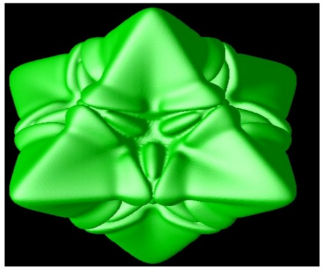
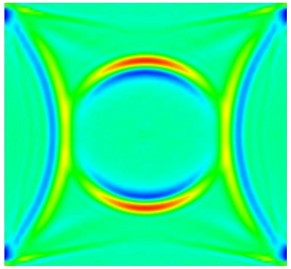
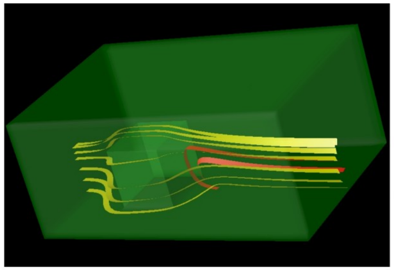
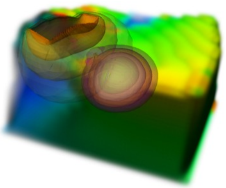
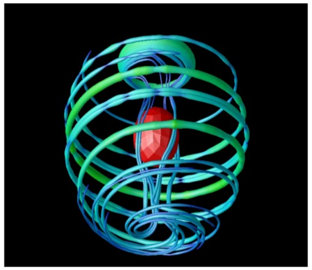
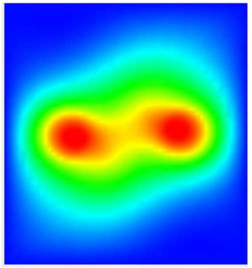
What I learned from 20 years of leading open source projects

Wolfgang Bangerth
Colorado State University

In collaboration with many many others around the world.

Where I'm coming from

My open source, scientific software experience:

- Started the deal.II project in 1997:
  - now 1.5M lines of C++
  - library that provides general finite element support
  - currently 11 "principal developers"
  - ~300 contributors over the years
  - 200+ papers/year that use it

- Started the ASPECT project in 2011:
  - now 150,000 lines of C++
  - simulates convection in the Earth mantle, deformation of the lithosphere
  - currently 9 "principal developers"
  - ~100 contributors over the years

What I learned

Building
– long-term sustainable software
– successful software communities
comes down to this:

It's not about being a "good programmer".
It's really all about (limitations of) people.

Specifically, dealing with human limitations to:
1) learn and work with *complex systems*
2) work with *people* in *complex organizations*

(Humans dealing with) Technical complexity

Managing technical complexity

There is a fundamental difference between
- **where projects start, and**
- **where projects end up.**

Using deal.II as an example. In the beginning:

- Started 1997 by myself: a single grad student
- Wrote 20k lines of code in year 1
- Acquired 2 co-authors in the same lab
- After 2 years:
  - 3 people
  - 100k lines of code
  - no external dependencies
  - no external users
- Website "because we can" in 2000

- This is probably quite typical of many scientific codes in academia and the national labs

Managing technical complexity

There is a fundamental difference between

– where projects start, and

– where projects end up.

Using deal.II as an example. Now:

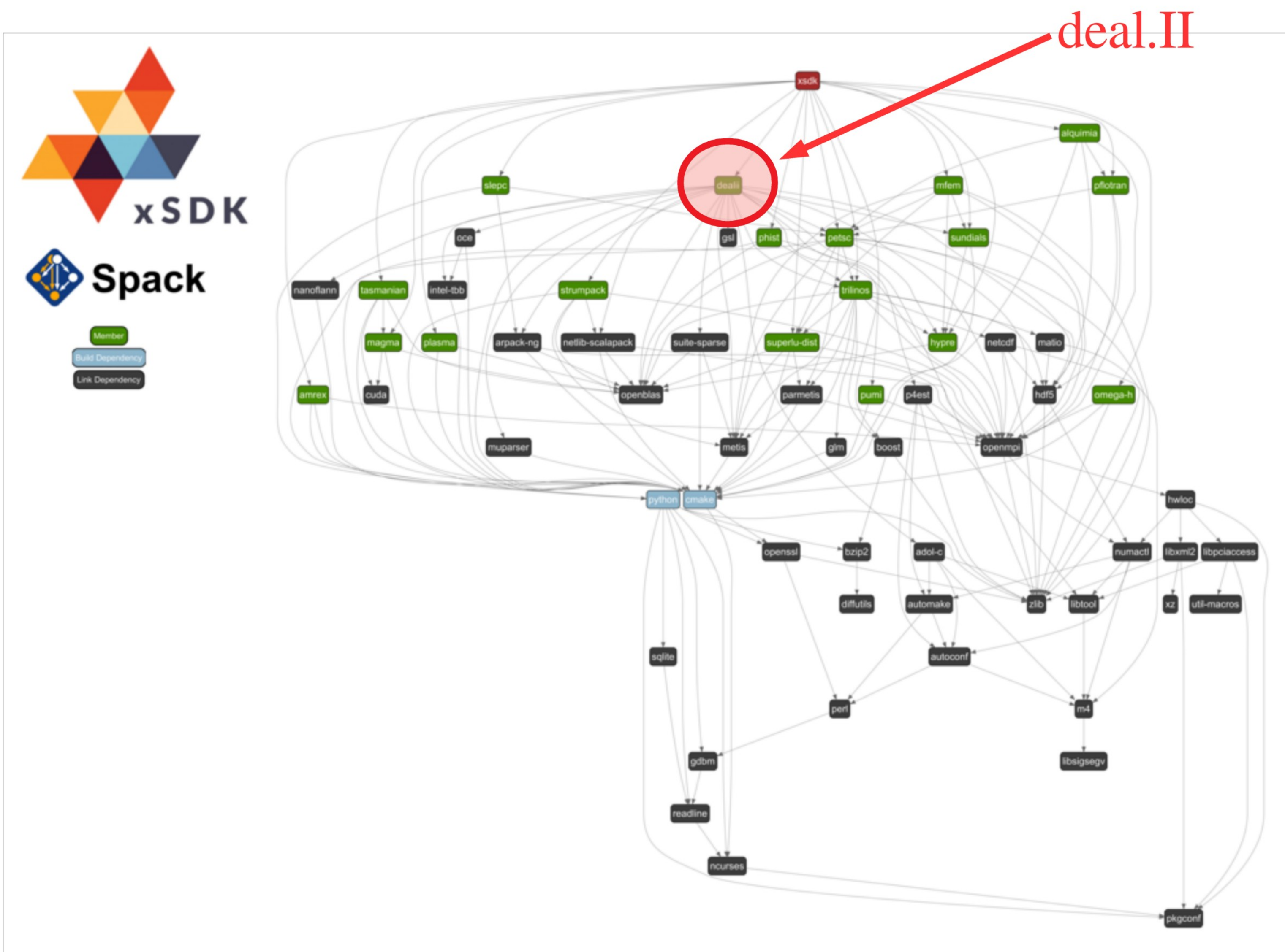
- 1.5M lines of code, grows by 40k lines/year
- 11 principal developers
- 300 contributing authors
- 1200 people on the mailing list

- Used in many individual research projects

- Uses many other packages

Managing technical complexity

Example: deal.II in the context of the xSDK collection



xSDK 0.4 dependency graph

Managing technical complexity

What this means:

- Scientific software today is no longer a "collection of sub-routines" (like BLAS or LAPACK originally were)

- Packages form an "interconnected web" where each builds on others

- Many packages are themselves composed of "modules":
  – deal.II itself
  – Trilinos
  – PETSc

Managing technical complexity

Why are things this way?

- Because no single developer can *know* this much
- Because no single user can *learn* this much

Managing technical complexity

There are costs associated with this:

- Installation complexity
- Different styles of coding, documenting, teaching
- Each dependency is in itself a moving target
- Which developer knows which dependency, and how do we make sure that knowledge is preserved?
  (→ what is the project's “bus factor”?)

Managing technical complexity

There are costs associated with this:

- Installation complexity
- Different styles of coding, documenting, teaching
- Each dependency is in itself a moving target
- Which developer knows which dependency, and how do we make sure that knowledge is preserved?
(→ what is the project's "bus factor"?)

From Wikipedia: The "bus factor" is the minimum number of team members that have to suddenly disappear from a project before the project stalls due to lack of knowledgeable or competent personnel.

Studies conducted in 2015 and 2016 calculated the bus/truck factor of 133 popular GitHub projects. The results show that most of the systems have a small bus factor (65% have bus factor ≤ 2) and the value is greater than 10 for less than 10% of the systems.

Managing technical complexity

How do we deal with this:

- Poorly

- We talk about “software design”, which is as much *art and craft* as it is *science* – because we don’t really understand it

Managing technical complexity

How do we deal with this:

- Poorly
- We talk about "software design", which is as much *art and craft* as it is *science* – because we don't really understand it
- We learn about human limitations – *specifically that human time is much more valuable than computer time*:

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."
(Donald Knuth).

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
(Martin Fowler)

Managing technical complexity

But, we also have good technical solutions for human limitations:

– We forget → we use autocomplete
– We make mistakes → we write test suites
→ we peer review codes
– We break code → we use continuous integration
– It's repetitive/boring → we use package managers
– Can't keep things in sync → we use in-code documentation

Examples of tools:
– autocomplete: Eclipse, Visual Studio, Qt Creator
– tests: ctest, google test, …
– code review: github
– continuous integr.: jenkins, github actions
– package managers: cmake, spack, linux repositories
– documentation: doxygen

Managing technical complexity

There are also many collections of best practices:

– How to write documentation
– How to write teaching materials
– How to onboard new people
– Coding styles, software patterns, naming conventions, …

Examples:

– Code Complete (Steve McConnell)

– Design Patterns (Gamma et al., also several others)
– Producing Open Source Software (Karl Fogel)

– Look at how other projects write documentation, tutorials, manuals
– Check out BSSw

Managing technical complexity

Summary:

- Building workable scientific software packages has really become about *managing complexity* and *human limitations around complex systems*

- A large amount of time and thought goes into:
  – breaking things into manageable chunks
  – writing documentation
  – writing teaching materials
  – building *infrastructure*

- The difficulty is not with the technical tools, but with the human ability to learn/understand/manage *complex systems*

(Humans dealing with) Human complexity

Managing people

Scientific software has some unique aspects:

- Often part of *research* projects – there are no standard solutions one can look up
- Often built by *temporary employees*:
  – graduate students
  – postdocs
- Often built by *unpaid volunteers*
- Generally built by people without formal C.S. education

This brings some interesting human challenges with it!

Managing people

Regarding temporary employees:

- A lot of responsibility on a few senior leaders:
  – constant onboarding of new contributors
  – a lot of teaching/mentoring
  – importance of code review

- Contributing authors do not feel the same level of "ownership", have other priorities

- Leadership needs to make up for lack of experience/quality

Managing people

Regarding volunteers (1):

- Development directions are sometimes unclear: Functionality grows by what user-developers need, not what the project wants
  → It's difficult to establish "road maps"

- Volunteers can't be treated like employees

Managing people

Regarding volunteers (2):

- A lot of responsibility on a few senior leaders:
  – constant onboarding of new contributors
  – a lot of teaching/mentoring
  – importance of code review

- Leadership needs to provide key infrastructure improvements

- Leadership needs to work on growing the pool of volunteers

Managing people

Regarding the "principal developers"? (1)

- Have to fill many roles:
  – manage technical infrastructure
  – maintain "institutional knowledge"
  – onboard and mentor contributors
  – review patches

  – work on foundational functionality

Managing people

Regarding the “principal developers”? (2)

- Manage their own careers with all of the other demands:
  – as faculty
  – as permanent technical staff

- Obtain funding for their work
- Document the work that is being done

Problem: There are a lot of other demands on principal developers’ time.

But: This is also an awesome job if you enjoy working with people!

Some recommendations

Recommendations

Technical aspects:

- Use the tools that are out there:
  – Eclipse/Visual Studio instead of emacs/vi
  – cmake instead of homegrown installation scripts
  – doxygen
  – github
- Teach the use of these tools

- Read up on best practices (e.g. “Code Complete”, books on software design patterns)
- Teach these best practices

Recommendations

Human aspects:

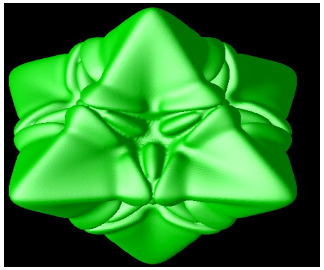
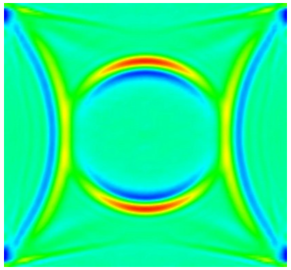
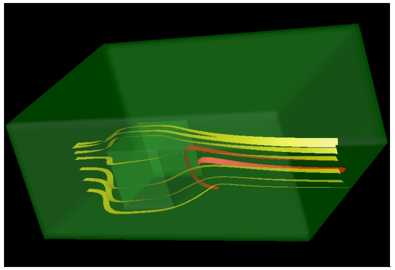
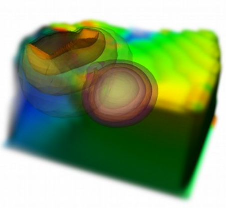
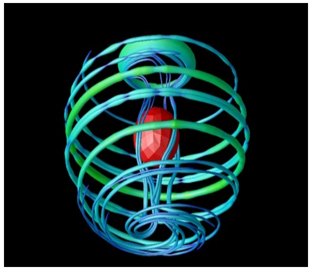
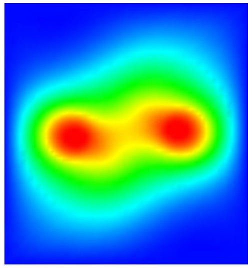
- Commit to a project only if that is compatible with career aspirations

- If you lead a project:
  – Understand where people are coming from
  – Spend the time mentoring
  – Be welcoming and generous with praise

Conclusions

Scientific software packages have become so large that they are *fundamentally different* from small academic codes:

- Managing the limits of humans to understand complexity is the key technical challenge
- Managing the humans in these projects
  – with different skills
  – with different motivations
  is the key human challenge.

More information:

- Wolfgang Bangerth:
  "Leading a Scientific Software Project: It's All Personal"

  Better scientific software (BSSw) blog post, 2019

  https://bssw.io/blog_posts/leading-a-scientific-software-project-it-s-all-personal

- Wolfgang Bangerth and Timo Heister:
  "What makes computational open source software libraries successful?"

  Computational Science & Discovery 6 (2013), 015010

  doi:10.1088/1749-4699/6/1/015010