm_print(xmax)
                endif
                call fm_integrate(f, n, a1, xmax, tol, c1, kprt, nw)
                if (c1%mfm%mp(2) == munkno) then
                    prior_hs = c1
                    call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                                    num_f, r_level, kprt, nw, kwsave)
                    return
                endif
                if (kprt >= 2) then
                    write (nw, "(a)") ' '
                    write (nw, "(a)") ' Split the integral.  Second half:  a, b = '
                    call fm_print(xmax)
                    call fm_print(b1)
                endif
                call fm_integrate(f, n, xmax, b1, tol, c2, kprt, nw)
                prior_hs = c1 + c2
                call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                                num_f, r_level, kprt, nw, kwsave)
                return
            endif
            call fm_int_fail(n, a, b, tol, m, err2, prior_hs, nw)
            prior_hs = to_fm(' unknown ')
            call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                            num_f, r_level, kprt, nw, kwsave)
            return
        endif
    enddo
enddo

prior_hs = absign*h*s

!           Round the result and return.

call fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                num_f, r_level, kprt, nw, kwsave)

return
end subroutine fm_integrate

subroutine fm_int_end(prior_hs, result, ndsave, time1, time2, n, a, b, tol,  &
                      num_f, r_level, kprt, nw, kwsave)
use fmvals
use fmzm
implicit none
integer :: ndsave, r_level, kprt, nw, n, num_f, kwsave
type (fm) :: prior_hs, result, a, b, tol
```

```
real :: time1, time2
intent (in) :: prior_hs, ndsave, time1, n, a, b, tol, num_f, kprt, nw, kwsave
intent (inout) :: result, time2, r_level

character(80) :: st1

call fm_equ(prior_hs, result, ndig, ndsave)

ndig = ndsave
ncall = ncall - 1
call cpu_time(time2)

if (kprt >= 2 .or. ( r_level <= 1 .and. kprt == 1 ) ) then
    write (nw, "(a)") ' '
    write (nw, "(a, i9, a)") ' Return from fm_integrate.    Function n = ', n, '.    a, b ='
    call fm_print(a)
    call fm_print(b)
    call fm_form('es20.8', tol, st1)
    write (nw, "(a, a)") ' tol =', trim(st1)
    if (abs(time2-time1) > 0.0001 .and. abs(time2-time1) < 1000.0) then
        write (nw, "(1x, i9, a, f9.5, a)") num_f, ' function calls were made in ',  &
                                           time2-time1, ' seconds.'
        write (nw, "(a)") ' result ='
    else
        write (nw, "(1x, i9, a, es14.5, a)") num_f, ' function calls were made in ',  &
                                             time2-time1, ' seconds.'
        write (nw, "(a)") ' result ='
    endif
    call fm_print(result)
endif
kw = kwsave
r_level = r_level - 1

return
end subroutine fm_int_end

subroutine fm_int_fail(n, a, b, tol, m, err, val, nw)
use fmzm
implicit none
integer :: n, m, nw
type (fm) :: a, b, tol, err, val
intent (in) :: n, a, b, tol, m, err, val, nw

write (nw,*) ' '
write (nw,*) ' fm_integrate failed -- no convergence in ', m, ' iterations.'
write (nw,*) ' unknown has been returned in result.'
write (nw,*) ' Possible causes:  (1) highly oscillatory integrand'
write (nw,*) '                   (2) non-convergent integral'
write (nw,*) '                   (3) integrable singularity in the interior of interval (a,b)'
write (nw,*) '                   (4) narrow spike in the interior of interval (a,b)'
write (nw,*) '                   (5) integral too close to zero'
write (nw,*) ' a possible remedy for the last 3 is to split the integral into two pieces,'
write (nw,*) ' making two calls to fm_integrate and then adding the two results.'
write (nw,*) ' Put singularities or spikes at the endpoints of the intervals of integration.'
write (nw,*) ' '
write (nw,*) ' Function n = ', n, '.    a, b ='
call fm_print(a)
call fm_print(b)
```