TMA4320 Introduction to Scientific Computation

Version Management and Debugging

Ronny Bergmann
Department of Mathematical Sciences, NTNU.

January 12, 2023

NTNU

Norwegian University of Science and Technology

Version Management

Motivation.

Scenario 1. Writing a Thesis.
Goal. Avoid loss of data, keep important versions.
`thesis.tex`, `thesis2.tex`, `thesis-20240106.tex`, `thesis20240108.tex`, `thesis-20240108-1148.tex`,...

👎 Having several versions of a file in a folder get's chaotic fast

Scenario 2. Recovery of previous versions.

- "Yesterday this code still worked"
- "I liked my version of the image from last week better".

Scenario 3. Collaborative work on a paper/code.

- exchanging code
- being sure to discuss the same version
- working on the code / a file at the same time

Version Control.

Goals.

- Track changes of certain (but not all) files in a folder
- avoid race conditions and loss of data
- compare different versions

Approaches / History.

- copying files into subfolders, e.g. `v-2020-10-05/`
  (Source Code Control System, SCCS, 1972; Revision Control System, RCS, 1982)
- store files on a central Server, keep local copy
  (Concurrent Versions System, CVS, 1989; Perforce, 1995; Subversion, SVN, 2000)
- decentralized version control, whole history also local
  (Darcs, 2003; Bazaar, 2005; Mercurial, 2005; Git, 2005)

Git – Philosopy.

Taking a folder under `git` management means

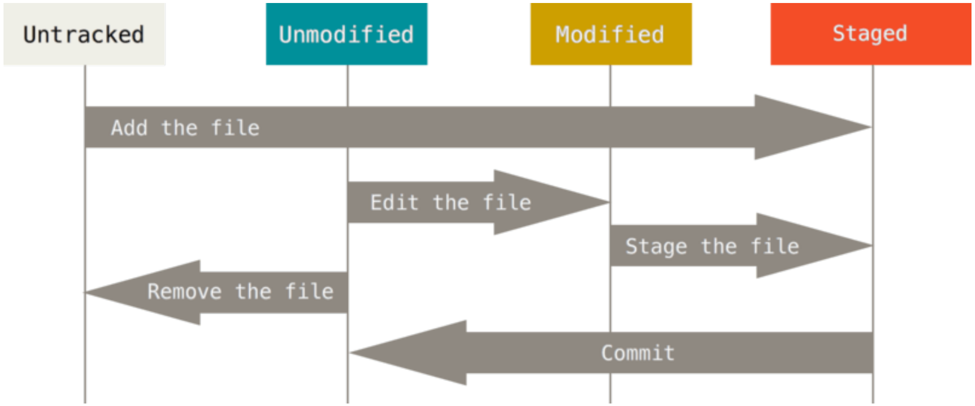
- with every `commit` we store a “snapshot” of the current state in a local database
- the folder together with this set of commits is called the repository.
- each commit can be uniquely identified by a `SHA-1`-Hash

Any new file that is not (yet) under version control is called `untracked`.

Files under version control are called `tracked`. They can be in 3 states

- `commited` – the current file is in the state also stored in the database
- `staged` – the file is marked to be in the next `commit` (snapshot)
- `modified` – the file was changed (compared to the last `commit`)

Files and their status – an Overview



CC-BY-NC Git Pro Book, https://git-scm.com/book/

Files & Version control

The following files are very well-suited for version control

- Text files
- Source files

The following files are ok for version control

- (small) images & illustrations
- sometimes PDFs (but for example not LaTeX-generated ones)

The following files should not be under version control

- automatically generated files (e. g. all LaTeX temporary-files)
- files that (might) change a lot even if one only changes one character somewhere. e. g. Word files, Jupyter files,...
- (large) data files.

Git Configuration

Git keeps 3 configurations (here as the setup on command line)

- system `git config --system`
- user `git config --global` stored in `~/.gitconfig` for all your repositories
- per repository: `git config` im stored in `.git/config` of that repo

The two user ones that are important are

```
git config --global user.name "Ronny Bergmann"
git config --global user.email "git@ronnybergmann.net"
```

But it is easier to set them in our GUI. We will use GitHub Desktop here.

These should be set before starting any repository.
But you also just have to do that once on a computer.

Initialization: `git init` and `git clone`

Scenarrio 1. Start a local project.
Either on terminal in the folder we want to start a repository

```
git init
```

or using our GUI.
$\Rightarrow$ A local folder with an empty repository, i. e. all files are `untracked`.

Scenarrio 2. Start working on a project someone already began.
These are either HTTPS or SSH addresses and we would use either of

```
git clone https://github.com/JuliaLang/julia.git
```

```
git clone git@github.com:JuliaLang/julia.git myFolder
```

$\Rightarrow$ we obtain a folder (here `julia/` or `myFolder` if specified)
containing the current project status (and all history).

The current status: `git status`

We again first look at the state on terminal. With

```
git status -s
```

we can see what status files are in. A new `Readme.md` is in status

```
?? Readme.md
```

The two `??` Indicate the status. We can `stage` (mark for next commit) with

```
git add Readme.md
```

Then the status reads (since it will be added)

```
A Readme.md
```

Changing the Readme again we get

```
AM Readme.md
```

Meaning: The file is marked to be Added into the next commit but was Modified since then. Repeat addition to get back to just `A`

More on `git add`

We can also remove files with `git add`:
Delete the file `deletethis.md` and use

```
git add deletethis.md
```

to tell git, that this change should be in the next commit

Shortcuts

- `git add -u` add all modified and deleted files (in this repository)
- `git add -A` add all files (in this repository, also untracked ones!)
- `git add *.md` add all (tracked and untracked) files ending in `.md` in the current folder

Usually this is again easier in the GUI.

Instead of deleting a file and using add, you can also use `git rm`.

Creating a snapshot: `git commit`

Having marked all files we want to include in a `commit`, we write

```
git commit -m "Description"
```

where the description string should describe what we changed.

The `Description` should

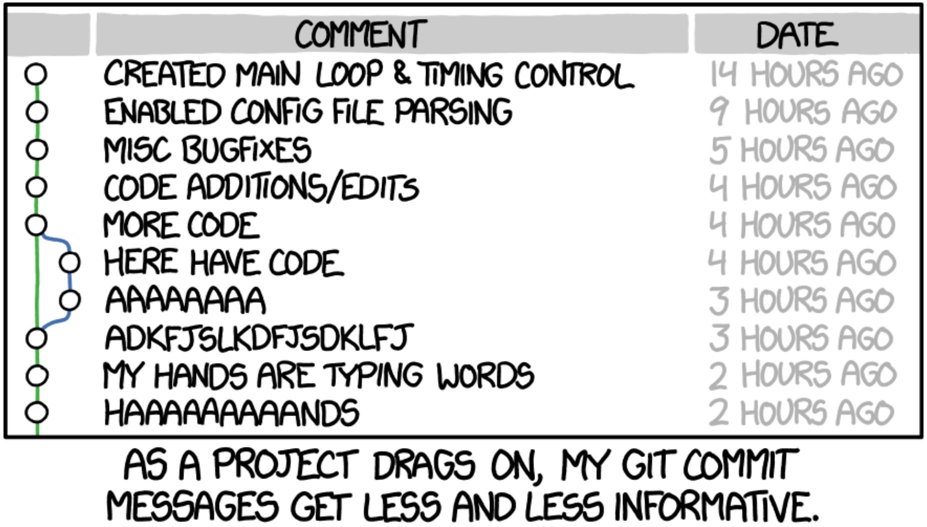
- be shorter than 72 characters
- be an active description of what the commit introduces/does, e. g.
  - `"Initliazizes the repository"`
  - `"Fixes the bug from Issue #1337"`
  - `"Finishes the text for Chapter 3"`

Longer commits with more than one line might often help.

```
git commit
```

opens an editor to enter the description. But this is easier in the GUI.

How not to write commit messages



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Ignore files

To not track `.log` files, we can avoid adding them ever:

In the `.gitignore` file we add the line

```
*.log
```

to ignore all log files.

$\Rightarrow$ these files will never be part of a commit and vanish from `git status`.

Of course the `.gitignore` itself should be version control.

Taking a detour: `git branch`

A `commit` is a snapshot of changes files and includes

- all changes we made
- Metadata: timestamp, author (name & e-mail),
- the description
- a previous commit (parent) it is built upon

so they build a “line” or “branch” of commits.

We want to build a new feature without “disturbing” colleagues / the main part. With

1. `git branch newFeature` we generate a separate branch starting at the current commit we are on
2. `git checkout newFeature` we switch to that branch

! before switching we have to be clean:
no modified or staged files allowed!

Back to just one branch: `git merge`

Goal. We want to “join” the two branches after working on `NewFeature`.
But. In the meantime: new commits on the default branch `main`.

We have to

1. switch to the branch, we want to merge into:

   ```
   git checkout main
   ```

2. Tell git to merge the other one into the current one that is combining all changes from both. We type

   ```
   git merge newFeature
   ```

Then one of the three following things might happen:

- “fast forward” – directly possible
- “recursively” – directly possible by a larger comparison
- “merge conflict” – not directly possible! More on that soon.

Merge Conflicts

A merge conflicts appears if the same line in one file was changed on both branches (since we branched).

Then the merge conflict replaces this by a marker to resolve this.

Example.

```
<<<<<<< HEAD:File.txt
The line how it was changed to on the main branch
=======
The line how it was changed to on the NewFeature branch
>>>>>>> NewFeature:File.txt
```

$\Rightarrow$ This has to be manually resolved replacing these 5 lines e. g.

```
The line how it should be resolved to
```

and `git add` this change.

After all conflicts have been resolved, finish by writing a `git commit`.

Cherry Pick & Rebase

Cherry Pick

Scenarrio. We have one commit on the `NewFeature` branch that we want to apply to another branch as well.

1. We look up its short Hash (`a722703` for example)
2. we switch to the branch where we want to apply the commit
3. we use `git cherry-pick a722703`

Rebase (let’s not do that)

Scenarrio. “Move” a set of commits onto another branch.

To take all commits from the `NewFeature` branch and put them on top of the changes on `main` we use (while on `NewFeature`)

```
git rebase main
```

Attention! This changes the first commits parent!
and rewriting/modifying commits should usually not be done.

Remotes I: Add remotes

We can add a server to out repository with

```
git remote add ghs git@github.com:JuliaLang/julia.git
```

$\Rightarrow$ our repository now has a sever it knows called `ghs`

When cloning a repository (see slide 8) that url is added `origin`.

We can list all remotes with `git remote -v`.

Keep in mind! We now have the `main` branch at least twice:

1. on our computer `main`
2. on that (remote) server, called `ghs:main` (or `origin:main`)

! in order to "work": both have to be "the same repository":

- have the same very first commit
- best: remote was pushed from us or we cloned from there

Remotes II: Pull

When we are on the branch `main`:
To get “new” commits from a server, we use

```
git pull
```

It performs

1. ask the server for updates and get its `main` locally as `origin:main`
2. perform `git merge origin:main`
   to merge new commits from the server into our our `main`.

! this might yield a merge conflict we then have to resolve.

Technical Detail. The long form here is `git pull origin main`.

More general: `git pull ghs TheirFeature` to get and merge `ghs:TheirFeature` into the branch we are currently on.

Remotes III: Push

To transfer our last changes to the server:

`git push` (again: short for `git push origin main`)

! If the server has commits “different” from us, the push is denied, since a potential merge conflict can not be resolved by the server!

Good workflow (if you do not just work alone)

1. Before you start working on something: `git pull`
2. Regular commits (`git add` & `git commit`) after every logical section (work package)
3. When you are finished working:
   - short: `git push`
   - better: first `git pull` only then `git push` (Why?)

3a good pratice: variant of 3 only push code that works.

2a even stronger: vary 2 to only commit code that works.

Summary

- Decentralized version control with `git` (after `git init`)
  - `git add`
  - `git commit (-m)`
- Especially in `branches` and via `merges`
  - `git branch`
  - `git checkout`
  - `git merge`
  - `git cherrypick` (& `git rebase`)
- Collaboration / Synchronization (multiple) `git` server
  - `git remote add`
  - `git pull`
  - `git push`
- Further topics a git server (like GitLab or GitHub) provides
  - Issues – discuss features / bugs
  - Merge-Requests (Pull-Requests)
  - Continuous Integration

NTNU

Norwegian University of Science and Technology

Debugging

Motivation

Scenarrio 1.
You wrote a function, say a novel method to compute the square root.
Then you optimize the memory usage.

It got much faster but suddenly always returns `5`.
This is the right answer if you plug in `25`, but of course wrong for all other input.

Scenarrio 2.
You wrote an iterative algorithm to solve large linear systems. The algorithms runs fine for the first 300 steps, but then suddenly throuws an error, that division by zero is not possible.

Goal. Discuss how to approach these Scenarrios in a structured way, and if time permits, discuss how to best avoid these in the first place.

Main types of errors

- Syntax errors (syntax = system of rules)
  - error message usually points to the line and character
  - error message usually provides enough information to fix the bug

```
1 = x
y = 3+ ;
```

- Semantic errors (the meaning is not correct)
  - running syntactically correct code with invalid instructions
  - appear during runtime

```
x = [1,2,3]; x[3] #classic: off-by-one-error
y = 5 - "a"
```

- Logic errors
  - appear when some code does not meet the specification / intention
  - best case: they cause a (semantic) run time error
  - worst case: they can only be found by validation.

```
def my_sum(a,b): #name indicates, it _should_ summ a and b
    return a - b
my_sum(1,2) # yields -1 not 3
```

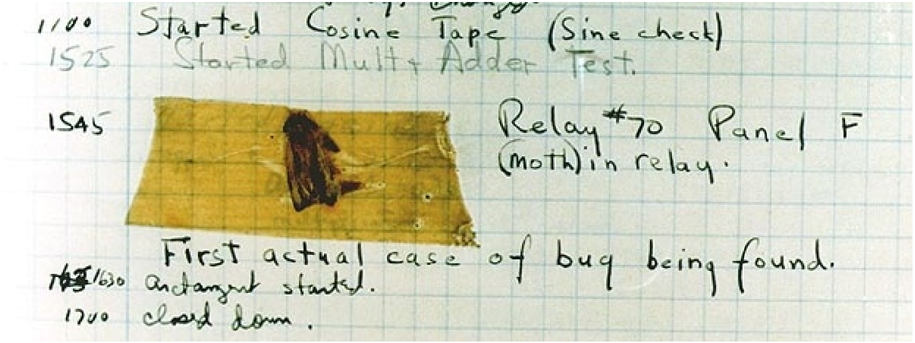
You ran into an error in your code. What would you do?

Assume you have a program that produces results that are either wrong results or there are even error messages.
In summary you have unwanted/unexpected behaviour.

Which approaches do you know?

Debugging

The terms bug and debugging are popularly attributed to Admiral Grace Hopper in the 1940s.[1] *While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.*



1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

[1] https://en.wikipedia.org/wiki/Debugging#cite_note-1

Aproaching Debugging[2]

1. Make the error reproducible.

⇒ Create a Minimum (non-)Working example (MnWE)

2. Locate the source of the error:
   - collect data that produces the defect
   - formulate a hypotheses about the error
   - plan how to proof or disproof the hypothesis, either by tests or examining the code
   - proof or disproof the hypothesis

! repeat step 2 this until the source is found

3. Fix the defect
4. Test the fix
5. Look for similar errors

[2] McConnell , Code Complete, 2nd edition, 2004, Microsoft Press, Chapter 23.

Further on locating the error

When locating the error

- take notes on the approaches you tested
- for mathematical problems: verify small examples by hand
- print interims results in the code `println()`
- test / verify that the input of a function is correct
- use a unified coding style for readability

Outlook I: Debugging Tools

Sometimes the scenario is hard to investigate with pure printing of interims results.
A debugger is a tool that gives you an overview over all variables status while you go step-by-step through your program.
Initially you set a breakpoint which (when reached) triggers the debugger to start.

Examples.

- Python has PDB `https://docs.python.org/3/library/pdb.html` when running python scripts (from terminal)
- JupyterLab has a debugger `https://jupyterlab.readthedocs.io/en/stable/user/debugger.html`

Discussion: How can we reduce the need for debugging?

Outlook II: Avoiding Bugs with Unit Tests

On larger projects $\rightarrow$ write Unit Tests.

These are small tests, that test single components or functionality of a larger software.

Combined with git these can for example be run every time you push.
$\Rightarrow$ this one aspect of Continuous Integration (CI)
These automated runs assure that the code at least works to the extend you cover with your tests.

Quotes on debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
Brian Kernighan

Program testing can be used to show the presence of bugs, but never to show their absence!
Edsger Dijkstra