12 – Git basics

Bálint Aradi

Scientific Programming in Python (2024)

https://atticlectures.net/scipro/python-2024/

Programming project (for lectures)

linsolver

- Program package for solving linear system of equation
- It should offer the Gaussian-elimination method (LU-decomposition)
- It should **read data** either from file or from console and write results to file or to the console
- It should have an **automatic test framework** for unit tests
- It should be well **documented** and **cleanly written**.

Note: This project serves didactical purposes only, the optimized routines of SciPy should be usually used to solve a linear system of equations.

Let’s start to develop!

Create the project folder

- Open a konsole (Linux, Mac) / Git Bash (Win)
- Initialize the right conda environment (scipro)
- Make a new directory (folder) “SciPro”

```
mkdir SciPro
```

- Change to the directory “SciPro”

```
cd SciPro
```

- Make the (new) directory “linsolver”

```
mkdir linsolver
```

- Change to the project directory “linsolver”

```
cd linsolver
```

Add initial content to the project

- Download the two project files and put them into the project folder

  solvers.py

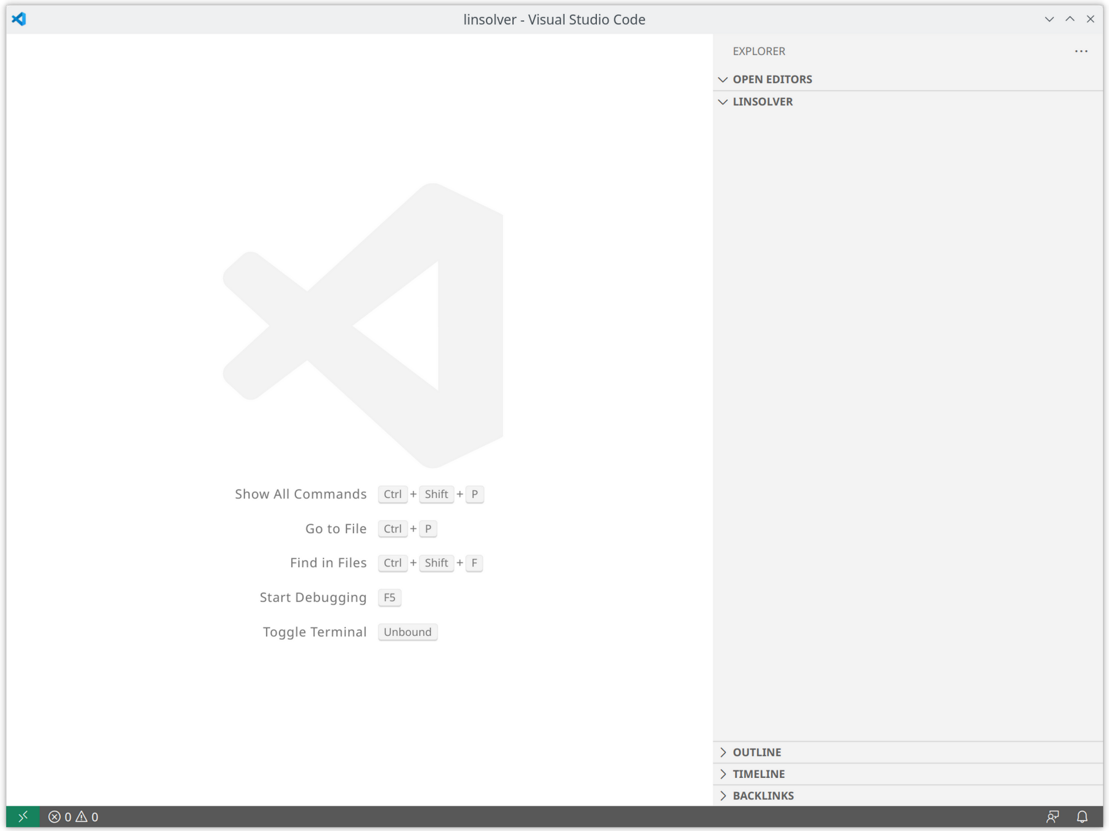
  test_solvers.py

Let’s start to develop!

Start VS Code from the project folder

```
code .
```

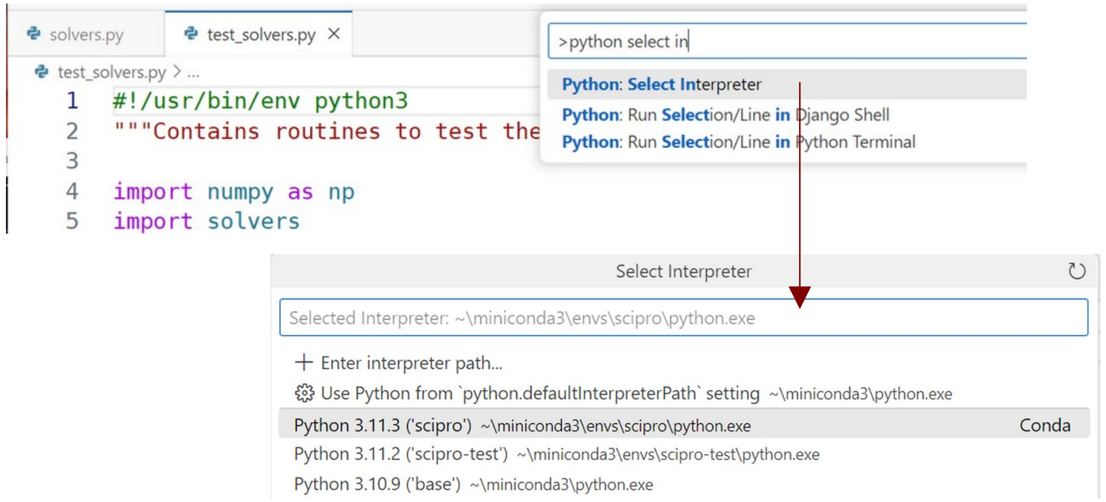
Pass the current directory as argument



(your editors appearance might differ sligthly)

- Open the two Python files and inspect them (**Ctrl-P** opens the file search menu)

- Select the Python Interpreter from your Conda environment (**Ctrl-Shift P** opens the command palette)

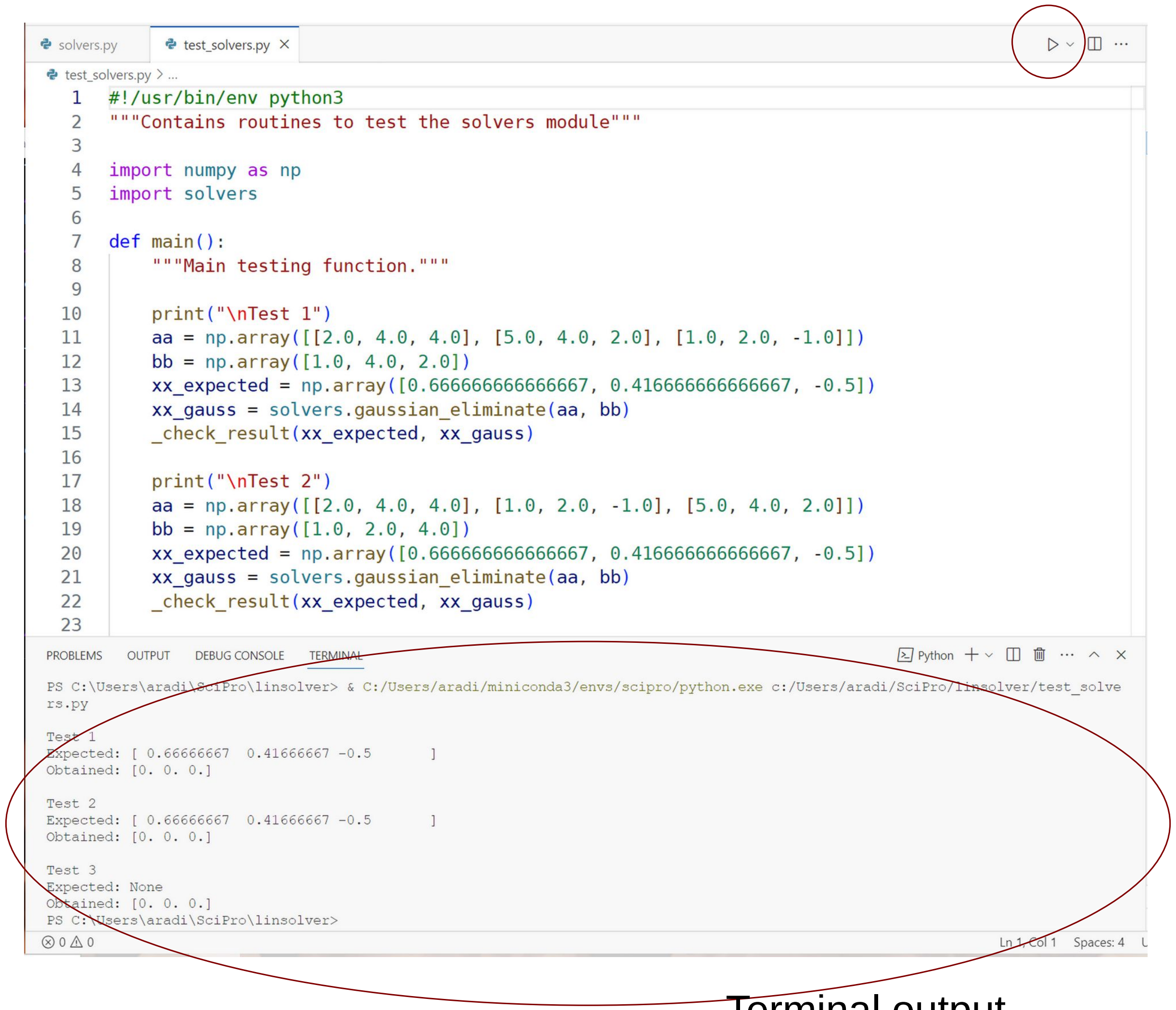

solvers.py
test_solvers.py
>python select in
Python: Select Interpreter
Python: Run Selection/Line in Django Shell
Python: Run Selection/Line in Python Terminal
Select Interpreter
Selected Interpreter: ~\miniconda3\envs\scipro\python.exe
Enter interpreter path...
Use Python from `python.defaultInterpreterPath` setting ~\miniconda3\python.exe
Python 3.11.3 ('scipro') ~\miniconda3\envs\scipro\python.exe Conda
Python 3.11.2 ('scipro-test') ~\miniconda3\envs\scipro-test\python.exe
Python 3.10.9 ('base') ~\miniconda3\python.exe

```python
#!/usr/bin/env python3
"""Contains routines to test the

import numpy as np
import solvers
```

Let’s start to develop!

- Run `test_solvers.py` from within your IDE



```python
#!/usr/bin/env python3
"""Contains routines to test the solvers module"""

import numpy as np
import solvers

def main():
    """Main testing function."""

    print("\nTest 1")
    aa = np.array([[2.0, 4.0, 4.0], [5.0, 4.0, 2.0], [1.0, 2.0, -1.0]])
    bb = np.array([1.0, 4.0, 2.0])
    xx_expected = np.array([0.666666666666667, 0.416666666666667, -0.5])
    xx_gauss = solvers.gaussian_eliminate(aa, bb)
    _check_result(xx_expected, xx_gauss)

    print("\nTest 2")
    aa = np.array([[2.0, 4.0, 4.0], [1.0, 2.0, -1.0], [5.0, 4.0, 2.0]])
    bb = np.array([1.0, 2.0, 4.0])
    xx_expected = np.array([0.666666666666667, 0.416666666666667, -0.5])
    xx_gauss = solvers.gaussian_eliminate(aa, bb)
    _check_result(xx_expected, xx_gauss)

```

```
PS C:\Users\aradi\SciPro\linsolver> & C:/Users/aradi/miniconda3/envs/scipro/python.exe c:/Users/aradi/SciPro/linsolver/test_solve
rs.py

Test 1
Expected: [ 0.66666667  0.41666667 -0.5       ]
Obtained: [0. 0. 0.]

Test 2
Expected: [ 0.66666667  0.41666667 -0.5       ]
Obtained: [0. 0. 0.]

Test 3
Expected: None
Obtained: [0. 0. 0.]
PS C:\Users\aradi\SciPro\linsolver>
```

Terminal output

If you get various error messages about connection to pylint in Code, install pylint (we’ll need it later anyway):

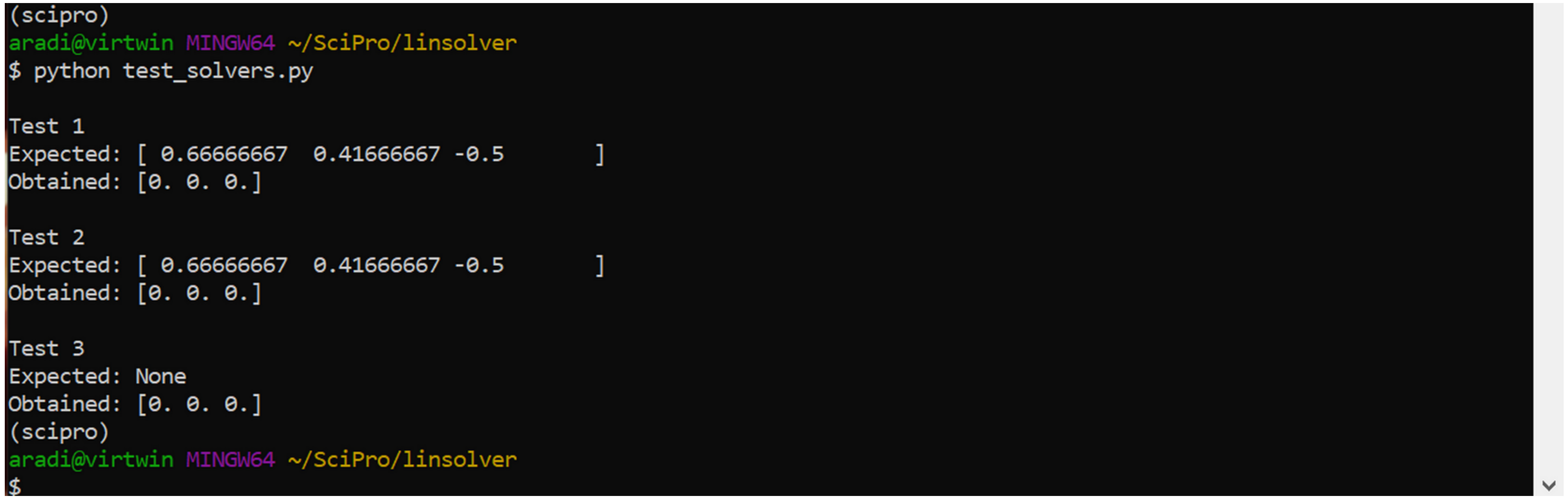
```
conda install pylint
```

Let’s start to develop!

- Run “test_solvers.py” from the command line
  (in a command line window, where Conda had been already activated)

```
python test_solvers.py
```
Windows

```
python3 test_solvers.py
```
Linux



```
(scipro)
aradi@virtwin MINGW64 ~/SciPro/linsolver
$ python test_solvers.py

Test 1
Expected: [ 0.66666667  0.41666667 -0.5       ]
Obtained: [0. 0. 0.]

Test 2
Expected: [ 0.66666667  0.41666667 -0.5       ]
Obtained: [0. 0. 0.]

Test 3
Expected: None
Obtained: [0. 0. 0.]
(scipro)
aradi@virtwin MINGW64 ~/SciPro/linsolver
$
```

The project apparently needs some development ...

- Before you change anything, the project should be set under **version control**

Typical scenario with version control

Scenario

- New project is started
- Program tested, everything works OK
- New functionality is added
- Suddenly, something does not work as supposed, although it was working before (note: testing framework apparently not satisfactory)

Solution work-flow with version control

- Go back in history
  to the last revision (evtl. by bisection), until a correctly working version is found
- Inspect the changes
  introduced in the snapshot (commit) and find out the reason for the failure
- Fix the bug
  in the most recent program version

Version control

Main tasks

- Document development history (store snapshots of the project)
- Help coordinating multiple developers working on the same project
- Help coordinating development of multiple versions of a project

Centralized VC (CVS, Subversion, …)

- Central server stores history database
- Developer must have connection to the server for most operations (especially for commits, checkouts or browsing history).

Distributed VC (Git, Mercurial, ...)

- Every developer has a local copy of the full development history
- Most operations do not require network connection (except synchronization between developers)

Introduce yourself to git

- Enter your name and email address (needed for the logs)

```
git config --global user.name "Bálint Aradi"
git config --global user.email "aradi@uni-bremen.de"
```

- Specify standard tools to be used

```
git config --global core.editor YOUR_EDITOR
git config --global diff.tool meld
```

- **--global** stores option globally, otherwise they apply to current project only
- Global options are stored in the **~/.gitconfig** file
- Current options can be listed with **--list**

```
git config --list
```

Windows `notepad`

MacOS `nano`

Linux

- `nano`
- `gedit` / `kate` / `featherpad` / `leafpad`
- `vim`, `emacs` (if you know what you're doing…)

Create a repository

- Initialize a repository in the project direcotry

```
cd ~/SciPro/linsolver/
git init
     Initialized empty Git repository in [...]/SciPro/linsolver/.git/
```

- **Files within the project directory** can be placed under version control
- Files within the .git directory should not be changed manually
- When copying project directory recursively (including the **.git** subdirectory) the entire revision history is copied

Put files under version control

```
git status
On branch main

No commits yet

Untracked files:
 (use "git add <file>..." to include in what will be committed)
       __pycache__/
       solvers.py
       test_solvers.py

nothing added to commit but untracked files present
(use "git add" to track)
```

Put files under version control

```
git add solvers.py test_solvers.py
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   solvers.py
        new file:   test_solvers.py

Untracked files:
  (use "git add <file>..." to include [...]
        __pycache__/
```

- Puts files under version control and makes a snapshot of their current state (**stage**)
- Staged files are written to the database at the next commit

Ignoring files

- **Files that should not be be version controlled** can be listed in **.gitignore** in the project directory

```
YOUR_EDITOR .gitignore
git add .gitignore
git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   solvers.py
        new file:   test_solvers.py
```

$\xrightarrow{\quad}$

```
__pycache__
```

- The .gitignore file should be also placed under version control

Commit staged files

- When commit is issued, staged files (in their staged state) are written to the database

```
git commit
[main (root-commit) 5270fa1] Kick off project
 3 files changed, 58 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 solvers.py
 create mode 100644 test_solvers.py

git status
On branch main
nothing to commit, working tree clean
```

Opens editor

Write log message (“Kick off project”), save & exit

Checking project history

- Show project history:

```
git log
commit 5270fa191b5cbe7a83e4b1e3d406c37793e4b27a (HEAD -> main)
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   ...

    Kick off project
```

- Individual commits are identified by **hash checksums**
- Checksums can be shortened as long as they are unambiguous
- **--oneline** option gives a short summary of the log messages (shows also shortened checksums)

```
git log --oneline
5270fa1 (HEAD -> main) Kick off project
```

Checking project history

- Revision history and log messages are shown in **reverse time order**

```
commit 2a3186299e14575a40b870cc3f8eb21c1e886809
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   ... [earlier]

    Add readme file

commit 04d386638495386aa29ee99e4928aad2e7731f39
Author: Bálint Aradi <aradi@uni-bremen.de>
Date:   ...[later]

    Add first stub files
```

- If history is longer than a page, it is shown page-wise via the **default pager** (e.g. less)

Navigation:	**[Page Up/Down]**	Move up/down
	q	Quit

Git-workflow

- Set up git global for your account

```
git config --global ...
```

- Set up the repository for your project

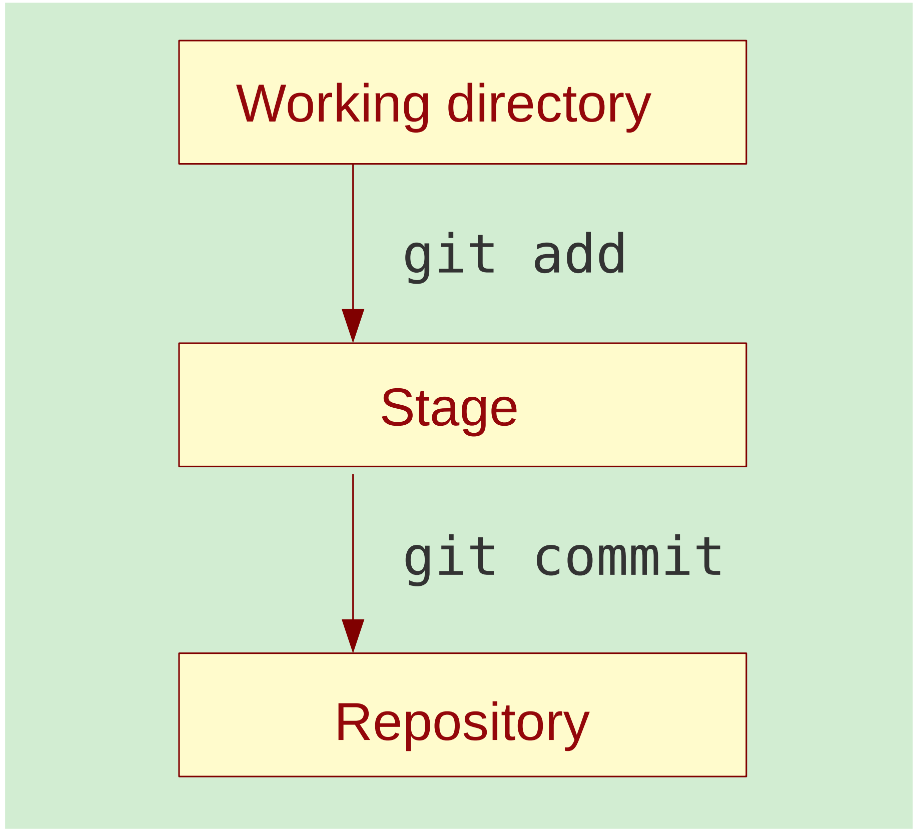
```
git init
```

- Edit files in your project
- Stage files / changes

```
git add ...
```

- Commit staged changes into repository

```
git commit ...
```



- It is possible to stage all changes in all files which are already under version control:

```
git add -u
```

Some git remarks

- Changes should be commited, if implementation of a feature is finished
- Development history should be easy to follow based on the log messages
- Changes within a commit should be small enough so that a developer can easily follow and understand them.
- Log messages should contain a **short sentence** (max. 50-60 chars), optionally **followed by an empty line and a more detailed description**.
(See for example: How to Write a Git Commit Message)

```
Implement LU-decomposition with back substitution

LU-decomposition is implemented without permutation. Check
for linear dependency is not implemented yet.
```

- Short (one-liner) log messages can be passed on the command line

```
git commit -m "Add first stub files"
```