Crafting Quality Research Software and Navigating Publication in Software Journals

August 10, 2023

Rene Gassmoeller, Lorraine Hwang, Mohamed Gouiza

Computational Infrastructure for Geodynamics

Take the 1-min
pre-workshop quiz

menti.com
Code:
5846 8672

National
Science
Foundation

dynamo · seismology · computational science · short term crustal dynamics · CIG · mantle convection · melt and volatiles · long term tectonics · education

COMPUTATIONAL
INFRASTRUCTURE
for GEODYNAMICS

Introduction and Logistics

- 3 Blocks, 90 min each, 30 min breaks
  - Session 1 - Fundamentals of Research Software Development
  - Session 2 - Documentation and Reproducibility
  - Session 3 - Navigating Publication in Software Journals
- We want interactive discussions
  - Please use your real names in Zoom configuration
  - Use chat to interact with us or other participants
  - Unmute if you have a question that may be of broader interest
  - Keep your camera on, if you feel comfortable with it
- This is a CIG workshop, please follow our Code of Conduct:
  - We want an inclusive environment, where everyone can express their ideas free of harassment or discrimination
- Follow the slides and other materials on: http://bit.ly/2023-cig-joss

Who are we?

COMPUTATIONAL
INFRASTRUCTURE
for GEODYNAMICS

- Computational Infrastructure for Geodynamics (geodynamics.org):

  "NSF-funded community-driven organization advancing Earth science by providing the infrastructure for the development and dissemination of software for geophysics and related fields."

- Rene Gassmoeller, University of Florida, Technical Lead

- Lorraine Hwang, UC Davis, co-Director

- Mohamed Gouiza, UC Davis, Project Scientist

Who are you?

- Let's take a look at the pre-assessment quiz we sent out

Starting remarks

- Crafting good enough research software is no magic,
  there are published guidelines and checklists
- Many steps to create good research software are purely mechanical, this will not solve the inner design of your software, but improve a lot of the "scaffolding" around its core functionality
- These mechanical steps take some effort, but pay off over time

Best Practices

- Useful Materials:
  - The Computational Infrastructure for Geodynamics maintains software best practices https://geodynamics.org/software/software-bp
    And a contributing checklist for CIG software
    https://github.com/geodynamics/best_practices/blob/main/ContributingChecklist.md
  - CIG also maintains a repository that can be used as a template to create/improve projects: https://github.com/geodynamics/software_template
  - The Journal of Open Source Software maintains submission instructions https://joss.readthedocs.io/en/latest/submitting.html
    And a reviewer checklist
    https://joss.readthedocs.io/en/latest/review_checklist.html
- If your software fulfills CIG or JOSS criteria, it fulfills most criteria for both

Publication criteria for software

- We will present the CIG and JOSS criteria
  - Licensing
  - Version Control
  - Contribution + Authorship
  - Installation
  - Software design
  - Documentation
  - Testing
  - Reproducibility
- Discuss the meaning, justification, and implementation of each
- Provide further resources for a deeper dive on your own

Licensing

CIG Best practices	**Minimum**	**Standard**	**Target**
Licensing	Open source	Same as Minimum	Same as Minimum

- CIG best practices:

  Include a 'LICENSE' file: Add a plain text copy of your OSI approved software license to your repository: https://opensource.org/licenses

- JOSS best practices:

  License: Does the repository contain a plain-text LICENSE file with the contents of an OSI approved software license?

Licensing

- Open source software is a clearly defined concept: https://opensource.org/osd/
- Make your software open source by applying an OSI approved license: https://choosealicense.com/
- You retain the authorship + copyright, but grant a license to use and modify
- Most common OSI licenses fall in two categories:
  Permissive (e.g. MIT license) and copyleft (e.g. GPL license)
- Permissive licenses allow modifications to be distributed under different licenses (e.g. closed source commercial), while copyleft licenses require distribution as open-source
- Permissive licenses can be beneficial if you imagine commercial applications, or if you simply want to put the least restrictions on the use of your software

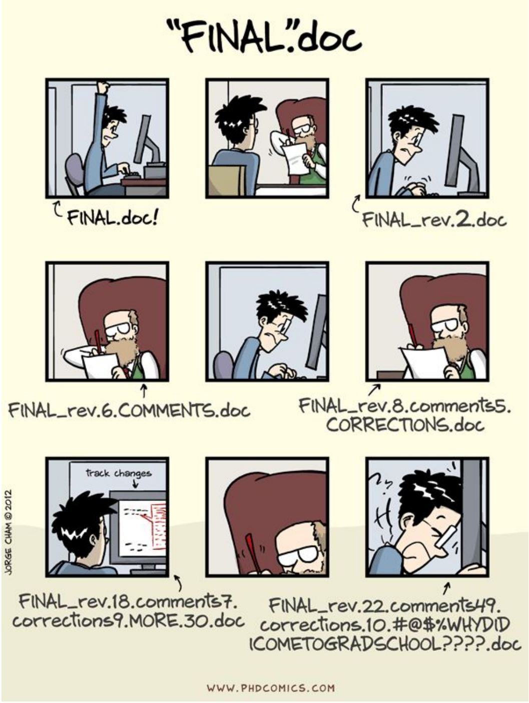
Version control

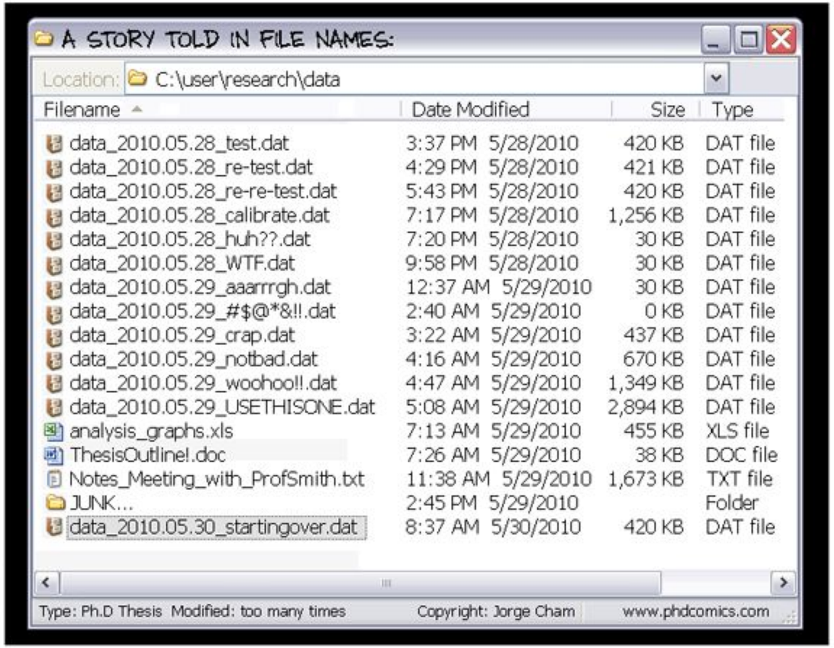
CIG Best practices	**Minimum**	**Standard**	**Target**
Version Control	All source in version control.	Differentiation between maintenance and new development.	(a) New features added in separate branches. (b) Stable development branches for rapid release of new features.

- JOSS:
  Repository: Is the source code for this software available at the repository url?
  The software must be hosted at a location where users can open issues and propose code changes without manual approval of (or payment for) accounts.

Version control

- Version Control Systems were created to prevent this:





A STORY TOLD IN FILE NAMES:

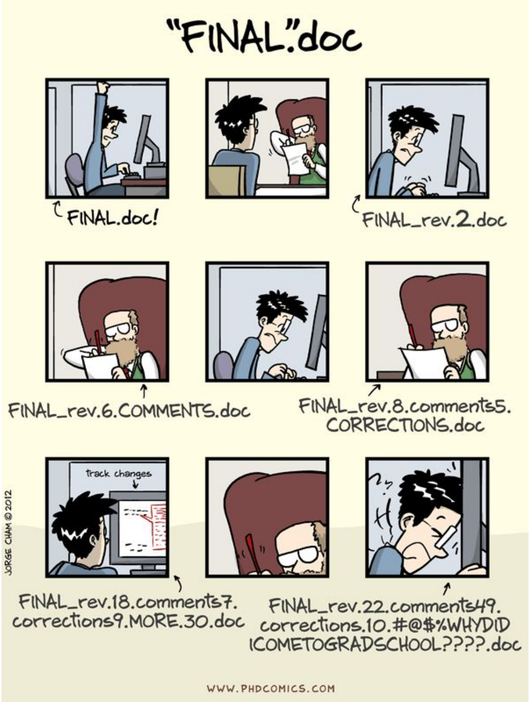
Location: C:\user\research\data

Filename	Date Modified	Size	Type
data_2010.05.28_test.dat	3:37 PM 5/28/2010	420 KB	DAT file
data_2010.05.28_re-test.dat	4:29 PM 5/28/2010	421 KB	DAT file
data_2010.05.28_re-re-test.dat	5:43 PM 5/28/2010	420 KB	DAT file
data_2010.05.28_calibrate.dat	7:17 PM 5/28/2010	1,256 KB	DAT file
data_2010.05.28_huh??.dat	7:20 PM 5/28/2010	30 KB	DAT file
data_2010.05.28_WTF.dat	9:58 PM 5/28/2010	30 KB	DAT file
data_2010.05.29_aaarrrgh.dat	12:37 AM 5/29/2010	30 KB	DAT file
data_2010.05.29_#$@*&!!.dat	2:40 AM 5/29/2010	0 KB	DAT file
data_2010.05.29_crap.dat	3:22 AM 5/29/2010	437 KB	DAT file
data_2010.05.29_notbad.dat	4:16 AM 5/29/2010	670 KB	DAT file
data_2010.05.29_woohoo!!.dat	4:47 AM 5/29/2010	1,349 KB	DAT file
data_2010.05.29_USETHISONE.dat	5:08 AM 5/29/2010	2,894 KB	DAT file
analysis_graphs.xls	7:13 AM 5/29/2010	455 KB	XLS file
ThesisOutline!.doc	7:26 AM 5/29/2010	38 KB	DOC file
Notes_Meeting_with_ProfSmith.txt	11:38 AM 5/29/2010	1,673 KB	TXT file
JUNK...	2:45 PM 5/29/2010		Folder
data_2010.05.30_startingover.dat	8:37 AM 5/30/2010	420 KB	DAT file

Type: Ph.D Thesis Modified: too many times Copyright: Jorge Cham www.phdcomics.com

Version control

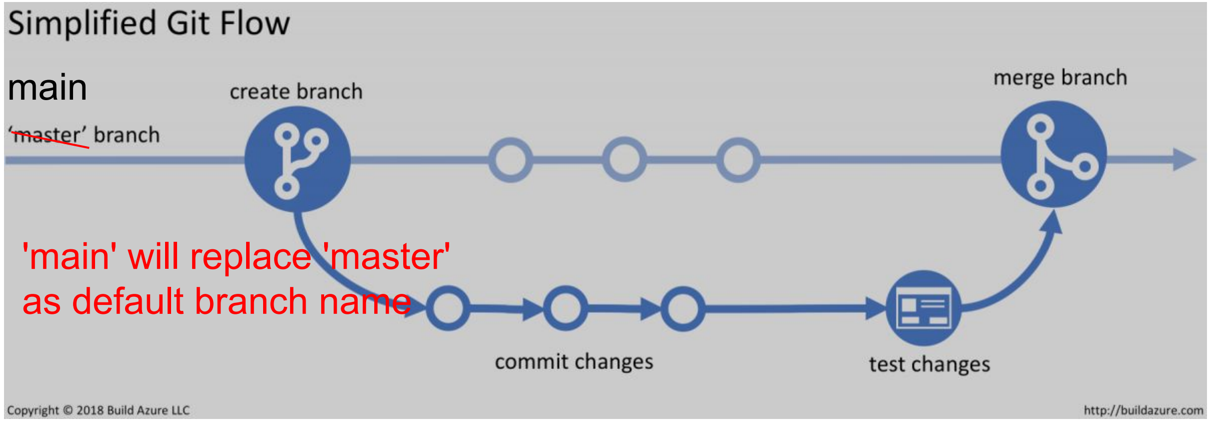
- Version Control Systems were created to prevent this:



- A VCS is like an unlimited undo/redo
- **git** is the de-facto standard version control system for software (other: svn, mercurial)
- You (and everyone else) are likely using git
- Git can help to answer:
  - Who included this code and why?
  - Was this bug already in the code I presented at a conference last year?
  - How can we merge these changes into that other version?
- An excellent and complete git tutorial (~8hrs) is provided by software carpentry: https://swcarpentry.github.io/git-novice/
- If you just need a summary of git commands, use this cheat sheet: https://education.github.com/git-cheat-sheet-education.pdf

Git best practices

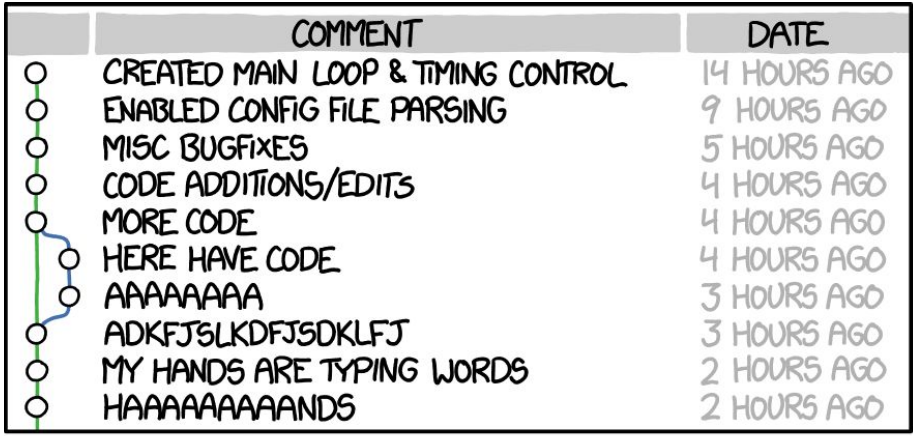
1. Commit logical (atomic) changesets
2. Commit Early, Commit Often
3. **Write Reasonable Commit Messages**
4. Don’t Commit Auto-generated Files
5. Don’t Merge Half-Done Work
6. Test Before You Merge
7. Use Branches
8. Agree on a Workflow



'main' will replace 'master' as default branch name

https://ruleoftech.com/2019/best-practices-for-version-control-in-8-steps

Reasonable commit messages



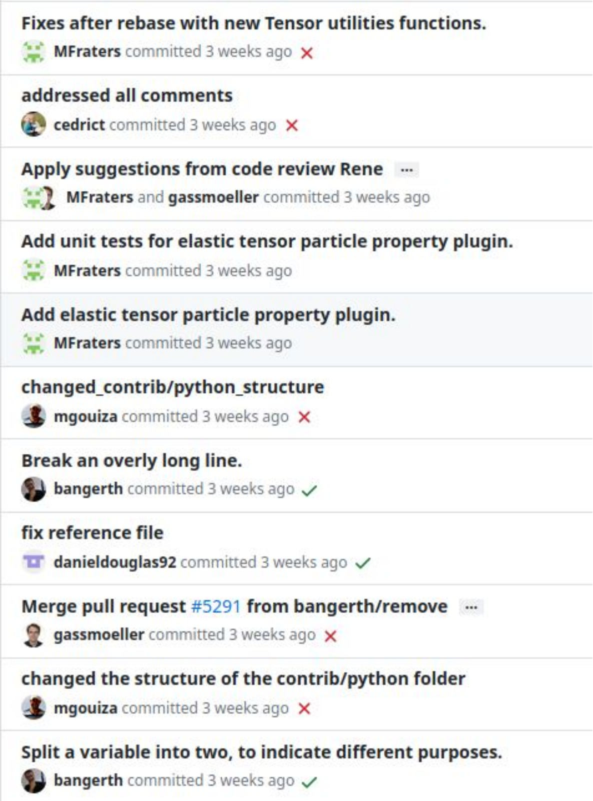
	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. **Use the imperative mood**
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

https://cbea.ms/git-commit/

Reasonable commit messages



Fixes after rebase with new Tensor utilities functions.
MFraters committed 3 weeks ago ✗

addressed all comments
cedrict committed 3 weeks ago ✗

Apply suggestions from code review Rene ...
MFraters and gassmoeller committed 3 weeks ago

Add unit tests for elastic tensor particle property plugin.
MFraters committed 3 weeks ago

Add elastic tensor particle property plugin.
MFraters committed 3 weeks ago

changed_contrib/python_structure
mgouiza committed 3 weeks ago ✗

Break an overly long line.
bangerth committed 3 weeks ago ✓

fix reference file
danieldouglas92 committed 3 weeks ago ✓

Merge pull request #5291 from bangerth/remove ...
gassmoeller committed 3 weeks ago ✗

changed the structure of the contrib/python folder
mgouiza committed 3 weeks ago ✗

Split a variable into two, to indicate different purposes.
bangerth committed 3 weeks ago ✓

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. **Use the imperative mood**
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Publicly hosted: Online software repositories

- There are commercial and open-source online hosted git repositories and collaboration platforms (GitHub, Bitbucket, GitLab)
- GitHub is the largest platform for open-source and commercial software development today (>50 million users) and hosts most of CIG's software repositories in the organization name **geodynamics**
- CIG and JOSS are platform agnostic, use whichever platform you prefer (as long as it is open and accessible without manual approval or cost for users)
- See the software carpentry tutorial: https://swcarpentry.github.io/git-novice/07-github.html

Online software repository recommendations

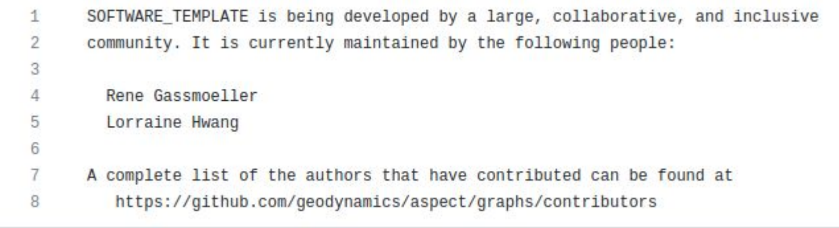
- Protect your main branches, even against maintainers
- Only merge changes via pull requests, no direct pushes
- Every merge requires a review by another maintainer (if possible)
- Preparing to grow your software: Provide write access to other developers generously, but judiciously
- Make use of automation features (discussed later)

Contribution + Authorship

- **JOSS Contribution and authorship:** Has the submitting author made major contributions to the software? Does the full list of paper authors seem appropriate and complete?
- CIG best practices:
  a. Optional: Include an ‘AUTHORS’ file
     List the authors of your code. This may or may not be the same as those listed in the citation and/or the list of project committers.
  b. Optional: Include an ‘ACKNOWLEDGE’ file
     Acknowledge funding agencies or other sources of support to credit.
  c. Alternatively a more modern approach is CodeMeta, e.g. by including a codemeta.json file all citation information is standardized and machine readable

- Share ownership generously
- Openly acknowledge and credit contributors just like you would in publications
- Credit makes it more likely to gain new contributors

Example AUTHORS file:



```
SOFTWARE_TEMPLATE is being developed by a large, collaborative, and inclusive
community. It is currently maintained by the following people:

  Rene Gassmoeller
  Lorraine Hwang

A complete list of the authors that have contributed can be found at
   https://github.com/geodynamics/aspect/graphs/contributors
```

Installation

CIG Best practices	**Minimum**	**Standard**	**Target**
Portability, configuration, and building	(a) Codes builds on Unix-like machines with free tools. (b) Portable build system.	Minimum + (a) Dependency checking. (b) Automation and portability of configuration and building. (c) Each simulation outputs all configuration and build options for reproducibility.	Standard + (a) Selection of compilers, optimization, build flags during configuration without modifying files under version control. (b) Multiple builds using same source. (c) Allows installation to a central location.

JOSS:
Does installation proceed as outlined in the documentation?
Is there a clearly-stated list of dependencies? Ideally these should be handled with an automated package management solution.

Installation Best Practices

- A portable build system, e.g.,
  Python: https://packaging.python.org/en/latest/tutorials/packaging-projects/
  CMake: https://cmake.org/
  GNU Autoconf: https://www.gnu.org/software/autoconf/
- Example CIG software installation instructions:
  - Python: BurnMan
  - C/C++: CitcomS (GNU Autoconf), ASPECT (CMake)
  - Fortran: Rayleigh
- Hand-written Makefiles are discouraged, because they limit operating system and compiler choice!
- We will discuss dependency management as part of the reproducibility section in session 2

A 2-page CMake tutorial

- CMake is an open-source, operating-system independent, powerful build system that is widely used and available
- It stores its configuration in a file called **CMakeLists.txt**
- CMakeLists.txt contains information to **compile, install, and package** a code
- Its commands are **case-insensitive**
- A minimal CMakeLists.txt could look as simple as this:

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello CXX)
set(CMAKE_CXX_STANDARD 14)
add_executable(cmake_hello main.cpp)
```

```
# Sets minimum cmake version
# Names the project and enables the language
# minimum language standard
# Create executable depending on all listed source files
```

A 2-page CMake tutorial

- CMake has a tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/index.html
- This for example discusses how to include external libraries, build libraries, install, test, and package software, inject configuration/system information into your source code, e.g. to output library versions, configuration options, …
- Example CMakeLists configuration files:
  - ASPECT: https://github.com/geodynamics/aspect/blob/main/CMakeLists.txt

Software Design and Coding Practices

CIG Best practices	**Minimum**	**Standard**	**Target**
Coding		(a) User-friendly specification of parameters at run time. (b) Development plan, updated annually. (c) Comments in code with purpose of each function. (d) Users can add features or alternative implementations without modifying main branch. (e) User errors generate message that helps user correct the problem.	Standard + (a) Functionality implemented as a library rather than an application. (b) Output of provenance information. (c) Parallel access to inputs/outputs. (d) Checkpointing.

JOSS: not specified, but helpful for reviewers

Software Design and Coding Practices

- This workshop is not primarily a software design workshop
- Software design is a career - Software Engineers exist for a reason
- Increasingly common career path in science: Research Software Engineer
- Tips based on books and personal experience
- What is good design?

"Good Design Is Easier to Change Than Bad Design"

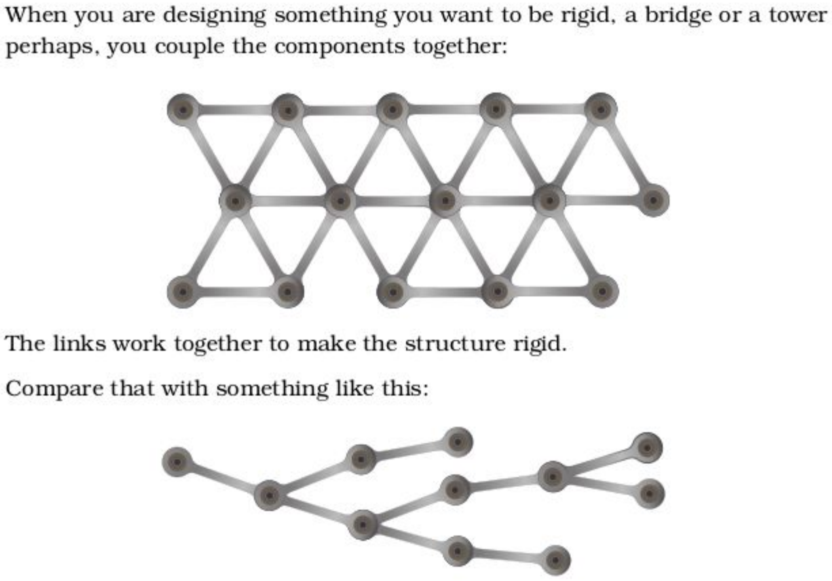
The Pragmatic Programmer

- Decoupling
- DRY
- Naming things
- Run-time configuration

Decoupling

When we try to pick out anything by itself, we find it hitched to everything else in the Universe.

➤ John Muir, My First Summer in the Sierra



When you are designing something you want to be rigid, a bridge or a tower perhaps, you couple the components together:

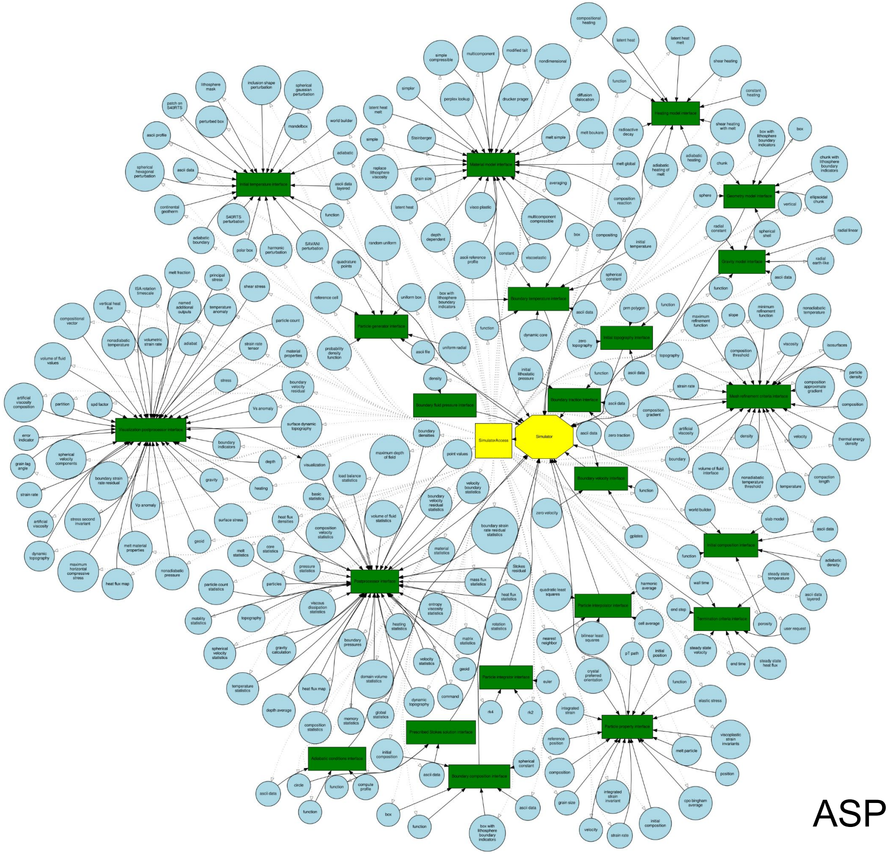
The links work together to make the structure rigid.

Compare that with something like this:

"Decoupled code is easier to change."

Easier change means better design.

Decoupling

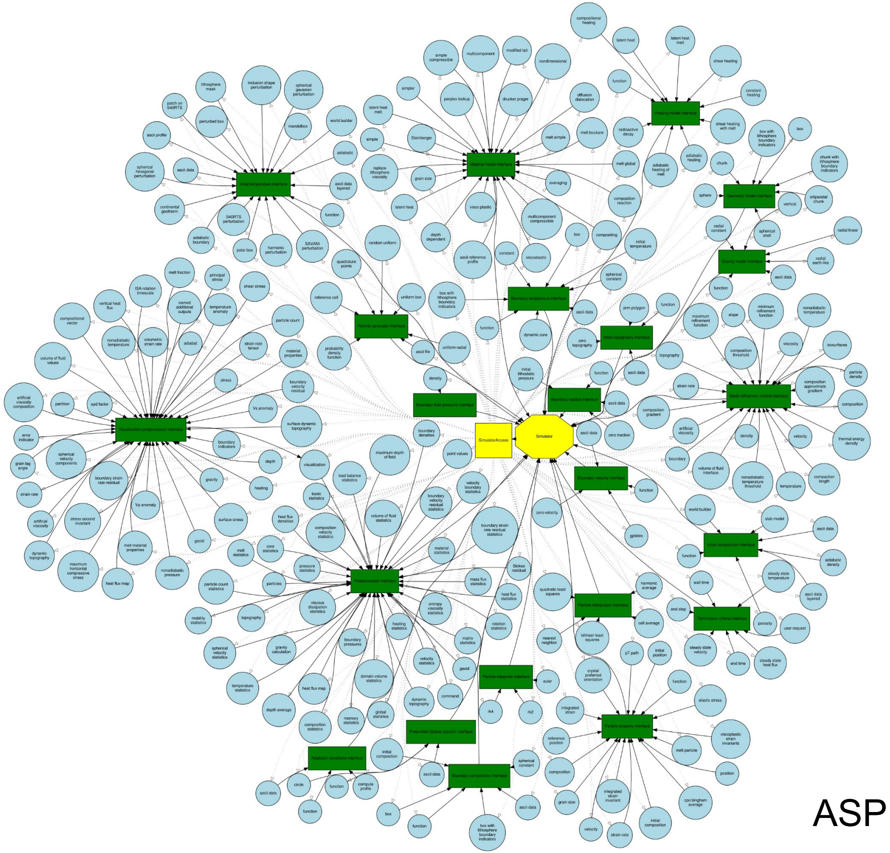

Simulator
SimulatorAccess
initial temperature interface
material model interface
heating model interface
geometry model interface
gravity model interface
boundary temperature interface
boundary traction interface
boundary velocity interface
postprocessor interface
visualization postprocessor interface
mesh refinement criteria interface
initial composition interface
boundary composition interface
particle property interface
particle generator interface
particle integrator interface
adiabatic conditions interface
prescribed Stokes solution interface
termination criteria interface

ASPECT plugin graph

- Decouple functionality through interfaces
- Each of ASPECT's >100 plugins (blue circles) implements only one of a small number of interfaces (green rectangles)
- Each interface defines how to interact with a plugin (e.g. every initial condition for temperature has a function called *initial_temperature*)
- This structure can be implemented for all object-oriented languages, and can be simulated in C and Fortran using function pointers

"Prefer Interfaces to Express Polymorphism"

Decoupling



ASPECT plugin graph

- It is often tempting to store information as global data
- However, global data couples parts of your program together
- Separate data according to responsibility
- If you cannot avoid global data, hide it behind a function (e.g. "get_timestep()"), this allows you to change the function implementation later

"Avoid global data"

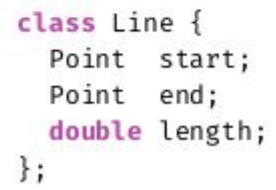
DRY - Don't repeat yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." *The Pragmatic Programmer*

DRY - Don't repeat yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

What does that mean?



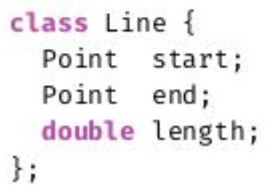
```
class Line {
  Point  start;
  Point  end;
  double length;
};
```

A line consists of a start point, and end point, and a length.

DRY - Don't repeat yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

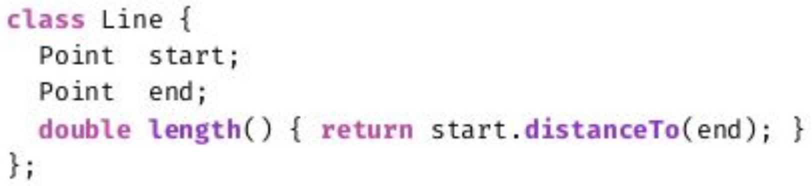
What does that mean?



```
class Line {
  Point  start;
  Point  end;
  double length;
};
```

A line consists of a start point, and end point, and a length.

What if we change the end and forget to update the length?



```
class Line {
  Point  start;
  Point  end;
  double length() { return start.distanceTo(end); }
};
```

DRY prevents this potential bug

DRY - Don't repeat yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

- Many people think DRY means “don’t copy-and-paste lines of source code.”
- But DRY is about the duplication of knowledge, of intent.
- If you perform the same operation many times in your code: Unify it's knowledge in a single function or class

Naming things

The beginning of wisdom is to call things by their proper name.
➤ Confucius
There are only two hard things in computer science:
cache invalidation and naming things.

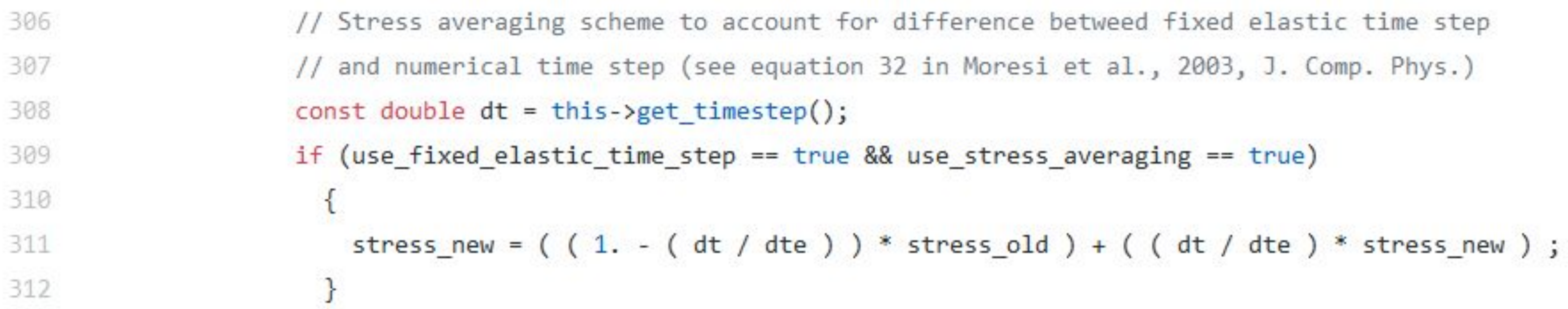
- Use self-explanatory variable names that convey intent
- Follow the culture in your field or project
- Start function names with verbs (get_…, transform_…), use nouns for variables (time_step, density)
- The time of 80-character screens and 1-character variable names is over -> avoid abbreviations
- The better you name things, the less comments you have to write

Naming things

The beginning of wisdom is to call things by their proper name.
➢ *Confucius*

There are only two hard things in computer science:
cache invalidation and naming things.

- Use self-explanatory variable names that convey intent

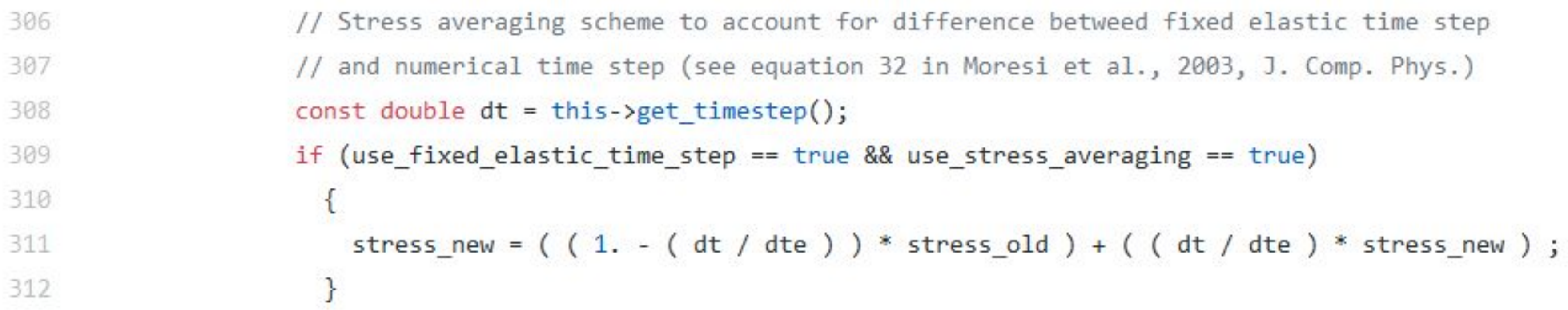


```
// Stress averaging scheme to account for difference betweed fixed elastic time step
// and numerical time step (see equation 32 in Moresi et al., 2003, J. Comp. Phys.)
const double dt = this->get_timestep();
if (use_fixed_elastic_time_step == true && use_stress_averaging == true)
  {
    stress_new = ( ( 1. - ( dt / dte ) ) * stress_old ) + ( ( dt / dte ) * stress_new ) ;
  }
```

ASPECT source code example

Comments

- Document intent and motivation, not mechanics
- *"Build Documentation In, Don’t Bolt It On"*
- When possible: Simplify code instead of explaining how it works



```
// Stress averaging scheme to account for difference betweed fixed elastic time step
// and numerical time step (see equation 32 in Moresi et al., 2003, J. Comp. Phys.)
const double dt = this->get_timestep();
if (use_fixed_elastic_time_step == true && use_stress_averaging == true)
  {
    stress_new = ( ( 1. - ( dt / dte ) ) * stress_old ) + ( ( dt / dte ) * stress_new ) ;
  }
```

ASPECT source code example

Run-time configuration

- Expressing run-time configuration as files makes your code more flexible
- *"Bend, don't break"*
- Simplifies testing with different input (see later session on testing)
- Use standard file formats (e.g. json, yaml, ini), this way you can use existing libraries to read them
- Or: Copy input systems from existing software
- If you cannot use input files: At least place configuration options in the same place (file)
- Output the configuration options

Resources for better software design

- David Thomas, Andrew Hunt: "The Pragmatic Programmer"
  - One of the defining books for good software design practices
- Robert C. Martin: "Clean Code", And "The Clean Coder"
  - More in-depth guides to good software design and best practices
- Michael C. Feathers: "Working Effectively with Legacy Code"
  - Great guide if you already have old code and don't know where to start improving

- None of these books are introductions to programming! For that see, e.g.: https://swcarpentry.github.io/python-novice-inflammation/

Exercise: Apply the software template

- Questions?

- Take time to download or clone the CIG software template: https://github.com/geodynamics/software_template
- Copy + paste the parts that apply to your software into your project
- Ask questions so that everyone can learn from them
- Link to these slides: http://bit.ly/2023-cig-joss
- We will continue at 11 am Pacific, 2 pm East Coast with Session 2 (Documentation, Testing, Reproducibility)

Session 2 - Documentation, Testing, Reproducibility

"English is Just Another Programming Language"
The Pragmatic Programmer

Documentation

CIG Best practices	**Minimum**	**Standard**	**Target**
Documentation	(a) Instructions for installation. (b) Description of all parameters. (c) Explanation of physics the code simulates. (d) Cookbook examples with input files. (e) Citable publication. (f) Documentation provided online or offline.	(a) Description of workflow for research use. (b) Description of how to extend code in anticipated ways.	Standard + (a) Guidelines on parameter scales/combinations for which code is designed/tested. (b) FAQs or knowledge base. (c) Documentation provided in dynamic form and available offline.

Documentation

JOSS:

- A statement of need: Do the authors clearly state what problems the software is designed to solve and who the target audience is?
- Installation instructions: Is there a clearly-stated list of dependencies? Ideally these should be handled with an automated package management solution.
- Example usage: Do the authors include examples of how to use the software (ideally to solve real-world analysis problems).
- Functionality documentation: Is the core functionality of the software documented to a satisfactory level (e.g., API method documentation)?
- Automated tests: Are there automated tests or manual steps described so that the functionality of the software can be verified?
- Community guidelines: Are there clear guidelines for third parties wishing to 1) Contribute to the software 2) Report issues or problems with the software 3) Seek support

A statement of need

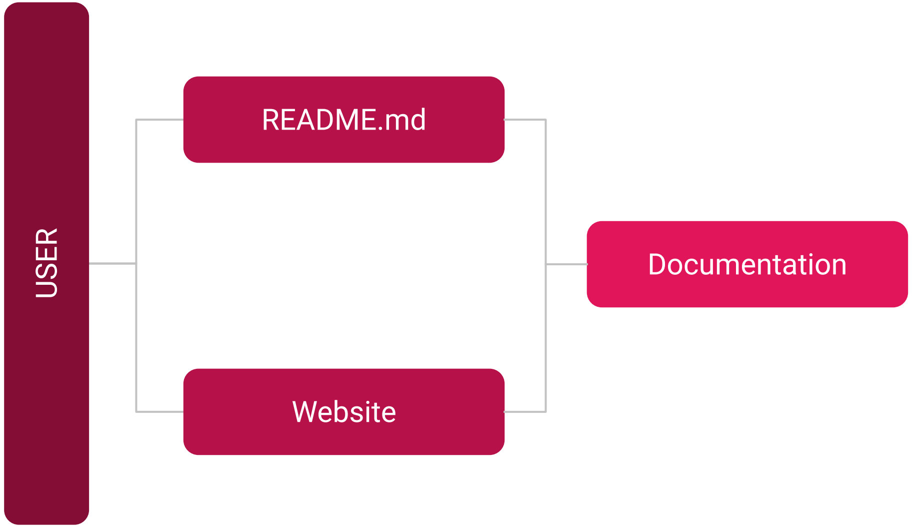
- Do the authors clearly state what problems the software is designed to solve and who the target audience is?
- Let your users know what they can expect from your software
- An example: ASPECT

  ASPECT is a code to simulate convection in Earth's mantle and elsewhere. It has grown from a pure mantle-convection code into a tool for many geodynamic applications including applications for inner core convection, lithospheric scale deformation, two-phase flow, and numerical methods development.

- Let's write a statement of need for your software:
  https://www.menti.com/al2afk43embn
  or menti.com, code: 3277 0881

Documentation Overview: Structure

- A file called README.md in the root folder acts as one entrypoint
- The webpage of your project is another entrypoint
- You want to minimize duplication of information
- We will discuss the implementations
  in the CIG software template



Documentation: README.md

- README.md is the most likely entrypoint for GitHub user and users who download first and ask questions later
- You can fulfill many publication criteria with a well written README
- Keep in mind that README is one file/one page. Keep it short and link to more extensive sections of your documentation.
- We will discuss the sections of the template README.md: https://github.com/geodynamics/software_template

Documentation: Website

- While your source-code lives in your online repository, you typically want a separate project website to showcase pictures, videos, interactive/dynamic content
- 3 typical options:
    - Self-hosted
    - Hosted by your online repository (GitHub pages, GitLab pages)
    - Hosted by readthedocs.org or another service
- Advantages of online repository and readthedocs.org:
    - Automatically built from latest version - Keep code and doc in sync
    - Easy links between website and code
    - Store multiple versions (e.g. release and development)

Documentation: Community Guidelines

- Both CIG and JOSS require a file called CONTRIBUTING.md that describes how to participate in your project
- Community Guidelines help by:
  - Making it more likely users will contribute
  - Reducing the amount of questions you receive
  - Forming a more friendly and open community

opensource.guide

- Let's take a look through the CONTRIBUTING.md of the software template

Documentation: Offline vs Online

- Documentation of geoscientific codes happens in mostly two varieties:
  - A PDF manual generated from LaTex source files
  - An online website (html), generated from source files in a markdown format (.md, or .rst)
- Both have pros/cons, which one you choose will depend on the history of your project, the needs of your community, and available development resources
- LaTex (offline) and Sphinx (online) templates can be found here: https://github.com/geodynamics/software_template/tree/main/doc
- The Sphinx template is hosted online here: https://software-template.readthedocs.io/en/latest/

LaTex Documentation: pros/cons

Pros	Cons
Commonly used for publications	Static content, how often regenerated?
Render formulas and complex tables	Where to host to make available?
PDFs are easily archivable	Binary included in repository?
Easy to package and ship	No easy online link to sections
	Package dependencies

Online Documentation: pros/cons

Pros	Cons
Always updated + available without limit	Uncommon directives (yet another language to write in)
Easy links to repository / files	First time configuration effort
Host multiple versions (devel, release)	Bound to a hosting service
Host multiple formats (pdf, html)	Harder to test locally
Easy to search + navigate	

2-page tutorial: Setting up a Sphinx documentation

We will work through the Sphinx template included in the CIG software template:
https://github.com/geodynamics/software_template/tree/main/doc/sphinx_template
Which results in the following documentation:
https://software-template.readthedocs.io

In particular we will discuss the files:

- the start page of the documentation
- the conf.py configuration file
- the environment.yml dependency file
- the .readthedocs.yml configuration file

2-page tutorial: Setting up a Sphinx documentation

To reiterate the important terms:

- Sphinx is the software that converts .md/.rst files into .html documentation
- Markdown (.md) / ReStructuredText (.rst) are the formats of source files
- ReadTheDocs is a website that allows the automatic building and hosting of generated documentation (e.g. generated by Sphinx)
- ReadTheDocs can be integrated into GitHub repositories to automatically rebuild documentation
- Let's take a look at the ReadTheDocs project for the software template: https://readthedocs.org/projects/software-template/

5 min break: Questions about Documentation?

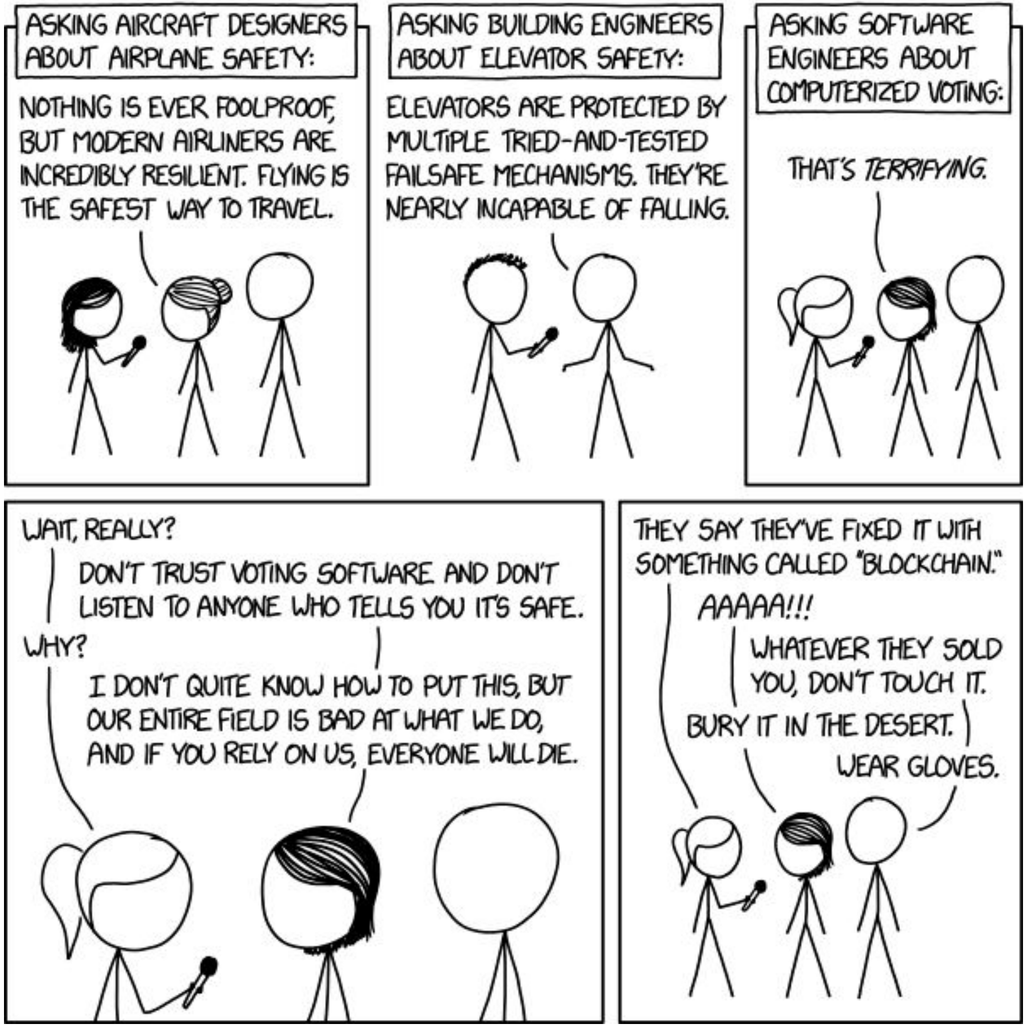
What type of documentation would you like to create?

https://www.menti.com/al2afk43embn

or menti.com, code: 3277 0881

Testing

Why?


ASKING AIRCRAFT DESIGNERS ABOUT AIRPLANE SAFETY:
NOTHING IS EVER FOOLPROOF, BUT MODERN AIRLINERS ARE INCREDIBLY RESILIENT. FLYING IS THE SAFEST WAY TO TRAVEL.
ASKING BUILDING ENGINEERS ABOUT ELEVATOR SAFETY:
ELEVATORS ARE PROTECTED BY MULTIPLE TRIED-AND-TESTED FAILSAFE MECHANISMS. THEY'RE NEARLY INCAPABLE OF FALLING.
ASKING SOFTWARE ENGINEERS ABOUT COMPUTERIZED VOTING:
THAT'S *TERRIFYING.*
WAIT, REALLY?
DON'T TRUST VOTING SOFTWARE AND DON'T LISTEN TO ANYONE WHO TELLS YOU IT'S SAFE.
WHY?
I DON'T QUITE KNOW HOW TO PUT THIS, BUT OUR ENTIRE FIELD IS BAD AT WHAT WE DO, AND IF YOU RELY ON US, EVERYONE WILL DIE.
THEY SAY THEY'VE FIXED IT WITH SOMETHING CALLED "BLOCKCHAIN."
AAAAA!!!
WHATEVER THEY SOLD YOU, DON'T TOUCH IT.
BURY IT IN THE DESERT.
WEAR GLOVES.

https://xkcd.com/2030/

Testing

CIG Best practices	**Minimum**	**Standard**	**Target**
Testing	(a) Code includes tests that verify it runs properly. (b) Results of accuracy and/or performance benchmarks (if established by the community).	Code includes pass/fail tests that verify it runs properly. (b) Development pipeline uses continuous integration to automate running test suite.	(a) Pass/fail unit testing for verification at a fine grain level. (b) Method of Manufactured Solutions for verification at a coarse grain level. (c) Use of code coverage tools to assess gaps in test coverage.

JOSS: Automated tests: Are there automated tests or manual steps described so that the functionality of the software can be verified?

Testing prevents software bugs

- A software bug is an error, flaw or fault in a computer program or system that causes it to produce an **incorrect** or **unexpected** result, or to behave in **unintended** ways. Wikipedia
- Bugs were known and found as far back as the 19th century
- Ada King, Countess of Lovelace, computing pioneer and possibly the first to write a computer program in 1842 (an algorithm for the analytical engine invented but never finished by Charles Babbage and her in 1837) wrote:

> ... an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary *operative* data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the *cards* may give it wrong orders.

Software bugs

- "First actual case of bug being found" is reported by Grace Hopper in 1947, her colleagues found a moth in the Harvard Mark II computer, blocking a relay

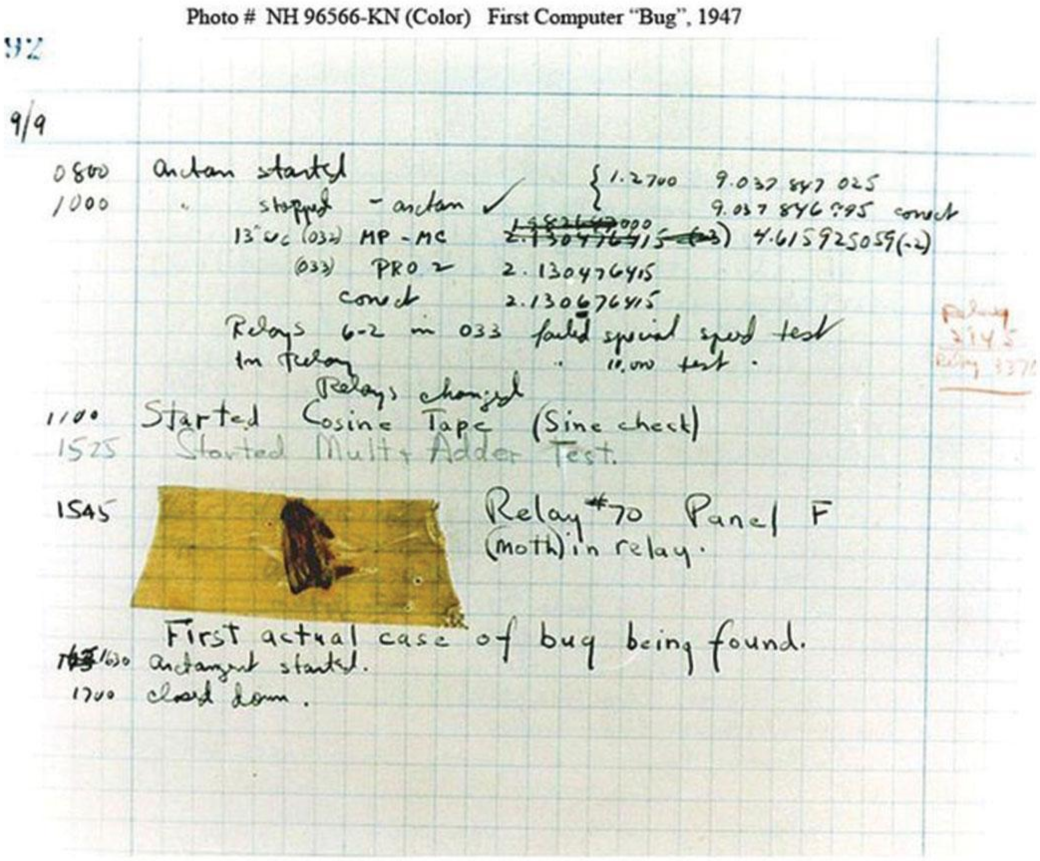


Photo # NH 96566-KN (Color) First Computer "Bug", 1947

```
92

9/9

0800   Antan started
1000     "     stopped - antan ✓      { 1.2700      9.037 847 025
                                         {             9.037 846 995 conect
         13" UC (032) MP - MC   ~~1.982147000~~
                                ~~2.130476415~~ (3)  4.615925059(-2)
               (033) PRO 2       2.130476415
                     conect      2.130676415
         Relays 6-2 in 033 failed special speed test
         In Relay           .     11.000 test .
                  Relays changed
1100   Started Cosine Tape (Sine check)
1525   Started Mult + Adder Test.
1545                                  Relay #70 Panel F
                                      (moth) in relay.
         First actual case of bug being found.
1630   Antangent started.
1700   closed down.
```

Software bugs

- Most security risks you read about in the news are caused by a bug
- Bugs create a huge economic cost and create a risk for society
- Famous bugs:
    - 1962: NASA Mariner 1 lost contact to ground control during launch, the backup system had a bug (a missing hyphen ‘-’) and steered the rocket towards a populated area on Earth, NASA aborted the launch by destroying the rocket
    - 1986: The THERAC-5 radiation therapy device had a software bug, the device administered a radiation overdose, 3 patients died and 3 suffered permanent health damage
    - 1996: The first Ariane 5 rocket crashed because of a variable type conversion error, estimated damage: $370m
    - 2016: ESA’s Schiaparelli Mars lander crashed, because a vibration of one of the landers arms was interpreted as having landed, leading to engine cutoff at an altitude of 3.7km, $230m
    - 2017: WannaCry (~$4 billion) and NotPetya (>$10 billion) attacks are the most expensive individual cyber attacks so far, enabled by software bugs

What is a test?

- A test is the process of feeding a program or part of a program selected input data that allows to check the programs output against an expected result.
- **A test:** Calling an averaging function with the array [1,2,3] and expecting the output [2].
- **A test:** Calling a format conversion script with a given input file, for which a correct output file in the new format exists, and comparing the output against the existing file.
- **Not a test:** Running a program with input data and guessing if the output looks more or less correct.

- **Is this a test?** A computation produced an output a few months ago. The output was stored and every time the program is changed the program is rerun with the same input and the new output is compared against the original.

What is a test?

- A test is the process of feeding a program or part of a program selected input data that allows to check the programs output against an expected result.
- **A test:** Calling an averaging function with the array [1,2,3] and expecting the output [2].
- **A test:** Calling a format conversion script with a given input file, for which a correct output file in the new format exists, and comparing the output against the existing file.
- **Not a test:** Running a program with input data and guessing if the output looks more or less correct.

- **Is this a test?** A computation produced an output a few months ago. The output was stored and every time the program is changed the program is rerun with the same input and the new output is compared against the original. **Yes**

What are different types of tests?

- Testing is often divided into two categories:
  - **Verification**: Proof that you do what you intended *this is topic today*
  - **Validation**: Proof that what you intended is the correct thing to do *this is hard*

- *Example: A program that does not detect landslides based on satellite data, because it was trained/coded in a wrong way fails in **verification.** A program that does not detect landslides based on satellite data, because they are impossible to detect based on this data fails in **validation**.*

- What types of verification tests are there?
  - Many more variants, but relevant for scientific applications:

  - **Manual / Exploratory tests**
  - **Integration / Regression tests**
  - **Unit tests**
  - **Benchmarks / Method of Manufactured Solution (MMS)**

Manual / Exploratory tests

- This is what most scientists think of when they are asked to test their code
- They may run their script, look at the figure, check if it looks reasonable
- Maybe run with different data sets or variations of parameters to make sure the parameters work
- Problems:

  Each run takes your valuable time!

  Humans are bad at spotting small errors!
- **Acceptable** if:
  - You are throwing together a one-off figure
  - You are plotting a dataset without modifying it, and know the dataset is reliable
- **Not acceptable** if:
  - You are planning to use the result in a publication or presentation
  - Colleagues (incl your future self) will rely on your results for additional work

Integration / Regression tests

- Integration tests automate the manual testing procedure
- Idea: Run the manual test with defined input and defined output, and make sure the program creates the expected output
- Example:
  - Store a small artificial dataset in the data/ directory
  - Run your software on it and store the (manually checked) output data in a tests/ directory
  - Write a function ‘test()’ that runs the script on the input data and compares the result with the previously stored output data
  - Run this function automatically every time a commit is made
- These tests not only save you time, they also catch things you break accidentally after they worked before (called **‘regressions’**)

Do’s and Don’ts for Integration Tests

- Cover all important application cases, do not have a significant feature without an integration test
- Use smaller versions of your full datasets to keep the test cases fast (<1 min per test, ideally seconds)
- Do not write tests that test the same functionality (2 integration tests that test different time series of the same structure are not useful), this concept is called **orthogonal** testing
- Output images are hard to test this way. Make it easier by writing the data you plot to a file as well

Unit tests

- Like an integration test, but instead of running the whole script, run the smallest unit of your script possible (often one function at a time)
- Much easier to create, because you control input and output of the function
- ***Exercise**:*
    - Assume you have a function that computes the running average of a time series
    - What are input data examples you should test?

Unit tests

- Like an integration test, but instead of running the whole script, run the smallest unit of your script possible (often one function at a time)
- Much easier to create, because you control input and output of the function
- *Exercise:*
  - Assume you have a function that computes the running average of a time series stored in an array. The function allows to select the size of the averaging window.
  - What are input scenarios you should test?
  - A time series of constant values
  - A time series of a linearly increasing values (you can easily compute the running average)
  - A time series with no values
  - A time series with corrupted values
  - A time series that is shorter than the averaging window
- Unit tests are extremely important, because they can cover your whole code and run extremely fast

Do’s and Don’ts for Unit Tests

- Give unit tests expressive names (e.g. test_averaging_function_constant, test_averaging_function_linear, test_averaging_function_empty, …)
- Use a testing framework like **pytest**, **unittest** (Python) or **testthat** (R) or **cmake/ctest**, googletest, catch (C/C++), Fortran to test easy and fast
- Do not write tests for the same functionality (2 unit tests that test different time series of the same structure are not useful)
- Do not write unit tests that depend on each other, they should be independent (one should not use variables or files generated by another)

Benchmarks

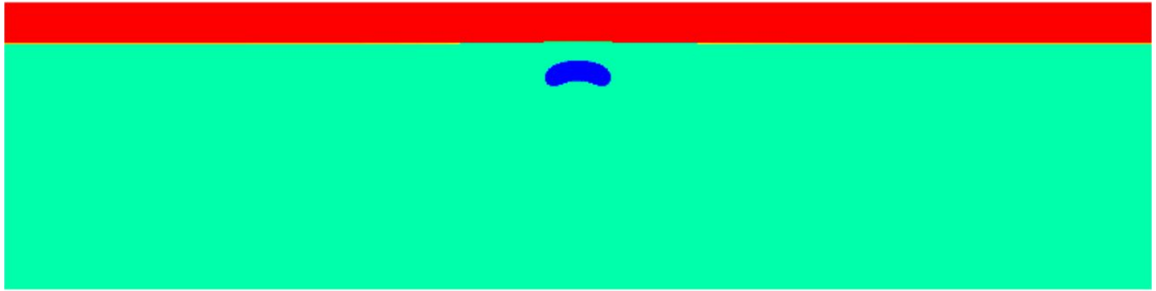
- **Benchmarks** are defined applications of a whole program, which are relevant for the application field and allow measuring the accuracy of different algorithms.
- **Benchmarks** are expensive to create and require significant thought to produce a relevant, yet tractable problem with well defined **performance metrics** (how well did the program perform)
- **Benchmarks** are beyond the scope of this workshop, creating a new benchmark or performing a number of benchmarks with a new code usually warrants a new publication

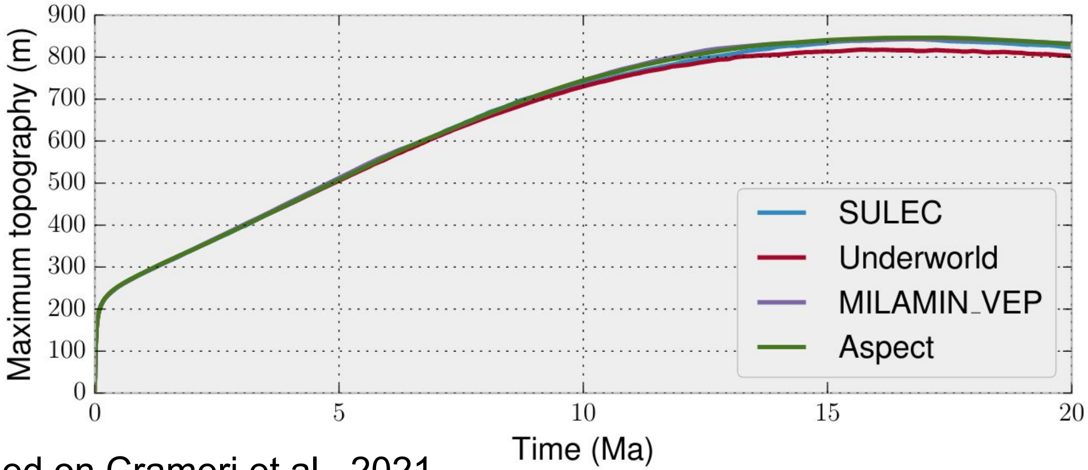
Benchmark examples

- There are two subgroups of benchmarks:
    - **Code comparisons** (sometimes called **community benchmarks**) compare the output of several programs when applied to the same application and try to judge their relative accuracy and speed based on the results.
    - **Manufactured solutions** are application scenarios with known solutions, to which the program output can be directly compared.

Community benchmark example

- Compare 4 codes for a given application (surface topography above a thermal mantle anomaly)
- Other examples:
- [SCEC community benchmark]
- [Geodynamo community benchmark]
- [Mantle Convection community benchmark]





ASPECT Manual, based on Crameri et al., 2021

MMS example

- Construct an analytical solution to a simple problem to measure accuracy as difference between numerical and analytical solution.

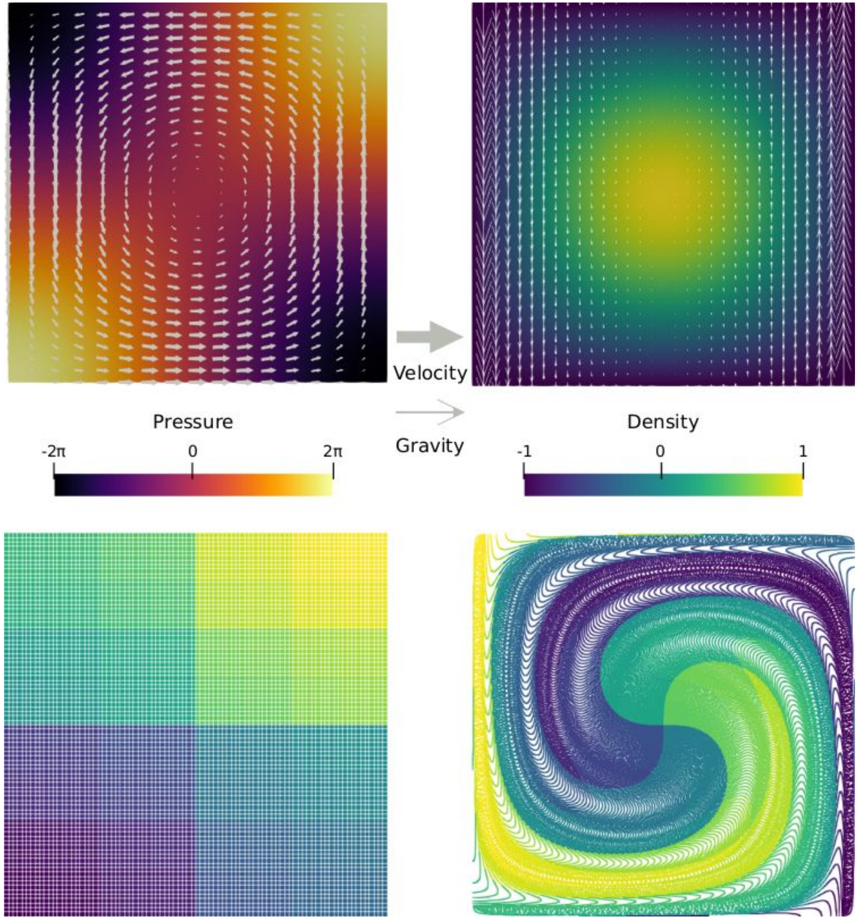
$$\boldsymbol{u} = \begin{bmatrix} \sin(\pi x)\cos(\pi y) \\ -\cos(\pi x)\sin(\pi y) \end{bmatrix},$$

$$p = 2\pi\cos(\pi x)\cos(\pi y),$$

$$\rho = \sin(\pi x)\sin(\pi y),$$

$$\boldsymbol{g} = \begin{bmatrix} 0 \\ -4\pi^2 \frac{\cos(\pi x)}{\sin(\pi x)} \end{bmatrix}.$$

Gassmoeller et al., 2020



MMS example

- Construct an analytical solution to a simple problem to measure accuracy as difference between numerical and analytical solution.

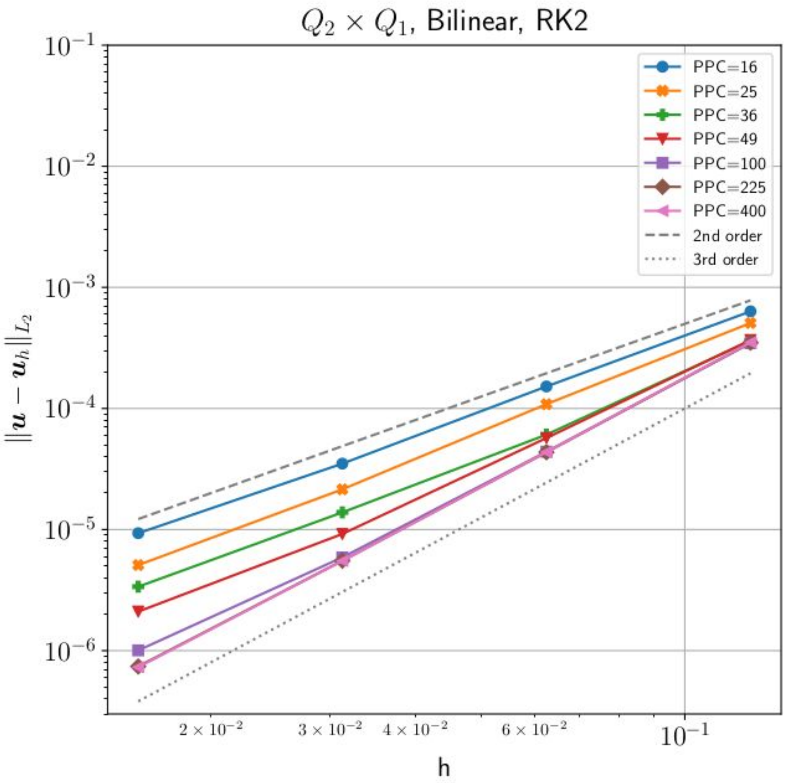
$$\boldsymbol{u} = \begin{bmatrix} \sin(\pi x)\cos(\pi y) \\ -\cos(\pi x)\sin(\pi y) \end{bmatrix},$$

$$p = 2\pi\cos(\pi x)\cos(\pi y),$$

$$\rho = \sin(\pi x)\sin(\pi y),$$

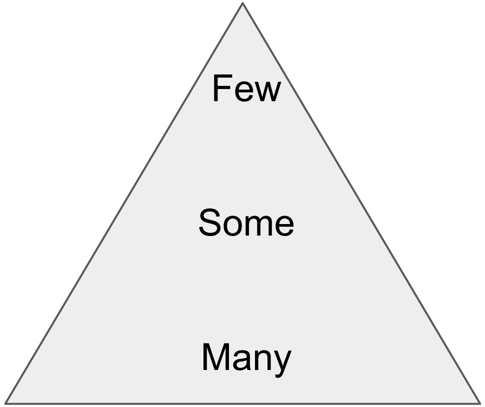
$$\boldsymbol{g} = \begin{bmatrix} 0 \\ -4\pi^2 \frac{\cos(\pi x)}{\sin(\pi x)} \end{bmatrix}.$$

Gassmoeller et al., 2020



The Testing Pyramid

- How should you test? *"Test Early, Test Often, Test Automatically"*



Test speed	*Number of tests*	*Type of test*	*Cost*
Slow (minutes)	Few	**Manual tests / Benchmarks**	Expensive
Medium (<1 min)	Some	**Integration tests**	Moderate
Fast (<1 s)	Many	**Unit tests**	Cheap

The benefits of tests

- Catching bugs early is much cheaper and less embarrassing than having to debug after results have been published
- Having a well tested software allows you to make changes with the certainty of not breaking things
- “Refactoring” (improving the code quality without changing functionality) is much much easier with tests:
  “refactoring without tests isn’t refactoring, it is just moving shit around.”
  Corey Haines (@coreyhaines), Dec 20, 2013, Twitter

An alternative approach: Test-driven development (TDD)

- A common approach to programming is to first write the code, then test if it works
- However, over the last two decades a different approach has proven to be more effective in commercial software development: Write the test first, then write the code that satisfies this test
- This model is called **Test-driven development**

Test-driven development (TDD)

- *Example: Write a function that computes the average of an array*

  1. *Write a test that calls the (not yet existing function) with the array [0], you expect the function to return [0]*
     a. *Check that the test fails*
     b. *Write the function and always return 0*
     c. *Check that the test succeeds.*
  2. *Write a second test that uses [1] as input and expects [1] as output*
     a. *Check that the second test fails*
     b. *Modify the function to pass both tests*
     c. *Check that both tests succeed*
  3. *Write a test that uses [0,1] as input and expects [0.5] as output*
     a. *Check that the third test fails*
     b. *...*

Test-driven development (TDD)

- TDD has been shown to produce better code in less time than traditional software development, because:
  - You think more deeply about what you expect your program to do
  - You automatically create tests for all functionality in your program (no more forgotten tests)
  - You structure your program in a way that is easier to understand

- 3 Rules of test-driven development (Robert C Martin, The Clean Coder):
  1. You are not allowed to write any production code until you have written a failing unit test for it.
  2. You are not allowed to write more of a unit test than is sufficient to fail - and not compiling is failing.
  3. You are not allowed to write more production code than is sufficient to pass the currently failing unit test.

Summary

- Test Early, Test Often, Test Automatically!
- Tests are critical to proof that your program does what you think it does!
- Whether you embrace TDD or classical code-first test-later strategies, make sure you have tests in place that tell you your program is not *approximately* correct, *apparently* correct, or *almost* correct, but that it is **correct** for defined test cases!
- Think about what you expect your program to do before running it and quantitatively test the output. If something looks funny or unexpected, it is likely a bug!

Tests in Action

- As example the software ASPECT has a test suite of 1736 unit tests (using catch), and 1044 integration tests (using ctest)
- These tests are executed for every pull request using something that is called continuous integration (next topic)
- Let’s take a look: https://github.com/geodynamics/aspect/tree/main/tests

Continuous integration and automated testing

“**Continuous Integration** (**CI**) is a development practice where developers **integrate** code into a shared repository frequently, preferably several times a day. Each **integration** can then be verified by an automated build and automated tests. While automated testing is not strictly part of **CI** it is typically implied.” https://www.cloudbees.com/continuous-delivery/continuous-integration

- Continuous integration is a practice first introduced in 1991, nowadays employed for nearly every software project

Continuous integration and automated testing

- Many aspects of CI are mostly relevant for large projects, but for us it means:
  - Make small, incremental changes to your software, then test and commit
  - If you collaborate with someone else, make sure you frequently merge all of your changes
  - Make sure to always keep your software in a working state, do not commit broken functionality
  - Make sure to have automated tests on GitHub to detect accidental mistakes
  - If a test is broken fix it before working on something else
  - Test your tests!
- Continuous integration tools make CI and automated testing easier, examples are GitHub Actions/Workflows (https://github.com/features/actions), GitLab Pipelines (https://docs.gitlab.com/ee/ci/pipelines/), Jenkins (https://www.jenkins.io/), Azure pipelines (https://azure.microsoft.com/en-us/products/devops/pipelines)
- We will discuss **GitHub Actions** for this workshop

What are GitHub Actions?

- GitHub Actions are **workflows** that GitHub will execute in your repository, whenever a certain **event** happens
- It will use cloud resources for these commands, e.g. a virtual machine in Microsoft’s Azure cloud
- **Events** could be:
  - Every 15 minutes
  - Every time someone pushes to your repository on a certain branch
  - Every time someone opens a pull request (PR)
- **Workflows** could be:
  - Compile your software
  - Run all of your tests
  - Build your documentation
  - Update a webpage

What are GitHub Actions?

- GitHub actions are controlled by configuration files inside the folder:
  .github/workflows
- If a file of the correct type is inside this folder GitHub will try to execute it
- Workflow files need to be in a special format, called **yaml** (Yet Another Markup Language), an example is here:

  https://github.com/geodynamics/Rayleigh/blob/main/.github/workflows/main.yml

- Example results are here
- Because these actions will happen on a virtual machine, we need to make sure all dependencies are installed correctly. This can be done using containers, or dependency management systems like conda or cmake (next topic).

5 min break. Questions about tests?

Reproducibility

Categories:

1. Installation + system dependencies
2. Software version
3. User input + data

Reproducibility

CIG Best practices	**Minimum**	**Standard**	**Target**
Portability, configuration, and building	(a) Codes builds on Unix-like machines with free tools. (b) Portable build system.	Minimum + (a) Dependency checking. (b) Automation and portability of configuration and building. (c) Each simulation outputs all configuration and build options for reproducibility.	Standard + (a) Selection of compilers, optimization, build flags during configuration without modifying files under version control. (b) Multiple builds using same source. (c) Allows installation to a central location.

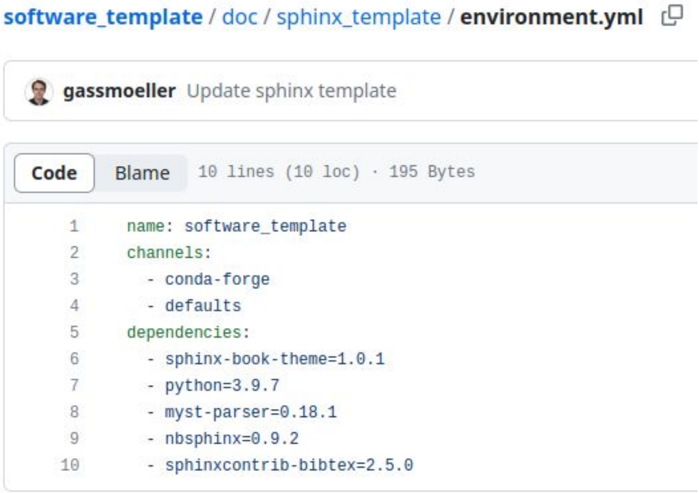
JOSS:
Is there a clearly-stated list of dependencies? Ideally these should be handled with an automated package management solution.

Installation + system dependencies

- Software often depends on system libraries or other projects
- The versions of these dependencies can vary, affecting your software results. Bugs can be specific to compilers, library versions, operating systems
- Managing dependencies can be split into two tasks:
  - Documenting supported dependency versions
  - Documenting dependency versions that were used for a specific installation

Documenting supported dependency versions

- Easiest, but brittle: Document in README.md or installation instructions
  - Can be outdated, incomplete, overlooked
- Better: Document in code, e.g.:
- Inside an environment.yml file for Python projects
- Inside a CMakeLists.txt if using cmake

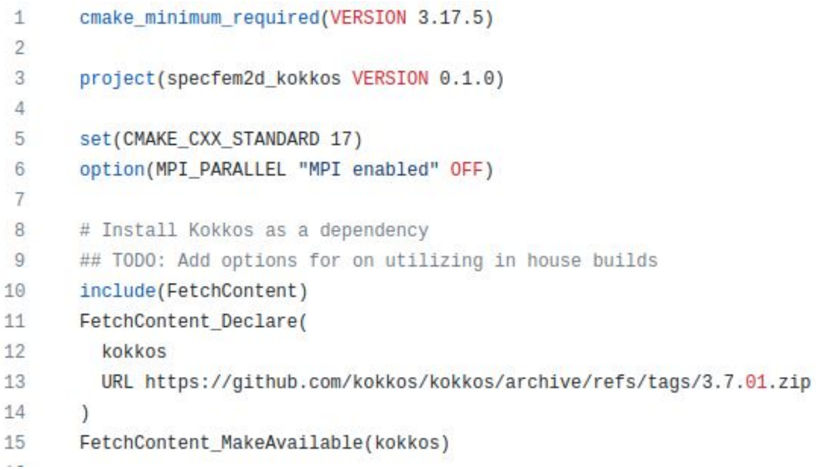


software_template / doc / sphinx_template / environment.yml

gassmoeller Update sphinx template

10 lines (10 loc) · 195 Bytes

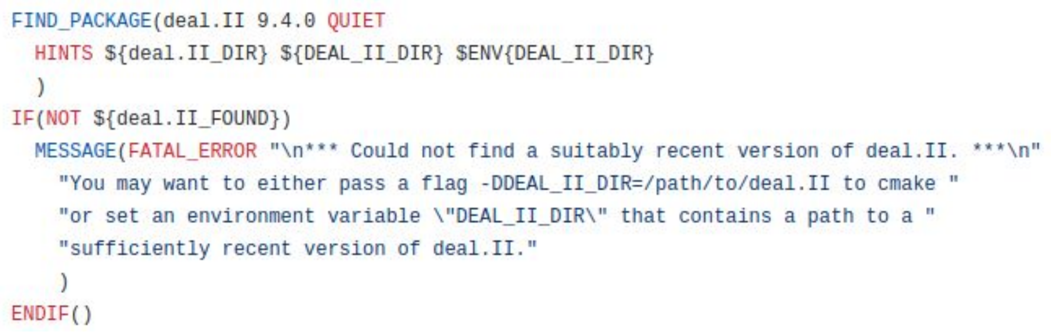
```
name: software_template
channels:
  - conda-forge
  - defaults
dependencies:
  - sphinx-book-theme=1.0.1
  - python=3.9.7
  - myst-parser=0.18.1
  - nbsphinx=0.9.2
  - sphinxcontrib-bibtex=2.5.0
```



```
cmake_minimum_required(VERSION 3.17.5)

project(specfem2d_kokkos VERSION 0.1.0)

set(CMAKE_CXX_STANDARD 17)
option(MPI_PARALLEL "MPI enabled" OFF)

# Install Kokkos as a dependency
## TODO: Add options for on utilizing in house builds
include(FetchContent)
FetchContent_Declare(
  kokkos
  URL https://github.com/kokkos/kokkos/archive/refs/tags/3.7.01.zip
)
FetchContent_MakeAvailable(kokkos)
```

Documenting supported dependency versions

- Easiest, but brittle: Document in README.md or installation instructions
- Better: Document in code
- Even if you do not install the dependencies, always check installed dependency versions, e.g. in CMakeLists.txt:

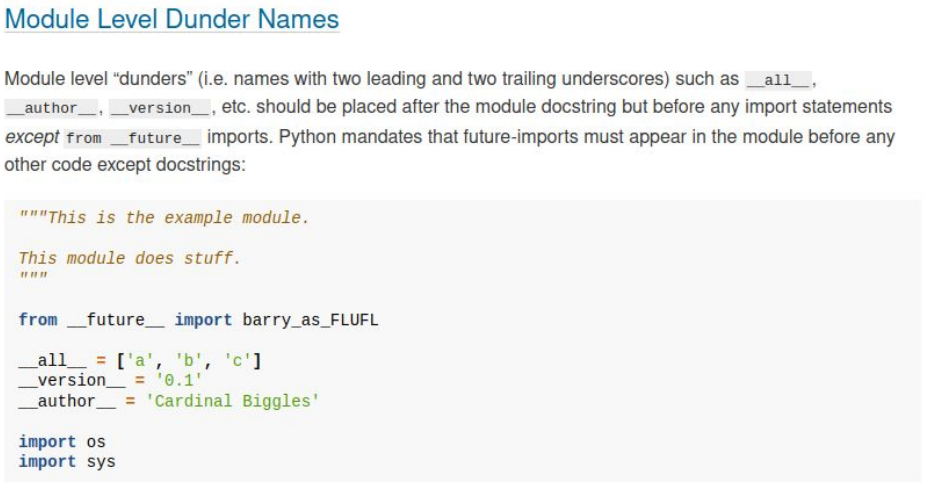


```
FIND_PACKAGE(deal.II 9.4.0 QUIET
  HINTS ${deal.II_DIR} ${DEAL_II_DIR} $ENV{DEAL_II_DIR}
  )
IF(NOT ${deal.II_FOUND})
  MESSAGE(FATAL_ERROR "\n*** Could not find a suitably recent version of deal.II. ***\n"
    "You may want to either pass a flag -DDEAL_II_DIR=/path/to/deal.II to cmake "
    "or set an environment variable \"DEAL_II_DIR\" that contains a path to a "
    "sufficiently recent version of deal.II."
    )
ENDIF()
```

Documenting dependency versions in use

Documenting dependency versions in use

- Make version of software and dependencies available at run-time
- Use a single source of truth for your version number! (e.g. a VERSION file), do not put your version number in every file! Makes releases easier.
- For Python: Store version number in member variable, E.g. following PEP8:



Module Level Dunder Names

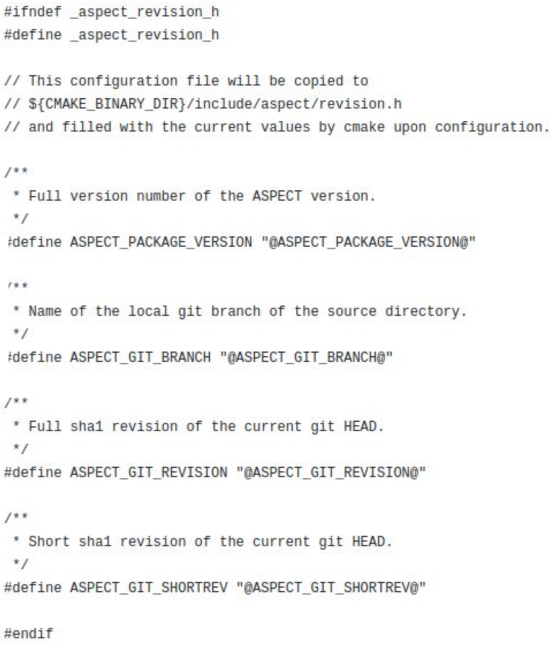
Module level "dunders" (i.e. names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc. should be placed after the module docstring but before any import statements *except* `from __future__` imports. Python mandates that future-imports must appear in the module before any other code except docstrings:

```
"""This is the example module.

This module does stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

Documenting dependency versions in use

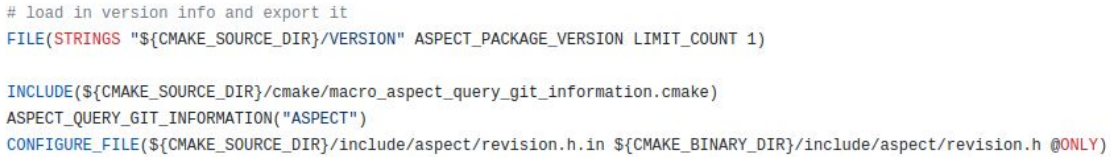
- Make version of software and dependencies available at run-time
- For compiled languages: Make use of CMake configuration options:

include/aspect/revision.h.in:



```
#ifndef _aspect_revision_h
#define _aspect_revision_h

// This configuration file will be copied to
// ${CMAKE_BINARY_DIR}/include/aspect/revision.h
// and filled with the current values by cmake upon configuration.

/**
 * Full version number of the ASPECT version.
 */
#define ASPECT_PACKAGE_VERSION "@ASPECT_PACKAGE_VERSION@"

/**
 * Name of the local git branch of the source directory.
 */
#define ASPECT_GIT_BRANCH "@ASPECT_GIT_BRANCH@"

/**
 * Full sha1 revision of the current git HEAD.
 */
#define ASPECT_GIT_REVISION "@ASPECT_GIT_REVISION@"

/**
 * Short sha1 revision of the current git HEAD.
 */
#define ASPECT_GIT_SHORTREV "@ASPECT_GIT_SHORTREV@"

#endif
```

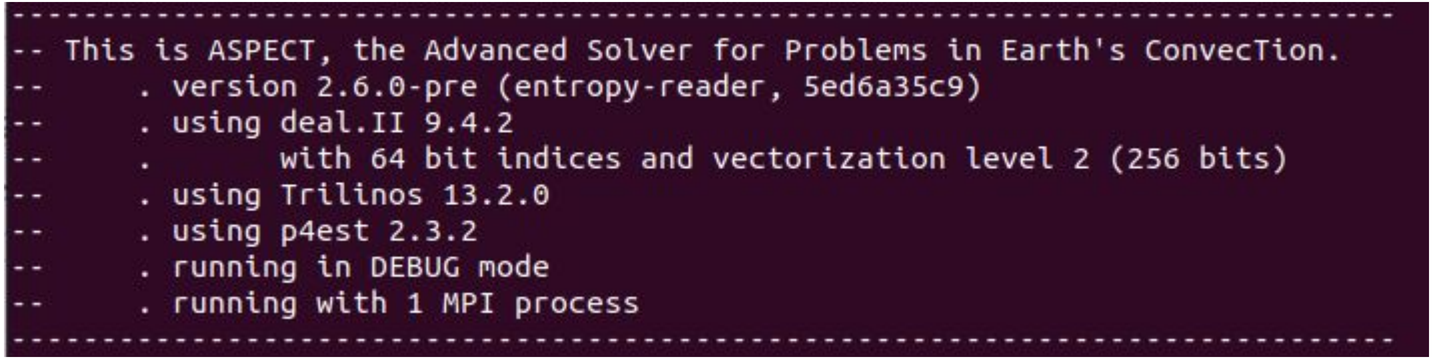
CMakeLists.txt:



```
# load in version info and export it
FILE(STRINGS "${CMAKE_SOURCE_DIR}/VERSION" ASPECT_PACKAGE_VERSION LIMIT_COUNT 1)

INCLUDE(${CMAKE_SOURCE_DIR}/cmake/macro_aspect_query_git_information.cmake)
ASPECT_QUERY_GIT_INFORMATION("ASPECT")
CONFIGURE_FILE(${CMAKE_SOURCE_DIR}/include/aspect/revision.h.in ${CMAKE_BINARY_DIR}/include/aspect/revision.h @ONLY)
```

Documenting dependency versions

- Great help for reproducibility and publishing
- Output versions during software run, e.g. ASPECT header:



```
-----------------------------------------------------------------------------
-- This is ASPECT, the Advanced Solver for Problems in Earth's ConvecTion.
--     . version 2.6.0-pre (entropy-reader, 5ed6a35c9)
--     . using deal.II 9.4.2
--     .       with 64 bit indices and vectorization level 2 (256 bits)
--     . using Trilinos 13.2.0
--     . using p4est 2.3.2
--     . running in DEBUG mode
--     . running with 1 MPI process
-----------------------------------------------------------------------------
```

- Output on screen and write to a file

Document user input + data

- Store the user input of a software run in a file
- If your code is controlled by a user input file: Make a copy of the file, place it in the output directory
- This data is important to keep and publish (e.g. as data package on Zenodo) to fulfil data availability policies of publishers
- It is also useful for you, once you finished your 50 model study, you still have the configuration for each model

- CIG has draft model publishing guidelines here: https://tinyurl.com/cig-publish

Containers

- Software containers are a way to isolate a software environment and all its dependencies from the host system
- You can ship containers around and run them (almost) independent of the hardware they are using
- Creating a container of your software will allow reproducibility of a workflow for a long time
- A full tutorial on containers: https://carpentries-incubator.github.io/docker-introduction/index.html

Exercise: Testing, Documentation, Reproducibility

- Questions?

- How do you test your software? How do you document? How can you increase reproducibility?
- Ask questions so that everyone can learn from them
- Link to these slides: http://bit.ly/2023-cig-joss
- We will continue at 1 pm Pacific, 4 pm East Coast with Session 3 (Software Publication)

Session 3 - Navigating Publication in Software Journals

- Why do software journals exists?
- Which software journals exist?
- Why the Journal of Open Source Software (JOSS)?
- How does a submission to JOSS proceed?
- Discussion time

Software journals - Why?

- Software is a research product, just like scientific results
- Traditional publications are not focussed on the software, they are focussed on the scientific result, the "implementation" is typically not reviewed
- Results depend on software, so there should be a review
- A mechanism for credit
- A mechanism for reproducibility

Software journals - Which one?

- A number of old+new journals accept software publications:
    - JOSS: The Journal of open-source software
    - GMD: Geoscientific Model Development
    - AGU journals: Technical Report:Methods
    - SoftwareX
    - For more see this list by the Software Sustainability Institute: https://www.software.ac.uk/which-journals-should-i-publish-my-software
- We will mainly discuss a JOSS publication, but will briefly introduce the other mentioned journals

GMD (EGU, Copernicus, IF: 5.1)

Geoscientific Model Development (GMD) is a not-for-profit international scientific journal dedicated to the publication and public discussion of the description, development, and evaluation of numerical models of the Earth system and its components. The following manuscript types can be considered for peer-reviewed publication:

- geoscientific model descriptions, from statistical models to box models to GCMs;
- development and technical papers, describing developments such as new parameterizations or technical aspects of running models such as the reproducibility of results;
- new methods for assessment of models, including work on developing new metrics for assessing model performance and novel ways of comparing model results with observational data;
- papers describing new standard experiments for assessing model performance or novel ways of comparing model results with observational data;
- model experiment descriptions, including experimental details and project protocols;
- full evaluations of previously published models.

Technical Reports (AGU e.g. G3, IF: variable)

"Technical reports: Methods" describe novel analytical or experimental methods that enable new science, as well as other technical advances, including computer programs and instrumentation. These papers are limited to 13 publication units (1PU = 500 words / 1 figure) and will typically include at least one illustrative example application. Please contact journal staff to determine if that journal offers this paper type.

- Domain-relevant journal
- Reviewers may not be software developers
- Software itself may not be reviewed, only description of methods

SoftwareX (Elsevier, IF: 3.4)

SoftwareX aims to acknowledge the impact of software on today's research practice, and on new scientific discoveries in almost all research domains. *SoftwareX* also aims to stress the importance of the software developers who are, in part, responsible for this impact.

To this end, *SoftwareX* aims to support publication of research software in such a way that:

- The software is given a stamp of scientific relevance, and provided with a peer-reviewed recognition of scientific impact;
- The software developers are given the credits they deserve;
- The software is citable, allowing traditional metrics of scientific excellence to apply;
- The academic career paths of software developers are supported rather than hindered;
- The software is publicly available for inspection, validation, and re-use.

Journal of Open-Source Software

"The Journal of Open Source Software (JOSS) is an academic journal (ISSN 2475-9066) with a formal peer review process **that is designed to improve the quality of the software submitted.** Upon acceptance into JOSS, a Crossref DOI is minted and we list your paper on the JOSS website."

"If you’ve already developed a fully featured research code, released it under an OSI-approved license, and written good documentation and tests, then we expect that it should take perhaps an hour or two to prepare and submit your paper to JOSS."

- We will use the rest of this workshop to work through the JOSS submission process
- Implementing all the best practices makes JOSS submission easy

Software journals brief comparison

	JOSS	GMD	AGU	SoftwareX
Software best practices	🟩	🟧	🟥	🟧
Open source / free to use	🟩	🟧	🟧	🟩
Domain relevance	🟥	🟩	🟩	🟥
Methods	🟧	🟩	▨	▨
Open review	🟩	🟩	🟥	🟥
Open access (article)	🟩	🟩	🟧	🟥
Free to publish	🟩	🟥	🟥	🟥

🟩	yes
🟧	maybe
🟥	no
▨	no information

JOSS Submission requirements

1. The software must be open source as per the OSI definition.
2. The software must be hosted at a location where users can open issues and propose code changes without manual approval of (or payment for) accounts.
3. The software must have an **obvious** research application.
4. You must be a major contributor to the software you are submitting, and have a GitHub account to participate in the review process.
5. Your paper must not focus on new research results accomplished with the software.
6. Your paper (`paper.md` and BibTeX files, plus any figures) must be hosted in a Git-based repository together with your software (although they may be in a short-lived branch which is never merged with the default).

JOSS paper contents

A JOSS paper is a short article

1. A list of the authors of the software and their affiliations, using the correct format (see the example below).
2. A summary describing the high-level functionality and purpose of the software for a diverse, *non-specialist audience*.
3. A *Statement of need* section that clearly illustrates the research purpose of the software and places it in the context of related work.
4. A list of key references, including to other software addressing related needs. Note that the references should include full names of venues, e.g., journals and conferences, not abbreviations only understood in the context of a specific discipline.
5. Mention (if applicable) a representative set of past or ongoing research projects using the software and recent scholarly publications enabled by it.
6. Acknowledgement of any financial support.

Submission template

JOSS provides a template for a software paper here: https://joss.readthedocs.io/en/latest/submitting.html#example-paper-and-bibliography

Other examples for Geoscientific JOSS papers are:

- https://joss.theoj.org/papers/10.21105/joss.05389
- https://joss.theoj.org/papers/10.21105/joss.01797
- https://joss.theoj.org/papers/10.21105/joss.02043
- https://joss.theoj.org/papers/10.21105/joss.05073

Checking that your paper compiles

JOSS offers two ways to check that your paper has the right structure:

- Using a GitHub action
- Using a Docker container

You can use either one.

JOSS submission workflow

Submitting your paper

Submission is as simple as:

- Filling in the short submission form
- Waiting for the managing editor to start a pre-review issue over in the JOSS reviews repository: https://github.com/openjournals/joss-reviews

No submission fees

There are no fees for submitting or publishing in JOSS. You can read more about our cost and sustainability model.

Preprint Policy

Authors are welcome to submit their papers to a preprint server (arXiv, bioRxiv, SocArXiv, PsyArXiv etc.) at any point before, during, or after the submission and review process.

Review process

The review process happens in a github issue in the following repository: https://github.com/openjournals/joss-reviews. The review process is public.

The review process is initiated by the editor, and performed by independent reviewers.

An EditorialBot supports and automates common operations.

Let's walk through an example review process: https://github.com/openjournals/joss-reviews/issues/5073

Questions about any part of the workshop?

- Ask your questions now or later in the CIG forum: https://community.geodynamics.org/c/computational-science/17
- Let us know how we did and if you will participate in person for part II in October: https://www.menti.com/al2afk43embn or menti.com, code: 3277 0881

- Consider contributing your software to CIG or JOSS to receive more publicity and credit.
- The CIG Seismic Cycles Working Group is planning a special issue in JOSS. Let us know if you are interested.

Thank you for your participation

And a big thank you to the National Science Foundation for supporting our work.

National Science Foundation

NSF - 21490126

dynamo seismology computational science short term crustal dynamics

CIG

COMPUTATIONAL INFRASTRUCTURE for GEODYNAMICS

mantle convection melt and volatiles long term tectonics education