Part V

Version Control

Why version control?

- When writing software, you may imagine it in terms of continual improvement.
- More likely, you will usually be making improvements, but you may want to experiment with new features, completely refactor your code, or similar.
- Or, you may manage to introduce a bug, and fail to notice it for a few weeks...
- In all of these cases, it would be helpful to roll-back your code (or parts of it) to an earlier version.
- One way to do this would be to save a copy of your program every day with a date-related filename. This way lies `EulerSolver_WORKING_1D_050422_b` and worse.

What version control provides

- Version control generally allows you to save changes made to particular files with an associated message.
- You can interrogate the system to extract an old version of the code into a separate directory.
- You can roll-back just the changes to one file without touching others.
- You can share your code with other people and ensure that you all know what your different code versions are based on.

Version control models

There are two main models for version-control:
Client-server model:

- The repository is (usually) stored an a remote server.
- All users have appropriate access to this server to extract code or to commit their changes.
- This is suited to a company model, where there is an official version of the project, centrally stored.
- Commits, and some history queries will require network access to this server.
- In some cases, the client-machine can double-up as the server.

Distributed model

Distributed model:

- The entire repository is stored on each user's client.
- If developers need to share the repository, they can either set up a central server with an official repository version, or give other developers read-access to their repository, so that they can pull changes from that repository.
- Some systems with this model (e.g. git) allow multiple repositories for one project, so you can have a local repository for your personal changes, and a centralized one for collaboration.

Version control systems

- The most popular ones in use at the moment are git and Subversion
- Other options are available, such as CVS and Mercurial, as well as other less-used open-source options, and some proprietary ones.
- Subversion was developed in the early 2000s by CollabNet, a company started by Tim O'Reilly (of O'Reilly books)
- git was developed by Linus Torvalds to aid development of the Linux kernel around 2005.

Which one should I use?

- If you are working on an existing project, then the decision will have been made for you already.
- If this is a major project (commercial or academic) that does not use version control, back away slowly...
- If you are instigating a project that will require collaboration between many people, then Subversion may be the better option.
- If you are working on your own personal project, git is probably the answer.

Git

Getting started:

- Either you will be working on an existing repository, in which case:

```
git clone https://mycode.me/git/trunk/MainCode
```

- or you will need to create your own repository:

```
git init /home/pmb39/WorkingCode
```

- Note that here the repository only exists once in your home directory. There is no central storage of the data.
- All repository history is stored in special `.git` directories.
- You should ensure that the files are all backed up.
- In order to update to the latest version of the code from the repository, use:

```
git pull
```

Basic usage

- You need to indicate that certain files are to be added to your repository:

```
git add ./Driver.C ./IdealGas.C ./IdealGas.H settingsFiles/
```

- At this point, the files are not committed to the repository, just marked as being files to be eventually put into the repository.
- Addition of directories will happen recursively, and may add more than you intended.
- To add only files that are aleady tracked by git, use:

```
git add —u settingsFiles/
```

Deleting files

- If you need to delete files from your code, then:

```
git rm ./IdealGas.C
```

- You may wish to keep the file locally on your file-system, but only delete it from the head of the repository:

```
git rm --cached ./IdealGas.C
```

- Remember that if a file has been committed to the repository then it remains there in perpetuity in earlier revisions, and you can access it by checking-out the appropriate revision.

Initial commit

- In order to commit the newly added files to the repository:

  ```
  git commit
  ```

- Any additions/changes to your files will be transferred to the repository, and the revision no. of your checked-out version will be advanced.

Other operations

- Other operations are possible:
  ```
  git mv IdealGas.C StiffenedGas.C
  ```
  will move one file to the new place, and take account of the file's history.
- If you do:
  ```
  cp IdealGas.C StiffenedGas.C
  git add StiffenedGas.C
  ```
  then `git` will (very cleverly) work out that the second file shares history with the first.
- Having this link can be useful for checking who wrote a line of code even if the code has been copied and/or moved.

Local changes

- While developing your code, you may need to check what changes you have made recently.

```
> git status .
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#       new file:   StiffenedGas.C
# 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:   IdealGas.C
#
# Untracked files:
#    (use "git add <file>..." to include in what will be
    committed)
#       MieGruneisen.C
```

- This shows that `IdealGas.C` has local changes (but not staged for commit), `StiffenedGas.C` has been staged for addition (but not committed), and `MieGruneisen.C` has not been staged for addition.

Local changes ctd - git

- While developing your code, you may need to check what changes you have made recently.

```
> git diff ./IdealGas.C
diff --git a/IdealGas.C b/IdealGas.C
index 2d7a527..83ea3fc 100644
--- a/IdealGas.C
+++ b/IdealGas.C
@@ -69,5 +69,5 @@ git commit
   const double v_y = state[V_Y];
   // Calculate kinetic energy
-  const double ke = rho*(v_x*v_x + v_y*v_y);
+  const double ke = 0.5*rho*(v_x*v_x + v_y*v_y);
   const double totalEnergy = ke + internalEnergy;
```

  which shows the differences between your local version and the staged version of the file.

- This is not (necessarily) the same as the absolute latest version of the repository.

Roll-back to lastest version

- You may decide that all the changes you have made since your last commit or update are useless.
- In this case, you can roll back your working copy to be the same as the latest commit:
  ```
  git checkout ./IdealGas.C
  ```
- which completely discards the local changes, and there is no way to recover them.

Further committing - git

- Once you have made various changes to your files, you indicate to git that you want a particular set of files to be committed to the repository:

  ```
  git add IdealGas.C
  ```

  and then commit the changes

  ```
  git commit IdealGas.C
  ```

- A text-editor will open (`nano` by default, or whatever you specify via `git config`)
- You should enter a useful log-message, e.g. “Corrected kinetic-energy calculation”.
- Save the file, and exit the editor.
- The repository is stored locally, so no authentication is required.

Notes on git's staging

- You will note that git uses a two-stage process when committing changes.
- In git, you have to explicitly schedule modified files to be committed to the repository.
- This allows you to break commits down into a set of logically separate commits, and forces you to think about each modified file.
- git therefore forces clearer thinking by default, as compared to Subversion, for example.

What should be in the repository?

- The repository should contain everything required to recompile and run your program(s).
- Source files (`.f .F .c .C .h .H` etc.), Makefiles, settings files, post-processing scripts, etc.
- It should not contain files that can be generated from these, e.g. object files, executables, PDFs of papers that are needed as documentation, and other output files.
- Most version-control systems work best with pure-text-based files, as differences are easy to determine.
- Binary (non-text) files can sometimes go into repositories, but you are unlikely to get anything useful except individual revisions of the files as they are probably in a complex and compressed format, so that `diff` does not work well.
- Pictures for inclusion in documentation are one type of binary file that might go in a repository.

Querying the repository

`git log ./IdealGas.C`
displays the commit messages associated with changes to the named file.

```
commit d6f39d56a5179d150437827abf7292f91827cbe2
Author: Philip Blakely <pmb39@cam.ac.uk>
Date:   Tue Aug 9 11:55:43 2022 +0100
    Add Ideal—Euler and Euler—Ideal mixed Riemann solvers.
    Required addition of extra functions to EoS for solving
   Riemann
    problems.
```

Working with the repository

- If you find that one part of your code has stopped working, and you are sure which file(s) are involved, you can go back to previous revisions.
- Firstly, if you have any local changes to those files, you should save a separate copy of them to avoid any loss of work. (Either **`git commit`** or **`git stash`**.)
- Then, roll those files back to an earlier revision:

  ```
  git checkout 1c1f9c2dd44044ba1ab777be89fe08ae4e351cff \
               IdealGas.C
  ```

  and compile and test your program again.
- You therefore have mixed revision-numbers of files in one source-tree, which can become confusing.
- Once you have, for example, determined that the code worked with the earlier version of the file, you can update it again:

  ```
  git checkout IdealGas.C
  ```

  and try to fix it.

Branching

- There are at least two reasons to use branches in version control.
- Firstly, you may want to develop a complex feature into your code, which will take a lot of work (i.e. many commits) to get right. A branch will allow separate development of this feature while your original version can still be used, and modified independently.
- Secondly, you may want to preserve an older version of your code, and maintain it as a bug-fix version, i.e. you can make minor corrections to it, even though you are making large modifications to the main code.
- You will notice that these are more-or-less the same thing viewed from different angles.

Branching in git

- To create a branch of your code in git:

  ```
  git branch Refactor
  git switch Refactor
  ```

- Now, the original `trunk` code version still points to the original code version, but so does `Refactor`.
- Any further commits will happen on the `Refactor` branch instead.
- To switch back to the original branch:

  ```
  git switch main
  ```

  (or possibly `development` instead of `main`).

Bisection

Imagine: You discover that your code fails to solve a standard problem correctly, even though it did so three months (and 100 commits) ago.

- Write a simple script that compiles your code and runs it against the standard problem and checks it against the correct solution.
- The script should exit with code 0 if the correct solution is found, otherwise with code 1.
- Then, in a checked-out version of your repository:

  ```
  git bisect start
  git bisect good a7398ffdb32
  git bisect bad 9e51c89b32f
  git bisect run ./test.sh
  ```

- This will use a bisection method to find the first revision where the test-case breaks.
- This will not work well if the settings-file format has changed, for example. In this case, you may need to examine the `svn-bisect` script and modify it for your purposes.

Commit separation

- What happens if you are working on two separate changes to the code at once, or notice a simple mistake while working on a major rewrite?
- Here, you should commit only the changes to one file, while keeping the rest as a local modification.
- If you have two sets of changes to one file that are somehow independent, then you will need to take a copy of the file, revert the original, and make only one set of changes before committing again, and then making the remaining set of changes.
- If this happens a lot, you may need to reconsider how well your project is divided into separate files.
- `git` only works with files at its most granular level.

VCS as documentation

- One day you may find yourself wondering why a particular line is what it is.
- Of course, this should be obvious, given that your code is well-documented...
- If not, though, you could run:

```
git blame EulerEquations.C
   5243f11e pmb39   for(int d=0 ; d < 3 ; d++)
   5243f11e pmb39   {
   5243f11e pmb39     const real V_d = prim[VEL+d];
   c1ed290f stm31     cons[MOM + d] = rho * V_d;
   5243f11e pmb39     vSquared += V_d * V_d;
   5243f11e pmb39   }
```

  which will list the file with the last revision at which each line was changed.
- You can then look up the commit message associated with that commit, which may also provide some help. This is not really a substitution for good documentation, however.

Backups

- As ever, backups are vital.
- Use the `git bundle` command or ensure that the directory containing your repository is backed up.
- If you lose the repository, you will have lost the development history of your code, even if your code is intact.
- `git` can also push commits to a remote repository, e.g. GitHub (owned by Microsoft) or GitLab (run by GitLab Inc.)

GUIs

- In some cases, it is useful to see an overview of the repository and its history in a more visual form.
- git comes with its own GUIs: git-gui and gitk
  - `git gui` geared towards staging files for commit
  - `gitk` geared towards examining repository history

Merging and conflicts

- If you are working with others on a project, then they may be making changes at the same time as you.
- The first person to commit their work has priority, so what happens if you then want to commit changes?
- There are various scenarios, which have different solutions.
- If you are working alone on your own local repository, similar problems apply, where you and ”your colleague” are the same person, but working on different branches of the project.

Scenario 1: Changing different files

- If your colleague changes a different set of files from you, then when you commit, there may be no conflicts detected.
- This is *not* the same as there not being any conflicts at all. You may have changed different files in such a way that the code no longer compiles, or has an obscure bug, or worse.
- You should therefore always update your local version from the main repsository, then test the updated code with your proposed changes, and only then commit your changes.

Scenario 2: Changing the same file(s)

- If both you and your colleague attempt to change the same file(s), then the repository will detect the conflicts with your colleague's version when you attempt to pull or merge theirs into your local repository.
- You must then resolve any conflicts before continuing.

```
> git merge trunk
Auto-merging IdealGas.C
CONFLICT (content): Merge conflict in IdealGas.C
Automatic merge failed; fix conflicts and then commit the
    result.
```

Scenario 2: Changing the same file(s)

- Scenario 2a)
  - Your changes are to different places in the same file *and* these changes do not affect each other.
  - `git` can probably resolve the former without any trouble.
  - The latter is up to you to determine.
- Scenario 2b)
  - Your changes are to the same point in the file and these changes do not affect each other.
  - This could happen if two changes are made to the same line.
  - `git` will allow you to examine the files and make appropriate edits to ensure that both your changes have effectively been made.
  - When a conflict is detected, `git` allows you to edit them in, for example, emacs, which has a plugin to make this easier.
  - Opening a git-conflicted file in `emacs` shows some SMerge regions that illustrate the conflict.

Scenario 3: Incompatible changes

- When trying to resolve conflicts between the changes that you and your colleague have made, you may discover that your modifications are completely incompatible.
- For example, you might be converting the code to use a stiffened-gas equation of state, while they are converting it to use a Hugoniot equation of state.
- At this point, you will need to pause and consider how to proceed.
- You may want to create separate branches for the code, so that you can both work on your different versions without conflicting.
- More likely, you will want to make wider changes to the code, refactoring it to be able to use any equation of state.

Summary

- This should have given you some idea of the features of git.
- It is worth trying them on a small or dummy project first to get the hang of them before using them on a major project.

EPSRC open-data

- You should be aware that EPSRC requires all publications arising from its funding to publish associated data in a public place.
- This may include source-code.
- You will therefore need to be able to extract the precise version of code used to generate published results.
- Before doing this, you should discuss this with your supervisor, as there are sometimes reasons not to have to publish source-code.

References

The main reference to `git` is freely available in PDF and web-based format and is very well written with various examples and diagrams.

- `https://git-scm.com/doc`
- `https://www.atlassian.com/git`