Handout Six

March 21, 2006

1 Fortran 90 Modules

So far you have only been introduced to one type of 'program unit', that is, the main program unit. At first, in the main program unit, you were shown how to write simple executable code and then later how to use a 'CONTAINS' statement to include your own functions and subroutines. There is another form of program unit in Fortran 90 and it is called a 'MODULE'. A module has a range of applications in Fortran 90 and it is good practice to use them. A module can be seen as a related collection of entities that can be made available to one or more other program units. In this case an entity could be data, functions or subroutines. So, for example, you could have a module that contains all the 'functions' and 'procedures' you have written that relate to matrices. This matrix module could then be 'USED' by any other program you write that makes use of matrices, this provides a neat and structured way of including code you have already written into new code.

1.1 The structure of a module

like a program unit a module has a 'typical' structure. This structure is similar to the main program unit.

```
MODULE <module name>

  <USE [other modules]>

  IMPLICIT NONE

  <Specification Section>


  CONTAINS

  <module procedure one>
  <module procedure two>
     :          :
  <module procedure  n >

END MODULE <module name>
```

- A module has its own name like any other program unit.
- A module can also make use of any entities in other modules, so you could have a 'USE' statement between the module header and the 'IMPLICIT NONE' statement.
- A module has its own 'IMPLICIT NONE' statement.
- In the specification part of a module you can declare any data that you want to be 'globally' visible to all the procedures defined later in the module.

- A module can contain functions and subroutines just like a main program unit by placing them between a 'CONTAINS' statement and the 'END MODULE' statement.
- Modules are compiled separately from the main program unit so you should put them in a separate file. You can group more than one module in a single file but in most cases it is sensible to use a separate file for each module. So, for example, you could call the matrix module 'matrix_mod.f90'.

1.2 Making 'USE' of a module

The last section explained what a fortran 'MODULE' is, explained its structure and how to write one in fortran 90 code. In this section you will learn how to 'USE' a fortran module in your programs so you can gain access to its 'SUBROUTINES', 'FUNCTIONS' and data.

Any declared entities in a module that are intended to be used externally can be accessed by Fortran 90's 'USE' statement. The 'USE' statement can be included in any program unit, function or subroutine of a Fortran 90 code. Its syntax can be described in **two** forms.

1.2.1 The simple 'USE' statement

USE <module_name>

This makes accessible all the available entities of the module, so every procedure or item of data declared in the module would be made available. So in an example code this would look like,

```
PROGRAM modtest

  !*** Get access to "fact" and "bisect" in module
  USE moduletest

  IMPLICIT NONE

  REAL :: fact_local
  LOGICAL :: bisect_local=.TRUE.
  INTEGER :: a=5
  REAL :: b=0,c=1

  fact_local=fact(a)
  IF (bisect_local) CALL bisect(b,c)

END PROGRAM modtest
```

Note that the 'USE' statement is included before the 'IMPLICIT NONE' statement.

In some cases this could cause problems, if a local data type in the main program was declared with the same name as an entity declared in the specification section of a module it was using, this would cause a name conflict. For example if we had a module that 'CONTAINS' a procedure called 'fact' and a subroutine, that has a data type declared with the name 'fact', then 'USE' of this module as in the line above will result in a conflict! To get round this problem Fortran 90 uses a re-name symbol '=>' as in;

USE <mod_name>, <new_local_name> => <new_name_mod>[, <new_local_name2 > => <new_name_mod2>]

The small example program below uses the module 'moduletest' and renames the function 'fact' in the module to be called 'factorial' locally. It also renames the subroutine 'bisect' in the module to be called '`bisection`' locally. It does this as both '`fact`' and '`bisect`' already exist as local variables in the main program unit and hence avoids any name conflicts!

```
PROGRAM modtest

  USE moduletest, factorial => fact, bisection => bisect

  IMPLICIT NONE

  REAL :: fact
  LOGICAL :: bisect=.TRUE.
  INTEGER :: a=5
  REAL :: b=0,c=1

  fact=factorial(a)
  IF (bisect) CALL bisection(b,c)

END PROGRAM modtest
```

1.2.2 The '`USE, ONLY :`' statement

It is clear that it would be advantageous if there was a way to 'select' which entities we would like '`USE`' from a module. Not surprisingly there is an easy way of doing this and it involves including the '`ONLY`' attribute inside the '`USE`' statement.

`USE` <module_name>, `ONLY :` <name1>[,<name2>,.....]

So for the above example we could have.

```
USE moduletest, ONLY : fact, bisect
```

Note that the renaming '`=>`' can also be used in the comma separated list, for example

```
USE moduletest, ONLY : factorial => fact, bisection => bisect
```

This means we make available from the module '`moduletest`' only the entities `fact` and `bisect` and, in the process, rename them locally `factorial` and `bisection`.

1.3 Modules used to hold common data

Modules can be used to pass data between program entities. This is done by declaring the commonly used data in the specification section of the module. It is then simply included in the desired procedure or program unit using the 'USE' statement.

```
MODULE commondata

  IMPLICIT NONE

  SAVE
  REAL :: pi=3.14159
  INTEGER :: married=10, age=45

END MODULE commondata

! **************************************************************

PROGRAM example

  USE commondata, ONLY : age, married, pi

 IMPLICIT NONE

 PRINT*,age,married,pi

END PROGRAM example
```

This method of common data transfer between different sections of code should be not be liberally used instead of argument lists. However, it is often the case that many argument lists in the same code can get very long and complicated and full of the same common data elements. It is in cases like this that common data can be passed to the different sections through modules in this way.
Note that following the 'IMPLICIT NONE' statement there is a 'SAVE' statement. This is recommended in Fortran 90 to avoid the data being forgotten by the module! You should always include it in a module when you declare data.

1.4 An Example Code

```
MODULE series_routines  !***** example module *****

  IMPLICIT NONE

CONTAINS

  FUNCTION expand_sine(x) !*** Finds series expansion

    REAL, DIMENSION(:) :: x
    REAL, DIMENSION(SIZE(x)) :: expand_sine

    expand_sine=x-x**3/factorial(3)+x**5/factorial(5)-&
                x**7/factorial(7)+x**9/factorial(9)

  END FUNCTION expand_sine
! *********************************************************
  FUNCTION factorial(n)
  !*** calculates factorials

    INTEGER, INTENT(IN) :: n
    REAL :: factorial, a
    INTEGER :: i

    a=1.0
    DO i = 1,n
       a = a*i
    END DO
    factorial=a

  END FUNCTION factorial

END MODULE series_routines
! *********************************************************
PROGRAM sine_expansion

  USE series_routines, ONLY : expand_sine

  IMPLICIT NONE

  REAL, PARAMETER :: pi = 3.14159265359
  REAL, DIMENSION(20) :: x, series
  INTEGER :: i

  DO i=1,20   !*** Define the range of x values to use
     x(i)=2.0*pi*(i-1)/ 19
  END DO
  series=expand_sine(x) !****  sin(x) up to five terms
  !**** Prints the results to the screen
  PRINT'(2e12.4)', (x(i), series(i), i = 1, 20)
END PROGRAM sine_expansion
```

NOTE in the code that the function '`expand_sine`' uses the expression

```
expand_sine=x-x**3/factorial(3)+x**5/factorial(5)-x**7/factorial(7)+x**9/factorial(9)
```

This is an example of array arithmetic, there is nothing complicated here. Arrays of the **same shape and size** can be added together, subtracted from each other and raised to a power just like scalar numeric data types. When array arithmetic takes place like this each element in each operand operates on the corresponding element in the other operand. This is often referred to as an array element by element operation, for example.

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

If for example '`mat1, mat2 and mat3`' were matrices of the same size and shape then the following would be legitimate operations in fortran 90.

```
mat1=mat2+mat3
mat1=mat2-mat3
mat1=mat2*mat3
```

NOTE the expression '`mat2*mat3`' is NOT the same as normal matrix multiplication i.e. as done by the intrinsic function '`MATMUL`'. Instead each element in '`mat2`' is multiplied by the corresponding element in '`mat3`' and the result stored in the corresponding element of '`mat1`'. If an array is operated on by a scalar then the scalar operates on all the elements in that array eg.

```
(1,2,3)*3=(3,6,9)
```

The above code demonstrates how you can use Modules to build a code in a modular fashion. You can copy this code, and any files needed to compile it, to a directory in your home area by typing

```
cp ~/info/examples/sinmod/* .
```

in the directory you want to copy the files to. Do not forget the dot ('.') at the end to indicate you want the file to be copied to the directory you are sitting in. To compile and run the code you will have to read the next section that introduces the '`Makefile`' at a very basic level.

1.4.1 Compiling more than one file : '`Makefile`'

If you copied the files as directed in the above section you will now have in the directory a file called '`Makefile`'. This file allows you to compile the two source code files and link together the object files (created by the compilation) into a single executable by just using a single command '`make`'. Try it, just type '`make`' and hit [`Return`]. You can look at the file '`Makefile`' in '`pico`'. You do not need to understand any of the main part you just have to know how to change the first section that contains the lines 'main=sin_main' and 'mod1=sin_module'.

```
#########################################################
# REPLACE the main_prg_name below with the name of
# your main program without the ".f90" extension
# REPLACE the module_name below with your module
# name without the ".f90" extension

# PROGRAM NAME (no .f90)
main=sin_main
# MODULE NAME  (no .f90)
mod1=sin_module
#########################################################
cmplr=f90
objects1 = $(mod1).o $(main).o

$(main)   : $(objects1)
        $(cmplr) -o $(main) $(objects1)

$(mod1).o    : $(mod1).f90
        $(cmplr) -c $(mod1).f90

$(main).o  : $(main).f90 $(mod1).f90
        $(cmplr) -c $(main).f90

clean :
        rm -f *.mod *.pcl *.pc *.o $(main) *.inc *.vo *.d
```

The 'Makefile' contains 'required' tabs at the beginning of certain lines so please copy 'cp' this file around instead of typing it in yourself!

This makefile will compile and link any code that has a main program file and a module. At the moment the 'Makefile' is setup for a main program file called 'sin_main.f90' and a module file called 'sin_module.f90'. You will want to compile and link codes that have different names for the main program unit and the 'Makefile'. To do this you must copy the 'Makefile' to the directory containing the main program file and the module it uses. Then change the lines 'main=sin_main' and 'mod1=sin_module' to 'main=<newname1>' and 'mod1=<newname2>'. Where 'newname1' and 'newname2' are your main program file name and your module file name without the '.f90' extension.

1.5 Some reasons for using Fortran 90 modules

- Modules help organise programs into groups of related procedures.
- Modules help keep the main program file at a tidy manageable size.
- Modules allow for better 'data protection' by limiting the scope of main program variables.

- Modules can be used to provide commonly used data to different program units and procedures.

Class Project :: Part Three :

Create a new directory for your matrix library code and copy your class project fortran file into it, this is so you will still have a copy of the old one. Rearrange your matrix fortran source code file into two 'new' files, one main program file and the other a '`MODULE`' file containing all the procedures you have written so far. Call the module '`matrix_mod.f90`' and in addition to the procedures you have already written, add the two following procedures to your '`matrix_mod.f90`' file.

- Write a Function called '`transmat`' to return the transpose of a matrix. The transpose of a matrix $\mathbf{A}$ is written $\mathbf{A}^{\mathrm{T}}$ and is simply $\mathbf{A}$ with the rows and columns swapped round. So row one in $\mathbf{A}^{\mathrm{T}}$ is column one in $\mathbf{A}$ etc.
- Write a Function called '`normtwo`' to return the Euclidean norm, ($\| \mathbf{v} \|_2$) of a vector $\mathbf{v}$. The Euclidean norm is often referred to as the length of a vector and is the square root of the sum of the squares of all the elements in the vector.
- Write a Function to return the Infinity norm, ($\| \mathbf{v} \|_\infty$) of a vector $\mathbf{v}$. The $\| \mathbf{v} \|_\infty$ is simply the modulus of the element in $\mathbf{v}$ with the largest absolute value.

Let $\mathbf{A}$ be an $m \times m$ real matrix with m real eigenvalues. Moreover, assume that $\mathbf{A}$ has precisely one eigenvalue (λ_1) that is 'dominant' (largest in absolute magnitude) with a corresponding eigenvector ($\mathbf{x}$). Then $\mathbf{x}$ and λ_1 can be calculated using the 'power method' for approximating eigenvalues. The 'power method' is an iterative method and calculates a new estimate of the eigenvector $\mathbf{x}^{(n+1)}$ from the previous estimate $\mathbf{x}^{(n)}$.

[Q] How does the 'power method' work?

$$\begin{aligned}
\mathbf{x} &= c_1\mathbf{v}_1 + c_2\mathbf{v}_2 +, ..., +c_m\mathbf{v}_m && (1)\\
\mathbf{A}\mathbf{x} &= c_1\lambda_1\mathbf{v}_1 + c_2\lambda_2\mathbf{v}_2 +, ..., +c_m\lambda_m\mathbf{v}_m && (2)\\
\mathbf{A}(\mathbf{A}\mathbf{x}) = \mathbf{A}^2\mathbf{x} &= c_1\lambda_1^2\mathbf{v}_1 + c_2\lambda_2^2\mathbf{v}_2 +, ..., +c_m\lambda_m^2\mathbf{v}_m && (3)\\
\mathbf{A}^n\mathbf{x} &= c_1\lambda_1^n\mathbf{v}_1 + c_2\lambda_2^n\mathbf{v}_2 +, ..., +c_m\lambda_m^n\mathbf{v}_m && (4)
\end{aligned}$$

The equations $1 \rightarrow 4$ show how repeatedly premultiplying $\mathbf{x}$ with $\mathbf{A}$ builds up the m terms involving the m eigenvectors. Now since the first term on the RHS of (4) contains λ_1, the dominant (largest in absolute value) eigenvalue, as n gets larger this first term will start to dominate the other terms in absolute value. So the RHS will eventually converge to our dominant eigenvector as the first term containing $\mathbf{v}_1$ is providing the largest contribution to the summation. There is a problem, however, that as n gets large λ^n gets very large in fact too large for the computer to accurately represent its value. So, to solve this problem, after each time we premultiply by $\mathbf{A}$ we normalise the result to keep our new estimate of the dominant eigenvector of order one. Remember that dividing an eigenvector by a scalar still leaves the same eigenvector but simply rescaled.

The main iterative body of power method can be written as;

- Take a normalised estimate of the eigenvector $\mathbf{x}^{(n)}$ and calculate $\mathbf{y}^{(n)} = \mathbf{A} \cdot \mathbf{x}^{(n)}$.
- Calculate the new estimate of the dominant eigenvalue $\lambda_1^{(n)} = \mathbf{y}_k^{(n)} / \mathbf{x}_k^{(n)}$, where $k \in \{1, 2, .., m\}$

– Calculate the new normalised $(n+1)^{th}$ estimate of the dominant eigenvector from the $(n)^{th}$ estimate of $\mathbf{y}$

$$\mathbf{x}^{(n+1)} = \frac{\mathbf{y}^{(n)}}{\parallel \mathbf{y}^{(n)} \parallel_\infty}$$

– Repeat the above three steps until a termination criterion has been satisfied. Note that $\mathbf{x}^{(n)}$ means the n^{th} iterative estimate **not** $\mathbf{x}$ raised to the power n.

(a) In your matrix library' `MODULE`' add three Fortran 90 procedures, described in (i),(ii) and (iii) below, that together can be used to implement the 'power method'.

(i) First write a '`FUNCTION`' called '`infnorm`' that calculates and returns the 'infinity' norm of a vector $\mathbf{v} \in \mathbb{R}^m$. The infinity norm, $\parallel \mathbf{v} \parallel_\infty$, is simply the modulus of the element in $\mathbf{v}$ with the largest absolute value.

(ii) Write a '`FUNCTION`' called '`cont`' that accepts **only** the four dummy arguments

'`(y,x,tol,max_iters)`'

and returns '`.FALSE.`' if a given tolerance has been met or a maximum number of iterations has been exceeded else it returns '`.TRUE.`'. Use for the tolerance condition,

$$\parallel \mathbf{x}^{(n+1)} - \mathbf{x}^{(n)} \parallel_\infty < \text{tol}$$

use tol = 0.001.
HINT : Declaring a variable inside a procedure with the attribute '`SAVE`', means that the value of that variable will be retained between calls to that procedure.

```
INTEGER, SAVE ::  count=1
```

declares '`count`' to be initialised with the value '`1`' only on the first call to the procedure it is declared in, from then on it retains its value between calls.

(iii) Write a '`SUBROUTINE`' called '`power`' that has as its argument list;

`(mat,x,tol,eigv,conv)`

Where '`mat`' is the matrix for which the dominant eigenvalue is to be found, '`x`' is to input the initial eigenvector estimate and return the final estimate, '`tol`' is the measure of convergence, '`eigv`' is to hold the returned eigenvalue and '`conv`' is of type '`LOGICAL`' and returns '`.TRUE.`' only if the method converged.

(iv) You will need a function '`mulmatvec(mat,vec) that pre-multiplies a vector by a matrix`'

Use you code to calculate the dominant eigenvalue and corresponding eigenvector of the matrix;

$$\begin{pmatrix} 1 & 5 & 3 \\ 6 & 3 & 5 \\ 2 & 8 & 5 \end{pmatrix}$$

You should have a dominant eigenvalue of 13.063. with a corresponding eigenvector of

$$\begin{pmatrix} 0.6039 \\ 0.8569 \\ 1.0000 \end{pmatrix}$$
