Beginner Fortran 90 tutorial

1 Basic program structure in Fortran

A very basic program in Fortran contains:

- The program statement (which tells the compiler where the program begins)
- Variable declarations (which tells the compiler what variables will be used, and what type they are)
- Instructions as to what to do
- An end statement (which tells the program where the program ends)

This looks something like the following example:

```
program nameofprogram
implicit none

integer :: i,j,k
real :: x,y,z

x = 3.61
y = cos(x)
z = x + y

i = 3
j = i**2
k = i - j

end program nameofprogram
```

Exercise 1: Write this little program up in a text editor, save the file as `myprogram.f90` then compile the code using the command:

```
gfortran myprogram.f90 -o myprog
```

If it returns any compiling error, try to read what the compiler says and correct the error. If it returns no error, this means that the compiler successfully turned the code into an executable called `myprog`. To run the code, type **`./myprog`** at the prompt. What happens when you do that?

2 Outputting data

In the previous example, your code probably ran but has nothing to show for it – it did not print out any results that you could look at. To let the code actually print out what the result is, you have to tell it to do so.

There are several options for outputting the results:

- To print the results to the screen
- To print the results to a file

In both case, there are several ways of outputting the data, in normal text form, in compressed form, etc. Here, we will just focus on small problems where the data can be printed in text form.

To print something out, you have to use a `write` statement. This statement usually looks like

```
write(X,Y) Z/
```

where `Z` is a list of things to print, `X` tells the code where to print `Z`, and `Y` tells the code in what format to print `Z`. The most basic `write` statement is `write(*,*) Z`, which tells the code to write `Z` in its default format to the screen. For instance:

```
write(*,*) 'The value of x is ',x,' and the value of y is', y
```

will write the sentence *The value of x is*, followed by the actual value of `x`, and then will write *and the value of y is* followed by the value of `y`, to the screen. Note the quotes around the sentences, and the commas separating each of the elements of the list of objects to print.

Exercise 2: In the program above, write out (whichever way you want), the values of `i`,`j`,`k`,`x`,`y` and `z`. Recompile the code, and execute is. Is the output what you expected?

Alternatively, you may want to write this information into a file. To do so, you first need to open the file, then write to the file, then close the file. At the most basic level, this is done by the following commands:

```
open(n,file=filename)
write(n,*) Z
close(n)
```

where `n` is any integer of your choice (larger than, say, 10) that will refer specifically to the file from the open statement to the end statement, and `filename` is the name of your file (defined as a string of characters, which should therefore be written in quotes. As an example, we can write

```
open(10,file='mydata.dat')
write(10,*) 'The value of x is ',x,' and the value of y is', y
```

```
close(10)
```

Exercise 3: In the program above, write all the data to a file instead of the screen. Recompile the code, and execute is. Is the output what you expected?

3 Reading data

In many cases, you want to write a program that can be applicable to different input data without having to recompile it each time. For instance, suppose that we wish to take, as in the code above, a value of x, then takes its cosine, then add the two together and print out the result. But instead of writing the value of x in the code, we want to read it "online", from a prompt, or from a file. To read information, the command is very similar to that of the write statements: they usually take the form of `read(X,Y) Z`. For instance, to read the value of `x` from a screen prompt, and then write it back to the screen, you could add the following command to the code:

```
write(*,*) 'What is the value of x?'
read(*,*) x
write(*,*) 'x is equal to ',x
```

Exercise 4: Modify the code above to prompt the user to input `x` and `i`. Compile and run the code, and then run it on a few examples. Is the result what you expect?

Instead, one may want to read the value of `x` and `i` from a file. Suppose you create a data file called `input.dat` that contains `x` on the first line, and `i` on the second line. To open the file and read the two values, simply add the following section to the code:

```
open(11,file='input.dat')
read(11,*) x
read(11,*) i
close(11)
```

Note that the information in the file must match what the code expects (i.e. it must contain a real number in the first line, and an integer number on the second).

Exercise 5: Modify the code above to read `x` and `i` from a file instead of the prompt. Compile and run the code. Is the result what you expect? Then switch the two lines in the input file, and re-run the code. What happens then?

4 Do loops

Suppose you now want to create a code that repeats very similar (but not necessarily identical) instructions many times. Examples of this would be, for instance, to calculate successive numbers in the Fibonacci sequence, or to evaluate the same function $f(x)$ for many different values of x. A good way of doing this is through the use of do loops. A do loop repeats a set of instructions for a set number of iterations, where the only thing that differs in each repeated set is the value of the iteration number. The do loop structure is

```
 do iter = startiter,enditer
 instruction 1
 instruction 2
    ...
 enddo
```

Here `iter` is an integer that will be varied in increments of 1 from `startiter` to `enditer`. The following program evaluates the first 10 numbers of a geometric sequence:

```
program geometric
 implicit none

 integer :: iter
 real :: a0, r, res

 write(*,*) 'What is the value of a0?'
 read(*,*) a0
 write(*,*) 'What is the value of r?'
 read(*,*) r

 do iter=1,10
 write(*,*) iter, a0
 a0 = a0 * r
 enddo

end program geometric
```

Exercise 6: Write, compile and run this code. Are the results what you expect? How would you modify it to calculate the values of an arithmetic sequence? How would you modify it to prompt the user to tell the code how many iterations to run? How would you modify it to print the results to a file instead of the screen? Do all of these modifications, run the code and compile it. Are the results what you expect?

5 Functions

Functions in Fortran have the same purpose and act in very much the same way as normal mathematical functions: they take in a number of arguments, and return the quantity that is the result of applying the function to its arguments. Note that the quantity returned can be of any different data types, and can either be a number, a character, a vector, a matrix, etc...

There are three ways of writing a function: the latter can be embedded in the original program (in which case, it can only be called from that original program), it can be added after the original program, or it can be put in a separate file (in which case different programs can appeal to the same function).

To understand the difference between the various cases, imagine that we want to write a code that produces a file that can be plotted which contains in the first column values of x in a given interval, and in the second column the corresponding values of $\cos(x)$.

Example 1: This first example contains the function as part of the original program:

```
program plotfunction
 implicit none

 integer :: i
 real :: x
 real, parameter :: xmin = 0.,xmax=10., a=-2.

 open(10,file='myplot.dat')
 do i = 1,100
   x = xmin + xmax*(i-1.0)/(100.0-1.0)
   write(10,*) x,f(x)
 enddo
 close(10)

 contains

 function f(x)
 implicit none
 real :: f,x

 f = cos(x+a)

 end function f

end program plotfunction
```

A few things to note here:

- `xmin`, `xmas` and `a` are defined as parameters of the original program. These are variables that are not meant to ever be changed by any operation in the program. Their values are forever fixed by the declaration statement.
- Note how we did not declare the type of `f` in the bulk of the calling program. This is actually done *within* the function.

Exercise 7: Write, compile and run this code. Plot the results using gnuplot, or any visualization routine of your choice. Is the result what you expect? Now change the values of `xmas`, recompile and re-run the code. Look at the results: are they what you expect. Do the same but this time change the value of `a`. Are the results what you expect?

Example 2: We now consider the alternative program in which `f(x)` is appended in the same file after the end of the program. Your code should now look like this:

```
program plotfunction
 implicit none

 integer :: i
 real :: x
 real, parameter :: xmin = 0.,xmax=10., a=-2.

 open(10,file='myplot.dat')
 do i = 1,100
  x = xmin + xmax*(i-1.0)/(100.0-1.0)
 write(10,*) x,f(x)
 enddo
 close(10)

end program plotfunction

function f(x)
 implicit none
 real :: f,x

 f = cos(x+a)

end function f
```

Exercise 8: Modify your code from Exercise 7 to append `f(x)` at the end of the original program, as shown above. Compile it. What happens? How would you correct the problem ?

This example illustrates that when the function is written outside of the original program

- The program needs to be notified what is the data type of the quantity returned by the function (here, `f`)
- The function needs to be notified of the type of *all* the variables it contains (here, `f`, `x` and `a`).

Exercise 9: Correct your code from Exercise 8 accordingly, until it compiles correctly. Plot the results using gnuplot, or any visualization routine of your choice. Is the result what you expect? Now change the values of `xmas`, recompile and re-run the code. Look at the results: are they what you expect. Do the same but this time change the value of `a`. Are the results what you expect?

In this example, the function `f` does *not* know what the value of `a` is. Because of this, it usually (but not always, that depends on the compiler) just sets this unknown value to 0. To correct the problem `a` must also be passed as an argument of the function.

Exercise 10: Correct your code from Exercise 9 accordingly. Run it a few times with different values of `a`. Does it now behave as it should?

Example 3: Finally, you can also take the last example and put the function in an entirely different file instead of appending to the end of the program. To do so, simply copy and paste the function into a file called, say, `fcosx.f90`. To compile the code with the program and the function in different files, simply type: `gfortran myplot.f90 fcosx.f90 -o myplot`. The compiler will then take any program and function that it finds in the two listed files and attempt to link them to one-another. If successful, it will generate the executable called `myplot`.

Exercise 11: Move the function to a separate file, recompile as suggested, and re-run the program. Does it behave as expected?

The advantage of the last form is that you can now call the same function from an entirely different program, simply by adding the file `fcosx.f90` to the list of files that the compiler of the new code must link.

6 Arrays

In Fortran on can easily construct and manipulate vectors, matrices, and higher-dimensional arrays. The use of vectors/arrays is, at least superficially, very intuitive. By default, the range of indices of a vector is between 1 and the vector dimension (and similarly for matrices). So, in order to access the third component of a vector `v` we write `v(3)`, and to access the coefficient in the second line, first column of a two-by-two matrix `A`, we simply write `A(2,1)`.

Suppose for instance that we want to create two square matrices (one super-diagonal and one sub-diagonal, for simplicity) and add them together. The following program illustrates how one would declare, create, and add the matrices.

```
program addmats
 implicit none

 integer, parameter :: dimmat = 3
 real, dimension(dimmat,dimmat) :: a,b,c
 integer :: i,j

 ! This creates the matrices.
 a(1,2) = 2.0
 do i=2,dimmat-1
 a(i,i+1) = 2.0
 b(i,i-1) = 1.0
 enddo
 b(dimmat,dimmat-1) = 1.0

 ! This adds the matrices a and b
 do i=1,dimmat
 do j=1,dimmat
 c(i,j) = a(i,j)+b(i,j)
 enddo
 enddo

 ! This prints c
 write(*,*) c

end program addmats
```

Note:

- The arrays were defined to be arrays through the declaration statement `dimension` followed by the dimension of the array. This is one way of doing it called ”static allocation”, in which the size of the array is predetermined right at the beginning of the program. This used to be the old Fortran way of doing it, and is still a very useful method for simple problems. Later on, we will learn about dynamic array allocation, in which one can create arrays ”on the fly”.
- Note how comments have been added to the program to make it more readable. This is done with the exclamation mark. Everything on the line after the exclamation mark is ignored by the compiler.

Exercise 12: Write, compile and run this program. What do you notice?

The program as written has two issues: the first is that all the coefficients of `c` end up written in a line, which is confusing, and second, some of them have values that are clearly gibberish.

To understand and correct the first problem, note that the way that Fortran *actually* stores a matrix is the *column-major* order, meaning that it stores, one after the other, first all the elements of the first column of the matrix, then all the elements of the next column, and so forth. So the matrix `c`, which should be equal to

$$\mathbf{c} = \mathbf{a} + \mathbf{b} = \begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 0 \\ 1 & 0 & 2 \\ 0 & 1 & 0 \end{pmatrix}$$

in the program above, is stored and therefore is (or rather, should be) "returned" to the screen as `0 1 0 2 0 1 0 2 0`. To print it out in a more readable form, one can use what is called an "implied do list" in the write statement (a nice feature of modern Fortran):

```
do i=1,dimmat
 write(*,*) ( c(i,j), j=1,dimmat )
enddo
```

Exercise 13: Replace the relevant lines of code in the previous program by these ones, re-compile and re-run it. Has this corrected the first problem?

Let's now deal with the second issue. Clearly, some of the elements are returned correctly, and some are not. The reason for this is that numbers in Fortran are *not necessarily zero upon starting the program!*. One should *never* assume that they are. To correct the problem, we then either have to zero the matrices by hand just after entering the program, or, we can explicitly enter all the coefficients of `a` and `b`, including all of the ones that should be zero.

Exercise 14: Correct the program above using either of the two methods described. Recompile and re-run it. Does it now give the correct answer?

At this point, it is worth mentioning that there are actually much more elegant ways of doing the same things, using intrinsic array manipulation routines in Fortran. For instance, it is possible to multiply a matrix `a` by a scalar `s` simply with the command `s*a`. It is also possible to add the matrices `a` and `b` simply with the command `a+b`. This is often a *much* faster way of manipulating arrays, since good compilers will optimize the array operations in a way that is difficult/cumbersome to do by hand.

Exercise 15: Write a separate program that makes use of the scalar multiplication and matrix addition commands to do the same thing as before. Run and compile it to check that it gives the same result as your original code. Now crank up the dimension of the array `dimmat` to a very large value (10,000 or more), comment out the parts where you write the result out to the screen, recompile both codes, and run each of them by adding the command `time` in front of your execute command. How long does it take for each of the two codes to do the same thing.

As you can see, the intrinsic functions are *significantly* more efficient than writing out the commands by hand – so use them as much as possible. To understand the origin of the difference, see course on High-Performance Computing.

Warning: Note that the compiler interprets these intrinsic commands as applying to the coefficients, not to the matrices. So the command `a+b` creates the matrix whose coefficients are `a(i,j) + b(i,j)`. This is fine, since this is how matrix addition/subtraction works. However, the similar command `a*b` will create the matrix whose coefficients are `a(i,j)*b(i,j)`. This is *not* the result of a normal matrix multiplication. Similarly the command `a/b` creates the matrix whose coefficients are `a(i,j)/b(i,j)`, which is not the result of a matrix inversion of `b` followed by a multiplication with `a`.

To multiply two matrices, there is an intrinsic Fortran 90 function called `matmul`, which simply works by writing `c = matmul(a,b)` (this creates the matrix `c` as the matrix-product of `a` and `b`). No similar intrinsic function exists for matrix inversions, although many Libraries exist that can be of help. See later for this.

7 Subroutines

Subroutines are essentially sub-programs. While functions are usually used to perform rather simple operations (though this doesn't need to be always true), subroutines commonly have hundreds or thousands of lines. They do not have to return an argument, as in the case of functions, but on the other hand they *can* return *many* arguments if needed. The standard structure of the subroutine is

```
subroutine nameofroutine(arg1,arg2,arg3,...,argn)
 implicit none
```

declarations

instructions

```
end subroutine nameofroutine
```

Among the arguments of the routine, there could be input arguments, or output arguments, or mixed ones (that is, arguments that provide the routine with information on input, and are changed on output). As in the case of functions, the routine can either be *contained* in the calling program, or written separately (either in the same file, after the body of the program, or in a separate file). If the routine is contained within the program, then it knows about any variable that is being used by the calling program. If on the other hand the routine is not contained in the program, then any variable it needs *must* be passed as an argument. Remarkably, routines can also take other routines, or other functions as arguments. This is very useful if we want to create, for instance, a routine that uses the bisection method to find out if a user-supplied function has a zero in a given interval.

Here is the code for the routine. It takes as argument the interval over which we want to search for roots, the name of the function we want to test, the number of iterations to perform, and the value of the solution returned, if it exists.

```
subroutine bisect(xmin,xmax,func,sol,iter,err)
 implicit none

 ! On entry this routine must be provided with the interval [xmin,xmax]
 ! over which to search for a solution,
 ! the name of the function fund to search for
 ! and the number of iterations iter to apply.

 ! On exit this routine returns the solution in sol, and its error in err.

 real :: xmin,xmax,sol,func
 integer :: iter
 real :: x1,x2,res,err
 integer :: i
 external func

 x1=xmin
 x2=xmax
 ! Check if there could be a unique solution in interval.
 res=func(x1)*func(x2)
 if(res.gt.0.0) then
 write(*,*) 'There is either no solution or more than one solution in this interval.'
 write(*,*) 'Try again with a different interval.'
 endif

 do i=1,iter
 sol=(x1+x2)/2  ! Calculate the mid-point of the interval
 res=func(sol)*func(x2)
 if(res.gt.0.0) then
 x2 = sol ! The solution is between x1 and sol, shrink x2
 else
 x1 = sol ! The solution is between sol and x2, increase x1
 endif
 enddo

 sol = (x2+x1)/2.
 err = (x2-x1)/2.

end subroutine bisect
```

Note the rather self-explanatory use of the `if`, `then`, `else` statements. Also note the declaration of `func` as an external function. This is needed both in the routine and in the calling program (see below) if `func` is passed as an argument of the routine.

Suppose we now want to use this routine, with the function cosine created and written out in the separate file `fcosx.f90`, we could use the driver program

```
program solbybisection
 implicit none

 real, parameter :: xmin=0.0, xmax=2.0
 integer, parameter :: iter = 10
 real :: sol,err,fcosx
 external fcosx

 call bisect(xmin,xmax,fcosx,sol,iter,err)

 write(*,*) sol,err

end program solbybisection
```

Exercise 16: Create the three files containing the calling program, the function `fcosx` (if this wasn't done earlier) and the subroutine. Compile them all together, as shown earlier, and run the program. In this particular example, the solution should be $\pi/2$. Is it? Try other functions and other intervals. Correct the code as appropriate if things don't go as expected.