Intrinsic Functions

- A fortran compiler has many built-in or intrinsic function for standard mathematical operations.
- Appendix A in your book has a complete list and Table 2.2, page 33 has an abbreviated list..
- The trig functions are standard - for example

  `sin(x)` `cos(x)` `tan(x)` `asin(x)` `sinh(x)`

  - Here `asin` is the $\arcsin$ (i.e., $\sin^{-1}$) and `sinh` is the hyperbolic sine.
  - Note that each function has an argument enclosed in ( ) which is an angle in radians.
  - Note that `cos (pi/ 3)` is $\cos 60^\circ$ not `cos (60)`, assuming of course that `pi` has been appropriately defined.

- The natural log, $\log_{10}$, e, and square root function are defined by

  `log(x)` `log10(x)` `exp(x)` `sqrt(x)`

- If we want to compute $\log_2 x$ then we must convert this to the natural log (or $\log_{10}$), i.e., $\log_2 x = \ln x / \ln 2$.
- Note that the natural log is not $\ln$
- Note that there is no built-in function for $\frac{1}{x}$ like on your calculator.
- As an example, consider the quantity

$$4e^{.5} + \sin 90° - \ln 2.79$$

$$\texttt{4.0} * \texttt{exp}(\texttt{0.5}) + \texttt{sin}(\texttt{pi}/\texttt{2.0}) - \texttt{log}(\texttt{2.79})$$

  where `pi` has been appropriately defined.
- There are many more intrinsic functions or procedures which we will introduce as we need them

Variables

- Most of the time we will perform a calculation and assign the value to some variable (which has been declared in a type statement). Alternately, we may want to define a variable as a parameter (fixed forever in the program) such as `pi`.
- Of course fortran has rules for naming variables.
  - must begin with a letter
  - other characters may be letters, numbers or underscores
  - must be ≤ 30 characters
- You can not name a variable a name that already means something in fortran. For example, you can't name a variable `sin` since it is an intrinsic function.

- I have a rule for naming variables - The name must be meaningful!
- As an example, consider the following two lines of code for calculating the area of a circle. Which one do you think is easier to follow?

```
a = 5.0
b = pi * a **2
```

or

```
radius = 5.0
area_circle = pi * radius**2
```

The Assignment Statement

- We have already seen that the assignment statement just assigns a value to a variable.
- However there is one assignment statement which may seem confusing at first. Consider the statements

```
a = 3.0
```

- This statement assigns the value 3 to a.

```
a = a + 5.0
```

- This statement says take the current value of a (which is 3) and add 5 to it; a is now 8.

```
a = a / 4.0
```

- This says take the current value of a (8) and divide it by 4. a is now 2.

A Simple Do Loop for Repetition

- Many times we will want to perform a calculation repeatedly.
- For example, if we want to add the first n integers then a strategy would be to
  - initialize the sum to 0
  - repeatedly add the next integer to the sum until you reach n
- Do loops allow us to easily repeat a section of code.
- Here we will only investigate counter-controlled do loops
- Syntax for the `do` construct (counter-controlled)

```
do    control variable = initial value, final value, increment
      statements
end   do
```

- The counter control, initial value, final value and increment must be integers

- The increment is optional; if omitted it is assumed that the increment is 1
- Negative increments are allowed.
- The initial or final values can be zero
- How does it work? Consider the statements

```
do    i = 2, 10
    ...
end do
```

1. The counter control `i` is set to the initial value, here to 2
2. `i` is then checked against the final value (here 10) to see if it is $\leq$ the final value (assuming final value is positive)
   - If less than or equal to the final value, proceed to step 3
   - If greater than the final value, terminate loop (go to next statement after `end do`)
3. all statements between the `do` and the `end do` statements are executed

4. the increment is added to `i` (here the increment is 1 so that i $=$ i $+1$)
5. Return to step (2)

Example

```
integer ::  sum_integers
integer ::  i
    sum_integers = 0
    do i = 1, 5
      sum_integers = sum_integers + i
    end do
```

- `i = 1` - sum_integers $= 0 + 1 = 1$
- `i = 2` - sum_integers $= 1 + 2 = 3$
- `i = 3` - sum_integers $= 3 + 3 = 6$
- `i = 4` - sum_integers $= 6 + 4 = 10$
- `i = 5` - sum_integers $= 10 + 5 = 15$
- the loop is terminated when `i = 6` since this value is greater than the final

value

- Note that for readability we have indented the commands inside the do loop

An example of a do loop with a negative increment

Suppose that you want to program the relationship

$$a_{i-1} = a_i/2\,, \quad i = 5, 4, \ldots, 1, \quad a_5 = 100$$

```
real ::  a_i

integer ::  i

   a_i = 100.0

   do i = 4,1,-1

     a_i = a_i / 2.0

   end do
```

- `i` is first set to 4, `a_i` computed as 100./2.=50.
- `i` is then incremented by -1, i.e., it is 3 and `a_i` is computed to be 50./2.=25.
- The loop is repeated until the value of the counter is < 1

A Simple Conditional

- Often we need to test to see if a particular condition has been met. For example, if our error is less than some tolerance.
- Fortran provides several conditional statements for this purpose.
- Now we look at the simple `if` statement where we test a condition and have only one alternative. The alternative can consist of a single statement or several statements.
- Syntax for case when we only want to perform one statement if the condition is satisfied

```
if ( condition ) statement
```

Example

```
if ( a < 2)   b = a *4
if ( error < tolerance )   stop
```

- Syntax for the case when we want to perform several statements if the condition is satisfied

```
if (   condition )  then
     statements
end    if
```

```
if ( error < tolerance )   then
      print *, " method has converged"
      stop
    end if
```

An `IF ELSE` Construct

- Often when we test we want to do one thing if the condition is satisfied and another if it is not; i.e., we have 2 alternatives. In this case we can simply add an `else` to our `if then` construction

```
if (   condition ) then
         statements
    else
         statements
end if
```

- For example, if we want to take the square root of a if $a \geq 0$ but if a is negative then we want to take the square root of the absolute value of a, we would have the following `IF ELSE` construct

```
if ( a >= 0.0 ) then
        a = sqrt(a)
    else
        a = sqrt( abs (a) )
end if
```

Symbols for Logical Expressions

less than	$<$	`.lt.`
less than or equal to	$<=$	`.le.`
great than	$>$	`.gt.`
greater than or equal to	$>=$	`.le.`
equal to	$==$	`.eq.`
not equal to	$/=$	`.ne.`

- You can use either the symbol or the text syntax. For example if you want less than you can use either $<$ or `.lt.`
- If we wanted to test if a is less than 4

```
if ( a < 4.0   ) then
```

- If we wanted to test if a is greater than 2

```
if (   a > 2.0 ) then
```

- What if we want to combine these two expressions to check that $2 < a < 4$?

Compound Expressions

and	`.and.`
or	`.or.`
not	`.not.`

- To check that $2 < a < 4$

```
if ( a < 4.0   .and.  a > 2.0 ) then
```

- To check that the error is less than or equal to the tolerance or the number of steps (say n) is greater than some maximum number of steps

```
if ( error <= tolerance .or.  n > max_steps ) then
```

Printing our Results to the Screen

- After we do some calculations we want to output the results.
- Typically we will either output the results
  - to the screen
  - or write them to a file which we can open and look at later.
- If we don't have too much output, then we can simply print to the screen.
- We can either print text or the value of some variable
- To print text to the screen we must enclose the text in either single or double quotes

```
print *, " the method has converged "
```

- Note that `print *` means to print to the screen
- We can print the stored value of a variable to the screen by

```
print *, variable name
```

- If we want to print two variables, say `approximation` and `error` then we put them in a list and separate them by a comma

```
print *, approximation, error
```

- We can combine these two into one statement; for example to print out the final error (call it `error`) after the method has converged to an acceptable answer we type

```
print *, " the method has converged; the error is ", error
```

- These are all called unformatted writes. We can also specify the format we want to use to write out a variable; e.g., how many decimal places to include, etc. We will return to the print statement later.

Reading Input from the Screen

- If you recall, the first day of class you downloaded a code which summed the first n integers. The program queried the user to enter the value of n. You simply typed it when asked and hit return.
- To do this, we can simply use the `read` command and tell the compiler we want to read the data from the terminal window.

```
read *, n
```

- Again the syntax `read *` means to read from the screen just like `print *` writes to the screen .
- When the program is executed, execution is halted until this value is read in. It will NOT prompt you to do this.
- Consequently, if you are reading from the screen, you should always put a write statement before this to tell the user that he/she will be asked to input a value.

For example, if the user needs to input the number of integers from the screen, then you should include the lines

```
print *, " enter the number of integers you want to sum"

read *, n
```

Debugging Codes

- Writing a program often takes less time than actually debugging it - i.e., finding all the errors.
- Errors are of two basic types :

1. Compile time error
   - errors that violate syntax rules such as:
     * typos which result in a violation of fortran rules
     * we inadvertently violate some rule such as forgetting to declare a variable
2. Runtime or Execution errors
   - dividing by zero
   - errors in input/output
   - referencing an entry in an array that doesn't exist
   - logical errors - usually the hardest to find

- Remember that just because a code compiles and runs doesn't mean it's right.

- First, let's look at some simple codes which have errors in them to see the actual error messages and try to debug the codes. We will look at the codes `debug1.f90`, `debug2.f90`, `debug3.f90` and `debug4.f90` now to try to get them to compile.

Accessing SCS Computers Remotely

- You can compute on the SCS machines remotely - from a computer at your home or other locations on campus.
- You will need to use the `ssh` command to do this.
- The command `sftp` is useful for moving files from you class account to your personal computer.
- We will now try to log on to another classroom machine using the `ssh` command.

Homework

- Open up the file `quadratic_formula.f90` (which of course implements the quadratic formula for finding the roots of a quadratic polynomial) and attempt to find the errors so that the program compiles. (I think there are 3 compile errors)
- The code `quadratic_formula.f90` has a logic error in it. Find it and verify the code gives the correct answer for the quadratic polynomial that is specified in the code.
- Add read statements to `quadratic_formula.f90` to enter the coefficients of the quadratic $ax^2 + bx + c$ from the terminal instead of setting their values in the actual program. For example, you might say

```
print *, "enter the coefficient of x squared"
read *, a
```

- Try your program on the quadratic $x^2 + 4$. What happens and why?

- Add a conditional to check if the discriminant $b^2 - 4ac < 0$; if it is, print out a statement indicating that no real roots are found.
- The final version of your code should be submitted as homework. See Homework I.1