Introduction to git and version control

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.