Handout Seven

April 24, 2006

1 Numeric Arrays : Part Two

So far you have learnt how to declare arrays and you have made use of '`REAL`' arrays to represent matrices in your matrix library module, where the first dimension indexes the 'rows' of the matrix and the second dimension indexes the 'columns'.

```
REAL, DIMENSION(4,5) :: mat1 !** 2D array to represent the matrix "mat1"
```

1.1 Array terminology

Here is a useful glossary of terms associated with arrays that will help you to understand later descriptions and information you may read in other texts.

- **size** - refers to the total number of elements in an array or if a 'dimension' is specified, then it refers to the number of elements in the specified dimension.
- **rank** - the total number of dimensions in the array ie. our class project deals with matrices that are represented by '`REAL`' arrays of rank two.
- **extent** - the number of elements in a particular dimension. For example, in the declaration above, '`mat1`' has an extent of four in its first dimension and an extent of five in its second dimension.
- **shape** - this refers collectively to both the rank and extents of an array. For arrays to have the same 'shape' they must have the same number of dimensions and the same 'exent' in each corresponding dimension.

  So to pass on the 'shape' of an array we need to pass on the number of dimensions and the number of elements in each dimension.
- **conformable** - Two, or more, arrays can be described as conformable if they have the same 'shape' (remember knowing the shape of an array means knowing the number of dimensions and also the extent in each dimension). Therefore to perform the standard fortran operations of addition, subtraction, multiply and divide the arrays must conform.

1.2 Array references

An individual array element can be referenced by supplying a valid index to each of the dimensions in the array. For example '`mat1(2,3)`' would return the value of the element in the second row and the third column of the matrix '`mat1`'. It is often useful to reference a sub-section of an array, for example consider the following bit of code.

```
REAL, DIMENSION(6,6) :: mat !** 2D array
REAL, DIMENSION(2,3) :: submat !** 2D array

mat=getmat(6,6) !** Call your "getmat" function to read in values into "mat"
submat=mat(2:3,3:5) ! ** Set submat to be a subsection of the matrix "mat"
```

The sub-array '`submat`' is declared to be a rank two array with an extent of two in the first dimension and three in the second dimension. The array '`mat`' is declared to be a rank two array with an extent of six in the first dimension and six in the second dimension. Your '`getmat`' function is then called to read in the values of the matrix '`mat`'. Next the sub-matrix '`submat`' is set to be a sub-section of the matrix '`mat`', in particular '`submat`' is set to be equal to the elements referenced by the rows two and three, and the columns three four and five of matrix '`mat`'. NOTE how the 'shape' of both sides of the assignment 'conform'. That is the number of rows in '`submat`' equals the extent referenced in the first dimension of '`mat`' and the number of columns in '`submat`' equals the extent referenced in the second dimension of '`mat`'.

1.3 Array construction

Arrays like other data objects can be constructed in your code at run-time. For example the following bit of code declares a one dimensional array '`aa`' with an extent of ten and then assigns to it the values one through to ten, one in index one, two in index two, three in index three and so on.

```
INTEGER, DIMENSION(10) :: aa !** 1D array

aa=(/1,2,3,4,5,6,7,8,9,10/)
PRINT '("The array aa = ",10i3)',aa
```

NOTE the use of a formatted '`PRINT`' statement used to print out the array in a more appropriate fashion.
The main restriction to this method of array construction is that it can only be used for arrays of rank one. For arrays of higher dimensions the '`RESHAPE`' intrinsic function must be used in conjuction with the above.

1.4 The '`RESHAPE`' intrinsic function

The '`RESHAPE`' function takes as its first argument the 'source' array and as its second argument it takes an array that represents required shape of the returned array. So for example the code

```
INTEGER, DIMENSION(10) :: aa !** 1D array
INTEGER, DIMENSION(2,5) :: bb
aa=(/1,2,3,4,5,6,7,8,9,10/)

bb=RESHAPE(aa,(/2,5/))
CALL outmat(bb)  !*** Your own "outmat" subroutine
```

the output of the above code would be

```
1 3 5 7 9
2 4 6 8 10
```

The important points to note here are that as requested the result matrix has two rows and five columns, the one dimensional array '`aa`' has been copied and reshaped into the result array by filling in the the first column then the second then the third and so on this is referred to as 'column major'. There are actually two optional arguments to the '`RESHAPE`' function we will only look at one of them and that is the keyword argument '`ORDER`'. The default order is '`ORDER=(/1,2/)`' and results in the the 'column major' ordering. The reverse is specifying '`ORDER=(/2,1/)`' and would result in the source array being copied into the result array filling in the result array row by row, this is referred to as 'row major'. Consider the following bit of code.

```
INTEGER, DIMENSION(10) :: aa !** 1D array
INTEGER, DIMENSION(2,5) :: bb
aa=(/1,2,3,4,5,6,7,8,9,10/)

bb=RESHAPE(aa,(/2,5/),ORDER=(/2,1/))
CALL outmat(bb)
```

the output of the above code would be

```
1 2 3 4 5
6 7 8 9 10
```

Exercise One : In a '`handout7/exercise1`' directory write a piece of code to create a rank one array of sixteen '`REAL`' numbers. Construct the array so that is holds the numbers one to sixteen in numeric order, ie. index one holds the number one and index two holds the number to etc. Then '`RESHAPE`' the rank one array into an array to represent the matrix below.

$$\begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

Now use the rank one array and '`RESHAPE`' it into the transpose of the above matrix. Check your answers using your own '`transmat`' function.

Class Project Four (I) : A small addition to your library module. Add another transpose function so that it takes the transpose of a matrix **without** using any '`DO`' loops and does the tranpose operation itself in one line using the reshape function. Call this new version of your '`transmat`' function '`transmat2`'.

1.5 Array Syntax and Expressions

The Fortran arithmetic operators can be used on arrays of the same shape as explained in section 1.4 of handout six. When array arithmetic takes place like this each element in each array-operand operates on the corresponding element in the other array-operand, for example.

$(1, 2, 4, 3) + (2, 5, 2, 1) = (3, 7, 6, 4)$
$(2, 1) * (4, 2) = (8, 2)$

If an array is operated on by a scalar then the scalar operates on all the array elements eg.

$(1, 2, 3) * 3 = (3, 6, 9).$

Subsections of arrays can also be used in expressions as long as each operand in the array expression has the same shape. For example,

```
aa=bb(2:4,6:8)-cc
```

is fine as long as '`cc`' and '`aa`' are arrays of shape '`(/3,3/)`' and '`bb(2:4,6:8)`' is a valid reference.

1.6 Some Array Intrinsics

- '`MAXLOC(<source-array>)`' : Accepts as its argument an array and returns an array of rank one. The number of elements in the returned rank one array is equal to the rank of the source array argument. The returned rank one array then holds the location of the element that has the maximum value in the source-array argument. The first element of the returned array then holds the index in the first dimension, the second element holds the index of the second dimension and so on. NOTE therefore that '`MAXLOC`' always returns an array, **even** if the source-array is of rank one, then the returned array will be a rank one array with only one element. Consider the following example,

$$\mathrm{aa} = \begin{pmatrix} 1 & 2 & 9 & 13 \\ 4 & 6 & 1 & 4 \\ 3 & 2 & 6 & 15 \\ 8 & 8 & 32 & 3 \end{pmatrix}$$

  Then the statement '`ind=MAXLOC(aa)`' will put the array '`(/4,3/)`' into the rank one size two array '`ind`'. This is because the maximum value in the array '`aa`' is '32' and it is in row four and column three. So '`PRINT*,aa(ind(1),ind(2))`' would print the value '32'.

- '`MINLOC(<source-array>)`' : The same as '`MAXLOC`' except that it returns the location of the minimum value.

- '`MAXVAL(<source-array>)`' : Returns the maximum value in the source-array, note the result will be a scalar of the same type as the source array.

- '`MINVAL(<source-array>)`' : Returns the minimum value in the source-array, note the result will be a scalar of the same type as the source array.

- '`SUM(<source-array>)`' : Returns the sum of ALL the elements in the source-array, note the result will be a scalar of the same type as the source array.

- '`PRODUCT(<source-array>)`' : Returns the product of ALL the elements in the source-array, note the result will be a scalar of the same type as the source array.

Class Project Four (II) : In your Matrix library code, if you have not already done so, use the 'MAXVAL' intrinsic function to find the 'maximum' value of a vector in the function you wrote to calculate the infinity norm. HINT you will also need to use the 'ABS' function. Also in your 'power method' procedure to calculate the dominant eigenvalue you were told to use the first index of the 'y' and 'x' vectors to calculate the next estimate of the eigenvalue. ie

```
eigenval=y(1)/x(1)
```

Although this will work for your example it is not strictly speaking a good idea, what if 'x(1)' was zero? Use the 'MAXLOC' function to return the index of the maximum value in 'y' and use this index to calculate the eigenvalue instead of just using the first index. HINT remember that the 'MAXLOC' function will return a rank one single element array for a source-array of one dimension. So you need to declare a rank one single element array to catch the return value from 'MAXLOC'!

2 Dynamic Allocation of Arrays

It will have become clear to you that so far all of your arrays have had to have their sizes 'explicitly' declared in your codes. It has not been possible for you to 'read in' the required size of your arrays bfore declaration and therefore make your program more general and powerful. This, however, can be done in Fortran, arrays can be 'allocated' the memory they require at runtime and when they are no longer required they can be 'deallocated' and the memory freed. You can regard the process of 'Dynamic Allocation' in your code as three separate stages, declaration, allocation and deallocation. Allocatable arrays behave **exactly** the same in your code as explicitly declared arrays and can be passed through argument lists to subroutines and functions.

2.1 Declaration

Allocatable arrays are declared in an almost identical fashion to explicit arrays. There are two differences, no array bounds are given as their size is not known at the declaration stage, their size is assumed by using a colon ':'. Secondly they have the attribute 'ALLOCATABLE' in their attribute list.

```
REAL, DIMENSION(:), ALLOCATABLE :: vec1
REAL, DIMENSION(:,:), ALLOCATABLE :: mat1
```

The first declares a rank one array of type 'REAL'. The second declares a rank two array of type 'REAL'. Note neither have been allocated any memory yet! The rank of the array is of course the same as the number of colons in the 'DIMENSION' attribute. We can not use these two array until somewhere in our code we 'ALLOCATE' them some memory by telling fortran how large each extent of the array using the 'ALLOCATE' command.

2.2 Allocation

In order for any 'ALLOCATABLE' arrays to be used they must first be 'ALLOCATED' with a 'size' so some memory can be reserved to store their values. This is done using the Fortran 'ALLOCATE' statement. You can 'ALLOCATE' the memory for 'ALLOCATABLE' arrays anywhere in your code but this must be done

before you attempt to use the array in your code.

```
ALLOCATE(vec1(16)) !** Allocate space for 16 elements
ALLOCATE(mat1(4,4))!** Allocate space for 4 rows and 4 cols
```

2.3 Deallocation

When you do not need your allocatable array anymore it is **good programming practice** to '`DEALLOCATE`' the array and free the reserved memory. This is done using the '`DEALLOCATE`' command.

```
DEALLOCATE(vec1) !** Free space used by vec1
DEALLOCATE(mat1) !** Free space used by mat1
```

Exercise Two : Some of your earlier codes could have benefited from the use of allocatable arrays, for example the code you wrote to multiply together two matrices. Go back and amend this code to use dynamic memory allocation for all the arrays.

3 Fortran 90 Keyword & Optional arguments.

3.1 Keyword arguments

Up to now, when we call a procedure, the arguments in the calling statement map onto the 'dummy' arguments in the procedure's header argument list by order of appearance. That is the first argument in the calling statement is assigned to the first argument in the procedure header's argument list, the second to the second and so on. ie. in the call

```
CALL addnumbers(a,b,c)
```

with the procedure header for '`addnumbers`' being

```
SUBROUTINE addnumbers(num1,num2,num3) REAL ::  num1,num2,num3
```

When the '`CALL`' statement is reached and program execution moves into the subroutine 'addnumbers' '`num1`' gets passed the value of '`a`', '`num2`' gets passed the value of '`b`', '`num3`' gets passed the value of '`c`'. There is a way in Fortran 90 of making this more flexible and allowing the arguments to be listed in any order. This is done using 'keyword' arguments and they work as follows. In the calling statement argument list the value to be passed is equated to the name of the dummy argument. So the following section of code works identically to the above section,

```
CALL addnumbers(num3=c,num1=a,num2=b) REAL ::  num1,num2,num3
```

with the procedure header for '`addnumbers`' being

```
SUBROUTINE addnumbers(num1,num2,num3)
```

So because we are using keywords the strict ordering of the arguments can be relaxed. There is a strict rule to be remembered here though, as soon as one argument is passed as a 'keyword' argument then ALL subsequent arguments (going left to right) must also be passed as keyword arguments!

3.2 Optional arguments

So far when you use a subroutine or a function, all the data objects that appear in the argument list in the function header must then appear in the argument list of the calling statement. So for example in the subroutine defined as,

```
SUBROUTINE addnumbers(a,b,c,d,e,f)
```

When you call this subroutine the same number of arguments ie. '`a,b,c,d,e,f`' must be included in the calling statement ie,

```
CALL addnumbers(a,b,c,d,e,f)
```

There is a way in Fortran 90 to make arguments optional, that is to define the arguments such that they do not have to be included in the calling statements argument list. This is done by giving them the attribute '`OPTIONAL`' in the argument declaration statement. The only rule here is that ALL optional arguments must appear AFTER all non-optional arguments in the procedure's argument list. It is GOOD programming practice when using 'optional' arguments to pass them through as 'keyword' arguments as it makes the code more readable and clearer to understand. ie given the rather useless subroutine '`addnumbers`',

```
SUBROUTINE addnumbers(num1,num2,num3,add,minus)

  REAL, INTENT(IN) :: num1,num2
  REAL, INTENT(OUT) :: num3
  LOGICAL, INTENT(IN), OPTIONAL :: add, minus

  IF  (PRESENT(add)) THEN
     IF (add .EQ. .TRUE.) num3=num1+num2
  ELSE IF (PRESENT(minus))  THEN
     IF (minus .EQ. .TRUE.) num3=num1-num2
  ENDIF

END SUBROUTINE addnumbers
```

Exercise Three : In a '`handout7/exercise3`' Code up the above mini program and make sure you understand how it works!

The following calls to '`addnumbers`' are all valid but not necessarily sensible as the first call would do nothing.

```
CALL addnumbers(a,b,c)
CALL addnumbers(a,b,c,minus=.TRUE.)
CALL addnumbers(a,b,c,add=.FALSE.,minus=.TRUE.)
CALL addnumbers(a,num3=c,num2=b,minus=.TRUE.)
```