



Advanced Programming for Scientists

Lecture 2

Versioning and Reproducibility with git
From scripts to scientific-grade software

v1.0

Prof. Mateusz Bawaj



Why versioning in science?

Scientific results must be reproducible

Code evolves during:

- Model refinement
- Parameter tuning
- Numerical stabilization
- Performance optimization

We need:

- Traceability of changes
- Collaboration
- Recovery of previous states

Scientific principle:

If you cannot reproduce the exact version of the code, you cannot reproduce the result.



What is git?

- Distributed Version Control System
- Tracks changes in source code
- Stores project history as snapshots
- Enables collaboration

Key property:

Every commit is a complete scientific state



Distributed vs centralized

Centralized (SVN - old model):

- One central server
- Local copies are partial

Distributed (git):

- Every clone contains full history
- Enables:
 - Offline work
 - Experimental branches
 - Robust backups

Scientific implication:

- Your laptop contains the full research history.

git basic workflow

Working directory

- Staging area
- Commit (local repository)
- Push (remote repository)

Commands:

```
$ git init  
$ git add  
$ git commit  
$ git push  
$ git pull
```

First repository exercise

Example: create locally your project

```
$ mkdir your-project
```

```
$ cd your-project
```

```
$ git init
```

Add first file:

```
$ touch simulator.py
```

```
$ git add simulator.py
```

```
$ git commit -m "Initial empty project structure"
```



What is a commit scientifically?

A commit is:

- A documented modification
- A minimal logical unit of change
- A reversible transformation

Bad commit message:

"fix"

Good commit message:

"Fix phase accumulation in round-trip propagation model"

Think:

Could I cite this commit in a paper?

git log as automatic changelog

Provides:

- Author
- Date
- Commit hash
- Message

This is your **automatic laboratory notebook for code**.

Important for:

- Publications
- PhD thesis traceability
- Collaboration in large experiments



Branching as scientific exploration

Scientific workflow is rarely linear.

Example:

- Stable solver
- Experimental optimization
- Alternative physical model

```
$ git branch your-model
```

```
$ git checkout your-model
```

Branches allow:

- Parallel exploration
- Safe experimentation
- Risk isolation



Merge and scientific integration

When your model works:

```
$ git checkout main
```

```
$ git merge your-model
```

Merge represents:

- Integration of validated scientific development
- Consolidation of tested improvements

Discussion:

- Why merging untested code is dangerous in research?

Collaboration model

Remote repository (GitHub, GitLab)

Workflow:

- 1) Clone
- 2) Create branch
- 3) Commit changes
- 4) Push branch
- 5) Open Pull Request
- 6) Code review
- 7) Merge + solve conflicts

Scientific benefit: Peer review at code level



git and publications

Best practices:

- Tag significant version e.g. used for paper:

```
$ git tag -a v1.0-paper -m "Version used in  
Journal submission"
```

```
$ git push --tags
```

- Include commit hash in paper
- Archive release (e.g. Zenodo DOI)

This enables: Full reproducibility



git and publications

Even better practice:

- Publish only tags relevant to the community:

```
$ git push origin tag <tag_name>
```

- This command published all tags

```
$ git push --tags
```

Continuous Integration (CI)

Concept:

- Automated testing on each push

Example checks:

- Unit tests
- Numerical regression tests
- Formatting
- Linting (programmatic and stylistic errors)

Scientific impact:

- Detect numerical drift
- Prevent silent physics changes

Example: numerical regression

Suppose:

- You optimize code with Numba
- Performance improves
- But numerical result changes slightly

CI test:

- Compare output with reference data
- Fail if deviation > tolerance

This protects:

- Physical correctness

Numerical errors

Floating-Point Representation

Computers use finite precision (standard IEEE 754):

- Real numbers are approximated (32, 64 bit long)
- Most decimal numbers are not exactly representable

Example (works in python):

What is the results of: `0.1 + 0.2 == 0.3 ?`

Types:

- Rounding error
- Cancellation error
- Overflow / underflow

(occurs when subtracting two nearly equal numbers, the relevant part of the result may disappear and is prone to rounding error)

Scientific impact:

- Loss of precision in subtraction of close values

Numerical errors

Accumulation of Errors

In iterative algorithms:

- Each step introduces small error
- Errors propagate and amplify

Examples:

- Numerical relativity
- Long-time orbital simulations
- Recursive cavity field updates
- Chaotic systems

Important: Stability \neq Accuracy

Numerical errors

Algorithmic Errors

Wrong method for the problem:

- Using simple integrator
- Using naive summation

Better alternatives (all reduce numerical error):

- Higher-order methods
- Kahan summation

Numerical errors

Discretization and Resolution Limits

Physics is continuous – computers are discrete

Errors arise from:

- Insufficient spatial resolution
- Aliasing (sampling theorem violation)
- Time-step too large

Example:

- Undersampling high-frequency oscillations
- Numerical dispersion in wave equations

Numerical errors

Parallel and Hardware Effects

- Different CPU architectures
- Different compiler optimizations
- GPU vs CPU reductions (check supported data types)
- Order of floating-point operations

Floating-point arithmetic is not associative:

$$(a + b) + c \neq a + (b + c)$$

Scientific implication:

Bitwise reproducibility may fail across systems

```
numpy.allclose(a, b, rtol, atol, ...)
```



Numerical errors

Key Message

Numerical error is not a bug.

It is an intrinsic property of computation.

The scientific programmer must:

- Understand it
- Measure it
- Control it
- Document it

That is part of reproducible computational science.

.gitignore for scientific projects

Never commit:

- Large raw data
- Compiled binaries
- Temporary output
- Virtual environments/dockers

Example .gitignore:

Principle: Repository contains code,
not transient results.

<https://github.com/github/gitignore>

```
# Prerequisites
*.d

# Compiled Object files
*.slo
*.lo
*.o
*.obj

# Precompiled Headers
*.gch
*.pch

# Linker files
*.ilk
```



git and large collaborations

In big experiments:

- Multiple developers
- Long lifetime projects
- Strict review process

git enables:

- Traceability
- Accountability
- Structured integration

Even small student projects should follow similar discipline.



Hands-on exercise

Students (who do not have their project on this course git):

- 1) Create local repository
- 2) Implement:
 - Simple numerical integration (e.g. Euler method)
- 3) Make:
 - At least 3 meaningful commits
- 4) Create branch:
 - Implement improved integrator (e.g. Runge–Kutta)
- 5) Merge after testing



Integration with course workflow

During this course:

- Every assignment is submitted via git
- Issues used to:
 - Report bugs
 - Propose improvements
 - Discuss numerical behaviour

You are not just writing code.

You are developing scientific software collaboratively.

Key takeaways

- git is not optional in modern science
- Commits are scientific states
- Branches are controlled experiments
- CI protects physical correctness
- Tags ensure reproducibility

Next lesson:

Virtualization and containers for reproducible environments.