ASP4100 Introduction to honours computing

Version control with git + advanced Fortran

1 Version control with GIT

It begins with "I broke the code, what if I could just go back to how the code was yesterday?". Next you start making backup copies of your code directory. Soon, you end up with 13 copies named `code_backup`, `code_backup2`, `code_yesterday` etc. and you start thinking "there must be a better way...". Well there is. It is called a version control system, and has been around for a long time now.

Version control systems have evolved from RCS to CVS, then SVN and now 'distributed' version control systems of which the most popular and widespread is GIT. Essentially though, the concepts are similar.

1.1 The version control model

The basic model of any version control system is that you never work directly on the "master" code. Instead you 'checkout' a copy from the 'repository', make changes to the copy and 'commit' changes back to the repository. In this way you can see at any point how your copy differs from the 'master' copy, and how your copy or the master copy differs from previous versions of the code.

1.2 Distributed vs. centralised version control

With traditional programs such as SVN and CVS the repository was stored on some central server, whereas the idea of 'distributed' version control systems such as GIT is that *every copy is also a repository* (in other words, every copy also contains the full history of the code). This is more democratic and decentralised, but also more complicated. More confusing still is that one can use still use git to pull/push from a central repository by using code hosting services like `bitbucket.org` or `github.com`.

1.3 Initialising a git repository

The best way to get started is with a simple example. Let's take our code directory from the previous session and initialise a git repository in this directory:

```
$ git init
Initialized empty Git repository in honours-computing/Fortran-examples/.git/
```

Typing `ls -a` you will see that a 'hidden' directory called `.git/` has been created in the current directory. Thus you can remove all of the version information from a directory (e.g. if you are sending a finished document and don't want the version history lying around) by simply deleting the `.git` directory (`rm -rf .git`)

Typing `git status` tells you the current state of the repository:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

Makefile
area.f90
arrays.f90
hello
hello.f90
hello.s
make.deps
maths.f90
precision.f90
test
test.f
test.f90
```

1.4 Adding files to the repository

As yet there is nothing in the repository but you can see a number of "untracked files". As you see, git already tells you what to do: just use `git add` to add the files you want to keep under version control:

```
$ git add Makefile
$ git add *.f90
$ git add make.deps
```

You should now see:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

new file:   Makefile
new file:   area.f90
new file:   arrays.f90
new file:   hello.f90
new file:   maths-old.f90
new file:   maths.f90
new file:   precision.f90
new file:   test.f90

Untracked files:
...
```

The operation is not completed until you *commit* the changes. The main thing is that *you must always specify a message when you commit files*. Hence the commit command always requires the `-m` option, otherwise it will pop up a text editor into which you must type the commit message. Here you just want to indicate something useful about what you're doing:

```
$ git commit -m 'added Fortran examples to the repository' Makefile *.f90 make.deps
```

You must either specify all of the filenames you want to commit, or use the `-a` option to 'commit all'. You should see a message along the lines of:

```
[master (root-commit) f30d247] added Fortran examples to the repository
 8 files changed, 147 insertions(+)
 create mode 100644 Makefile
 create mode 100644 area.f90
 create mode 100644 arrays.f90
 create mode 100644 hello.f90
 create mode 100644 maths-old.f90
 create mode 100644 maths.f90
 create mode 100644 precision.f90
 create mode 100644 test.f90
```

Then typing `git status` you should simply see (perhaps with some untracked files if you did not add everything)

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

maths
```

In the above example `maths` is the binary file, and we never want git to track this. You can prevent the "Untracked files" message by creating a file called `.gitignore` in the current directory containing the list of files that git should ignore (but make sure you add and commit the `.gitignore` file itself to the repository).

1.5 Making and committing changes

Let's make a change to one of the files. For example, let's change one of the print statements in the main code `maths.f90`. After saving the file, we should see

```
$ git status
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   maths.f90
```

We can see how the current code differs from the repository using `git diff`:

```
$ git diff
diff --git a/maths.f90 b/maths.f90
index 1eac0ad..0b4b6a8 100644
--- a/maths.f90
+++ b/maths.f90
@@ -9,7 +9,7 @@ program maths
  call print_kind_info()
  print*,' radius is of kind ',kind(r)
  r = 2.0
- print*,' pi is ',pi
+ print*,' the ratio between circumference and diameter is ',pi
  print*,' the area of a circle of radius ',r,' is ',area(r)

 end program maths
```

The key to good version control is to *commit often*. Basically *you should commit every*

change no matter how small, as long as it doesn't break the code. To commit this change we proceed as before:

```
$ git commit -m 'changed pi comment' maths.f90
[4c0edbd] changed pi comment
 1 file changed, 1 insertion(+), 1 deletion(-)
```

1.6 Tracking changes and recovering old versions

The version history can be seen by typing `git log` or `git shortlog`:

```
$ git log
commit 4c0edbde78085243685250e4171d94370483c9d2
Author: Daniel Price <daniel.price@monash.edu>

    changed pi comment

commit f30d247053eab9472595826d970974255c8fb3f0
Author: Daniel Price <daniel.price@monash.edu>

    added Fortran examples to the repository
```

You refer to, go back to, or diff against an old version by referencing some or all of of the 'sha key' (the garbled string of letters and numbers). Hence to see what changed in a given commit, we could use

```
$ git show 4c0ed
```

and we could use a similar reference in `git diff`, `git checkout` and so on.

1.7 Renaming or deleting files

To rename a file, instead of the unix `mv` command, use:

```
$ git mv file.f90 newfile.f90
```

and likewise use `git rm` instead of `rm` to ensure that files are removed from both the repository and the current directory.

1.8 More advanced git

We've only scratched the surface of what git can do — it is an amazingly powerful tool. For example, you can use `git bisect` to quickly find the exact commit that broke your code, `git tag` to tag specific revisions. *Branches* are a very powerful concept in git, but take some getting used to. We also haven't covered the `git pull`, `git push` and `git merge` commands necessary when you are working with a remote repository.

I would encourage you to teach yourself more by working through the many fine git tutorials on the web. One of the things you should already realise is that git can be used for much more than code. Many people use it to keep track of changes in LaTeX documents and there are many other possible applications.

2 Advanced Fortran

While not designed to be comprehensive, in the remainder of this session we will cover some very useful "advanced" aspects of Fortran that are hard to discover without being told about them. Try to use `git` as you go along to save versions of the code that you write.

2.1 Making functions operate on whole arrays

In the previous session we wrote an 'area' function that returns the area for a circle. But the power of modern Fortran is in array operations. For example try performing the following operation in a simple program:

```
program array
 implicit none
 real :: x(10)

 do i=1,size(x)
    x(i) = real(i)
 enddo

 print*,' sqrt(x) = ',sqrt(x)

end program array
```

The sqrt operation can be performed on the whole array at once, and return an array as the output. It is easy to make your own functions do this. The key is that they must have the `elemental` property. Elemental functions have certain restrictions. For example

they must not print anything to the screen. From the above we see that it would be weird for the sqrt function to print anything. A related keyword is `pure`. Pure subroutines act entirely on their input, do not change global (shared) variables and do not perform any input or output. These kinds of functions can be *easily parallelised*.

Exercise

1. Modify your area function from the earlier lab session so that it can operate on a whole array. To do this, just add the elemental keyword to the function definition:

```
module blah
 implicit none

contains

real, elemental function area(r)
 real, intent(in) :: r
 ...
```

2. Fill an array with a bunch of different radii, and return the area calculated for each element of the array by calling the `area` function with an array as the input argument.

2.2 Generic interfaces

A second useful 'modern Fortran' concept is the use of generic interfaces. The idea is that you can use the same function name to refer to different routines with different kinds of arguments. For example, the intrinsic function `sqrt()` can take either a real or a double precision variable as an argument, but in both cases the same function name (`sqrt`) is used (in old FORTRAN these had separate names: `sqrt` and `dsqrt`). You will see an example of this in the `prompting.f90` module in the examples folder.

To make your own 'generic interface', you must define the two different functions in your module with *different names* and then use an `interface` statement to give the generic name. For example:

```
module geometry
  implicit none
  ...
  interface area
    module procedure area_real, area_double
  end interface area
```

```
  public :: area
  private :: area_real, area_double

contains

 real, elemental function area_real(r)
  real, intent(in) :: r

  area_real = pi*r**2

 end function area_real

 real(kind=8), elemental function area_double(r)
  real(kind=8), intent(in) :: r
  ...
 end function area_double

end module geometry
```

In the above we made the generic name public but the specific names private, so that an external routine can *only* refer to the routine by its generic name.

Exercise

1. Modify your module so that it has both real and double precision version of the area function, and write a program to test this, by calling area with a real and double precision argument (for example `area(1.0)` and `area(1.d0)`). Make sure that you use enough precision in π for both cases!

2. Add a subroutine to your module that computes the area of a rectangle. The interface to this routine should look something like:

   ```
   real function area_rectangle(width,height)
    real, intent(in) :: width, height

    ...

   end function area_rectangle
   ```

3. Add this routine to the generic interface definition, and show that calling `area` with *two* arguments e.g. `area(2.0,3.0)` returns the area of a rectangle instead of a circle.

2.3 Optional arguments

Subroutines that are defined properly in modules can also have *optional* arguments. Whether or not these have been passed can be checked by the `present()` function. For example, let's add an optional argument `factor`, to the `area_rectangle` routine:

```
real function area_rectangle(width,height,factor)
 real, intent(in) :: width, height
 real, intent(in), optional :: factor

 area_rectangle = width*height
 if (present(factor)) then
    area_rectangle = area_rectangle*factor
 endif

end function area_rectangle
```

Exercise

1. With the optional argument implemented as above, try calling the `area` function from the main program with the following argument lists:

   ```
   area(1.0,2.0)
   area(1.0,2.0,10.)
   area(1.0,2.0,factor=1.)
   area(1.0,2.0,factor=10.)
   ```

2.4 Calling C from Fortran

A key part of Fortran 2003 is the ability to interoperate with the C language. This is important because a lot of software libraries are written in C and it is useful to be able to call them directly. Let's give an example of how to do this.

First, download the `fastsqrt.c` routine from the "Fortran examples" directory and save it to your current working directory. Then, open a file called `fastmath.f90` that we will use to create the interface to this routine. Let's then define the interface to the C routine in this module:

```
module fastmath
 use iso_c_binding, only:c_float, c_double
 implicit none

 private
```

```
 public :: fast_invsqrt

 interface fast_invsqrt
  pure function finvsqrt(x) bind(C)
    import
    implicit none
    real(kind=c_float), intent(in) :: x
    real(kind=c_float) :: finvsqrt
   end function finvsqrt
 end interface fast_invsqrt

end module fastmath
```

Notice that we used an *intrinsic* module called `iso_c_binding` to be able to define a real variable that is the same as a `float` or `double` in the C language. The `bind(C)` attribute tells the compiler that this is actually a function call to C. There is no `contains` statement in the module above, since it does not contain any subroutines. We can then call the function with the 'Fortran' name we have given (`fast_invsqrt`) rather than it's name in C. So in your program add the following lines:

```
program testmath
 use fastmath, only:fast_invsqrt
 implicit none
 integer, parameter :: nmax = 10000000
 real :: x(nmax),y(nmax),t1,t2,t3
 integer :: i

 do i=1,nmax
     x(i) = real(i)
 enddo

 call cpu_time(t1)
 do i=1,nmax
    y(i) = 1./sqrt(x(i))
 enddo
 call cpu_time(t2)

 print*,'completed in ',t2-t1,' s'
 do i=1,nmax
    y(i) = fast_invsqrt(x(i))
 enddo
 call cpu_time(t3)
 print*,' fast version completed in ',t3-t2,'s'
```

```
end program testmath
```

To compile the code, we just have to compile the C routine using the C compiler, and the rest with `gfortran`:

```
$ gcc -o fastsqrt.o -c fastsqrt.c
$ gfortran -o fastmath.o -c astmath.f90
$ gfortran -o test.o -c test.f90
$ gfortran -o test test.o fastmath.o fastsqrt.o
$ ./test
```

Exercise

1. Which computation of the fast inverse square root is faster?
2. Add the appropriate interface definition for the double precision version of the fast sqrt routine (`dinvsqrt`) in `fastsqrt.c`. Include this in the definition of the generic interface `fast_invsqrt` and show that you can successfully call `fast_invsqrt` with either a real or double precision argument.

2.5 OpenMP parallelisation

A final aspect of Fortran and C is that both languages support parallelisation across multiple CPUs using OpenMP directives. OpenMP is a way to run a code in parallel in the case where all cpus share the same memory (i.e. are located on the same computer). Parallelisation across different machines connected across a network (known as 'distributed memory parallelisation') is significantly more complicated and is usually achieved using the Message Passing Interface (MPI). MPI (confusingly including the openMPI implementation) and openMP are *very* different things.

A simple openMP parallelisation of a loop would proceed as follows (in a file called `par.f90`):

```
program par
 implicit none
 integer :: i

 !$omp parallel do default(none) private(i)
 do i=1,10
    print*,i
 enddo
 !$omp end parallel do
```

```
end program par
```

If the code is not compiled in parallel then the `!$omp` is just treated as a comment. To compile in parallel, use the `-fopenmp` flag (the exact flag is specific to the compiler):

```
$ gfortran -fopenmp -o testpar par.f90
```

Exercise

1. Compare the output of the code compiled with and without the `-fopenmp` flag. Do you understand what is happening here?
2. Create a loop that evaluates a square root 1 million times. Run this in parallel and in serial and compare timings (using the `cpu_time` intrinsic function)

There is obviously a lot more to learn about writing parallel code, but this is merely to show you that such a possibility exists — you can proceed further by taking yourself through some of the openMP tutorials or documentation on the web.

2.6 Co-array Fortran

The Fortran 2008 standard defined a language standard for distributed memory parallelism within Fortran itself. This is known as 'co-array Fortran' and is now implemented in a few compilers though not yet widely used. Again, there are excellent tutorials on the web.