Beginner Fortran 90 tutorial (part 2)

1 Modules

Modules were introduced in Fortran 90, and greatly help with the organization of a single program, and with linking routines across many different programs. They are very versatile, and have many different uses. In this Section, we will learn a few of them.

First, note that a module should be viewed as a library. That library can contain different things, such as a list of important universal constants (if you are writing a program for physics computations for instance), or a list of functions or subroutines that you often use, or a list of variables that are common to many routines in a single program, and that need to be shared by all these routines. The structure of a module is usually of the form:

```
module nameofmodule
```

declarations of variables

```
 contains
```

list of functions and routines

```
end module nameofmodule
```

Any program, function or subroutine that needs to access a particular module usually has, just after its first line (e.g. `program thisprog`, or `subroutine myroutine`), the statement `use nameofmodule`. Here is an example of a module that contains various important constants, and then of a program that uses them.

```
module mathconsts
 implicit none

 real, parameter :: pi = acos(-1.0)
 real, parameter :: e = exp(1.0)

end module mathconsts
```

```
program sillyprog
 use mathconsts

 real :: x

 x = cos(pi)
 write(*,*) 'The cosine of pi is ',x
 x = log(e)
 write(*,*) 'The natural log of e is ',x

end program sillyprog
```

The advantage of doing this rather than declaring π and e in the program `sillyprog` is that we can from now on call this module from *any* program, routine or function we ever write.

Exercise 1: Save the module in a file called `mathconsts.f90` and the program in a file called `sillyprog.f90`. To compile this module with the program, simply type the command `gfortran mathconsts.f90 sillyprog.f90 -o test.exe`. Run the program. Does this behave as you expected?

Modules, as described earlier, can also be used to create a library of the routines and functions you may commonly use. In fact, one of the goals of the first part of this course will be to create a "linear algebra module" that contains all of your linear algebra routines. Next, we will start creating that module, and study the advantage of putting functions and routines in a module rather than separately.

2 Allocatable arrays

In section last week we learned about arrays, and defined them in a *static* way at the start of the program. This means that some memory space has to be reserved at the beginning of the program for the array, and that memory space cannot be modified later. This is usually not a problem if the task at hand is very predictable, e.g. if the program always has to deal with matrices of a predictable size. However consider an example in which you may want to do some data processing (e.g. multiple images, or time-signals), but the data in question varies a lot in size. With this static allocation, the only way to deal with varying size is to reserve upfront the largest possible memory space (to accommodate for any possible dataset size) and later to only use a part of it if the problem is smaller than this maximum size. This method is quite wasteful, and often requires a lot of inelegant commands to deal with the difference between the actual array size and the reserved array size.

This used to be the standard in Fortran 77. From Fortran 90 onward, it has become possible to allocate arrays "on the fly", that is, even after the program has started. To do so, the arrays have to be defined at the beginning of the program as `allocatable`. Once the desired array size is known, we can then create it using the command `allocate`. Once we're done using it, we release the memory space using `deallocate`. Here is an example, in which the program first reads the array size, then creates the array, then reads it in, then calculates its trace, then deallocates the memory space. The "read and allocate" routine, and trace function are both stored in a module.

```
program firstLinAlprog
 use LinAl

 real,dimension(:,:), allocatable :: mat
 real :: x
 character*100 filename

 if(iargc().ne.1) then
 write(*,*) 'Wrong number of arguments (need file name)'
 stop
 endif
 call getarg(1,filename)

 call readandallocatemat(mat,filename)

 x = trace(mat)
 write(*,*) 'The trace of this matrix is ', x

 deallocate(mat)

end program firstLinAlprog

module LinAl
 implicit none

 integer :: nsize,msize
 integer :: i,j

 contains

subroutine readandallocatemat(mat,filename)

 character*100 filename
 real, dimension(:,:), allocatable :: mat

 open(10,file=filename)
 read(10,*) nsize,msize
 allocate(mat(nsize,msize))
 do i=1,nsize
    read(10,*) ( mat(i,j), j=1,msize )
    write(*,*) ( mat(i,j), j=1,msize )
 enddo
 close(10)

end subroutine readandallocatemat

real function trace(mat)

 real, dimension(nsize,msize) :: mat

 trace = 0.
 if(nsize.ne.msize) then
    write(*,*) 'This is not a square matrix, cannot calculate trace'
 else
    do i=1,nsize
       trace = trace + mat(i,i)
    enddo
 end if

end function trace

end module LinAl
```

Exercise 2: Create a file that contains, in the first line, 2 integers (separated by a tab) that will be the number of lines and number of columns in the matrix, and then write the matrix line by line. Separate each element by a tab. Call this, for instance `mymatrix.dat`. Then compile and run this program as usual. What happens? Note how in this case the program actually expects an argument right after calling the executable. To do so, (supposing your executable is called `myprog`) type ./`myprog mymatrix.dat`. What happens then?

To understand this program step by step, note that

- This time the executable expects an argument. This is contained in the `iargc()` command in the main program. That command counts the number of arguments given to the main program. The lines from `if(iargc().ne.1)` ... to `endif` simply say that if the number of argument given to the executable is not 1, (which is the expected number) then the program has to stop. Otherwise, the next command `getarg` reads in the argument, which is the name of the file to use. We shall use more of this later on, and learn of the subtleties of the `getarg` command.
- The program itself is very short! Most of the action happens in the module.

- The module itself is written in such a way that there are many variables global to all the functions and routines of the module. Note how `nsize` and `msize` are declared before the `contains` statement, and therefore are implicitly known by all the functions and routines of the module. Same for `i` and `j`. This avoids having to pass them back and forth between the program and the subprograms, and having to redeclare them every time.

- The array `mat` doesn't really exist until it has been allocated. In the main program, it is merely a pointer to a position in memory space. It is only in the subroutine that this pointer is actually allocated an address, together with all the bits afterwards that are needed to contain the whole array.

Exercise 3: Create a new routine in the module `LinAl` that finds and returns the largest element (in absolute value) of the matrix, as well as its position. Call that routine from the main program, and write a statement to the screen about that element.

Exercise 4: Now modify your input matrix to have a different dimension. Check that the program still works fine.