Last time we learned more about the syntax in Fortran programs. Also, we used some of the intrinsic Fortran subroutines to write a small program that gives the third side of a right triangle, as well as the angles at each vertex, given the length of the two smaller sides. In this lesson, I will give you the program I wrote, and explain each part of the program in the context of the five step problem solving guide we used in the previous lesson. Next we will tackle IF and GO TO statements and DO loops. We will finish by writing a small program that prints the factorial (n!) of a given number, n.

1 My Triangle Program

C This program returns the third side and angles at the vertices
C of a right triangle given the length of the two smaller sides
PROGRAM triangle
REAL a,b,c,anga,angb,angc,rad2d
C Initialize variables
C Converts Radians to degrees
rad2d=180./(4.*ATAN(1.))
C Ask user for input of the two smaller sides
PRINT *, 'Please enter the length of the two smaller sides'
PRINT *, 'example: a,b or a <SPACE> b, or a <ENTER> b'
READ *, a,b
C Use Pythagorean Theorem to calculate the third side
c=SQRT(a**2+b**2)
C Use SOHCAHTOA to calculate angles
anga=ASIN(b/c)*rad2d
angc=ASIN(a/c)*rad2d
C Use dot product to calculate angle between a and b
C Note, this would work if not already assuming angb is 90
C So set a dot b to 0. to get 90

\[ \text{angb} = \text{ACOS}(0./((a*b))*\text{rad2d}) \]
C Print third side then the angle

\[ \text{PRINT *, 'The length of the third side is: ', c} \]
\[ \text{PRINT *, 'The angle between a and c is: ', anga} \]
\[ \text{PRINT *, 'The angle between b and c is: ', angc} \]
\[ \text{PRINT *, 'The angle between a and b is: ', angb} \]
\[ \text{PRINT *, 'The sum of the angles is: ', anga+angb+angc} \]
C End of the program

Let’s analyze this program in terms of the five step problem solving process we mentioned in Lesson 2. **Step 1 and Step 2** are to state the problem clearly and describe the input/output, respectively. Both of these steps were basically complete in my statement of the problem, i.e. write a program that *Prints the length of the third side, and angles at each vertex, of a right triangle, (Step 1), given the length of the two smaller sides (Step 2).*

**Step 3** is working the problem by hand. As physics students, we have done this problem many times by hand, and we already know the answer. For example, a right triangle with short sides 3 and 4 will have an hypotenuse of 5. Knowing this gives us a test for the program which is **Step 5**.

As we mentioned before, **Step 4** is the hardest step to complete. However, we can make the task easier if we decompose the problem then refine the decomposition by writing pseudocode. Decomposition of the triangle program above leads to the following:

- Read length of the two smaller sides.
- Calculate the third side.
- Print the results

Refining the decomposition yields our pseudocode.

1. Initialize needed variables.
2. Ask user for input of the smaller sides.
3. Use Pythagorean Theorem to calculate the third side.
4. Use SOHCAHTOA to calculate angles.

5. Print the results.

Compare the pseudocode here with the comments throughout the program. The comparison gives us an idea of how to begin writing our program i.e. Write a program that is only comments made of the pseudocode, then write the code between the comments.

Don’t use you comments sparingly. Personally, I use as many comments as I can. For example if I modify a code written by someone else, to make it run faster say, I first comment out the old code, then add a comment saying something to the effect of C Ayan modified this section to use bisection, July 5, 2005. This will let other users know how the code has developed to its present state and remind you of what specific parts of the program do and how those parts do them, i.e. bisection.

2 Section by Section

Below is a section by section analysis of the triangle program from above. The program starts by giving the purpose in the two comment statements. Next, the program is named triangle using the PROGRAM statement. This name doesn’t follow the Fortran 77 standards, it is more than 6 characters long, but the compiler we are using has no trouble with it. If the compiler had complained, we would have to rename the program to something like triang which is only 6 letters.

Next we define the data type for the variables using the REAL statement. We could have done this implicitly, by naming all of our variable with first letters A-H or O-Z, but to ensure Fortran is using REAL arithmetic and not INTEGER or etc. arithmetic, we define all of our variables in the REAL statement.

Two more comments follow, the first to label that we are initializing needed variables, and the second comment lets us know that rad2d Converts Radians to degrees. We could have used a PARAMETER statement to do the same thing. The statement to initialize rad2d is next, using the fact that ATAN(1.) is $\pi/4$ to get $\pi$.

The next statement is a comment which lets us know the next few lines of code will get the input from the user. Following are two print statements and the read statement. The first print statement asks for the proper input, the
length of the two smaller sides. The next PRINT gives an example of proper input format. This is helpful for the end users who may not know they can separate the length variables as $a, b$ or $a <SPACE> b$ or $a <ENTER> b$. Unfortunately, it assumes the user will press enter to end user input. Maybe another statement to let the user know that <ENTER> must be pressed after b is entered would be helpful.

Another comment that explains the calculation of $c$ uses the Pythagorean Theorem, the calculation of $c$, another statement explaining the calculation of the angles and the calculation of the angles are next.

The next three comments and the calculation of $ang b$ were put in an attempt to make the program more general. As noted, the dot product of $a$ and $b$ is 0, a definition that the angle between the two is 90 degrees.

Six more lines follow, indicating the output will be printed with each PRINT statement giving the meaning of each variable.

Finally the program is ended with a comment and the END statement.

3 PRINT and READ statements

In the triangle program we have used the PRINT and READ statements extensively. A formal introduction to syntax and usage is below.

3.1 PRINT

The print statement is used, as can be guessed, to print variables, messages, percentage of the program complete, and etc.. The print statement has the following syntax:

```
PRINT *, expression list
```
or

```
PRINT k, expression list
```
where $k$ is a label number.

3.1.1 PRINT *, expression list

The first statement represents unformatted output. This is relayed to the compiler by the ‘*’ after the PRINT. Thus, PRINT *, -2.0/4.0 would produce “-0.5”. PRINT *, 'Negative one half is: ', -2.0/4.0 would produce the output, “Negative one half is: -0.5” (two spaces between the colon and ‘-’). In the
previous example our expression list contains 2 items, a character string, “Negative...: ”, and the number, that still needs to be calculated, -2.0/4.0. 

*Note: we can do arithmetic in the print statement. I did this in the triangle program above, anga+angb+angc.*

The character string in this statement is printed to the terminal verbatim. If we had forgotten to add the <SPACE> after the colon, the output would have printed “Negative one half is: -0.5” (only one space between the colon and the ‘-‘). Using PRINT *, allows the program to automatically adjust to negative signs, floating point output, exponential notation output and etc. I prefer unformatted output for the reasons I listed above, but at times we need to have formatted output, which we will come to momentarily.

We can print more than one string at a time. For example, PRINT *, ’The average with error is: ’, averg, ’ +/- ’, errave would print, “The average with error is: 137 +/- 17”. Make sure each item in the expression list is separated by the comma and everything will work fine. In this example, we have 4 expressions, the character string ‘The average with error is: ’, the variable averg, another character string ’ +/- ’, and the final variable, errave.

### 3.1.2 PRINT k, expression list

The second statement represents formatted output. I will continue to use _k_ to represent a space below. This will make it easier to understand the format. Formatted output requires 2 statements, the PRINT _k_, expression list and the format statement labeled by _k_, i.e. **FORM (format specification)**.

Returning once again to the **Average.f** program we can begin to understand the formatted output. Below is the formatted print statement:

```
PRINT 5, averg
FORMAT (1X,’THE AVERAGE IS ’,F5.2)
```

PRINT 5, averg is the PRINT statement. The ‘5’ tells the compiler to format the expression list using the FORMAT statement labeled ‘5’. The format specification has three parts, “1X”, “ ‘THE AVERAGE IS ’ ”, and “F5.2”.

The X Specification will insert blanks into the buffer. Its general form is nX, where n represents the number of blanks to be inserted. Above we used, 1X, which would give us 1 space before the character string is printed. If we had written 10X we would get 10 spaces.

Anytime we enter a character string, also known as Literal Specification into the expression list or format specification list, it is reproduced
verbatim in output (Actually, this depends on the printing device. For us we will print most things to the terminal, more advance FORMAT specifications will be provided as needed when we print to files.) Therefore, the 'THE AVER...' is reproduced exactly.

The **F Specification** will format the output as real or floating point output as opposed to exponential output. Thus, F5.2 means format the variable, `averg`, as floating point or decimal. It could be a double precision variable, but will be forced into floating point output by the specification. Note, printing a double precision variable as floating point will not change it to floating point for future calculations. What does the ‘5.2’ mean? The ‘5’ means use the next five spaces for printing the variable. The ‘.’ means there will be a decimal place, and the ‘2’ means use two of the five allotted space for the decimal places. Thus, the buffer would hold `bb` for output. If the average is 47.5678 or 8.8903 or -3.472, then we would see 47.57, 8.89 or -3.47 printed to the screen.

Note, printing a double precision variable as floating point will not change it to floating point for future calculations. What does the ‘5.2’ mean? The ‘5’ means use the next five spaces for printing the variable. The ‘.’ means there will be a decimal place, and the ‘2’ means use two of the five allotted space for the decimal places. Thus, the buffer would hold `bb` for output. If the average is 47.5678 or 8.8903 or -3.472, then we would see 47.57, 8.89 or -3.47 printed to the screen. Here, the output is rounded, not truncated as discussed earlier. Now, if the average were 137.8023, we would see `##` printed to the screen. This is because we did not allow enough spaces total width, we only allowed 5 for printing. If the average were -23.63, we would again get `##` because the total width has been exceeded. Why not just move the decimal place over? We have forced two decimal places by using the ‘.2’ and this will always take three of our five allotted spaces. If we had anticipated a larger or negative average, we could make the format specification, F7.2, this would properly print averages from 9999.99 to -999.99. Hopefully you can begin to see why I prefer unformatted output, all of this stuff is handled automagically.

We can overcome some of these problems with the **E Specification**. The E specification allows you to print real numbers in exponential notation. This specification is used for very small or very large numbers or when you are uncertain of the magnitude of a number. In contrast to the F specification, a real number will always fit in an E specification field. The specification is ‘Ew.d’. E means use exponential notation output. ‘w’ is the total width to use and ‘d’ is the number of decimal places. Using the E specification, `E10.3` will set up the following buffer content, `s0.00bEb0bb`, where the `s` is a reserved character for the sign of the value and the sign of the exponent and the first number is always a zero i.e. 0.345E+01. In this case, `E10.3`, the E specification uses 7 of the ten spaces. If we want to to know average 137.8023 to 2 decimal places, we would have to have E12.5. The output would be, 0.13780E+03. As a general rule we can see if we use the E specification,
we should make the total width, $d+7$, $d$ being the number of desired decimal places.

One other specification, the **I specification**, can be used in the format statement. The I stands for integer, and thus, I3 would allow -99 to 999 to be printed whereas I2 would allow -9 to 99 to be printed. Again we have to have some idea of the order of magnitude of the output before we can format all situations properly.

Once we understand the FORMAT specifications and get used to how they work, they can be really handy, especially when printing output to a file, which we will return to later. Say we want to print the average of three different data sets on the same line and we wanted them formatted exactly the same. Then our program might contain something like:

```
... 
PRINT 5, averg1,averg2,averg3
FORMAT (1X,'The averages are: ',3E12.5)
... 
```

As with the n in the X specification, the ‘3’ before the ‘E12.5’ lets the program know that there are 3 variables that will be printed with the same format of E12.5. This can be used with the I and F specifications as well i.e. 3F5.2, 10I2.

We can also have mixed format statements, i.e.

```
PRINT 5, i,averg,lnnj
FORMAT (1X,'The loop value is: ',I2,' The average is: ',F7.2,' The natural log is: ',E10.3)
```

I have accidentally snuck in a continuation character on you. The ‘a’ in the third line is in column 6 and thus is read by the compiler as a continuation character. We could have used b, c, 1, *, or . for the character. It is wise to be consistent though, if you have twenty 26 continuation lines, use a-z to label each, though you could continue them all with a. Note, it doesn’t matter where I split the line to make the continuation, I split it here at column 72. I could have put the whole character string ‘The natural log is:’ on one line, but the compiler doesn’t care. It sees the continuation and reads it as such and goes on, no matter where the continuation occurs.

### 3.2 READ

Read statements, as the name implies, will read in data from either a file or the default user input device. Nowadays the standard input device is the
keyboard; in the not so distant past it was punch cards!

Read statements also take two different forms. We will continue to use the unformatted read in this lesson, and I will discuss formatted statements when we learn about opening, reading and writing data files.

### 3.2.1 Unformatted READ

As with the unformatted PRINT, unformatted reads will handle all types of data. For example, `READ *, a, b, c, d` will read in four variables. a and b could be integers while c is a real and d is a character string. When reading input from the keyboard, the program will wait until all variables have been read in. each variable can be separated by either a comma, a space or by hitting enter after each variable is entered. After all variables have been entered, a final enter is necessary for the program to continue.

### 4 IF Statements

We are now ready to discuss IF statements. An IF statement allows the user to make comparisons throughout the program. In the `Average.f` program we used the IF statement below:

```fortran
IF (x.NE.0.0) THEN
```

This example shows the basic construct of all IF statements. First we have the IF command then in parenthesis we have a logical expression and finally we have what we want to do if the logical expression is true. A more general construct would be:

```fortran
IF (logical expression) executable statement
```

An example will be helpful. Say we compute the average of a data set and if the value is negative we will set the average to zero, whereas if it is positive we will multiply it by 10. We can do this with a couple of IF statements. The program might look something like this:

```fortran
... IF (averg.LT.0.0) averg=0.0 IF (averg.GE.0.0) averg=averg*10. ...
```
Here we have checked the logical condition, stated in words, “IF averg is Less Than 0 the new value averg equals 0”, or “IF average is Greater than or Equal to 0 the new value of averg equals the old value of averg multiplied by 10”. There are 6 relational operator that can be used in the logical statement they are preceded and ended with a period i.e. ‘.EQ.’. The relational operators are listed below.

1. .EQ. or == EQual to
2. .NE. or /= Not Equal to
3. .LT. or < Less Than
4. .LE. or <= Less than or Equal to
5. .GT. or > Greater Than
6. .GE. or >= Greater than or Equal to

There are also 5 logical operators we could use in the logical statement:

1. .NOT.
2. .OR.
3. .AND.
4. .EQV.
5. .NEQV.

A logical expression may contain simple relations as above, but we can also make them more complicated i.e. .NOT.(A.LT.15.4).OR.kt.EQ.ISUM; meaning, if it is not true that A is Less Than 15.4 or it is true that kt is EQual to ISUM then...

.EQV. and .NEQV. are for comparing logical data types. If we had defined endprg=.TRUE. and notend=.FALSE. we cannot us the logical expression (endprg.EQ.notend). This is an invalid statement. That is where .EQV. and NEQV. come in. We would have to write (endprg.EQV.notend) to get a valid comparison because we are comparing two logical data types.

If we wanted to do several operations if a condition is met, we can use the IF THEN construct. This time, we want to print an error message and stop the program if the average value is negative. We modify the program above to be:
IF (averg.LT.0.0) THEN
   PRINT *, 'The average is negative, exiting'
   STOP
END IF
IF (averg.GE.0.0) averg=averg*10

The STOP command tells the program to stop all calculations and exit. A stop is inserted by the compiler before the end statement automatically, so if the program encounters a STOP it will do exactly as it would before exiting the program where the END statement has been used.

Back to the IF structure. To let the program know we have more than one executable statement, after the logical expression we add THEN. This is similar to an IF THEN statement in logic.

Another way to write the above code would be to use an IF THEN ELSE structure. Since the value of averg is either positive or negative, assuming zero is positive, we could have written:

IF (averg.LT.0.0) THEN
   PRINT *, 'The average is negative, exiting'
   STOP
ELSE
   averg=averg*10
END IF

Now we have combined the two if statements into one IF THEN ELSE statement.

If we had three comparisons that needed to be made we can use an IF THEN ELSE IF statement. Say we want to multiply the average by 10 only if it is less than 10 and subtract 10 if it is greater than or equal to 10, we can do that by:

IF (averg.LT.0.0) THEN
   PRINT *, 'The average is negative, exiting'
   STOP
ELSE IF (averg.LT.10.) THEN

10
averg=averg*10.
ELSE
averg=averg-10.
END IF

4.1 Looping with IF GO TO

The While loop is an important structure for repeating a set of statements as long as a certain condition is true. The while loop can be achieved using the IF GO TO construct we used in the Average.f program. The code in that program is below:

... READ*, x
1 IF (x.NE.0.0) THEN
sum = sum + x
count = count +1
READ*, x
GO TO 1
END IF
...

The program asks the user to enter the value x. The next statement, labeled 1, checks to see if the value entered is not equal (.NE.) to 0.0. If it is not equal to 0.0 then, 1) make the new value of sum equal to the old value of sum plus the value of x just entered, 2) make the new value of count equal to the old value of count plus one, 3) read another value of x from the user and 4) Go to the statement labeled 1, the if statement. If the condition is not true, i.e. x=0.0 then skip to the first step after the END IF. If we had used the IF THEN ELSE structure, the ELSE would have been evaluated. The same holds for each type of statement we have introduced, except more conditions would be searched before making the final decision.

We will need the If construct when we write our factorial program.

5 DO Statements

DO statements are Fortran’s way of doing things repeatedly without using a GO TO statement. In some circles, the GO TO statement is looked down
on. It can lead to very badly written programs, that is all I will say about. To do calculations in an iterative fashion, we use the DO loop construct. The general structure of the DO is below:

```
DO k index=initial,limit,increment
    ... CONTINUE ...
```

There is a DO, the Fortran command, a label number k, and the index that will go from initial to limit in the stepsize increment. If we leave off increment the default is to make stepsize equal to 1. The statement labeled by k is what is done after the loop finishes. This is usually a continue statement, but can be any other executable statement. Another (more common) variation of the Do Loop is as follows:

```
DO index=initial,limit,increment
    block of statements
    END Do
```

Suppose we wanted to take the sum of all the numbers from 1 to 50. We now have two ways to do this, the IF GO TO statement from above and now the DO statement in this section. An example of each will be helpful:

**While loop solution**

```
... sum=0
    number=1
10 IF(number.LE.50) THEN
        sum=sum+number
        number=number+1
    GO TO 10
END IF ...
```

**DO loop solution**

```
... sum=0
    DO number=1,50
        sum=sum+number
    End Do
...
Note, the DO loop is shorter because the program will automatically repeat the loop, so there is no need for the GO TO statement. Also, the value of number automatically changes during each step in the loop, thus eliminating the number=number+1 statement. Finally, at the end of the loop, we evaluate the statement labeled 10, which just continues the program control to statements below.

A more detailed summary of the DO loop is below.

1. The index of the DO loop must be a variable, but it may be either real of integer.

2. The parameters of the DO loop may be constants, variables, or expressions and can also be real or integer types.

3. The increment can be positive or negative but it cannot be zero.

4. A DO loop may end on any executable statement that is not a transfer, an IF statement, or another DO statement. The CONTINUE or END DO statements are executable statements that were designed expressly for closing a DO loop. I strongly encourage the consistent use of the END DO statement to indicate the end of the loop.

Execution of the DO loop follows these rules:

1. The test for completion is done at the beginning of the loop, as in a While loop. If the initial value of the index is greater than the limit and the increment is positive, the loop will not be executed i.e. DO 10 I=5,2 will immediately evaluate statement 10, no looping will be done.

2. The value of the index should not be modified by other statements during the execution of the loop. We can use the index as a multiplier, answ=answ*index or divisor, answ=answ/index or exponent, answ=x**index or etc., but executing index=index+1 would result in an error message.

3. After the loop begins execution, changing the values of the parameters will have no effect on the loop.

4. If the increment is negative, the exit from the loop will occur when the value of the index is less than the limit.
5. Although it is not recommend, you may branch out of a DO loop before it is completed. The value of the index will be the value just before the branch. (If you need to exit the DO loop before it is completed, you should restructure the loop as a While loop to maintain a structured program.)

6. Upon completion of the DO loop, the index contains the last value that exceeded the limit.

7. Always enter the DO loop through the DO statement so that it will be initiated properly.

8. It is invalid to use a GO TO statement to transfer from outside a DO loop to inside the DO loop.

9. The number of times that a DO loop will be executed can be computed as \([(\text{limit} - \text{initial})/\text{increment}] + 1\). The parentheses represent a truncation or dropping the fractional part of the quotient. If the value is negative the loop is not executed. i.e. DO 35 k=5,83,4 would be executed \(((83 - 5)/4) + 1 = (78/4) + 1 = 20\) times. The values of k would be 5,9,13,...81.

6 SELECT CASE

Another statement developed for the fortran 90 compilers is the SELECT CASE statement. The SELECT CASE statement takes a integer as an argument and executes statements according to different ‘cases’ of that integer. An example might help you here:

\[
\text{Select Case(case expression)}
\text{Case(case selector)}
\text{block of statements}
\text{Case(case selector)}
\text{block of statements}
\text{..}
\text{..}
\text{..}
\text{Case Default}
\text{block of statements}
\]
In this block of statements ‘case expression’ is a variable whose variation will determine which block of statements have to be executed. The ‘case selector’ refers to the value of the variable defined at the beginning of the SELECT CASE statement. A typical case statement would be as follows:

```vbnet
Select Case(month)
Case(1,3,5,7,8,10,12)
    days = 31
Case(4,6,9,11)
    days = 30
Case(2)
    IF(MOD(year,4 == 0)) days = 29
Case Default
    days = 28
End Select
```

This block of statements select the month of the year and tell you how many days there are in the month. The CASE SELECT statement is very useful when you have to use and/or calculate data according to which slab they fall into. Things like Income Tax calculation and grade assignment can be done very easily with this statement. If you want to specify a range of values of the variable for which a certain case holds we use the argument (a:b) for the command CASE, whence it will hold for the range a to b. We can also label blocks of SELECT CASE commands by giving it a name. The command would look like:

```vbnet
Average: Select Case(code)
    .
    .
End Select Average
```

7 Your next task

1. Write a program to find the largest of three numbers. See if you can extend the program to find the largest of five numbers (or for that matter, any set of numbers).
2. Write a program to calculate the Factorial of a number, (n!), and print the output. Remember to follow the 5 step problem solving procedure. Ask
yourself these questions while thinking about a solution: What is the factorial function? Can n take on all values? Can we calculate 5! correctly? 30!? 100!?

3. Write a code to assign grades to students in a class. (For once you have the liberty to choose your own grade!!). Try using the SELECT CASE command. If g77 does not work for your machine try using ifort in its place.