

---

WARWICK RESEARCH SOFTWARE ENGINEERING

# Introduction to Version Control with Git

*H. Ratcliffe and C.S. Brady*  
Senior Research Software Engineers



“The Angry Penguin”, used under creative commons licence  
from Swantje Hess and Jannis Pohlmann.

March 12, 2018

# Contents

1	About these Notes	1
2	Introduction to Version Control	2
3	Basic Version Control with Git	4
4	Releases and Versioning	11
	Glossary	14

## 1 About these Notes

These notes were written by H Ratcliffe and C S Brady, both Senior Research Software Engineers in the Scientific Computing Research Technology Platform at the University of Warwick for a series of Workshops first run in December 2017 at the University of Warwick. This document contains notes for a half-day session on version control, an essential part of the life of a software developer.

**This work, except where otherwise noted, is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.**



The notes may redistributed freely with attribution, but may not be used for commercial purposes nor altered or modified. The Angry Penguin and other reproduced material, is clearly marked in the text and is not included in this declaration.

The notes were typeset in L<sup>A</sup>T<sub>E</sub>X by H Ratcliffe. Errors can be reported to [rse@warwick.ac.uk](mailto:rse@warwick.ac.uk)

### 1.1 Other Useful Information

Throughout these notes, we present snippets of code and pseudocode, in particular snippets of commands for shell, make, or git. These often contain parts which you should substitute with the relevant text you want to use. These are marked with {}, such as

```
1 git branch {name}
```

where you should replace “name” with the relevant text.

We also include a series of glossaries, one per chapter, as well as a general one at the end. Words throughout the text that look like this: [Version Control System \(VCS\)](#) are links to these. Numbers<sup>1</sup> are links to footnotes placed throughout the text.

---

<sup>1</sup>Like this

## 2 Introduction to Version Control

This section will cover the basics of [VCSs](#), including why to use them, and then gives a quick walkthrough of git, and some information about tools such as Github.

### 2.1 What is Version Control?

Version control is also known as source code management (SCM) in context of software, although note that most systems can deal with assets<sup>2</sup> too. Version control systems are designed to record changes you make to code. They track when, and by whom the changes were made, and usually allow you to add some explanation. They allow you to go back to an old version of the code, or of just some files. They also include tools to help you to merge incompatible changes.

### 2.2 Why Use It?

Many reasons:

- “I didn’t mean to do that”

You can go back to before a breaking change

- “What did this code look like when I wrote that?”

You can go back as far as you want to the version you used for a particular talk or paper

- “How can I work on these different things without them interfering?”

[Branches](#) let you work on two features independently and only merge them at the end

- “I want a secure copy of my code”

Most [VCSs](#) have some concept of a client and a server, so make it easy to store offsite backups<sup>3</sup>

Many free services exist online, and you can easily set up your own too

- “How do I work with other people collaboratively?”

Most modern version control systems include specific tools for working with other people

Also powerful (often paid for) tools to make it even easier

For collaborative editing of a single file (e.g. papers), there are better options

---

<sup>2</sup>Such as images and data files

<sup>3</sup>Proper backups should account for the moderately likely failure of your hard drive (i.e. use an external drive) and, for important things, the quite unlikely destruction of your office (i.e. use fully mirrored system like RTPSC desktop home, files.warwick, or a cloud service)

- “My funder demands it”

More and more funding bodies expect code to be managed and made available

Online version control is one way to do this

While the basic functions are quite similar in all VCSs the more complex features often differ quite a lot. The terminology often differs too. The most likely system you’ll be using is “git”<sup>4</sup>, so that is the one we are going to talk about here. Note that it is not the only good option. You’re also likely to use some sort of online service, likely “github”<sup>5</sup>. Alternately, the Warwick SCRTP has an online system.<sup>6</sup>

### 2.2.1 Why NOT Use It?

**The most important thing about version control is to do it. It doesn’t matter how, as long as it works.** If you’re working alone, on one thing at a time, and are very conscientious, there is nothing actually wrong with simply keeping a dated copy of your files. In particular, freeze a copy every time you write a paper or run new simulations, and make sure to keep careful offsite backups (see footnote 3). This does now require more effort than using a VCS, although it will suffice for small or one-off projects.

## 2.3 A Brief History

Version control is as old as computers. The US National Archives Records Service kept copies of code on punched cards back in 1959, which managed a data density of about 100MB per forklift pallet. Important programs would be kept in the archives, and if changed a complete new card deck would be created. The birth of UNIX in the 70s gave rise to the first file-system based version control, storing file changes and allowing permissions to be set for different users (read and/or write etc).

Since then, there have been at least six major version control systems, roughly one every ten years. Several of these are currently in wide use. Those you are likely to meet at some point are

- Git: the topic of these notes
- Mercurial: <https://www.mercurial-scm.org>
- Bitkeeper: originally paid-for, now open source <http://www.bitkeeper.org/>
- Subversion: still around, needs a server running, but that can be on the local machine <https://subversion.apache.org/>
- Visual Studio Team Services: Microsoft only, but quite good

---

<sup>4</sup><https://git-scm.com>

<sup>5</sup><https://github.com>

<sup>6</sup><https://wiki.csc.warwick.ac.uk/twiki/bin/view/Main/GitServer>

## 2.4 Features of Git

Git was created by (and named after) Linus Torvalds (of the Linux Operating System) in 2005, because the system they were using, bitkeeper, removed its free community edition. Git shares many of the useful features developed by earlier version-control systems. In particular:

- Moved/renamed/copied/deleted files retain version history
- Commits are atomic (either succeed completely or fail to do anything)
- Sophisticated branching and merging system (see Secs 3.3 and 3.4 for details)
- Used a distributed storage system, where each developer has as much of the repository as wanted in a local copy and merges onto central server when ready

Note that **Git is not Github, and Github is not Git**. Github is one popular online host of git [repositories](#) but it has its own model for how to work and adds features like issue-trackers.

## 3 Basic Version Control with Git

### 3.1 Setting up a Repository

Once you have installed git, you first want to set up some basic information. We noted that git stores the author of every change, and this means you have to provide your identity. If you try the steps below before doing this, git will insist you do. Usually, it is enough to set a single identity globally, for all your git use. You do this using<sup>7</sup>

```
1 git config --global user.name "John Doe"
2 git config --global user.email johndoe@example.com
```

However, you can use several different email addresses, for example for work and for personal projects. In this case, after “git init” but before anything else, you should

```
1 git config user.name "John Doe"
2 git config user.email johndoe@example.com
```

without the global flag.

Now before you can do anything else, you have to set up a git [repository](#). You can do this in an empty directory or one already containing files. *Be careful if this directory isn't at the bottom of your directory tree as any subdirectories will also be included.* Simply type

```
1 git init
```

Now you can add files to the repo. You usually do this in two steps. First you add, or [stage](#) the change, that is get things ready, and then you [commit](#). You can add multiple files, or parts of files, before carrying on.

---

<sup>7</sup>If you copy and paste these, note that before “global” should be two hyphens

```
1 git add src/
2 git commit
```

The second line results in a text editor opening to allow you to specify the “commit message” to explain what and why you are adding. The editor can be changed<sup>8</sup> and often defaults to vim or nano.



Figure 1: Messages Matter. Don’t do this! Permalink: [https://imgs.xkcd.com/comics/git\\_commit.png](https://imgs.xkcd.com/comics/git_commit.png) Creative Commons Attribution-NonCommercial 2.5 License.

Git commit messages should follow a particular format, which originates from its use controlling the code of the Linux Kernel.<sup>9</sup> A typical message looks like

First check in of wave.f90

wave.f90 will be a demo of using a ‘‘wave’’ type MPI cyclic transfer 0- > 1- > 2 etc. in order.

The first line is the subject, and should generally be less than 50 characters. The second line must be blank. Any text here is ignored. The subsequent lines are the message body, and should generally be less than 72 characters. You can use as many lines as you like, but be concise.

You now save and exit the editor, and git gives a short summary of what was committed. If you quit without saving the commit is aborted. The state of the files we committed has now been saved. Now we can make some changes to the files, and commit those. If we just try

```
1 git commit
```

```
we get a message like
On branch master
Changes not staged for commit:
... no changes added to commit
```

<sup>8</sup>e.g. <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

<sup>9</sup>These details are surprisingly hard to find written down, and you will probably meet many people who don’t know them. Be considerate and share!

which tells us we didn't `stage` the new changes with `git add`. We can do many `add` steps before we finally `commit`. We can also see what changes have been made at any point using

```
1 git status
```

which tells us the current state of the working directory: which files have changes that have been added, which have unstaged changes, and which files are not included in the repository. If the last list is very long, you may want to use a `.gitignore` file to tell git to ignore some file types. See e.g. <https://git-scm.com/docs/gitignore>

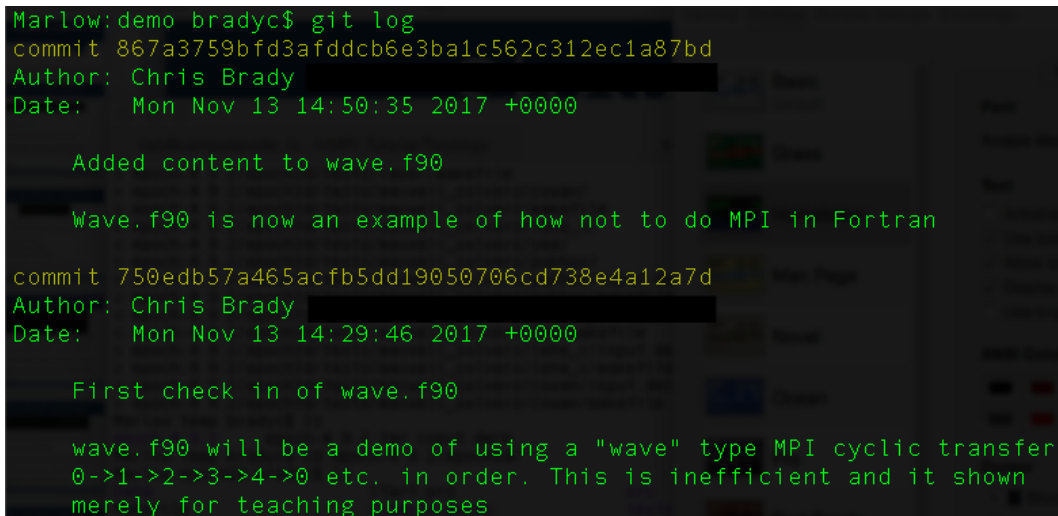
There are two useful shortcuts: for a few files that have been previously added so are known to git, we can explicitly commit them, without an `add` step like

```
1 git commit file1.txt file2.txt
```

or we can commit everything which is changed using

```
1 git commit -a
```

In all cases, we get the editor, we write a useful commit message and then we get some report like 1 file changed, 2 insertions, 3 deletions



```
Marlow:demo bradyc$ git log
commit 867a3759bfd3afddcb6e3ba1c562c312ec1a87bd
Author: Chris Brady
Date:   Mon Nov 13 14:50:35 2017 +0000

    Added content to wave.f90

    Wave.f90 is now an example of how not to do MPI in Fortran

commit 750edb57a465acfb5dd19050706cd738e4a12a7d
Author: Chris Brady
Date:   Mon Nov 13 14:29:46 2017 +0000

    First check in of wave.f90

    wave.f90 will be a demo of using a "wave" type MPI cyclic transfer
    0->1->2->3->4->0 etc. in order. This is inefficient and it shown
    merely for teaching purposes
```

Figure 2: Typical “git log” output

We can see all of the commits we have made using the log.

```
1 git log
```

gives us output like Fig 2 Note the string after the word “commit”. This is the “commit id” which uniquely identifies the commit. Git also accepts a shorter form of this, usually the first 8 characters.<sup>10</sup>

<sup>10</sup>If you know about hashes, you may know about hash collisions, where different data gives the same output. Git needs the hashes to be unambiguous. For very large projects, the first 12 characters may be needed to ensure this, as e.g. <https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection#Short-SHA-1>

## 3.2 Viewing and Undoing Changes

```
diff --git a/src/wave.f90 b/src/wave.f90
index ffaa053..c2e694e 100644
--- a/src/wave.f90
+++ b/src/wave.f90
@@ -3,7 +3,7 @@ PROGRAM wave
    USE mpi
    IMPLICIT NONE

-   INTEGER, PARAMETER :: tag = 100
+   INTEGER :: dummy_int

    INTEGER :: rank, recv_rank
    INTEGER :: nproc
```

Figure 3: Typical “git diff” output. A line referring to “tag” has been removed, a line defining “dummy\_int” has been added.

Git can show you a list of differences between two commits, or a list of differences between a given commit and the current state using the command “git diff”, as e.g.

```
1 git diff abc123xyz          #All changes since abc...
2 git diff abc123xyz efg456uvw #Changes between abc... and efg...
3 git diff abc123xyz file1.py file2.py #Changes since abc... in file1 and
   file2 only
```

The output is in a “git-diff” format:

Lines with a “+” in the left-hand gutter have been added

Lines with a “-” have been removed.

Changed lines are shown as a removed line and then an added line.

The other lines are there to give context.

You will also see some sections starting with “@@” which give the line-number and column where the changes begin. The first pair is in the original, the second in the final, version. Example output is in Fig 3.

In vim these are coloured (usually green for adds, red for removes, blue for line numbers and your default colour (here bright-green) for everything else).

Undoing changes can become quite messy. Git is a distributed system, so if the code has ever left your control, you can’t simply remove changes by changing the history, or everybody else’s state will be broken. “reverts” are new commits which remove old changes, to put things back to how they were. They leave both the original commit and the new revert commit in the log. **If you accidentally commit something protected, like a password or personal data, a git revert will not remove it. Take care, because fixing it will not be fun!**<sup>11</sup>

To revert one or more commits, use

<sup>11</sup>E.g. <https://stackoverflow.com/questions/31057892/i-accidentally-committed-a-sensitive-password-into-source-control>



```
1 git revert {lower_bound} {upper_bound}
```

where the lower bound is exclusive (last commit you want to leave unchanged) and the upper bound is inclusive (last commit you want to undo). When you do this, you will get the commit message editor for each reverted commit, saying `Revert ?original commit message?`. You rarely want to change these.

### 3.3 Branching

If you are working on several things at once, you may find branches useful. These are versions of code that git keeps separate for you, so that changes to one branch do not affect another. Whenever you create a repository, a default “master” branch is created. Adds and commits are always on the current branch. The command

```
1 git branch
```

will show the name of the branch you are on.

You can create a new branch using

```
1 git branch {name}
```

The branch is based on the last commit (on whatever branch you are on when running the command)<sup>12</sup>

The branch command doesn’t move you to the new branch. You do this using

```
1 git checkout {name}
```

You will get a message, usually `Switched to branch 'name'`, or an error message. To create a branch and change to it in a single step, use

```
1 git checkout -b {new_branch_name} {existing_branch_name}
```

where the existing branch name is optional. This is very useful when working with a branch from a remote server, for example.

Checkout also lets you go back to some previous version of the code, and create a branch from there using

```
1 git checkout -b {new_branch_name} {commit ID}
```

You can checkout old versions without changing branches too, but this puts your repository into an odd state, so is best avoided for now.

Note that if you have uncommitted changes when you run `git branch`, those changes will come with you, and can be committed. If you try and change branches when you have uncommitted changes, you may get an error, saying `error: Your local changes to the following files would be overwritten by checkout:`. You can either commit those changes, or consider using “git stash” to preserve them to use later. See e.g. <https://git-scm.com/docs/git-stash> for the latter.

---

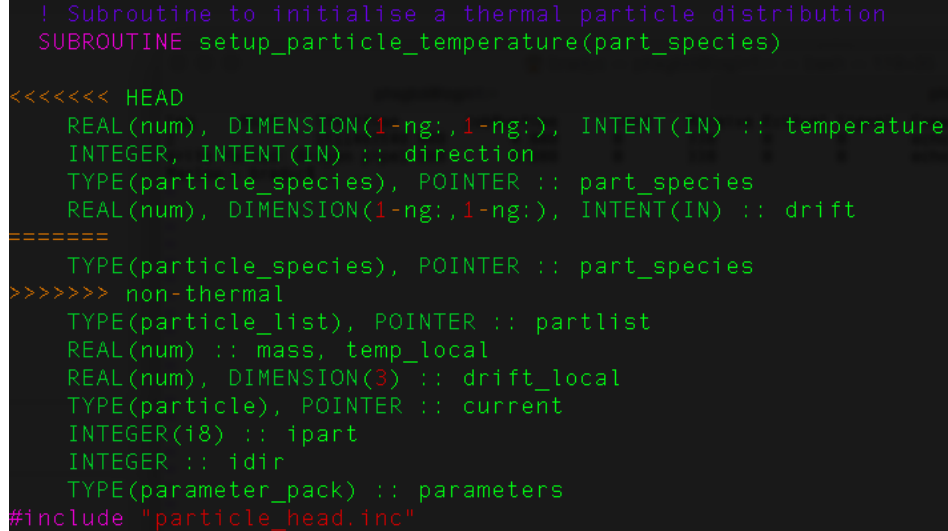
<sup>12</sup>You can branch from branches, and create very complex trees, but for now you will mostly want to create branches based on master.

## 3.4 Merging

When you use branches to develop features, you usually eventually want to bring them back into the main version, when they're ready to be shared with users, or are fully complete. This is a [merge](#), and uses the command

```
1 git merge {other_branch_name}
```

which brings changes from the other branch into the current one.



```
! Subroutine to initialise a thermal particle distribution
SUBROUTINE setup_particle_temperature(part_species)

<<<<<<< HEAD
  REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: temperature
  INTEGER, INTENT(IN) :: direction
  TYPE(particle_species), POINTER :: part_species
  REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: drift
=====
  TYPE(particle_species), POINTER :: part_species
>>>>>> non-thermal
  TYPE(particle_list), POINTER :: partlist
  REAL(num) :: mass, temp_local
  REAL(num), DIMENSION(3) :: drift_local
  TYPE(particle), POINTER :: current
  INTEGER(18) :: ipart
  INTEGER :: idir
  TYPE(parameter_pack) :: parameters
#include "particle_head.inc"
```

Figure 4: A conflicted “git merge”. non-thermal contains a change incompatible with our current branch (labelled HEAD as we’re currently on it)

If you’re lucky, the merge will be automatic and you will see a message about **Fast-forward** and are done. Otherwise, you will end up with files containing markers using the git diff format. Figure 4 shows an example. You will have to go through each file and “resolve the conflicts” (fix what git didn’t know how to merge) before git lets you commit them. When you are done, finish using

```
1 git commit      #As normal
2 git merge --continue #Alternative in newer git versions
```

There are tools to help with merges, but they can get quite complicated, and while git tries to understand the language, it is a difficult problem in general. For example, if you have changed the indentation of a whole block of code, you may see the entire thing being removed and added again, and showing as a merge conflict.

Fig 5 shows a typical flow of branching and merging. When feature 1 is complete, it is merged back to master, and the feature 2 branch pulls in those changes to stay up-to-date, before continuing work. When feature 2 is finished, it is merged too.

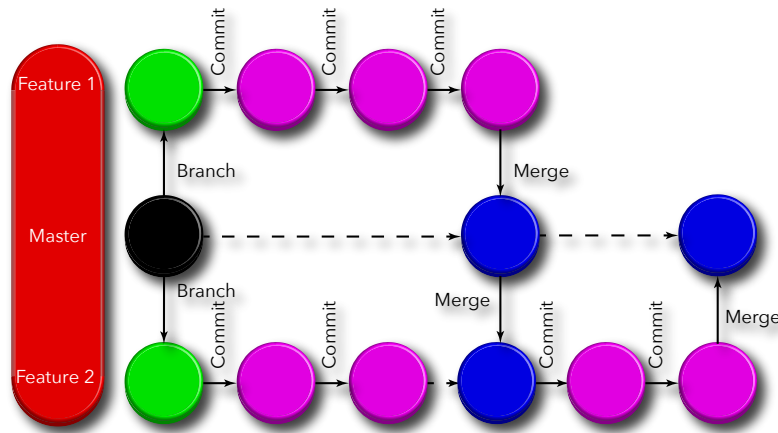


Figure 5: A schematic of typical git workflow with two feature branches and one master.

```

Marlow:demo2 bradyc$ git clone https://github.com/LMFDB/lmfdb.git
Cloning into 'lmfdb'...
remote: Counting objects: 46154, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 46154 (delta 0), reused 2 (delta 0), pack-reused 46148
Receiving objects: 100% (46154/46154), 19.45 MiB | 1.52 MiB/s, done.
Resolving deltas: 100% (34294/34294), done.
Checking connectivity... done.
Marlow:demo2 bradyc$
  
```

Figure 6: Typical “git clone” command, for a Github repo.

### 3.5 Remote Git Servers

Git is a distributed, networked version control system, which is the core of its real power. You can link between a local repository and a remote one, on a server, or on e.g. Github, and git remembers that. You can clone code from a remote repository and git will remember the origin. To clone code, you need the url of a remote server, then use the command

```
1 git clone {url}
```

Fig 6 shows an example using github - here you can get the url showing the green “clone or download” button on the repo’s page. This completely sets up the repo, and stores the “remote tracking” information (mostly the url you used). Note this will be a subdirectory of where you ran the command.

```
1 git branch -a
```

will tell you about all branches, including those on the remote you now have references to. Your master will now be linked to a remote master branch. The other branches are not downloaded by default, so if you check them out you will see similar text to Fig 6 about counting and receiving.

## 3.6 Pull and Push

When the copy of the code on the remote is updated, you will need to **pull** in those changes, with

```
1 git pull
```

This happens on a per-branch basis. Note that there is a related command, **fetch**, which just updates branch information and downloads changes, but doesn't **merge** them into yours. If your local copy has also changed, you will have to deal with merging changes from other developers with your own.

To upload your changes to the remote, you can **push** them, using

```
1 git push
```

**If you are working with somebody else's repository, check whether they allow you to push directly. On e.g. Github, a different model is used, see Sec 3.7** Git tries to merge your changes with the remote copy, so make sure to pull first, or it will fail.

## 3.7 Github Flow

Once again, github is not git. However, it is one of the most popular public remote systems, and is quite easy to use, and also adds nice features like issue trackers.<sup>13</sup> Once you sign up for a Github account, you can push a local repository to github's server. Instructions are at <https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/> or are given when you create a new repository via your github profile. A quick walkthrough of basic git for Github is at <https://guides.github.com/activities/hello-world/>

The other common task is to work on somebody else's code, and share your modifications with them and their users. They may give you push access to their github repository, but usually do not. Instead, you create a **fork** (basically a copy, but which knows where it was copied from) of their repository, push your work to it, and then make a **pull request** asking the owner of the main repository to pull (as in "git pull") the changes from your version of the repository. Figure 7 shows a typical pattern, and more details are at <https://guides.github.com/introduction/flow/>.

# 4 Releases and Versioning

## 4.1 Versions of Code

If your code produces output files, you will at some point find yourself wanting or needing to know which version of your code a particular file was created with. You could check the dates, and try to work it out, but much better is to give your code

---

<sup>13</sup>To allow your users to tell you about bugs, requests etc.

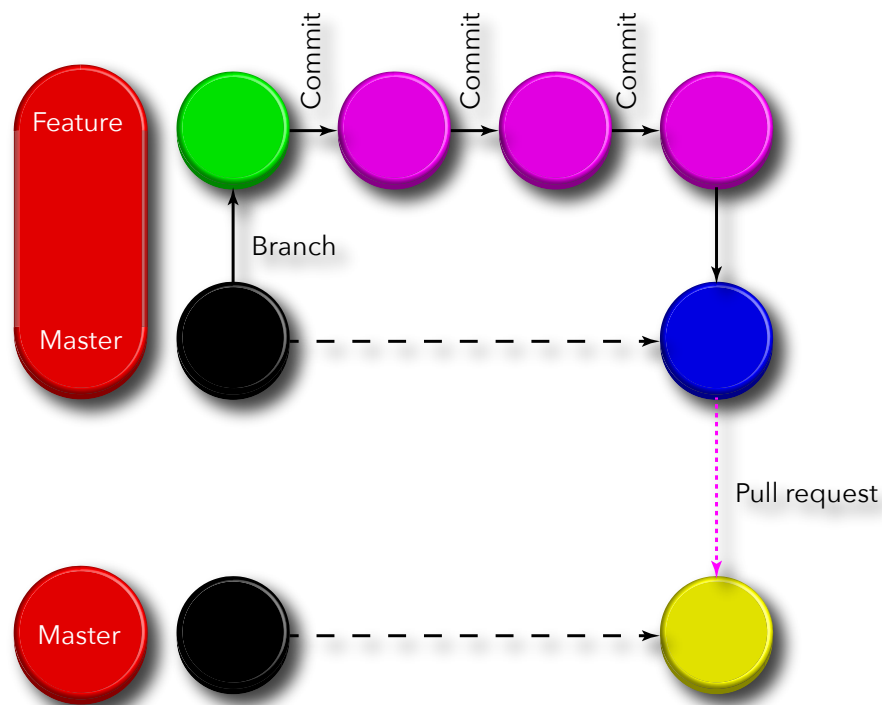


Figure 7: A schematic of typical Github workflow. You develop a feature on your fork, and then submit a pull request to have it included in the main repository.

version numbers whenever you make a change, and to write this version number into your files.

One very common, and very useful, versioning system is to use 3 numbers combined into a string like “1.1.4”. The exact purpose of the three numbers varies, but in most cases they reflect levels of change:

- 2.x.y is a major version. This usually introduces significant new features beyond those of version 1.p.q. Sometimes it means old behaviour has been removed, changed, or otherwise broken.
- x.2.y is a minor version. This usually adds smaller new features, adds new behaviour without breaking the old, etc.
- x.y.2 has many meanings. It may be called patch, build, release or otherwise, and usually starts with a number. Sometimes this is followed by a dash and an alphanumeric sequence. Usually this version represents small changes, fixes to bugs, etc.

In other words, you should “bump” the major version when results change substantially, for example you may change algorithm, or when you add some major new feature. Major version changes can break **forwards compatibility**. **Avoid breaking backwards compatibility where practical. That is, older data files should still work with**

**newer code.**<sup>14</sup> Bump the minor version when you have small changes, perhaps adding a new feature but preserving all the old. Finally you can include a third identifier which gets incremented for every change.

## 4.2 Tagging your Output Files

**The first thing to do, once you've decided on a version scheme such as the 3-part one above, is to make sure it is embedded into your output files.** This means you can know which version created the files, and that is the first step to making them reproducible. The easiest way to do this is to put a (constant) string in your source code with the number, and to have the code print it into your files. Make sure to update the string when you make changes though!

## 4.3 Git Tags

Because it is easy to forget to change the version number when you commit changed code, you can take advantage of git's way to connect a version number to a particular code state, which is to use tags. Other **VCSs** also have methods to do this. When you wish to set or to change version number, you use one of the commands

```
1 git tag {tag_name}
2 git tag -a {tag_name} -m {tag_message}
```

The latter stores your name as creator and has various other advantages, such as ability to specify a message, so is generally recommended. Using the 3-part system we may do something like

```
1 git tag -a v0.0.1 -m "Prototype version"
```

We can then find out about the tag and the commit it is attached to using

```
1 git show v0.0.1
```

Tags aren't included in a **push** by default, so you have to do the special

```
1 git push {tag_name}
```

to share them.

The major advantage of this is that you can then include some recipe in your makefile which can extract the version information and pass it on to your code. Because git tags are separate to your source code, you can go back and do this after you have committed changes. The code to extract the information is a bit horrible, but for example, if you are sure your code will only be obtained via git (and not e.g. downloaded, in which case you'll need something more complex) you add the following to the preamble part of your makefile

---

<sup>14</sup>Note that often backwards compatibility is achieved by keeping the old code, and having the system use it when given an old file, but this does often mean code duplication.

```
1 GIT_VERSION := $(shell git describe --abbrev=4 --dirty --always --tags)
2 CFLAGS += -DVERSION=\"$$(GIT_VERSION)\"
```

as described at <https://stackoverflow.com/a/12368262>, and then use the variable VERSION inside your code.

More info on using tags is at <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

## Glossary

**backwards compatibility** A guarantee that anything possible or valid in an older version remains possible, so that e.g. input or output files from an older version can still be used. Sometimes this means that you can make the code behave exactly as it used to, sometimes it means only that you can use the files as a base for new files. For example Excel can read any Excel file, from any version correctly. *See also* [forwards compatibility](#) & [sideways compatibility](#), [12](#)

**branch** Repositories can have more than one branch, an independent set of changes, often for some purpose such as a specific feature or fix. Branches can be [merged](#) together to combine their changes. [2](#), [14](#), [15](#)

**commit** (A commit) A chunk of changes, usually along with a message and an author etc. (To commit) To record files or changes in the version-control system, i.e. to add its existence and state to the history and store files and content however the system decides (often as incremental diffs). [4](#), [14](#), [15](#)

**fetch** To download changes to a repository. In git, this includes information on new branches but does not update your local copy of the code. *See also* [pull](#), [11](#)

**fork** To split off a copy of code to work on independently. Often this implies some sort of difference of opinion as to how things should be done. In the Github model, forks are encouraged in normal development: one forks code, adds features and then may or may not make a [pull request](#) back to the original repository. This way only core developers can commit to the main repository, but anybody can easily modify the code. [11](#), [15](#)

**forwards compatibility** A guarantee that files etc from a newer code version will still work with the older version, although some features may be missing. E.g. a file format designed to be extended: extended behaviour will be missing, but will simply be ignored by old versions. *See also* [backwards compatibility](#) & [sideways compatibility](#), [12](#)

**master** A common name for the primary [branch](#) of a code. Master should always be working and ready to release, as it is the default branch in git. Some consider it

a bad idea to [commit](#) changes directly to master, preferring to work on branches and then merge or rebase.

**merge** Combining one set of changes with another, usually by merging two [branches](#). The combined version usually shares its name with the branch that has been merged “onto”. *See also* [rebase](#), [8](#), [11](#), [14](#), [15](#)

**pull** To download changes to a repository. In git this then integrates them with your local copy. If you have local changes, the remote changes are [merged](#) into yours. [10](#)

**pull request** A request to pull-in code from elsewhere. In the Github model, one [forks](#) code, makes changes, and then raises a pull request for them to be integrated back to the original repo. [11](#), [14](#)

**push** To upload your local state to a remote repository (see [repository](#)). You can push commits, whole branches, git tags etc. [11](#), [13](#), [15](#)

**rebase** Replay changes as though they had been applied to a different starting point. Because systems like git work in terms of changes to code, they can go back through history and redo changes from a different base. For example, one can “rebase” a [branch](#) onto another. The changes in the first branch are taken one by one, and applied to the second branch. This differs from merging mainly in how changes are interleaved in the history. *See also* [merge](#),

**repository** (Aka repo) A single project or piece of software under version control. In general a local repository is a working (in the sense of “to be worked on”) copy of the code, whereas a remote repository is a copy everybody shares, [pushing](#) their work and combining changes. The remote copy can be sitting on somebody’s machine - remote is a designation not a requirement. Note that git does not require a remote repo (or server), but some systems like subversion do. [4](#), [15](#)

**sideways compatibility** A guarantee that code remains compatible with other code. For example you may create files for another program to read, and you want to make sure that your output remains compatible with their input requirements, even when these may change. *See also* [forwards compatibility](#) & [backwards compatibility](#),

**stage** Staging means preparing changes to be added ([committed](#)) and comes from the similar concept in transport, [https://en.wikipedia.org/wiki/Staging\\_area](https://en.wikipedia.org/wiki/Staging_area). [4](#), [5](#)

**VCS** Version Control System; a tool for preserving versions of code and details about who, why and when they were created. [1](#), [2](#), [3](#), [13](#)