Introduction to Scientific Computing with Fortran 90 – ISC3313

Course Description: This course introduces the student to the science of computations. Algorithms for standard problems in computational science are presented. The basics of the object-oriented programming language Fortran are taught to facilitate the student's implementation of algorithms.

If you make a "C-" or better in this course you can use it to satisfy the FSU Computer Competency Requirement.

Check out the syllabus on website for details.

Course Objectives: At the conclusion of the course, the students will be able to

- identify the components of scientific computing;
- identify standard problems in scientific computing;
- implement basic algorithms for standard problems in computational science using the programming language Fortran 90;
- write, debug, and verify computer codes;
- output results of computer simulations in a meaningful manner.

Grading Policy: The student grade for the course will be based upon homework, projects, a midterm and a final exam. This work is weighted as follows:

- Homework/Classwork - 50%
- Projects - 15%
- Midterm Exam - 15%
- Final Capstone (Individual) Project - 20%

Expectations: Before each class you should go through the notes for the upcoming class so that you can get more out of each lecture. After each class you should review the notes to solidify the concepts introduced, complete any classwork which you were unable to finish during the lecture period and work on assigned homework. Please take advantage of office hours. If you have a short question, then emailing is fine. Be warned that sometimes debugging a computer code can take longer than you think, so start homework early!

Instructor: Janet Peterson, 444 DSL, jpeterson@.fsu.edu

Office Hours: M 11-12, W 9-10, other times by appointment

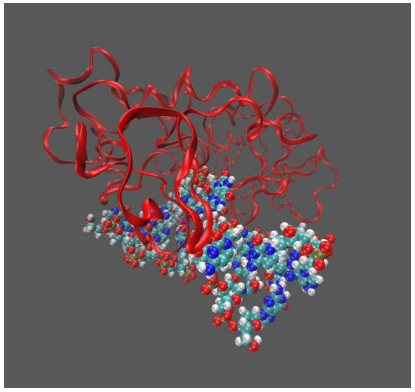
Teaching Assistant: Isaac Lyngaas

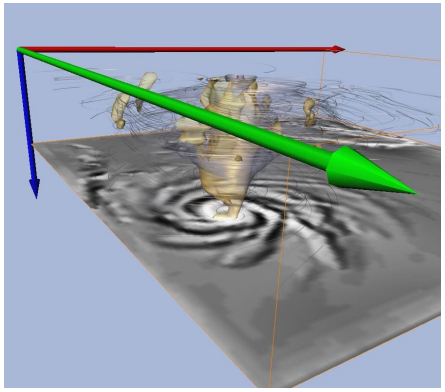
Coursework postings: Website: http://people.sc.fsu.edu/∼peterson/

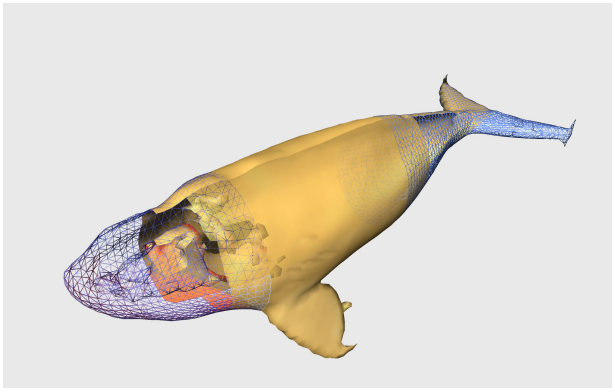
Late Homework Assignments. You are allowed to turn in one homework and one Computer Laboratory assignment up to one week late during the semester with no questions asked. Other assignments turned in late will have their value reduced by 5% daily up to one week late; after one week no late assignments will be accepted. Exceptions to these rules will only be made in the case of illness, etc. which can be properly documented as dictated by the University Attendance Policy

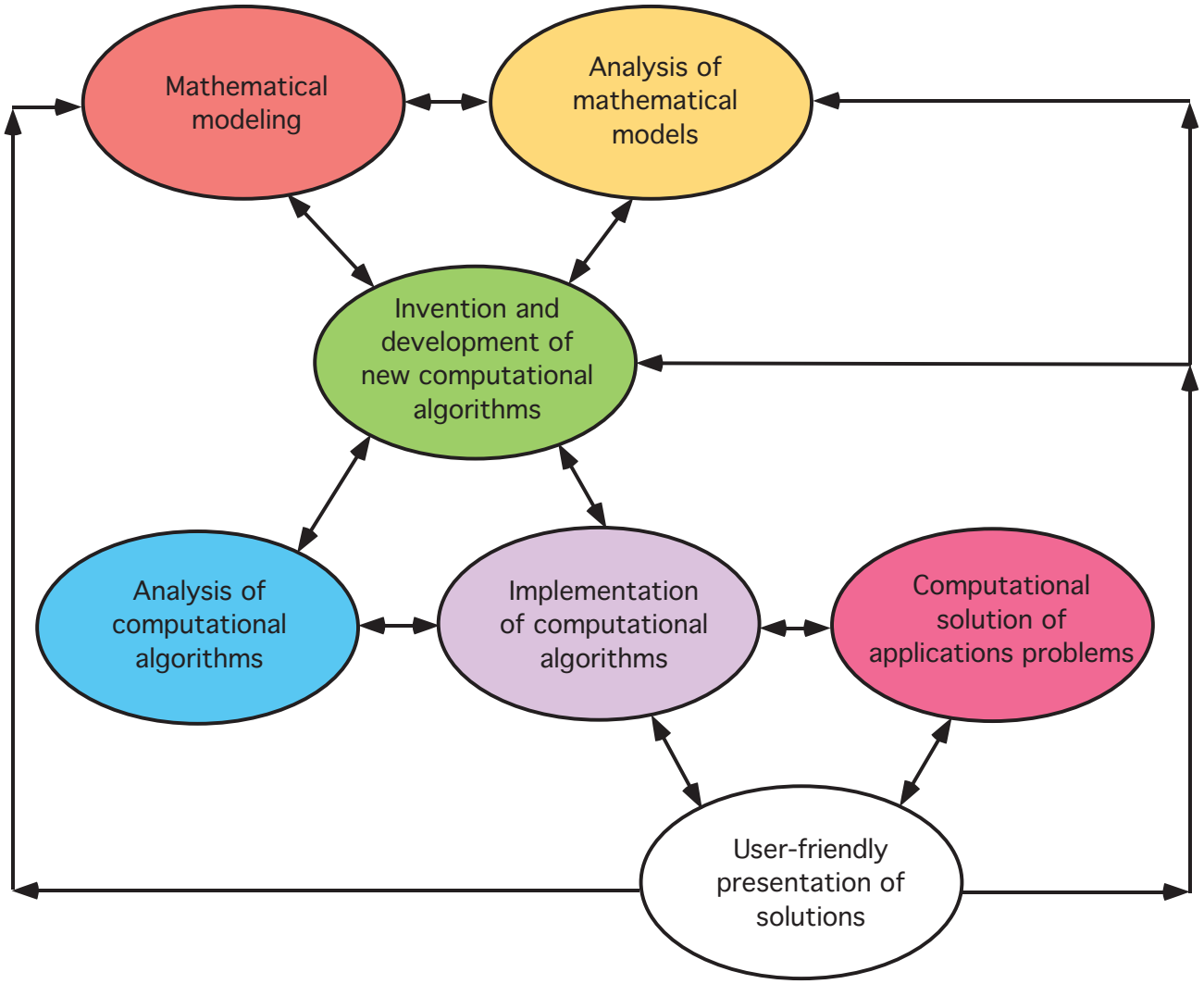
WHAT IS SCIENTIFIC COMPUTING?

Scientific Computing (or computational science) is doing whatever it takes to solve scientific and engineering problems using computers.









The universe of scientific computing/computational science

In this course . . .

- We are mainly interested in the implementation of computational algorithms which can be used for scientific models
- We will be using the programming language fortran 90 to implement algorithms

You will not be required to understand the science where the problems come from (although it makes it more interesting) and you only need a bit of calculus such as knowing about derivatives and the concept of integration. We will review the math as needed.

- We will learn the basics of fortran 90 in the context of some basic problems in computational science such as
  - finding values by iteration (try, try again)
  - using random numbers generation

– approximation of integrals $\int_a^b f(x)\,dx$

– solving a single nonlinear equation e.g., find x such that $x = \sin x$

– interpolating or fitting data e.g., find a line $y = mx + b$ fitting data

– vector and matrix operations e.g., solve $A\vec{x} = \vec{b}$

– solving differential equations $\frac{dy}{dt} = e^{-\sqrt{t}}, \quad y(0) = y_0$

- We will learn to use gnuplot as a tool for visualizing our results

Talking to Computers

- A computer always does exactly as you tell it
- Instructions for computers are in basic machine language that tells the hardware to do things like move a number stored in one memory location to another location, or do simple binary arithmetic.
- Almost no computational scientist really talks to a computer in a language the computer understands.
- To talk with the computer we will first open a terminal window. The terminal runs a shell. These commands get translated to basic machine language which the user does not have to know.
- A shell is a name for a command-line interpreter; i.e., a place where you enter a command for the computer to obey. Another term for shell might be user interface. You can think of these shells as the outer layers of the computer's

operating system. Multiple copies of the same shell can be running on the same machine with one or several users.

- CSH or bash are commonly used shells
- The operating system is a group of instructions used by the computer to communicate with users and devices, to store and read data, and to execute programs.
- The operating system is itself a group of programs that tells the computer what to do in an elementary way. It views you, other devices, and programs as input data for it to process. The operating system is like an office manager.
- The reason for this complicated system is to make life easier for the user by letting the computer and its operating system take care of the tedious work and let you communicate with the computer in a language closer to your everyday language.
- Common examples of operating systems are Unix, OSX, Windows.
- We will use Unix on the terminal window on these computers.

For example, suppose in the terminal window we type the Unix command

```
rm  myfile
```

where `myfile` is the name of some existing file and `rm` is the command to remove a file. The shell searches the filestore to find the file containing the program `rm` and then requests the Unix kernel to execute the program on the named file `myfile`. When the process is completed the shell returns the UNIX prompt "%" to the user, indicating that it is waiting for further commands. Of course this happens almost instantaneously.

Computer languages

- The basic classes of languages are high-level interpreted languages and high-level compiled languages.
- Examples of high-level interpreted languages are symbolic algebras such as Mathematica, Maple, Python and the early programming language Basic.
- Examples of high-level compiled languages are C/C++, Fortran, Java (actually Java is considered semi-compiled)
- When you submit a program to a computer in a compiled language, the computer uses a compiler to process it. The compiler is another program that treats your program like a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language.

- If you make a mistake in your program which violates one of the rules or has a command which is not in its built-in dictionary, then the compiler will issue error messages and the program will fail to compile.

- Once all the errors are resolved, the compiler turns the Fortran text into instructions appropriate for that machine.

- The translated statements ultimately form an executeable code that runs when loaded into the computer's memory.The resulting executable code can be run on ANY machine of that type.

Steps in running a code

1. Use an editor such as `gedit` to create a fortran file with the extension `.f90`
2. Open a terminal window and compile the code using `gfortran`. You must be in the same directory as your code (or indicate where the file is)
3. Debug the code, i.e., remove all compile-time errors; this make take several tries
4. Once the code compiles, execute the code

Compiled Language History

1957	Fortran I		
1966	Fortran IV		
1972		C	
1978	Fortran 77		
1986		C++	
1991	Fortran 90	ANSI C, GNU C	
1996		C++ Release 3	Java
2006	Fortran 2003		Java 1.5

Fortran 90

- Why aren't we using Fortran 2003? - no compilers available
- Fortran is probably the easiest of the three languages (Fortran, C/C++, Java) to learn
- Properties
  - not case sensitive
  - increasingly object-oriented
  - very efficient
  - easy array operations
  - multi-dimensional arrays allowed
  - modules
  - polymorphism, function overloading, etc.
  - lacks some capabilities of C++

- all fortran 90 files should have the extension .f90
- if your file has the extension .f then the compiler assumes it is a fortran 77 file

On which computers can you do your homework?

- Class accounts exist for everyone enrolled and you log in using your FSU ID and password.
- Computers you have access to with your class account that have Fortran 90 compilers
  - Classroom computers
    - can be used when you are sitting in front of the machine
    - can be accessed remotely by the ssh command (more on this later)
  - Other computers
    - computers next to elevator on 4th floor Dirac
- If you want to use your personal computer, then there are downloadable Fortran 90 compilers (both Macs and PC's)
  - If you want information about this, please see TA

– If you are having trouble downloading this, see TA

Setting up your work space

- An important part of scientific computing is being organized. Good organization is key since you will create many files during the semester and possibly many versions of the same file.
- Where will you store your files?
  - When you get a class account, you receive storage space on a common file storage system.
  - When you log onto one of the departmental computers, then your files will be there.
- How do you name files?
  - Choose names for files that are descriptive; e.g., bisection_method.f90 to represent the code which implements the bisection method rather than exercise1.f90 because in a few weeks time you will have forgotten what exercise 1 was.

  - Keep a Readme file which is a text file listing file names and what they do. For example, on the section for solving nonlinear equations you might have a file `Readme_nonlinear.txt` which lists programs and a brief explanation of what the code does. (This can easily be generated by cutting and pasting documentation from the actual program.)

- Organize your programs into folders or directories. For example, we will program several methods for solving a nonlinear equation so you may have a directory entitled `nonlinear_methods`.

- On the classroom machines you will use UNIX commands to make directories, move files, duplicate files, etc.

- If you use your personal computer, you may need to transfer files between your FSU file storage system and that of your personal computer. The `sftp` command is used for this. More on this later.

A Simple Program

Now we want to look at a program whose purpose is to "say hello", i.e., it just prints out the statement Hello World

```
program hello_world

! program to say hello

print *, "Hello World"

end program hello_world
```

- The program begins and ends with a program statement.
- Comment statements begin with an `!` and are ignored by the compiler.
- A simple `print` statement is used to write out the desired statement.

- Remember that fortran is NOT case sensitive so we could have typed, e.g.

```
PRINT *, "Hello World!"
```

or

```
PrinT *, "Hello World!"
```

instead of

```
print *, "Hello World!"
```

Note that in the print statement everything inside of quotations is printed.

Running the sample program

1. Log in using your class account; you should have been sent the information via email.
2. Go to the class website and look for the file hello_world.f90. To download it RIGHT CLICK the link and use the save command that appears in the menu. You need to specify where you want the file saved; the best place is your account (not desktop)
3. Open a terminal window; to do this go to the lower left hand corner and click the icon next to the browser icon and look under accessories.
4. To compile the program - type `gfortran hello_world.f90`
5. The executable will be automatically called `a.out`
6. Execute the file by typing `./a.out`
7. The output should appear on your screen.

What happens if we make an error in writing our program?

If we wrote the program as

```
program hello_world
! program to say hello

print  "Hello World!"

end program hello_world
```

Then the compiler gives us the error

```
print  "Hello World!"
       1
Error: Missing leading left parenthesis in format string at (1)
```

which tells us which statement it doesn't understand. If we left off the `end`

`program` statement then we get the error

```
Error: Unexpected end of file in 'hello_world.f90'
```

Downloading, Compiling, and Executing a Code from the Website

- Start Firefox and go to the department website

  www.sc.fsu.edu

- Click on faculty; click my picture
- Click on personal homepage (you may want to bookmark this)
- Click on teaching and scroll down to our course
- Under **Fortran Codes/ Week 1** put the cursor over the link "hello_world.f90"
- Right click the mouse and scroll down to "Save "; the file will be saved in your directory (with your name) which appears on the desktop. It will give the code the same name as on the website unless you change it. This is fine for now.
- Now you have the file on the local machine and we want to compile and execute it.

- Under the red hat icon click and go to system tools
- Click system tools and go down to terminal and click; a window opens
- To compile the program type

```
gfortran  hello_world.f90
```

- A prompt on a new line should appear if there are no errors.
- To execute the program type

```
./a.out
```

- The output "Hello World" should appear on a new line in the terminal window.

Modifying/Creating a Program

- When you write a program you will need to create it with an editor just like when you create a text document.
- The easiest editor available on these machines is gedit.
- Certain editors are especially useful for creating a program because they automatically color code the statements which improves readability and debugging (finding errors)
- I would suggest starting with `gedit` and then later you can switch to a more sophisticated editor if you so desire.
- Help is available online by googling "gedit".
- To modify the existing program click the "red hat" icon and scroll down to accessories then click the item gedit text editor and click to use edit.

- Under file click open and it should give a list of your files which includes hello_world.f90 Click on it and the file should open.
- Notice the color coding in the file.
- To add statements or modify current statements simply move the cursor around.
- To copy a line just highlight it and use copy/paste commands under the edit menu
- Save your new version of the file

Classwork/Homework

- Download the code `hello_world.f90`
- Compile and execute
- Open `gedit` and open the code
- Modify the code to print "Hello World" two times; compile and execute. You can use the arrow keys to go back to your command to compile or run the code instead of retyping.
- Now edit the second print statement to print out "That's all folks"; compile and run
- Read the Introduction to a Unix tutorial found in

`http://www.ee.surrey.ac.uk/Teaching/Unix/`

Our Strategy for Learning Fortran 90

- We want to consider some computational problems which build in complexity.
- We want to investigate some algorithms (or methods) for solving these problems.
- We want to learn enough Fortran 90 to implement these methods.
- At first we will be writing very simple codes but as the semester goes on, we will see how to write more general codes using object-oriented techniques.
- We will also look at ways to output the results of our computer programs.
- Before we begin to look at a particular problem we want to get familiar with some basic fortran syntax.

Goals today: learn some basic fortran statements and write a code to perform basic scientific calculator operators

A More Complicated Program

This program sums the first n integers. The program queries the user to tell it the value of n and then it uses an iteration loop (called a do loop to sum these integers.

```
   program sum_integers
!
! This program sums the integers from 0 to n and prints the result
! The user is asked to input the number of integers to sum.
!
!*****************************************************
!*****************************************************
!
  implicit none
!
  integer :: n
  integer :: i
```

```
  integer :: value
!
!*****************************************************
!
!  Ask user to read in the number of integers you want to sum
!
   print *, " Enter the number of integers you want to sum "
   read *, n
!
   value = 0    !  initialize the sum to zero
!
! Repeatedly add the integers up to n

   do i = 1, n

     value = value + i

   end do
```

```
print *, " The sum of the first ", n, " integers is ", value

end program sum_integers

!*****************************************************
```

Fortran Basics

- Fortran 90 is a free format language which means that you can start or end a statement in any column
- Fortran 77 is not free format
- Fortran is not case sensitive

  `Program` is the same as `program` is the same as `PROGRAM`

- A simple code will consist of the following

- program statement
- opening comment statements for documentation of program
- `implicit none` statement (to be discussed later)
- declaration statements
- executable statements
- end program statement

Comments statements should be added throughout program for readability and clarity.

- Many editors use text coloring to make your code more readable and to find errors more easily.
- The `program` and `end program` , declaration are usually one color
- Comments are usually printed in another color.
- We will also see that do loops, conditionals are highlighted in a specific color.

- Read and write statements are highlighted.

Program Statements

- First statement of each computer program is a program statement
- Syntax
  - the word `program` followed by at least one space followed by the name of the program
  - for example, the program which uses Monte Carlo for determining π might be called

```
program  mc_pi
```

  - the name of the program does not have to be the same as what you call the program but typically you would call this program mc_pi.f90

- `end program` statement is last statement of your program
- Syntax
  - the words `end program` followed by the name of the program
  - the program name must be the same as in the initial program statement
  - for our example, the end program statement is

```
end program mc_pi
```

- For example, if your code consists of the following you will get an error message upon compilation. Why?

```
program test1
implicit none
    print *, "test"
end program test
```

Comment Statements

- Comment statements are essential for documentation.
- You will write many codes during this course and by the end of the semester (let alone next year) you will have forgotten what a code does or how you are implementing an algorithm.
- Comment statements will help you (and others) figure out what your code does.
- Fortran ignores comment statements - they are for human use only
- Fortran must know which statements are to be executed and which statements are just there for your information.

- How does fortran know when a statement is a comment statement?
  - An exclamation mark `!` tells fortran to ignore what comes after it
  - The exclamation mark is often at the beginning of a line if you are adding a sentence describing what the code does

    ```
    !  This program sums the first n integers.
    ```

  - Other times you may want to add a comment at the end of a line

    ```
    average = total / n !  compute average of n numbers
    ```

  - Note that in this case the compiler ignores everything after `!`

Declaration Statements

- When we write a code we use variables.
  - For example, the integer `n` may be the total number of integers we are summing.
  - Or `average` may be our average of n numbers which is a real number
- Each variable must be assigned a memory location to be stored in.
- In order for the compiler to do this, it must know what type of variable it is. For example, an integer requires less storage than a real number.
- Consequently we must declare each variable that we use.
- In Fortran 77 programmers used to declare that any variable starting with i through n were automatically integers. This is now considered bad programming practice.

Implicit None Statement

- We will always use the declaration

```
implicit none
```

- This statement appears before statements declaring our variables as real, integer, etc.
- Its purpose is to tell the compiler that we will declare every one of the variables we use.
- This is very useful in debugging a code.
- For example, if we have a variable named `psi` which we have declared as a real number and in a statement we accidentally type `phi` then the compiler will give us an error that says undefined variable.
- If you do not use this statement then the compiler will assume that any variable beginning with i, j, k, l, m, n you do not declare is automatically an integer; variables beginning with the remaining letters are automatically reals.

Data Types

There are 5 basic data types

- integer
- real
- complex (we will not be using complex arithmetic)
- character
- logical

Every variable name you use must be declared to be one of these five types because the compiler needs to associate this variable with a memory location.

For example

```
integer :: n
```

declares the variable n as an integer.

Integers

- An integer is a whole number
- It can be positive, negative or zero
- Examples

$$0 \qquad 5 \qquad -45 \qquad 1024 \qquad -234567$$

- Syntax for declaring a variable n as an integer

```
integer :: n
```

- Format for declaring variables m, n as integers (in one statement)

```
integer :: m,n
```

- It is usually better programming practice to declare each variable separately
- Counters for recursive loops will always be integers.

Real Numbers

- We will use the terminology real number or floating point number interchangeably.
- Real numbers will always contain a decimal point but no commas.
- Real numbers may be entered using standard decimal expression or in exponential notation.
- Examples

$$0.0 \qquad 0.5231 \qquad -45.1 \qquad 1024.79$$

$$0.0E0 \qquad 5.231E-1 \qquad -4.51E1 \qquad 1.02479E3$$

- Syntax for declaring a variable `average` a real number

```
real :: average
```

- Format for declaring variables a, b as reals (in one statement)

```
real :: a, b
```

- It is usually better programming practice to declare each variable separately
- Later we will talk about precision which will tell the computer how many significant digits to use to store a floating point number.

Characters

- A character (sometimes called a string) is a sequence of symbols from the fortran character set.
- For us, characters will usually include letters, numbers, periods, underscores, and blanks. See page 24 of the suggested text or the web for a complete list of the fortran character set. Note that your author left off the underscore character " _"
- Examples

  `approx_pi.txt` (13 characters) `jane doe` (8 characters including blank)

- Syntax for declaring a variable filename as a character

$$\texttt{character(len = 20) :: filename}$$

- Note that the syntax `(len=20)` allocates a maximum of 20 characters for the string filename

Logicals

- There are two logical constants in fortran:

```
.true.    .false.
```

- So when you declare a variable as a logical, you are saying that it can only have the value true or false
- Syntax for declaring a variable flag_converge as a logical

```
logical :: flag_converge
```

- As an example we might initially set `flag_converge=.false.`, perform some iterations of an algorithm and when convergence is acheived we set `flag_converge=.true.`. That way we can test this variable to check if convergence has been achieved.
- Note that when we set a logical to be either true or false the syntax requires that we use a period before and after true or false.

The Parameter Statement

- Sometimes we want to declare a variable in such a way that it can never be changed throughout the entire program.
- For example, we might want to define π or some physical parameters in our problem.
- Fortran allows us to do this when we declare the variable through the parameter statement.
- Examples of syntax for the parameter statement declaration

```
real, parameter ::  pi = 3.14159

integer, parameter ::  two = 2
```

Continuation of a statement

- If a statement extends over more than one line you can continue the statement by putting the symbol & at the end of the line
- For example, the following two statements are equivalent

```
b = a + 7
```

```
b = a   &
    +7
```

- When writing a code in an editor you don't want to make the statements too long

More than one expression on a line

- Almost all the time, we will have a single expression on a single line of the file.
- Sometimes, however we may have several assignments that we want to put on the same line of the file for compactness.
- This can be done by separating the expressions by a semicolon

$$\texttt{x1} = 1.0; \qquad \texttt{x2} = 1.0; \qquad \texttt{x3} = 1.0$$

- This should be used sparingly because it reduces the readability of your code which makes debugging harder.

The Assignment Statement

- The assignment statement just assigns a value to a variable; the variable may be integer, real, etc.
- For example, if `radius` is declared real and `n` is declared an integer then we could have

  ```
  radius = 5.0
  n = 5
  ```

- If `n` is declared an integer and we write

  ```
  n = 5.1
  ```

  then most compilers won't give you an error, it will simply truncate `n` to 5.
- If `radius` is declared real and you type

  ```
  radius = 5
  ```

  then the compiler will set it to 5.0 but you should always distinguish between

reals (with decimals) and integers (without decimals) for good programming practices.

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

- We looked at a code which summed the first n integers. The program queried the user to enter the value of n. You simply type it when asked and hit return.
- To do this, we can 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
```

Writing a fortran program to use as a basic scientific calculator

- One of the simplest (but not too useful) programs to write is to perform calculations like a scientific calculator
- For example, we could compute the value of

$$\sin\frac{\pi}{9} + e^{0.234} + (2.345 - \sqrt{4.5})^5$$

- Fortran has some built-in functions which are called intrinsic functions for trig functions, square roots, exponentials, logs, etc. - just like you have keys on your calculator.
- The first thing we need to know is what are the symbols for numeric operations - adding, multiplying, exponentiation, etc.
- We also have to be concerned about order of operations. However, just like on paper, if you use parentheses freely then you can minimize problems. For example, does this expression mean $\frac{7}{2}$ or $3 + \frac{4}{2} = 5$?

```
3 + 4 / 2
```

Numeric Operations

- In fortran the numeric operations use standard symbols except exponentiation.

addition	+		
subtraction	-		
multiplication	*		
division	/		
exponentiation	**	(not ^)	(irritating, I know)

- For example, if we want to compute

$$(5.1)^6 - \frac{908.5}{7.36}$$

we type

```
5.1 **6 - 908.5/7.36
```

Mixed Arithmetic

- Most of the time, we will be performing numeric operations on variables. For example, we may want to compute $\frac{a}{b}$ where a, b have been defined.
- Care must be taken to distinguish between dividing two integers and dividing two real numbers.

`5/4` is not the same as `5.0/4.0`

For example,

```
integer ::  n,m

real ::  value

n=5;   m=4

value = n/m
```

computes `value = 1` whereas

```
real ::  n,m
real ::  value
n=5.0;   m=4.0
value =n/m
```

computes `value = 1.25`

- Moreover, we will see that if we use mixed-arithmetic (i.e., trying to combine integers and reals) we can sometimes get in trouble.
- Using mixed arithmetic is considered bad programming practice. In this class you will lose points on your program if you use it!
- For example, instead of writing `2.0 +3 / 7` you should use `2.0 + 3.0/7.0`

Order of Operations

- Fortran has a set of rules for the order in which operations are computed. The order is the way we normally expect in mathematical expressions.
- The priority rules for parentheses-free expressions are:

  1. perform all exponentiation; if more than one go right to left
  2. multiplications and divisions are performed next - left to right
  3. additions and subtractions are performed last - left to right

- We can modify the standard priority rules by including parentheses.
- The order of priority for parentheses is inner to outer.
- The rule of thumb you should follow, is when in doubt, use parentheses.
- Parentheses can also make complicated expressions easier to read.

Examples

- The expression

$$\texttt{x} + \texttt{y} * \texttt{z}$$

means to multiply y times z and add the result to x .

- It is not the same as the expression

$$(\texttt{x} + \texttt{y}) * \texttt{z}$$

In this case the parentheses take precedence and x is first added to y and the result multiplied by z .

- In the expression

$$\texttt{x} + \texttt{y} \texttt{**2} * \texttt{z}$$

exponentiation takes precedence so it is done first; then the multiplication and finally the addition. Thus the expression means to first square y , then multiply the result times z and add the result to x .

- Note that the following expressions are not the same

$$\texttt{2.0 ** 3 **2} = 2^9 = 512 \qquad (\texttt{2.0 **3}) \texttt{**2} = 8^2 = 64$$

- Here's an example of an expression using nested parentheses

$$4.1 * ((5.5 \ / \ 2.35) **4 \ / \ 2.0)$$

  We first perform the division 5.5/2.35 (inner most parentheses) , then the expression inside the outer parentheses is done using priority rules - raise the result to the fourth power and then divide by two . Finally we multiply result by 4.1

- Warning - a programming error that we often get is mismatched parenthesis. This means that we have inadvertently missed a left or right parenthesis. For debugging ease, I like to put a space between the parenthesis and the quantities to be computed so that they are more obvious.

- In the classwork you will see this compilation error when you debug a code.

Classwork/Homework

1. Download the file `sum_integers.f90`; compile .
2. Execute the program. The program will ask you to type in n, the number of integers from 1 to n that you want to sum; if you want to sum the first 100, simply type 100 and hit the return key
3. Execute the program again using a different value of n you do not have to re-compile it since you are not changing the program.

   We can check the answer by remembering the formula from calculus

   $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

   so the sum of the first 100 integers is 50(101)=5050 and the sum of the first 50 integers is 25(51)=1275.
4. Modify the code to **also** compute the sum of the squares of the first n integers by using the exponentiation operator `**` and print out result.

5. Edit your code and use (incorrectly) $\wedge$ instead of ** for exponentiation and compile the code; notice the error message. Fix your code and make sure it is running correctly.
6. Read the first Unix tutorial found in

   http : //www.ee.surrey.ac.uk/Teaching/Unix/

You should demonstrate to us that you have a properly compiled and executed the code by the start of next class. It does not have to be handed in.

Intrinsic Functions

- A fortran compiler has many built-in or intrinsic functions for standard mathematical operations.
- I have posted a list of intrinsic functions in fortran under the Miscellaneous Files heading on the website
- 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

$$4e^{.5} + \sin 90^\circ - \ln 2.79$$

```
4.0 * exp(0.5) + sin(pi/2.0) - log(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

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

and print out a_i for each i

```
real ::  a_i
integer ::  i

    a_i = 100.0;      print *, a_i
    do i = 4,1,-1
      a_i = a_i / 2.0;      print *, a_i
    end do
```

- `i` is first set to 4, `a_i` computed as 100./2.=50.
- `i` is then incremented by -1, i.e., on the second step $i = 3$ 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.
- First we look at the simple `if` statement where we test a condition and have only one alternative.

Syntax for the case when we only want to perform one statement if the condition is satisfied

```
if ( condition ) statement
```

Examples

```
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 you want to print out an error message, we would have the following `IF ELSE` construct

```
if ( a >= 0.0 ) then
        a = sqrt(a)
    else
        print *, "error: a is less than zero"
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
```

Classwork/Homework

1. Download the code `calculator.f90` which computes the expression $5.7 + 2^{5/2} + \sin 45^\circ$. Note that π is defined as a parameter in the code. Compile and execute the program. The correct answer is approximately 12.064
2. Modify your code to compute
$$(5.7 + 2^{5/2})^{1/2} + \sin 60^\circ$$
The correct answer is approximately 4.23602
3. Modify the code to add $\ln 2.79$ to your answer. The correct answer is 5.26206
4. Now define a new real variable named `value_new` to compute and print out the quantity given by
$$4e^2 + \cos 30^\circ + \tan 45^\circ$$
whose value is approximately 31.4222 . Note that the argument of `exp` must be a real number.
5. Now declare the variable `i` as an integer parameter and set it to 2. Add a conditional to print out `value_new` only if $i > 1$.