Scientific Computing Using Python

Scientific Software Development

14th of June, 2023

Thomas Arildsen
tari@its.aau.dk

CLAAUDIA

Aalborg University

Agenda

Introduction

Learning objectives for this lecture:

- to know the principles of reproducible research and know how to apply its basic ideas.
- to know the characteristics of computational science software development.
- to be able to apply a pragmatic software development procedure for computational science.

Reproducible Research

Reproducible research

Reproducible research:

- Almost since the beginning of research; scientific ethics is: "Openness: Scientists should share data, results, methods, and tools." [Res14]
- Unfortunately there are many examples where lack of reproducibility prevented others discovering errors [Edi11; BP05].
- Reignited discussion started around 20 years ago [BD95].
- Increased focus in almost all scientific disciplines – specific advice depends on the scientific domain.

Exercise:

- Is reproducibility of your research (and software) important? Why – or why not?

Reproducible research
Levels of reproducibility

[VKM09] proposed 6 levels of reproducibility:

1. The results cannot be reproduced by an independent researcher.
2. The results cannot seem to be reproduced by an independent researcher.
3. The results could be reproduced by an independent researcher, requiring extreme effort.
4. The results can be reproduced by an independent researcher, requiring considerable effort.
5. The results can be easily reproduced by an independent researcher with at most 15 minutes of user effort, requiring some proprietary source packages (MATLAB, etc.).
6. The results can be easily reproduced by an independent researcher with at most 15 min of user effort, requiring only standard, freely available tools (C compiler, etc.).

Reproducible research
Best practices for software

- Make code and data available such that the complete program can be executed.
- For a scientific paper . . . each figure can be reproduced by executing e.g. `$ python figure1.py`, . . .
- Provide testing of the code.
- Provide documentation of the code.
- . . .

Why?

- Support research ethics.
- Quality of code increases when you know other people will see it.
- Increase impact of publication (e.g. citations) [VKM09].

Computational Science Characteristics

Computational science characteristics

Computational scientists are . . .:

- often identical to the immediate customers [KTH11] – there may though be secondary customers of the research output being the sponsor or research council financing the research.
- limited by typically not being able to write a clear specification from the beginning – normally the specifications develop iteratively as the research develops and results appear [Car+07; Seg05; SM08; Han+09; Nan+14]. The development process is therefore iterative in nature as specifications, test, algorithmic methods etc. develop over time.
- domain experts in their respective scientific areas – they normally see computer science as a tool and not the objective of their interest in itself [Kel07; Wil06; Han+09; MS09].
- limited in nature when considering tests as the software is developed to gain new insights and knowledge – thereby no ‘oracle’ exists which can serve as a reference for tests [SM08; HK09; Han+09]. Although correctness may not be possible to test, trustworthiness could be the objective of testing [HK09].

Computational science characteristics

Some relevant background information:

- The software is not the objective in itself but merely a tool to do the actual research [Kel07; Wil06; Edd09].
- Supercomputers and supercomputing takes many headlines in scientific papers, Top500 etc. but in reality the majority of scientific computing is done via desktop computers, servers and small clusters [Han+09].
- One big challenge is that the computational science community lacks knowledge of basic tools like debuggers, profilers, version control systems etc. [Wil06].
- The teams developing computational software are often very small – contrary to many software development teams. (Often belonging to another organization — tool and skill mismatch can occur).

Software Development

Software development methods

Many approaches. A few well-known

- Code-and-fix ("Cowboy programming").
- Plan-/stage-driven methods ....
- Iterative methods ....
- Agile ....
- Test driven development ...

Software development methods

Many approaches. A few well-known

- Code-and-fix ("Cowboy programming").
- Plan-/stage-driven methods ....
- Iterative methods ....
- Agile ....
- Test driven development ...
- We have nothing to add, instead we will present a number of ideas we believe can help improve *your* scientific software development process.

How to improve your software development

Development process

Specifications

- Describe the objective or functionality as precisely as possible.
- Already start considering tests. Two aspects are critical here:
  1. what should be tested; and
  2. how should the test be done.

How to improve your software development

Development process

Algorithm

- formulate a simple, naïve, and robust algorithm where the only aim is to deliver the correct result. The algorithm is a mathematical or formal description of how we need to relate input parameters to the output so that we get the desired functionality.
- Do not care if this means increased execution time since we are aiming for a reference implementation of the code. It is crucial not to be tempted to optimise at this point – in best case it is a waste of time. In worst-case, errors are introduced or we end up optimising things that do not really need to be optimised. No matter what, optimised code is very often less easy to understand and thereby also more difficult to maintain.

How to improve your software development

Development process

Code

- Selection of folder structure, names of modules, functions, classes and files. Changing these at a later stage is a nightmare as these names are used in both the software and the documentation.
- Start structuring using a version control system.
- The coding part covers two areas: 1) the functional module; and 2) unit testing.
- Consider libraries – not least in Python there are many high-quality libraries for easy use.
- Testing may require a substantial effort when you start to learn testing: accept it! When your testing skills improve, so does your efficiency. In the end it will pay off.
- Do not have two copies of anything — modularise the code.
- Be open for code-reviews to let others see the code — you might learn something.

How to improve your software development
Development process

Documentation

- No documentation = useless software. Making the documentation *must* be easy – otherwise it is not made or is obsolete.
- A paper is a form of documentation – but may not document *how* to use the code.
- When using Python it is quite obvious to use docstrings for the documentation. This means that the documentation co-exists with the code and that we are more likely to keep it up to date. May be updated as the project evolves.
- It is possible to later use tools such as Sphinx to automatically extract documentation for nice formatting in HTML or PDF.
- Use code comments as recommended by PEP8. Use comments when needed and not to explain the obvious. Do not use comments as a remedy for poor coding. Comments are useful for explaining smaller functional parts that might not otherwise be obvious.

How to improve your software development
Development process

Testing

- Write test code before/alongside writing functional code. Use whatever test procedure you like best – the test you actually write is the best test. If `doctest` is best for you then use it. If `unittest` is best for you then use that.
- Some testing is better than none: test at least the main functionality.
- Watch out for false npositives.
- (Automatic) tests should generally be quiet in case they do not reveal any problems and only provide information when errors occur.
- Comprehensive examples may provide more realistic test cases which often exercise many parts of the code (for example end-to-end use cases).
- Bug reporting system necessary even for single-person projects (every-one can forget a bug they wanted to fix).

How to improve your software development

Optimisation

- Optimisation is only an issue if the validation shows that the code or parts of the code does not meet the requirements with respect to e.g. execution time, memory usage and accuracy.
- Do not focus excessively on execution time — your time is important too.
- Base optimization on actual measurements. Do not guess.
- Depending on what performance metric is not within bounds, it may be necessary to investigate a new algorithm, coding a critical part in Fortran/C, parallelise the code, vectorise the code or whatever matches the objective best.

How to improve your software development

Recommendations

Code Development Tools

- Use appropriate tools for debugging, testing. . .
- Give new inventions a chance and find your preferred selection of tools that provide you with a toolbox that supports your coding effort.

Libraries

- The Python Package Index (PyPI) lists more than 380,000 projects as of June 2022.
- Do not reinvent the wheel – but ensure the packages have the quality you expect by ensuring they include proper documentation, testing etc.
- Many compiled libraries exist for numerical linear algebra, optimisation, Ordinary Differential Equations (ODEs) etc.

Conclusions

Conclusions

Different software development ingredients have been presented:

- Reproducible research — for your community and yourself.
- Scientific software development and developers differs from e.g. administrative software development and developers.
- Hints and recommendations to improve your software development.

Conclusions I
Literature

[BD95] J. B. Buckheit and D. L. Donoho. “WaveLab and Reproducible Research”. In: *Lecture Notes in Statistics: Wavelets and Statistics*. Ed. by A. Antoniadis and George Oppenheim. Springer-Verlag, 1995, pp. 55–81.

[BP05] M. Barni and F. Perez-Gonzalez. “Pushing Science into Signal Processing”. In: *IEEE Signal Processing Magazine* 22.4 (2005), pp. 119–120. DOI: 10.1109/MSP.2005.1458324.

Conclusions II
Literature

[Car+07] J. C. Carver et al. "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies". In: *29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, Minnesota, USA, May 2007, pp. 550–559. DOI: 10.1109/ICSE.2007.77.

[Edd09] Steven L. Eddins. "Automated Software Testing for MATLAB". In: *Computing in Science & Engineering* 11 (1 2009), pp. 48–54. DOI: 10.1109/MCSE.2009.160.

[Edi11] Editorial. "Devil in the Details". In: *Nature* 470 (2011), pp. 305–306. DOI: 10.1038/470305b.

Conclusions III
Literature

[Han+09] Jo Erskine Hannay et al. "How Do Scientists Develop and Use Scientific Software". In: *ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE'09)*. Vancouver, Canada, May 2009, pp. 1–8. DOI: `10.1109/SECSE.2009.5069155`.

[HK09] Daniel Hook and Diane Kelly. "Testing for Trustworthiness in Scientific Software". In: *ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE'09)*. Vancouver, Canada, May 2009, pp. 59–64. DOI: `10.1109/SECSE.2009.5069163`.

Conclusions IV
Literature

[Kel07] Diane F. Kelly. “A Software Chasm: Software Engineering and Scientific Computing”. In: *IEEE Software* 24 (6 2007), pp. 118–120. DOI: `10.1109/MS.2007.155`.

[KTH11] Diane Kelly, Stefan Thorsteinson, and Daniel Hook. “Scientific Software Testing: Analysis with Four Dimensions”. In: *IEEE Software* 28 (3 2011), pp. 84–90. DOI: `10.1109/MS.2010.88`.

Conclusions V
Literature

[MS09] Chris Morris and Judith Segal. "Some Challenges Facing Scientific Software Developers: the Case of Molecular Biology". In: *Fifth IEEE International Conference on e-Science*. Oxford, UK: Springer-Verlag, Dec. 2009, pp. 216–222. DOI: `10.1109/e-Science.2009.38`.

[Nan+14] A. Nanthaamornphong et al. "Building CLiiME via Test-Driven Development: A Case Study". In: *Computing in Science & Engineering* 16 (3 2014), pp. 36–46. DOI: `10.1109/MCSE.2014.33`.

Conclusions VI
Literature

[Res14] David B. Resnik. "Scientific Misconduct and Research Integrity". In: *Handbook of Global Bioethics*. Ed. by Henk A. M. J. ten Have and Bert Gordijn. Springer Netherlands, 2014, pp. 799–810. DOI: 10.1007/978-94-007-2512-6_128.

[Seg05] Judith Segal. "When Software Engineers Met Research Scientists: A Case Study". In: *Empirical Software Engineering* 10.4 (2005), pp. 517–536. DOI: 10.1007/s10664-005-3865-y.

Conclusions VII
Literature

[SM08] Judith Segal and Chris Morris. “Developing Scientific Software”. In: *Computing in Science & Engineering* 25 (4 2008), pp. 18–20. DOI: `10.1109/MS.2008.85`.

[VKM09] P. Vandewalle, J. Kovačević, and M.Vetterli. “Reproducible Research in Signal Processing [What, why, and how]”. In: *IEEE Signal Processing Magazine* (May 2009), pp. 37–47.

[Wil06] Greg Wilson. “Where’s the Real Bottleneck in Scientific Computing?” In: *American Scientist* 94.10.1511/2006.1.5 (2006), pp. 5–6.