L09 – Fortran Programming - Part 1

1. Compiled Languages

Thus far in this class we've mostly discussed shell scripting. Here we bring a bunch of different utilities together into a recipe – or a script. Once we do that we can execute the script and the job runs. This is a very low level style of programming if you can even call it programming. Note, that this Shell scripting relied almost entirely on programs that were developed by other people. So, how do people develop these programs? The answer lies mostly in compiled languages (although many programmers today are getting away from compiled languages). That is, most of the programs we used were written in a language like **C** or **Fortran**.

Fortran is an example of a compiled language. What does this mean? It means that we write code in human readable format. E.g.,

```
IF (variable == 10 ) THEN
  write(*,*) "The variable is 10"
ENDIF
```

But, our computer does not have the capability to read such code. The shell might, but that takes time. Instead we want to convert the above statement into something that the processor on our computer can understand. Usually, the processor can understand instructions given to it in a binary form, that we call **machine language** or **assembly code**. In the old days, one used to have to write code in these rather obtuse looking machine languages, some people still do if they are seriously interested in making their code run fast. But, eventually some people started writing different codes, called **compilers**, that can convert a language that is easily readable by humans into something that is easily readable by your cpu.

In short, a compiler is like a language translator. It converts the statements you make in a language (a language like Fortran or C) to a language the cpu can speak.

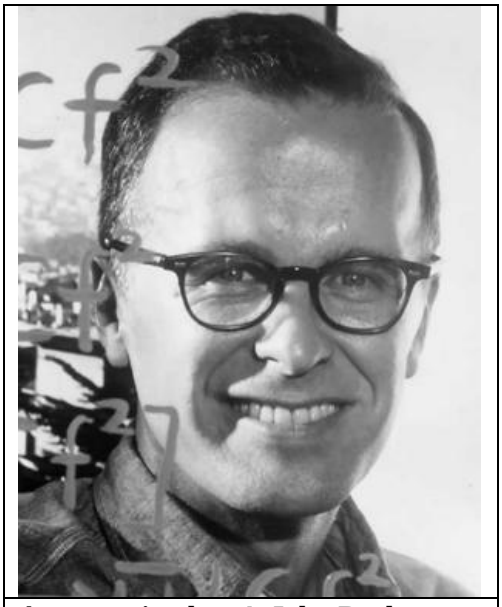
In this class we will focus on the human readable language called **Fortran 90**. The reasons are partly historical. In many of the physical sciences (geophysics, meteorology, etc.) most of the original work was written in Fortran. Hence, we have a long history of Fortran codes in our disciplines. But, Fortran still persists as a primary language in these fields. I have heard many computer scientists express their dismay that Fortran is still used with exclamations of, "I thought that was a dead language." I think it ultimately comes down to the majority of computer scientists these days spending their time on web apps instead of serious problems like simulating the weather, or seismic wave propagation on the global scale, or gravity wave propagation across the universe. Hence, my standby response, "Fortran still produces the fastest executable code." This makes a serious difference when you are talking about days or weeks of simulation time versus months or years when trying to solve the same problem in something like Java. Fortran is also quite simple for trying to solve mathematical problems – that is what it was designed for. It's not fancy. But it is powerful.

A Brief Fortran History

When we talk about Fortran programming it is important to distinguish which version of Fortran we are referring to. You will see several versions (e.g., **f66**, **f77**, **f90**, **f95** etc.) so a brief background is in order. John Backus developed the first Fortran compiler in 1954 – it was the

first high-level programming language! Different versions of Fortran compiler were being developed almost from the start and the need for some kind of standard became immediately apparent. The first official ANSI standard was introduced in 1966 and Fortran 66 (**f66**) was born. The next major standard was a huge improvement to the 1966 version and was released in 1978. Because the standard was agreed upon in 1977 it was termed Fortran 77 (**f77**). This was a major improvement to the standard and included much needed program elements such as **IF THEN ELSE** blocks! The majority of older code you will encounter was written with the **f77** standard (although if you ever get any old seismology related code from Don Helmberger it's likely still written in **f66**).

Since the 1977 standard was released there was a long hiatus before a new standard was released. A friend of mine tells an interesting story of how Sun Microsystems was responsible for the hold up. Alas, a much needed new standard was finally issued in 1990 (**f90**). This included a deluge of improvements including the free form source (in **f77** you had some incredibly annoying restrictions as to where you could write certain parts of the code). There were a number of bugs in the 1990 standard, which were quickly crushed and put into the 1995 standard (**f95**). Reportedly even before the 1995 standard was released new improvements were decided upon, which ultimately gave way to the current 2003 standard. But, at long last we are in a fluid time in the Fortran history and even newer standards are forthcoming.



A computing hero! John Backus.

In these lectures we will refer to programming in **f90**. This is because the major improvements were all made with the **f90** standard. Although, it should be noted sometimes we will talk about things that are actually parts of the newer **f95** and **f2003** standards.

Fortran Compilers

There are several different Fortran compilers on the market today. Some are free and others are commercially developed. Their primary differences are related to (1) which features of the most recent standard are included, and (2) how good are the optimizations (we will discuss what this means in a later lecture). Some common compilers available on our systems are:

Compiler	Developer	Cost	Website
g95		**free**	http://www.g95.org/
gfortran	GNU Fortran	**free**	http://gcc.gnu.org/fortran/
ifort	Intel	**$$$**	http://software.intel.com/en-us/intel-compilers/
pgf90	Portland Group	**$$$**	http://www.pgroup.com/
pathf95	Pathscale	**$$$**	http://www.pathscale.com/

2. Your first Fortran code

So, let's dive in with a really quick example. Open up a new file called **myprog.f90** and type in the following:

```
PROGRAM first
IMPLICIT NONE

write(*,*) "I am a fortran program."

END PROGRAM first
```

This is about as simple of a Fortran code as can be written. The important points are:

- A new program must be given a name with the **PROGRAM** statement.
- The end of the program must be specified with **END PROGRAM**.
- **IMPLICIT NONE** is not required before the main part of the code is written, but it is ***terrible***, ***terrible*** practice to leave it out. More on this later!
- **write(*,*)** says to write something to standard output (a fancy way to say write it out to the screen).

So, how do we run this code? Well as noted above, it cannot be run until we compile it. In this class we will use the **g95** Fortran compiler. So, at the command line type:

```
>> g95 myprog.f90
```

What happens is that a new file called **a.out** is created (on some systems this may be named **a.exe**). This new file which is created is in binary format (you cannot open it up with your text editor and view its contents) and is an executable file. To run the program just type:

```
>> ./a.out
```

Before we move on, let's introduce our first compile flag: the **–o** flag.

```
>> g95 myprog.f90 –o myprog.x
```

The **–o** flag let's us name the executable file that is created. In this case we have a new file called **myprog.x**. Note, Fortran 90 programs require a **.f90** extension or else some compilers assume the code is written with the **f77** standard. The **.x** extension above is just my preferred extension to let me know that I have an executable file. So, there you have it. You are now a Fortran programmer.

3. Numeric Variables

Creating variables in f90 programs is slightly more complicated than in Shell scripts. This is because we have a few different types of variables we can use. The three main types you need to worry about now are:

- **INTEGER** – These are just the set of integers (e.g., 0, 1, 2, 3, -1, etc.). Use integers for counting, but don't for example use integers to do math like 1/3 because the result is not an integer.
- **REAL** – The set of real numbers (for your normal mathematical operations).
- **CHARACTER** – Text strings.

There are other types of variables you may decleare as well (e.g., **Complex**) but most of what you will do revolves around those three types.

In a f90 program we define our variables at the beginning of the program. For example:

```
IMPLICIT NONE
REAL(KIND=8)        :: arg     ! argument for trig functions
INTEGER(KIND=4)     :: J       ! looping variable
```

In this example we named two variables:

(1) The first is named **arg** and is a real number,

(2) The second is named **J** and is an integer.

Note that in f90 programs we start writing **comments** with the **exclamation point!** We will discuss what the **KIND** statement means in Section 5 of this lecture. But first, let's create a new example and see how to assign variables.

```
PROGRAM variables
IMPLICIT NONE
! Define my variables
REAL(KIND=8)    :: A, B            ! some real numbers
INTEGER(KIND=4) ::  J, K           ! just a lowly integer

! Declaring Real Numbers
A = 10.0
B = 20.0
C = A*B - (A/B)
write(*,*) "C = ", C

! Declaring integers
J = 10
J = J + 10
write(*,*) "J =", J
```

```
! Improper use of an integer -- see what happens
K = J + 0.1
write(*,*) "K =", K

END PROGRAM variables
```

The above example shows a very important point:

Never declare a real number without a decimal point in it! For example, we declared **A = 10.0**, Never do this as A = 10 (without the decimal point). Why? Some Fortran compilers are buggy and will give you A=10.2348712410246287 or some kind of similar garbage if you don't put the decimal point there.

Another key point is that in Fortran variable names are ***not*** case sensitive:

```
PROGRAM casesense
IMPLICIT NONE
REAL(KIND=8) :: gH

gH = 10.0

! All these version of 'gh' are treated the same
write(*,*) gH, GH, Gh, gh

END PROGRAM casesense
```

Another fine point to make here is that in the older styles of Fortran programming (f77) one didn't have to declare their variables at the start of the program. There were specific rules one could follow. For example, if you created a variable name that started with an **i** then it was taken to be an integer. **Do Not do this**. Writing code like this is an example of the worst of programming practices. Always declare your variables at the beginning of the code. This will trap many errors (some with very subtle effects) that may go unnoticed otherwise. In fact, writing in the **IMPLICIT NONE** statement at the beginning assures that you will have to, as the **IMPLICIT NONE** statement means that the compiler should expect all variables to be declared.

4. Arithmetic

In the examples above we already showed a slight arithmetic example. Just to clarify our options in Fortran the following are out arithmetic operators:

Operation	Fortran Symbol
Addition	+
Subtraction	-
Division	/
Multiplication	*
Exponentiation	**

There are also many **intrinsic functions** that we can use:

Function	Action	Example
INT	convert real number to integer	**J = INT(X)**
REAL	convert integer to a real number	**X = REAL(J)**
ABS	absolute value	**X = ABS(X)**
MOD	remainder of I divided by J	**K = MOD(I,J)**
SQRT	square root	**X = SQRT(Y)**
EXP	exponentiation [e^x]	**Y = EXP(X)**
LOG	natural logarithm [ln (y)]	**X = LOG(Y)**
LOG10	common logarithm [$\log_{10}(y)$]	**X = LOG10(Y)**
SIN	sine	**X = SIN(Y)**
COS	cosine	**X = COS(Y)**
TAN	tangent	**X = TAN(Y)**
ASIN	arcsine	**Y = ASIN(X)**
ACOS	arccosine	**Y = ACOS(X)**
ATAN	arctangent	**Y = ATAN(X)**
ATAN2	arctangent(a/b)	**Y = ATAN2(A,B)**

Important Note: In Fortran (as in most programming languages) the arguments to the trigonometric functions are expected to be in **radians**. If your arguments are in degrees you need to first convert them to radians! For example: To take the sine of 45°:

```
PROGRAM example
IMPLICIT NONE
REAL(KIND=8) :: argument, answer
REAL(KIND=8) :: pi

pi = 3.141592654    !Define pi
argument = 45.0     !My argument is 45 deg

argument = argument*(pi/180.0)    !Convert from deg to rad

answer = SIN(argument)            !Take the sine

write(*,*) answer                 !Print out the answer

END PROGRAM example
```

One should at this point be excited. Remember how challenging it was to do math inside a shell script. By comparison this is downright easy in Fortran.

5. Numeric Types

In our examples we have specified the naming of our variables in terms of **KIND=4** or **KIND=8**. What this means is that we use numbers that are stored with either **4-** or **8-bytes**. Let's take a look at the following example to see what this means:

```
PROGRAM simpletest
IMPLICIT NONE
INTEGER(KIND=1) :: J

J = 0

DO
  write(*,*) J
  J = J + 1
  IF (J < 0) EXIT
ENDDO

END PROGRAM simpletest
```

This isn't a very complicated program, although we haven't looked at **DO** or **IF** statements yet. The program starts out with the variable **J = 0**, and starts a loop. It adds **1** to **J** at each step. So that **J = 0**, then **J = 1**, then **J = 2**, etc.

Then it makes a test that says **if J is less than 0** let's exit the program.

Run this program and write down that last value of **J** before it becomes < 0:_____

That's some odd behavior for sure. We keep adding a positive number to **J** and eventually **J** becomes negative.

Now let's change the **KIND type** in the above program:

```
INTEGER(KIND=2) :: J
```

Now what was the number we got to before **J** became < 0:_____

Definitely a bigger number. The important point is that we actually have to specify how much memory to use to store the numbers. In the above two examples we used **1** or **2 bytes** per integer number. Recall that there are 8 bits per byte and that a bit is either a 1 or 0. So, a **1-byte** or **8 bit** number might look like:

10010110,

Which would actually represent the number:

$$1\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0 = 150$$

The largest possible 8 bit number is:

$$11111111 = \quad 1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 255$$

However, we haven't considered sign (i.e., **+** or **-**). So we need to use one bit to store the numbers sign. Hence, the largest number we can store in an 8-bit integer is:

$$1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 127$$

The largest **2-byte** (**16 bit**) number possible is 32767, and so on.

Hence, if we are expecting to do some math with large numbers it is important to realize what the biggest number our memory can hold. Real numbers are another story and we need to be concerned with our precision. Remember that a true real number like 1/3 is 0.33333… where we extend off into infinity. Well, a computer does not have infinite memory so we don't have true real numbers in the computers representation, but just an approximation to them.

In general when doing mathematical operations we want to store our real numbers with 8-bytes (**KIND=8**) or else we will start to notice some real precision problems – especially when working with trigonometric functions. However, programs will run faster with smaller storage space per number (e.g., **KIND=4**) as we aren't spending as much time writing numbers into memory. So, we don't always want to go all out and use the largest **KIND** type available to us.

Fortran does provide an easy way to see what the largest available number is for the KIND type you are using by supplying the HUGE intrinsic function. For example, in the example program add the statement:

```
write(*,*) HUGE(J)
```

What is the largest number available for 8-byte integers? How about 8-byte reals?

6. More Information

It is not possible in these tutorials to describe all aspects of Fortran, or even to show you all of the intrinsic functions that are available. Fortunately there are several good web sites available. A couple are listed below:

Numerical Recipes in Fortran: http://www.nrbook.com/institutional/

Fortran Language Reference:
http://h21007.www2.hp.com/portal/download/files/unprot/fortran/docs/lrm/dflrm.htm

Michael Metcalf's Fortran tutorials:
http://wwwasdoc.web.cern.ch/wwwasdoc/f90.html

7. Homework

1) Write a program that allows you to define the latitude and longitude at a point on the Earth's surface and will return the x-, y-, and z- Cartesian coordinates for that point. Assume the positive z-axis goes through the geographic spin axis (N pole). Make sure your coordinate system is right handed.

2) By definition of the dot product we can find the angle between two vectors from the formula:

$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos\theta$$

Write a program that will let you define the x-, y-, and z- coordinates of two vectors in a Cartesian space and find the angle between the two vectors.

3) Combining the programs you wrote above write a program that will allow you to input two latitude and longitude coordinates on the Earth's surface and will return the angular distance between the two points in degrees. Assume the Earth shape to be a sphere with radius = 6371.0 km. Also have the program return the distance (in both km and miles) between the two points on the surface. This distance is the great circle arc distance.