```fortran
      program test

!   This is a sample program using version 1.4 of the fmzm and fm_rational_arithmetic modules
!   for doing exact rational arithmetic using the fm_rational derived type.

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

      use fmvals
      use fmzm
      use fm_rational_arithmetic

      implicit none

!             Declare the multiple precision variables.  The three types used in this program are:
!             (fm) for multiple precision real
!             (im) for multiple precision integer
!             (fm_rational) for multiple precision rational

      type (fm), save                            :: det_fm, error, max_rel_error
      type (fm), save, allocatable               :: a_fm(:,:), b_fm(:), c_fm(:,:), x_fm(:)
      type (im), save                            :: c1, c2
      type (fm_rational), save                   :: check, det_rm, f_rm, t_rm
      type (fm_rational), save, allocatable :: a_rm(:,:), b_rm(:), c_rm(:,:), x_rm(:)

!             Declare the other variables (not multiple precision).

      character(80)  :: st1
      character(175) :: fmt
      integer :: i, i_max, j, j_max, k, kout, kw_save, n, kerror, nerror, p(4), q(4),  &
                 roots_found
      real :: t1, t2
      double precision :: value

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

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

!             kw is the unit used for all automatically generated output from the FM routines.
!             This includes calls to the various print routines, as well as error messages.
!             kw should also default to screen output.

      kw_save = kw

      call fm_set(50)
      call fm_setvar(' kswide = 100 ')
      nerror = 0


!             1.  Find all rational roots of the equation
!                 f(t) = 21*t**5 + 43*t**4 - 113*t**3 - 46*t**2 + 49*t + 10 = 0.

!                 The rational root theorem says that if a polynomial with integer
!                 coefficients has rational roots, they must be of the form p/q where
!                 p divides the constant term (10 here) and q divides the high-order
```

```fortran
!                 coefficient (21 here).

!                 This gives a short list of possibilities that can be quickly checked.
!                 p could be + or - { 1, 2, 5, 10 }, and q could be { 1, 3, 7, 21 }.
!                 That gives 2*4*4 = 32 possible roots to check.

!                 to_fm_rational is a conversion function for creating fm_rational numbers.
!                 There are several versions, including 1 or 2 integer arguments, 1 or 2
!                 string arguments, etc.  See the user manual for all the options.

      fmt = "(///' Sample 1.  Find all rational roots: " //  &
            " f(t) = 21*t**5 + 43*t**4 - 113*t**3 - 46*t**2 + 49*t + 10 = 0'/)"
      write (*   , fmt)
      write (kout, fmt)

      p = (/ 1, 2, 5, 10 /)
      q = (/ 1, 3, 7, 21 /)
      roots_found = 0
      do i = 2, 1, -1
         do j = 1, 4
            do k = 1, 4
               t_rm = (-1)**i * to_fm_rational( p(j), q(k) )
               f_rm = 21*t_rm**5 + 43*t_rm**4 - 113*t_rm**3 - 46*t_rm**2 + 49*t_rm + 10
               if (f_rm == 0) then
                   write (*   , "(a)") "     Exact rational root found:"
                   call fm_print_rational( t_rm )
                   write (kout, "(a)") "     Exact rational root found:"
                   kw = kout
                   call fm_print_rational( t_rm )
                   kw = kw_save

!                  Check the results.

                   roots_found = roots_found + 1
                   if (roots_found == 1) then
                       if (.not.(t_rm == to_fm_rational(' 2/3 '))) then
                           nerror = nerror + 1
                       endif
                   else if (roots_found == 2) then
                       if (.not.(t_rm == to_fm_rational(' -5 / 7 '))) then
                           nerror = nerror + 1
                       endif
                   else if (roots_found > 2) then
                           nerror = nerror + 1
                   endif
               endif
            enddo
         enddo
      enddo

      if (nerror > 0 .or. roots_found /= 2) then
          write (*   , "(/' Error in sample case number 1.'/)")
          write (kout, "(/' Error in sample case number 1.'/)")
      endif


!             2.  Exact solution of linear systems (integer coefficients)
```

```fortran
!                 Many linear systems of equations have integer coefficients, making the solutions
!                 rational.  Others have rational coefficients, also giving rational solutions.

!                 Generate several systems of the type that come from some kinds of least-squares
!                 problems.

!                 The coefficient matrix is n x n for n = 25, 50, 75, 100.

!                 Compare the accuracy and speed of the floating-point routine fm_lin_solve
!                 with the exact rational routine rm_lin_solve.

!                 Note that for systems with integer coefficients, it can be faster to find
!                 the exact rational solution than to find a 50-digit approximate solution,
!                 even though in the 100x100 case the numerators and denominators have over
!                 200 digits each.

      fmt = "(///' Sample 2.  Solve four n x n linear systems having small integer coefficients.')"
      write (*    , fmt)
      write (kout, fmt)
      kerror = 0

      do n = 25, 100, 25
         write (*    ,*) ' '
         write (kout,*) ' '
         allocate(a_fm(n, n), b_fm(n), x_fm(n), a_rm(n, n), b_rm(n), x_rm(n))

         a_fm = 0
         b_fm = 0
         x_fm = 0
         a_rm = 0
         b_rm = 0
         x_rm = 0
         do j = 1, n*n
            call fm_random_number(value)
            i = n*value + 1
            call fm_random_number(value)
            k = n*value + 1
            a_rm(i, i) = a_rm(i, i) + 1
            a_rm(i, k) = a_rm(i, k) - 1
            a_rm(k, k) = a_rm(k, k) + 1
            a_rm(k, i) = a_rm(k, i) - 1
            call fm_random_number(value)
            b_rm(i) = b_rm(i) + (i-k) + int(12*value - 6)
            b_rm(k) = b_rm(k) - (i-k) + int(12*value - 6)
         enddo
         a_rm(n, 1:n) = 0
         a_rm(n, n) = 1
         b_rm(n) = n
         a_fm = to_fm( a_rm )
         b_fm = to_fm( b_rm )

!             Solve the system with floating-point 50 significant arithmetic.

!             Routines with names like fm_lin_solve_rm are copies of the corresponding routines
!             (fm_lin_solve here) from the standard floating-point FM sample routine file.  The
!             ones with names ending "_rm" are available in the fm_rational_arithmetic module.

         call cpu_time(t1)
```

```
         call fm_lin_solve_rm(a_fm, x_fm, b_fm, n, det_fm)
         call cpu_time(t2)

         fmt = "(/'    fm_lin_solve approximate solution for ', i4, ' x', i4, ' system in', " //  &
               "f12.2, ' seconds.')"
         write (*   , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*   ,*) '             Determinant ='
         write (kout,*) '             Determinant ='
         call fm_print(det_fm)
         kw = kout
         call fm_print(det_fm)
         kw = kw_save
         write (*   ,*) '             x(1) ='
         write (kout,*) '             x(1) ='
         call fm_print(x_fm(1))
         kw = kout
         call fm_print(x_fm(1))
         kw = kw_save

!             Solve the system with exact rational arithmetic.

         call cpu_time(t1)
         call rm_lin_solve(a_rm, x_rm, b_rm, n, det_rm)
         call cpu_time(t2)

         fmt = "(/'    rm_lin_solve       exact solution for ', i4, ' x', i4, ' system in', " //  &
               "f12.2, ' seconds.')"
         write (*   , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*   ,*) '             Determinant ='
         write (kout,*) '             Determinant ='
         call fm_print_rational(det_rm)
         kw = kout
         call fm_print_rational(det_rm)
         kw = kw_save
         write (*   ,*) '             x(1) ='
         write (kout,*) '             x(1) ='
         call fm_print_rational(x_rm(1))
         kw = kout
         call fm_print_rational(x_rm(1))
         kw = kw_save

!                  Check the results.

         if (.not.(abs(det_fm - to_fm(det_rm)) <= 1.0d-45*abs(det_fm))) then
             kerror = kerror + 1
         endif
         do j = 1, n
            if (.not.(abs(x_fm(j) - to_fm(x_rm(j))) < 1.0d-45)) then
                kerror = kerror + 1
            endif
         enddo

         deallocate(a_fm, b_fm, x_fm, a_rm, b_rm, x_rm)
      enddo

      if (kerror > 0) then
```

```
         write (*    , "(/' Error in sample case number 2.'/)")
         write (kout, "(/' Error in sample case number 2.'/)")
         nerror = nerror + 1
     endif


!             3.  Exact solution of linear systems (rational coefficients).

!                 Modify sample 2 so that the coefficients are rationals with numerators and
!                 denominators having no more than 2 digits.

!                 This causes the number of digits in the rational solution's numerators and
!                 denominators to get much larger, slowing rm_lin_solve compared to fm_lin_solve.

!                 Use smaller n's for the coefficient matrix here: n x n for n = 10, 20, 30, 40.

     fmt = "(///' Sample 3.  Solve four n x n linear systems, this time having non-integer" //  &
           " rational coefficients.'/)"
     write (*    , fmt)
     write (kout, fmt)
     kerror = 0

     do n = 10, 40, 10
        write (*    ,*) ' '
        write (kout,*) ' '
        allocate(a_fm(n, n), b_fm(n), x_fm(n), a_rm(n, n), b_rm(n), x_rm(n))

        a_fm = 0
        b_fm = 0
        x_fm = 0
        a_rm = 0
        b_rm = 0
        x_rm = 0
        do j = 1, n*n
           call fm_random_number(value)
           i = n*value + 1
           call fm_random_number(value)
           k = n*value + 1
           a_rm(i, i) = a_rm(i, i) + to_fm_rational( i, abs(k) + 1 )
           a_rm(i, k) = a_rm(i, k) - to_fm_rational( i, abs(k) + 1 )
           a_rm(k, k) = a_rm(k, k) + to_fm_rational( i, abs(k) + 1 )
           a_rm(k, i) = a_rm(k, i) - to_fm_rational( i, abs(k) + 1 )
           call fm_random_number(value)
           b_rm(i) = b_rm(i) + (i-k) + int(12*value - 6)
           b_rm(k) = b_rm(k) - (i-k) + int(12*value - 6)
        enddo
        a_rm(n, 1:n) = 0
        a_rm(n, n) = 1
        b_rm(n) = n
        a_fm = to_fm( a_rm )
        b_fm = to_fm( b_rm )

!             Solve the system with floating-point 50 significant arithmetic.

        call cpu_time(t1)
        call fm_lin_solve_rm(a_fm, x_fm, b_fm, n, det_fm)
        call cpu_time(t2)
```

```
         fmt = "(/'    fm_lin_solve approximate solution for ', i4, ' x', i4, ' system in', " //  &
               "f12.2, ' seconds.')"
         write (*    , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*    ,*) '             Determinant ='
         write (kout,*) '              Determinant ='
         call fm_print(det_fm)
         kw = kout
         call fm_print(det_fm)
         kw = kw_save
         write (*    ,*) '             x(1) ='
         write (kout,*) '              x(1) ='
         call fm_print(x_fm(1))
         kw = kout
         call fm_print(x_fm(1))
         kw = kw_save

!             Solve the system with exact rational arithmetic.

         call cpu_time(t1)
         call rm_lin_solve(a_rm, x_rm, b_rm, n, det_rm)
         call cpu_time(t2)

         fmt = "(/'    rm_lin_solve       exact solution for ', i4, ' x', i4, ' system in', " //  &
               "f12.2, ' seconds.')"
         write (*    , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*    ,*) '             Determinant ='
         write (kout,*) '              Determinant ='
         call fm_print_rational(det_rm)
         kw = kout
         call fm_print_rational(det_rm)
         kw = kw_save
         write (*    ,*) '             x(1) ='
         write (kout,*) '              x(1) ='
         call fm_print_rational(x_rm(1))
         kw = kout
         call fm_print_rational(x_rm(1))
         kw = kw_save

!                  Check the results.

         if (.not.(abs(det_fm - to_fm(det_rm)) <= 1.0d-45*abs(det_fm))) then
             kerror = kerror + 1
         endif
         do j = 1, n
            if (.not.(abs(x_fm(j) - to_fm(x_rm(j))) < 1.0d-45)) then
                kerror = kerror + 1
            endif
         enddo

         deallocate(a_fm, b_fm, x_fm, a_rm, b_rm, x_rm)
      enddo

      if (kerror > 0) then
          write (*    , "(/' Error in sample case number 3.'/)")
          write (kout, "(/' Error in sample case number 3.'/)")
          nerror = nerror + 1
```

```
      endif


!            4.  Exact matrix inverse.

!                One possible use for exact rational arithmetic is in looking for patterns
!                in the answers.

!                For an example, there is a formula for the determinant of the Hilbert matrix,
!                    a(j, k) = 1 / ( j + k - 1 ).
!                We might have a similar matrix where no formula is known and we could try
!                to discover one by examining factorizations of numerator and denominator.

!                Try this for the Hilbert matrix with n = 1, 2, ..., 5

!                n =              1         2             3                 4                    5
!                det = 1 /        1        12          2160           6048000         266716800000
!                factorization:   1     2^2 3     2^4 3^3 5     2^8 3^3 5^3 7     2^10 3^5 5^5 7^3

!                There are some clues that might help us guess a formula, but the first thing
!                to try is the On-line Encyclopedia of Integer Sequences, https://oeis.org/
!                entering   1, 12, 2160, 6048000, 266716800000   produces several references
!                to the inverse Hilbert matrix, where we can find a formula.

      fmt = "(///' Sample 4.  Examine determinants of several small Hilbert matrices.'/)"
      write (*   , fmt)
      write (kout, fmt)
      kerror = 0

      do n = 1, 5
         write (*   ,*) ' '
         write (kout,*) ' '
         allocate(a_fm(n, n), c_fm(n, n), a_rm(n, n), c_rm(n, n))

         a_fm = 0
         c_fm = 0
         a_rm = 0
         c_rm = 0
         do j = 1, n
            do k = 1, n
               a_rm(j, k) = to_fm_rational( 1, j+k-1 )
            enddo
         enddo
         a_fm = to_fm( a_rm )

!            Invert the matrix with floating-point 50 significant arithmetic.

         call cpu_time(t1)
         call fm_inverse_rm(a_fm, n, c_fm, det_fm)
         call cpu_time(t2)

         fmt = "(/'    fm_inverse approximate inverse for ', i4, ' x', i4, ' matrix in', " //  &
               "f12.2, ' seconds.')"
         write (*   , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*   ,*) '              Determinant ='
         write (kout,*) '              Determinant ='
         call fm_print(det_fm)
```

```
         kw = kout
         call fm_print(det_fm)
         kw = kw_save

!             Invert the matrix with exact rational arithmetic.

         call cpu_time(t1)
         call rm_inverse(a_rm, n, c_rm, det_rm)
         call cpu_time(t2)

         fmt = "(/'    rm_inverse       exact inverse for ', i4, ' x', i4, ' matrix in', " //  &
               "f12.2, ' seconds.')"
         write (*   , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*   ,*) '              Determinant ='
         write (kout,*) '              Determinant ='
         call fm_print_rational(det_rm)
         kw = kout
         call fm_print_rational(det_rm)
         kw = kw_save

!                  Check the results.

         if (.not.(abs(det_fm - to_fm(det_rm)) <= 1.0d-45*abs(det_fm))) then
             kerror = kerror + 1
         endif
         do j = 1, n
            do k = 1, n
               if (.not.(abs(c_fm(j, k) - to_fm(c_rm(j, k))) < 1.0d-45)) then
                   kerror = kerror + 1
               endif
            enddo
         enddo

         deallocate(a_fm, c_fm, a_rm, c_rm)
      enddo

      if (kerror > 0) then
          write (*   , "(/' Error in sample case number 4.'/)")
          write (kout, "(/' Error in sample case number 4.'/)")
          nerror = nerror + 1
      endif


!             5.  Exact matrix inverse.

!                 Use the Hilbert matrix with some larger values for n, and compare times with FM.

!                 There are two things to notice about this case:
!                 (1) The Hilbert matrix becomes so ill-conditioned as n increases that even
!                     carrying over 50 digits with floating-point arithmetic in fm_inverse
!                     is not enough.  The maximum relative error for elements of c_fm are:
!                         n =         10          20          30          40
!                         error =  1.09e-50    7.10e-36    1.15e-20    2.19e-5
!                     If we wanted 50-digit accuracy from fm_inverse for n=40, we would need
!                     to set precision to at least 100 digits.
!                 (2) The numerators and denominators in the Hilbert matrix are all fairly
!                     small, so the modular method is faster than fm_inverse, even though
```

```fortran
!                     the exact numerators and denominators have more than 50 digits.
!                     Timing will vary, but a typical result is for rm_inverse to run in
!                     less than half the time of fm_inverse.  The determinant for n = 40
!                     has over 900 digits in the denominator, but the largest element in
!                     the (integer-valued) inverse matrix has only 58 digits.

      fmt = "(///' Sample 5.  Examine determinants of several larger Hilbert matrices.'/)"
      write (*    , fmt)
      write (kout, fmt)
      kerror = 0

      do n = 10, 40, 10
         write (*    ,*) ' '
         write (kout,*) ' '
         allocate(a_fm(n, n), c_fm(n, n), a_rm(n, n), c_rm(n, n))

         a_fm = 0
         c_fm = 0
         a_rm = 0
         c_rm = 0
         do j = 1, n
            do k = 1, n
               a_rm(j, k) = to_fm_rational( 1, j+k-1 )
            enddo
         enddo
         a_fm = to_fm( a_rm )

!              Invert the matrix with floating-point 50 significant arithmetic.

         call cpu_time(t1)
         call fm_inverse_rm(a_fm, n, c_fm, det_fm)
         call cpu_time(t2)

         fmt = "(/'    fm_inverse approximate inverse for ', i4, ' x', i4, ' matrix in', " //  &
               "f12.2, ' seconds.')"
         write (*    , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*    ,*) '               1 / Determinant ='
         write (kout,*) '               1 / Determinant ='
         call fm_print(1/det_fm)
         kw = kout
         call fm_print(1/det_fm)
         kw = kw_save

!              Invert the matrix with exact rational arithmetic.

         call cpu_time(t1)
         call rm_inverse(a_rm, n, c_rm, det_rm)
         call cpu_time(t2)

         fmt = "(/'    rm_inverse       exact inverse for ', i4, ' x', i4, ' matrix in', " //  &
               "f12.2, ' seconds.')"
         write (*    , fmt) n, n, t2-t1
         write (kout, fmt) n, n, t2-t1
         write (*    ,*) '               Determinant ='
         write (kout,*) '               Determinant ='
         call fm_print_rational(det_rm)
         kw = kout
```

```
        call fm_print_rational(det_rm)
        kw = kw_save

!                Check the results.

!                Because the Hilbert matrix is pathologically ill-conditioned, even using
!                50 digits for the input to fm_inverse can give little accuracy in the
!                solution.  Use the mathematically exact values to check the results
!                from rm_inverse.

!                The correct determinant of the Hilbert matrix is always 1 / integer

!                = 1 / ( c(2n) / c(n)^4 ),  where c(n) = product( j^(n-j) ; j=1, n-1 )

        c1 = 1
        do j = 1, n-1
           c1 = c1 * to_im(j)**to_im(n-j)
        enddo
        c2 = 1
        do j = 1, 2*n-1
           c2 = c2 * to_im(j)**to_im(2*n-j)
        enddo
        c2 = c2 / c1**4
        if (.not.(det_rm == to_fm_rational( to_im(1), c2 ))) then
            kerror = kerror + 1
            if (kerror == 1) then
                write (*   , "(/' Error in determinant for sample case number 5.'/)")
                write (kout, "(/' Error in determinant for sample case number 5.'/)")
            endif
        endif

!                The correct elements of the inverse Hilbert matrix are:

!                c_rm(i, j) = (-1)^(i+j) * (i+j-1) * binomial(n+i-1, n-j) * binomial(n+j-1, n-i) *
!                                                   binomial(i+j-2, i-1)^2

        do i = 1, n
           do j = 1, n
              c1 = binomial(to_fm(n+i-1), to_fm(n-j)) * binomial(to_fm(n+j-1), to_fm(n-i))
              c2 = binomial(to_fm(i+j-2), to_fm(i-1))**2
              check = (-1)**(i+j) * (i+j-1) * c1 * c2
              if (.not.(c_rm(i, j) == check)) then
                  kerror = kerror + 1
                  if (kerror == 1) then
                      write (*   , "(/' Error in inverse element for sample case number 5.'/)")
                      write (kout, "(/' Error in inverse element for sample case number 5.'/)")
                  endif
              endif
           enddo
        enddo

!            Check how badly conditioned each matrix is by finding the least accurate element
!            in the computed inverse matrix from fm_inverse.
!            Use the relative error between c_fm and c_rm, since the numbers are large.

        max_rel_error = -1
        do i = 1, n
           do j = 1, n
```

```
            error = abs( ( c_fm(i, j) - to_fm(c_rm(i, j)) ) / to_fm(c_rm(i, j)) )
            if (error > max_rel_error) then
                max_rel_error = error
                i_max = i
                j_max = j
            endif
         enddo
      enddo

      fmt = "(/'     fm_inverse inverse matrix largest relative error" //  &
            " was in row', i3, ' column', i3, '.  Error =', a)"
      call fm_form('es14.5', max_rel_error, st1)
      write (*    , fmt) i_max, j_max, st1
      write (kout, fmt) i_max, j_max, st1

      deallocate(a_fm, c_fm, a_rm, c_rm)
   enddo

   if (kerror > 0) then
       write (*    , "(/' Error in sample case number 5.'/)")
       write (kout, "(/' Error in sample case number 5.'/)")
       nerror = nerror + 1
   endif


!             6.  Exact matrix inverse.

!                 Like sample 5, except use random a-matrices with numerators and denominators
!                 having no more than 2 digits.

   fmt = "(///' Sample 6.  Find four n x n inverse matrices, having random" //  &
         " 2-digit numerators and denominators.'/)"
   write (*    , fmt)
   write (kout, fmt)
   kerror = 0

   do n = 10, 40, 10
      write (*    ,*) ' '
      write (kout,*) ' '
      allocate(a_fm(n, n), c_fm(n, n), a_rm(n, n), c_rm(n, n))

      a_fm = 0
      c_fm = 0
      a_rm = 0
      c_rm = 0
      do i = 1, n
         do j = 1, n
            call fm_random_number(value)
            k = 198*value - 99
            a_rm(i, j) = k
            call fm_random_number(value)
            k = 99*value + 1
            a_rm(i, j) = a_rm(i, j) / k
            a_fm(i, j) = to_fm(a_rm(i, j))
         enddo
      enddo

!             Invert the matrix with floating-point 50 significant arithmetic.
```

```
      call cpu_time(t1)
      call fm_inverse_rm(a_fm, n, c_fm, det_fm)
      call cpu_time(t2)

      fmt = "(/'    fm_inverse approximate solution for ', i4, ' x', i4, ' system in', " //  &
            "f12.2, ' seconds.')"
      write (*   , fmt) n, n, t2-t1
      write (kout, fmt) n, n, t2-t1
      write (*   ,*) '              Determinant ='
      write (kout,*) '              Determinant ='
      call fm_print(det_fm)
      kw = kout
      call fm_print(det_fm)
      kw = kw_save

!              Invert the matrix with exact rational arithmetic.

      call cpu_time(t1)
      call rm_inverse(a_rm, n, c_rm, det_rm)
      call cpu_time(t2)

      fmt = "(/'    rm_inverse       exact solution for ', i4, ' x', i4, ' system in', " //  &
            "f12.2, ' seconds.')"
      write (*   , fmt) n, n, t2-t1
      write (kout, fmt) n, n, t2-t1
      write (*   ,*) '              Determinant ='
      write (kout,*) '              Determinant ='
      call fm_print_rational(det_rm)
      kw = kout
      call fm_print_rational(det_rm)
      kw = kw_save

!                   Check the results.

!                   These random matrices are not ill-conditioned, so the results can be checked
!                   by comparing the FM and rm inverses.

      if (.not.(abs(det_fm - to_fm(det_rm)) <= 1.0d-45*abs(det_fm))) then
          kerror = kerror + 1
      endif
      do i = 1, n
         do j = 1, n
            if (.not.(abs(c_fm(i, j) - to_fm(c_rm(i, j))) < 1.0d-45)) then
                kerror = kerror + 1
            endif
         enddo
      enddo

      deallocate(a_fm, c_fm, a_rm, c_rm)
   enddo

   if (kerror > 0) then
       write (*   , "(/' Error in sample case number 6.'/)")
       write (kout, "(/' Error in sample case number 6.'/)")
       nerror = nerror + 1
   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
```