```fortran
      program test
      use fmzm
      implicit none

!  Examples and advice for using fm_integrate.

      type (fm), save :: a, b, check, err, pi, r1, r2, result, seven, tol
      type (fm), external :: f
      integer :: k, kprt, n, n_errors, nw
      character(80) :: st1

      n_errors = 0



!            1.  Start with an integral without singularities on the interval of integration.

!                integrate  log(t) * cos(t)  from  pi/4 to pi/2.

!                Set the tolerance to get at least 40 significant digits.
!                fm_integrate does a sequence of iterations using the tanh-sinh quadrature formula.
!                Each iteration uses more points until the last two iterates agree within the
!                specified tolerance.  Since the next-to-last iterate satisfies the tolerance
!                and the last iterate (returned as result) is even more accurate, the value
!                returned from fm_integrate is usually slightly more accurate than requested.
!                In this case, the error check below shows that result is actually correct
!                to about 60 digits.

!                It is usually best to set FM's precision level to be slightly higher than the
!                number of digits requested with the tolerance.  Here we set precision to 20
!                more digits.

      n = 1

!            Call fm_set to define FM's precision level before any multiple precision variables
!            are defined.  This sets 60-digit precision and tol is 1.0e-40.

      k = 40
      call fm_set(k+20)
      tol = to_fm(10) ** (-k)

      write (*, "(//)")
      call fm_pi(pi)

!            a and b are the limits for the integral.

      a = pi / 4
      b = pi / 2

!            kprt controls trace printing in fm_integrate.  Setting it to 1 will print a summary
!                 of the call, giving the result, number of function evaluations, and time.
!            nw   is the unit number for this trace output.

      kprt = 1
      nw = 6

      call fm_integrate(f, n, a, b, tol, result, kprt, nw)
```

```
!           For these sample problems the integrals have known closed-form results, so
!           we can check the accuracy of fm_integrate.

      check = log(pi/2) - log(pi/4)/sqrt(to_fm(2)) + sin_integral(pi/4) - sin_integral(pi/2)

      err = abs( result - check )
      call fm_form('es12.4', err, st1)
      write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
      if (err > tol) n_errors = n_errors + 1



!           2.  Next do an integral with a singularity at zero
!               (from "Integrals of Powers of LogGamma" by T. Amdeberhan, M. Coffey, O. Espinosa,
!               C. Koutschan, D. Manna, and V. Moll, in Proc. Amer. Math. Soc., #139, 2011)

!               integrate  log( gamma(t) )  from  0 to 1.

!               Leave precision and tolerance the same as above.

!               The tanh-sinh algorithm is good at handling singularities at the endpoints,
!               so this case takes about the same number of function evaluations as case 1.

      n = 2
      write (*, "(//)")
      a = 0
      b = 1
      tol = to_fm(10) ** (-k)
      kprt = 1
      nw = 6

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

      check = log( sqrt( 2*pi ) )
      err = abs( result - check )
      call fm_form('es12.4', err, st1)
      write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
      if (err > tol) n_errors = n_errors + 1



!           3.  This integral has a pole at pi/2 and a sqrt singularity at zero.
!               (from "a Comparison of Three High-Precision Quadrature Schemes" by D. H. Bailey,
!               K. Jeyabalan, and X. S. Li, in Experimental Mathematics, Vol 14 (2005), No. 3)

!               integrate  sqrt( tan(t) )  from  0 to pi/2.

!               Set tolerance to give 100 digits.

!               Here the fact that the pi/2 endpoint is not exactly representable in floating
!               point form causes a problem.  fm_integrate will increase precision above the
!               user's level while computing the integral.  But the endpoints a and b are input
!               values that were defined at the user's precision, and their extra digits will be
!               zeros when precision is raised in fm_integrate.

!               That is fine for a = 0, but b = pi/2 will still be accurate only to the user's
!               precision, not to the higher intermediate precision.  The fact that b is a
```

```
!                 singularity for the function means that fm_integrate will need to know the
!                 position of b to higher precision to evaluate the integral accurately.

!                 The fix is to make a change of variables to get an equivalent integral where
!                 both endpoints are exact in floating point.  Leaving a singular endpoint inexact
!                 will usually cause fm_integrate to run much slower, and sometimes fail.

!                 Let  u = t * 2 / pi.  Then  t = u * pi / 2  and  dt = pi/2 du.
!                 The new form of this integral becomes:

!                 integrate  sqrt( abs( tan( u * pi / 2 ) ) ) * pi / 2  from  0 to 1.

!                 Now pi will be computed inside function f, so it will be done at whatever higher
!                 precision fm_integrate uses.

!                 When changing variables in this case, we also need to defend against rounding
!                 errors when computing  u * pi / 2.  When u is very close to 1, rounding could
!                 cause u * pi / 2 to round up, giving a value slightly greater than pi/2.
!                 Then tan would return a negative value and then sqrt would return unknown,
!                 causing the integration to fail.  The fix is to take the absolute value before
!                 doing the square root.

      n = 3
      k = 100
      call fm_set(k+20)

!             Precision has increased, so we must get pi at the new precision.

      call fm_pi(pi)
      write (*, "(//)")
      a = 0
      b = 1
      tol = to_fm(10) ** (-k)
      kprt = 1
      nw = 6

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

      check = pi * sqrt( to_fm(2) ) / 2
      err = abs( result - check )
      call fm_form('es12.4', err, st1)
      write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
      if (err > tol) n_errors = n_errors + 1



!             4.  integrate  exp( -t**2 / 2 )  from  0 to infinity.
!                 (from "a Comparison of Three High-Precision Quadrature Schemes" by D. H. Bailey,
!                 K. Jeyabalan, and X. S. Li, in Experimental Mathematics, Vol 14 (2005), No. 3)

!                 Set tolerance to give 100 digits.

!                 Infinite regions must be converted to finite ones.  Let u = 1/(t+1) to get:

!                 integrate  exp( -(1/u - 1)**2 / 2 ) / u**2  from  0 to 1.

!                 Exponential functions pose another problem for fm_integrate.
!                 When u is very close to zero the exponential can underflow.  FM does not flush
```

```
!                 underflows to zero like most floating point systems, so when that value is
!                 then divided by the small u**2 FM detects the possibility that this result
!                 could be above the underflow threshold.  Since FM can't be sure whether the
!                 true function value is below the underflow threshold, unknown is returned.

!                 The fix in this case is to see that whenever underflow occurs in this integration
!                 the final function value is too small to change the integral.  That is the usual
!                 situation whenever f underflows and the final value of the integral is greater
!                 than 10**(-10**6) in magnitude, because FM's underflow is less than 10**(-10**8).
!                 So we check for underflow after doing the exponential in function f and replace
!                 underflowed function values by zero.

!                 Starting with the 2022 version of FM this check for intermediate underflow can
!                 usually be skipped.  Some extra information is now included in underflowed or
!                 overflowed results, so that usually the program can tell in cases like this
!                 that when the exp function underflows, after dividing by u**2 for a small u the
!                 function value is still in the underflow region.  Then FM can return underflow
!                 for the function value instead of the unknown result in previous versions.

!                 The old check for underflow has been left in the f(x) routine in this program,
!                 since there are still some rare cases where it might be needed.

      n = 4
      k = 100
      call fm_set(k+20)
      call fm_pi(pi)
      write (*, "(//)")
      a = 0
      b = 1
      tol = to_fm(10) ** (-k)
      kprt = 1
      nw = 6

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

      check = sqrt( pi / 2 )
      err = abs( result - check )
      call fm_form('es12.4', err, st1)
      write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
      if (err > tol) n_errors = n_errors + 1



!             5.  integrate  log( abs( ( tan(t) + sqrt(7) ) / ( tan(t) - sqrt(7) ) ) )
!                            from  pi/3 to pi/2.
!                 (from "High-Precision Numerical Integration: Progress and Challenges"
!                 by D. H. Bailey and J. M. Borwein (2009))

!                 Set tolerance to give 150 digits.

!                 There is only one singularity, but it is atan(sqrt(7)), which is not an endpoint.
!                 fm_integrate will initially have very slow convergence and then will try to
!                 isolate the singularity and split into two integrals with the singularity at
!                 endpoints.

!                 This strategy works here, but it is slower and doesn't always succeed.
!                 Case 6 shows a better way to handle interior singularities.
```

```
      n = 5
      k = 150
      call fm_set(k+20)
      call fm_pi(pi)
      write (*, "(//)")
      a = pi/3
      b = pi/2
      tol = to_fm(10) ** (-k)
      kprt = 1
      nw = 6

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

      seven = 7
      check = ( sqrt(seven) / 168 ) * ( polygamma(1, 1/seven) + polygamma(1, 2/seven) -  &
                                        polygamma(1, 3/seven) + polygamma(1, 4/seven) -  &
                                        polygamma(1, 5/seven) - polygamma(1, 6/seven) )
      err = abs( result - check )
      call fm_form('es12.4', err, st1)
      write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
      if (err > tol) n_errors = n_errors + 1



!             6.  Same integral as case 5.

!                 integrate  log( abs( ( tan(t) + sqrt(7) ) / ( tan(t) - sqrt(7) ) ) )
!                            from  pi/3 to pi/2.

!                 Set tolerance to give 150 digits.

!                 Split into two integrals and change variables to make the endpoints exact.
!                 This will be faster than making fm_integrate search for the interior singularity
!                 as in case 5.
!                 Call the two function numbers 61 and 62.

!                 1.  from  pi/3  to  atan(sqrt(7)).
!                     Let  u = ( t - pi/3 ) / ( atan( sqrt(7) ) - pi/3 )

!                 2.  from  atan(sqrt(7)) to pi/2.
!                     Let  v = ( t - atan(sqrt(7)) ) / ( pi/2 - atan( sqrt(7) ) )

!                 This gives two integrals from 0 to 1, then we add the two results.

      n = 6
      k = 150
      call fm_set(k+20)
      call fm_pi(pi)
      write (*, "(//)")
      a = 0
      b = 1
      tol = to_fm(10) ** (-k)
      kprt = 1
      nw = 6

      call fm_integrate(f, 61, a, b, tol, r1, kprt, nw)
      call fm_integrate(f, 62, a, b, tol, r2, kprt, nw)
      result = r1 + r2
```

```
    write (*,*) ' '
    write (*,*) ' Adding these last two integrals gives the case 6 result:'
    write (*,*) ' '
    call fm_print(result)

    seven = 7
    check = ( sqrt(seven) / 168 ) * ( polygamma(1, 1/seven) + polygamma(1, 2/seven) -  &
                                      polygamma(1, 3/seven) + polygamma(1, 4/seven) -  &
                                      polygamma(1, 5/seven) - polygamma(1, 6/seven) )
    err = abs( result - check )
    call fm_form('es12.4', err, st1)
    write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
    if (err > tol) n_errors = n_errors + 1



!             7.  Same integral as cases 5 and 6.
!                 Combine these two integrals into one, so only one call to fm_integrate is needed.
!                 This will be faster than doing two calls as in case 6.

!                 integrate  log( abs( ( tan(t) + sqrt(7) ) / ( tan(t) - sqrt(7) ) ) )
!                            from  pi/3 to pi/2.

!                 Set tolerance to give 150 digits.

!                 Split into two integrals and change variables to make the endpoints exact.
!                 Both new integrals are from 0 to 1.

!                 1.  from  pi/3  to  atan(sqrt(7)).
!                     Let  u = ( t - pi/3 ) / ( atan( sqrt(7) ) - pi/3 )

!                 2.  from  atan(sqrt(7)) to pi/2.
!                     Let  v = ( t - atan(sqrt(7)) ) / ( pi/2 - atan( sqrt(7) ) )

    n = 7
    k = 150
    call fm_set(k+20)
    call fm_pi(pi)
    write (*, "(//)")
    a = 0
    b = 1
    tol = to_fm(10) ** (-k)
    kprt = 1
    nw = 6

    call fm_integrate(f, 7, a, b, tol, result, kprt, nw)

    seven = 7
    check = ( sqrt(seven) / 168 ) * ( polygamma(1, 1/seven) + polygamma(1, 2/seven) -  &
                                      polygamma(1, 3/seven) + polygamma(1, 4/seven) -  &
                                      polygamma(1, 5/seven) - polygamma(1, 6/seven) )
    err = abs( result - check )
    call fm_form('es12.4', err, st1)
    write (*, "(/10x, a, i2, a, a)") ' Error for case ', n, ' = ', trim(st1)
    if (err > tol) n_errors = n_errors + 1
```

```
      write (*,*) ' '
      write (*,*) ' '
      if (n_errors == 0) then
          write (*,*) ' All results were ok -- no errors were found.'
      else
          write (*,*) n_errors, ' error(s) were found.'
      endif
      write (*,*) ' '

      stop
      end program test

      function f(x, n)     result (return_value)
      use fmzm
      implicit none

      type (fm) :: return_value, x
      integer :: n
      intent (in) :: x, n
      type (fm), save :: c1, c2, pi, sqrt7, tanx

      if (n == 1) then
          return_value = log(x) * cos(x)
      else if (n == 2) then
          return_value = log( gamma( x ) )
      else if (n == 3) then

!             The original limits from 0 to pi/2 have been changed to 0 to 1.

          call fm_pi(pi)
          return_value = pi * sqrt( abs( tan( pi * x / 2 ) ) ) / 2
      else if (n == 4) then

!                Before the 2023 version of FM, exp could underflow and then make f unknown.
!                The previous code here checked for underflow and set f = 0 in that case.

!          return_value = exp( -(1 - 1/x)**2 / 2 )
!          if ( is_underflow(return_value) ) then
!              return_value = 0
!          else
!              return_value = return_value / x**2
!          endif

!                Starting with the 2023 version, FM's exception handling is stronger, so
!                now these undeflows in exp don't need to be trapped and the integration
!                works as intented.

          return_value = exp( -(1 - 1/x)**2 / 2 ) / x**2
      else if (n == 5) then
          sqrt7 = sqrt(to_fm(7))
          tanx = tan(x)
          return_value = log( abs( ( tanx + sqrt7 ) / ( tanx - sqrt7 ) ) )
      else if (n == 61) then
          call fm_pi(pi)

!             It is tempting to compute constants like c1, c2, sqrt7 once and then save them for
!             use in subsequent calls to f.  That can be done, but it is trickier than it seems,
```

```
!              since fm_integrate may call f with different precision levels during one integration,
!              so it is easy to not have the right precision in a saved variable.
!              Here we just compute them each time, making the logic straightforward while the
!              function evaluations are somewhat slower.

          sqrt7 = sqrt(to_fm(7))
          c1 = atan(sqrt7) - pi/3
          tanx = tan( c1*x + pi/3 )
          return_value = c1 * log( abs( ( tanx + sqrt7 ) / ( tanx - sqrt7 ) ) )
      else if (n == 62) then
          call fm_pi(pi)
          sqrt7 = sqrt(to_fm(7))
          c2 = atan(sqrt7)
          c1 = pi/2 - c2
          tanx = tan( c1*x + c2 )
          return_value = c1 * log( abs( ( tanx + sqrt7 ) / ( tanx - sqrt7 ) ) )
      else if (n == 7) then

!              Combine the two integrals into one.

          call fm_pi(pi)
          sqrt7 = sqrt(to_fm(7))
          c2 = atan(sqrt7)
          c1 = c2 - pi/3
          tanx = tan( c1*x + pi/3 )
          return_value = c1 * log( abs( ( tanx + sqrt7 ) / ( tanx - sqrt7 ) ) )

          c1 = pi/2 - c2
          tanx = tan( c1*x + c2 )
          return_value = return_value + c1 * log( abs( ( tanx + sqrt7 ) / ( tanx - sqrt7 ) ) )
      endif

      end function f
```