Computational Reproducibility in Machine Learning: A Hands-On Workshop

Waheed U. Bajwa

Department of Electrical and Computer Engineering
Rutgers University–New Brunswick, NJ USA
www.inspirelab.us

Data Science in Multi-Messenger Astrophysics Program
University of Minnesota
February 25, 2025

CCF-1907658
CNS-2148104

W911NF-21-1-0301

Key Reference for This Workshop

Joseph Shenouda and Waheed U. Bajwa

A Guide to Computational Reproducibility in Signal Processing and Machine Learning

A computational experiment is deemed *reproducible* if the same data and methods are available to replicate quantitative results by any independent researcher, anywhere and at any time, granted that they have the required computing power. Such computational reproducibility is a growing challenge that has been extensively studied among computational researchers as well as within the signal processing and machine learning research community [1], [2].

Introduction

Signal processing research is in particular becoming increasingly reliant on of new libraries and frameworks (such as NumPy [3], Scikit-learn [4], MATLAB Toolboxes [5], and TensorFlow [6]) that provide a layer of abstraction allowing for rapid implementation of complex algorithms. Unfortunately, this changing research landscape is also bringing with it new obstacles and unseen challenges in developing reproducible experiments.

Computational experiments today often incorporate various scripts for preprocessing data, running algorithms, and plotting results, all while utilizing huge datasets that require computing clusters that often take days or weeks to finish computing with multiple manual intercorrect results only when executed on the original machine of the researcher. Furthermore, the nature of these data-driven experiments often requires careful parameter tuning, random number generations, and data preprocessing, all of which are independent from the main finding, such as a new algorithm, being implemented or investigated.

Due to these new challenges, most experiments have become difficult, if not impossible, to be reproduced by an independent researcher. As an anecdote, when attempting to reproduce computational results in our lab from an article published just months prior,

J. Shenouda and W. U. Bajwa, "A guide to computational reproducibility in signal processing and machine learning," *IEEE Signal Processing Magazine*, vol. 40, no. 2, pp. 141–151, Mar. 2023, doi: 10.1109/MSP.2022.3217659.

Outline

Outline

Reproducibility in Science

What is reproducibility?

- The ability to independently verify scientific findings.
- Ensures that the same results can be obtained beyond the original experiment.
- *Not to be confused with confirming scientific conclusions!*

Facets of reproducibility

- Repeatability: Same team, same setup, same results.
- Replicability: Different team, same setup, same results.
- Reproducibility: Different team, different setup, same results.

Reproducibility is about re-running the experiment—not about confirming its broader scientific meaning!

Why Does Reproducibility Matter?

Scientific integrity depends on it!

- Ensures validity and trustworthiness of findings.
- Helps detect errors, biases, or even fraud.
- Encourages innovation by allowing others to build upon previous work.

Famous reproducibility issues

- Psychology's Reproducibility Crisis: Many studies failed to replicate.
- High-Profile Retractions: Some well-known scientific papers had to be withdrawn.
- ML Benchmarks Changing Over Time: Reported improvements often rely on specific dataset splits, preprocessing steps, or software versions, making direct comparison difficult.

Reproducibility is a cornerstone of reliable science!

Computational Sciences and Reproducibility

Computational sciences include:

- Machine learning
- Statistics
- Experimental physics
- Computational neuroscience
- Computational physics
- And more...

Why do these fields need reproducibility?

- Computational results must be verifiable and repeatable.
- Many studies rely on specific software environments and dependencies.
- Ensuring computational reproducibility prevents misleading conclusions.

This workshop focuses on computational reproducibility—the foundation of reliable research in ML and beyond.

What is Computational Reproducibility?

Definition: *A computational experiment is deemed reproducible if the same data, methods, and implementations are available to replicate quantitative results by any independent researcher, anywhere and at any time, granted they have the required computing power.*

Why is this challenging?

- Software and Dependency Issues: Many workflows rely on proprietary or outdated software, making exact reproduction difficult.
- Computing Environment Differences: Hardware variations, OS versions, and library updates can alter results.
- Lack of Standardization: No universal guidelines exist for documenting computational experiments, leading to inconsistent evaluation.

Computational reproducibility remains a major challenge in computational sciences!

The Spectrum of Reproducibility in Computational Sciences

Reproducibility in computational sciences is not binary—it exists on a spectrum

- Computational Reproducibility: Ensuring the same data, code, and methods produce identical results.
- Results Reproducibility: Also called "replicability" in some literature—an independent team reproduces the same results using the same methodology.
- Inferential Reproducibility: Different studies reach qualitatively similar conclusions, even if exact results differ.

Terminology confusion

- Some researchers use "computational reproducibility" interchangeably with "results reproducibility" or "inferential reproducibility."
- Others consider these as distinct concepts, each addressing different aspects of scientific validation.

Computational reproducibility is just the starting point—achieving it lays the foundation for tackling broader reproducibility challenges.

Outline

Computational Reproducibility: More Than Just a Methods Section and Code

"The methods section describes everything needed to reproduce my results." **Not true!** The methods section often leaves out crucial details:

- How was the exact implementation done?
- Which specific library versions were used?
- How were hyperparameters tuned?
- What random seeds were set in different parts?
- What stopping criteria were applied?

"I shared my code on GitHub—this should be enough." **No, it's not!** Code alone is often insufficient without proper packaging, metadata, context, and persistent sharing:

- Is the code well-commented and documented?
- Which scripts were actually used, and which files were executed?
- What was the exact computing environment?
- Does the code compile and run correctly on a different machine?
- What happens when the repository is updated with new versions?
- Will the code remain persistently available?

Reproducibility demands more than just a methods section and code—it requires structured documentation, environment tracking, proper packaging, and persistent sharing.

The Reality of Computational Reproducibility

Computational reproducibility is often harder than expected

- Small changes in software versions, dependencies, or environments can cause results to diverge.
- Without careful documentation, even the original authors may struggle to reproduce their results months later.
- A *Nature* survey (2016) found that **50% of researchers** could not reproduce their own experiments.


News Feature | Published: 25 May 2016

1,500 scientists lift the lid on reproducibility

Monya Baker

Nature **533**, 452–454 (2016) | Cite this article

If you can't reproduce your own results, how can anyone else?

Computational Reproducibility Takes Effort—So Why Bother?

- For the love of science (altruistic reason): Science thrives on transparency, verifiability, and trust, ensuring that discoveries are reliable and reproducible for future research.
- Reproducible papers get cited more (selfish reason): Studies with open, well-documented code and data tend to receive more citations.
- Someone will ask you to reproduce your results (selfish reason): If you can't regenerate your own results later, your credibility is at risk.

Reproducibility isn't just a scientific ideal—it's a practical necessity.

Is Reproducibility Too Much Work?

- Yes, there is an initial effort required.
- But once you learn the tools and best practices, the process becomes much easier.
- This workshop is designed to teach you these techniques!

Investing in reproducibility today saves time and effort in the long run, makes you a good steward of science, and sets your research apart by demonstrating rigor, reliability, and transparency.

Preview of the Hands-On Activities: Four Key Themes

- Managing Experiment Development: When exploring new ideas, your codebase can quickly become disorganized with multiple versions of different files. We will cover strategies for using Git to manage version control, writing clear documentation for future reference, and structuring/refactoring your code to enhance clarity and efficiency.

- Managing Dependencies: Software dependencies can change over time, breaking your code months later or making it impossible for others to run. We will introduce virtual environments using Conda for managing package versions and explore Docker for creating fully reproducible execution environments.

- Ensuring Reproducibility with Random Seeds: Many computational experiments rely on randomness, and failing to control it can lead to results that are impossible to reproduce. We will examine how different programming environments use multiple random number generators, why seeding one does not necessarily seed all, and how parallelization introduces additional challenges.

- Packaging Code for Sharing: Simply uploading your code to GitHub is not enough for reproducibility. We will discuss how to write clear README files with step-by-step instructions, document dependencies using an `environment.yml` file that is exported from the Conda environment, use GitHub Releases for stable versioning, and leverage Zenodo to assign persistent DOIs for long-term accessibility.

This workshop is about practical tools and strategies—let's dive in!

Outline

Managing Experiment Development

Experiments evolve—without structure, chaos follows

- Exploring new ideas leads to multiple versions of the same files.
- Unorganized codebase makes it hard to track progress and reproduce results.
- Proper codebase management saves time, prevents errors, and improves collaboration.

Key strategies covered in this workshop

- Using Git for Version Control: Track and manage changes effectively.
- Writing Clear Documentation: Improve readability and maintainability.
- Refactoring Messy Codebase: Structure your codebase for efficiency.

What is Git and What Does It Do?

Git is a distributed version control system that helps track changes in code

- Originally developed by Linus Torvalds in 2005 for Linux development.
- Enables multiple contributors to work on the same project without conflicts.
- Keeps a history of all changes, making it easy to revert or track modifications.
- Essential for managing evolving codebases in research and software development.

Git is more than just a backup tool—it's a structured way to track and manage changes

- Tracks every change: Keeps a detailed history of modifications.
- Prevents accidental loss: Easily revert to previous versions.
- Facilitates collaboration: Allows multiple contributors without conflicts.
- Creates a structured workflow: Helps organize evolving codebases.

Let's dive into hands-on Git usage and explore its essential operations!

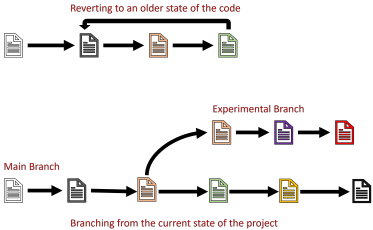
Git Basics: Key Terminology

- Repository (repo): A project folder managed by Git that tracks all changes to files over time.
- Commit: A snapshot of changes made to tracked files, creating a version history.
- Staging: A temporary area where changes are prepared before committing them to the repository.
- Checkout: Moving between different commits or branches in the repository history.
- Branch: A separate line of development, allowing multiple features or fixes to be worked on in parallel.
- Switch: A modern, streamlined command for changing branches, replacing some uses of 'checkout'.
- Merge: Integrating changes from one branch into another to combine work.
- Rebase: Reapplying commits from one branch onto another, creating a cleaner project history.

- Remote: A version of the repository stored on a server (e.g., GitHub) for collaboration and backup.
- Push: Uploading local commits to a remote repository, making them accessible to others.
- Fetch: Retrieving updates from a remote repository without automatically merging them.
- Pull: Downloading and merging changes from a remote repository into a local repository.
- Pull Request (PR): A request to review and merge changes from a branch into the main repository.

Common Workflows in Git

- Basic Version Control: Track changes in a single-user project by committing edits and maintaining history.
- Feature Branching: Create a new branch for each feature or bug fix, then merge it into the main codebase.
- Collaborative Workflow: Multiple contributors work on different branches, pushing and pulling changes via GitHub.
- Release Management: Maintain stable branches for production while developing new features in separate branches.



Hands-On Activity: Understanding Git in Practice

Now that we've covered Git workflows, let's apply these concepts in a hands-on session using VS Code, WSL, and GitHub.

In this activity, we will:

- Set up a local Git repository within WSL.
- Track changes, stage files, and commit updates.
- Sync our local repository with GitHub and manage remote changes.
- Revert to an earlier version and explore commit history.
- Work with branches, merge updates, and discuss rebasing.

Getting started:

- Open VS Code and connect to WSL.
- Ensure Git is configured with your username and email:

```
$ git config --global user.name "<your name>"
$ git config --global user.email <email>
```

Git in Practice: Creating a Local Repository

Step 1: Create a Project Folder

- Open a terminal in VS Code and create a new directory:
  ```
  $ mkdir <my-project> && cd <my-project>
  ```
- Open the newly created folder in VS Code using the Explorer tab.

Step 2: Initialize a Git Repository

- Go to the Source Control tab in VS Code.
- Click **Initialize Repository**.
- Alternatively, use the terminal:
  ```
  $ git init
  ```

Step 3: Verify or Add a `.gitignore` File

- VS Code may create a default `.gitignore` based on your project type.
- If not, create one manually:
  ```
  $ touch .gitignore
  ```
- Open the file and add common ignore patterns.

Example `.gitignore`

```
# Compiled files
*.o
*.exe
*.class

# Python files
__pycache__/
*.pyc

# Jupyter notebooks checkpoints
.ipynb_checkpoints/

# Logs & temp files
*.log
*.tmp

# VS Code settings
.vscode/

# Environment files
.env
venv/
node_modules/
```

Git in Practice: Creating an Initial Commit

Step 1: Create One or More Files

- In VS Code, create one or more new files (e.g., `main.py`, `utils.py`).
- Add some code to each file.

Step 2: Stage Files

- Only staged files are included in commits.
- Use the Source Control tab in VS Code to stage files.
- Alternatively, use the terminal:

```
$ git add .
```

Step 3: Commit Files

- Provide a meaningful commit message.
- Use the Source Control tab or run:

```
$ git commit -m "<Initial commit>"
```

Git Commands

- Create new files:
  `$ touch main.py utils.py`
- Stage all changes:
  `$ git add .`
- Commit with a message:
  `$ git commit -m "<Initial commit>"`

Best Practices

- Use clear commit messages (avoid "fixed bug").
- Commit small, logical changes frequently.
- Verify changes before committing:
  `$ git status`

Git in Practice: Syncing Local Repo to GitHub

Step 1: Publish Your Repository to GitHub

- In VS Code, go to the Source Control tab.
- Click **Publish to GitHub** and follow the prompts.
- Once published, VS Code automatically sets up the remote repository.

Step 2: Push Your Code to GitHub

- After publishing, push any local commits:

```
$ git push origin main
```

- Go to GitHub and verify your repository.

Step 3: Collaborating with Others

- A collaborator can now clone or pull the repository.
- They can make local changes, commit, and push updates.

Git Commands

- Verify remote repository:
  `$ git remote -v`
- Push local changes:
  `$ git push origin <branch-name>`
- Pull updates from GitHub:
  `$ git pull origin <branch-name>`
- Clone an existing repo:
  `$ git clone <repository-url>`

Best Practices

- Always pull before pushing to avoid conflicts.
- Use descriptive commit messages for better collaboration.
- Verify remote setup before pushing.

Git in Practice: Reverting to an Older Version

Step 1: Modify a File and Commit Changes

- Open an existing tracked file in VS Code and modify its content.
- Stage and commit the changes:

```
$ git add .
$ git commit -m "<Updated file>"
```

- Push the commit to GitHub:

```
$ git push origin main
```

Step 2: Identify an Older Version

- View the commit history to find the commit ID:

```
$ git log --oneline
```

- Copy the commit ID of the version you want to revert to.

Step 3: Revert to the Older Version

- Restore the file to the previous state:

```
$ git checkout <commit-id> -- <file-name>
```

- Stage and commit the reverted version:

```
$ git add <file_name>
$ git commit -m "<Reverted to older version>"
```

Git Commands

- View commit history:
  `$ git log --oneline`
- Restore a specific file:
  `$ git checkout <commit-id> -- <file-name>`
- Stage the reverted file:
  `$ git add <file-name>`
- Commit the reversion:
  `$ git commit -m "<Reverted file>"`
- Revert an entire commit:
  `$ git revert <commit-id>`
- Push the changes:
  `$ git push origin main`

Best Practices

- Always verify commit history before reverting.
- Use `git diff` to inspect changes before checkout.
- Use `git revert` to create a new commit that undoes the changes of an entire commit.

Git in Practice: Creating a New Branch

Step 1: Create a New Branch

- In VS Code, open the Source Control tab.
- Click on the **Branch** icon and select **Create New Branch**.
- Alternatively, use the terminal:

```
$ git branch <feature-branch>
$ git switch <feature-branch>
```

Step 2: Add New Files and Commit Changes

- Create new files in your repository and modify existing ones.
- Stage and commit the changes:

```
$ git add .
$ git commit -m "<Added new feature>"
```

Step 3: Verify the Branches

- Check the current branch:

```
$ git branch
```

- You now have two branches:
  - `main`: The main development branch.
  - `<feature-branch>`: The new feature branch you just created.

Git Commands

- Create a new branch:
  `$ git branch <feature-branch>`
- Switch to the new branch:
  `$ git switch <feature-branch>`
- List all branches:
  `$ git branch`
- Add and commit changes:
  `$ git add .`
  `$ git commit -m "<Feature update>"`

Best Practices

- Use descriptive branch names (e.g., `feature-login`).
- Keep feature branches short-lived; merge back soon.
- Regularly sync with `main` to prevent conflicts.

Git in Practice: Rebasing and Merging a Branch

Step 1: Ensure the Main Branch is Up to Date

- Switch to the main branch and pull the latest changes:
  ```
  $ git switch main
  $ git pull origin main
  ```

Step 2: Rebase the Feature Branch onto Main

- Switch back to your feature branch:
  ```
  $ git switch <feature-branch>
  ```
- Rebase onto the latest main branch:
  ```
  $ git rebase main
  ```
- If needed, resolve conflicts and continue:
  ```
  $ git rebase --continue
  ```
- Push the rebased branch to GitHub (force required):
  ```
  $ git push --force
  ```

Git Commands (1)

- Pull latest changes from main:
  `$ git pull origin main`
- Rebase feature branch onto main:
  `$ git rebase main`
- Force push after rebasing:
  `$ git push --force`
- Merge feature branch into main:
  `$ git merge <feature-branch>`

Step 3: Merge the Feature Branch into Main

- Switch to the main branch:
  ```
  $ git switch main
  ```
- Merge the feature branch into main:
  ```
  $ git merge <feature-branch>
  ```
- Push the merged changes to GitHub:
  ```
  $ git push origin main
  ```

Step 4: Clean Up the Feature Branch

- After merging, you can delete the feature branch:
  ```
  $ git branch -d <feature-branch>
  ```
- To remove it from GitHub as well:
  ```
  $ git push origin --delete <feature-branch>
  ```

Git Commands (2)

- Delete the feature branch locally:
  `$ git branch -d <feature-branch>`
- Delete the feature branch from GitHub:
  `$ git push origin --delete <feature-branch>`

Best Practices

- Always pull before rebasing to avoid conflicts.
- Use `git rebase --abort` if something goes wrong.
- Only use `git push --force` when necessary.

The Importance of Clear Documentation

Why is documentation essential?

- Helps you and others understand the purpose of your code.
- Makes debugging and future modifications easier.
- Ensures smooth collaboration in team projects.

What should be documented?

- Project-Level Documentation: README files, setup instructions.
- File-Level Documentation: Purpose of each script/module.
- Function & Class Documentation: What it does, inputs, outputs.
- In-Code Comments: Explain complex logic or key operations.

Best Practices

- Keep comments concise but informative.
- Update documentation when code changes.
- Use consistent formatting (e.g., docstrings).
- Avoid commenting obvious code.

Example: Poor vs. Good Comments

```
# Poor: Adds 1 to x
x += 1

# Good: Increment counter for next iteration
counter += 1
```

Writing Effective Comments and Docstrings

- Use inline comments sparingly, only where necessary.
- Use block comments for explaining sections of code.
- Use docstrings for documenting functions, classes, and modules.
- Describe parameters, return values, and behavior in docstrings.
- Keep comments meaningful and up to date.

Python Docstring Example

```
def add_numbers(a, b):
    """
    Adds two numbers and returns the sum.

    Parameters:
        a (int): First number
        b (int): Second number

    Returns:
        int: Sum of a and b
    """
    return a + b
```

Refactoring for Messy Codebases

Without proper management, a codebase quickly becomes unmanageable

- Multiple versions of the same functionality: `new_analysis.py`, `old_analysis.py`, `final_analysis.py`, `final_results.py` — Hard to track which version is correct.
- Messy directory structure: Files are scattered without clear organization.
- Poor naming conventions: Variables, functions, and files named ambiguously (e.g., `tmp1`, `x2`, `final_final.py`).
- Monolithic code without functions: Long scripts without modularization make debugging difficult.

Git for versioning, refactoring for code quality

- With Git, multiple versions of files should not exist—track changes instead.
- Refactoring is the process of restructuring code without changing its functionality to improve readability, maintainability, and efficiency.

Guidelines for a Well-Structured Directory

Key principles for organizing codebases

- Group logically related files into separate folders.
- Use clear, descriptive names for files and directories.
- Separate raw data, scripts, and results for clarity.
- Keep generated outputs (figures, logs) separate from source code.

Example: ML project structure

- `data/` → Store datasets (raw, processed)
- `implementations/` → Core implementations (Python scripts)
- `notebooks/` → Jupyter notebooks for exploration
- `results/` → Model outputs, logs
- `scripts/` → Scripts to generate figures/results
- `figures/` → Plots, graphs, and visuals
- `temp/` → Temporary files (ignored in Git)

Before: Messy Structure

```
ml-project/
├── data1.csv
├── data2.csv
├── script1.py
├── script2.py
├── notebook1.ipynb
├── notebook2.ipynb
├── results1.txt
├── results2.txt
├── figure1.png
├── figure2.png
├── generate_figures.py
├── process_data.py
├── final_script.py
└── temp/
    ├── temp_data.csv
    └── temp_script.py
```

After: Organized Structure

```
ml-project/
├── data/
│   ├── training_data.csv
│   └── test_data.csv
├── implementations/
│   ├── data_preprocessing.py
│   └── model_training.py
├── notebooks/
│   ├── exploratory_analysis.ipynb
│   └── model_evaluation.ipynb
├── results/
│   ├── training_results.txt
│   └── evaluation_metrics.txt
├── scripts/
│   └── generate_figures.py
├── figures/
│   ├── accuracy_plot.png
│   └── loss_plot.png
└── temp/
    ├── temporary_data.csv
    └── temporary_script.py
```

Refactoring in VS Code: Renaming and Modularization

VS Code automates basic refactoring, and extensions can enable more advanced restructuring. We will perform hands-on refactoring using the provided file `messy_script.py`.

Step 1: Rename Variables for Clarity

- Right-click on a variable (e.g., `tmp1`).
- Select **Rename Symbol** (or press `F2`).
- Change `tmp1` → `<new_name1>`, `tmp2` → `<new_name2>`.

Step 2: Extract Code into Functions

- Highlight repeated code (e.g., sum computation).
- Right-click and select **Refactor: Extract Method**.
- Rename the function meaningfully (e.g., `compute_sum`).

Step 3: Modularize by Moving Functions to a New File

- Create a new Python file: `math_utils.py`.
- Cut and paste extracted functions into `math_utils.py`.
- Import functions into the main script using:

```
from math_utils import compute_sum, compute_product
```

Step 4: Organize and Save Outputs Properly

- Store generated plots in a `figures/` folder.
- Keep scripts in `src/` and avoid cluttering the root folder.

Outline

Why Dependency Management Matters?

Software dependencies change over time, breaking code and making experiments irreproducible.

Common issues without proper dependency management

- Version Mismatch: Libraries evolve—your code may stop working with newer versions.
- Missing Dependencies: Running the code on a new machine may fail if required packages aren't installed.
- Inconsistent Python Environments: Different projects may require different versions of the same package.
- OS-Specific Differences: Package behavior may vary across operating systems.

How do we solve this?

- Use **virtual environments** (e.g., Conda) to isolate dependencies for each project.
- Use **Docker containers** for fully reproducible execution environments.

What is a Virtual Environment?

A virtual environment is an isolated workspace where dependencies can be installed without affecting other projects.

Why use virtual environments?

- Prevents Dependency Conflicts: Projects can require different versions of the same package.
- Ensures Reproducibility: The environment remains unchanged even if system packages update.
- Safe Experimentation: Installing a new package won't break global dependencies.

Common virtual environment managers

- **venv** – Python's built-in environment manager.
- **virtualenv** – An improved version of `venv`.
- **pyenv** – Manages different Python versions (but not dependencies).
- **pipenv** – Combines `pip` and `virtualenv`.
- **Conda** – Manages both environments and dependencies.

Comparing Virtual Environment Managers

Feature	venv	pyenv	pipenv	Conda
Manages dependencies?	✓	✗	✓	✓
Handles system packages?	✗	✗	✗	✓
Supports non-Python dependencies?	✗	✗	✗	✓
Works with different Python versions?	✗	✓	✓	✓
Checks dependency compatibility?	✗	✗	✓	✓
Cross-platform support?	✓	✓	✓	✓
Ideal for ML & Scientific Computing?	✗	✗	✗	✓

Why Conda is the Best Choice?

Conda is more than just a virtual environment manager—it also manages packages and dependencies.

Key advantages of Conda

- Handles Both Python and Non-Python Dependencies: Supports NumPy, TensorFlow, CUDA, OpenCV, etc.
- Automatic Dependency Resolution: Ensures package compatibility before installation.
- Supports Multiple Python Versions: Each environment can have a different Python version.
- Works Across Operating Systems: Ensures reproducibility across Windows, macOS, and Linux.
- Easy to Share Environments: Export and recreate environments using `<environment.yml>`.

Conda in Practice: Creating Isolated Environments

Step 1: Create Two Conda Environments

- Open a terminal in VS Code.
- Create an older environment with an older NumPy version:

```
$ conda create --name <env_old> numpy=1.15
```

- Create a newer environment with an updated NumPy version:

```
$ conda create --name <env_new> numpy=1.19
```

Step 2: Activate and Compare Environments

- Activate <env_old> and check NumPy version:

```
$ conda activate <env_old>
$ python -c "import numpy; print(numpy.__version__)"
```

- Activate <env_new> and check NumPy version:

```
$ conda activate <env_new>
$ python -c "import numpy; print(numpy.__version__)"
```

Conda Commands

- Create a new environment:
  `$ conda create --name <env-name> <packages>`
- Activate an environment:
  `$ conda activate <env-name>`
- Deactivate the current environment:
  `$ conda deactivate`
- Check all available Conda environments:
  `$ conda env list`
- Clone an existing environment:
  `$ conda create --name <new-env-name> --clone <existing-env>`
- Remove an environment:
  `$ conda remove --name <env-name> --all`
- Export an environment to a file:
  `$ conda env export [--from-history] > <environment.yml>`
- Recreate an environment from a file:
  `$ conda env create -f <environment.yml>`

Step 3: Create Two Python Scripts

- Open a terminal and create two files:

```
$ touch <script_A.py> <script_B.py>
```

- Copy the corresponding code into each file.

Step 4: Run the Scripts in Different Environments

- Activate an environment and try running both of the scripts:

```
$ conda activate <env_old>
$ python <script_A.py>
$ python <script_B.py>
```

- Switch environments and again try running both of the scripts:

```
$ conda activate <env_new>
$ python <script_A.py>
$ python <script_B.py>
```

Script A

```
import numpy as np

# Create a sample array
arr = np.array([1, 2, 3])

# Function may or may not work
scalar = np.asscalar(arr[0])
print("Scalar value:", scalar)
```

Script B

```
import numpy as np

# Create a sample array
arr = np.array([1, 2, 3])

# Function may or may not work
scalar = arr[0].item()
print("Scalar value:", scalar)
```

Which script works in which environment? What error do you see?

So What Exactly Happened?

We ran two scripts. One worked, and one failed. Why?

Basically, this is what happened:

- One script used `np.asscalar()`, which was removed in NumPy 1.19.
- The other used `arr.item()`, its replacement in newer versions.
- Since our environments had different NumPy versions, only one script worked in each.

What if no virtual environment was provided?

- The code would break immediately, with no clue about the required package versions.
- You'd have to guess and manually install older versions.
- For larger projects, reproducing past results could become impossible.

This is why we always export and share virtual environments with our code!

Why Do We Need Containers?

Limitations of virtual environments

- OS-Specific Differences: Conda manages Python dependencies, but system-level packages may still differ.
- Hardware Compatibility Issues: Running ML models may require specific CUDA versions.
- Difficult to Share: Conda environments must be manually recreated on each machine.
- Legacy Software Problems: Some applications (e.g., old versions of MATLAB) may no longer be installable on modern systems.

What if we could package everything—including the OS, libraries, and code—into a single unit? This is where **Docker** comes in.

What is Docker?

Docker is a containerization platform that packages everything needed to run software into an isolated environment.

How is Docker different from Conda?

- Conda manages Python libraries, but Docker includes everything (OS, drivers, and libraries).
- Docker ensures that code runs the same on any machine, regardless of installed system dependencies.
- Containers are portable and can be deployed anywhere, from a laptop to the cloud.
- Reviving Old Software: Docker allows running outdated applications (e.g., older MATLAB versions) even on modern OSes.

Think of Docker as a "lightweight virtual machine"—but much faster!

Docker in Practice: From Local to DockerHub to Other Machines

Goal: Package a NumPy 1.15 script into a Docker container and run it anywhere.

What we will do:

- Build a Docker container inside WSL.
- Push the container to DockerHub using VS Code.
- Pull and run the container from another machine (Windows/Mac).
- Verify that NumPy 1.15 works inside the container.

Why this matters?

- Ensures full reproducibility of experiments.
- Eliminates OS-specific dependency issues.
- Allows code to run on different machines without modification.

Docker in Practice: Create the Python Script Inside WSL

Step 1: Create the Script Inside WSL

- Open WSL in VS Code.
- Navigate to your project folder in the terminal:

```
$ mkdir <docker-numpy> && cd <docker-numpy>
```

- Create a Python script:

```
$ touch <docker_script.py>
```

Step 2: Add the Following Code

Python Script (<docker_script.py>)

```
import numpy as np
import platform
import sys

print("=======================================")
print("Running inside Docker container")
print(f"Python version: {sys.version}")
print(f"Operating System: {platform.system()} {platform.release()}")
print("=======================================")

# Create a sample array
arr = np.array([1, 2, 3])

# Show NumPy version
print(f"Using NumPy version: {np.__version__}")

# Deprecated function (only works in NumPy 1.15)
scalar = np.asscalar(arr[0])
print("Scalar value:", scalar)
```

Step 3: Create and Configure the Dockerfile

- Inside the same folder, create a Dockerfile:

```
$ touch <Dockerfile>
```

- Open the file and add the following content:

```
# Use an official Python base image
FROM python:3.7

# Set working directory
WORKDIR /app

# Install specific NumPy version
RUN pip install numpy==1.15

# Copy script into container
COPY <docker_script.py> .

# Command to run script
CMD ["python", "<docker_script.py>"]
```

Why use Python 3.7? NumPy 1.15 is incompatible with newer Python versions!

Step 4: Build the Docker Image

- Open a WSL terminal in VS Code.
- Navigate to your project directory and run:

```
$ docker build [-f <Dockerfile>] -t <numpy-container> .
```

Step 5: Run the Docker Container

- Execute the container and observe the output:

```
$ docker run <numpy-container>
```

Step 6: Verify Running Containers

- Check active and stopped containers:

```
$ docker ps -a
```

Basic Docker Commands

- List all images:
  `$ docker images`
- Show running containers:
  `$ docker ps`
- Show all containers (including stopped ones):
  `$ docker ps -a`
- Stop a running container:
  `$ docker stop <container-id>`
- Remove a container:
  `$ docker rm <container-id>`
- Remove an image:
  `$ docker rmi <image-id>`
- View container logs:
  `$ docker logs <container-id>`

If everything is set up correctly, the script should run successfully inside Docker!

Step 7: Log in to DockerHub from WSL

- Run the following command and enter your credentials:

```
$ docker login
```

Step 8: Tag and Push the Image

- Tag the image using your DockerHub username:

```
$ docker tag <numpy-container> <username>/<numpy-container>[:<tag>]
```

- Push the image to DockerHub:

```
$ docker push <username>/<numpy-container>
```

Your image is now available for anyone to pull and use!

Step 9: Pull the Image on Windows/Mac from DockerHub

- Open a terminal and run:

```
$ docker pull <username>/<numpy-container>[:<tag>]
```

Step 10: Run the Container

- Run the image on Windows or Mac:

```
$ docker run <username>/<numpy-container>
```

The script should now execute on Windows/Mac just like it did in WSL!

Outline

Managing Randomness for Reproducibility

Why does randomness matter in computational experiments?

- Many machine learning and simulation-based experiments rely on random sampling.
- Without proper control, the same code may produce different results on different runs.
- Failing to manage randomness makes results difficult (or impossible) to reproduce.

Challenges in managing randomness

- Multiple Random Number Generators (RNGs): Python has `random`, NumPy has its own RNGs, and frameworks like PyTorch and TensorFlow maintain separate RNG states.
- Seeding Is Not Enough: Setting a seed in one library does not guarantee reproducibility in another.
- Parallelization Issues: Randomness behaves differently when running across multiple CPU or GPU threads.

The Role of Seeds in Random Number Generation

Random number generators (RNGs) are not truly random—they are deterministic

- Computers generate "random" numbers using mathematical formulas.
- These formulas start with an initial value, called a "seed".
- If you use the same seed, you get the same sequence of numbers every time.

Why is `seed=42` everywhere?

- Many tutorials use 42 as a default seed.
- 42 is a pop culture reference to *The Hitchhiker's Guide to the Galaxy*.
- It has no special mathematical meaning—just a fun tradition!

Example: First Run

```
import random
random.seed(42)
print(random.random()) # 0.6394267984578837
```

Example: Second Run (Same Seed)

```
import random
random.seed(42)
print(random.random()) # 0.6394267984578837
```

Some Libraries Have Multiple Random Number Generators

Not all randomness comes from the same RNG!

- Some libraries maintain separate RNGs for different functions.
- Seeding one RNG may not affect all random operations.

Examples of multiple RNGs in the same package

- **NumPy**: The `numpy.random` module has two major RNG types:
  - Legacy RNG (`numpy.random.RandomState`) – uses a global RNG state shared across the program.
  - New Generator API (`numpy.random.Generator`) — each generator has its own independent RNG state.
- **Scikit-learn:** Some models (e.g., `RandomForestClassifier`) can use an independent RNG for data shuffling, making results inconsistent if not properly controlled.
- **PyTorch and TensorFlow:** Have their own RNGs separate from NumPy and Python's `random` module.

RNG in Practice: Seeding Multiple Random Number Generators

Step 1: Create a Python Script

- Open VS Code and create a Python script:

```
$ touch <rng_seeding_script.py>
```

Step 3: Execute the Script Multiple Times

- Observe if setting one seed influences the others.
- Check if the results are consistently the same.
- Remove some of the seed settings and rerun the script. What changes do you notice?

Step 2: Add the Following Code

Python Script (<rng_seeding_script.py>)

```
import random
import numpy as np
import torch
import tensorflow as tf

# Seeding Python's built-in RNG
random.seed(42)
print("Python random:", random.random())

# Seeding NumPy's legacy RNG (global state)
np.random.seed(42)
print("NumPy (legacy) random:", np.random.rand())

# Seeding NumPy's new Generator API (independent RNG)
rng = np.random.default_rng(42)
print("NumPy (new Generator) random:", rng.random())

# Seeding PyTorch's RNG
torch.manual_seed(42)
print("PyTorch random:", torch.rand(1).item())

# Seeding TensorFlow's RNG
tf.random.set_seed(42)
print("TensorFlow random:", tf.random.uniform((1,)).numpy()[0])
```

Random Number Generation and Parallelization

Parallel execution makes reproducibility harder

- Framework-Specific RNGs: PyTorch, TensorFlow, and NumPy use RNGs that must be seeded independently.
- GPU vs. CPU Differences: Random operations and seeding behaviors can vary across different hardware.
- Uncontrolled RNGs: Each thread may generate different random sequences if not properly seeded.
- Race Conditions: Threads may execute in a different order each time, leading to unpredictable results.

Best practices for reproducibility

- NumPy: Use `numpy.random.Generator` instead of global RNG for thread safety.
- PyTorch: Set both `torch.manual_seed(<seed>)` and `torch.cuda.manual_seed(<seed>)` for determinism.
- Assign Different Seeds to Parallel Workers: Avoid all threads sharing the same RNG state.
- Force Determinism in PyTorch:

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

RNG and Parallelization in Practice: Single Seed Across Workers

Step 1: Create a Python Script

- Open VS Code and create a Python script:

```
$ touch <parallel_rng_single_seed.py>
```

Step 3: Execute the Script Multiple Times

- Do all workers generate the same numbers?
- Why does this happen?

Step 2: Add the Following Code

Python Script (<parallel_rng_single_seed.py>)

```
import numpy as np
import multiprocessing

# Initialize the random number generator with a fixed seed
rng = np.random.default_rng(42)

def generate_random_numbers(worker_id):
    print(f"Worker {worker_id}: {rng.random(3)}")

if __name__ == "__main__":
    pool = multiprocessing.Pool(4)
    pool.map(generate_random_numbers, range(4))
```

RNG and Parallelization in Practice: Unique Seeds Per Worker

Step 1: Create a Python Script

- Open VS Code and create a second Python script:

```
$ touch <parallel_rng_unique_seeds.py>
```

Step 3: Execute the Script Multiple Times

- Do the workers now generate different numbers?
- Why does assigning different seeds fix the issue?
- What are some cases where independent RNGs matter?

Step 2: Add the Following Code

Python Script (<parallel_rng_unique_seeds.py>)

```
import numpy as np
import multiprocessing

def generate_random_numbers(worker_id):
    # Unique seed per worker
    rng = np.random.default_rng(42 + worker_id)
    print(f"Worker {worker_id}: {rng.random(3)}")

if __name__ == "__main__":
    pool = multiprocessing.Pool(4)
    pool.map(generate_random_numbers, range(4))
```

Outline

Why Just Uploading Code to GitHub Is Not Enough?

Sharing code on GitHub is a great first step, but not enough for full reproducibility.

Common problems when code is not properly packaged

- No execution order: No clear instructions on which scripts to run and in what sequence.
- Missing dependencies: Code relies on external libraries or private assets that are not included or documented.
- Undefined computing environment: Results may depend on OS type, hardware specifications, or software versions that are not documented.
- No stable versioning: Without proper versioning, results may change as code is updated.
- Lack of long-term accessibility: GitHub does not guarantee permanent storage or archival stability.

What Every Public Repository Should Include?

A well-documented public repository should contain:

- README File: Clearly documents all necessary details:
  - **Project Overview:** Brief description and links to associated research papers.
  - **Licensing Information:** How others can use, modify, and cite the code.
  - **Installation Instructions:** Steps to set up dependencies and environment.
  - **External Private Assets:** Any required proprietary datasets or software and how to acquire them.
  - **Execution Order:** Which scripts to run and in what order.
  - **Computing Environment Details:** OS, hardware configuration, runtime estimates.
- Dependency List: An exported `<environment.yml>` or `<requirements.txt>` file.
- Stable Versioning: Use GitHub Releases and tags for frozen versions.
- Permanent Archiving: Use Zenodo to assign a DOI for long-term accessibility.

Exporting Dependencies for Reproducibility

Step 1: Export Your Conda Environment

- Ensure your environment is activated:

```
$ conda activate <your-env>
```

- Export all installed dependencies:

```
$ conda env export > <environment.yml>
```

Step 2: Recreate the Environment

- A user can recreate the exact same environment using:

```
$ conda env create -f <environment.yml>
```

Always include `<environment.yml>` in your repository for reproducibility!

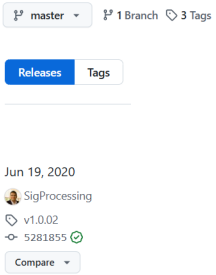
Tagging and Releasing a Version in GitHub

Why use GitHub Releases?

- Freeze a version of your code: Ensure others can access the exact version you used.
- Create a snapshot for citation: Useful for referencing a specific state of your project.
- Integrates with Zenodo: Enables long-term archiving.

Steps to tag and release a version in GitHub

- Go to your GitHub repository and click on **Tags**.
- Go to **Releases** and click on **Create a new release** or **Draft a new release**.
- Under **Choose a tag**, enter a new tag name (e.g., `v1.0.0`) or select an existing tag.
- Provide a release title and its description.
- (Optional) Mark the release as a **Pre-release** if it is not yet final.
- Click **Publish release** to finalize.



Zenodo for Long-Term Archiving

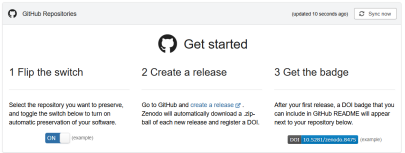
GitHub alone does not guarantee long-term accessibility!

Why use Zenodo?

- **Permanent DOI:** Ensures a stable reference to your code.
- **Versioned Snapshots:** Stores an immutable copy of your repository.
- **Integration with GitHub:** Automatically archives new releases.

Steps to enable Zenodo integration

- Sign in to **Zenodo** and link your GitHub account.
- Flip the toggle for your repository in Zenodo's GitHub settings.
- Each GitHub release will now trigger an archive on Zenodo.



GitHub Releases and Zenodo in Practice: Archiving for Reproducibility

Goal: Set up GitHub–Zenodo integration and verify that a GitHub release is archived in Zenodo.

Important Note: Zenodo (**zenodo.org**) is used for actual archiving, but for this exercise, we use **sandbox.zenodo.org** to avoid creating permanent DOIs.

Steps

- Go to **sandbox.zenodo.org** and create an account.
- In Zenodo, navigate to **GitHub** settings and enable repository access.
- Find your repository in Zenodo and flip the toggle to **On**.
- In GitHub, create a **Release** with a tag (e.g., `v1.0.0`).
- Verify that Zenodo picks up the release and generates a record.

The repository must be public for Zenodo to archive it!

Outline

Final Thoughts: Computational Reproducibility in Practice

What we covered:

- Experiment Management: Keep your codebase organized with Git, documentation, and refactoring.
- Dependency Management: Use Conda for virtual environments and Docker for full reproducibility.
- Randomness Control: Seeding matters—across libraries and in parallel execution.
- Code Sharing: README files, GitHub Releases, and Zenodo ensure long-term accessibility.

Key takeaway: *Tools evolve, but the principles of reproducibility remain the same!* Whether it's Git today or some futuristic AI-driven version control tomorrow, the core ideas—clear documentation, stable environments, and careful versioning—will always be relevant.

Final words: Reproducibility isn't just about science—it's also about kindness to your future self. *Six months from now, you won't remember why* `final_final_v3.py` *exists!*