Managing Open Source Scientific Software Projects

Timo Heister (Clemson University)
heister@clemson.edu

2016-03-15

CLEMSON UNIVERSITY
National Science Foundation WHERE DISCOVERIES BEGIN
CIG COMPUTATIONAL INFRASTRUCTURE for GEODYNAMICS

Hello, my name is ...

Timo Heister

- Assistant Professor at Clemson since 2013 in Mathematical Sciences
- PhD from Göttingen, Germany
- Research: Finite Elements, Software, Parallelization, Fluid Flow

Summary

- What goes into a successful computational open source library
- Pitfalls, Tricks, etc.
- Why do we do this?
- What worked for us

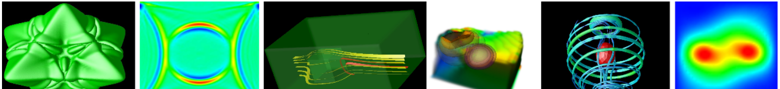
Warning: lots of opinions!

Wolfgang Bangerth and Timo Heister.
What makes computational open source software libraries successful?
Computational Science & Discovery, 6(1):015010, 2013.

Project: deal.II

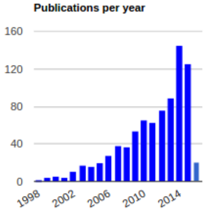
- A finite element library
- Open source project (LGPL 2.1+) written in C++, (15 years old)
- Features:
  - Various finite element of arbitrary order (CG, DG, RT, ...)
  - Manifold descriptions (local space descriptions using Charts)
  - Massively parallel computations (MPI and/or TBB)
  - Linear algebra: own/Trilinos/PETSc
  - Matrix free computations
  - Adaptive mesh refinement
  - Support for large number of optional packages

Wolfgang Bangerth, Timo Heister, Luca Heltai, Guido Kanschat, Martin Kronbichler, Matthias Maier, and Bruno Turcksin.
The dealii library, Version 8.3.
Archive of Numerical Software, 4(100), 2016.



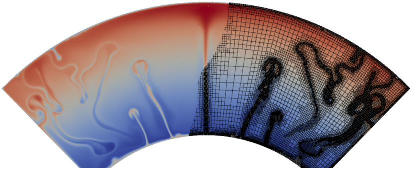
Continued: about deal.II

- Documentation:
  - 50+ tutorials with extensive descriptions, mathematical background, code explanations, results, and exercises
  - Extensive online documentation
  - Video lectures by W. Bangerth, 45+ videos each 30-60mins, 1000s of views
- widely used:
  - 800+ papers using and citing deal.II
  - Small core: 4 principal developers, +4 developers, many contributors
  - Used in teaching at Texas A&M, Clemson, Heidelberg, Italy, South Korea, India, . . .



Project: ASPECT

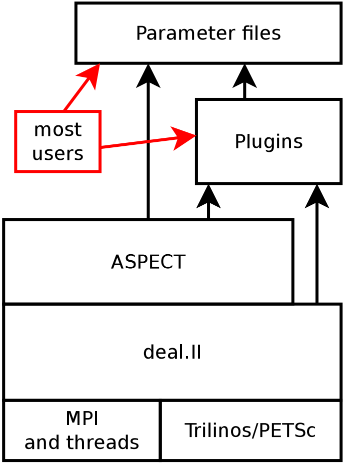
ASPECT = **A**dvanced **S**olver for **P**roblems in **E**arth's **C**onvec**T**ion



- Mantle convection using modern numerical methods
- Open source, C++, based on deal.II
- Available at: `http://aspect.dealii.org`
- In active development since 2011

Kronbichler, Heister and Bangerth.
High Accuracy Mantle Convection Simulation through Modern Numerical Methods.
Geophysical Journal International, 2012, 191, 12-29.

Structure of ASPECT



Problem setup, configuration

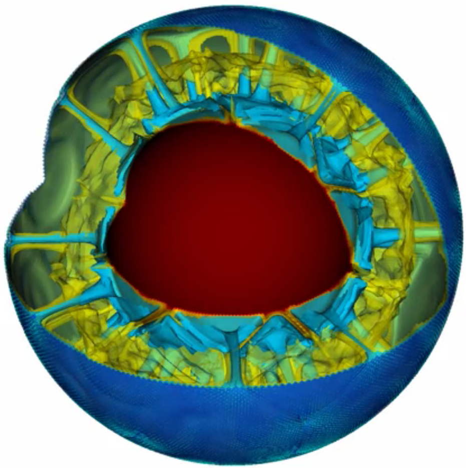
Materials, Geometries/Boundaries,
Adiabats, Postprocessing, Visualization,
Interfacing to other tools

Equations, Numerical schemes,
Framework

Finite Elements, AMR,
Parallel abstraction,
Postprocessing, Visualization

Parallelization, IO, linear algebra,
linear solvers

ASPECT: an example



https://www.youtube.com/watch?v=j63MkEc0RRw

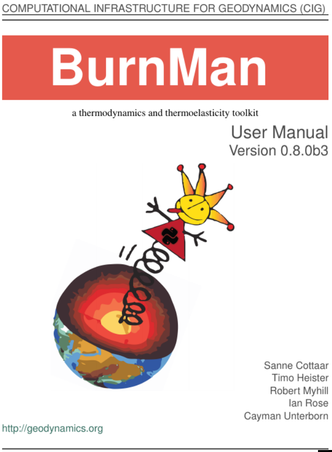
Limits of This Talk

- Libraries, not applications
  - API vs. input-driven or graphical interface
  - $\rightsquigarrow$ different requirements
- Computational science, not generic software:
  - Audience: scientists only, much smaller (100s, not millions)
  - Limited commercial options/motivation
  - Developers need advanced skillsets
  - Additional challenges: super computers, floating point math, etc.

Project: BurnMan

"a thermodynamics and thermoelasticity toolkit"

- Written in Python
- Based on numpy, scipy, . . .
- see `burnman.org`



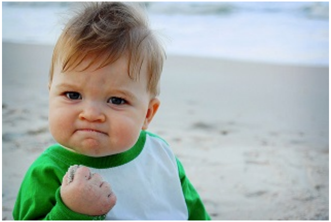
Why do we make software?

- Fun!
- Reasonable good at it
- Use as a tool for our work
- Use for teaching, students, etc.
- Enabling other people, maximize impact of my work
- Find jobs (worked for me!)
- Caveat: not optimal for academic career? (more on that later)

"Success"

Why do we want "success"? How to define it?

- maximize impact with my work
- want at least a self-sustaining user base
- better: maximize userbase
- (definition might be different for you)



en.wikipedia.org/w/index.php?curid=46523616

Why Open Source

- Useful software is a many year, many person effort
- Need to get contributors / successors to sustain project
- Spreading of knowledge is task of academic institutions
- Note: not just throwing over the fence, but open development

Primary Reasons for Success

1. Utility and Quality
2. Documentation
3. Community

1. Utility and Quality

- provide some utility (opinion: not the "best" wins)
- quality: "it works" and does what it promises to do
- easy to get/install
  - no registration required
  - if no instructions or complicated: lost user
    - no hand-editing of Makefiles, etc. (see MUMPS), use cmake
  - this is hard to do (Windows/OSX support, dependencies, . . .)
- Code is reasonably bug free
  - assertions, testing (see later)
  - fix bugs as quickly as possible (large audience helps, need encouragement to report)
  - provide help
- Summary: maximize audience by offering utility and quality

2. Documentation

- crucial to have extensive documentation
- what is documentation:
  - manual
  - tutorials
  - API reference
  - code comments (yes, inside the library)
  - installation instructions
  - Wiki, FAQs
- but also:
  - emails (mailing list archives, but also private)
  - lectures and recordings
  - videos
  - conversations

Levels of traditional documentation

High to low:

1. What is this library about?
2. Worked out examples in tutorial form
3. Modules: high-level, how to combine classes, differences between alternatives
4. Object/class level: meaning, how to use, limits
5. Function level: inputs, outputs, notes, pre/post conditions
6. Internal code comments (algorithm explanation, gotchas, TODOs, . . .)

Note: higher level information is crucial and often missing!

Best Practices

- Document on all levels, also cover installation, FAQ, how to contribute, etc.
- Different forms should complement each other
- Start early!
  - very time intensive
  - writing after the fact is unrealistic
  - consider: write documentation first/while developing
- Scalability
  - think about reach of documentation form
  - developer time does not scale (answering private emails)
  - mailing list answer vs. writing an FAQ entry
- Avoid out-of-date documentation
  - is worse than no documentation
  - use forms that are easy to update
  - hard: mailing list archives, printed books

Best Practices II

- Mailing lists:
  - even mailing lists don't scale well (strategy: update and link)
  - many are afraid or too lazy to ask
  - use as signal about documentation quality (where to improve)
  - we get many more high-level questions and bug reports, but also installation questions
- use a tool like doxygen to extract documentation
  - cross referenced, well-structured, easy to search
  - crucial: documentation and code at same location
  - easier to keep updated and in sync than a PDF manual

Demo

- Sphinx for Python
- Doxygen: `https://www.dealii.org/developer/doxygen/deal.II/classDataOut.html` and `https://github.com/dealii/dealii/blob/master/include/deal.II/numerics/data_out.h`
- Tutorial programs in deal.II: `https://www.dealii.org/8.4.0/doxygen/deal.II/Tutorial.html`
- show ASPECT manual
- show BurnMan pdf

An Example

NumPy/SciPy "SciPy documentation project":

- create documentation as community
- easy workflow for users to fix/add docs (later: reducing friction to contribute)
- wiki style, reviewed by devs and commited
- Note: after the fact is hard!

Do it right from the start and concurrently

- "write later" is much more difficult
- many years of work (deal.II)
- documentation first: "design by contract"

S. J. Van der Walt.
The SciPy documentation project (technical overview).
In *SciPy Conference–Pasadena, CA, August 19-24, 2008.*, page 27.

3. Community

- Three groups:
  1. maintainers: run the project, testing, fixing bugs, politics, . . .
  2. contributors (from small fixes to main developers)
  3. users
- People move dynamically up and down or leave completely
- communities don't just happen, need to be engineered:
  This is hard work!

⇝ requires "humility, respect, and trust" as said in:

B. Fitzpatrick and B. Collins-Sussman.
Team Geek: A Software Developer's Guide to Working Well with Others.
O'Reilly Media, Incorporated, 2012.

How to reduce friction to have people move up?

"Lowering the Bar"

- make it easy to submit patches, fixes, documentation, bug reports
- encourage contributions
- be friendly, open, social
- provide, help with, highlight incentives:
  - intrinsic:
    - do not need to maintain your fixes (I got into it this way)
    - get improvements from others
  - provide:
    - appreciate contributions, "your work matters" (even if tiny)
    - free t-shirts: scipy documentation project
  - advantages:
    - get known, "street cred"
    - involvments in research projects
    - invitations to speak
    - publications, citations
    - jobs
    - funding

Accepting contributions

- don't turn people away
- but help: requires lots of patience and work
- don't be territorial
- develop/discuss things in public (+welcoming atmosphere)
  $\rightsquigarrow$ "bazaar" model

E. S. Raymond.
The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.
O'Reilly Media, 1999.

Example: deal.II

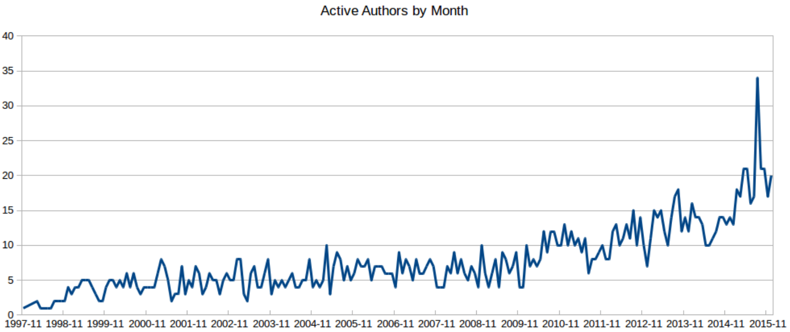
Old:

- public subversion repository
- need account for write access
- is blank check, no review process!
- many “oops” commits

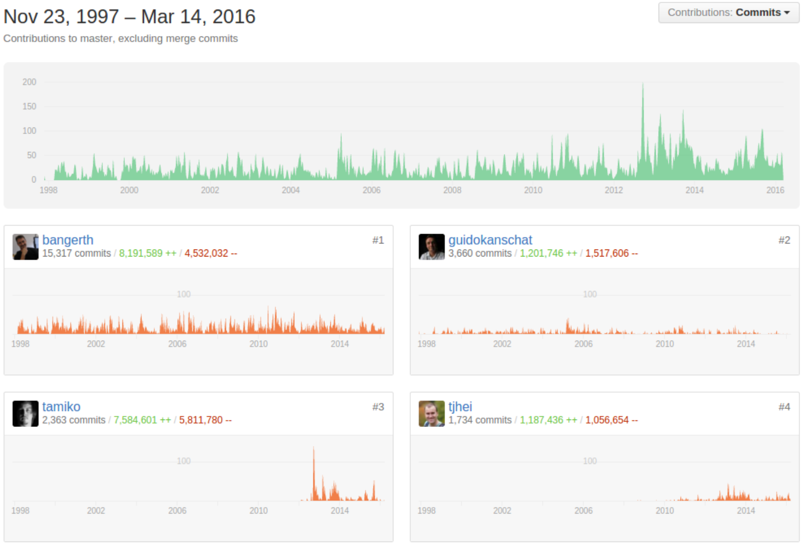
New:

- git hosted on github.com
- Everything by pull request only, also for main devs
- Very little friction to contribute!
- Run checks with continuous integration
- cleaner project history
- Transformed our project: easier contribution, better quality, review between devs is great

Monthly Contributors



Commits to deal.II



https://github.com/dealii/dealii/graphs/contributors

Secondary Reasons

1. Timing
   - projects need to serve a market
   - success: easier with little competition
2. Make a project usable
   - catalog common use cases; example codes (deal.II / PETSc)
   - asserts()
   - backward compatiblity (TriBits model), linux ("don't break userspace")
3. Maintainability
   - standardize, modularize, do it right
   - verify correctness (automatic tests)
4. Software license
   - license change is hard/impossible
   - choice complicated, but: do not exclude people
5. Marketing

Testing

- Testing is essential
- Correctness of numerical methods (convergence rates, etc.)
- Also normal testing of functionality
- my opinion: untested code is broken
- Required to be able to maintain a large project

Types of Tests

- Unit testing:
  - Test single function/class/functionality for correctness
  - Example: norm of a vector gives what you expect
- Integration testing:
  - Interaction of different modules
  - Complete examples

Testing in BurnMan

- Story: Python needs close to 100% test coverage, run all examples, etc. because it is dynamically typed
- Testing framework: unittest, show demo!
  see
  `https://docs.python.org/2/library/unittest.html`
- Example: tests/test_averaging.py
- Handwritten tool to compare example output
- Example, somebody changed argument order:
  `https://github.com/geodynamics/burnman/pull/174/files`
- `http://burnman.org/coverage/`

Caveats for scientific software

- Typically: floating point computations
- Need comparisons with epsilon! (demo)
- Results can depend on architecture, optimization flags, compilers, etc. :-(
- Another issue: number of iterations of iterative solver can change
- numdiff
- Problem: many dependencies, for example in deal.II: c++11, c++14, MPI?, threading?, PETSc?, Trilinos, UMFPACK, etc.

Testing in deal.II

- Testsuite with 7000+ tests, hand-written based on ctest, numdiff
- http://cdash.kyomu.43-1.org/index.php?project=deal.II
- slow turn-around, 2+ hours on fast machine
- ideally: split into fast unit tests, slow integration tests
- split into modules: grid, fe, hp, etc.
- show example: numerics/project_01

Testing in ASPECT

- Run application based on parameter files
- Check output
- Additional code via plugins
- Example: tests/solver_history.cc

Continuous Integration

- Several machines run tests after every commit with different configurations
- Autmatic testing of pull requests: `https://github.com/geodynamics/burnman/pull/243` or `https://github.com/geodynamics/aspect/pull/790`
- Partly based on docker images (work in progress)
- Demo:

Software and your academic career

- making software:
  - fewer papers and/or smaller impact
  - huge time sink (project management, testing, bugfixing, support, . . .)
- In academia:
  - writing software counts very little in promotion
  - own project: high risk – doubtful reward
  - getting credit is hard (especially when joining an existing project)
- but:
  - many oportunities to cooperate
  - networking
  - jobs

Conclusions

- Primary reasons for success:
  1. Provide utility and quality (testing!)
  2. Be smart about and provide documentation on many levels
  3. Engineering a community
- This is hard work and requires a big time investment
- Social skills not optional!

Thanks for your attention!

Bibliography

Wolfgang Bangerth and Timo Heister.
What makes computational open source software libraries successful?
Computational Science & Discovery, 6(1):015010, 2013.

W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, and B. Turcksin.
The `deal.II` library, version 8.3.
Archive of Numerical Software, 4(100):1–11, 2016.

S. J. Van der Walt.
The SciPy documentation project (technical overview).
In *SciPy Conference–Pasadena, CA, August 19-24, 2008.*, page 27.

B. Fitzpatrick and B. Collins-Sussman.
Team Geek: A Software Developer's Guide to Working Well with Others.
O'Reilly Media, Incorporated, 2012.

E. S. Raymond.
The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.
O'Reilly Media, 1999.