ASP4100 Introduction to honours computing

Introduction to modern Fortran (incl. Makefiles)

1 Introduction to programming

1.1 What is a programming language?

Computers speak binary. Each chipset (e.g. Intel x86 or the ARM A5 processor in your smartphone) has it's own set of low-level instructions known as "assembly language" that translates to the raw binary instructions. Assembly language is very low level: "push onto stack, move to this memory address, pop number off the stack, etc.".

A *programming language* is a human-readable language that you can use to specify your human intent (i.e. "solve this equation", "add these numbers and print the answer"). The *compiler* translates this into raw assembler code and ultimately into an executable binary file. Thus all programming languages roughly do the same thing - the difference is just how easy it is to specify what you want to do in a given language.

In Unix-speak a program is just a "text file" that you create with any standard text editor such as vi, emacs or gedit. The compiler converts your text into a binary file that it can execute. You then *run* the program. *Scripting languages* such as PYTHON and PERL differ from more traditional *compiled* languages because these two steps are combined into one, that is the program is compiled 'on-the-fly' when you run it.

1.2 Hello world

With any language, you should always start with the simplest possible program, or:

> "*Make it work before you make it complicated*"

To type and run a basic program in Fortran, create a text file with the extension `.f90`. For example type `gedit hello.f90 &` and enter contents as follows:

```
program hello
 print*,'hello world'
end program hello
```

We will use the *Gnu Fortran compiler*, `gfortran` (since it is free), so the relevant command is as follows (having made sure you've saved the file to disk):

```
$ gfortran -o bob hello.f90
```

Assuming this did not throw up any errors, this creates an executable file with the name `bob`, that you can run from the unix shell:

```
$ ./bob
 hello world
```

It is instructive to see what the assembler code actually looks like. You can do this by typing:

```
gfortran -S hello.f90
```

This creates a text file called hello.s containing the assembler code. The contents of this file shows how unreadable assembler language really is and hence why programming languages exist:

```
$ more hello.s
        .cstring
LC0:
        .ascii "hello.f90\0"
        .const
LC1:
        .ascii " hello world"
        .text
_MAIN__:
LFB0:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
LCFI1:
        subq    $480, %rsp
        leaq    LC0(%rip), %rax
        movq    %rax, -472(%rbp)
        movl    $3, -464(%rbp)
        movl    $128, -480(%rbp)
        movl    $6, -476(%rbp)
        leaq    -480(%rbp), %rax
```

2 Introduction to modern Fortran

Fortran (*For*-mula *Trans*-lation is a language written to solve mathematical equations. The modern language (Fortran), defined via standards set in 1990, 2003 and 2008, is backwards-compatible with the old language (referred to as FORTRAN, standardised in 1977) meaning that it can be used to run old codes as well, but certain features have been declared obsolete. Most criticisms of Fortran refer to problems in FORTRAN that were fixed more than 25 years ago.

2.1 Why use modern Fortran instead of F77?

Here is why. Type the following into a file called `test.f`. Note that in the old FORTRAN language each instruction needed to start in the 6th column and cannot exceed 72 characters in length — this was to fit on the punchcards they used in the 1960s:

```
      program badfort

      do 30 i=1.20
         print*,i
30    continue

      end
```

Compile and run this:

```
gfortran -o test test.f
```

First, notice that the compiler compiled this with no warnings or errors. But what is the output? Does the program do what you expect?

The above example has an obvious bug in it. But the compiler was not able to detect the bug because the program is written sloppily. Most of your time programming will be spent debugging. So anything in a language that can help you find problems more easily will save you a lot of time and pain. The big one in Fortran is the *implicit none* statement. Type this program in modern syntax, but with the same bug, into a file called `test.f90` (no special line spacing is required if the file extension is `.f90` instead of `.f`):

```
program badfort
 implicit none

 do i=1.20
    print*,i
```

```
  enddo

 end program badfort
```

Compiling this, the compiler will immediately tell you what is wrong with it:

```
$ gfortran -o test test.f90
test.f90:4.10:

 do i=1.20
          1
Error: Syntax error in iterator at (1)
test.f90:6.4:

 enddo
    1
Error: Expecting END PROGRAM statement at (1)
test.f90:5.12:

    print*,i
            1
Error: Symbol 'i' at (1) has no IMPLICIT type
```

This means you can fix the problems *before* you go launching your satellite. The line and column numbers are given in the error messages so you can pinpoint the problematic lines of code.

A longer explanation here — F77 allowed any variable to be assumed to be real or integer depending on the first letter. Specifically variables starting with letters a–h or o–z are assumed to be real (floating point) while variables starting with l–n are assumed integers. Thus in our example above the compiler assumed we had set a new variable called "do 30 i" and set it's value to 1.2. We then print i, which is a different variable that has not been set and so assumes a random value from memory, which is what was printed. Thus the famous joke:

> *"God is real unless declared integer"*

Hence, *always* declare implicit none at the top of your code. With gfortran you can also enforce this even if you forgot to type it by adding the flag `-fimplicit-none`, i.e.:

```
$ gfortran -fimplicit-none -o test test.f90
```

2.2 A simple program

Let's perform a simple mathematical calculation in Fortran (e.g. in a file called `maths.f90`):

```
program maths
 implicit none
 real :: r, area
 real, parameter :: pi = 4.*atan(1.)

 r = 2.0
 area = pi*r**2
 print*,' pi is ',pi
 print*,' the area of a circle of radius ',r,' is ',area

end program maths
```

Compile and run this and confirm the output is as follows:

```
$ gfortran -o maths maths.f90
$ ./maths
  pi is    3.14159274
  the area of a circle of radius    2.00000000      is    12.5663710
```

2.3 Organising your code

Before we go into detail about how to do specific things, we must learn how to organise a code. The guiding principle is the following:

> *Never repeat code.*

You *will* be tempted. What's a little cut-and-paste between friends? Cut-and-paste equals death to good programming. Instead, you need to organise your code so that it is broken up into small, re-usable pieces.

Let's take the example above, and put the area calculation into a function. The modern way to do this is to put it in a `module`. To do this, create a new file called `area.f90` where the area function is defined. Let's also add some *comments* to the code:

```
! A module containing functions to compute the area of a circle
! Written by Daniel Price, 2015
module geometry
 implicit none
```

```
 real, parameter :: pi = 4.*atan(1.)
 public :: area, pi
 private

contains
 !
 ! A function to calculate the area of a circle of given radius
 !
 real function area(r)
  real, intent(in) :: r

  area = pi*r**2

 end function area
end module geometry
```

The main purpose of comments is *for your future self*. It is very easy to forget the thought process you had when you wrote the program, so use comments to record this information. At bare minimum, state the purpose of the module, subroutine, function or program in the comments. Making this a habit *will* save you pain later.

Let's re-write the main program (in `maths.f90`) to use the module:

```
program maths
 use geometry, only:area,pi
 implicit none
 real :: r

 r = 2.0
 print*,' pi is ',pi
 print*,' the area of a circle of radius ',r,' is ',area(r)
end program maths
```

To compile this we need to tell the compiler about the different files and also compile them in the order that they are needed

```
$ gfortran -o maths area.f90 maths.f90
$ ./maths
```

Does your program give the same output as before? By putting functionality into modules in this way we can construct a software project as large as we like, and maximise the re-use of code. It also enables us to debug and test the functions in the module separately from the main code (in software parlance this is known as *unit testing*) which is a huge help in keeping the program bug-free.

You can see some other important things in the programs we just typed:

- We declared `implicit none` in both the main program and at the top of the module (it is unnecessary in the function, the global declaration at the top of the module already ensures this is the case)
- In the module we explicitly denoted which parts are `public` (that is, available for use in other parts of the program) and which are `private` (available only to functions inside the module).
- In the function we declared the radius variable as `intent(in)` which tells the compiler that this variable is expected to have a value on input and should not be changed by the function.
- In the program, we explicitly stated which bits of the module we wanted using `only:area,pi` rather than importing everything in the module.

Use these for same reason as the "do 30 i" example above — the more information you can give the compiler about what you are trying to do, the easier it is for the compiler to report an error and the less time you will waste debugging your code.

2.4 Compiler flags

Another time saver is to turn on *compiler warnings*. The syntax is different with different compilers, but with `gfortran` this is achieved by adding the `-Wall` flag:

```
$ gfortran -Wall -o maths area.f90 maths.f90
```

To use the maximum possible level of warnings and the strictest adherence to the latest standards, we would use:

```
$ gfortran -Wall -Wextra -pedantic -std=f2008 -o maths area.f90 maths.f90
```

Finally, when writing simulation code we want it to run fast. Achieve this by turning on *optimisation flags*:

```
$ gfortran -O3 -o maths area.f90 maths.f90
```

Note that the flag is the letter O followed by a 3 (for optimisation level 3), not zero.

3 Using Makefiles to compile your code

We only used two files in the example above, but it quickly becomes laborious to remember the various files that go together to make a program and the combination of compiler flags you last used. Also with larger codes, is is inefficient to recompile the whole code even though only one file has changed. To do this we must compile each file separately into an *object* file (`.o`) and then *link* the object files together to create the binary executable. We do this using the `-c` flag as follows:

```
$ gfortran -o area.o -c area.f90
$ gfortran -o maths.o -c maths.f90
$ gfortran -o maths maths.o area.o
```

Doing this by hand is laborious, but we can automate it using a `Makefile`. The `make` program is actually a programming language in itself — one designed to make compiling codes easy.

3.1 Hello world in Gnu Make

To use `make`, simply create a file in the working directory with the name `Makefile` (that is, make a makefile), with contents as follows:

```
VAR=hello

default:
<tab>  echo ${VAR}
```

Here we have just set a string variable called `VAR` and set its value to 'hello'. `<tab>` means press the tab key. Tab is treated very different to spaces in Makefiles, so this is important. You should already be familiar with the unix `echo` command and what it does. Typing `make` on the command line will run this:

```
$ make
echo 'hello'
hello
```

The key elements of Makefiles are *targets* and *dependencies*.

3.2 Targets

To understand targets edit your Makefile to have two different targets, 'merry' and 'fun':

```
VAR=hello
VAR2=world

merry:
<tab>  @echo ${VAR}

fun:
<tab>  @echo ${VAR2}
```

(The `@` stops the Makefile from printing the shell command to the screen before it executes it). You can now execute each of these targets separately:

```
$ make
hello
$ make merry
hello
$ make fun
world
```

Hopefully this makes targets clear — by convention the first target in the file is also the default (i.e. the one executed when you just type `make`).

3.3 Dependencies

Dependencies are specified on the right hand side of the colon. Let's modify the above example so that the 'fun' target *depends on* the 'merry' target by amending the appropriate line to read:

```
fun: merry
<tab>  @echo ${VAR2}
```

Now type 'make fun' and 'make merry' as previously. Do you understand the output? In particular the 'merry' target should give:

```
$ make fun
hello
world
```

3.4 Variables in Makefiles

Variables are fairly straightforward in the examples above. You can override variable setting(s) on the command line:

```
$ make VAR=goodbye fun
goodbye
world
```

There are also some special predefined variables — `$@` refers to the name of the target (the text to the left of the colon), while `$<` refers to the dependencies (everything to the right of the colon). Try the following example:

```
fun: merry
<tab>  @echo ${VAR2} is $@ and $<
```

What is the output?

We can also define new variables from existing variables, with various string replacement options, e.g.

```
VAR=hello
VAR2=world
VAR3=${VAR2:world=planet}

hay: merry
<tab>  @echo ${VAR3}
```

and you should find:

```
$ make hay
hello
planet
```

3.5 Using a Makefile to compile your code

From the above you should be able to see how to use a Makefile to compile your code. Since this is what `make` is designed for there are standard names for the appropriate variables to use. A very simple Makefile for compiling our Fortran program would be:

```
FC=gfortran
FFLAGS=-O3 -Wall -Wextra
SRC=area.f90 maths.f90
OBJ=${SRC:.f90=.o}

%.o: %.f90
<tab> $(FC) $(FFLAGS) -o $@ -c $<
```

```
maths: $(OBJ)
<tab> $(FC) $(FFLAGS) -o $@ $(OBJ)

clean:
<tab> rm *.o *.mod maths
```

A couple of things:

- The first target is a special 'wildcard' rule telling Make how to compile *any* file ending in `.o` from the corresponding `.f90` file. This specifies that a file called `fred.o` would depend on the corresponding `fred.f90` and if the `.o` does not already exist should be created using the rule specified on the line below.
- We added a `clean` target which cleans up the 'dog poo' generated by the intermediate step of creating the .o files. So we can just type `make clean` to clean up these temporary files.
- Notice the use of the special variables `$@` and `$<`.

Now compiling your code should be as simple as typing 'make':

```
$ make
gfortran -O3 -Wall -Wextra -o area.o -c area.f90
gfortran -O3 -Wall -Wextra -o maths.o -c maths.f90
gfortran -O3 -Wall -Wextra -o maths area.o maths.o
$ ./maths
```

Even better, if nothing has changed then typing `make` again does nothing:

```
$ make
make: 'maths' is up to date.
```

but if we make a change to just one file then `make` knows to only recompile that one file and redo the link step:

```
$ touch maths.f90
$ make
gfortran -O3 -Wall -Wextra -o maths.o -c maths.f90
gfortran -O3 -Wall -Wextra -o maths area.o maths.o
```

You should see, however, that this is dangerous if we apply it to the area.f90 module. That is, if `area.f90` changes then we should also recompile `maths.f90` as well, since our program calls functions in that module. How should we ensure this happens? By specifying the dependency in our Makefile!

```
...
maths.o: geometry.mod
```

Specifying all of the dependencies is hard to do for bigger codes, and only relevant if the interface to the routine changes, I often don't bother and instead just make sure the files are complied in the correct order (i.e. `SRC=area.f90 maths.f90` not `SRC=maths.f90 area.f90`). Then I just type `make clean` if I know that I have changed the way a routine is called.

3.5.1 (Advanced) Generating Fortran dependencies automatically

With gfortran there *is* a way of generating the information needed in the Makefile automatically using the `-M` flag. To do this, add another target as follows:

```
DEPFLAGS=-M -cpp
deps:
<tab> @$(FC) $(DEPFLAGS) $(SRC)
```

and you should find

```
$ make deps
area.o geometry.mod: area.f90
maths.o: maths.f90 geometry.mod
```

Even better, we can put the output of this command into a file `make.deps` that we include in our Makefile automatically using an `include` line. We can even generate this file automatically by making `make.deps` depend on `deps`. The whole thing looks like:

```
#-- optional stuff for specifying dependencies automatically
DEPFLAGS=-M -cpp
deps: $(SRC)
<tab> @$(FC) $(DEPFLAGS) $(SRC) > make.deps

make.deps: deps

include make.deps
```

4 Key Fortran concepts

4.1 Floating point precision in Fortran

The basic types of variables in Fortran are `integer`, `logical`, `real` and `character`. Declaration of these takes an optional `kind` parameter. So for example to specify an

8-byte real instead of a 4-byte real one would use:

```
real(kind=8) :: x
```

or just

```
real(8) :: x
```

Technically, using the kind parameter to specify the number of bytes is compiler dependent, but it is very widespread. The correct way is to let Fortran define the kind according to the desired precision of the calculation using `selected_real_kind`. To do this, let's create another file called `precision.f90` with contents as follows:

```
! module to define precision of real variables
module prec
 implicit none
 integer, parameter :: dp = selected_real_kind(P=10,R=30)
 integer, parameter :: sp = selected_real_kind(P=5,R=15)
 integer, parameter :: dp_alt = kind(0.d0)

 public :: dp,sp,print_kind_info
 private

contains
 ! routine to print information about the kinds being used
 subroutine print_kind_info()
  real(sp) :: pi_single
  real(dp) :: pi_double

  print*,' double precision is kind=',dp
  print*,' single precision is kind=',sp
  print*,' kind of a double precision number is ',dp_alt

  pi_single = 4.0_sp*atan(1.0_sp)
  pi_double = 4.0_dp*atan(1.0_dp)
  print*,' pi in single precision is ',pi_single
  print*,' pi in double precision is ',pi_double

 ! see what happens if we accidentally mix precisions
  pi_double = pi_single
  print*,' pi converted from single is = ',pi_double

 end subroutine print_kind_info
end module prec
```

Then add this file to the line in the `Makefile`:

```
SRC=precision.f90 area.f90 maths.f90
```

and edit the main program to call this subroutine and also print the kind of a default real. That is:

```
program maths
 use geometry, only:area,pi
 use prec,     only:print_kind_info
 implicit none
 real :: r

 call print_kind_info()
 print*,' radius is of kind ',kind(r)
 ...

end program maths
```

Compiling and running this, you should find:

```
$ ./maths
  double precision is kind=           8
  single precision is kind=           4
  pi in single precision is    3.14159274
  pi in double precision is    3.1415926535897931
  pi converted from single is =    3.1415927410125732
  radius is of kind            4
  ...
```

Notice the precision loss in converting from double to single precision.

Exercise 4.1.1: Rewrite the `area` function so that it works in double precision instead of single precision. To do this, redefine all the `real` variables to be `real(dp)`, with `dp` imported from our `prec` module as above.

4.2 Arrays and array operations

4.2.1 One dimensional arrays and array operations

Arrays are very powerful in Fortran, and this is one of the key advantages over other languages such as C and C++. Defining an array just means giving one or more *dimensions* to a variable. For example a vector of 3 numbers can be defined using:

```
real :: x(3)
```

Let's create an example module for array operations, in a file called `arrays.f90`:

```
module arrays
 implicit none

contains
 subroutine array_examples()
  use prec, only:sp
  real(sp) :: x(3),y(3)

  ! set each element individually
  x(1) = 1.0
  x(2) = 2.0
  x(3) = 3.0
  print*,' x = ',x

  ! set whole array equal to 1
  x = 1.
  print*,' x = ',x

  ! set array parts in one line
  x = (/1.0, 2.0, 3.0/)
  print*,' x = ',x

  ! set y=x
  y = x
  print*,' y = ',y

  ! array operations
  print*,'sum=',sum(y)
  print*,'maxval=',maxval(y)
  print*,'minval=',minval(y)
  print*,'maxloc=',maxloc(y)
  print*,'minloc=',minloc(y)

  ! magnitude
  print*,' |x| = ',sqrt(dot_product(x,x))
  print*,' |x| = ',norm2(x)

 end subroutine array_examples

end module arrays
```

As previously, use your module from the main code:

```
program maths
 use arrays, only:array_examples
 implicit none

 call array_examples()

end program maths
```

4.2.2 Multi dimensional arrays

Multi-dimensional arrays are similar. For a 3×3 matrix just define:

```
real :: x(3,3)
```

Again, you can add, subtract and multiply arrays just as if they were scalars.

5 Logic and loops: if/then/else and select case

Here is a simple example of how to construct and use logical statements:

```
program ifanimal
  implicit none
  logical :: isacow,isadog,hastwohorns
  integer, parameter :: nhorns = 2

  isacow = .true.
  isadog = .false.
  if (isacow) then   ! check if our animal is a cow
     write(*,''(a)'',advance='no') my animal is a cow...'
     if (nhorns==2) write(*,*) ' ...with two horns'
  elseif (isadog) then
     print*,' my animal is a dog. Woof.'
  else
     print ``(a)'','my animal is not a cow'
     hastwohorns = (nhorns==2)
     if (hastwohorns) print ``(a)'',' but it has two horns'
  endif

end program ifanimal
```

6 Writing to and reading from files

Writing to/reading from files is just a matter of using the `open`,`close`,`write` and `read` statements.

```
program io
 implicit none
 character(len=20) :: filename
 real :: x, y

 filename='results.out'
 x = 1.
 y = 2.

 ! write to ascii file
 open(unit=1,file=filename,status='replace',form='formatted')
 write(1,*) x, y
 close(1)

 ! reset vars
 x = 0.
 y = 0.

 ! read from ascii file
 open(unit=2,file=filename,status='old')
 read(2,*) x,y
 close(2)

 ! print vars
 print*,' x = ',x,' y = ',y
end program io
```

7 Exercises

The actual mechanics of programming in Fortran is just a matter of learning the syntax. The best way is to try to implement practical examples. Here are a few simple exercises.

1. Implement a subroutine to return the n real solutions to a quadratic equation, i.e.

$$ax^2 + bx + c = 0, \tag{1}$$

using the exact solution, i.e.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2}$$

Check that the solution returned by your routine is correct by evaluating (1) in your program.

2. Write a subroutine that sets up two one dimensional arrays for position (x) and velocity (v_x). Set the position in equally spaced increments between 0 and 1, and velocity array equal to:

$$v_x = \sin(2\pi x) \tag{3}$$

Write the position and velocity arrays to a file and use `gnuplot` or similar to plot the arrays.

3. Write a simple program that asks for your name and birth year and prints back your name and age. Google for the `date_and_time` intrinsic routine. Rewrite this program to use the `prompting` module (given in the examples on the moodle page) instead of using `read`.

4. Write a program that takes a command line argument and prints it. Use the intrinsic `get_command_argument` routine and the associated `get_number_arguments`