Software Engineering for Mathematicians

Lecture Overview

In this lecture we step back from writing individual programs and look at how larger projects are developed and maintained.

We will discuss tools and ideas that help you:

- manage growing code bases,
- collaborate effectively with others,
- and ensure that your computational results remain reproducible.

Many of these ideas originate in industry, but they are equally valuable in mathematical research.

Why Should Mathematicians Care?

In modern mathematics, code is no longer just a convenience but often an essential part of the research process.

Numerical experiments, symbolic computations, and simulations all produce results that must be:

- reproducible,
- understandable,
- and extendable by others.

Without proper structure, even your own code becomes difficult to reuse after a few months.

Typical Problems

Many researchers encounter the same issues when working without proper tools.

Files accumulate with names such as:

- `final.py`
- `final_v2.py`
- `final_really_final.py`

At the same time:

- old results cannot be reproduced,
- changes are not tracked,
- and collaboration becomes error-prone.

What is Version Control?

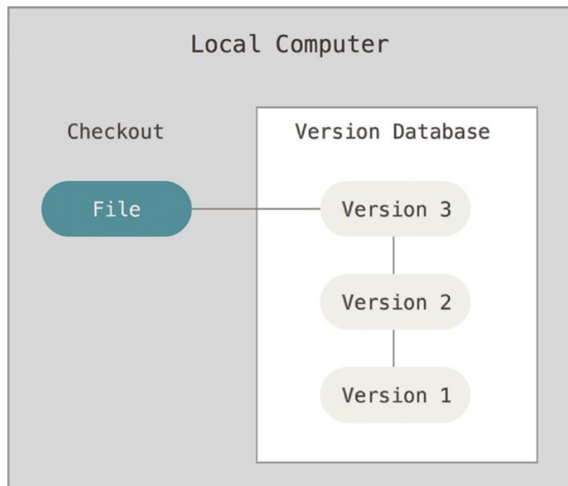
Version control systems record the history of your project over time.

Instead of manually copying files, you obtain:

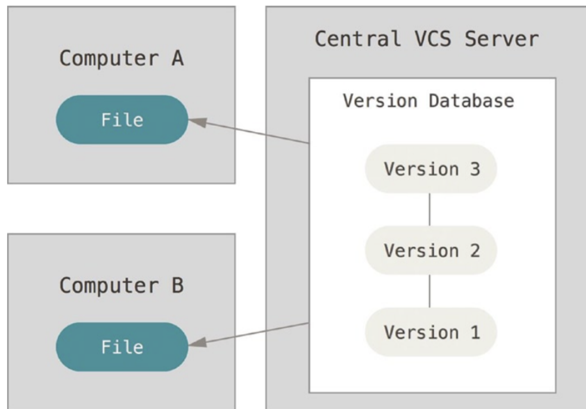
- a complete history of all changes,
- the ability to revert to earlier states,
- and a precise record of who changed what.

This is essential both for collaboration and for reproducible research.

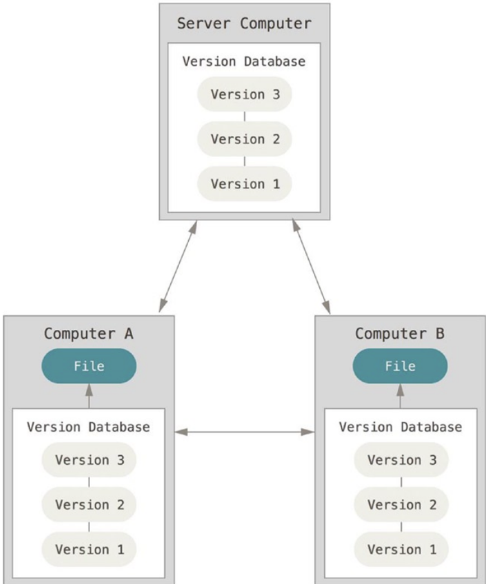
Local version control



Centralised version control



Distributed version control

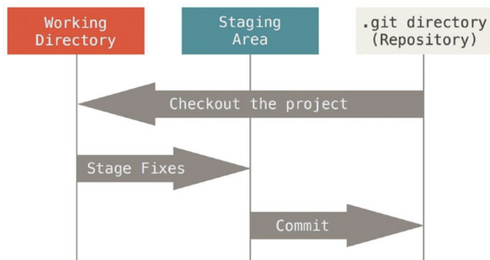


Basic Idea of Git

Git is a distributed version control system.

Your work progresses in three conceptual stages:

- the working directory, where you edit files,
- the staging area, where you prepare changes,
- and the repository, which stores committed history.



This structure allows you to carefully control what becomes part of the permanent record.

Creating a Repository

(I suppose below that GIT is installed.)

We begin by turning a directory into a Git repository.

```
git init
git status
```

The `status` command is especially important: it tells you which files are tracked and what has changed.

Recording Changes

To record changes, we first add files to the staging area and then commit them.

```
git add file.py
git commit -m "Initial version of algorithm"
```

A commit should represent a meaningful step in your work, similar to a logical step in a proof.

Inspecting History

Git provides tools to inspect the evolution of your project.

```
git log
git diff
```

This allows you to understand how a result was obtained and to identify when changes were introduced.

Branching

Branches allow you to explore alternative ideas without affecting the main line of development.

```
git switch -c new-idea
```

This is particularly useful in research, where multiple approaches are often tested in parallel.

Merging

Once a branch is complete, it can be merged back.

```
git switch main
git merge new-idea
```

If two branches modify the same part of a file, Git will require you to resolve the conflict manually (see later).

Making a New Commit and Rolling Back

After your initial commit, you will typically continue modifying your code and creating further commits.

Example: making a second commit

```
# modify file.py

git commit -m "Improve algorithm"
```

Now suppose this change introduced an error and you want to go back.

Option 1: Safe rollback (preserve history)

```
git revert HEAD
```

This creates a new commit that undoes the previous one.

Option 2: Reset (rewrite history)

```
git reset --hard HEAD~1
```

This removes the last commit entirely.

Collaborative Workflows

Modern platforms such as GitHub or GitLab provide infrastructure for collaboration.

Typical workflows include:

- proposing changes via pull requests,
- reviewing code before merging,
- and tracking tasks through issue systems.

These practices improve both correctness and clarity.

Cloning an Existing Repository

In practice, you will often start from an existing project rather than creating one from scratch.

Cloning creates a complete local copy of a remote repository, including its history.

```
git clone https://github.com/user/project.git
cd project
```

After cloning, you can immediately inspect the code, run it, or start contributing.

Connecting to a Remote Repository

A local repository can be connected to a remote server.

This allows you to synchronize your work with others.

```
git remote add origin https://github.com/user/
    project.git
git remote -v
```

The name `origin` is conventional for the main remote.

Fetching Changes

Fetching downloads changes from the remote repository but does not modify your working files.

```
git fetch
```

This allows you to inspect incoming changes before integrating them into your work.

Pulling Changes

Pulling incorporates remote changes into your current branch.

```
git pull
```

Conceptually, this combines:

- fetching updates, and
- merging them into your current branch.

This is the standard way to stay up to date with collaborators.

Pushing Changes

After committing locally, you can share your work with others by pushing it to the remote repository.

```
git push origin main
```

This updates the remote branch and makes your changes visible to collaborators.

Viewing the Current State

It is important to always know the state of your repository.

```
git status
```

This command tells you:

- which files have been modified,
- which changes are staged,
- and which files are untracked.

Unstaged vs Staged Changes

Git distinguishes between unstaged and staged changes.

To see unstaged changes:

```
git diff
```

To see staged changes:

```
git diff --staged
```

This allows you to carefully review what will be included in the next commit.

Unstaging Changes

If you accidentally staged a file, you can remove it from the staging area.

```
git restore --staged file.py
```

The file remains modified, but it will no longer be included in the next commit.

Discarding Local Changes

If you want to discard changes in a file and return to the last committed version:

```
git restore file.py
```

This should be used with care, as the changes will be permanently lost.

What is a Merge Conflict?

A merge conflict occurs when Git cannot automatically combine changes from two different sources.

This typically happens when:

- the same part of a file was modified in two branches, or
- one branch modifies a file that another branch deletes.

In such cases, Git requires human intervention to decide the correct result.

When Do Conflicts Arise?

Conflicts most commonly occur in the following situations:

- During a `git merge`
- During a `git pull` (which performs a merge)
- During a `git rebase`

They are especially likely when multiple collaborators work on the same file simultaneously.

How Git Detects Conflicts

Git does not understand the meaning of your code. Instead, it compares files as sequences of lines.

Internally, Git:

- computes differences ("diffs") between versions,
- identifies which lines were added, removed, or modified,
- and attempts to combine these changes automatically.

If two changes affect overlapping regions of the file, Git cannot decide which version is correct.

Three-Way Merge Concept

Git typically performs a *three-way merge*.

It compares:

- the common ancestor (base version),
- your current branch (local),
- and the incoming branch (remote or other branch).

If both branches changed the same lines relative to the base, a conflict is detected.

Example Conflict in a File

When a conflict occurs, Git marks it directly in the file:

```
<<<<<<< HEAD
def f(x):
    return x + 1
=======
def f(x):
    return x + 2
>>>>>>> new-branch
```

You must edit the file to produce the desired final version.

Understanding Conflict Markers

The markers indicate competing versions:

- `HEAD`: your current branch
- the separator =======
- the incoming changes from another branch

Your task is to remove the markers and keep the correct code.

Resolving a Conflict

To resolve a conflict:

1. Open the affected file
2. Edit the code to the desired final version
3. Remove all conflict markers
4. Stage the resolved file

```
git add file.py
git commit
```

Resolution Strategies

There is no single correct strategy; it depends on context.

Common approaches include:

- choosing one version entirely,
- manually combining both versions,
- rewriting the affected code more cleanly.

Careful understanding is essential—this is not a mechanical process.

Avoiding Conflicts

While conflicts are unavoidable, their frequency can be reduced.

- Commit small, focused changes
- Pull frequently to stay up to date
- Avoid long-lived branches
- Communicate with collaborators about shared files

Modularity and Structure

Large programs should be decomposed into smaller components.

Each function or module should have a clear purpose and a well-defined interface.

This makes code easier to:

- understand,
- test,
- and reuse.

Example

```
def compute_norm(v):
    return sum(x*x for x in v)**0.5

def normalize(v):
    n = compute_norm(v)
    return [x/n for x in v]
```

Separating these functions avoids duplication and clarifies intent.

Why Testing Matters

Errors in scientific code can silently invalidate results.

Testing provides a systematic way to check correctness and to ensure that future changes do not break existing functionality.

Example Test

```
def test_compute_norm():
    assert compute_norm([3,4]) == 5
```

Even simple tests can prevent subtle mistakes.

Documentation

Good documentation ensures that code remains usable over time.

It should explain:

- what a function does,
- what assumptions it makes,
- and how it should be used.

Reproducible Environments

A common issue is that code works only on one machine.

This is usually due to differing versions of libraries or dependencies.

Virtual Environments

```
python -m venv env
source env/bin/activate
pip install numpy
```

This isolates dependencies and makes experiments reproducible.

Scientific Reproducibility

To ensure reproducibility, one should:

- fix dependency versions,
- store data together with code,
- and record parameters used in experiments.

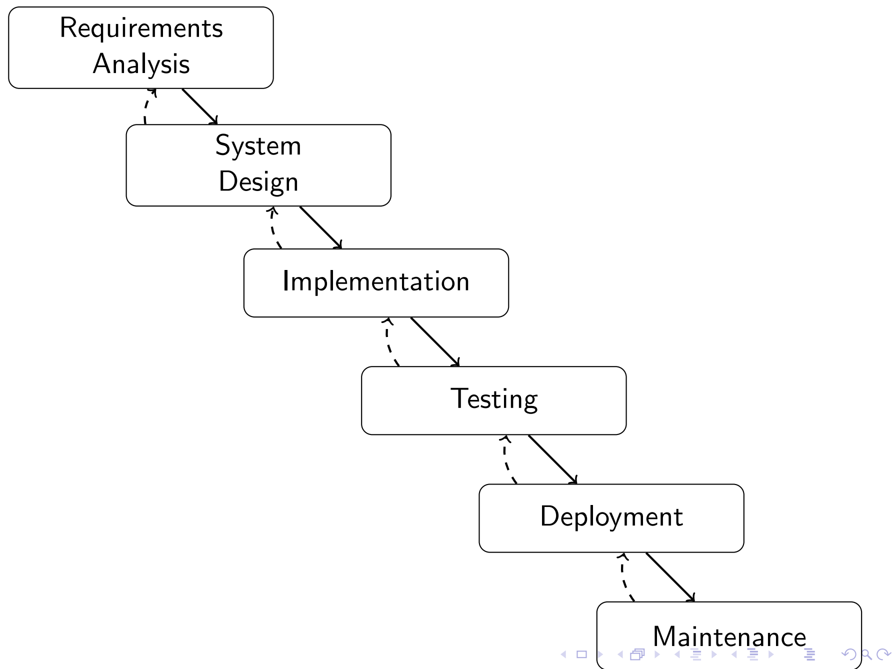
Development Approaches

Different strategies exist for organizing development.

The waterfall model assumes a fixed plan, while agile approaches emphasize iteration and adaptation.

In research, problems evolve, so flexible approaches are usually more appropriate.

Waterfall Model



Agile software development

A manifesto adopted by all sorts of software development gurus in 2001:

- individuals and collaboration instead of processes and tools
- working software trumps all kinds of documentation
- the software is the main thing, so that it works
- collaboration with the customer instead of a contractual agreement
- it is easier to agree with someone than to write a contract
- instead of following some kind of plan, we constantly react to needs
- instead of having rigid requirements and plans in advance, it will turn out somehow

Scrum: Core Components of the Workflow

Scrum organizes development into short, iterative cycles in which a small, usable increment of the project is produced and evaluated.

The process revolves around a few key components:

- Product Backlog: A prioritized list of tasks, ideas, and features. It evolves continuously as the project develops.
- Sprint Planning: Selection of a subset of backlog items to complete in the next iteration (the sprint).
- Sprint: A fixed-length iteration (typically 1–4 weeks) during which development and testing take place.
- Daily Stand-up: Short coordination updates to track progress and identify obstacles early.
- Sprint Review: Presentation and evaluation of the completed work.
- Sprint Retrospective: Reflection on the process itself and identification of improvements for the next sprint.

Scrum is particularly useful in research contexts where requirements are not fixed in advance and evolve as understanding improves.

Final Thought

Well-structured software resembles a well-written proof.

It is:

- logically organized,
- clearly communicated,
- and verifiable by others.

These principles are not specific to programming—they are part of good scientific practice.