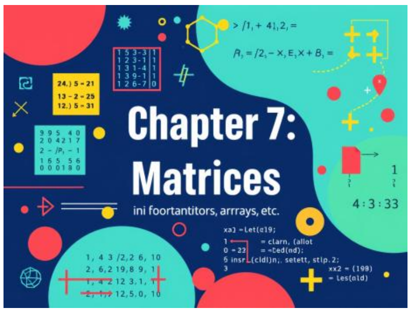
Chapter 7:
Matrices (Arrays)



7.1 Introduction:

In Fortran programming, matrices, or arrays, play a key role in processing data and performing complex calculations. They allow collections of values to be stored in an organized form, facilitating operations on multiple data sets. This chapter will introduce you to the essential concepts of matrices in Fortran, including their declaration, initialization, and common operations you can perform.

We'll also cover array dimensions, manipulation techniques, and how to leverage matrices to solve mathematical and scientific problems. Whether you're working on simulations, data analysis, or advanced algorithms, mastering matrices in Fortran is essential for optimizing your programs and improving their efficiency.

7.2 Definition:

An array is a set of elements of same type, spotted At means of indices whole. THE elements of a painting are tidy according to A Or several axes calleddimensions of painting. In THE paintings has a dimension (Who allow of represent for example vectors in the mathematical sense of the term), each elementis identified by a single integer, but Fortran accepts arrays of up to 7 dimensions, Or each element East designated by a 7 clues.

A one-dimensional array is sometimes called a vector. It can be represented as shape next :

L(1)	L(2)	L(3)	L(4)	……………………		L(n)

- Dimension of painting : 1
- Size of painting : n
- THE L(i), i=1, 2, …, n (must be of even kind)

7.2.1 Statement of the paintings :

As with simple variables, there is the problem of the type of the array, that is to say of its elements. All elements of the array are of the same type. If the name of the array appears in a typical order (REAL, INTEGER, ….) the problem is solved. Otherwise it is that the table appeared in a DIMENSION order, and the first letter lifts all confusion, if this letter is I, I, K, L, M or n THE painting East entire, Otherwise real.

Examples :

REAL, DIMENSION :: A (10), B (15 , 5), C (3, 7, 9)

These statements indicate :

A : is A painting real has 1 dimension, of 10 elements (vectors)

B : is a 2-dimensional real array, with 15 rows and 5 columns 9(matrices)

J is a 3-dimensional real array equal to 3, 7 and 9

7.2.2 Terminology of the paintings :

- **Rank** of a painting : number of his dimensions
- **Extent** (extent) of a painting according to a of his dimensions : number of elements in this dimension
- **Bounds** of a painting according to a of his dimensions : boundaries lower and superior of the clues In this dimension. There terminal inferior by default worth 1.
- **Profile** (shape) of a painting : vector whose THE components are THE expanses of table according to his dimensions ; its size East THE rank of painting.
- **Size** of an array: total number of elements that constitute it, i.e. the product of the elements of vector that constitutes his profile.

(two paintings are said conforming (conformable) if they have THE even profile .)

7.3 declaration of an array:

- There declaration of an array is done using the DIMENSION attribute which indicates the profile of the table, but also possibly the terminals, separated by the symbol " : ".

Examples :

REAL, DIMENSION X(15) REAL, Y DIMENSION (1:5,1:3) REAL, DIMENSION Z (-1:3, 0:2)

THE painting X East of rank 1, Y And Z are of rank 2.

The extent of X East 15, Y And Z have a extent of 5 and 3.

The profile of X is the vector (/ 15 /), that of Y and Z is the vector (/ 5,3 /).The size of the paintings X, Y And Z is 15.

The paintings Y And Z are conformants .

7.4 Building and displaying a table:

To construct a 1-dimensional array, we can list its elements and surround them with (/ ... /) .

Here are some examples:
* (/ 1, 2, 3 /) produces the array [1 , 2 , 3]
(/ (i,i =1,100) /) and (/ (i,i =1,100,2) /) produce the array [1 , 2 , . . . , 100] (the array of odd numbers between 1 and 100) respectively.
* We can combine this as in the example (/ (0 ,(j,j =1,5),0, i=1,6) /) , which makes the table of size 42, made up of 6 times the sequence 0 , 1 , 2 , 3 , 4 , 5 , 0.
* The command **reshape** (X ,(/ m,n /) allows to create from a linear table of dimension *mn* a rectangular table of size $m\times n$, by successively filling the first line, then the second, ...
Thus reshape ((/ ((i+ j,i =1,100),j=1,100) /),(/100,100/)) constructs the addition table of the first 100 integers.

Examples:

```
Program
Implicit none
Real, dimension( 10) :: x
Integer : : i
Do i = 1, 10
x(i) = i**2 ! Assign values to each element of the array
```

```
Dnd do
Do i = 1, 10
Write( *,*) "x(", i , ") = ", x( i ) ! Print the elements of the array
End do
End
```

7.4 RESHAPE FUNCTION:

The **RESHAPE function** in Fortran is used to change the shape of a multidimensional array. This function takes two arguments: the original array and the desired new shape. The original array is rearranged to match the new shape, filling the elements in the order they are stored in memory.

The array's fill when changing its shape depends on the order in which the elements are stored in memory. In Fortran, the default storage order for multidimensional arrays is the column-major convention, also called FORTRAN order. This means that the elements are stored in columns, so that the elements in the first column are stored contiguously, then the elements in the second column, and so on.

When the RESHAPE function is called with a new number of dimensions or a new shape for the array, the compiler calculates the new size and shape of the array. Then, it fills the array elements in storage order, taking the elements from the original array in the corresponding order.

For example, if the original array has the shape (3, 4) and the storage order is FORTRAN, so that the elements are stored in columns, the array can be thought of as follows:

A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2) A(1,3) A(2,3) A(3,3) A(1,4) A(2,4) A(3,4)

If the RESHAPE function is called with the new shape (6, 2), the compiler will fill the new array by taking the elements from the old array in the following order:

A(1,1) A(2,1) A(3,1) A(4,1) A(5,1) A(6,1) A(1,2) A(2,2) A(3,2) A(4,3) A(5,2) A(6,2)

Noticed : If the new shape specified for the array does not match the total size of the original array, a compilation or runtime error may occur.

EXAMPLES:

a = RESHAPE([2.0,3.0,1.0,0.0,-1.0,4.0,-2.0,5.0,2.0], [3,3])

Writing **a = RESHAPE([2.0,3.0,1.0,0.0,-1.0,4.0,-2.0,5.0,2.0],[3,3])** in Fortran creates an array **a** of dimensions 3x3 and stores the elements in it

$$\begin{vmatrix} \mathbf{2} & \mathbf{0} & \mathbf{2} \\ \mathbf{3} & \mathbf{1} & \mathbf{5} \\ \mathbf{1} & \mathbf{4} & \mathbf{2} \end{vmatrix}$$

In this case, the elements **[2.0,3.0,1.0]** correspond to the first column of **a** , the elements **[0.0,-1.0,4.0]** correspond to the second column, and the elements **[-2.0,5.0,2.0]** correspond to the third column. The new array **a** will therefore have the following form:

7.5 Dynamic allocation

7.5.1 **Definition:**

Dynamic memory allocation (or dynamic memory allocation) is a computer process used to reserve memory space in the heap of a running program. This technique allows the creation of variables, arrays, and data structures of unknown or variable size, unlike static allocation, which requires knowing the size in advance.

In programming language, allocatable is a variable type that allows you to declare a variable that can be dynamically allocated. The dynamic allocation operation (allocate) allows you to reserve memory space for this variable, while the deallocate operation allows you to free this space and make it available for other uses.

Here is an example of Fortran 90 code for dynamically allocating one-dimensional arrays using the allocate syntax :

```
program allocation_tab
implicit none
integer , allocatable :: tab(:)
integer : : n, i
write ( *,*) "Enter the size of the array:"
read( *,*) n
allocate (tab(n))
do i = 1, n
tab( i ) = i
end do
write ( *,*) "Contents of the table:"
write ( *,*) tab
deallocate ( tab)
End
```

In this example, we declare an array of integers **tab** with the syntax **integer , allocatable :: tab(:)** . We then ask the user to enter the size of the array using **read (*,*)n** . We then use the dynamic allocation operation **allocate (tab(n))** to allocate memory for the array **tab** with size **n** . We then initialize the array with increasing values from 1 to **n** , before displaying its contents with **write (*,*)tab** . Finally, we free the memory allocated for the array with **deallocate (tab)** .

It is also possible to dynamically allocate multi-dimensional arrays in Fortran 90. Here is an example code for a two-dimensional array:

```
program allocation_dyn_tab_2d
implicit none
integer , allocatable :: tab(:,:)
integer : : n, m, i , j
write ( *,*) "Enter the dimensions of the array:"
read( *,*) n, m
allocate( tab( n,m ))
do i = 1, n
do j = 1, m
tab( i,j ) = i *j
end do
end do
write( *,*) " Contents of the array: "
```

```
do i = 1, n
write( *,*) (tab( i,j ), j = 1, m)
end do
deallocate ( tab)
end program allocation_dyn_tab_2d
```

In this example, we declare a two-dimensional integer array **tab** with the syntax **integer , allocatable :: tab(:,:)** . We then ask the user to enter the dimensions of the array with **read (*,*) n, m** . We then use the dynamic allocation operation **allocate (tab(n,m))** to allocate memory for the array **tab** with dimensions **n** and **m** . We then initialize the array with values equal to **i*j** , before displaying its contents with **write (*,*) (tab(i,j), j = 1, m)** . Finally, we free the memory allocated for the array with **deallocate (tab)** .

Notes :

REAL, DIMENSION :: V(100)

*It is important to note that the size of the memory area reserved for the array or vector **v** depends on the size of the elements in the array. If you declare **v** as an array of integers, each element will typically take up 4 bytes of memory (32 bits) on most architectures, and so the total size of the memory area reserved for **v** would be 400 bytes. If you declare **v** as an array of reals (or floats), each element will typically take up 8 bytes of memory (64 bits) on most architectures, and so the total size of the memory area reserved for **v** would be 800 bytes.*

Examples : tab (100,100)

8 for a 32-bit operating system, the reservation is (100x100x4=40000 bytes

9 for a 64-bit operating system, the reservation is (100x100x8=80000 bytes

7.6 Options for alignment:

ADVANCE='no': Prevents moving to the next line after each element during display. For example, write(*, '(F6.2)', ADVANCE='no') displays the elements on the same line. These formatting options and specification can be used in the write statement to control the display of matrices on the screen in Fortran .

```
Example 1 (Left alignment):
program display_matrix
implicit none
integer : : i , j
integer , parameter:: n = 3, m = 4
 real:: matrix(n, m) = reshape([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0,
12.0], [n, m])
do i = 1, n
do j = 1, m
write( *,'(F6.2)', advance='no') matrix( i , j)
end do
 write( *,*) ! Moves to the next line after each matrix row
end do
end program display_matrix
```

In this example, the elements of the matrix are displayed with a field width of 6 characters and 2 decimal places of precision. The advance='no' option is used to prevent moving to the next line after each element, aligning the elements to the left.

Example 2 (Right alignment):

In this example, the elements of the matrix are displayed with a field width of 6 characters and 2 decimal places of precision. The advance='no' option is used to prevent moving to the next line after each element. Additionally, a string consisting of a space is added after each element to align the elements to the right.

These examples illustrate how to use the advance='no' option in combination with format specifications to align the elements of a matrix to the left or right during display. You can adjust the formats and options according to your specific needs.

```
program display_matrix
  implicit none
  integer : : i , j
  integer , parameter:: n = 3, m = 4
  real:: matrix(n, m) = reshape([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0], [n,
m])
  do i = 1, n
    do j = 1, m
      write( *,'(F6.2, A)', advance='no') matrix( i , j), ' '
    end do
    write( *,*) ! Moves to the next line after each matrix row
  end do
end program display_matrix
```

7.6 Intrinsic Functions for Arrays:

In Fortran , there are several intrinsic functions that can be used to perform operations on arrays. Here are some commonly used intrinsic functions :

1. SIZE(array [, dimension]) : This function returns the size of an array or a specific dimension of an array. For example, SIZE(array) returns the total number of elements in the array, while SIZE(array, dimension) returns the size of the specified dimension of the array.
2. SHAPE(array) : This function returns the shape (dimensions) of an array as an array of integers. For example, SHAPE(array) returns an array containing the sizes of each dimension of the array.
3. RESHAPE(source, shape) : This function reshapes an array using a new shape specified by the shape array. The total size of the source array must match the total size of the resulting array.

4. LBOUND(array [, dimension]) : This function returns the lower bound index of an array or a specific dimension of an array. For example, LBOUND(array) returns the lower bound index of the array, while LBOUND(array, dimension) returns the lower bound index of the specified dimension of the array.
5. UBOUND(array [, dimension]) : This function returns the upper bound index of an array or a specific dimension of an array. For example, UBOUND(array) returns the upper bound index of the array, while UBOUND(array, dimension) returns the upper bound index of the specified dimension of the array.

D. Reduction Operations on Arrays (sum, minimum, maximum, etc.):

Fortran also provides reduction operations that allow you to calculate aggregated quantities from the elements of an array. Here are some commonly used reduction operations :

1. SUM(array [, dimension]) : This function calculates the sum of the elements of an array or a specific dimension of an array.
2. MINVAL(array [, dimension]) : This function returns the minimum value among the elements of an array or a specific dimension of an array.
3. MAXVAL(array [, dimension]) : This function returns the maximum value among the elements of an array or a specific dimension of an array.
4. PRODUCT(array [, dimension]) : This function calculates the product of the elements of an array or a specific dimension of an array.
5. COUNT(array [, dimension]) : This function counts the number of elements in an array or a specific dimension of an array.

In Fortran , the keyword " allocatable " is used to declare arrays whose size can be dynamically allocated during program execution. This allows creating arrays whose size can vary based on the specific needs of the program.

The use of allocatable arrays offers several advantages:

1. Flexibility of size: Allocatable arrays allow defining arrays whose size can be dynamically adjusted. This enables programmers to create arrays of variable size based on the specific requirements of a given situation.
2. Memory efficiency: By using allocatable arrays, memory is allocated only when necessary. This helps save memory by avoiding allocating space for large arrays that wouldn't be fully utilized.
3. Reusing the same variable name: Allocatable arrays can be reallocated with different sizes during program execution. This means that you can reuse the same variable name for different array sizes, which can make the code more readable and modular.

Example:

Here's a simplified example to illustrate the use of " allocatable ":

```
------------------------------------------------------------------------------------------------------------------
program allocatable_example
  implicit none
  integer , parameter :: n = 5
  real , allocatable :: array(:)
  allocate( array(n))
! Using the allocatable array
  array = [1.0, 2.0, 3.0, 4.0, 5.0]
! Reallocating the array with a new size
n = 10
  allocate( array(n))
! Using the reallocated array
  array = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
  deallocate ( array)
end program allocatable_example
program table_sort
  implicit none
  integer , parameter :: n = 3
  integer:: A(n, n) = reshape([91, 22, 4, 56, 7, 9, 13, 8, 82], [n, n])
  integer : : i , j, temp

  ! Sort table A in ascending order
  do i = 1, n
    do j = 1, n-1
      if (A(j) > A(j+1)) then
        temp = A(j)
        A( j) = A(j+1)
        A( j+1) = temp
      endif
    end do
  end do

! Displaying the sorted table
  write( *,*) " Sorted array :"
  do i = 1, n
    do j = 1, n
      write( *,*) A( i , j)
    end do
  end do

end program table_sort
------------------------------------------------------------------------------------------------------------------
```