L12 – Fortran Programming - Part 4

In all of our Fortran lectures so far we have told our codes what we want them to do with our data in a linear fashion. If we are performing a lot of operations on our data our code can get really messy. There are also occasions where we want to perform the same operations to our data many times. All of this begs for an elegant solution. Never fear, Fortran provides a nice class of solutions: **subroutines**, **functions**, and **modules**. That will be the subject of this lecture!

1. Functions

We have used all kinds of intrinsic functions in Fortran such as **ABS**, **NINT**, **SIN**, etc. But, there may be other functions not supplied in Fortran that you want to have access to. For example, we are always converting our latitudes and longitudes from degrees into radians. A nice function like **DTR** that converted from degrees to radians would be nice. Let's see how we could create such a function.

```
PROGRAM funex
IMPLICIT NONE
REAL(KIND=8) :: lat, DTR

lat = 45.0
lat = DTR(lat)
write(*,*) lat

END PROGRAM funex

!-------------------------------------------------------------!
! DTR - convert degrees to radians
!   Input 'argument' is the angle in degrees
!-------------------------------------------------------------!
FUNCTION DTR(argument)
IMPLICIT NONE
REAL(KIND=8) :: DTR
REAL(KIND=8), PARAMETER :: conv=.01745329251994329444  !pi/180
REAL(KIND=8), INTENT(IN) :: argument

DTR = argument*conv

END FUNCTION DTR
!-------------------------------------------------------------!
```

This is as simplistic of an example as one can define, however it illustrates all of the main points one needs to know to create more elaborate functions:

- The function is named **DTR** and **DTR** has a type. That is, in the function we set **DTR** as a **REAL** number. This means our function will output a **REAL** number.

- The function can use input parameters. In this case, we only use the input parameter we named **argument**. We tell the function that we want to read this variable into the function with the **INTENT(IN)** attribute.

- Our function is defined entirely within the **FUNCTION – END FUNCTION** block. Note that we don't include this definition within the block of the main program. That is, our program is written within the block **PROGRAM – END PROGRAM**, and our function is defined elsewhere.

- Our function uses variables (e.g., the **conv** variable) that are not used by our main program. Hence, we need to define all variables we use inside our function. The main program does not see these variables, so there is no problem if functions have some variable names that are also used in the main program.

- Even though we define **DTR** to be a **REAL** number in our function. We also still have to define **DTR** as a **REAL** number in the main program.

- Note that in the function we call the **latitude** we are reading in by the variable **argument**. In the main program we call it **lat**. This doesn't matter. What matters is that they have the same type!

Functions are useful for cleaning up our codes, but more advanced operations can be defined using **subroutines**.

2. Subroutines

Most programmers tend to have the majority of their coding time invested in writing good subroutines. Subroutines are generally small bits of code designed to do one specific task, but to do that specific task well. The main program tends to be a collection of **CALLS** to these subroutines. To see how we write a subroutine I show an example below that I wrote for calculating vector cross products.

```
      !Cross Product for determining euler rotation pole
      !Angle between two points on a sphere and
      !Distance between two points on a sphere
      !************************************************
      SUBROUTINE cross(v1, v2, v3, w1, w2, w3, u1, u2, u3, alpha, s)
      ! For vectors v and w, where
      ! v = (v1, v2, v3)
      ! w = (w1, w2, w3)
      ! the cross product v x w = x = (x1, x2, x3)
      ! output is u = (u1, u2, u3); where u is the unit
      ! vector in the direction of the vector x
      ! the angle between v and w is given as alpha;
      ! where alpha = arcsin(|v x w|/|v||w|)
      ! s is the distance along the arc between the two
      ! endpoints of vectors v and w, where s=r*theta
      ! alpha and s are only valid for angles <= 90 deg
      IMPLICIT NONE
      INTEGER, PARAMETER :: k=kind(0d0)
      REAL(k), PARAMETER :: pi=3.141592653589793_k
      REAL(k), INTENT(IN) :: v1, v2, v3, w1, w2, w3
      REAL(k), INTENT(OUT) :: u1, u2, u3, alpha, s
      REAL(k) :: magV, magW, magX, arg, rtd, theta
      REAL(k) :: x1, x2, x3
```

```
      rtd = 180_k/pi

      x1 = v2*w3 - v3*w2
      x2 = -v1*w3 + v3*w1
      x3 = v1*w2 - v2*w1

      magV = SQRT(v1**2 + v2**2 + v3**2)
      magW = SQRT(w1**2 + w2**2 + w3**2)
      magX = SQRT(x1**2 + x2**2 + x3**2)

      u1 = x1/magX; u2 = x2/magX; u3 = x3/magX

      arg = magX/(magV*magW)
      theta = ASIN(arg)
      s = magV*theta
      alpha = (ASIN(arg))*rtd

      END SUBROUTINE cross
```

The subroutine looks very similar to our function. The only additional points are:

- We use **SUBROUTINE** and **END SUBROUTINE** to define it.
- We use a number of input arguments – defined with **INTENT(IN)** and also output a number of different arguments – defined with **INTENT(OUT)**. This is not shown in the above example but we can use the same variables in a subroutine to bring variables in and out of the subroutine with the **INTENT(INOUT)** attribute.
- We can also pass **arrays** into and out of subroutines (not shown in the above example).
- As with our functions the **SUBROUTINE – END SUBROUTINE** definition does not occur within the main program.

Using subroutines in a program is slightly different than with functions. The following example shows how to use the cross product subroutine in a program.

```
PROGRAM calc_cross
IMPLICIT NONE
INTEGER, PARAMETER :: k=kind(0d0)
REAL(k) :: x1, y1, z1                !vector 1
REAL(k) :: x2, y2, z2                !vector 2
REAL(k) :: x3, y3, z3                !vector 3
REAL(k) :: alpha, s                  !angle and distance

!Define vector 1
x1 = 0.0; y1 = 5.0; z1 = 1.0

!Define vector 2
x2 = 0.0; y2 = -5.0; z2 = -1.0

!Take the cross product between the two vectors
CALL cross(x1, y1, z1, x2, y2, z2, x3, y3, z3, alpha, s)
```

```
!write out the coordinates of the output vector
write(*,*) "x3 = ", x3
write(*,*) "y3 = ", y3
write(*,*) "z3 = ", z3

END PROGRAM calc_cross
```

The main point is:

- We **CALL** the subroutine to use it.

Now as a point of where do we keep our subroutines. Here we have three basic options:

1) We add the subroutine to the same file as our main program file; adding it in after the **END PROGRAM** statement, or

2) We create an entirely new file that just contains the subroutine, or

3) We add it to a **MODULE** file as described in the next section.

If we chose option number two we have to compile our code in a special way. As it turns out this is exactly the same way we compile modules and is thus described in the next section.

3. Modules

Modules provide a convenient way to package subroutines and functions and other more exotic features we haven't talked about into a single program file that can then be used by other programs.

For example we could create a module file called **mod_constants.f90** that looks like:

```
MODULE constants
IMPLICIT NONE

REAL(KIND=4), PARAMETER ::  Avogadro = 6.022137E23
REAL(KIND=4), PARAMETER ::  G = 6.6726E-11
REAL(KIND=4), PARAMETER ::  c = 2.99792458E8

END MODULE constants
```

If we populated this file with all kinds of constants we use on a regular basis, then it might be useful to have this file around for use in all of our programs. In fact, this is exactly what we can do. We can now create a program file that uses these constants. E.g., let's now create a file called **program_main.f90**:

```
PROGRAM main
USE constants
IMPLICIT NONE
REAL(KIND=4) :: numbr

numbr = Avogadro
write(*,*) numbr

END PROGRAM main
```

Note that in the above program all we had to do was state **USE constants** (where constants represented the name we gave to the module) to have access to these variables.

All we really have to concern ourselves with is compiling these codes. First of all we need to compile the module file:

```
>> g95 –c mod_constants.f90
```

We use the **–c** flag which doesn't produce an executable file but an object file (**mod_constants.o**) and a module file (**constants.mod**) that the main program can use.

Now we need to compile the main program:

```
>> g95 program_main.f90 –o main.x ./mod_constants.o
```

Where the last argument links the object file to the main program.

In the last example we just added a bunch of variables for use in other programs. But we can also add subroutines and functions to a module. The basic syntax looks like:

```
MODULE module_name
CONTAINS

          SUBROUTINE mysub1(arguments)
          IMPLICIT NONE
          ...
          END SUBROUTINE mysub1

          SUBROUTINE mysub2(arguments)
          IMPLICIT NONE
          ...
          END SUBROUTINE mysub2

          ...

END MODULE module_name
```

All we did here is add the **CONTAINS** statement, which allows us to pack the module file full of subroutines. We **USE** this type of module in the same way as the above example.

4. Good Programming Practice – Large Scale Problems

In the above section we saw that we could pack our subroutines and functions into Modules.

The key to good programming is a **modular** approach. That is, we typically write small subroutines or functions aimed at solving specific problems. If we have a good subroutine that solves a problem we encounter over and over then we should use that well-tested subroutine over and over. The best approach is then to pack these subroutines into module files that we can easily use in our codes. In this manner, we can build very large and sophisticated programs from very small pieces. A few keys that I use are:

1) Pack subroutines and functions that are aimed at solving similar types of problems into their own module.

   For example, I have a series of subroutines that involve manipulating latitudes and longitudes points, finding great circle paths between them, distances, angular distances etc. So, I have packed all of these subroutines into a module called **gcarcs** stored in the file **mod_gcarcs.f90**. Hence, any time I am writing a program that involves the need to do calculations involving math on a sphere then I can just link my programs to the **gcarcs** module.

   Another example is that I have a module I call **fdoperators** stored in the file **mod_fdoperators.f90**. This module contains subroutines for calculating finite difference (FD) derivatives with a variety of different options. Hence, any time I want to calculate derivatives I just link my program to the **fdoperators** module and use the subroutines I have already built.

2) For really large programs we may have hundreds of variables. A really good practice is to define a module called **global** in the file **mod_global.f90**. This just defines all of the global variables our program may use. This also allows us to use just a few of those variables in specific subroutines with a statement like

```
USE global, ONLY: variable_1, variable_2
```

3) Create a single file that just drives the flow of the program. The name of this file is arbitrary, but something like **program_main.f90** is a good choice so that we know it is the **main** driving routine. This program then basically consists of a well ordered list of what happens in the code. For example,

```
PROGRAM myprog_main
     USE global
     USE check
     USE funapps
     USE output
     USE input
     IMPLICIT NONE

    !Initialize program by reading the Input
    CALL readinput


    !Check that input info makes sense
    CALL check


    !Do something useful like calculating some derivatives
    DO it = 1,last
        CALL nice_application(inputdata)
        CALL useful_operation(inputdata)
    ENDDO

    !write output
    CALL writeoutput

END PROGRAM myprog_main
```

4) Create input files that define the major program options. Make the input files readable and include at least a rudimentary explanation of what the variables are. Using input files in this manner allows you to run your codes for various options without having to recompile the codes. It also allows you to keep a record of what options were used. The following shows an example of an input file I use for my code to generate models of small scale random seismic heterogeneity.

```
================================================================================
Model  ---------------------------------------------------------------------
rseed         =1                  ! Random Seed Number (positive integer)
dimension     =2                  ! Model Dimension (1,2,3)
nx            =1000               ! Number of X grid points
ny            =500                ! Number of Y grid points
nz            =1                  ! Number of Z grid points
dx            =200.0              ! X-grid increment (m)
dy            =200.0              ! Y-grid increment (m)
dz            =200.0              ! Z-grid increment (m)
Autocorrelation Functions  -------------------------------------------------
acf           =2                  ! (1-Gauss, 2-Exponential, 3-von Karman)
alx           =10000.0            ! X- autocorrelation wavelength (m)
aly           =2000.0             ! Y- autocorrelation wavelength (m)
alz           =2000.0             ! Z- autocorrelation wavelength (m)
Order         =0.0                ! Bessel function order (acf=3)
Perturbation Parameters  ---------------------------------------------------
dVs           =2.0                ! Vs Standard Deviation  (%)
dVp           =2.0                ! Vp Standard Deviation  (%)
drho          =0.0                ! rho Standard Deviation (%)
alpha         =3.0                ! Multiples of STD to keep
Wavenumber Filter  ---------------------------------------------------------
kmin          =0.0                ! Currently Unsupported
kmax          =10.0               ! Currently Unsupported
Input  ---------------------------------------------------------------------
modeltype     =1                  ! (0-Homogeneous, 1-crfl)
modelfile     =./models/crfl.dat
Vs0           =3000.0             ! Vs for homogeneous models (m/sec)
Vp0           =6000.0             ! Vp for homogeneous models (m/sec)
rho0          =2000.0             ! rho for homogeneous models (kg/m^3)
Output  --------------------------------------------------------------------
prefix        =./output/test1     !
status        =1                  ! (0,1 = off,on) show status messages
oformat       =2                  ! 1-Ascii; 2-E3D
otype         =1                  ! 0- output %dV pert.; 1- output actual
ofiles        =0                  ! 0-just final model files; 1-all files



================================================================================
rseed:  Positive integer.  Sets the random seed of the random number generator
        Works such that the same distribution will always be returned for the
        same 'rseed' number.

nx,ny,nz:  These do not have to be powers of 2.  MUST be even numbers.


Order:  Order of Bessel Functions for von Karman type media. Must be set
        between 0.0 and 0.5

otype:  if otype is set to 0 then the %dV perturbation is written and not
        the actual perturbed velocity or density values from the input
        model.
```

5) Create a **makefile** that includes directions for properly compiling your codes. The next section will talk more about this.

6) If you think your code will be distributed widely go the extra step and create a **man page**. The class website contains a hand out on how to prepare man pages.

5. Makefiles

This section is not aimed at telling you how to write **makefiles**. There is a hand out on the web page that describes this. However, many codes use **makefiles** to provide instructions on how to compile them. At the very least you should know how to use these **makefiles** to compile codes you get from other sources. Below is an example of what a makefile looks like:

```
#Makefile for program sac2xy
#-----------------------------------------------------------------------
F90=g95
FFLAGS=-O4
RM=/bin/rm -f
BINDIR=../../bin

all : main

#Compile modules
mod_sac_io.o : mod_sac_io.f90
      $(F90) $(FFLAGS) -c mod_sac_io.f90

#Compile Source-code and link modules
sac2xy : sac2xy.f90
      $(F90) $(FFLAGS) sac2xy.f90 -o sac2xy ./mod_sac_io.o

#Copy executable to appropriate directories
main : mod_sac_io.o sac2xy
      cp sac2xy $(BINDIR)

clean :
        $(RM) sac2xy mod_sac_io.o sac_i_o.mod
```

I won't talk about all the details. But here are the important points:

- I have a variable **F90** which sets which Fortran compiler to use.
- Another variable is **FFLAGS** which sets which compile flags to use during compilation.
- An important variable here is **BINDIR** which is where the final executable will be copied to.
- The primary actions are: (1) all of the Fortran modules are compiled, (2) the main program is compiled and linked with the pre-compiled modules, and (3) the executable gets copied to the **BINDIR** location.

If a file named **makefile** exists as above, then to compile the code all you have to do is type:

```
>> make
```

Occassionaly the file will have another name from **makefile** (e.g., **makefile_mycode**). Then you can compile it by typing:

```
>> make –f makefile_mycode
```

Note that in the above example there are some instructions called **clean**. This is common practice and usually gives directions of how to remove the compiled codes. One can use this by typing:

```
>> make clean
```

6. Homework

1) One of the most important concepts signal processing is that of convolution. For example, in seismology a seismic signal is just the convolution between a source time function, the receiver structure, and the Earth structure (otherwise known as the Green's functions). Hence, having a convolution code handy is a must. For example, if we compute the Green's functions in the Earth for a seismic signal, we can adequately synthesize what is recorded on a seismometer by convolving those Green's functions with a source time and receiver structure function. In this exercise we will create a basic convolution code in Fortran 90.

If we have discrete signals h[n] and x[n] the convolution between them is defined as:

$$y[n] = h[n] * x[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k]$$

Write a code that will read in an arbitrary signal *x*[n] and convolve it with one of a set of pre-defined functions *h*[n].

The predefined functions that should be available are: (1) box car, (2) triangle, and (3) Gaussian function.

The code should output the convolved signal. Also write a C Shell script that will drive the convolution program and produce a plot of the input and output signals.

Your code should be written such that it is easy to read, which means that you should break up the main constituents of the program into subroutines. For example, you may want separate subroutines that (a) read in the data, (b) create box car functions, (c) create triangle functions, (d) create a Gaussian function, (e) performs the convolution, and (f) writes out the convolved signal.