What is Git?

Git is a version control system to efficiently manage and track code modifications locally and interface with collaborative coding spaces like e.g. GitHub or Bitbucket. Its most important uses are:

- tracking code changes and different versions
- tracking who made changes
- coding collaborations
- code back-up

Since the majority of plasma and astrophysics codes are hosted on GitHub, this is the platform we will focus on here.

How do Git and GitHub work?

Git is a program that you use locally on your computer to register updates to your code. To start working with Git on a project, Git converts the directory containing the project files to a **repository**, which, simply speaking, is a directory with a logbook. Each entry in a repository's logbook is called a **commit**, which contains the changes you made, when you made them, and adds a short description why the change was made. Commits are *not created automatically*, but require the user to log their changes when they are done editing.

Each repository on your local machine can be set up to connect to one or more **remote** repositories, hosted on GitHub. When new commits are made, either by you or collaborators, commits can be sent to or retrieved from a remote repository.

Getting started with Git

To convince you of Git's ability to improve your quality of life, this section walks you through setting up your first repository and how to manage it correctly.

Installing Git

If you run a Unix system, there is a chance Git is already installed. To check whether it is, open a terminal window and run

```
git --version
```

If it is already installed, you should get a message like `git version X.Y.Z`. If not, you can install git using the following commands, based on your operating system:

- MacOS (using Homebrew): `brew install git`
- Debian / Ubuntu (apt / apt-get): `sudo apt update; sudo apt install git`
- Fedora (dnf / yum): `sudo dnf install git` or `sudo yum install git`

In any case, you can verify the installation afterwards with `git --version`.

Configuring Git

When collaborating on a project, Git tracks who committed which changes. To automatically add this information to your commits, it is important to configure Git correctly. To do so, run the following commands to set this information for the current user (i.e. you only have to do this once):

```
git config --global user.name "<your name>"
git config --global user.email "<your email address>"
```

You can check everything is configured correctly by running

```
git config --list
```

If you need to change this information for a specific repository, you can do so by running the above commands inside the repository and changing `--global` to `--local`.

Creating your first repository

For this example, we will assume we are writing a paper in LaTeX for which we want to track our revisions. We start by creating a new folder called *paper* and initialising Git to turn it into a repository:

```
mkdir paper
cd paper
git init
```

This should return `Initialized empty Git repository in <path>/paper/.git/`. It may also mention which name it is using for the initial branch. We will come back to this later. Now, to add some content.

Initial commits

Oftentimes, if you know how a project will be structured, you can just create a first commit with some simple files. In our example, we are writing a paper in LaTeX, so we will probably have at least

- a main .tex file, `main.tex`
- a bibliography .bib file, `bibliography.bib`

Start by creating two empty files, `main.tex` and `bibliography.bib`.

If we now run

```
git status
```

it will return

```
On branch <branch name>

No commits yet

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

nothing added to commit but untracked files present (use "git add" to
track)
```

For now, focus on the part following `Untracked files`. Before committing, we need to tell Git which files (or modified files) we want to add to the commit. We will create two commits now, one for each file, starting with `main.tex`. Run the following:

```
git add main.tex
git status
```

Again, we get a similar output:

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

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

Note that `main.tex` is now **staged**, i.e. it will be added to the next commit, whilst `bibliography.bib` is not. To commit it, type

```
git commit -m 'init'
```

Here, the `-m` flag (message) is followed by a short descriptor of the changes made. In this case, we are simply initialising the chain of commits.

Running `git status` again now, `main.tex` has disappeared from the list since there have not been any changes to that file since the last commit. Now we repeat this for the bibliography:

```
git add bibliography.bib
git commit -m 'bibliography added'
git status
```

The last command will now return `nothing to commit, working tree clean`, indicating that there have been no changes in the repository since the last commit.

To check out the logbook, run

```
git log
```

This should now return two entries of the form

```
commit <commit hash>
Author: <user.name> <user.email>
Date: <date> <time>

    <commit message>
```

The <commit message> show the message we entered when making the commit. The other important bit of information is the string of numbers and letters called the **commit hash**. Each commit is assigned a unique hash such that it can be referred to later if needed. Also note that the last commit states `(HEAD -> <branch name>)` after its hash. We will return to this in a moment.

Staging and unstaging multiple files

Now, create a directory for images `img/` (Git will recognise your repository has subdirectories) and add a figure to it, say `image.png`. Let us modify `main.tex` to display the image and compile it with LaTeX to produce `main.pdf`. To stage all the new files at once for a commit, use

```
git add .
```

Running `git status` reveals that all new files have been committed, but if we did not clean up after the LaTeX compilation, we may have staged a bunch of auxiliary files that we do not necessarily want to track, like `main.aux`. To undo this, we use the `restore` command,

```
git restore --staged main.aux
```

or

```
git restore --staged .
```

to unstage all files again.

Managing commits

But what if you accidentally committed the auxiliary files because you forgot to check the staged files with `git status`? As long as you have not sent your commits to a remote repository, we can simply undo the commit. To try this out, commit all files

```
git add .
git commit -m 'compilation'
```

With `git log`, we will now see the statement `HEAD -> <branch name>` after our latest commit. To undo this commit, we simply tell Git to move the HEAD back 1 commit:

```
git reset --soft HEAD~1
```

With the `--soft` flag, this removes the last commit and moves the committed changes back to the staged status, after which we can simply unstage the unwanted files. Instead of moving the HEAD by 1, we can undo multiple commits at once or replace `HEAD~1` with a commit hash to move back to that specific commit. The `--soft` flag can be replaced with `--hard` if the changes are to be discarded rather than moved to the staged status.

Important: Do not do this if the commit was already sent to a remote repository! This will make your local version diverge from the remote repository, and you will not be able to send your update to the remote without overwriting history. This can lead to severe issues when multiple people are contributing to the same remote repository.

Excluding files from commits

To avoid unwanted files from getting staged, you can tell Git to ignore certain files or file extensions. There are two ways of doing this: locally and for everyone interacting with a remote repository.

If you want to prevent your collaborators from adding certain files / file types to your remote repository when you set it up, you have to add a `.gitignore` file to your repository. Inside this file, we can add specific files or all files with a certain extension. For a LaTeX repository, I would add the following to my `.gitignore` file:

```
main.aux
main.log
main.synctex.gz
```

This excludes these specific files from staging. If we want to exclude all files of these type (e.g. there will be more than one .tex file to compile), instead we can use the `*` operator,

```
*.aux
*.log
*.synctex.gz
```

If we execute `git status` now, we see that the unwanted files are no longer listed. Similarly, if we use `git add .` now, they are not staged. *Note*: `.gitignore` does not have to be committed to do its job, but you do have to commit it if you want to include it in a remote repository and work its magic there.

If you are instead excluding a local file that other users of your repository don't have, you would add the corresponding line to `.git/info/exclude` instead of to `.gitignore`.

Branching out

As you might have noticed every time we called `git status`, we have been working on the `master` or `main` branch so far. However, a single directory can have multiple branches, and, if you are collaborating with others on a project, it probably should. During code development, the main branch is usually reserved for the latest release, whilst other branches are used for developing new features before they are released.

Let us assume we are setting up a new branch to work on the revision of our paper. Type

```
git branch revision
git branch
```

The first command creates a new branch called `revision`, which is a copy of the current branch with the same history. The second command simply lists all branches in the directory, marking the current branch (`master`) with an asterisk. To change branch to the new branch `revision`, type

```
git checkout revision
```

Alternatively, you can also combine the creation of a new branch and moving to it in a single command with the `-b` flag,

```
git checkout -b <new branch>
```

When you make a commit on a specific branch, it does not affect any other branches. To demonstrate this, add your favourite book to `bibliography.bib` and commit it with the message *my favourite book*. We can now checkout the differences between the `master` and `revision` branch with the `diff` command, like so:

```
git diff master revision
```

Working with different branches comes with a caveat though: you cannot switch between branches when there are untracked changes. If we want to checkout a different branch without committing the current changes, we use

```
git stash
```

This removes any untracked changes and places them in the **stash**. Once you do this, you will be free to check out different branches. To reapply stashed changes, simply type

```
git stash pop
```

The stashed changes will be reapplied on the *current* branch. If at any point you do not need your stashed changes anymore, you can instruct Git to forget them with

```
git stash drop
```

Merging

Once you are done editing on your secondary branch, you can **merge** the changes into the `master` branch. To do so, first navigate to the `master` branch before using the `merge` command,

```
git checkout master
git merge revision
```

This will add the changes on the `revision` branch into the `master` branch.

Depending on when the two branches diverged, a couple of things may happen:

- If the `revision` branch contains all the commits on the `master` branch and more, the additional commits are simply appended to those present on the `master` before the merge. This is what happens in our example.
- If the `master` branch received new commits since the creation of the `revision` branch, a true merge will have to occur. If the changes in the `revision` branch do not alter the same parts of the files in the new commits on the `master` branch, Git will merge the changes automatically. However, if the same lines of code were altered in both branches after the branching point, Git will flag these lines as **merge conflicts** until you resolve them manually.

Merge conflicts

When Git reports merge conflicts, `git status` will tell you which files are affected. To check out the conflict, open the file in the terminal or an editor and look for

- `<<<<<<< HEAD`
- `=======`
- `>>>>>>> revision`

The part between `<<<<<<< HEAD` and `=======` gives the content in the `master` branch (the branch being merged into) whilst the part between `=======` and `>>>>>>> revision` represents the content in the `revision` branch (the branch being merged). Replace this whole block, including the dividers with appropriate text/code, stage the file and commit it with a message like *Merge conflict resolution in <file>*.

Personally, I recommend handling merge conflicts in Visual Studio Code or Atom. Both offer Git functionalities for easy resolution of conflicts.

Setting up GitHub

To set up a GitHub account, go to GitHub, select *Sign Up* and follow the instructions. To connect to GitHub from the terminal, we have to add an ssh public key (or similar protocol) to the account. If you do not have an key pair yet, you can create one first with

```
ssh-keygen -t <key type>
```

where common `<key type>` options are `ed25519` and `rsa`. For GitHub, the key type does not matter that much.

Once you have a working key pair (usually located in `$HOME/.ssh/`), go to GitHub and log in with your email address and password. Press the account avatar in the top right corner and select *Settings*. In the menu to the left, navigate to *SSH and GPG keys*. Once there, select *New SSH key*. Give it a name to identify it (usually a reference to the computer that uses the key) and copy the contents of `<key name>.pub` into the *Key* field. Set *Key type* to *Authentication Key*. Select *Add SSH key* and you are good to go!

Creating and linking a remote repository

On your GitHub dashboard, select the green button with the book icon *New*. Choose a repository name, say `paper`. You can add a description if you like. Set the repository to *Private* (use *Public* if you want the contents to be accessible to anyone; your repository will also show up in searches then). Since we will connect this remote to our local repository, you do not need to add a README file, a .gitignore file or license at this point. When you are done, select *Create repository*.

Under the *Quick setup* header, select SSH and copy the link. It will look like this:

```
git@github.com:<username>/paper.git
```

Now go back to the terminal and navigate to the repository we created previously. To tell Git to which remote it should connect, type

```
git remote add origin <copied link>
```

Note that `origin` can be replaced with any identifier you like. `origin` is the traditional choice if you are adding only one remote repository.

Once we have supplied Git with a remote link, we can move our content to the remote:

```
git push --set-upstream origin master
```

The `push` command takes two arguments: the remote identifier we chose above and a branch name. If the branch name you give is not present yet, GitHub will create a new remote branch from your local branch. It is good practice to use the same branch names locally and remotely. The flag `--set-upstream` tells the local branch to push to `origin master` if we do not provide `git push` with arguments next time. Now check out the repository on GitHub!

If we add a markdown document called README.md to the repository, GitHub will display it by default. Now create a file README.md locally and put in the text

```
# My first GitHub repository
This is my first remote repository!
```

Stage it, commit it and type

```
git push
```

without any arguments. It will push to the master branch automatically thanks to the `--set-upstream` flag earlier!

Now our `master` and `revision` branches have diverged again. Let us bring them in line again by pulling the information from GitHub:

```
git checkout revision
git pull origin master
```

This will pull the changes from the `master` branch in the remote repository and apply them to our local `revision` branch. Note that you cannot pull a remote branch if you have local, untracked changes. Again, we can use `git stash` before pulling and reapplying the changes after the pull with `git stash pop`.

To reiterate: `push` to send local commits to GitHub, `pull` to download remote commits from GitHub.

Pull requests

To collaborate on a larger project (especially code development), GitHub introduces **pull requests**. A pull request is simply a request to merge a branch into another one. This allows collaborators to review the changes and if necessary, leave comments or make modifications, before the changes are merged into the main branch.

Navigate to your local `revision` branch and add a line to `main.tex`. Commit your change and push it to a new remote branch `revision` with

```
git push origin revision
```

On GitHub, beneath the repository name it will now say `2 Branches`. In the dropdown menu to the left, select `revision` now. It should now say

```
This branch is 1 commit ahead of master.
```

and there may already be a button saying *Compare and pull request*. Press this button or navigate to the right and select *Contribute* > *Open pull request*. Give it a title and description, and select *Create pull request*. This creates an overview of the new commits and the option to comment. Note that it is available in the tab *Pull requests* at the top. Once it has been reviewed, and if it shows

```
This branch has no conflicts with the base branch
```

it can be merged by pressing *Merge pull request*. Do so now. If you navigate back to the *Code* tab, you will see that this merged `revision` into `master` and created a new commit called

```
Merged pull request #1 from <username>/revision
```

Cloning and forking

To conclude this tutorial, we will **clone** and **fork** the repository. Navigate to the *Code* tab of your repository on GitHub and select the green *Code* button. Copy the SSH link and open a terminal in a new location. Type

```
git clone <copied link>
```

to create a local repository which is a copy of the GitHub repository's `master` branch. By default, the GitHub repository will automatically be set up as the remote with the name `origin`. This is the easiest way of obtaining full codes hosted on GitHub.

If you wish to contribute to public codes on GitHub, the proper way of doing so is by forking the repository, committing your changes there, and opening a pull request from your fork to the original repository. On your GitHub repository page, select *Fork* and give it a new name. To set up the fork in your local repository (the cloned one), copy its SSH link and run

```
git remote add fork <fork link>
```

You can now make changes locally, commit them to your own fork, and if desired, open pull requests on GitHub from your fork to the original repository. This is safest way to avoid complicated errors.

For more, check out the GitHub documentation!

Good luck out there. Go Git 'em!