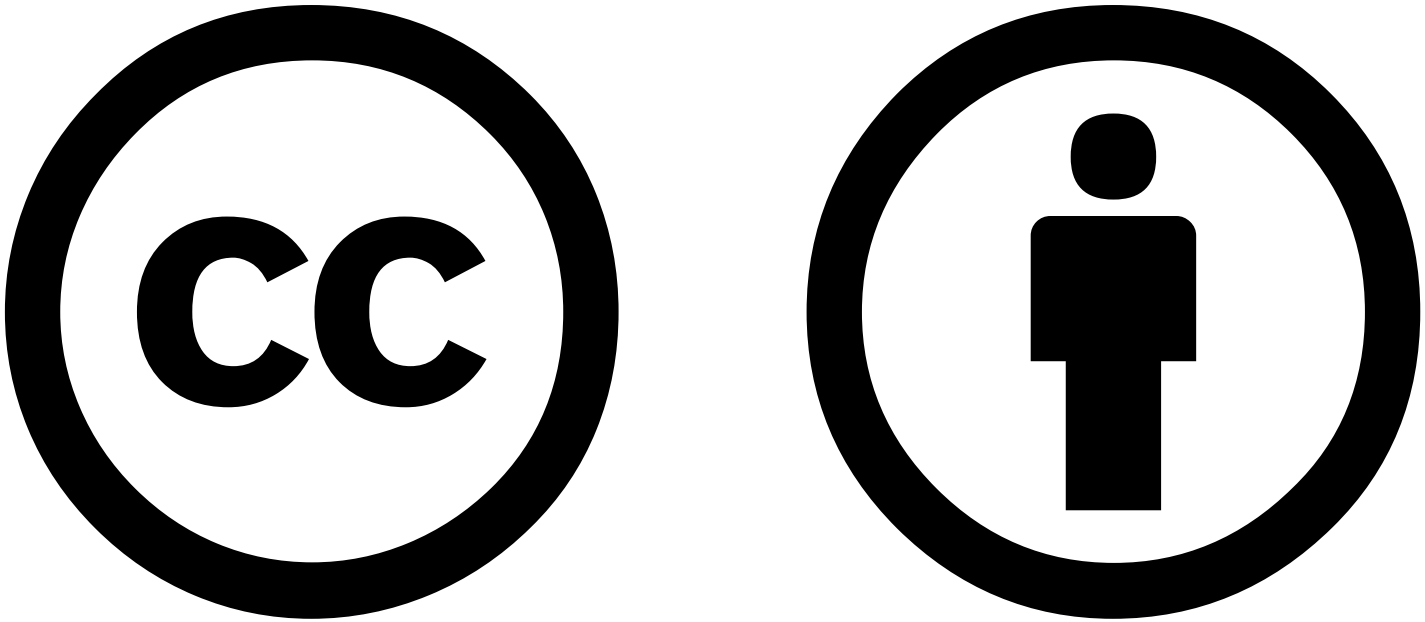
Scientific software development best practices

Thomas Arildsen
tha@es.aau.dk

Dept. of Electronic Systems
Aalborg University

October 31, 2014



Scientific software development best practices

Agenda

- Introduction
- Scientific Software Development
- Write Programs for People
- Let the Computer Do the Work
- Make Incremental Changes
- Don't Repeat Yourself
- Plan for Mistakes
- Optimize Software Only after It Works
- Document Design and Purpose
- Collaborate
- Summary

Introduction

Introduction

Scientific software development

- When we code models, simulation scripts, and scientific computing in general, we should not just sit down in front of our computers and start typing as we begin to consider what the program should do.
- Going about this in a sensible order and with some structure around the process is necessary to avoid serious mistakes that will cost us a lot of time and energy to fix afterwards.
- Numerous programmers, software engineers and scientists before us have made countless mistakes that we can learn from and at least try to do it "right" from the beginning.

Introduction

Intended learning outcomes

After today's lecture you will have knowledge about:

- Some "best practice" guidelines for developing software.
- General advice for developing software with a scientific software angle.
- Principles of testing and validation of scientific software.

You will be able to:

- Apply a set of guidelines to the development of software in your projects.
- Structure your software development process for clarity and reliability of the code.

Introduction

Source

These slides are based on and quote from:

Greg Wilson et al. “Best Practices for Scientific Computing”. In: *PLoS Biology* 12.1 (Jan. 2014), e1001745. DOI: 10.1371/journal.pbio.1001745.

Scientific Software Development

Scientific Software Development

What do we mean by that?

Scientific software vs. "other software"

- When you read about software development practices, these are often concerned with software that is aimed at helping users do something (word processing, spreadsheets, web browsing etc., etc.).
- Or, maybe we are talking about *embedded software* – software that is not directly visible to a user but typically plays a more technical role of making some technological "apparatus" work.
- What is scientific software?

Scientific Software Development

Scientific software

- Often written by scientists/researchers for their own use.
- Not designed to satisfy detailed requirements of human users.
- Mainly intended to answer one or more fairly specific scientific, often mathematically formulated questions.
- Should support reproducibility of results.
- Validation important.

Scientific Software Development

- Several of these slides' pieces of advice are not aimed at scientific software *development* as such.
- Rather, they are more generally aimed at making science *reproducible*.
- However, this is also a very useful practice for us to learn.

Write Programs for People

Write Programs for People

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*
>
> – *Martin Fowler, 1999*

We need to write software that both executes correctly *and* is readable and understandable to others.

- If not, it becomes difficult for others (and you) to understand what the program does.
- Also makes it difficult to check whether the program does it correctly.

Write Programs for People

Humans are simple-minded...

A program should not require its readers to hold more than a handful of facts in memory at once. [1].

- We can only hold about a handful of items in "working memory" at a time.
- "Items" can be a single fact or some chunk of facts.
- Take this principle into account by dividing our programs logically into functions, each of which carries out a single task.
- Functions – and the tasks they carry out – should be easily understandable.

Write Programs for People

Humans are simple-minded...

Example:

```
1 def line_slope(x1, y1, x2, y2):
2     # Some calculation
```

vs.

```
1 def line_slope(point1, point2):
2     # Some calculation
```

Write Programs for People

Naming conventions

Make names consistent, distinctive, and meaningful. [1].

- It makes the code easier to understand when the reader does not have to switch between different naming conventions through the document.
- Use names that describe what the given variable or function is.
- For example:
  - Not just `a` or `foo`.
  - Not too similar – `result1` and `result2`.
- On the other hand, if a function's documentation clearly states that its purpose is to evaluate the polynomial $f(x) = ax^2 + bx + c$, maybe `a`, `b`, and `c` are not such a bad idea...

Write Programs for People

Style and formatting

Make code style and formatting consistent. [1].

- Do not mix different case styles.
- For example:
  - Do not use both `CamelCase` and `pothole_case`.
- In Python, for example, do not mix tabs and spaces.
- Keep indentation levels consistent.

Write Programs for People

Style and formatting

Make code style and formatting consistent. [1].

- Do not mix different case styles.
- For example:
  - Do not use both `CamelCase` and `pothole_case`.
- In Python, for example, do not mix tabs and spaces.
- Keep indentation levels consistent.

Example:

```
1 def function1(arg1, arg2):
2       sum = arg1 + arg2
3       difference = arg1 - arg2
4       return (sum, difference)
5
6 def function2(arg1, arg2):
7     product = arg1 * arg2
8     fraction = float(arg1) / arg2
9     return (product, fraction)
```

Let the Computer Do the Work

Let the Computer Do the Work

The computer is your slave. . .

Make the computer repeat tasks and save recent commands in a file for re-use. [1].

- If we repeat things over and over again, such as processing large numbers of files the same way or regenerating figures for new data, we waste time and are bound to make mistakes at some point.
- In Python, for example, script tasks to avoid typing things over and over.
- In command-line interfaces, use command history to recall recent commands and only change the necessary parameters. (Also available for commands typed manually in IPython).
- Generally, script it if you can in preference to recalling and editing commands manually.

Let the Computer Do the Work

Workflows

Use a build tool to automate workflows. [1].

- Particularly for compiled languages, it is common to use a *build tool* to compile and link program files, i.e. keeping track of compiling and linking only the necessary files.
- Examples: *Make* – a classic for C/C++ etc., *ant* – particularly used for Java.
- These tools can also be used for more general tasks than compiling and linking.
- We can use them for our "computing" tasks, such as: "To produce figure 4, we need to make sure that `program1` and `program2` have been run and that `analysis` has been run afterwards to combine their data before we finally run `generate_figure`."
- For Python, see `http://paver.github.io/paver/`, `http://www.scons.org/`.

Make Incremental Changes

Make Incremental Changes

"Agile" development

Work in small steps with frequent feedback and course correction. [1].

- We may not know from the beginning, exactly all of the steps that the program must go through (particularly true for scientific software development).
- Better to work in small steps than trying to plan everything months or years ahead.
- Work in steps of about an hour grouped into iterations of about a weeks duration.
- Accommodates cognitive constraints of (human) programmers.

Make Incremental Changes

Versioning

Use a version control system. [1].

- A version control system (VCS) is a program that stores and keeps track of different *versions* of files in a *repository*.
- It is typically used for software development, i.e. storing versions of source code files etc., but it can be applied much more generally to all types of files.
- Working with a VCS is often based on *differences* between versions of the individual files.
- Ideally for text-based files. Not so well-suited for binary files.
- Can be used for example for (LaTeX) report documents as well as your source code.
- VCS examples: Subversion, git, Mercurial.

Make Incremental Changes

Versioning

Put everything that has been created manually in version control. [1].

- For example:
  - Program source code.
  - Report documents.
  - Working notes.
- Files that can be derived automatically from the above are not necessary to track in the VCS (maybe in some cases for convenience).
- Especially in the case of large binary files, it might be a good idea to store them externally but store meta-data about them in the VCS.
  - git has add-ons for this sort of thing: git-media, git-annex, git-submodules.

Don't Repeat Yourself

Don't Repeat Yourself

Uniqueness

Every piece of data must have a single authoritative representation in the system. [1].

- Anything that is duplicated in two or more places is difficult to keep track of.
- Sooner or later, we are going to forget to update one of these representations of the data or code, and our data or program becomes inconsistent.
- For example, physical constants used in a program should be define in only one place so it is unambiguous which definition is being used by the program.
- Raw data files (e.g. images, audio, measurements of some kind) should exist in only one copy.

Don't Repeat Yourself

Modularity

Modularize code rather than copying and pasting. [1].
(Small scale, own code)

- If a certain functionality is needed several places in a program, modularize it by defining it as a function which is called each of these places.
- Cloning code and writing this functionality into the source code at different places in the program introduces several places we need to maintain when for example correcting bugs etc.
- Also helps people remember the functionality as one single "chunk", making it easier to keep track of.
- Modularized code is easier to re-use elsewhere.

Don't Repeat Yourself

Re-use

Re-use code instead of rewriting it. [1].
(Large scale)

- If you need particular functionality in your program that others have programmed before you, use available code instead of programming it yourself.
- Huge amounts of software that might solve just your problem are available as free, open-source software online.
- You probably cannot program it more efficiently yourself anyway.
- For example, we utilise lots of functionality from NumPy and Matplotlib instead of implementing it ourselves.

...

Don't Repeat Yourself

Re-use

Re-use code instead of rewriting it. [1].
(Large scale)

- If you need particular functionality in your program that others have programmed before you, use available code instead of programming it yourself.
- Huge amounts of software that might solve just your problem are available as free, open-source software online.
- You probably cannot program it more efficiently yourself anyway.
- For example, we utilise lots of functionality from NumPy and Matplotlib instead of implementing it ourselves.
- That being said, for educational purposes it can be a good idea to program certain project-relevant functionality yourselves as it helps you understand it better.

Plan for Mistakes

Plan for Mistakes

S... happens...

Add assertions to programs to check their operation. [1].

- An assertion is a conditional statement of something we know must be true.
- We add the statement as a "sanity check".
- If the statement fails, we know something is wrong.
- Example:

```
1 def rectangle_area(height, width):
2     assert height >= 0
3     assert width >= 0
4     return height * width
```

- Serve to catch errors as soon as possible – simplifies debugging.
- Also serve as documentation – help explain how the program works.

Plan for Mistakes

Testing

Use an off-the-shelf unit testing library. [1].

- Testing is important to make sure that the program works correctly.
- Unit tests test whether a single part of the software, for example a function, works as intended.
- Integration tests test whether different parts of the code work correctly when used together.
- For example, if we have implemented a function to numerically evaluate some function, maybe we can analytically determine the correct value in some selected points.
  A test can then be to run the function on these points and compare the returned values to the analytically calculated values.
- In Python, we can use the module `unittest` to automate this kind of testing.

Plan for Mistakes

Testing and debugging

Turn bugs into test cases. [1].

- When we find bugs in our programs (and we will. . .), we should turn those bugs into tests with code that trigger this bug.
- This prevents this (kind of) bug from re-appearing at a later point in time and/or at another place in the program.
- Proper testing improves confidence in our program.
- Also encourages programmers to write code that is easier to test. This way it usually becomes easier to understand as well.

Plan for Mistakes

Debugging

Use a symbolic debugger. [1].

- You may be able to debug your program using for example `print` statements here and there to watch the values of certain variables.
- This is time-consuming and it can be very tricky to track down the source of a problem.
- Using a so-called symbolic debugger lets you inspect your program at run-time – much more efficient.
- You can set break-points where the debugger stops the program and you can inspect the values of different variables at that point in the program.
- You can also step through the code line by line and observe what happens in each line.

Optimize Software Only after It Works

Optimize Software Only after It Works

Premature optimization is the root of all evil.
– Donald Knuth, 1974

- The father of TEX once said this and these words contain a lot of useful wisdom...
- Once our program works correctly (which we can identify through testing), we can start looking at how to make it faster, use less memory, or maybe just more easily understandable.

Optimize Software Only after It Works

Profiling

Use a profiler to identify bottlenecks. [1].

- In many cases, we cannot predict correctly which parts of our program take up the largest amount of time.
- A profiler tool analyses our program at run-time and records how much time the program spends in each line or function of the program.
- This allows us to identify which parts of the program are particularly heavy (consume the largest fractions of the run time).
- Having identified these "bottlenecks", we can then efficiently concentrate on optimizing the one or few places where it really makes sense.
- It usually only makes sense to optimize a few selected parts of the code.

Optimize Software Only after It Works

Programming language

Write code in the highest-level language possible. [1].

- Programmers tend to produce the same amount of lines of code per time regardless of the programming language.
- Using a higher-level programming language, you can usually do more with less code and so, programming is more efficient here.
- Then we can, if necessary, optimise specific parts of the code in a lower-level language later on.
- In both of the above aspects, Python is an excellent choice of language.
  - It is high-level and thus efficient to program in.
  - It makes it easy to integrate code from other, lower-level languages such as C or Fortran if needed.

Document Design and Purpose

Document Design and Purpose

Documentation

Document interfaces and reasons, not implementations. [1].

- It is important to keep code well-documented to make it easier for the reader to understand and for others to maintain later on.
- Documentation on design decisions, i.e. why things were done the way they were done are useful.
- Documentation of interfaces is useful. For example documentation at the beginning of a function that explains what the function does, which arguments it takes, and what it returns.
- Documentation of the detailed mechanics in the code is often not very useful:

```
i = i + 1    # Increment variable 'i' by one
```

Document Design and Purpose

Refactoring

Refactor code in preference to explaining how it works. [1].

- In some cases, particular parts of code may need unusually long and complicated descriptions.
- In such cases it is often better to re-write that part of the code.
- Known as *refactoring* the code: changing the code such that it still produces the same results.
- Refactor difficult-to-document parts of the code to make them easier to understand.
- Not always possible – some things just are complicated with no simpler way to do them.

Document Design and Purpose

Embed documentation

Embed the documentation for a piece of software in that software. [1].

- The documentation of a program should be embedded within the code itself rather than in an external document.
- Makes it easier to read the documentation while inspecting the code.
- Increases the probability that the documentation gets updated accordingly when the code is changed.
- If we want external, nicely-formatted documentation as well, this can be generated from the embedded documentation. For example using Sphinx or Dexy.

Collaborate

Collaborate

Review

Use pre-merge code reviews. [1].

- Code review is having fellow programmers read through your code.
- Code reviews are a good way to assure the quality of the code you produce – catch bugs and improve readability.
- Also helps spread knowledge of how the code works to other team members.
- Pre-merge code review, as suggested here, means that the code should be reviewed before being checked into the VCS repository.
- The argument for this here is that if reviews do not have to be done before being checked in, they will probably not get reviewed at all.

Collaborate

Pair programming

Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems. [1].

- Pair programming is a fairly extreme form of code review.
- One person ("driver") writes the actual code while the other ("navigator") is watching, providing comments on the way the code is implemented.
- The navigator can maintain a better overview while the driver is "buried" in the details.
- This can be quite intrusive (similar to having a passenger constantly commenting on how you drive your car).
- Therefore the recommendation here is to use it when introducing someone new to the code or when dealing with particularly difficult problems.

Collaborate

"Issues"

Use an issue tracking tool. [1].

- This is a tool to keep track of bugs in the program, what needs to be programmed, what needs to be reviewed etc.
- A sort of to-do list kept together with the code.
- Helps provide all team members with a joint overview of the project.
- Usually integrated with the version control.
- Comes integrated with several free online VCS repository platforms such as Bitbucket and GitHub or in the form of locally installed tools such as Trac.

Summary

Summary

Today we have gone through a number of general recommendations for best practices in scientific software development:

- Scientific Software Development
- Write Programs for People
- Let the Computer Do the Work
- Make Incremental Changes
- Don't Repeat Yourself
- Plan for Mistakes
- Optimize Software Only after It Works
- Document Design and Purpose
- Collaborate