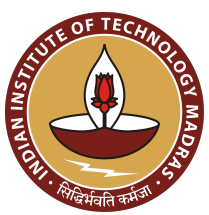
Version Control and Collaborative Project Management

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

October 19, 2026



Why do we need version control?

Let's say a developer has the following files:

- `solver.py`
- `solver_final.py`
- `solver_final_updated.py`
- `solver_final_revised2.py`
- etc. You get the idea !!!

Few questions we should ask

- Which version contains the correct algorithm?
- Can we identify exactly what changed between Tuesday and Thursday?
- How do we recover a working version after introducing a bug?
- What happens when two someone else edits the same function?

Why do we need version control?

- Saving multiple copies may seem convenient, but it quickly becomes confusing.
- We need a systematic way to track changes, understand their history, and recover earlier work.

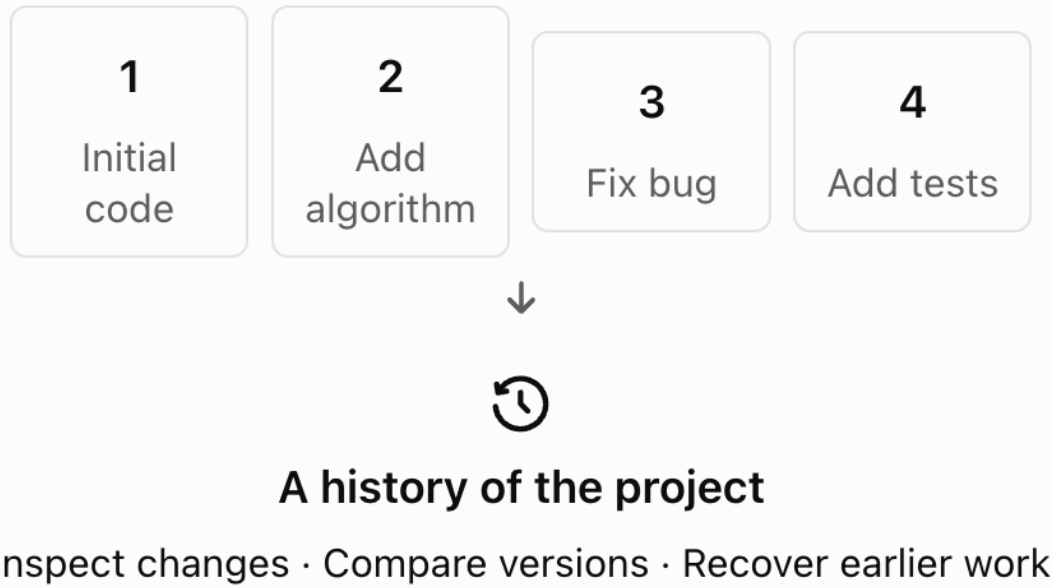
We need version control, not a collection of filenames.

What Is Version Control?

- Keeps track of changes to text files line-by-line over time;
- tracks change between any two versions;
- enables to recover or revert to an earlier version.
- back up and distribute copies of files;
- collaborate on projects.

Version control records the evolution of a project.

Instead of relying on separate files, we record meaningful snapshots of our work.



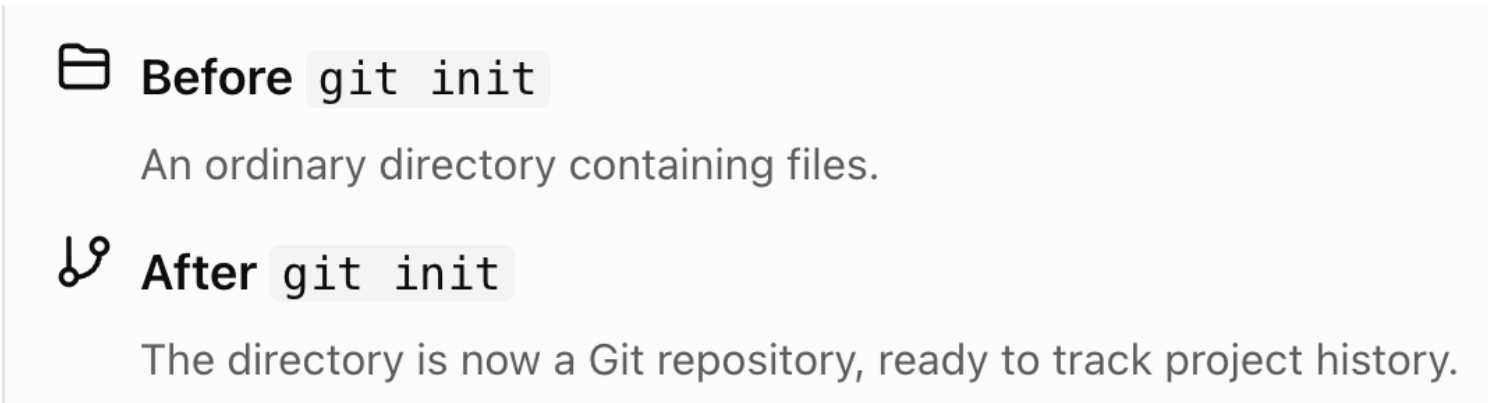
Creating Our First Git Repository

We will develop a small Python library for numerical methods.

```
mkdir numerical-toolbox
cd numerical-toolbox
git init
```

What do these commands do?

- `mkdir`: creates the project directory.
- `cd`: moves into the project directory.
- `git init`: initializes a new Git repository in the directory.



Creating Our First Git Repository

What has happened yet?

- Git has created the repository's internal metadata, traditionally stored in a hidden `.git` directory.
- Our Python files have not yet been recorded in the history well we haven't even created them yet !!!.

Initializing a repository does not automatically track (commit) every file. Run `git status`

First file tracking

- create `solver.py`
- Run `git status`
- ```
  Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
  solver.py
  ```
  ```

Creating Our First Git Repository

Untracked : A newly created, untracked file
What does "untracked" mean?

- The file exists in the working directory.
- Git has not yet been instructed to include it in version control.
- It does not appear in the repository's commit history.

If we delete solver.py now, can Git recover it from a previous commit?
No. The file has not been committed, so Git has no committed version of it to recover.

Committing (adding to version control) our first file

```bash
git add solver.py
git status
git commit -m "Add initial solver structure"
git log --oneline
```

- "`git add`" new or modified files to the index.
- "`git status`" tells us the state of our repository:
  - which files are untracked and not indexed;
  - which files are modified with respect to the index / staging area;
  - which files are staged (indexed) and are ready to be committed;
- "`git add`" selects changes for the next commit; it does not itself create a commit.
- The staging area lets us choose which changes belong together.

Committing (adding to version control) our first file

- "`git commit`" records the staged snapshot in the repository's history.
- "`git log --oneline`" displays the recorded commits in a compact format.

When do we `commit`?

- `commit` is not like `ctrl + s`
- A `commit` is a meaningful checkpoint in the development of the project.

`-m` stands for message. It lets us provide the commit message directly in the command.

- `git commit`: records the staged changes.
- `-m`: specifies the commit message.
- "`Add initial solver structure`": describes what the commit does.

Committing (adding to version control) our first file



What Has Changed? Introducing `git diff`

- Modify solver.py by replacing the placeholder with a simple implementation
- Now run:
  - `git status`
  - `git diff`
- it shows
  - + Lines added or modified
  - - Lines removed or replaced

What does `git diff` show?

It compares the current unstaged changes with the last committed version.

Have we recorded this implementation in Git history yet?

- Run `git diff` before staging. Git lets us inspect exactly what we are about to record.
- Then run `git add solver.py` followed by `git diff --staged` to

Understanding the Project History

After staging and committing the implementation, run:

```
git add solver.py
git commit -m "Implemented bisection method"
git log --oneline
```

The history now contains two commits:

- Commit 2 - Implemented bisection method
  Recorded the addition of the algorithm
- Commit 1 - Add initial solver structure
  Recorded initial project structure

To inspect a particular commit:

```
git show HEAD
```

`git log` displays the sequence of commits

A repository is not just the current version of a file. It also contains a history of committed snapshots. That history allows collaborators to understand how the project evolved.

Branches: Dev Without Disturbing the Stable Version

- Suppose bisection is working, and we now want to implement Newton's method.
- We want to develop the new feature without directly changing our stable main branch.

```
git switch -c feature/newton-method
```

This creates a new branch and switches to it.




```
git branch
git switch master
git switch feature/newton-method
```

Branches: Dev Without Disturbing the Stable Version

- A branch is a separate line of development, not a separate copy of the entire project that must be managed manually.
- Both branches initially point to the same latest commit.
- New commits on the feature branch can advance independently of main.
- Switching branches changes which version of the project you are working on.

If we discover a bug in bisection while developing Newton's method, why is working on a separate branch useful?

It isolates the new feature from the stable version and makes parallel development easier to manage.

Commit Work on a Feature Branch

- We are currently working on `feature/newton-method`.
- Add Newton's method to `solver.py`.

```
git status
git diff
git add solver.py
git commit -m "Add Newton method"
git log --oneline --all --graph
```

What do we observe?

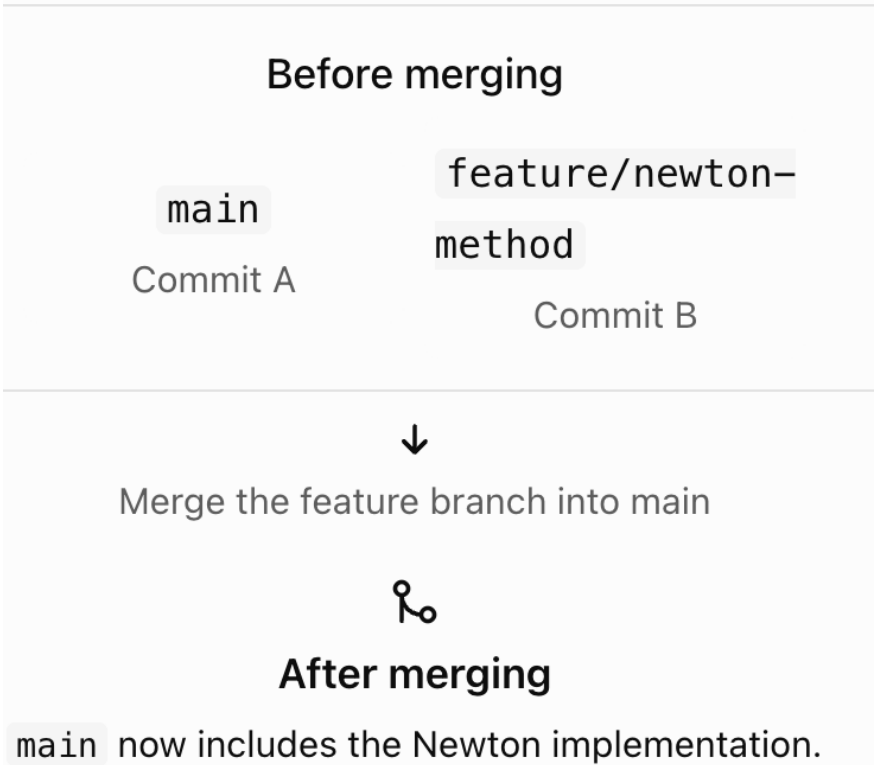
- The new implementation is committed on the feature branch.
- The `master` branch still points to its earlier commit.
- Git records the new work without requiring us to modify `master`.

- A branch lets us develop and commit a feature independently.
- The commit is part of the repository's history, but it is not automatically incorporated into `main`

Merging a Feature into main

- The Newton implementation is ready.
- We now want to integrate it into the stable project.

```
git switch master
git merge feature/newton-method
```



Merging a Feature into main

- A branch isolates development.
- A commit records a change.
- A merge integrates changes from one branch into another.

- Because main has not changed since the feature branch was created, Git can usually integrate this particular change automatically.
- Git may perform a fast-forward merge, moving the main pointer forward without creating a separate merge commit.

Does merging mean that Git has verified Newton's method is mathematically correct?

No. Git integrates changes; correctness still requires tests and review.

What Happens When Changes Conflict?

- Now imagine two developers independently modify the same line in `solver.py`.
- Developer A changes the function to:

```
def f(x):
    return x**2 - 2
```

- Developer B changes it to:

```
def f(x):
    return x**2 - 3
```

- Both changes may be valid for different problems, but they cannot both define the same function in the same way.
- When Git cannot automatically reconcile the changes, a merge conflict occurs.

What Happens When Changes Conflict?

Let's assume your branches are:

- `master:  contains return x**2 - 2`
- `feature/another-change:  contains return x**2 - 3`

Switch to master

```
git switch master
```

Check the current version of the file:

```
cat solver.py
```

- we see the version committed on `master`
- Now Merge the other branch

```
git merge feature/another-change
```

- Because both branches changed the same line differently, Git should report a conflict

What Happens When Changes Conflict?

Inspect the conflict

```
git status
```

When we open `solver.py` in our editor. We should see something like:

```
def f(x):
<<<<<<< HEAD
    return x**2 - 2
=======
    return x**3 - 3
>>>>>>> feature/another-change
```

Here:

- `HEAD` refers to the version in your current branch, main.
- The section below `=======` is the incoming version from `feature/another-change`.
- Git is asking you to decide which version should remain.

What Happens When Changes Conflict?

Resolve the conflict

- For this demonstration, suppose the intended problem is $f(x) = x^2 - 2$.
- Edit the file to remove the conflict markers and retain the correct implementation.
- Save the file.

Tell Git that the conflict is resolved

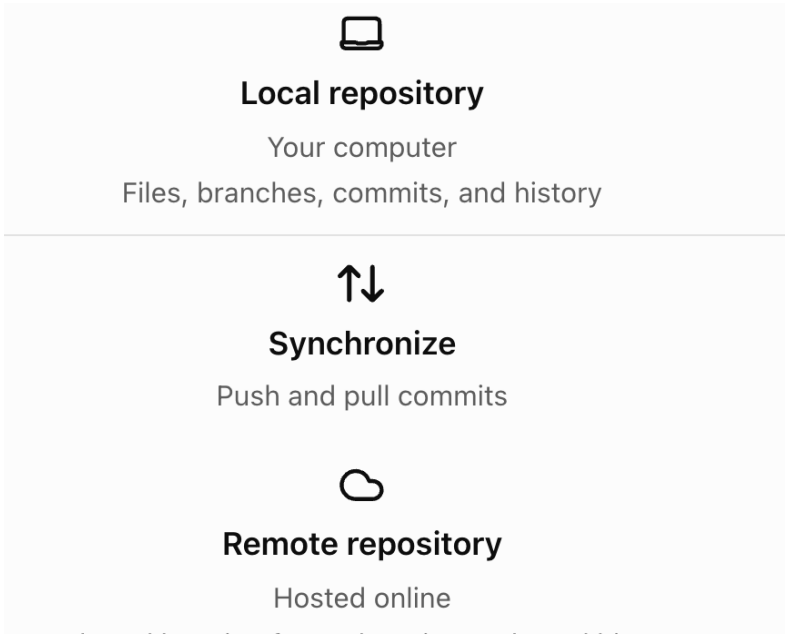
```
git add solver.py
git status
git commit -m "Resolve merge conflict"
```

We can inspect the resulting history:

```
git log --oneline --graph --all
```

From Local Git to Remote Repositories

- So far: Our repository exists on one computer.
- What if another student needs the code?
- What if we want to work from a different computer?
- How do we share our commits?
- A remote repository allows developers to exchange commits. Each developer can still have a complete local repository and work independently.



Welcome to GitHub: Your Online Home for Code

Git manages version history. GitHub lets us share and collaborate on that history.

- Imagine that you have written a Python program on your laptop.
- How can another student access your code, suggest improvements, and contribute to the project without exchanging files over email?
- GitHub provides an online home for your Git repositories.

What can we do with GitHub?

- Store and share code: Make a repository accessible to collaborators.
- Track project development: Inspect commits, branches, and changes.
- Collaborate: Let multiple developers contribute to the same project.
- Review code: Discuss proposed changes before integrating them.
- Organize work: Assign tasks using issues and manage contributions through pull requests.

Your First Step: Create a GitHub Account

GitHub is not a replacement for Git. It is a platform that builds collaboration tools around Git.

Set up your account

- Step-1: Open GitHub - Visit https://github.com/.
- Step-2: Sign up
- Step-3: Verify your account: Complete the email verification and any other verification steps requested by GitHub.
- Step-4: Sign in

Create Your First Repository on GitHub

Now we will connect the Git repository you created locally to GitHub.

Create the remote repository

On GitHub:

1. Click the + menu in the top-right corner.
2. Select New repository.
3. Enter the repository name: numerical-toolbox.
4. Add a short description, such as A Python project for numerical methods.
5. Choose Public for this classroom demonstration, or Private if you prefer to restrict access.
6. Leave "Add a README file" unchecked.
7. Click Create repository.

Why leave the README unchecked?

Your local repository already has commits. Starting with an empty remote makes the first push straightforward.

Connect your local repository

- GitHub will display instructions for connecting an existing repository.
- Return to the terminal on your computer, inside the numerical-toolbox directory, and run:

```
git remote add origin <repository-url>
git remote -v
git push -u origin main
```

Replace `<repository-url>` with the URL of the repository we just created. Copy it from GitHub.

- `git remote add origin ...` : Associates our local repository with the GitHub repository.
- `git remote -v`: Displays the configured remote URLs.
- `git push -u origin main`: Publishes your local `master` branch and establishes its upstream tracking relationship.

Clone a Repository: Get a Project on Your Computer

- Your teammate has created a GitHub repository containing a numerical solver.
- You want to contribute to the project.
- How do you get the code onto your computer?

You clone the repository.

1. Open the `numerical-toolbox` repository.
2. Click the Code button.
3. Copy the repository URL.

In your terminal, run:

```
git clone <repository-url>
cd numerical-toolbox
```

What does cloning give us?

Cloning creates a local repository containing the project's files and Git history, along with a remote configuration that normally points to the original repository.

We can now:

- inspect existing code.
- Examine previous commits.
- Create branches.
- Modify files and commit changes locally.
- Synchronize with the remote repository.

`git clone`: used to obtain a repository for the first time.
`git pull`: used later to bring changes from a remote repository into an existing local repository.

Two Developers, One Project: Push and Pull

If you clone a repository, do you automatically get permission to push changes to it?

- No. Cloning and writing permission are different.
- You need appropriate access to push directly to the original repository.
- Otherwise, you may contribute through a fork and pull request.

Let's simulate a collaboration between two students.

- Developer A has the project on their laptop.
- Developer B has cloned the same project onto another laptop.

Both have their own local repositories, but they share a common remote repository on GitHub. Developer A: Makes a change

```
git add solver.py
git commit -m "Improve solver documentation"
git push
```

Two Developers, One Project: Push and Pull

- A commit is local until it is pushed.
- A change pushed by someone else does not automatically appear in your local working directory; you must synchronize with the remote.

- Developer B: Obtain the change
- In their own local repository, run:

```
git pull
```

What If Two Developers Work at the Same Time?

- Consider a more realistic situation.
- Both students have cloned the same project and are working independently.
- Developer A modifies the stopping criterion in `solver.py`.
- Developer B modifies the same function to handle a different edge case.
- Both make local commits.
- Developer A pushes first. Developer B then attempts to push.

Git may reject Developer B's push because the remote branch contains commits that are missing from their local branch.

Git avoids silently overwriting the remote branch's newer history.

How should Developer B proceed?

First, inspect the state of the repository and obtain the remote changes:

```
git status
git pull
```

- If the changes can be integrated automatically, Git updates the local branch.
- If the edits conflict, Git asks the developer to resolve the conflict
- After resolving any conflicts and completing the integration, the developer can push the combined history

Why Not Let Everyone Push Directly to `master`?

Imagine four students developing a numerical toolbox.

- Student A implements bisection.
- Student B implements Newton's method.
- Student C writes tests.
- Student D develops numerical comparisons and visualizations.

Suppose everyone edits and pushes directly to `master`.

What could go wrong?

- An unfinished implementation becomes part of the stable project.
- A change breaks another student's code.
- A numerical bug goes unnoticed.
- Two students make incompatible changes.
- Nobody knows whether a contribution has been reviewed.

The solution: Isolate development from integration

- Instead of working directly on main, each student develops a feature on a separate branch.
- `master` Stable, integrated code
- `feature/bisection-tests` — Tests for bisection
- `feature/newton-method` — Newton's method
- `feature/convergence-plots` — Convergence visualization
- Changes are proposed for integration only after they are ready for review.

A shared repository needs a workflow, not just shared access.

The Feature Branch Workflow

Step-1: Start from the latest `master`

```
git switch master
git pull
```

Step-2: Create a feature branch

```
git switch -c feature/newton-method
```

Step-3: Implement and test the feature

```
git switch master
git pull
```

Step-4: Commit the work

```
git add solver.py
git commit -m "Implement Newton method"
```

Step 5: Publish the branch

```
git push -u origin feature/newton-method
```

Pull Requests: Proposing a Change

What has happened?

- The feature is committed locally and its branch is now available on GitHub.
- The stable `master` branch has not been changed by this contribution.
- A branch contains the work.
- A pull request (PR) proposes integrating that work into another branch.

After pushing feature/newton-method to GitHub:

- Open the repository on GitHub.
- Select Pull requests.
- Click New pull request.
- Select main as the base branch.
- Select feature/newton-method as the comparison branch.
- Create the pull request.

Code Review: Is This Implementation Good Enough?

Questions for the reviewer

- Correctness
- Testing: tested on a function with a known root; failure cases been considered?
- Readability and maintainability

a review comment

On the pull request, click the relevant line and add a comment:

> *What happens if the derivative becomes zero or very small? Could we handle this case explicitly and add a maximum iteration limit?*

The author can revise the implementation, commit the changes, and push them to the same branch. The pull request updates automatically.

From Review to Integration

A pull request should not be merged merely because the code has been uploaded.

1. **Developer opens a PR**
   Explains the change and links the relevant task.
2. **Reviewer inspects the code**
   Examines the implementation, tests, and numerical edge cases.
3. **Developer addresses feedback**
   Makes corrections and pushes the revised commits.
4. **Checks pass**
   Automated tests run, where configured, and the team verifies the expected behaviour.
5. **Maintainer approves and merges**
   The accepted changes become part of `main`.