```
      program test

!   Version 1.4.

!   This is a sample program using the fmzm modules for doing arithmetic using the FM, IM, and ZM
!   derived types.

!   The program's output to the screen is also saved in file SampleFM.out.
!   The program checks all the results and the last line of the output file should be
!   "All results were ok."

!   These examples show various ways to use the FM package, but the methods used are not always
!   the most advanced for the sample problem.

      use fmzm

      implicit none

!             Declare the multiple precision variables.  The three types are:
!             (fm) for multiple precision real
!             (im) for multiple precision integer
!             (zm) for multiple precision complex

      type (fm), save :: x1, x2, x3, x4
      type (fm), save, allocatable :: a(:,:), b(:,:), v1(:), v2(:)
      type (im), save :: i1, i2, i3
      type (zm), save :: z1, z2, z3, z4

!             Declare the function name of a type (fm) function that will be passed as an argument
!             to a subroutine called from this program.

      type (fm), external :: f

!             Declare the other variables (not multiple precision).

      character(80)  :: st1
      character(175) :: fmt
      integer :: iter, j, k, kout, nerror
      integer :: seed(7)
      double precision :: value

!             Write output to the screen (unit *), and also to the file SampleFM.out.

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

      nerror = 0


!             1.  Find a root of the equation  f(x) = x**5 - 3x**4 + x**3 - 4x**2 + x - 6 = 0.

!                 Set precision to give at least 60 significant digits.

      call fm_set(60)

!                 Use Newton's method with initial guess x = 3.12.
!                 Horner's rule is used to evaluate the function.
```

```
!                x1 is the previous iterate.
!                x2 is the current iterate.

!                to_fm is a function for converting other types of numbers to type FM.  Note that
!                to_fm(3.12) converts the real constant to fm, but it is accurate only to single
!                precision, since the number 3.12 cannot be represented exactly in binary and has
!                already been rounded to single precision.  Similarly, to_fm(3.12d0) agrees with
!                3.12 to double precision accuracy, and to_fm('3.12') or to_fm(312)/to_fm(100)
!                agrees to full FM accuracy.
!                Here, to_fm(3.12) would be ok, since Newton iteration will correct the error
!                coming from single precision, but it is a good habit to use the more accurate
!                form.

      x1 = to_fm('3.12')

!                Print the first iteration.

      fmt = "(//' Sample 1.  Real root of f(x) = x**5 - 3x**4 + x**3 - 4x**2 + x - 6 = 0.'//" //  &
            "' Iteration       Newton approximation')"
      write (*   , fmt)
      write (kout, fmt)

!                fm_form is a formatting subroutine.

      call fm_form('f65.60', x1, st1)
      write (*   , "(/i10, 4x, a)") 0, trim(st1)
      write (kout, "(/i10, 4x, a)") 0, trim(st1)

      do iter = 1, 10

!                x3 is f(x1).

         x3 = ((((x1-3)*x1+1)*x1-4)*x1+1)*x1-6

!                x4 is f'(x1).

         x4 = (((5*x1-12)*x1+3)*x1-8)*x1+1

         x2 = x1 - x3/x4

!                Print each iteration.

         call fm_form('f65.60', x2, st1)
         write (*   , "(/i10, 4x, a)") iter, trim(st1)
         write (kout, "(/i10, 4x, a)") iter, trim(st1)

!                Stop iterating if x1 and x2 agree to over 60 places.

         x4 = abs(x1-x2)
         if (x4 < 1.0d-61) exit

!                Set x1 = x2 for the next iteration.

         x1 = x2
      enddo

!                Check the answer.
```

```
      x3 = to_fm('3.120656215326726500470956013523797484654623935599066014988828435 8')

!                 It is slightly safer to do this test with .not. instead of
!                       if (abs(x3-x2) >= 1.0d-61) then
!                 because if the result of abs(x3-x2) is fm's unknown value,
!                 the comparison returns false for all comparisons.

      if (.not.(abs(x3-x2) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 1.'/)")
          write (kout, "(/' Error in sample case number 1.'/)")
      endif


!             2.  Higher Precision.  Compute the root above to 500 decimal places.

      call fm_set(500)

!                 It is tempting to just say x1 = x3 here to initialize the start of the higher
!                 precision iterations to be the check value defined above.  That will not work,
!                 because precision has changed.  Most of the digits of x3 may be undefined at
!                 the new precision.
!                 The most flexible way to pad a lower precision value with zeros when raising
!                 precision is to use subroutine fm_equ, but here it is easier to re-define x1
!                 from scratch at the new precision.

      x1 = to_fm('3.120656215326726500470956013523797484654623935599066014988828435 8')

      do iter = 1, 10

!                 x3 is f(x1).

         x3 = ((((x1-3)*x1+1)*x1-4)*x1+1)*x1-6

!                 x4 is f'(x1).

         x4 = (((5*x1-12)*x1+3)*x1-8)*x1+1

         x2 = x1 - x3/x4

!                 Stop iterating if x1 and x2 agree to over 500 places.

         x4 = abs(x1-x2)

!                 Compare this test to the similar one in case 1 above.
!                 For machines with 64-bit double precision, 1.0d-501 would be smaller than the
!                 smallest positive number.  So this error tolerance is converted to an FM number
!                 from character form.

         if (x4 < to_fm('1.0e-501')) exit

!                 Set x1 = x2 for the next iteration.

         x1 = x2
      enddo

!                 For very high precision output, it is sometimes more convenient to use fm_print
```

```
!                to format and print the numbers, since the line breaks are handled automatically.
!                The unit number for the output, kw, and the format codes to be used, jform1 and
!                jform2, are internal FM variables.
!                Subroutine fm_setvar is used to re-define these, and the new values will remain in
!                effect for any further calls to fm_print.

!                Other variables that can be changed and the options they control are listed in the
!                documentation at the top of file FM.f95.

!                Set the fm_print format to f505.500

      call fm_setvar(' jform1 = 2 ')
      call fm_setvar(' jform2 = 500 ')

!                Set the output screen width to 90 columns.

      call fm_setvar(' kswide = 90 ')

      fmt = "(///' Sample 2.  Find the root above to 500 decimal places.'/)"
      write (*   , fmt)
      write (kout, fmt)

!                Write to the output file.

      call fm_setvar(' kw = 18 ')
      call fm_print(x2)

!                Write to the screen (unit 6).

      call fm_setvar(' kw = 6 ')
      call fm_print(x2)

!                Check the answer.

      x3 = to_fm('3.120656215326726500470956013523797484654623935599066014988828435819026499 9'  // &
                 '5179546897832574500171510958119234313326828394200408405359545601181522453719'// &
                 '7928813052719510171189388982124036620583073039835473769132820001100582735049'// &
                 '2028386707098956192754134845215492825918911569452007894158183875295120109999'// &
                 '6021551313210767970990266642369928037034625701495597243893923318275975524609'// &
                 '6106122004855795291569104281155470137877144237085781610256415550974811799699'// &
                 '1750283901059047868316801283843311432593091555771716838424443527684191761390609')

      if (.not.(abs(x3-x2) < to_fm('1.0e-501'))) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 2.'/)")
          write (kout, "(/' Error in sample case number 2.'/)")
      endif


!             3.  Compute the Riemann zeta function for s=3.

!                Use Gosper's formula:  zeta(3) =
!                (5/4)*Sum[ (-1)**k * (k!)**2 / ((k+1)**2 * (2k+1)!) ]
!                while k = 0, 1, ....

!                x1 is the current partial sum.
!                x2 is the current term.
!                x3 is k!
```

```
!                x4 is (2k+1)!

      call fm_set(60)
      x1 = 1
      x3 = 1
      x4 = 1
      do k = 1, 200
         x3 = k*x3
         j = 2*k*(2*k+1)
         x4 = j*x4
         x2 = x3**2
         j = (k+1)*(k+1)
         x2 = (x2/j)/x4
         if (mod(k, 2) == 0) then
             x1 = x1 + x2
         else
             x1 = x1 - x2
         endif

!                Test for convergence.
!                Here the rate of convergence is much slower than in the Newton iterations above.
!                Asking for 60 digits in the call to fm_set will cause the internal precision to
!                be set slightly higher than that, giving the user a few guard digits.
!                x2 is the difference between the two most recent partial sums, so the test
!                below will stop the sum when the last two partial sums agree to at least 65
!                significant digits.

         if (abs(x2/x1) < 1.0d-65) then
             fmt = "(///' Sample 3.', 2x, i5, ' terms were added in the zeta sum.'/)"
             write (*   , fmt) k
             write (kout, fmt) k
             exit
         endif
      enddo

!                Print the result.

      x1 = (5*x1)/4
      call fm_form('f63.60', x1, st1)
      write (*   , "(' zeta(3) = ', a)") trim(st1)
      write (kout, "(' zeta(3) = ', a)") trim(st1)

!                Check the answer.

      x3 = to_fm('1.202056903159594285399738161511449990764986292340498881792271555')
      if (.not.(abs(x1-x3) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 3.'/)")
          write (kout, "(/' Error in sample case number 3.'/)")
      endif


!             4.  Integer multiple precision calculations.

!                Fermat's theorem says  x**(p-1) mod p = 1 when p is prime and x is not a
!                multiple of p.
!                If  x**(p-1) mod p  gives 1 for some p with several different x's, then it is
!                very likely that p is prime (but it is not certain until further tests are done).
```

```
!                Find a 70-digit number p that is "probably" prime.

!                Use fm_random_number to generate a random 70-digit starting value and search for
!                a prime from that point.

!                Initialize the generator.

      seed = (/ 2718281, 8284590, 4523536, 0287471, 3526624, 9775724, 7093698 /)
      call fm_random_seed_put(seed)

!                i1 is the value p being tested.

      i1 = 0
      i3 = to_im(10)**13
      do j = 1, 6
         call fm_random_number(value)
         i2 = 1.0d13*value
         i1 = i1*i3 + i2
      enddo
      i3 = to_im(10)**70
      i1 = mod(i1, i3)

!                To speed up the search, test only values that are not
!                multiples of 2, 3, 5, 7, 11, 13.

      k = 2*3*5*7*11*13
      i1 = (i1/k)*k + k + 1
      i3 = 3

      do j = 1, 100
         i2 = i1 - 1

!                Compute 3**(p-1) mod p

         i3 = power_mod(i3, i2, i1)
         if (i3 == 1) then

!                Check that 7**(p-1) mod p is also 1.

            i3 = 7
            i3 = power_mod(i3, i2, i1)
            if (i3 == 1) then
                fmt = "(///' Sample 4.', 2x, i5," //  &
                      " ' values were checked before finding a prime p.'/)"
                write (*   , fmt) j
                write (kout, fmt) j
                exit
            endif
         endif

         i3 = 3
         i1 = i1 + k
      enddo

!                Print the result.

      call im_form('i72', i1, st1)
```

```
      write (*   , "(' p =', a)") trim(st1)
      write (kout, "(' p =', a)") trim(st1)

!                 Check the answer.

      i3 = to_im('955213112905605831310353635773880488484082549850308894676076841949059l')
      if (.not.(i1 == i3)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 4.'/)")
          write (kout, "(/' Error in sample case number 4.'/)")
      endif


!             5.  Log Integral function.

!                 Estimate the number of primes less than 10**30.

      fmt = "(///' Sample 5.  Log integral.  Estimate the number of primes less than 10**30.'/" // &
               "'             It should be accurate to about 15 significant digits.'/)"
      write (*   , fmt)
      write (kout, fmt)

      i2 = to_im(log_integral(to_fm('1.0e+30')))

!                 Print the result.

      call im_form('i30', i2, st1)
      write (*   , "(' int(li(1.0e+30)) = ', a)") trim(st1)
      write (kout, "(' int(li(1.0e+30)) = ', a)") trim(st1)

!                 Check the answer.

      i3 = to_im('1469239889772044763907908766 9')
      if (.not.(i2 == i3)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 5.'/)")
          write (kout, "(/' Error in sample case number 5.'/)")
      endif


!             6.  Gamma function.

!                 Check that gamma(1/2) is sqrt(pi)

      fmt = "(///' Sample 6.  Check that gamma(1/2) = sqrt(pi).'/)"
      write (*   , fmt)
      write (kout, fmt)

      x2 = gamma(to_fm('0.5'))

!                 Print the result.

      call fm_form('f63.60', x2, st1)
      write (*   , "(' gamma(1/2) = ', a)") trim(st1)
      write (kout, "(' gamma(1/2) = ', a)") trim(st1)

!                 Check the answer.
```

```
      x3 = sqrt(acos(to_fm(-1)))
      if (.not.(abs(x3-x2) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 6.'/)")
          write (kout, "(/' Error in sample case number 6.'/)")
      endif


!             7.  Psi and polygamma functions.

!                 Rational series can often be summed using these functions.
!                 Sum (n=1 to infinity) 1/(n**2 * (8n+1)**2) =
!                 16*(psi(1) - psi(9/8)) + polygamma(1, 1) + polygamma(1, 9/8)
!                 Reference: Abramowitz & Stegun, Handbook of Mathematical Functions,
!                 chapter 6, Example 10.

      fmt = "(///' Sample 7.  Psi and polygamma functions.'/)"
      write (*   , fmt)
      write (kout, fmt)

      x2 = 16*(psi(to_fm(1)) - psi(to_fm(9)/8)) + polygamma(1, to_fm(1)) + polygamma(1, to_fm(9)/8)

!                 Print the result.

      call fm_form('f65.60', x2, st1)
      fmt = "(' Sum (n=1 to infinity) 1/(n**2 * (8n+1)**2) = '/9x, a)"
      write (*   , fmt) trim(st1)
      write (kout, fmt) trim(st1)

!                 Check the answer.

      x3 = to_fm('1.34994861454130247551078291050351479506449786358372708163273 96m-2')
      if (.not.(abs(x3-x2) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 7.'/)")
          write (kout, "(/' Error in sample case number 7.'/)")
      endif


!             8.  Incomplete gamma and gamma functions.

!                 Find the probability that an observed chi-square for a correct model should be
!                 less that 2.3 when the number of degrees of freedom is 5.
!                 Reference: Knuth, Volume 2, 3rd ed., Page 56, and Press, Flannery, Teukolsky,
!                 Vetterling, Numerical Recipes, 1st ed., Page 165.

      fmt = "(///' Sample 8.  Incomplete gamma and gamma functions.'/)"
      write (*   , fmt)
      write (kout, fmt)

      x1 = to_fm(5)/2
      x2 = incomplete_gamma1(x1, to_fm('2.3')/2) / gamma(x1)

!                 Print the result.

      call fm_form('f62.60', x2, st1)
      write (*   , "(' Probability = ', a)") trim(st1)
      write (kout, "(' Probability = ', a)") trim(st1)
```

```
!                Check the answer.

      x3 = to_fm('0.193733130114871446327510259182505999534723186071213869730662837 39')
      if (.not.(abs(x3-x2) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 8.'/)")
          write (kout, "(/' Error in sample case number 8.'/)")
      endif


!             9.  Error function.

!                Find the probability that a value drawn from a normal distribution is within
!                1 or 2 or 3 standard deviations from the mean.

      fmt = "(///' Sample 9.  Error function.  Probability that a value drawn from a normal'/" //  &
                "'             distribution is within k standard deviations from the mean.'/)"
      write (*   , fmt)
      write (kout, fmt)

      do k = 1, 3
         x1 = k / sqrt(to_fm(2))
         x2 = erf(x1)

!                Print the results.

         call fm_form('f52.50', x2, st1)
         write (*   , "(' k = ', i2, ',  probability = ', a)") k, trim(st1)
         write (kout, "(' k = ', i2, ',  probability = ', a)") k, trim(st1)

!                Check the answer.

         if (k == 1) then
             x3 = to_fm('0.68268949213708589717046509126407584495582593345320878197478890049')
         else if (k == 2) then
             x3 = to_fm('0.95449973610364158559943472566693312505644755259664313203266799974')
         else
             x3 = to_fm('0.99730020393673981094669637046481004524434126368323870127155602929')
         endif
         if (.not.(abs(x3-x2) < 1.0d-61)) then
             nerror = nerror + 1
             write (*   , "(/' Error in sample case number 9.'/)")
             write (kout, "(/' Error in sample case number 9.'/)")
         endif
      enddo


!             10. Array operations.

!                Find the dominant eigenvalue and a corresponding eigenvector for this 5x5 matrix:

!                        3  1  4  1  5
!                        9  2  6  5  3
!                   a =  5  8  9  7  9
!                        3  2  3  8  4
!                        6  2  6  4  3
```

```
!                Use the power method.  Compute b = a**n.  If v1 is an initial guess for the
!                largest magnitude eigenvector, v2 = b*v1 should be a more accurate approximation.
!                The ratio of the elements of v3 = a*v2 to those of v2 gives an estimate of the
!                corresponding eigenvalue.  By repeatedly squaring the matrix, each iteration uses
!                the next higher power of 2 for n.

      fmt = "(///' Sample 10.  Eigenvalue from matrix powers.')"
      write (*   , fmt)
      write (kout, fmt)

!             These type FM arrays were declared as allocatable.  Allocate them now, and initialize.

      allocate( a(5, 5) )
      allocate( b(5, 5) )
      allocate( v1(5) )
      allocate( v2(5) )

!             To initialize the matrix, we can use array sections to set one row at a time, and the
!             fmzm interface will take care of converting from integer to type (fm).  If the values
!             were not integers, we could say  a(1, 1:5) = (/ to_fm(' 3.7 '), to_fm(' 4.2 '), etc.

      a(1, 1:5) = (/ 3, 1, 4, 1, 5 /)
      a(2, 1:5) = (/ 9, 2, 6, 5, 3 /)
      a(3, 1:5) = (/ 5, 8, 9, 7, 9 /)
      a(4, 1:5) = (/ 3, 2, 3, 8, 4 /)
      a(5, 1:5) = (/ 6, 2, 6, 4, 3 /)

!             Initialize all elements of the initial guess vector to 1.

      v1 = 1

      b = a
      write (*   , "(/' Iteration    eigenvalue approximation ')")
      write (kout, "(/' Iteration    eigenvalue approximation ')")
      do j = 1, 7
         b = matmul(b, b)
         v1 = matmul(b, v1)
         v2 = matmul(a, v1)
         x1 = v2(1) / v1(1)
         call fm_form('f64.57', x1, st1)
         write (*   , "(/i10, a)") j, trim(st1)
         write (kout, "(/i10, a)") j, trim(st1)
      enddo

!             Normalize the eigenvector (l-2 norm).

      v2 = v2 / norm2(v2)
      write (*   , "(/' The corresponding eigenvector is'/)")
      write (kout, "(/' The corresponding eigenvector is'/)")
      do j = 1, 5
         call fm_form('f61.57', v2(j), st1)
         write (*   , "(a)") trim(st1)
         write (kout, "(a)") trim(st1)
      enddo

!                Check the answer.

      x3 = to_fm('23.912767172321328589357039228003304505549129195999272982168272478032 04')
```

```
      if (.not.(abs(x3-x1) < 1.0d-61)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 10.'/)")
          write (kout, "(/' Error in sample case number 10.'/)")
      endif


!             11. Function and subroutine example.

!                 Find the integral from 0 to 1/2 of 2*exp(-x**2)/sqrt(pi).

!                 The exact value of the integral is erf(1/2).
!                 Use a simple numerical integration routine to apply an integration rule
!                 using 100 intervals.

      call fm_set(40)

      fmt = "(///' Sample 11.  Function and subroutine example.'/)"
      write (*   , fmt)
      write (kout, fmt)

      x1 = 0
      x2 = to_fm(' 0.5 ')
      call plan_9(f, x1, x2, 100, x3)

!                 Print the result.

      call fm_form('f32.30', x3, st1)
      write (*   , "(' Integral = ', a)") trim(st1)
      write (kout, "(' Integral = ', a)") trim(st1)

!                 Check the answer.

      x4 = erf(to_fm('0.5'))
      if (.not.(abs(x3-x4) < 1.0d-31)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 11.'/)")
          write (kout, "(/' Error in sample case number 11.'/)")
      endif


!             Complex arithmetic.

!             Set precision to give at least 30 significant digits.

      call fm_set(30)


!             12. Find a complex root of the equation
!                 f(x) = x**5 - 3x**4 + x**3 - 4x**2 + x - 6 = 0.

!                 Newton's method with initial guess x = .56 + 1.06 i.

!                 z1 is the previous iterate.
!                 z2 is the current iterate.

      z1 = to_zm('.56 + 1.06 i')
```

```
!                Print the first iteration.

      fmt = "(///' Sample 12.  Complex root of f(x) = x**5 - 3x**4 + x**3 - 4x**2 + x - 6 = 0.',"  &
            //"//' Iteration       Newton approximation')"
      write (*   , fmt)
      write (kout, fmt)
      call zm_form('f32.30', 'f32.30', z1, st1)
      write (*   , "(/i6, 4x, a)") 0, trim(st1)
      write (kout, "(/i6, 4x, a)") 0, trim(st1)

      do iter = 1, 10

!                z3 is f(z1).

         z3 = ((((z1-3)*z1+1)*z1-4)*z1+1)*z1-6

!                z4 is f'(z1).

         z4 = (((5*z1-12)*z1+3)*z1-8)*z1+1

         z2 = z1 - z3/z4

!                Print each iteration.

         call zm_form('f32.30', 'f32.30', z2, st1)
         write (*   , "(/i6, 4x, a)") iter, trim(st1)
         write (kout, "(/i6, 4x, a)") iter, trim(st1)

!                Stop iterating if z1 and z2 agree to over 30 places.

         if (abs(z1-z2) < 1.0d-31) exit

!                Set z1 = z2 for the next iteration.

         z1 = z2
      enddo

!                Check the answer.

      z3 = to_zm('0.561958308335403235498111195347453 + 1.061134679604332556983391239058885 i')
      if (.not.(abs(z3-z2) < 1.0d-31)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 12.'/)")
          write (kout, "(/' Error in sample case number 12.'/)")
      endif


!             13. Compute exp(1.23-2.34i).

!                Use the direct Taylor series.

!                z1 is x.
!                z2 is the current term, x**n/n!.
!                z3 is the current partial sum.

      z1 = to_zm('1.23-2.34i')
      z2 = 1
      z3 = 1
```

```
      do k = 1, 100
         z2 = z2*z1/k
         z4 = z3 + z2

!                Test for convergence.

!                This is a common way to check for series convergence -- wait until the term
!                being added is so close to zero that the sum does not change.  That is fine
!                here, because we are using the default round-to-nearest rounding mode.

!                There is a pitfall if we were to re-run the program with a different rounding
!                mode.  For example, if we change the rounding mode to round toward +infinity,
!                then at 30-digit precision the addition  1.2 + 3.4e-100  rounds up to  1.200...001
!                and so the test to see if the sum did not change might never be satisfied.
!                This problem can occur with either type FM or ZM sums.

!                For cases where other rounding modes might be used, doing the convergence check
!                like we did in the zeta sum of example 3 above is better.  Here that would be
!                   if (abs(z2/z3) < 1.0d-35) then

         if (z4 == z3) then
             fmt = "(///' Sample 13.', 2x, i5, ' terms were added to get exp(1.23-2.34i).'/)"
             write (*   , fmt) k
             write (kout, fmt) k
             exit
         endif
         z3 = z4
      enddo

!                Print the result.

      call zm_form('f33.30', 'f32.30', z3, st1)
      write (*   , "(' Result= ', a)") trim(st1)
      write (kout, "(' Result= ', a)") trim(st1)

!                Check the answer.

      z4 = to_zm('-2.37968179685477751574545797769674 5 - 2.458032970832342652397461908326042 i')
      if (.not.(abs(z4-z3) < 1.0d-31)) then
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 13.'/)")
          write (kout, "(/' Error in sample case number 13.'/)")
      endif


!             14. Exception handling.

!                Iterate (real) exp(x) starting at 1.0 until overflow occurs.

!                Testing to see if a type FM number is one of the special cases (+-overflow,
!                +-underflow or unknown) by direct comparison can be tricky.  When x1 is
!                +overflow, the statement
!                      if (x1 == to_fm(' +overflow ')) then
!                will return false, since the comparison routine cannot be sure that two
!                different overflowed results would have been equal if the overflow threshold
!                had been higher.

!                Function is_overflow can be used to directly check whether a number is + or -
```

```
!                overflow, so that is a safer test.

!                The FM warning message is written on unit kw, so in this test it appears on the
!                screen and not in the output file.

      call fm_set(60)

      x1 = to_fm(1)

      fmt = "(///' Sample 14.  Exception handling.'//12x,"                &
            "' Iterate exp(x) starting at 1.0 until overflow occurs.'//"  &
            "12x,' An FM warning message will be printed before the last iteration.')"
      write (*, fmt)
      fmt = "(///' Sample 14.  Exception handling.'//"                    &
            "12x,' Iterate exp(x) starting at 1.0 until overflow occurs.')"
      write (kout, fmt)

      do j = 1, 10
         x1 = exp(x1)
         call fm_form('es60.40', x1, st1)
         write (*   , "(/' Iteration', i3, 5x, a)") j, trim(st1)
         write (kout, "(/' Iteration', i3, 5x, a)") j, trim(st1)
         if (is_overflow(x1)) exit
      enddo

!             Check that the last result was +overflow.

      if (is_overflow(x1)) then
          write (*   , "(/' Overflow was correctly detected.')")
          write (kout, "(/' Overflow was correctly detected.')")
      else
          nerror = nerror + 1
          write (*   , "(/' Error in sample case number 14.'/)")
          write (*   , "(/' Overflow was not correctly detected.')")
          write (kout   , "(/' Error in sample case number 14.'/)")
          write (kout   , "(/' Overflow was not correctly detected.')")
      endif

      if (nerror == 0) then
          write (*   , "(//a/)") ' All results were ok -- no errors were found.'
          write (kout, "(//a/)") ' All results were ok -- no errors were found.'
      else
          write (*   , "(//i3, a/)") nerror, ' error(s) found.'
          write (kout, "(//i3, a/)") nerror, ' error(s) found.'
      endif

      close(kout)
      stop
      end program test

      subroutine plan_9(f, a, b, n, result)

!  Sample subroutine usage for FM.

!  Integrate f(x) from a to b using n subintervals, and return the answer in result.

!  This does numerical integration using a 9-point rule.
!  It is not a very good way to do high-precision integration, but it is a short routine
```

```
!  and can often get 20 to 30 digits if f(x) is well-behaved and the interval of integration
!  is not too big.

      use fmzm
      implicit none
      type (fm) :: a, b, result
      type (fm), save :: h, h8, xj
      integer :: n, j
      type (fm), external :: f
      intent (in) :: n, a, b
      intent (inout) :: result

      h = (b - a)/n
      h8 = h/8
      result = 0
      do j = 1, n
         xj = a + (j-1)*h
         result = result +   989*f(xj)      + 5888*f(xj+  h8) -   928*f(xj+2*h8) +  &
                           10496*f(xj+3*h8) - 4540*f(xj+4*h8) + 10496*f(xj+5*h8) -  &
                             928*f(xj+6*h8) + 5888*f(xj+7*h8) +   989*f(xj+8*h8)
      enddo
      result = h*result/28350

      end subroutine plan_9

      function f(x)     result (return_value)

!  Sample function usage for FM.

!  The test function for the integration subroutine is  2*exp(-x**2)/sqrt(pi).

      use fmzm
      implicit none
      type (fm) :: return_value, x
      intent (in) :: x
      type (fm), save :: pi

!             Compare the usage here with the sqrt(acos(to_fm(-1))) usage in the gamma example
!             in the main program.  There pi was only used once, so acos(to_fm(-1)) is more like
!             what a non-multiple-precision program would do to get pi.

!             If we need pi in a function like f that will be called hundreds of times, the acos
!             call will be done every time.  Here, since the argument is -1, the acos routine will
!             recognize it as a special case and return the saved value of pi without needlessly
!             making the program slower.  But if another formula were used, like pi = 6*asin(1/2),
!             it would be better to call fm_pi, since pi would be computed only once and later calls
!             just use the saved value of pi.

!             Another reason to call fm_pi instead of using a formula is that in case the calling
!             program changed the trig function mode to degrees, instead of the default radians,
!             then acos(to_fm(-1)) would give 180, not pi.

!             For this case the 2/sqrt(pi) could have been factored out of the integral so pi would
!             not be needed every time f is called, but it was left in to illustrate similar but
!             more complicated situations.

      call fm_pi(pi)
      return_value = 2*exp(-x**2)/sqrt(pi)
```

```
end function f
```