Tools for reproducible science

Christophe Pallier

Sept. 11, 2018

Reproducible Science

You should strive to make your experiments, simulations and data analyses reproducible, that is, anyone should be able to check what you did, and reproduce it. *You* should be able to understand what you did, even years later.

This means that you should:

- keep track of exactly how you selected your materials (not only the end result)
- keep track of the precise recipes for the data analyses

This is very difficult to achieve! Here are some possible strategies:

1. keep a detailed lab notebook (few people manage to do it properly)
2. write computer scripts that automate the processing pipelines.
3. give up, hope you have not made mistakes, and will not need to check or rerun the experiment.

Although 3 is still the most widespread strategy among scientist, I cannot recommend it!

Two tools: literate programming and version control

Literate programming mixes code and text to produce a human readable document. It goes way beyond simply documenting one's code with comments. For example:

- rmarkdown:
  - comparing two treatments with R (source)
  - see more at `https://github.com/chrplr/statistics_with_R`
- jupyter notebook:
  - tutorial on Fourier analysis
  - tutorial on fMRI data analysis
  - see more at `https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks`

Remarks:

1. There are problems with tools such as Excel, E-Prime, . . .

- they are impossible to check thoroughly.
- the files are saved in binary format (not human readable)
- the compatibility is not assured between successive versions.

2. In literate programming tools, text is often written in the markup language **Markdown**. You should definitely learn it! (see `https://www.markdownguide.org/getting-started`). It is good enough to produce scientific reports (LaTeX not needed!).

Version control systems (e.g. `git`, `svn`, `mercurial`, ...) are useful for two reasons:

1. They allow you keep track of the complete history of a project. While your scripts or documents improve, it keeps *cached copies* of the previous versions. This is much, much better than M using a numbering scheme and multiple copies of the same file or directory (myfile001.doc, myfiles002.doc...).

2. They facilitate collaborating on projects (documents and programs) with other people.

- **Bad**: use email's attachments to exchange successive versions of a file.
- **Good**: collaborate on a project by using a shared git repository

To go further;

- "Software Carpentry" : a site dedicated to teaching basic computing skills to researchers. `https://software-carpentry.org/lessons`

- INRIA's MOOC "Recherche reproductible: principes méthodologiques pour une science transparente" `https://learninglab.inria.fr/mooc-recherche-reproductible-principes-methodologiques-pour-une-science-transparente/`

- Simon Tatham "How to Report Bugs Effectively" `https://www.chiark.greenend.org.uk/~sgtatham/bugs.html`

Lesson 101: how to compare files or directories

The most basic task is to compare two files. Your text editor may already have such a function built in — for example, in Emacs, it is accessible through the command `ediff` or the menu *Tools/Compare*).

You can also compare two files on the command-line with the command `diff`:

```
diff file1 file2
```

But I recommend another command, `meld`, as it has a more user-friendly output:

```
meld file1 file2
```

(you may need to install the program `meld` with `sudo apt-get install meld` if you are under Linux, or from `http://meldmerge.org`)

These tools also work on directories. To quickly check if there is any difference between two directories:

```
diff -r -q dir1 dir2
```

or

```
meld dir1 dir2
```

Note: If you compare text files that contain natural language, I recommend `wdiff` which ignores changes in whitespaces (line breaks, etc.). For latex files, `latexdiff` produces a formatted output that clearly shows the textual differences.

Introducing git

A version control system keeps track of the history of changes made to a set of documents and allows to recall specific versions later.

Many use a numbering scheme to keep track of the evolving versions of files, but this is not a good idea, especially when collaborating with several people.

To keep track of changes made to files in a directory, I highly recommend that you use a *version control* software. Personally, I use git.

In this document, I just describe a few basic git commands. You can teach yourself by following one of these tutorials:

- `https://backlogtool.com/git-guide/en/`
- `https://www.atlassian.com/git/tutorials`

The complete documentation is in the Git Book which is quite readable. To understand how git works, I recommend the Git Parable and Git from the bottom up.

Installing git:

- For Windows users, download the installer from `https://gitforwindows.org` and execute it (accept all the proposed default settings, notably the 'Use Git-Bash' and use 'Mintty').

- For a Debian-based Linux based system:

  sudo apt-get install git gitk

- For other systems, follow the instructions on `https://www.atlassian.com/git/tutorials/install-git`

- To configure git to use Meld for file comparison and merging: `https://marcin-chwedczuk.github.io/use-meld-as-git-merge-diff-tool`

Creating a local repository

A *local repository* is simply a folder where you have ran the command `git init`.

```
mkdir project
cd project
git init
Initialized empty Git repository in /home/pallier/cours/Python/version_control/git-test/.git/
```

Often, you will work on a local copy of a *remote repository*, imported either from another directory or from the Internet using `git clone`:

```
git clone https://github.com/chrplr/pyepl_examples
```

Note: If, when you create a repository you already know that you want to share it on the internet, I recommend to first create a repository on `http://github.com` or `http://bitbucket.org`, and then *clone* it on your local hard drive. In this way, the internet location will be automatically added to the list of remote repositories under the name `origin`.

Importantly, with git, you can still do version control locally, and only transfer your changes to the remote repository whenever you want, or never, because git is a *decentralized* version control system and all repositories are equal.

Adding files to the local repository

While working on the `project`directory, you can tag files to *track* using the `git add` command:

```
echo 'essai1' > readme.txt  # create a file "readme.txt"; you can also use an editor like atom
git add readme.txt
```

To check which files are currently being *tracked* (or *staged* in git's terminology), use the command 'git status“:

```
git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   readme.txt
#
```

Note that you can add entire directories, for example:

```
git add .
```

It is possible to prevent certain files to be tracked (see https://help.github.com/articles/ignoring-files)).

Creating a commit (a.k.a. committing)

Once you are satisfied with the files in your working directory, you can take a *snapshot*, that is make a permanent copy of all the tracked files. This operation is also called *commiting* your changes:

```
git commit -m ’my first attempt’
[master (root-commit) a7a3a47] First commit
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
```

This saves a snapshot (or *commit*) of the staged files in the hidden directory `.git` at the root of your project. Unless you delete this directory, this version of your files is saved there forever and will always be accessible.

Note: Before commiting, it is always useful to check which files are tracked and which are not, with `git status`.

Modifying the project

Let us now modify the file `readme.txt` in the working directory:

```
echo ’line2’ >> readme.txt
```

The command `git status` allows us to check the state of the files in the working directory:

```
git status
# On branch master
# 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:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")


git add readme.txt
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

Let us create a new file, ‘readme2.txt“:

```
echo ’trial2’ >readme2.txt
ls
readme2.txt  readme.txt
git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       readme2.txt
```

We now add `readme2.txt` to the repository:

```
git add readme2.txt
git commit
[master a7e25a1] First revision; added readme2.txt
2 files changed, 2 insertions(+), 1 deletion(-)
create mode 100644 readme2.txt
```

Let us consult the history of the project:

```
git log
```

```
commit a7e25a158ce52a75c62381420f7dc375de631b1b
Author: Christophe Pallier <christophe@pallier.org>
Date:   Mon Aug 27 10:49:54 2012 +0200

First revision; added readme2.txt

commit a7a3a47edfae9d7c720356b691000a81ded73906
Author: Christophe Pallier <christophe@pallier.org>
Date:   Mon Aug 27 10:47:32 2012 +0200

First commit
```

```
git status
# On branch master
nothing to commit (working directory clean)
```

Renaming a file

To rename a tracked file, you should use `git mv` rather then just ‘mv“:

```
git mv file.ori file.new
```

Recovering a file deleted by accident

Let us delete readme2.txt “by accident”:

```
rm readme2.txt # oops
ls
readme.txt
git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    readme2.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

To recover it:

```
git checkout -- readme2.txt
ls
```

```
readme2.txt  readme.txt
cat readme2.txt
trial2
```

Checking for changes

Let us now modify readme2.txt and then compare the file in the current directory from the ones in the last commit:

```
echo 'line2 of 2' > readme2.txt
git diff
diff --git a/readme2.txt b/readme2.txt
index 33d1e15..e361691 100644
--- a/readme2.txt
+++ b/readme2.txt
@@ -1 +1 @@
-trial2
+line2 of 2
git status
# On branch master
# 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:   readme2.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

You prefer meld, you can use

```
git difftool -t meld
```

Compare the working version of a file with the one in the last commit

```
git diff HEAD
```

Inspecting the history of the project

```
git log
```

For a graphical view of the history of the project:

```
gitk
```

Branches

One interest of git is that it is possible to create several branches to make independent developments and merge them later.

To create a new branch:

```
git checkout -b [new_branch_name]
```

To list all branches:

```
git branch -a
```

To switch to an existing branch:

```
git checkout [branch_name]
```

To compare two branches

```
git difftool -d branch1..branch2
```

To compare a specific file:

```
git difftool branch1..branch2 -- filename
```

To merge a branch to the master branch:

```
git checkout master
git merge [branch_name]
```

Working with remotes

To add a remote repository

```
git remote add -f nameforremote path/to/repo_b.git
git remote update
```

To list the remotes

```
git remote -v
```

To compare the current branch with one in a remote

```
git diff master remotes/b/master
```

To see branches on remotes

```
git branch -r
```

(To see local branches: `git branch -l`, all branches, `git branch -a`)

Downloading the most recent changes from the distant repository

If you imported your repository from the internet with `git clone`, you can import the recent changes with:

```
git fetch
git merge
```

Comparing the local working direcoty with a remote

If you want to compare the current working directory with the distant remote origin/master.

```
git fetch origin master
git diff --summary FETCH_HEAD
git diff --stat FETCH_HEAD
```

Pushing your changes to the distant repository

You can send your modified repository (after commiting) to the original remote internet repository:

```
git push
```

Handling very large files (e.g. data)

git-annex allows you to leave large files in some of the repositories and keep only links in others.

See `https://writequit.org/articles/getting-started-with-git-annex.html` and `https://git-annex.branchable.com/walkthrough/`