Motivation and Overview of Best Practices in HPC Software Development

Rinku Gupta (she/her)
Argonne National Laboratory

Better Scientific Software tutorial @ Improving Scientific Software 2022

Contributors: David E. Bernholdt (ORNL), Anshu Dubey (ANL), Patricia A. Grubel (LANL), Rinku K. Gupta (ANL), Katherine M. Riley (ANL)

LA-UR-21-29819

License, Citation and Acknowledgements

License and Citation

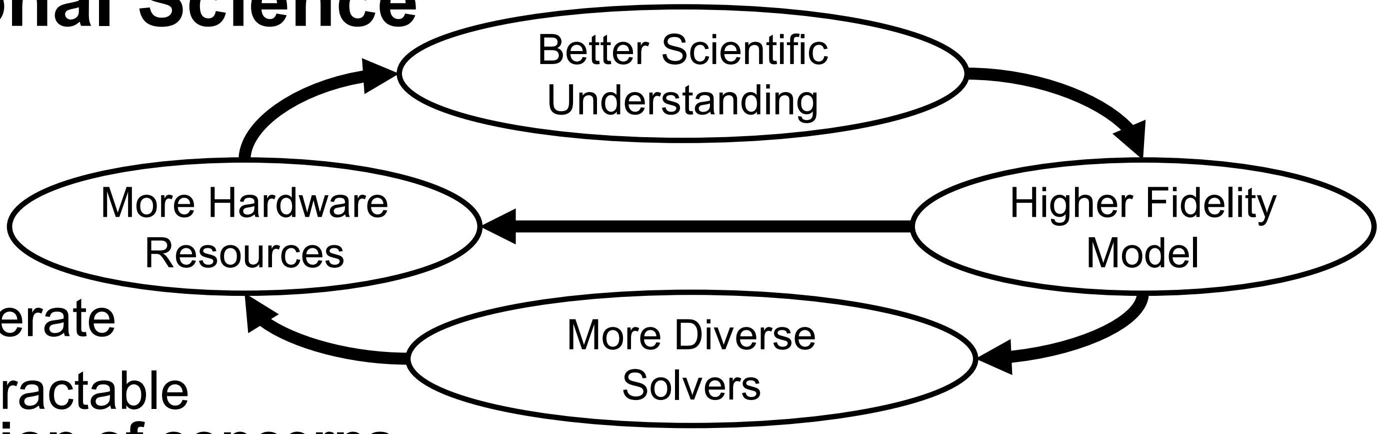
- **The requested citation the overall tutorial is: David E. Bernholdt, Patricia A. Grubel, Rinku K. Gupta, and David M. Rogers, Better Scientific Software tutorial, in Improving Scientific Software conference, online, 2022. DOI: 10.6084/m9.figshare.19416767**
- Individual modules may be cited as *Speaker, Module Title*, in Better Scientific Software tutorial…

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Science through computing is,
at best,
as credible as the software that produces it!

The Success of Computational Science Creates the Challenges of Computational Science



- Positive feedback loop
  - More complex codes, simulations and analysis
  - More moving parts that need to interoperate
  - Variety of expertise needed – the only tractable development model is through **separation of concerns**
  - **It is more difficult to work on the same software in different roles without a software engineering process**
- Onset of higher platform heterogeneity
  - Requirements are unfolding, not known *a priori*
  - **The only safeguard is investing in flexible design and robust software engineering process**

Supercomputers change fast
Especially now!

IDEAS productivity
ECP EXASCALE COMPUTING PROJECT

Challenges Developing Scientific Applications Today

Technical

- All parts of the model and software system can be under research
- Requirements change throughout the lifecycle as knowledge grows
- Verification complicated by floating point representation
- Real world is messy, so is the software
- Increasing architectural diversity

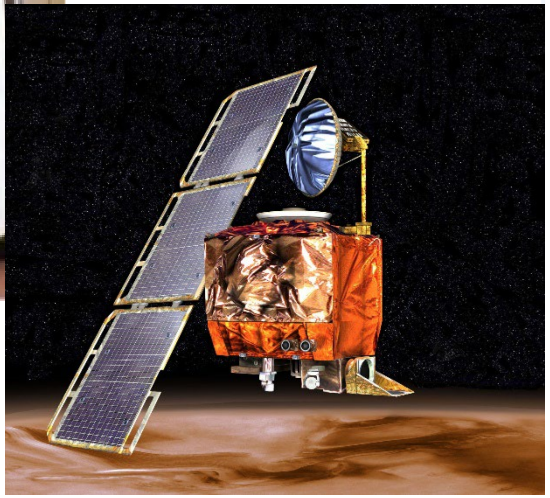
Sociological

- Competing priorities and incentives
  - Sponsors often care more about scientific publications than software per se
  - Balancing development and maintenance with research
- Limited resources
- Need for interdisciplinary interactions
  - Many different kinds of expertise to be successful

High-Consequence Software-Related Scientific Failures

Therac-25 (1985-1987)

- Computer-controlled radiation therapy system
- **Poor software design, development and testing practices** allowed flaws that let to at least six cases of substantial radiation overdoses, three fatal



Mars Climate Orbiter (1999)

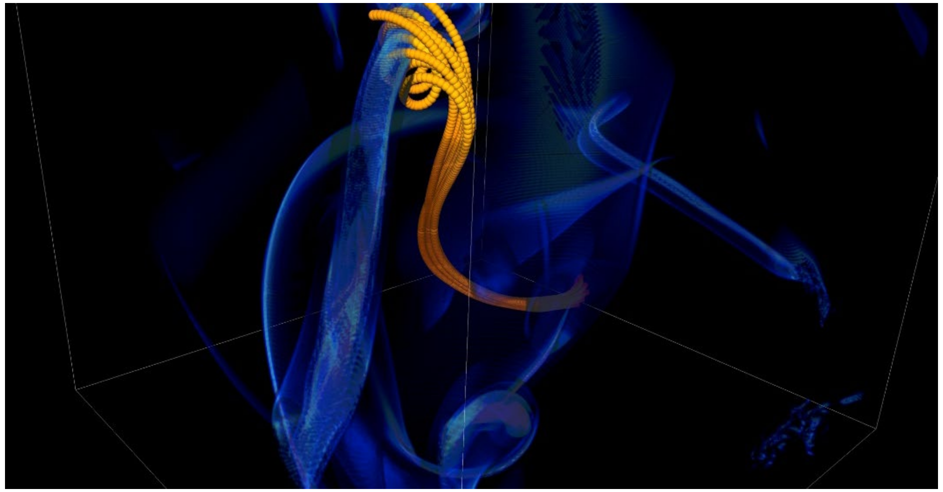
- Incorrect trajectory adjustment caused loss of the orbiter as it was supposed to enter Martian orbit
- Discrepancy in the units used in two different software components
- One component **didn’t follow specifications**
- **Inadequate testing** at the interface
- Concerns raised earlier in the mission were ignored because they **weren’t properly documented**

Just two of many examples

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

More Subtle Impacts on Scientific Productivity



- In 2005, the FLASH astrophysics team was offered a unique opportunity to access one of the biggest machines in the world at that time (BG/L) for a dedicated run
- Short notice to prepare
  - **< 1month to get ready for 1.5 week run**
- Quick and dirty development of particle capability in code
- Error in tracking particles resulted in duplicated tags from round-off
- Had to develop post-processing tools to correctly identify trajectories
  - **6 months to process results**

FLASH had a software process in place. It was tested regularly. This was one instance when the full process could not be applied because of time constraints.

Technical Debt

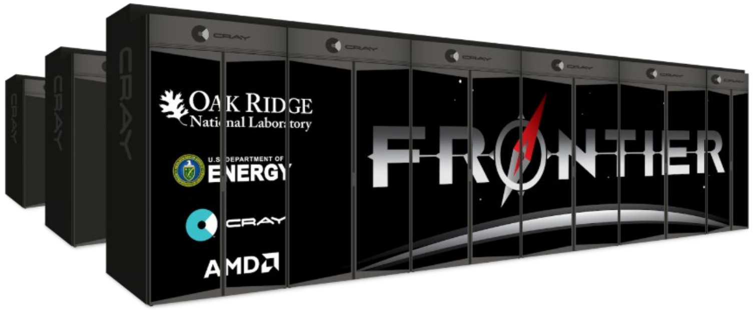
> The implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.
>
> -- Wikipedia

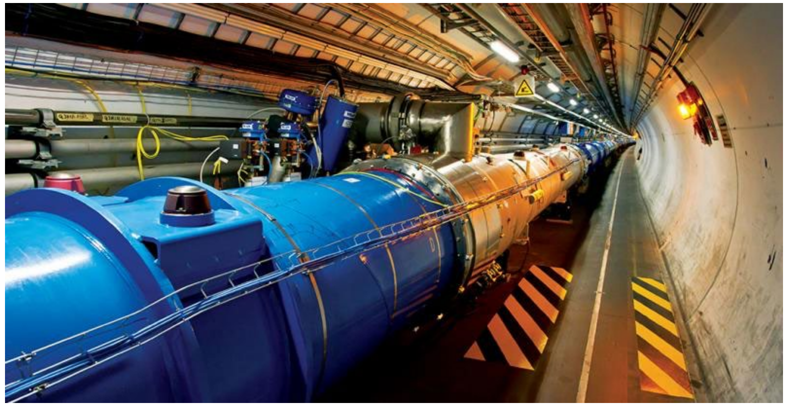
Like monetary debt, the more you accumulate, the harder it is to pay off

- Increases cost of maintenance
- Parts of software may become unusable over time
- Inadequately verified software produces questionable results
- Increases ramp-up time for new developers
- **Overall, reduces software and science productivity**

Scientific Facilities Provide Valuable Resources

- Major supercomputers often cost O($100M)
- All cost millions more to operate, annually
- Significant allocations on large supercomputers can be worth millions
- Even if you don’t pay the $ you have to spend the time and effort to get the allocation
- **Sponsors’ concern: Are you being a good steward of the resources?**
- **Your concern: Are you getting the most science possible out of your work (aka scientific productivity)?**





Good scientific process requires good software practices

Good software practices increase software sustainability

Good software practices increase scientific productivity

Software sustainability increases scientific productivity

So, What Are Good Software Practices?

- There is no fixed, universally agreed set of best practices for scientific software
  - Specifics of what’s appropriate will depend on the software, how it is used, and the team
- Let’s look at a few recommendations from different perspectives…

Example 1: Best Practices for Scientific Computing (1/2)

Wilson, et al., (2014) https://doi.org/10.1371/journal.pbio.1001745

1. Write programs for people, not computers.
   a. A program should not require its readers to hold more than a handful of facts in memory at once.
   b. Make names consistent, distinctive, and meaningful.
   c. Make code style and formatting consistent.
2. Let the computer do the work.
   a. Make the computer repeat tasks.
   b. Save recent commands in a file for re-use.
   c. Use a build tool to automate workflows.
3. Make incremental changes.
   a. Work in small steps with frequent feedback and course correction.
   b. Use a version control system.
   c. Put everything that has been created manually in version control.
4. Don't repeat yourself (or others).
   a. Every piece of data must have a single authoritative representation in the system.
   b. Modularize code rather than copying and pasting.
   c. Re-use code instead of rewriting it.
5. Plan for mistakes.
   a. Add assertions to programs to check their operation.
   b. Use an off-the-shelf unit testing library.
   c. Turn bugs into test cases.
   d. Use a symbolic debugger.

Example 1: Best Practices for Scientific Computing (2/2)

Wilson, et al., (2014) https://doi.org/10.1371/journal.pbio.1001745

6. Optimize software only after it works correctly.
   a. Use a profiler to identify bottlenecks.
   b. Write code in the highest-level language possible.
7. Document design and purpose, not mechanics.
   a. Document interfaces and reasons, not implementations.
   b. Refactor code in preference to explaining how it works.
   c. Embed the documentation for a piece of software in that software.
8. Collaborate.
   a. Use pre-merge code reviews.
   b. Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
   c. Use an issue tracking tool.

Example 2: Good Enough Practices in Scientific Computing (1/2)

Wilson, et al., (2017) https://doi.org/10.1371/journal.pcbi.1005510

1. Data management
   a. Save the raw data.
   b. Ensure that raw data are backed up in more than one location.
   c. Create the data you wish to see in the world.
   d. Create analysis-friendly data.
   e. Record all the steps used to process data.
   f. Anticipate the need to use multiple tables, and use a unique identifier for every record.
   g. Submit data to a reputable DOI-issuing repository so that others can access and cite it.

6. Manuscripts (out of order to save space)
   a. Write manuscripts using online tools with rich formatting, change tracking, and reference management.
   b. Write the manuscript in a plain text format that permits version control.

2. Software
   a. Place a brief explanatory comment at the start of every program.
   b. Decompose programs into functions.
   c. Be ruthless about eliminating duplication.
   d. Always search for well-maintained software libraries that do what you need.
   e. Test libraries before relying on them.
   f. Give functions and variables meaningful names.
   g. Make dependencies and requirements explicit.
   h. Do not comment and uncomment sections of code to control a program's behavior.
   i. Provide a simple example or test data set.
   j. Submit code to a reputable DOI-issuing repository.

Example 2: Good Enough Practices in Scientific Computing (2/2)

Wilson, et al., (2017) https://doi.org/10.1371/journal.pcbi.1005510

3. Collaboration
   a. Create an overview of your project.
   b. Create a shared "to-do" list for the project.
   c. Decide on communication strategies.
   d. Make the license explicit.
   e. Make the project citable.

4. Project organization
   a. Put each project in its own directory, which is named after the project.
   b. Put text documents associated with the project in the doc directory.
   c. Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory.
   d. Put project source code in the src directory.
   e. Put external scripts or compiled programs in the bin directory.
   f. Name all files to reflect their content or function.

5. Keeping track of changes
   a. Back up (almost) everything created by a human being as soon as it is created.
   b. Keep changes small.
   c. Share changes frequently.
   d. Create, maintain, and use a checklist for saving and sharing changes to the project.
   e. Store each project in a folder that is mirrored off the researcher's working machine.
   f. Add a file called CHANGELOG.txt to the project's docs subfolder.
   g. Copy the entire project whenever a significant change has been made.
   h. Use a version control system.

Example 3: Linux Foundation Core Infrastructure Initiative (CII) Best Practices Badging Program

https://bestpractices.coreinfrastructure.org/en

- Not specifically intended for scientific software
- Three levels
  - **Passing** focuses on best practices that well-run FLOSS projects typically already follow. Getting the passing badge is an achievement; at any one time only about 10% of projects pursuing a badge achieve the passing level.
  - **Silver** is a more stringent set of criteria than passing but is expected to be achievable by small and single-organization projects.
  - **Gold** is even more stringent than silver and includes criteria that are not achievable by small or single-organization projects.
- Combination of MUST and SHOULD criteria

CII Best Practices Criteria Summary

- Basics
  - Basic project website content (P, S)
  - FLOSS license (P)
  - Documentation (P, S)
  - Project oversight (S, G)
  - Accessibility and internationalization (S)
- Change control
  - Public version controlled source repo. (P, G)
  - Unique version numbering (P)
  - Release notes (P)
  - Previous versions (S)
- Reporting
  - Bug-reporting process (P, S)
  - Vulnerability reporting process (P, S)

(P, S, G) denotes additional criteria required at passing, silver, or gold certification levels

Each topic area listed will have one or more specific criteria

- Quality
  - Working build system (P, S, G)
  - Automated test suite (P, S, G)
  - New functionality testing (P, S)
  - Warning flags (P, S)
  - Coding standards (S, G)
  - Installation system (S)
  - Externally-maintained components (S)
- Security
  - Secure development knowledge (P, S)
  - Use basic good crypto. practices (P, S, G)
  - Secured delivery against MITM attacks (P, G)
  - Publicly known vulnerabilities fixed (P)
  - Secure release (S)
- Analysis
  - Static code analysis (P, S)
  - Dynamic code analysis (P, S, G)

Software Engineering Advice Often Needs Adaptation for Scientific Software

- The CII Best Practices are a good example of software engineering advice “in the wild”
- Experiences reported in the wild often don’t consider the special nature of scientific software
- But that doesn’t mean we should ignore all of the software engineering experience
  - Many useful concepts, approaches, and tools we can just adopt
- Some approaches may need to be adapted to work for scientific software
  - Find out how colleagues have addressed the challenges you’re facing
    - Probably you will find multiple ways
  - In the end, some approaches may not work well
- Don’t be afraid to experiment with adaptations
  - Consider using the PSIP process (coming up)

How Much (Time, Effort) Should I Spend on Software Engineering?

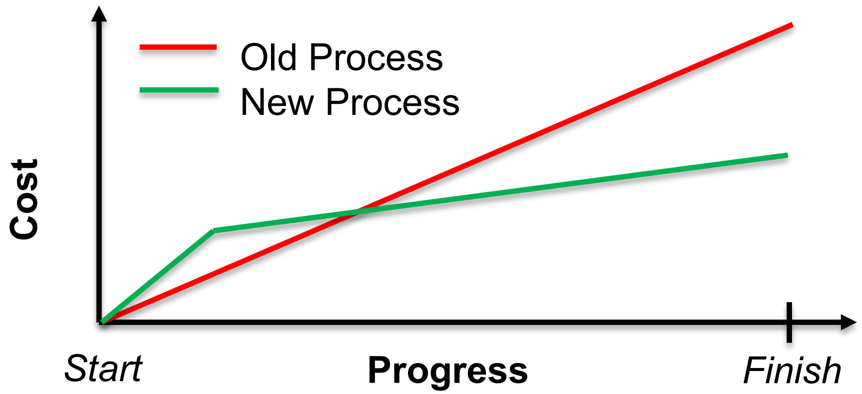
Your project should include “just enough” software engineering so that you can meet your short-term and longer-term scientific goals effectively

Continual, Incremental Software Process Improvement

Target: your project should include “just enough” software engineering so that you can meet your short-term and longer-term scientific goals effectively

1. Identify your team’s “pain points” in your software development processes
   – Help: RateYourProject assessment tool: https://rateyourproject.org/
2. Set a goal for something to improve
   – Target processes and behaviors, not just tasks
   – Pick something that you can address in a few months that will give you a noticeable benefit
3. Agree on a plan to address it, identify markers of progress and what is “done”
   – Write them down
   – Help: Progress tracking card examples: https://bssw-psip.github.io/ptc-catalog/catalog
4. Work your plan, track your progress
5. When you are done, celebrate…

…then pick a new pain point to address



The new process costs something to implement, but it pays off over time

Productivity and Sustainability Improvement Planning
https://bssw.io/psip

A goal of BSSw.io is to provide resources for improving your software processes. If you find useful resources that aren’t on BSSw.io, consider contributing. Its easy and quick.

About Today's Tutorial

- There are many useful topics that could help you improve your scientific software development process
- We're going to focus on a few where the software engineering advice in the wild typically doesn't address scientific software
  - Project management
  - Collaboration around software development
  - Designing software for flexibility and extensibility
  - Testing strategies for complex software systems
  - Systematic refactoring of large, complex software systems
  - Continuous integration testing
  - Reproducibility

Agenda (Morning)

The agenda is also available on the tutorial web page. Visit https://bssw-tutorial.github.io and click on the link for today's tutorial

Time (MDT)	Module	Title	Presenter
9:00 AM	0	Introduction and Setup	David E. Bernholdt (ORNL)
9:05 AM	1	Motivation and Overview of Best Practices in HPC Software Development	Rinku K. Gupta (ANL)
9:15 AM	2	Agile Methodologies	Patricia A. Grubel (LANL)
9:45 AM	3	Git Workflows	Patricia A. Grubel (LANL)
10:00 AM		*Break*	
10:20 AM	4	Software Testing Introduction	Rinku K. Gupta (ANL)
10:40 AM	5	Scientific Software Design	David E. Bernholdt (ORNL)
11:00 AM	6	Testing Complex Software	Rinku K. Gupta (ANL)
11:15 AM	7	Refactoring Scientific Software	David M. Rogers (ORNL)
11:40 AM	8	Improving Reproducibility Through Better Software Practices	David M. Rogers (ORNL)
11:55 AM	9	Summary	David M. Rogers (ORNL)
11:20 AM		*Adjourn presentation portion*	

productivity

COMPUTING PROJECT

Agenda (Afternoon)

The agenda is also available on the tutorial web page. Visit **https://bssw-tutorial.github.io** and click on the link for today's tutorial

Time (MDT)	Module	Title	Presenter
1:20 PM		Hands-on & Discussion (optional)	All
2:20 PM		*Break*	
2:40 PM		Hands-on & Discussion (optional)	All
3:40 PM		*Adjourn*	