```
      program test
      use fmzm
      use fm_interval_arithmetic

!  This program contains the code for the FM interval arithmetic examples that are discussed in
!  the paper "A Multiple-Precision Interval Arithmetic Package".

      implicit none

      integer :: i, j, j_max, j_min, k, n, nf, k_function, n_order, n_steps, kprt, kw, k_graph,  &
                 base, n_precision, digit_count(9)
      type (fm_interval) :: a, b, result, s, term, pi, c, x, y, v(5), det
      type (fm_interval), allocatable :: matrix(:,:), rhs(:), soln(:)
      integer, allocatable :: kswap(:)
      type (fm) :: a_fm, b_fm, s_fm, pi_fm, c_fm, x_fm, y_fm
      type (fm) :: error_fm, width_fm, least_width_fm
      double precision :: rand

!             Set the FM precision to 50 digits.

      call fm_set(50)

!             Write output to file IntervalExamplesFM.out.
!             The fm_setvar call directs all output from within the FM package to unit 22 also.

      open(22, file='IntervalExamplesFM.out')
      call fm_setvar(' kw = 22 ')

!             Unit 31 will be used to write several files that give the interval width
!             for each step of the different examples.

      k_graph = 31


!             Example 1.  Ignoring variable correlation in formulas causes interval arithmetic
!                         to get too wide a resulting interval.

      write (22,*) ' '
      write (22,*) ' '
      write (22,*) " Example 1.   Compute f(x) = x**2 - x + 3 using different formulas."
      write (22,*) ' '

      x = to_fm_interval( " -0.5 ", " 1.0 " )

      write (22,*) ' '
      write (22,*) ' x is the interval [ -0.5 , 1.0 ]'
      write (22,*) ' '

      x_fm = 0.5
      a_fm = x_fm**2 - x_fm + 3
      x_fm = left_endpoint(x)
      b_fm = x_fm**2 - x_fm + 3
      least_width_fm = b_fm - a_fm

      y = x**2

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
```

```
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a)") ' x**2 gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)), ' ]'

      y = x*x

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a)") ' x*x  gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)), ' ]'
      write (22,*) ' '
      write (22,*) ' '

      y = x**2 - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x**2 - x + 3       gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*x - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*x - x + 3        gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*(x - 1) + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*(x - 1) + 3      gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = (x - 0.5)**2 + 2.75

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' (x - 0.5)**2 + 2.75 gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      write (22,*) ' '
      write (22,*) ' '
      write (22,*) ' '

!             Results are not as bad when the interval doesn't include zero.

      x = to_fm_interval( " 0.1 ", " 1.0 " )

      write (22,*) ' '
      write (22,*) ' x is the interval [ 0.1 , 1.0 ]'
      write (22,*) ' '

      x_fm = 0.5
```

```
      a_fm = x_fm**2 - x_fm + 3
      x_fm = right_endpoint(x)
      b_fm = x_fm**2 - x_fm + 3
      least_width_fm = b_fm - a_fm

      y = x**2 - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x**2 - x + 3        gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*x - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*x - x + 3         gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*(x - 1) + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*(x - 1) + 3       gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = (x - 0.5)**2 + 2.75

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' (x - 0.5)**2 + 2.75 gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      write (22,*) ' '
      write (22,*) ' '
      write (22,*) ' '

!             Results are better when the interval doesn't include a local minimum.

      x = to_fm_interval( " 0.9 ", " 1.0 " )

      write (22,*) ' '
      write (22,*) ' x is the interval [ 0.9 , 1.0 ]'
      write (22,*) ' '

      x_fm = left_endpoint(x)
      a_fm = x_fm**2 - x_fm + 3
      x_fm = right_endpoint(x)
      b_fm = x_fm**2 - x_fm + 3
      least_width_fm = b_fm - a_fm

      y = x**2 - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
```

```
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x**2 - x + 3        gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*x - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*x - x + 3         gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*(x - 1) + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*(x - 1) + 3       gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = (x - 0.5)**2 + 2.75

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' (x - 0.5)**2 + 2.75 gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      write (22,*) ' '
      write (22,*) ' '
      write (22,*) ' '

!             Look at an even smaller interval

      x = to_fm_interval( " 0.99 ", " 1.0 " )

      write (22,*) ' '
      write (22,*) ' x is the interval [ 0.99 , 1.0 ]'
      write (22,*) ' '

      x_fm = left_endpoint(x)
      a_fm = x_fm**2 - x_fm + 3
      x_fm = right_endpoint(x)
      b_fm = x_fm**2 - x_fm + 3
      least_width_fm = b_fm - a_fm

      y = x**2 - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x**2 - x + 3        gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),         &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*x - x + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
```

```
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*x - x + 3        gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = x*(x - 1) + 3

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' x*(x - 1) + 3      gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)

      y = (x - 0.5)**2 + 2.75

      c_fm = (right_endpoint(y) - left_endpoint(y)) / least_width_fm
      write (22,*) ' '
      write (22, "(a, f6.3, a, f6.3, a, f6.3)") ' (x - 0.5)**2 + 2.75 gives  [ ',  &
                 to_dp(left_endpoint(y)), ' , ', to_dp(right_endpoint(y)),        &
                 ' ].   Magnification = ', to_dp(c_fm)
      write (22,*) ' '




!             Example 2.  Sum  1 / n^7  from 1 to 100,000.


      open(k_graph, file='interval_examples Sum')

      write (k_graph, "(//a//)") 'Example 2.  Sum  1 / n^7  from 1 to 100,000.'

      n = 10**5
      s = 0
      do j = 1, n
         term = j
         s = s + 1 / term**7
         if (mod(j, n/1000) == 1) then
             width_fm = max( right_endpoint(s)-left_endpoint(s) , epsilon(left_endpoint(s)) )
             write (k_graph, "(a, i7, a, f8.3, a)") '{', j, ', ',  &
                                                    to_dp(log(width_fm)) / log(10.0d0), '}, '
         endif
      enddo

      write (22,*) ' '
      write (22,*) " Example 2.  Sum  1 / n^7  from 1 to 100,000 ="
      call fmprint_interval(s)
      write (22,*) ' '
      error_fm = abs( (right_endpoint(s)-left_endpoint(s)) / right_endpoint(s) )
      j = -nint(log10(error_fm))
      write (22, "(a, i3, a)") '      The two endpoints agree to about', j, ' decimal digits.'
      write (22,*) ' '
      close(k_graph)
```

```
!             Example 3.  Use composite 9-point Gauss quadrature with 1,000 subintervals
!                         to integrate  sin(t) / t  from  t = 0  to  t = 20.


      open(k_graph, file='interval_examples Integration')

      write (k_graph, "(//a//)")  &
                     'Example 3.  Use composite 9-point Gauss quadrature with 1,000 subintervals'

      nf = 1
      n = 1000
      a = 0
      b = 20
      call gauss_9(nf, a, b, n, result)

      write (22,*) ' '
      write (22,*) " Example 3.  Integrate  sin(t) / t  from  t = 0  to  t = 20:"
      call fmprint_interval(result)
      write (22,*) ' '
      error_fm = abs( (right_endpoint(result)-left_endpoint(result)) / right_endpoint(result) )
      j = -nint(log10(error_fm))
      write (22, "(a, i3, a)") '      The two endpoints agree to about', j, ' decimal digits.'
      write (22,*) ' '
      close(k_graph)




!             Example 4.  Start with  ( x, y ) = ( 1, 0 )  and step around the unit circle
!                         using the recurrence ( x, y ) <-- ( x*c - y*d, y*c + x*d ),
!                         c = cos(2*pi/n) and  s = sin(2*pi/n)

!                         After 23 trips around the circle with 100 steps per trip,
!                         x = ( -58.6 , 60.6 )  and  y = ( -59.6 , 59.6 )

!                         After 21 trips around the circle with 1000 steps per trip,
!                         x = ( 0.49 , 1.51 )  and  y = ( -0.51 , 0.51 )


      open(k_graph, file='interval_examples Circle Recurrence')

      write (k_graph, "(//a//)")  &
                     'Example 4.  Start with  ( x, y ) = ( 1, 0 )  and step around the unit circle'

      n = 100
      x = 1
      y = 0
      pi = acos(to_fm(-1))
      c = cos(2*pi/n)
      s = sin(2*pi/n)

      write (22,*) ' '
      write (22,*) " Example 4.  Step around the unit circle multiple times using a recurrence"
      write (22,*) ' '

      do k = 1, 23
```

```
      do j = 1, n
         a = x*c - y*s
         b = y*c + x*s
         x = a
         y = b
         width_fm = max( right_endpoint(x)-left_endpoint(x) , epsilon(left_endpoint(x))/10**7 )
         write (k_graph, "(a, i7, a, f8.3, a)") '{', n*(k-1)+j,  &
                                                ', ', to_dp(log(width_fm)) / log(10.0d0), '}, '
      enddo
      error_fm = abs( (right_endpoint(x)-left_endpoint(x)) / right_endpoint(x) )
      j = -nint(log10(error_fm))
      write (22, "(i4, a, i3, a)") k, ' trips.  The two endpoints for x agree to about',  &
                                   j, ' decimal digits.'
   enddo

   write (22,*) ' '
   write (22,*) ' x ='
   call fmprint_interval(x)
   write (22,*) ' '
   write (22,*) ' y ='
   call fmprint_interval(y)
   write (22,*) ' '
   error_fm = abs( (right_endpoint(x)-left_endpoint(x)) / right_endpoint(x) )
   j = -nint(log10(error_fm))
   write (22, "(a, i3, a)") '      The two endpoints for x agree to about', j, ' decimal digits.'
   write (22,*) ' '


!                       Compare the same calculation in ordinary FM arithmetic.

!                       After 23 trips around the circle with 100 steps per trip,
!                       x = 1.0 to 60 digits  and  y = -2.3e-55

!                       After 23 trips around the circle with 1000 steps per trip,
!                       x = 1.0 exactly  and  y = -2.3e-55

   write (22,*) ' '
   write (22,*) ' Compare the same calculation in ordinary FM arithmetic.'
   write (22,*) ' '
   n = 100
   x_fm = 1
   y_fm = 0
   pi_fm = acos(to_fm(-1))
   c_fm = cos(2*pi_fm/n)
   s_fm = sin(2*pi_fm/n)

   do k = 1, 23
      do j = 1, n
         a_fm = x_fm*c_fm - y_fm*s_fm
         b_fm = y_fm*c_fm + x_fm*s_fm
         x_fm = a_fm
         y_fm = b_fm
      enddo
      error_fm = abs( x_fm - 1 ) + to_fm(' 1.0e-99 ')
      j = -nint(log10(error_fm))
      write (22, "(i4, a, i3, a)") k, ' trips.  x agrees with 1.0 to about',  &
                                   j, ' decimal digits.'
   enddo
```

```
      write (22,*) ' '
      write (22,*) ' x ='
      call fm_print(x_fm)
      write (22,*) ' '
      write (22,*) ' y ='
      call fm_print(y_fm)
      write (22,*) ' '
      close(k_graph)


!             Test to see if it is x and y oscillating between positive and negative values
!             that causes the exponential growth of interval width, or if oscillation through
!             positive values alone can cause it.

!             Use the recurrence to oscillate back and forth between step number 8 and
!             step number 18 of the original recurrence.
!             Once back and forth takes 20 steps, so 5*23 = 115 trips back and forth will
!             take the same number of steps as the 23 trips around the circle above.


      open(k_graph, file='interval_examples Back and Forth Recurrence')

      write (k_graph, "(//a/a//)")                                                    &
            'Example 4b.  Start with  ( x, y ) = ( cos( 8* 2*pi/n), sin( 8* 2*pi/n) ) ',  &
            ' and step around the unit circle to ( cos(18* 2*pi/n), sin(18* 2*pi/n) ) and back'

      write (22,*) ' '
      write (22,*) " Example 4b.  Step back and forth around the unit circle using a recurrence"
      write (22,*) ' '

      n = 100
      pi = acos(to_fm(-1))
      x = cos(8* 2*pi/n)
      y = sin(8* 2*pi/n)
      c = cos(2*pi/n)
      s = sin(2*pi/n)

      write (22,*) ' '
      write (22,*) ' Starting point for x ='
      call fmprint_interval(x)
      write (22,*) ' '
      write (22,*) ' Starting point for y ='
      call fmprint_interval(y)
      write (22,*) ' '

!             5 back and forth trips equals 100 steps, the same as one circle trip above.

      do k = 1, 23 * 5
         do j = 1, 20
            a = x*c - y*s
            b = y*c + x*s
            x = a
            y = b
            width_fm = max( right_endpoint(x)-left_endpoint(x) , epsilon(left_endpoint(x))/10**7 )
            write (k_graph, "(a, i7, a, f8.3, a)") '{', (20*(k-1)+j),  &
                                                   ', ', to_dp(log(width_fm)) / log(10.0d0), '}, '
```

```
      if (mod(j, 10) == 0) then
          s = -s
      endif
   enddo
   if (mod(k, 5) == 0) then
       error_fm = abs( (right_endpoint(x)-left_endpoint(x)) / right_endpoint(x) )
       j = -nint(log10(error_fm))
       write (22, "(i4, a, i3, a)") k, ' trips.  The two endpoints for x agree to about',  &
                                    j, ' decimal digits.'
   endif
enddo

write (22,*) ' '
write (22,*) ' x ='
call fmprint_interval(x)
write (22,*) ' '
write (22,*) ' y ='
call fmprint_interval(y)
write (22,*) ' '
error_fm = abs( (right_endpoint(x)-left_endpoint(x)) / right_endpoint(x) )
j = -nint(log10(error_fm))
write (22, "(a, i3, a)") '      The two endpoints for x agree to about', j, ' decimal digits.'
write (22,*) ' '
close(k_graph)




!             Example 5.  2 * Product  4*n^2 / (4*n^2-1)  from 1 to 10,000.


open(k_graph, file='interval_examples Product')

write (k_graph, "(//a//)") 'Example 5.  2 * Product  4*n^2 / (4*n^2-1)  from 1 to 10,000.'

n = 10**4
s = 2
do j = 1, n
   k = 4 * j * j
   term = k
   term = term / ( k - 1 )
   s = s * term
   if (mod(j, n/1000) == 1) then
       width_fm = max( right_endpoint(s)-left_endpoint(s) , epsilon(left_endpoint(s)) )
       write (k_graph, "(a, i7, a, f8.3, a)") '{', j, ', ',  &
                                              to_dp(log(width_fm)) / log(10.0d0), '}, '
   endif
enddo

write (22,*) ' '
write (22,*) " Example 5.  2 * Product  4*n^2 / (4*n^2-1)  from 1 to 10,000. ="
call fmprint_interval(s)
write (22,*) ' '
error_fm = abs( (right_endpoint(s)-left_endpoint(s)) / right_endpoint(s) )
j = -nint(log10(error_fm))
write (22, "(a, i3, a)") '      The two endpoints of y(30) agree to about', j,  &
                         ' decimal digits.'
```

```
      write (22,*) ' '




!             Example 6.  Differential equation.  y'' = -y' / 10  -  2 * y / (x+2),
!                                                 y(0) = 0,   y'(0) = 1

!                         This solution has 2 roots between 0 and 30.


      open(k_graph, file='interval_examples Diff Eq 1')

      write (k_graph, "(//a//)")  &
                     "Example 6.  Differential equation.  y'' = -y' / 10  -  2 * y / (x+2), "


      write (22,*) ' '
      write (22,*) " Example 6.  Differential equation.  y'' = -y' / 10  -  2 * y / (x+2)"
      write (22,*) ' '
      n_order = 2
      a = 0
      b = 30
      n_steps = 10000
      v(1) = 0
      v(2) = 1
      k_function = 1
      kprt = n_steps / 10
      kw = 22
      call rk4(n_order, a, b, n_steps, v, k_function, kprt, kw)
      write (22,*) ' '
      write (22,*) ' y(30) = '
      write (22,*) ' '
      call fmprint_interval(v(1))
      write (22,*) ' '
      write (22,*) " y'(30) = "
      write (22,*) ' '
      call fmprint_interval(v(2))
      write (22,*) ' '
      error_fm = abs( (right_endpoint(v(1))-left_endpoint(v(1))) / right_endpoint(v(1)) )
      j = -nint(log10(error_fm))
      write (22, "(a, i3, a)") '      The two endpoints of y(30) agree to about', j,  &
                               ' decimal digits.'
      write (22,*) ' '




!             Example 7.  Differential equation.  y'' = -y' / 10  -  200 * y / (x+2),
!                                                 y(0) = 0,   y'(0) = 1

!                         This solution has 38 roots between 0 and 30.
!                         The rapid oscillation causes the interval calculation to be unstable
!                         and lose accuracy in a manner similar to example 4, the stepping
!                         recurrence making multiple trips around the unit circle.
```

```
      open(k_graph, file='interval_examples Diff Eq 2')

      write (k_graph, "(//a//)")  &
                     "Example 7.  Differential equation.  y'' = -y' / 10  -  200 * y / (x+2), "


      write (22,*) ' '
      write (22,*) " Example 7.  Differential equation.  y'' = -y' / 10  -  200 * y / (x+2)"
      write (22,*) ' '
      n_order = 2
      a = 0
      b = 30
      n_steps = 10000
      v(1) = 0
      v(2) = 1
      k_function = 2
      kprt = n_steps / 10
      kw = 22
      call rk4(n_order, a, b, n_steps, v, k_function, kprt, kw)
      write (22,*) ' '
      write (22,*) ' y(30) = '
      write (22,*) ' '
      call fmprint_interval(v(1))
      write (22,*) ' '
      write (22,*) " y'(30) = "
      write (22,*) ' '
      call fmprint_interval(v(2))
      write (22,*) ' '
      error_fm = abs( (right_endpoint(v(1))-left_endpoint(v(1))) / right_endpoint(v(1)) )
      j = -nint(log10(error_fm))
      write (22, "(a, i3, a)") '      The two endpoints of y(30) agree to about', j,  &
                               ' decimal digits.'
      write (22,*) ' '
      close(k_graph)




!             Example 8.  Solve a "random" n x n linear system.


      write (22,*) ' '
      write (22,*) ' Example 8.  Solve a "random" n x n linear system.'
      write (22,*) ' '

      do n = 10, 100, 10

         allocate( matrix(n, n), rhs(n), soln(n), kswap(n) )
         do i = 1, n
            do j = 1, n
               call fm_random_number(rand)
               matrix(i, j) = rand
            enddo
            rhs(i) = i
         enddo
```

```
      call fm_interval_factor_lu(matrix, n, det, kswap)
      call fm_interval_solve_lu (matrix, n, rhs, soln, kswap)

      write (22,*) ' '
      j_min = 99
      j_max = 0
      do i = 1, n
         error_fm = abs( (right_endpoint(soln(i))-left_endpoint(soln(i))) /  &
                         right_endpoint(soln(i)) )
         j = -nint(log10(error_fm))
         j_min = min( j_min, j )
         j_max = max( j_max, j )
      enddo
      write (22, "(a, i3, a, i3, a, i3, a)") ' For n = ', n,                          &
                                             ' the solution elements agree to between ',  &
                                             j_min, ' and ', j_max, ' significant digits.'

      deallocate( matrix, rhs, soln, kswap )
   enddo




!            Example 9.  Newton's method starting with a single point.  Solve  x*exp(x) - 2 = 0.


   write (22,*) ' '
   write (22,*) ' '
   write (22,*) " Example 9.  Newton's method starting with a single point."
   write (22,*) ' '

   x = 1
   do j = 1, 10
      a = x*exp(x) - 2
      b = (x+1)*exp(x)
      x = x - a/b
      write (22,*) ' '
      write (22, "(a, i2, a)") " Iteration ", j, '.  x ='
      call fmprint_interval(x)
   enddo




!            Example 10.  Newton's method starting with an interval containing a root.
!                         Solve  x*exp(x) - 2 = 0.
!                         The next interval is xt - f(xt)/f'(x) intersected with x.
!                         xt is a single point from the interval x (the midpoint is used here).
!                         The idea is to have a contracting sequence of intervals that
!                         each contain a root.

!                         Refs:  Hansen, E. R., and Greenberg, R. I., "An Interval Newton method",
!                                    Appl. Math. Comput. 12 (1983), 89-98.
!                                Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud,
!                                    "Introduction to Interval Analysis", chapter 8, "Interval
!                                    Newton Methods", SIAM (2009)
```

```
!                                Mayer, G., "Some remarks on two interval-arithmetic modifications
!                                    of the newton method", Computing 48, (1992),  125-128
!                                Pascal Van Hentenryck, David McAllester, and Deepak Kapur,
!                                    "Solving Polynomial Systems Using a Branch and Prune Approach"
!                                    SIAM Journal on Numerical AnalysisVol. 34, 2 (1997)

      write (22,*) ' '
      write (22,*) ' '
      write (22,*) " Example 10.   Newton's method starting with an interval containing a root"
      write (22,*) ' '

      x = to_fm_interval( " 0.5 ", " 1.0 " )
      do j = 1, 9
         x_fm = left_endpoint(x)/2 + right_endpoint(x)/2
         y_fm = x_fm*exp(x_fm) - 2
         b = (x+1)*exp(x)
         y = x_fm - y_fm/b

         x = to_fm_interval( max( left_endpoint(x), left_endpoint(y) ) ,  &
                             min( right_endpoint(x), right_endpoint(y) ) )
         write (22,*) ' '
         write (22, "(a, i2, a)") " Iteration ", j, '.  x ='
         call fmprint_interval(x)
      enddo




!             Example 11.  Check "Benford's Law", a result about the distribution of the
!                          leading digits is sets of numerical data.
!                          Count the exact number of leading digits 1, 2, ..., 9 found
!                          in the first billion powers of base = 2, 3, 5.
!                          Ref:  "The Surprising Accuracy of Benford's Law in Mathematics",
!                                Zhaodong Cai, Matthew Faust, A. J. Hildebrand, Junxian Li,
!                                and Yuan Zhang -- American Mathematical Monthly, March 2020.


      base = 2

      write (22, "(//a, i2/)") ' Example 11.   Count leading digits of powers of', base

      n = 10**9
      n_precision = 20
      call fm_set(n_precision)

      digit_count = 0
      x = 1
      do j = 1, n
         x = x * base

!             Check that the two endpoints of the interval have the same leading digit.
!             If so, that proves we have computed the correct leading digit.
!             If not, write a message about needing higher precision and stop.

         a_fm = left_endpoint(x)
         b_fm = right_endpoint(x)
```

```
!             In this comparison a_fm%mfm%mp(2) is the FM exponent and a_fm%mfm%mp(3) is the FM
!             first digit of a_fm.  So this is a quick way to check that the two endpoints
!             of interval x agree about the leading digit of base**j.

         if (a_fm%mfm%mp(2) /= b_fm%mfm%mp(2) .or. a_fm%mfm%mp(3) /= b_fm%mfm%mp(3)) then
             write (22, "(//a, i6)") ' In example 11 the two endpoints do not agree about the' //  &
                                     ' leading digit.  n_precision was ', n_precision
             write (22, "(//a//)") ' Try again with n_precision set 10 higher'
             stop
         endif

!              Extract the leading decimal digit from base**j.
!              FM is storing the numbers in a power-of-ten base, so the leading FM digit
!              may have several base 10 digits.  Divide by 10 until the base 10 leading
!              digit remains.

         k = a_fm%mfm%mp(3)
         do while (k > 0)
            i = k/10
            if (i == 0) then
                digit_count(k) = digit_count(k) + 1
                exit
            else
                k = i
            endif
         enddo

!              5^(1,000,000,000) could overflow if FM is being run on a machine using 32-bit
!              integers.  Since we only care about the leading digits of the powers, normalize
!              the FM exponent of x if it gets too big.

         if (a_fm%mfm%mp(2) > 5.0d+7) then
             x%left%mp(2)  = 0
             x%right%mp(2) = 0
         endif
      enddo

      write (22, "(//i2, a, 9i10//)") base, '^n counts = ', digit_count


      write (22,*) ' '
      write (22,*) ' '

      close(22)
      stop

      end program test


      function f(nf, x)     result (return_value)

      use fmzm
      use fm_interval_arithmetic
      implicit none
      integer :: nf
      type (fm_interval) :: return_value, x
      intent (in) :: nf, x
```

```
      if (nf == 1) then
          return_value = sin(x) / x
      endif

      end function f


      subroutine gauss_9(nf, a, b, n, result)

!  Sample subroutine usage for FM.

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

!  This does numerical integration using a 9-point Gauss quadrature rule.
!  It is not a very good way to do high-precision integration, but it is a short routine
!  and can often get 50 digits if f(x) is well-behaved and the interval of integration
!  is not too big.

      use fmvals
      use fmzm
      use fm_interval_arithmetic
      implicit none
      type (fm_interval) :: a, b, result
      type (fm_interval), save :: aj, h2, xj, xi(9), wi(9)
      type (fm_interval), external :: f
      type (fm), save :: width_fm
      integer :: nf, n, j, k
      intent (in) :: nf, n, a, b
      intent (inout) :: result
      integer, save :: coeff_base = 0, coeff_precision = 0

      if (coeff_base /= mbase .or. coeff_precision < ndig) then
          coeff_base = mbase
          coeff_precision = ndig
          xi(9) = to_fm(' 0.968160239507626089835576202903672870049404800491925329550023 ')
          xi(1) = -xi(9)
          xi(8) = to_fm(' 0.836031107326635794299429788069734876544106718124675996104372 ')
          xi(2) = -xi(8)
          xi(7) = to_fm(' 0.613371432700590397308702039341474184785720604940564692872813 ')
          xi(3) = -xi(7)
          xi(6) = to_fm(' 0.324253423403808929038538014643336608571956260736973088827047 ')
          xi(4) = -xi(6)
          xi(5) = 0

          wi(9) = to_fm(' 0.081274388361574411971892158110523650675661720782410750711108 ')
          wi(1) = wi(9)
          wi(8) = to_fm(' 0.180648160694857404058472031242912809514337821732040484498336 ')
          wi(2) = wi(8)
          wi(7) = to_fm(' 0.260610696402935462318742869418632849771840204437299951939997 ')
          wi(3) = wi(7)
          wi(6) = to_fm(' 0.312347077040002840068630406584443665598754861261904645554011 ')
          wi(4) = wi(6)
          wi(5) = to_fm(' 32768 ') / 99225
      endif

      result = 0
      h2    = ( b - a ) / ( 2 * n )
      do j = 1, n
```

```
         aj = a + (j-1) * 2 * h2
         do k = 1, 9
            xj = aj + h2 + h2 * xi(k)
            result = result + f( nf, xj ) * wi(k)
         enddo
         if (mod(j, n/1000) == 0) then
             width_fm = max( right_endpoint(result)-left_endpoint(result) ,  &
                             epsilon(left_endpoint(result)) )
             write (31, "(a, i7, a, f8.3, a)") '{', j, ', ',  &
                                               to_dp(log(width_fm)) / log(10.0d0), '}, '
         endif
      enddo

      result = result * h2

      end subroutine gauss_9



   module rk_funct

!  For a function to return an array as its function value, there must be an explicit interface.

!  f is the generic name of the function, f_rk is the specific version for 1-dimensional arrays.

      interface f
         module procedure f_rk
      end interface

      contains

      function f_rk( x, v, k_function )     result (return_value)
      use fmzm
      use fm_interval_arithmetic

      implicit none

      integer :: k_function
      type (fm_interval) :: x, v(5), return_value(5)
      intent (in) :: x, v, k_function

      return_value = 0

      if (k_function == 1) then

!             Function 1.   y'' = -y' / 10  -  2 * y / (x+2),
!                           y(0) = 0,   y'(0) = 1

!                           v' = {  y' ,  y'' } = {  u  ,  -u / 10  -  2 y / (x+2)  } = f(x, v)

          return_value(1) = v(2)
          return_value(2) = -v(2) / 10 - 2 * v(1) / ( x + 2 )

      else if (k_function == 2) then

!             Function 2.   y'' = -y' / 10  -  200 * y / (x+2),
!                           y(0) = 0,   y'(0) = 1
```

```
!                        v' = {  y' ,  y'' } = {  u  ,  -u / 10  -  2 y / (x+2)  } = f(x, v)

          return_value(1) = v(2)
          return_value(2) = -v(2) / 10 - 200 * v(1) / ( x + 2 )

       else
          return_value(1) = x*v(1)
       endif

       end function f_rk

    end module rk_funct


       subroutine rk4(n_order, a, b, n_steps, v, k_function, kprt, kw)
       use fmzm
       use fm_interval_arithmetic
       use rk_funct

!    n_order is the order of the de
!    a is the initial x-value
!    b is the final x-value
!    n_steps is the number of steps done.
!    v contains the initial values at x = a on input, and is returned with the solution
!      values at x = b.
!    k_function is the function number for the rhs.
!    kprt > 0 causes the solution to be printed on unit kw after each kprt steps.

       implicit none

       integer :: j, k, k_function, kprt, kw, n_order, n_steps
       type (fm_interval) :: a, b, h, x, v(5), k1(5), k2(5), k3(5), k4(5)
       intent (in) :: n_order, a, b, n_steps, k_function, kprt, kw
       intent (inout) :: v
       type (fm) :: error_fm, width_fm

       save :: h, x, k1, k2, k3, k4

       x = a
       h = (b-a)/n_steps
       if (n_order < 5) v(n_order+1:5) = 0

       do j = 1, n_steps
          k1 = h * f( x     , v      , k_function )
          k2 = h * f( x+h/2 , v+k1/2 , k_function )
          k3 = h * f( x+h/2 , v+k2/2 , k_function )
          k4 = h * f( x+h   , v+k3   , k_function )
          x = a + j*h
          v = v + ( k1 + 2*k2 + 2*k3 + k4 ) / 6

          if (kprt > 0) then
              if (mod(j, kprt) == 0) then
                  error_fm = abs( (right_endpoint(v(1))-left_endpoint(v(1))) / right_endpoint(v(1)) )
                  k = -nint(log10(error_fm))
                  write (kw, "(a, f15.10, a, 2f20.15, a, i3, a)") '  x = ', to_dp(x), '    v = ',  &
                             to_dp(v(1:n_order)), '     Endpoints of v(1) agree to ', k, ' digits.'
              endif
          endif
```

```
        if (mod(j, n_steps/1000) == 1) then
            width_fm = max( right_endpoint(v(1))-left_endpoint(v(1)) ,  &
                            epsilon(left_endpoint(v(1))) )
            if (k_function == 1) then
                write (31, "(a, i7, a, f8.3, a)") '{', j, ', ',  &
                                                  to_dp(log(width_fm)) / log(10.0d0), '}, '
            else
                write (31, "(a, i7, a, f8.3, a)") '{', j, ', ',  &
                                                  to_dp(log(width_fm)) / log(10.0d0), '}, '
            endif
        endif
      enddo

      end subroutine rk4


      subroutine fm_interval_factor_lu(a, n, det, kswap)
      use fmzm
      use fm_interval_arithmetic
      implicit none

!  Gauss elimination to factor the n x n matrix a (lu decomposition).

!  The time is proportional to  n**3.

!  Once this factorization has been done, a linear system  a x = b
!  with the same coefficient matrix a and n x 1 vector b can be solved
!  for x using routine fm_interval_solve_lu in time proportional to  n**2.

!  det is returned as the determinant of a.
!      Nonzero det means a solution can be found.
!      det = 0 is returned if the system is singular.

!  kswap is a list of row interchanges made by the partial pivoting strategy during the
!        elimination phase.

!  After returning, the values in matrix a have been replaced by the multipliers
!  used during elimination.  This is equivalent to factoring the a matrix into
!  a lower triangular matrix l times an upper triangular matrix u.

      integer :: n
      integer :: jcol, jdiag, jmax, jrow, kswap(n)
      type (fm_interval) :: a(n, n), det
      intent (in) :: n
      intent (inout) :: a, det, kswap
      type (fm_interval), save :: amax, amult, temp

      det = 1
      kswap(1:n) = 1
      if (n <= 0) then
          det = 0
          return
      endif
      if (n == 1) then
          kswap(1) = 1
          det = a(1, 1)
          return
```

```
      endif

!             Do the elimination phase.
!             jdiag is the current diagonal element below which the elimination proceeds.

      do jdiag = 1, n-1

!             Pivot to put the element with the largest absolute value on the diagonal.

         amax = abs(a(jdiag, jdiag))
         jmax = jdiag
         do jrow = jdiag+1, n
            if (abs(a(jrow, jdiag)) > amax) then
                amax = abs(a(jrow, jdiag))
                jmax = jrow
            endif
         enddo

!             If amax is zero here then the system is singular.

         if (amax == 0.0) then
             det = 0
             return
         endif

!             Swap rows jdiag and jmax unless they are the same row.

         kswap(jdiag) = jmax
         if (jmax /= jdiag) then
             det = -det
             do jcol = jdiag, n
                temp = a(jdiag, jcol)
                a(jdiag, jcol) = a(jmax, jcol)
                a(jmax, jcol) = temp
             enddo
         endif
         det = det * a(jdiag, jdiag)

!             For jrow = jdiag+1, ..., n, eliminate a(jrow, jdiag) by replacing row jrow by
!                 row jrow - a(jrow, jdiag) * row jdiag / a(jdiag, jdiag)

         do jrow = jdiag+1, n
            if (a(jrow, jdiag) == 0) cycle
            amult = a(jrow, jdiag)/a(jdiag, jdiag)

!             Save the multiplier for use later by fm_interval_solve_lu.

            a(jrow, jdiag) = amult
            a(jrow, jdiag+1:n) = a(jrow, jdiag+1:n) - amult*a(jdiag, jdiag+1:n)
         enddo
      enddo
      det = det * a(n, n)

      end subroutine fm_interval_factor_lu

      subroutine fm_interval_solve_lu(a, n, b, x, kswap)
      use fmzm
      use fm_interval_arithmetic
```

```
      implicit none

!  Solve a linear system  a x = b.
!  a is the n x n coefficient matrix, after having been factored by fm_interval_factor_lu.
!  b is the n x 1 right-hand-side vector.
!  x is returned with the solution of the linear system.
!  kswap is a list of row interchanges made by the partial pivoting strategy during the
!        elimination phase in fm_interval_factor_lu.
!  Time for this call is proportional to  n**2.

      integer :: n, kswap(n)
      type (fm_interval) :: a(n, n), b(n), x(n)
      intent (in) :: a, n, b, kswap
      intent (inout) :: x
      type (fm_interval), save :: temp
      integer :: jdiag, jmax

      if (n <= 0) then
          return
      endif
      if (n == 1) then
          x(1) = b(1) / a(1, 1)
          return
      endif
      x(1:n) = b(1:n)

!             Do the elimination phase operations only on x.
!             jdiag is the current diagonal element below which the elimination proceeds.

      do jdiag = 1, n-1

!             Pivot to put the element with the largest absolute value on the diagonal.

         jmax = kswap(jdiag)

!             Swap rows jdiag and jmax unless they are the same row.

         if (jmax /= jdiag) then
             temp = x(jdiag)
             x(jdiag) = x(jmax)
             x(jmax) = temp
         endif

!             For jrow = jdiag+1, ..., n, eliminate a(jrow, jdiag) by replacing row jrow by
!                 row jrow - a(jrow, jdiag) * row jdiag / a(jdiag, jdiag)
!             After factoring, a(jrow, jdiag) is the original a(jrow, jdiag) / a(jdiag, jdiag).

         x(jdiag+1:n) = x(jdiag+1:n) - a(jdiag+1:n, jdiag)*x(jdiag)
      enddo

!             Do the back substitution.

      do jdiag = n, 1, -1

!             Divide row jdiag by the diagonal element.

         x(jdiag) = x(jdiag) / a(jdiag, jdiag)
```