Stefano Cozzini

CRS – OGS Udine – 26 Novembre 2019

Introduction to scientific computing and programming..

What is scientific computing?

- from *Wikipedia.com*

  Scientific computing (or computational science) is the field of study concerned with

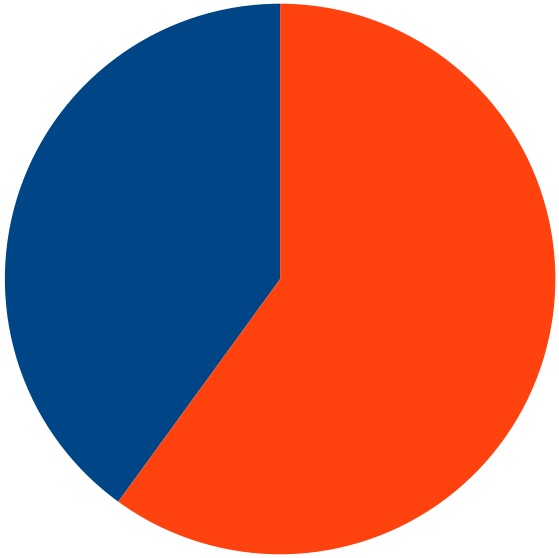
  - constructing mathematical models and (math/physics)
  - numerical solution techniques and (numerical analyis)
  - using computers (programming)

  to analyze and solve scientific and engineering problems.

Do you need this course?

How many participants

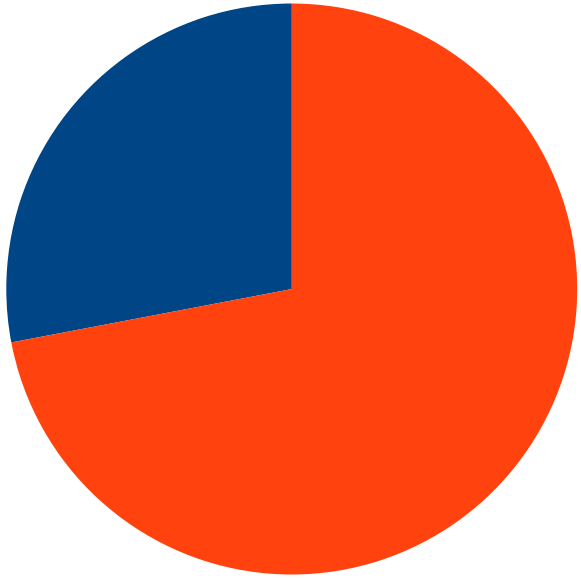
write shell scripts to analyze each new data set instead of running those analyses by hand?



Do you need this course?

How many of you

use version control to keep track of their work and collaborate with colleagues?

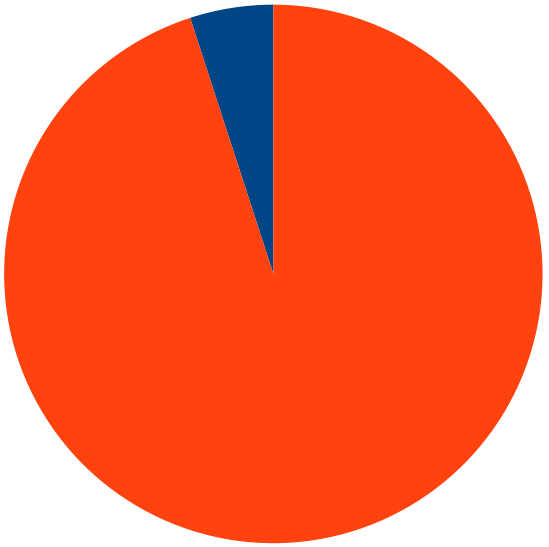


Do you need this course ?

How many

routinely break large problems into pieces small enough to be

- comprehensible,
- testable, and
- reusable?



And how many know those are the same things?

You can skip this course if

- You don't care if anyone else can ever use your software
- You are sure that people you've never met will be able to use and modify your software two years from now

A computationally competent scientist

A scientist is *computationally competent* if she can build, use, validate, and share software to:

- Manage and process data
- Tell if it's been processed correctly
- Find and fix problems when it hasn't been
- Keep track of what they've done
- Share work with others
- Do all of these things efficiently

Aims&Goals

- try to explain what software engineering is, how to apply to scientific computing
- give you some rules of thumb for figuring out how formal your software development process should be.
- Present some good practices for scientific software development

Software engineering

- The term “software engineering” was first used at a NATO conference in 1968
- chosen to get people thinking about how to build large, complex software systems
- Scientific codes are not based on software engineering techniques
- “software engineering” includes project management

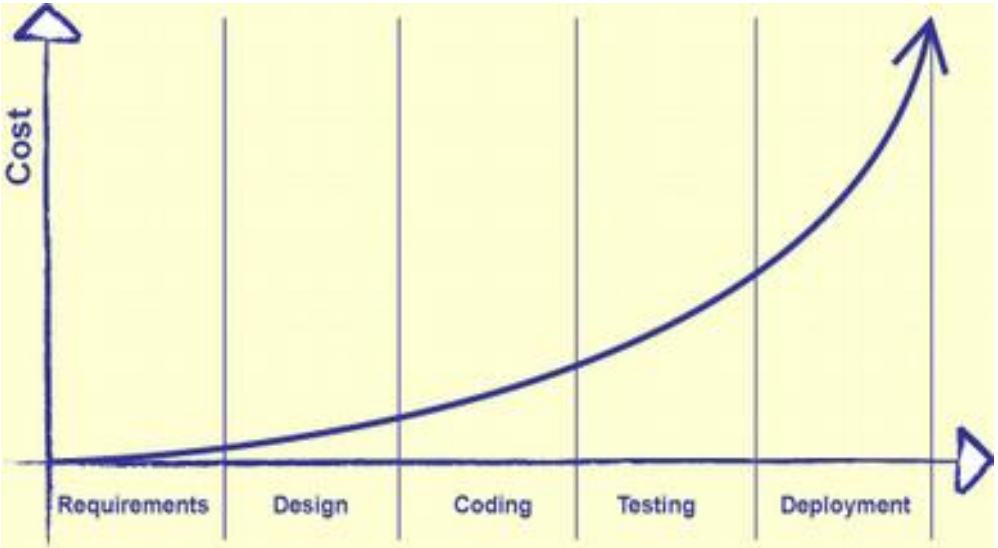
Sturdy vs. Agile

- Two camps currently dominate the debate about software development
- Sturdy: measure twice, cut once
  - Think through users' needs, design, and possible problems before starting to code

    Inspired by traditional engineering (in which late changes are very expensive)
- Agile: lots of small steps, with continuous testing and refactoring
  - "No battle plan ever survives contact with the enemy." (Helmuth von Moltke)
  - They refer to the sturdy camp as Big Design Up Front (BDUF)
  - Inspired by open source and 1990s web development
- Differences in practice are much less than the differences in rhetoric

Boehm's Curve

- Both methodologies are responses to Boehm's Curve



- Sturdy: aim carefully so that problems don't arise
  - The cheapest bug to fix is one that doesn't exist
- Agile: continuously correct course
  - No point trying to aim at a moving target, and real-world targets always move

What is the best for scientific code?

- Factors affecting choice:
  - How well the team understands the problem domain, tools, etc.
  - How stable the goals are
  - How many people are involved
- If you don't know enough to make long-term predictions with confidence, use agile
  - This means every genuinely new project ought to start agile
- If hundreds of people are involved, frequent course corrections will be painful
  - Though many agile advocates disagree

Ten simple rules for development of scientific software

https://www.software.ac.uk/blog/2016-09-26-ten-simple-rules-open-development-scientific-software

Rule 1: Don't Reinvent the Wheel

- Many fundamental scientific algorithms and methods **have already been implemented** in open-source libraries
- Evaluate them for your needs...
- Providing another solution to a problem, even if technologically novel, is only an accomplishment in engineering and rarely suitable for publication on its own.

Rule 2: code well

- Learn the basics of software development and trick
- Learn by practice: join an existing open-source project
  - Getting familiar with other people's code in this way is a great way to boost your experience and learn new techniques.

(more on this during the course)

Rule 3: Be Your Own User

- Youtr software should be useful for you: address important questions in a useful or novel way
- your software should be useful to other developers, and is not simply a demonstration of the solution.

Rule 4: Be Transparent

- open development allows many eyes to evaluate the code and recognize and fix any issues, which reduces the likelihood of serious errors in the final product.
- Use public repositories (github/gitlab/bitbucket) that greatly facilitate this kind of team development approach.

Rule 5: Be Simple

- simplicity is fundamental, since potential users will first evaluate how long it will take to install and get something out of your software against the time it will take them to find another way.
- Employ standard package or software installation models for as many platforms as possible.
- Use standard models for creating installable software packages,
- Try to support standard file formats and don't come up with new, custom formats.
- Spend time to create online documentation, sample data files, and test cases that will give others an easy start into your codebase.

Rule 6: Don't Be a Perfectionist

- “Release early, release often” (open-source mantra, and attributed to Linus Torvalds by Eric Raymond)
- your “customers” will quickly identify problems and new requirements,
- you will be then able to fix them rapidly

Rule 7: Nurture and Grow Your Community

- Form a team and communicate with the people who use your tool
- Make it easy for others to contribute ideas and act on feedback.
- Discuss with the community important changes
- Above all, avoid confusing your users—drastic differences between each release that introduce incompatibilities will win no friends.

Rule 8: Promote Your Project

- Spend time promoting your project:.
- Appearance matters, and a clean, well-organized website that will help your cause is not hard to achieve.
- Keep an eye out for ad hoc developer meetups and hackathons, where open-source coders get together to work on one, or many different projects.
- Promotion is hard work, but through it you will grow and strengthen your community

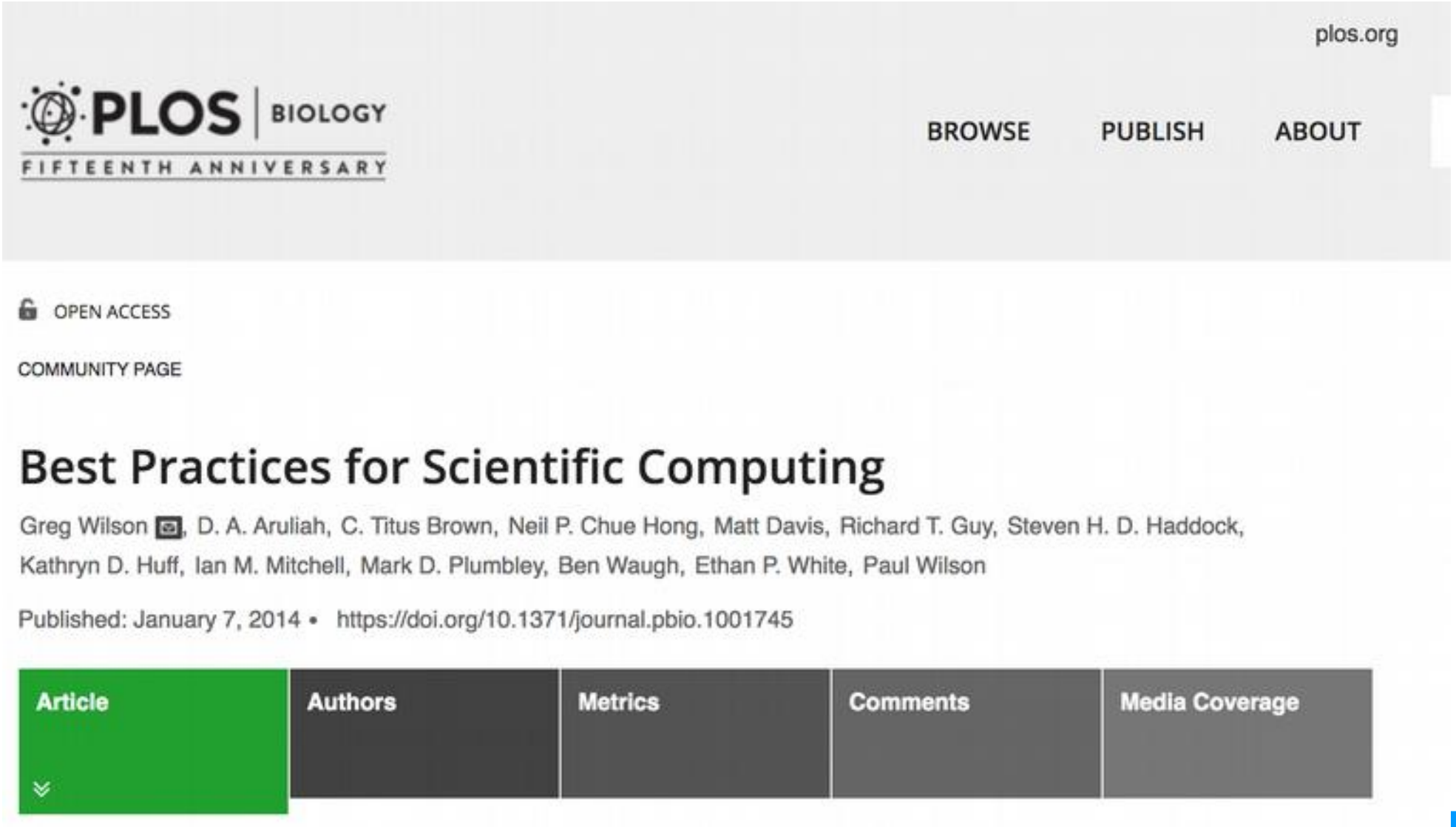
Rule 9: Find Sponsors

- No matter how large the community around your project and how efficiently it is developed and managed, some level of funding is essential.
- Scientific software can be successfully supported through grants, by writing applications to address new scientific problems through the development and use of software, or attaching development and upkeep of software as a deliverable on experimental grants.
- Open development directly addresses the section on sustainability in grant applications, but the emphasis here has to be on the community.
- Simply releasing code openly, without support and maintenance, will not ensure extended value; instead, you need to explain how you will actively foster your community of users and developers.

Rule 10: Science Counts

- the software you write is primarily a means to advance our research and, ultimately, achieve our scientific goals.
- Open-source development and maintenance is an intensely social process so it is even more important for scientists to keep an eye on the big picture, and stay true to our scientific goals.
- However open-source communities ensure persistence of projects by allowing project leadership to be shared and passed to other members.
- This offers you the opportunity to naturally progress to new challenges with the knowledge that the software you created will remain available and benefit others.

Best Practises in scientific computing..



plos.org

PLOS | BIOLOGY

FIFTEENTH ANNIVERSARY

BROWSE PUBLISH ABOUT

OPEN ACCESS

COMMUNITY PAGE

Best Practices for Scientific Computing

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, Paul Wilson

Published: January 7, 2014 • https://doi.org/10.1371/journal.pbio.1001745

Article | Authors | Metrics | Comments | Media Coverage

Best Practises in scientific computing..

- Write programs for people, not computers.
  - A program should not require its readers to hold more than a handful of facts in memory at once.
  - Make names consistent, distinctive, and meaningful.
  - Place a brief explanatory comment at the start of every program.
  - Make code style and formatting consistent.
- Let the computer do the work.
  - Make the computer repeat tasks.
  - Save recent commands in a file for re-use.
  - Use a build tool to automate workflows.
- Make incremental changes.
  - Work in small steps with frequent feedback and course correction.
  - Use a version control system.
  - Put everything that has been created manually in version control.
- Don't repeat yourself (or others).
  - Every piece of data must have a single authoritative representation in the system.
  - Modularize code rather than copying and pasting.
  - Re-use code instead of rewriting it.

Best Practises in scientific computing..

- Plan for mistakes.
  - Add assertions to programs to check their operation.
  - Use an off-the-shelf unit testing library.
  - Turn bugs into test cases.
  - Use a symbolic debugger.
- Optimize software only after it works correctly.
  - Use a profiler to identify bottlenecks.
  - Write code in the highest-level language possible.
- Document design and purpose, not mechanics.
  - Document interfaces and reasons, not implementations.
  - Refactor code in preference to explaining how it works.
  - Embed the documentation for a piece of software in that software.
- Collaborate.
  - Use pre-merge code reviews.
  - Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
  - Use an issue tracking tool.

The essential software tools

- Compiler
- Debugger
- Use Version Control Software (git)
- Use Automated Build Tools
  - Makefile / Autotools/ Cmake etc..
- Use a Testing Framework
- Use documentation tools (doxygen etc..)

Science Code Manifesto

Software is a cornerstone of science. Without software, twenty-first century science would be impossible. Without better software, science cannot progress.

But the culture and institutions of science have not yet adjusted to this reality. We need to reform them to address this challenge, by adopting these five principles:

- Code
  - All source code written specifically to process data for a published paper must be available to the reviewers and readers of the paper.
- Copyright
  - The copyright ownership and license of any released source code must be clearly stated.
- Citation
  - Researchers who use or adapt science source code in their research must credit the code's creators in resulting publications.
- Credit
  - Software contributions must be included in systems of scientific assessment, credit, and recognition.
- Curation
  - Source code must remain available, linked to related materials, for the useful lifetime of the publication.

Summary

- There is no "best process"
- Choose one based on:
  - How well you understand the problem and the technology
  - How stable the requirements are
  - How big (or distributed) the team is
- Most important thing is that everyone is playing by the same rules...
- ...and that you adjust the process to reflect reality, rather than trying to do the opposite