Git, Testing and Projects

Researchers need to be using a source code control system such as Git for backup, reprodicibility, sharing and general good software practices. Having tests for code is also part of good practices. There are many ways of organising things, here I explain one method. I also make use of MATLAB Projects which are MATLAB-specific, but most of the discussion around Git, GitHub and testing is common to many languages. First a few explanations:

Git - a software package that can keep track of changes you make to your code. Changes are recorded in a repository and you can link a local repository on your computer, to one in the cloud (e.g. GitHub). This linking provides an external backup and allows sharing or co-development with others.

Testing - functions dedicated to testing parts, or all, of your code. Tests can help to chack code is doing what you intended, but in research perhaps their primary benefit is that they enable code to be modified and extended whilst remaining confident it still works. An example might be if an input data format changes and you adapt code to read the new data. If a test framework is in place, you should easily be able to check the new code still reads older format data sets.

MATLAB Projects - a folder structure with additional functionality that helps to monitor changes, control the path and other useful features.

This all takes time, when shoud I start?

Mess around and explore as you wish, but as soon as something starts to look useful, or will be needed in more than a few days time, it should be in Git. At the time of submission to a conference or journal, or sharing code with colleagues, it should be in Git and have some tests available.

How should I get started?

The hardest part is linking to GitHub so if you are new to this, I suggest first experimenting using Git locally (note this will not make an external backup of local changes).

- create a new blank Project,
- associate that Project with a new local Git repository (repo),
- add code, experiment with Git branching and commits.

To do this:

1. In MATLAB, Create a blank Project: **Home Tab** -> **New** -> **Project** -> **Blank Project**
2. Add this Project to local Git source control: **Project Tab** -> **Use Source Control** -> A**dd Project to Source Control ...**
3. Commit the Project as it is now: **Project Ta**b -> Press the **Commit** button (in the Source Control part of the Toolstrip). Enter a commit message, e.g. "First commit".

Good practice now would be to create a branch on which to make changes. You can use the MATLAB Branches button, or manage Git with third party software - I use Sourcetree as it has a good interface. You can also use Sourcetree to store your name and email that will be included with all Git commits.

Ths YouTube video from Kevin Stratvert explains Git and GitHub well.

Project Structure

Projects have a root folder that contains files and further folders. A recommeded structure for a project called `stellaranalysis` is below. Intitially, you will just have the top files and folders. The `source` and others listed below will be added later.

```
stellaranalysis/
|   .gitattributes
|   .gitignore
|   README.md
|   LICENSE
|   stellaranalysis.prj
├───.git/
├───resources/
:
|---source/
|---tests/
|---buildfile.m
└───data/
```

Add `.m` files to a `source` folder. After adding files and folders, they need to be included in the Project. From the **Project Tab** (Tools section) -> **Add Files** and select the files/folders to add. Including in a Project this way helps with dependency checks and other useful features.

Linking to GitHub

This set-up is harder than routine use! Once you are familiar with the processes, it only takes about a minute to set up a new repo and Project.

1. In your MATLAB Project, make sure you have nothing like passwords, bank details, private addresses/ phone numbers etc in your files or folders. If you create a Public repo below, anyone will be able to see the files you commit.
2. In your local MATLAB Project, make sure you have commited any changes.
3. Get a GitHub account from github.com. Choose a username that you don't find emabarrasing and if you are part of an institution that pays for private GitHub repositories (such as UCL), register with your instiutional email. Set up passkey or two-factor authentication. Note this authentification is only for logging in to github.com website, not for the communication (`push` and `pull`) which needs additinal set-up.
4. Login to GitHub website. If you want to use your organization context, switch to that (arrow next to your username). Press the "+" button to select **New Repository**.
5. In the *Create a new repository* page, under Owner, choose the organization you want to use, type a Repository name. Choose Public or Private (if available) as you wish. Don't intialize with a README or license (we will do this later). Press Create Repository. The next page contains important information.
6. There are two ways for local repos to communicate with GitHub - https or ssh. The https route offers more flexibility and control but it needs Personal Access Tokens and some aspects are still in beta so here we will use ssh. You only need to set-up ssh once per computer.
7. On the GitHub page, push the SSH button in the blue box. You will see the URL for this new github repository e.g. `git@github.com:UCL/stellaranalysis.git` (in this example 'UCL' is my organization and '`stellaranalysis`' is the name of my repository on GitHub). Copy this URL. In MATLAB, press the **Remote** button (in the Toolstrip near the Commit). Paste in the URL and press **OK**.
8. ssh needs to be set-up once for you and the computer you are using. It should then work for multiple repos on GitHub. On GitHub, from your photo/logo (top right), select **Settings**. Then on the left, under Access, select **SSH and GPG keys**. You will need to follow the instructions for adding a **New SSH key**.
9. Once set-up, in MATLAB pressing the **Remote** button and then **Validate** should result in success.
10. In MATLAB, press the **Push** button. If you now press the **Branches** button, you should see reference to your local `main` branch and `origin/main` (which is on the 'remote' or GitHub). .
11. Go to the GitHub web page for this repo (if you have lost it, use the URL https://github.com/UCL/stellaranalysis , replacing the 'UCL' and 'stelaranalysis' with your organization and repo name.) On the `Code` page, you should see the files and commit messages.
12. Still on the GitHub page, press the **Add a README** button. You can edit and preview ths file. Then press **Commit changes ...** and then **Commit** changes.
13. At this stage, your repo on GitHub is ahead of your local one because the README.md file has been added to the repo on GitHub.
14. In MATLAB, press **Pull**. This will pull the README.md file into your local repo.
15. Back on GitHub, you can add a LICENSE file. Go to add file, create new file, type "license" and then you will get a button appear that says `Choose a license template`. Follow the instructions from there and commit the file to the main branch. Back in MATLAB, **Pull** to sync the repos.
16. From here, just edit, commit and Push from within MATLAB.

Testing

I recommend using MATLAB's Class-based tests. These are xUnit-style (used by other languages so the effort required to learn is transferable). MATLAB documentation provides a lot of help with these. Here I will just provide some personal observations and opinions:

- tests are primarily to check that code changes do not break things, rather than validate numerical correctness
- write the underlying code defenisively, for example, check the inputs to a function by using `argument` blocks, or explicitly in the main code. I wouldn't normally then write test code to check the checks.
- writing code in isolated functions makes it easier to test.
- target test code at areas where future changes might be error prone. This requires judgement, but typical areas for problems include code that processes input files, code that is very slow or otherwise ripe for re-factoring.
- in research it can be hard to know what the 'correct' output should be - after all you are likely to be doing something novel. Consider if there is a set of inputs with known output eg. all 0. Otherwise, the output from a 'working' version can be saved as used for comparison with later versions.
- The last point can be problematic because outputs might differ due to floating-point rounding errors and small differences in algorithm parameters etc. In such cases, outputs will only agree within a tolerance and this tolerance can be domain specific, or even vary within data - for example, regions of data that have noise and no signal might legitametly have large relative changes in outputs that have no physical consequence. The art here is to write a test that will allow through reasonable differences, but catch things that are wrong. It is worth trying to capture this in a test (rather than only relying on human inspection) because it becomes more transparent what you regard as 'correct' and enables automation.