Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 12

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

ARRAY PROCESSING AND MATRIX MANIPULATION

In section/week 7, it was discussed the basic principles of F's array facilities in the context of rank-one arrays. In mathematical terms, such arrays are suitable for representing vectors. In order to represent matrices or more complex rectangular structures, more than one subscript is required. The same general principles apply to rank-*n* arrays.

Matrices and two-dimensional arrays

Mathematically, a matrix is a two-dimensional rectangular array of elements. For example, a 3×4 matrix **A** consists of elements,

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \end{bmatrix}$$

F extends the concept of one-dimensional array (chapter or week 7) in a natural manner, by means of the `dimension` attribute. Thus, to define a two-dimensional array "a" requires storing the elements of the matrix A, and it would be written

```
real, dimension(3,4) :: a
```

In the dimension attribute, the number of rows is specified first and the number of columns second. *This order is important*.

As a second example, three 10×4, two-dimensional arrays b, c and d whose elements are logical would be created by,

```
logical, dimension(10,4) :: b, c, d
```

The elements of an array can be used anywhere it is legitimate to use a scalar.
They can occur in arithmetic expressions, be passed as actual arguments, occur in I/O statements, and so on.

```
a(3,4) = 2.0*a(3,4) + 1.0  ! Doubles a(3,4) and adds 1 to
  it.

      do i = 1 , 4
      a(1,i) = a(3,i)  ! Replace row 1 of a by row 3 of a.
      end do           ! Row 3 is unaltered.

      do i = 1 , 3
      a(i,2) = a(i,1)  ! Replace column 2 of a by a column
      end do
```

Name	Result
matmul	Matrix product of two matrices, or a matrix and a vector
dot_product	Scalar (dot) product of two vectors
transpose	Transpose of a matrix
maxval	Maximum value of all the elements of an array, or of all the elements along a specified dimension of an array
maxloc	The location in an array where the maximum value first occur
minval	Minimum value of all the elements of an array, or of all the elements along a specified dimension of an array
minloc	The location in an array where the minimum value first occur
product	Product of all the elements of an array, or of all the elements along a specified dimension of an array
sum	Sum of all the elements of an array, or of all the elements along a specified dimension of an array

An example of matrix and vector multiplication.

```
  program vectors_and_matrices
   integer, dimension(2,3) :: matrix_a
   integer, dimension(3,2) :: matrix_b
   integer, dimension(2,2) :: matrix_ab
   integer, dimension(2) :: vector_c
   integer, dimension(3) :: vector_bc

!  set initial value for vector_c
   vector_c = (/1, 2/)

!  set initial value for matrix_a
!
!              |1 2 3|
!  matrix_a =  |     |
!              |2 3 4|
```

```
      matrix_a(1,1) = 1
      matrix_a(1,2) = 2
      matrix_a(1,3) = 3
      matrix_a(2,1) = 2
      matrix_a(2,2) = 3
      matrix_a(2,3) = 4
    ! matrix_a = reshape((/1,2,2,3,3,4/), (/2,3/))
    ! set matrix_b as the transpose of matrix_a
      matrix_b = transpose(matrix_a)
      do i = 1 , 3
      write(6,*) (matrix_b(i,j),j=1,2)
      end do
    ! calculate matrix product
      matrix_ab = matmul(matrix_a, matrix_b)
      do i = 1 , 2
      write(6,*) (matrix_ab(i,j),j=1,2)
      end do
      vector_bc = matmul(matrix_b,vector_c)
     write(6,*) (vector_bc(j),j=1,3)
      end program vectors_and_matrices
```

Basic array concepts for arrays having more than one dimension

In F, an array may have from one to seven dimensions. The **rank** of an array is defined as the number of its dimension. The rank of an array is specified by using the dimension attribute in a type declaration statement. Therefore the three declarations

```
real, dimension(8) :: a
integer, dimension(3,10,2) :: b
type(point), dimension(4,2,100,8) :: c
```

specify an eight-element rank-one real array a, $3 \times 10 \times 2$ rank-three integer array b, and a $4 \times 2 \times 100 \times 8$ rank-four array c of the derived type `point`.

Array constructors for rank-*n* arrays

In 7th chapter (week) the concept of an array constructor was introduced as a means of specifying a literal array-valued constant. This takes the form

```
(/ value_list /)
```

where each item in *value_list* is either a single value or a list in parentheses controlled by an implied do. For example:

```
( /-1, (i, i=1, 48), 1/)
```

defines an array of size 50 whose first element is –1, whose $(i+1)^{th}$ element is i (for i=1,...,48), and whose 50^{th} element is 1.

An array constructor, however, always creates a rank-one array values. If an array constructor is to be used for arrays with rank higher than one, such as assigning the constructor to rank-two array, the further steps need to be taken to transform it into an array of the correct shape. This is achieved by using the intrinsic function **`reshape`**. For example

```
reshape( (/1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /), (/2, 3/)
```

takes the rank-one real array whose elements are 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 and produces, as a result the 2×3 real array whose elements are

$$\begin{bmatrix} 1.0 & 3.0 & 5.0 \\ 2.0 & 4.0 & 6.0 \end{bmatrix}$$

Notice that the elements of the source array are used in *array element order*; this is one of the few places in F where knowing the array element order is necessary.

Solution of simultaneous linear equations by Gaussian elimination

The solution of a system of linear equations is, perhaps, the most common need in engineering and scientific problems. The Gaussian elimination method is illustrated by a small 3×3 system of simultaneous linear equations:

$$x_1 + 2x_2 + x_3 = 9 \quad (1)$$
$$2x_1 + 3x_2 - 2x_3 = 7 \quad (2)$$
$$4x_1 + 4x_2 + x_3 = 18 \quad (3)$$

Subtracting 2 times equation (1) from equation (2), and then 4 times equation (1) from equation (3) it is obtained the equivalent set of equations:

$$x_1 + 2x_2 + x_3 = 9 \quad (4)$$
$$-x_2 - 4x_3 = -11 \quad (5)$$
$$-4x_2 - 3x_3 = -18 \quad (6)$$

Subtracting 4 times equation (5) from equation (6), it is obtained a further equivalent set of equations:

$$x_1 + 2x_2 + x_3 = 9 \quad (7)$$
$$-x_2 - 4x_3 = -11 \quad (8)$$
$$13\, x_3 = 26 \quad (9)$$

This completes the Gaussian elimination steps. Now it is performed the backward substitution step.

Using equation (9), it is obtained

$x_3 = 26/13 = 2$

Substituting this value for x_3 in equation (8), it is obtained

$-x_2 - 4 \times 2 = -11$

and hence

$x_2 = 11 - 8 = 3$

Substituting these values for x_2 and x_3 in equation (1), it is obtained

$x_1 + 2 \times 3 + 2 = 9$

Therefore

$x_1 = 9 - 6 - 2 = 1$

The solution of the original system of equation is therefore

$x_1 = 1, x_2 = 3, x_3 = 2$

Example: Write a program to read the coefficients of a set of simultaneous linear equations, and to solve the equations using Gaussian elimination.

An initial structure plan for the Gaussian elimination algorithm is as follows:

```
1  perform elimination steps
1.1      if a_ii = 0 then
1.1.1    Return an error message to indicate that
         no solution is calculated
1.1.2    Subtract multiples of the i^th equation
         from all subsequent equations so that
         the coefficients of x_i in the subsequent
         equations become 0

2  perform backward substitution process
```

```
    module linear_equations
    public :: gaussian_elimination
    contains
    subroutine gaussian_elimination(a,b,n,ndim,error)
!   This subroutine solves the linear system Ax = B
!   where the coefficients of A are stored in the array
!      a which is the matrix of coefficient
!   The solution is put in the array b which is
!      the right-hand side
!   Error indicates if errors are found
    real, dimension(ndim,ndim), intent(inout) ::a
    real, dimension(ndim), intent(inout) ::b
    integer, intent(out) :: error
!   local variables
    real :: m, s
    integer :: i,j,k
```

```
!   begin Gaussian elimination procedure
    error = 0
    do i = 1 , n-1
    if (abs(a(i,i)) < 1.e-5) then
!   no solution is possible
    error = -1
    exit
    end if
!   subtract multiples of row i from subsequent rows to
!   zero all subsequent coefficients of x sub i.
    do j = i+1, n
    m =  a(j,i) / a(i,i)
!   do k = 1,n
!   a(j,k) = a(j,k) - m*a(i,k)
!   end do
    a(j,:) = a(j,:) - m*a(i,:)
    b(j) = b(j) - m*b(i)
    end do
```

```
!*****************************************************
!   this part may be extracted after gaining experience
    write(6,'(" elimination stage - ",i2)') i
    do k = 1 , n
    write(6,'(1x,i2,1x,10f8.3)') k,(a(k,j),j=1,n),b(k)
    end do
    pause 'press enter key to continue'
!*****************************************************
    end do
!
!   perform back substitution process
!
    do i = n, 1, -1
    s = b(i)
    do j = i+1, n
    s = s - a(i,j)*b(j)
    end do
    b(i) = s / a(i,i)
    end do
    end subroutine gaussian_elimination
    end module linear_equations
```

```
      program test_gauss
      use linear_equations
      integer, parameter :: ndim = 10
      real, dimension(ndim,ndim) :: a
      real, dimension(ndim) ::b
      integer :: n, error
!     read the matrix of coefficients and right hand side
!        from the file named test_gauss.dat
      open(1,file="test_gauss.dat",action="read")
      read(1,*) n
      do i = 1 , n
      read(1,*) (a(i,j),j=1,n),b(i)
      end do
      close (1)
!     call gauss elimination procedure
      call gaussian_elimination(a,b,n,ndim,error)
!     print output
      write(6,'(2x,"the solution vector is")')
      do i = 1 , n
      write(6,'(1x,"x(",i2,")=",f8.3)') i,b(i)
      end do
      end program test_gauss
```

The solution of the set of equations

$$2x_1 + 3x_2 - x_3 + x_4 = 11$$
$$x_1 - x_2 + 2x_3 - x_4 = -4$$
$$-x_1 - x_2 + 5x_3 + 2x_4 = -2$$
$$3x_1 + x_2 - 3x_3 + 3x_4 = 19$$

and the results produced by the program are

```
elimination stage -  1
  1    2.000   3.000  -1.000   1.000  11.000
  2     .000  -2.500   2.500  -1.500  -9.500
  3     .000    .500   4.500   2.500   3.500
  4     .000  -3.500  -1.500   1.500   2.500
 elimination stage -  2
  1    2.000   3.000  -1.000   1.000  11.000
  2     .000  -2.500   2.500  -1.500  -9.500
  3     .000    .000   5.000   2.200   1.600
  4     .000    .000  -5.000   3.600  15.800
 elimination stage -  3
  1    2.000   3.000  -1.000   1.000  11.000
  2     .000  -2.500   2.500  -1.500  -9.500
  3     .000    .000   5.000   2.200   1.600
  4     .000    .000    .000   5.800  17.400
  the solution vector is
  x( 1)=   2.000
  x( 2)=   1.000
  x( 3)=  -1.000
  x( 4)=   3.000
```

This slide is the end slide of the semester

Seven golden rules of programming

The philosophy depicted in this course can be summarised under 7 golden rules.

1 Always plan ahead: It is invariably a mistake to start to write a program without having first drawn up a program design plan which shows the structure of the program and the various levels of details.

2 Develop in stages: In a program of any size it is essential to tackle each part of the program separately, so that the scale and scope of each new part of the program is of manageable proportions.

3 Modularise: The use of procedures and modules, which can be written and tested independently, is a major factor in the successful development of large programs, and is closely related to the staged development of the programs.

4 Keep it simple: A complicated program is usually both inefficient and error-prone. F contains many features, which can greatly simplify the design of the code and data structure.

5 Test thoroughly: A program must always be testes at every stage and cater for as many situations(both valid and invalid) as possible.

5 Document all programs: There is nothing worse than returning to an undocumented program after an absence of any significant time. Most programs can be adequately documented by the use of meaningful names, and by the inclusion of plenty of comments, but additional documentation should be produced if necessary to explain things that cannot be covered in the code itself. A program has to be written only once – but it will be read many times, so effort expended on self-documenting comments will be more than repaid later.

7 *Enjoy programming:* Writing computer programs, and getting them to work correctly, is a challenging and intellectually stimulating activity. It should also be enjoyable. There is an enormous satisfaction to be obtained from getting a well-designed program to perform the activities that is supposed to perform. It is not always easy, but it should be fun!