```
!   This is a sample program using version 1.4 of the fmzm and fm_interval_arithmetic modules
!   for doing interval arithmetic using the fm_interval derived type.

!   The output is saved in file SampleFMinterval.out.  A comparison file, SampleFMinterval.chk,
!   is provided showing the expected output from machines using 64-bit double precision and IEEE
!   arithmetic.  This would give about 16 significant digit accuracy for a stable calculation.
!   When run on other computers, all the multiple precision results should be the same, and the
!   results from the machine precision (d.p.) calculations will be different.
!   The program checks all the results and the last line of the output file should be
!   "All results were ok."

!  Sample 3 below uses an array-valued function of type fm_interval.
!  The function is defined here in a module with an explicit interface.

  module exp_sum_mod

   interface exp_sum
      module procedure exp_sum3
   end interface

   contains

      function exp_sum3(r_fm)     result (return_value)

!  Sample function usage for type fm_interval.

!  The test function is exp(x) = 1 + x + x**2/2! + x**3/3! + ...
!  summed for the three values of x in array r_fm

      use fmzm
      use fm_interval_arithmetic
      implicit none
      type (fm) :: r_fm(3)
      intent (in) :: r_fm
      type (fm_interval) :: return_value(3)
      type (fm_interval), save :: s, t, x
      integer :: j, k

      do j = 1, 3
         s = 1
         t = 1
         x = r_fm(j)
         do k = 1, 1000
            t = t * x / k
            s = s + t
            if (abs(t) < to_fm('1.0e-75')) exit
         enddo
         return_value(j) = s
      enddo

      end function exp_sum3

  end module exp_sum_mod



      program test
```

```
      use exp_sum_mod
      use fm_interval_arithmetic
      implicit none

!             Declare the multiple precision variables.
!             (fm) for multiple precision reals
!             (fm_interval) for multiple precision real intervals

      type (fm), save :: digits_lost_fm, error_fm, fact_fm, r_fm(3), s_fm, s2_fm, t_fm, x_fm, x2_fm
      type (fm_interval), save :: fact_fm_interval, s_fm_interval, t_fm_interval,  &
                                  x_fm_interval, x2_fm_interval, exp_sum_interval(3)

!             Declare the other variables (not multiple precision).

      character(80)  :: st1, stf
      integer :: e(3), j, k, kout, nerror
      double precision :: fact, s, t, x, x2

!             Write output to the file SampleFMinterval.out.

      kout = 18
      open (kout, file='SampleFMinterval.out')

      nerror = 0

!  ------------------------------------------------------------------------------------Sample 1

!             One of the common uses for multiple precision and also interval arithmetic is to
!             test the accuracy and stability of an algorithm.

!             Here is a sum that theoretically converges to the Bessel function
!             j(1, x) for x = 35.

!             1.  Try it using double precision.
!                 Printing the partial sums each 5 terms shows that this formula is unstable for
!                 x = 35, since some of the partial sums are more than 1.0e+14 times larger than
!                 the final sum.  This makes it seem that we have lost at least 14 significant
!                 digits to cancellation.
!                 For this example it is fairly clear from the double precision output that the
!                 final value of s is not accurate, but that might not be easy to see for a more
!                 complicated calculation.

      write (kout,*) ' '
      write (kout,*) ' '
      write (kout,*) ' Sample 1.  Unstable summation.'
      write (kout,*) ' '
      write (kout,*) ' '
      write (kout,*) ' 1.  Do the sum in double precision.'
      write (kout,*) ' '
      s = 0
      x = 35.0d0/2
      x2 = -(x**2)
      fact = 1
      do k = 0, 70
         t = x / ( (k+1) * fact**2 )
         if ( abs(t) < epsilon(s)*abs(s) ) then
             write (kout, "(a, i3, a, es25.15)") '      k = ', k, '    s = ', s
             exit
```

```
         endif
         s = s + t
         x = x * x2
         fact = fact * (k+1)
         if (mod(k, 5) == 0) then
             write (kout, "(a, i3, a, es25.15)") '      k = ', k, '    s = ', s
         endif
      enddo
      if (abs(s-4.399d-2) > 3.0d-3) then
          nerror = nerror + 1
          write (kout,*) ' '
          write (kout,*) ' Error in case 1 (or double precision accuracy is not 53 bits).'
          write (kout,*) ' '
      endif

!             2.  Try it using multiple precision, with 20, 30, 40, and 50 significant digits.
!                 To measure the error each time, compute it first with 100 digits.

!                 The error is measured in ulps (units in the last place), since the actual
!                 accuracy is slightly more than the number of digits requested.

!                 When FM uses the default large base (10^7 is typical for 64-bit double precision)
!                 this test will show about 18 (base 10) digits lost during the calculation.

!                 The reason for the "at least" in the descriptions below is that if the base is
!                 10^7, then the first word of the multiple precision number can have from 1 to 7
!                 base 10 digits.  So asking for 20 digit precision with call fm_set(20) gives
!                 5 digits base 10^7, since we want a few guard digits past 20, and using 4 digits
!                 base 10^7 would guarantee only 1 + 3*7 = 22 decimal digits.  With 5 digits every
!                 intermediate value in the computation will have from 29 to 35 significant digits.

      write (kout,*) ' '
      write (kout,*) ' 2.  Use FM with increasing precision.'
      write (kout,*) ' '
      write (kout,*) '     Setting precision to j digits via call fm_set(j) will actually set the'
      write (kout,*) '     equivalent number of decimal significant digits slightly higher than j.'
      write (kout,*) '     For example, if the base used internally in FM is 10**7, then asking for'
      write (kout,*) '     20 digits with call fm_set(20) gives at least 29 significant digits.'
      write (kout,*) '     call fm_set(30) gives at least 36 significant digits.'
      write (kout,*) '     call fm_set(40) gives at least 50 significant digits.'
      write (kout,*) '     call fm_set(50) gives at least 57 significant digits.'
      write (kout,*) ' '
      call fm_set(100)
      s2_fm = 0
      x_fm = 35.0d0/2
      x2_fm = -(x_fm**2)
      fact_fm = 1
      do k = 0, 700
         t_fm = x_fm / ( (k+1) * fact_fm**2 )
         if ( abs(t_fm) < epsilon(s2_fm)*abs(s2_fm) ) exit
         s2_fm = s2_fm + t_fm
         x_fm = x_fm * x2_fm
         fact_fm = fact_fm * (k+1)
      enddo

      do j = 20, 50, 10
         call fm_set(j)
         s_fm = 0
```

```
         x_fm = 35.0d0/2
         x2_fm = -(x_fm**2)
         fact_fm = 1
         do k = 0, 700
            t_fm = x_fm / ( (k+1) * fact_fm**2 )
            s_fm = s_fm + t_fm
            if ( abs(t_fm) < epsilon(s_fm)*abs(s_fm) ) then
                write (stf,*) ' f', j+3, '.', j
                call fm_form(trim(stf), s_fm, st1)
                write (kout, "(5x, i3, a, i3, a, a)") j, ' digits, ', k, ' terms gave   s_fm = ',  &
                                                      trim(st1)
                call fm_ulp(s_fm, t_fm)

!                  Since s2_fm was computed at a different precision than s_fm, we should
!                  round it to the current precision.  If we knew the two values of the FM
!                  internal variable ndig that were used in computing s2_fm and s_fm, the
!                  standard FM rounding routine fm_equ could be used.  Here we used fm_set
!                  to ask for slightly more than 100 and j decimal digits for s2_fm and s_fm,
!                  so the routine round_fm in this program gets those values of ndig and does
!                  the rounding.

                call round_fm(s2_fm, x2_fm, 100, j)
                error_fm = abs( (s_fm-x2_fm)/t_fm )
                digits_lost_fm = nint(log10(error_fm))
                write (kout, "(a, i3, a)") '               This calculation lost about ',  &
                                        to_int(digits_lost_fm), ' digits.'
                exit
            endif
            x_fm = x_fm * x2_fm
            fact_fm = fact_fm * (k+1)
         enddo
      enddo
      if (abs(s_fm-to_fm('.043990942179625639969698970659742471927005039845l1')) > 1.0d-35) then
          nerror = nerror + 1
          write (kout,*) ' '
          write (kout,*) ' Error in case 2.'
          write (kout,*) ' '
      endif

!             3.  Sometimes we want to measure the errors using base 2 arithmetic in fm,
!                 to more accurately reflect what is happening in the d.p. calculation.
!                 Set FM to use base 2, and do the calculation with 53 bits of precision
!                 (64-bit d.p.), then 73, 93, 113 bits.

!                 We want exact control over the base and precision, so use fm_setvar
!                 instead of fm_set.

!                 This shows a loss of 14 or 15 (base 10) digits when using base 2.

!                 This is typical of the comparison between using FM with the default large base
!                 and base 2.  Normalization error is larger with a large base, but the 30 s.d.
!                 calculation in case 2 gets about the same accuracy as the 113-bit calculation
!                 in case 3, and using base 2 is much slower.

      write (kout,*) ' '
      write (kout,*) ' 3.  Use FM with increasing precision in base 2.'
      write (kout,*) ' '
      call fm_setvar(" mbase = 2 ")
```

```
      call fm_setvar(" ndig = 150 ")
      s2_fm = 0
      x_fm = 35.0d0/2
      x2_fm = -(x_fm**2)
      fact_fm = 1
      do k = 0, 700
         t_fm = x_fm / ( (k+1) * fact_fm**2 )
         if ( abs(t_fm) < epsilon(s2_fm)*abs(s2_fm) ) exit
         s2_fm = s2_fm + t_fm
         x_fm = x_fm * x2_fm
         fact_fm = fact_fm * (k+1)
      enddo

      do j = 53, 113, 20
         write (stf,*) ' ndig =', j
         call fm_setvar(trim(stf))
         s_fm = 0
         x_fm = 35.0d0/2
         x2_fm = -(x_fm**2)
         fact_fm = 1
         do k = 0, 700
            t_fm = x_fm / ( (k+1) * fact_fm**2 )
            s_fm = s_fm + t_fm
            if ( abs(t_fm) < epsilon(s_fm)*abs(s_fm) ) then
                write (stf,*) ' f', nint(j*0.301)+3, '.', nint(j*0.301)+1
                call fm_form(trim(stf), s_fm, st1)
                write (kout, "(a, i3, a, i3, a, a)") '      Using ', j, ' bits, ', k,  &
                                                     ' terms gave   s_fm = ', trim(st1)
                call fm_ulp(s_fm, t_fm)

!                  Since s2_fm was computed at a different precision than s_fm, we should
!                  round it to the current precision.  For this case we have explicitly set
!                  ndig instead of using fm_set as in case 2 above, so we use fm_equ to do
!                  the rounding.

                call fm_equ(s2_fm, x2_fm, 150, j)
                error_fm = abs( (s_fm-x2_fm)/t_fm )
                digits_lost_fm = nint(log10(error_fm))
                write (kout, "(a, i3, a)") '                This calculation lost about ',  &
                                           to_int(digits_lost_fm), ' decimal digits.'
                exit
            endif
            x_fm = x_fm * x2_fm
            fact_fm = fact_fm * (k+1)
         enddo
      enddo
      if (abs(s_fm-to_fm('.043990942179625639968630235187619 6')) > 1.0d-20) then
          nerror = nerror + 1
          write (kout,*) ' '
          write (kout,*) ' Error in case 3.'
          write (kout,*) ' '
      endif

!             4.  A second way to check an algorithm's stability is to re-do the calculation
!                 at the same precision but with different rounding modes.

!                 Use base 2 with 53 bits and round down, then round symmetrically, then round up.
```

```
!                The results show the three values have no digits of agreement, confirming the
!                loss of about 15 or 16 s.d. in the ones rounded down and up.

      write (kout,*) ' '
      write (kout,*) ' 4.  Use FM with different rounding modes in base 2.'
      write (kout,*) ' '
      call fm_setvar(" mbase = 2 ")
      call fm_setvar(" ndig = 53 ")

      do j = 1, 3
         if (j == 1) then
             call fm_setvar(" kround = -1 ")
         else if (j == 2) then
             call fm_setvar(" kround = 1 ")
         else if (j == 3) then
             call fm_setvar(" kround = 2 ")
         endif
         s_fm = 0
         x_fm = 35.0d0/2
         x2_fm = -(x_fm**2)
         fact_fm = 1
         do k = 0, 700
            t_fm = x_fm / ( (k+1) * fact_fm**2 )
            s_fm = s_fm + t_fm
            if ( abs(t_fm) < epsilon(s_fm)*abs(s_fm) ) then
                if (j == 1) then
                    stf = 'rounding left         , '
                else if (j == 2) then
                    stf = 'rounding symmetrically, '
                else if (j == 3) then
                    stf = 'rounding right        , '
                endif
                call fm_form(' f20.17 ', s_fm, st1)
                write (kout, "(a, i4, a, a, i3, a, a)") '      Using', 53, ' bits, ', trim(stf),  &
                                                        k, ' terms gave   s_fm = ', trim(st1)
                exit
            endif
            x_fm = x_fm * x2_fm
            fact_fm = fact_fm * (k+1)
         enddo
         r_fm(j) = s_fm
      enddo
      error_fm = max( abs((r_fm(1)-r_fm(2))/r_fm(1)) , abs((r_fm(1)-r_fm(3))/r_fm(1)) ,  &
                      abs((r_fm(2)-r_fm(3))/r_fm(2)) )
      j = -nint(log10(error_fm))
      write (kout, "(a, i3, a)") '      These agree to about', j, ' decimal digits.'
      if (abs(s_fm-to_fm('.0519420065663800')) > 1.0d-4) then
          nerror = nerror + 1
          write (kout,*) ' '
          write (kout,*) ' Error in case 4.'
          write (kout,*) ' '
      endif

!                Use base 2 with 113 bits and round down, then round symmetrically, then round up.

!                Now the three values agree to about 18 decimal digits, which is again consistent
!                with a loss of about 15 or 16 s.d. in the ones rounded down and up.
```

```
      write (kout,*) ' '
      call fm_setvar(" mbase = 2 ")
      call fm_setvar(" ndig = 113 ")

      do j = 1, 3
         if (j == 1) then
             call fm_setvar(" kround = -1 ")
         else if (j == 2) then
             call fm_setvar(" kround = 1 ")
         else if (j == 3) then
             call fm_setvar(" kround = 2 ")
         endif
         s_fm = 0
         x_fm = 35.0d0/2
         x2_fm = -(x_fm**2)
         fact_fm = 1
         do k = 0, 700
            t_fm = x_fm / ( (k+1) * fact_fm**2 )
            s_fm = s_fm + t_fm
            if ( abs(t_fm) < epsilon(s_fm)*abs(s_fm) ) then
                if (j == 1) then
                    stf = 'rounding left         , '
                else if (j == 2) then
                    stf = 'rounding symmetrically, '
                else if (j == 3) then
                    stf = 'rounding right        , '
                endif
                call fm_form(' f20.17 ', s_fm, st1)
                write (kout, "(a, i4, a, a, i3, a, a)") '      Using', 113, ' bits, ', trim(stf),  &
                                                        k,' terms gave   s_fm = ', trim(st1)
                exit
            endif
            x_fm = x_fm * x2_fm
            fact_fm = fact_fm * (k+1)
         enddo
         r_fm(j) = s_fm
      enddo
      error_fm = max( abs((r_fm(1)-r_fm(2))/r_fm(1)) , abs((r_fm(1)-r_fm(3))/r_fm(1)) ,  &
                      abs((r_fm(2)-r_fm(3))/r_fm(2)) )
      j = -nint(log10(error_fm))
      write (kout, "(a, i3, a)") '      These agree to about', j, ' decimal digits.'
      if (abs(s_fm-to_fm('.0439909421796257')) > 1.0d-4) then
          nerror = nerror + 1
          write (kout,*) ' '
          write (kout,*) ' Error in case 4.'
          write (kout,*) ' '
      endif


!             5.  A third way to check an algorithm's stability is to re-do the calculation
!                 at the same precision but using interval arithmetic.

!                 Use base 2 with 53 bits.

!                 The result shows the two endpoints of the interval having opposite signs
!                 indicating a worst-case loss of all 16 s.d. during the calculation.

      write (kout,*) ' '
```

```
      write (kout,*) ' 5.  Use FM with interval arithmetic in base 2.'
      write (kout,*) ' '
      call fm_setvar(" mbase = 2 ")
      call fm_setvar(" ndig = 53 ")

      s_fm_interval = 0
      x_fm_interval = 35.0d0/2
      x2_fm_interval = -(x_fm_interval**2)
      fact_fm_interval = 1
      do k = 0, 700
         t_fm_interval = x_fm_interval / ( (k+1) * fact_fm_interval**2 )
         s_fm_interval = s_fm_interval + t_fm_interval
         if ( abs(t_fm_interval) < epsilon(s_fm_interval)*abs(s_fm_interval) ) then
             call fm_interval_form(' f20.16 ', s_fm_interval, st1)
             write (kout, "(a, i4, a, i3, a, a)") '      Using', 53, ' bits, ', k,  &
                                                  ' terms gave  s_fm_interval = ', trim(st1)
             exit
         endif
         x_fm_interval = x_fm_interval * x2_fm_interval
         fact_fm_interval = fact_fm_interval * (k+1)
      enddo
      error_fm = abs((right_endpoint(s_fm_interval)-left_endpoint(s_fm_interval)) /  &
                     right_endpoint(s_fm_interval))
      j = -nint(log10(error_fm))
      write (kout, "(a, i3, a)") '      The two endpoints agree to about', j, ' decimal digits.'

!                Use base 2 with 113 bits.

!                The result shows the two endpoints of the interval agree to about 18 s.d.
!                indicating a worst-case loss of about 16 s.d. during the calculation.

      write (kout,*) ' '
      call fm_setvar(" mbase = 2 ")
      call fm_setvar(" ndig = 113 ")

      s_fm_interval = 0
      x_fm_interval = 35.0d0/2
      x2_fm_interval = -(x_fm_interval**2)
      fact_fm_interval = 1
      do k = 0, 700
         t_fm_interval = x_fm_interval / ( (k+1) * fact_fm_interval**2 )
         s_fm_interval = s_fm_interval + t_fm_interval
         if ( abs(t_fm_interval) < epsilon(s_fm_interval)*abs(s_fm_interval) ) then
             call fm_interval_form(' f20.16 ', s_fm_interval, st1)
             write (kout, "(a, i4, a, i3, a, a)") '      Using', 113, ' bits, ', k,  &
                                                  ' terms gave  s_fm_interval = ', trim(st1)
             exit
         endif
         x_fm_interval = x_fm_interval * x2_fm_interval
         fact_fm_interval = fact_fm_interval * (k+1)
      enddo
      error_fm = abs((right_endpoint(s_fm_interval)-left_endpoint(s_fm_interval)) /  &
                     right_endpoint(s_fm_interval))
      j = -nint(log10(error_fm))
      write (kout, "(a, i3, a)") '      The two endpoints agree to about', j, ' decimal digits.'
      if (abs(s_fm_interval-to_fm('.0439909421796257')) > 1.0d-4) then
          nerror = nerror + 1
          write (kout,*) ' '
```

```
        write (kout,*) ' Error in case 5.'
        write (kout,*) ' '
      endif


      if (nerror == 0) then
        write (kout, "(//a)") ' Summary:'
        write (kout, "(a)") ' '
        write (kout, "(a)") ' For this calculation all four methods for measuring the degree of'
        write (kout, "(a)") ' instability worked well.  When done with double precision carrying'
        write (kout, "(a)") ' 16 significant digits and using the default symmetric rounding,'
        write (kout, "(a)") ' only 1 digit remained correct at the end of the sum.'
        write (kout, "(a)") ' '
        write (kout, "(a)") ' Interval arithmetic is probably the strongest of these checks, and'
        write (kout, "(a)") ' using FM arithmetic with 30 digits and a large base (method 2) is'
        write (kout, "(a)") ' the fastest of these methods for getting the sum correct to full'
        write (kout, "(a)") ' double precision accuracy.'
        write (kout, "(a)") ' '
        write (kout, "(a)") ' Comparing the last value of s in method 1 with the first value of'
        write (kout, "(a)") ' s_fm in method 3 should show whether this compiler carries extra'
        write (kout, "(a)") ' digits while evaluating expressions like x / ( (k+1) * fact**2 ).'
        write (kout, "(a)") ' If the two values are the same, no extra digits are carried in d.p.'
        write (kout, "(a)") ' '
        write (kout, "(a)") ' '
      endif

!  ------------------------------------------------------------------------------Sample 2

!             Here is a recurrence that is seriously unstable.

!             It is not a realistic problem that would come up in a practical application, but
!             is designed as a counter-example to show that just carrying much higher precision
!             cannot be proved to always cure numerical instability.
!             Reference:  William Kahan -- (2006)
!                "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?"
!                 http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

!             After 60 steps the result without any rounding errors would be very close to 5,
!             but rounding errors cause the result to converge to 100 instead.

      call fm_set(30)
      write (kout,*) ' '
      write (kout,*) ' '
      write (kout,*) ' Sample 2.  Unstable recurrence.'
      write (kout,*) ' '
      write (kout, "(a)") '      1.  Use non-interval FM arithmetic with 30 digits.'
      write (kout,*) ' '
      x_fm = 4
      x2_fm = to_fm('4.25')
      do j = 1, 60
         t_fm = 108 - ( 815 - 1500/x_fm ) / x2_fm
         x_fm = x2_fm
         x2_fm = t_fm
         if (mod(j, 5) == 0) then
             call fm_form(' f20.14 ', x2_fm, st1)
             write (kout, "(a, i2, a, a)") '          After ', j,  &
                                        ' terms with 30 digit accuracy, the result is', trim(st1)
         endif
```

```
      enddo
      write (kout,*) ' '

      write (kout, "(a)") '      2.  Use interval arithmetic with 30 digit accuracy.'
      write (kout,*) ' '
      x_fm_interval = 4
      x2_fm_interval = to_fm_interval('4.25')
      do j = 1, 60
         t_fm_interval = 108 - ( 815 - 1500/x_fm_interval ) / x2_fm_interval
         x_fm_interval = x2_fm_interval
         x2_fm_interval = t_fm_interval
         call fm_interval_form(' f20.16 ', x2_fm_interval, st1)
         write (kout, "(a, i3, a, a)") '          After', j, ' terms the result is', trim(st1)
         if (left_endpoint(x_fm_interval) <= 0 .and. right_endpoint(x_fm_interval) >= 0) exit
         if (j > 10 .and. j < 25) then
             if (abs(left_endpoint(x2_fm_interval)-5) > 0.01 .or.  &
                 abs(right_endpoint(x2_fm_interval)-5) > 0.01) then
                 nerror = nerror + 1
                 write (kout,*) ' '
                 write (kout,*) ' Error in sample 2, case 2.'
                 write (kout,*) ' '
                 exit
             endif
         endif
      enddo

      if (nerror == 0) then
          write (kout, "(//a)") ' Summary:'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' The general solution of this recurrence has a term that causes'
          write (kout, "(a)") ' the values to converge to 100, but for these initial conditions'
          write (kout, "(a)") ' 4, 4.25, the specific solution has a coefficient of zero on that'
          write (kout, "(a)") ' term, so this sequence converges to 5 mathematically.'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' When rounding errors occur in later x-values, they introduce a'
          write (kout, "(a)") ' very small but nonzero amount of the term that causes convergence'
          write (kout, "(a)") ' to 100, and that term grows rapidly and soon swamps the rest of'
          write (kout, "(a)") ' the solution.'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' Interval arithmetic tracks the growing uncertainty in the x-values,'
          write (kout, "(a)") ' and when an interval gets big enough to include zero, dividing by'
          write (kout, "(a)") ' that interval is undefined and the result is'
          write (kout, "(a)") ' [ -overflow , +overflow ].'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' Interval arithmetic works better than a sequence of increasing'
          write (kout, "(a)") ' precision FM results here, since comparing FM results at 30, 40, 50'
          write (kout, "(a)") ' digits gives 100 each time.'
          write (kout, "(a)") ' '
      endif

!  ------------------------------------------------------------------------------------Sample 3

!             Sum an unstable series.
!             exp(x) = 1 + x + x**2/2! + x**3/3! + ...
!             converges mathematically for all x, but is unstable for negative x.

      s_fm_interval = 1
      t_fm_interval = 1
```

```
      r_fm = (/ -25, -30, -35 /)
      exp_sum_interval = exp_sum3(r_fm)
      e = (/ -12, -9, -3 /)
      write (kout,*) ' '
      write (kout,*) ' '
      write (kout,*) ' '
      write (kout,*) ' Sample 3.  Unstable sum.'
      do j = 1, 3
         call fm_interval_form(' es25.16 ', exp_sum_interval(j), st1)
         call fm_form(' es25.16 ', exp(r_fm(j)), stf)
         write (kout, "(/a, f6.2, a, a)") '      For x = ', to_dp(r_fm(j)), ' The sum gave ', st1
         write (kout, "(20x, a, a)") '      correct = ', stf

!             Check the results.

         x_fm  = left_endpoint(exp_sum_interval(j))
         x2_fm = right_endpoint(exp_sum_interval(j))
         s_fm = exp(r_fm(j))
         t_fm = abs( (x2_fm - x_fm ) / s_fm )
         if (.not.(s_fm > x_fm) .or. .not.(s_fm < x2_fm) .or. .not.(t_fm < to_fm(10)**e(j))) then
             nerror = nerror + 1
             exit
         endif
      enddo

      if (nerror == 0) then
          write (kout, "(//a)") ' Summary:'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' As the input x becomes more negative, the instability increases.'
          write (kout, "(a)") ' This is shown as the left and right endpoints of the interval'
          write (kout, "(a)") ' result agree to fewer digits.'
          write (kout, "(a)") ' '
          write (kout, "(a)") ' The mathematically correct value of the sum lies within the'
          write (kout, "(a)") ' interval in each case.'
          write (kout, "(a)") ' '
      endif



      if (nerror == 0) then
          write (*   , "(//a, a/)") ' All results were ok.  (The output is in file ',  &
                                    'SampleFMinterval.out)'
          write (kout, "(//a/)") ' All results were ok.'
      else
          write (*   , "(//i3, a, a/)") nerror, ' error(s) found.  (The output is in file ',  &
                                           'SampleFMinterval.out)'
          write (kout, "(//i3, a/)") nerror, ' error(s) found.'
      endif

      close(kout)
      stop
      end program test

      subroutine round_fm(a, b, j1, j2)
      use fmvals
      use fmzm
      implicit none
```

```
!  a was computed with precision defined by call fm_set(j1).
!  b will be returned with the value of a rounded to precision defined by call fm_set(j2).
!  Do not use b in the calling program at any precision higher than j2.

      type (fm) :: a, b
      integer :: j1, j2, ndig1, ndig2, nsave
      intent (in) :: a, j1, j2
      intent (inout) :: b

      nsave = ndig
      call fm_set(j1)
      ndig1 = ndig
      call fm_set(j2)
      ndig2 = ndig
      call fm_equ(a, b, ndig1, ndig2)
      ndig = nsave

      end subroutine round_fm
```