WHAT ALL CODES SHOULD DO: OVERVIEW OF BEST PRACTICES IN HPC SOFTWARE DEVELOPMENT

MAY 4, 2016
ANSHU DUBEY
MATHEMATICS AND COMPUTER SCIENCE DIVISION
ARGONNE NATIONAL LABORATORY

Webinar Series: Collaboration among the IDEAS Scientific Software Productivity Project, Argonne Leadership Computing Facility, National Energy Research Scientific Computing Center, and Oak Ridge Leadership Computing Facility

OBJECTIVES OF THE SERIES

- To bring knowledge of **useful** software engineering practices to HPC scientific code developers
  - Not to prescribe any set of practices as **must use**
    - Be informative about practices that have worked for some projects
    - Emphasis on adoption of practices that help productivity rather than put unsustainable burden
    - Customization as needed – based on information made available
- We will do it through examples and case studies
  - References for available resources
  - Suggestions for further reading

WEBINARS IN THE SERIES

What All Codes Should Do: Overview of Best Practices in HPC Software Development – May 4, 2016

- ❑ Overview of the series and a few topics that won't have a webinar of their own
- ❑ Motivation – why should a computational scientist worry about software process ?
- ❑ Practices that many codes have adopted and found useful
- ❑ Customization examples
- ❑ Community codes – how are they helpful and how to build a community

WEBINARS IN THE SERIES

Developing, Configuring, Building, and Deploying HPC Software – May 18, 2016

- ❑ Tools and best practices for configuring and building
- ❑ Software design and development
- ❑ Helpful hints about developer productivity through use of development environments
- ❑ Customizing for the project needs

WEBINARS IN THE SERIES

Distributed Version Control and Continuous Integration Testing – June 2, 2016

- ❑Using Git for version control
- ❑GitHub as a development platform
- ❑Pull requests: a controlled change process
- ❑Mechanisms for Communicating and Tracking Progress
- ❑Continuous Integration with Travis CI

WEBINARS IN THE SERIES

Testing and Documenting your Code – June 15, 2016

- ❑ How much to test and document
  - ❑ Evaluating the team needs and the extent of testing that is helpful rather than burdensome
- ❑ Granularity of testing
  - ❑ How to leverage testing granularity to pinpoint failure
- ❑ Code coverage
  - ❑ Methodology for maximizing code coverage
- ❑ Getting buy-in from the development team

WEBINARS IN THE SERIES

Next three, details will come later

How the HPC Environment is Different from the Desktop (and Why) – *Planned for the week of June 27, 2016*

Basic Performance Analysis and Optimization – *Planned for the week of July 11, 2016*

Best Practices for I/O on HPC Systems – *Planned for the week of July 25, 2016*

OUTLINE

- ❑ **Motivation**
- ❑ Customization
- ❑ Best Practices
- ❑ Community Development

HEROIC PROGRAMMING

Usually a pejorative term, is used to describe the expenditure of huge amounts of (coding) effort by talented people to overcome shortcomings in process, project management, scheduling, architecture or any other shortfalls in the execution of a software development project in order to complete it. Heroic Programming is often the only course of action left when poor planning, insufficient funds, and impractical schedules leave a project stranded and unlikely to complete successfully.

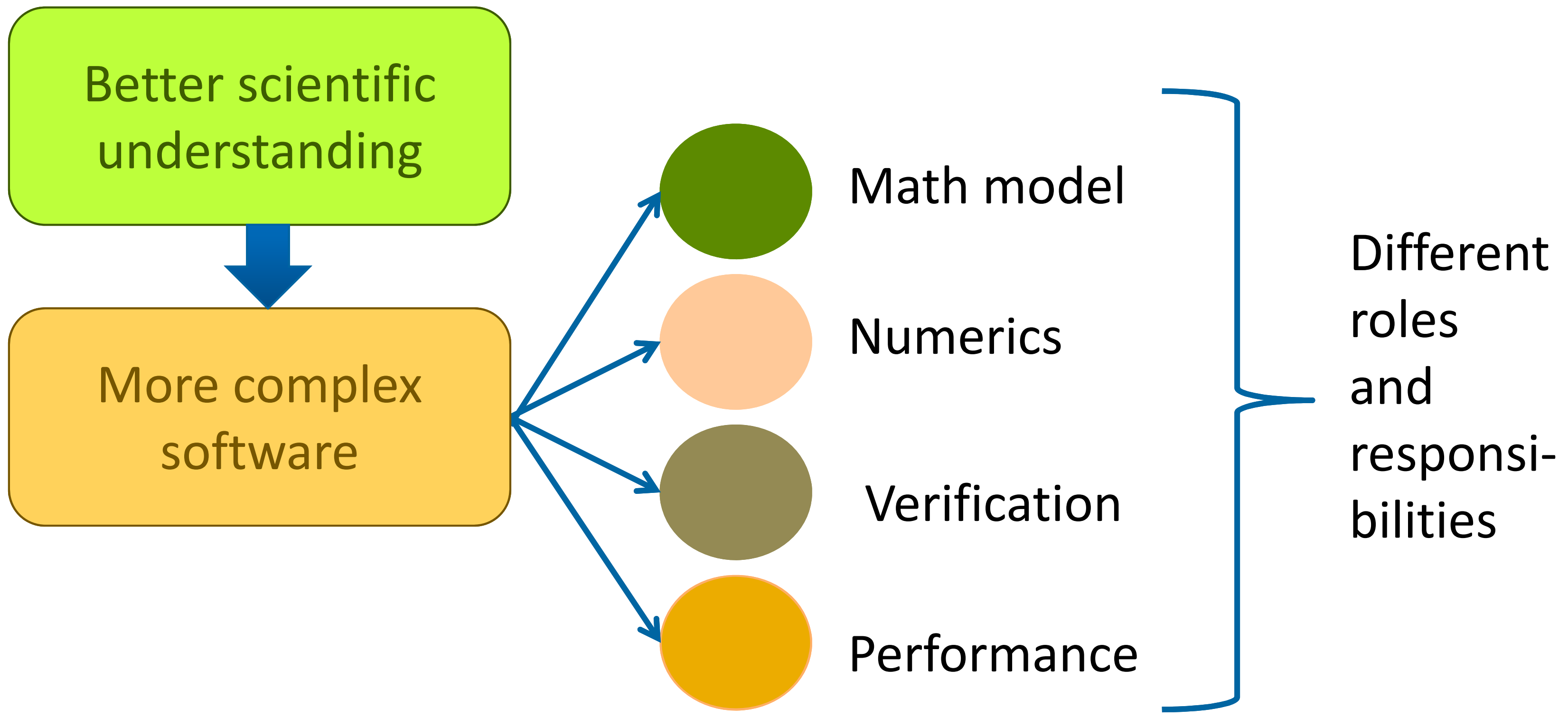
From http://c2.com/cgi/wiki?HeroicProgramming

Science teams often resemble heroic programming

Many do not see anything wrong with that approach

WHAT IS WRONG WITH HEROIC PROGRAMMING

Scientific results that could be obtained with heroic programming have run their course, because:


Better scientific understanding
More complex software
Math model
Numerics
Verification
Performance
Different roles and responsi-bilities

It is not possible for a single person to take on all these roles

IN EXTREME-SCALE SCIENCE

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

OTHER REASONS

Accretion leads to unmanageable software

- ❑ Increases cost of maintenance
- ❑ Parts of software may become unusable over time
- ❑ Inadequately verified software produces questionable results
- ❑ Increases ramp-on time for new developers
- ❑ Reduces software and science productivity due to technical debt

consequence of choices – quick and dirty incurs technical debt, collects interest which means more effort required to add features.

"... it seems likely that significant software contributions to existing scientific software projects are not likely to be rewarded through the traditional reputation economy of science. Together these factors provide a reason to expect the over-production of independent scientific software packages, and the underproduction of collaborative projects in which later academics build on the work of earlier ones."

Howison & Herbsleb (2011)

OUTLINE

- ❑ Motivation
- ❑ **Customization**
- ❑ Best Practices
- ❑ Community Development

SURVEY OF IDEAS USE-CASES

IDEAS scientific software productivity project: www.ideas-productivity.org

- ❑ Five application codes and four numerical libraries
- ❑ All use version control, and all but one use distributed version control
- ❑ Builds are evenly divided between GNU make and CMake
- ❑ All provide documentation with some form of user's guide, many use automated documentation generation tools
- ❑ All have testing in some form, a couple do manual regression testing, the rest are automated
- ❑ Roughly half make use of unit testing explicitly
- ❑ Majority are publicly available

SUMMARY FROM COMMUNITY CODES WORKSHOP (2012)

http://flash.uchicago.edu/cc2012/

- ❑ Codes – FLASH, Cactus, Enzo, ESMF, Lattice QCD code-suite, AMBER, Chombo, and yt
- ❑ Software architecture is almost always in the form of composable components
  - ❑ Need for extensibility
- ❑ All codes have rigorous auditing processes in place
- ❑ Gatekeeping for contributions, though models are different
- ❑ All codes have wide user communities, and the communities benefit from a common highly exercised code base

CHALLENGES

Technical

- ❑ All parts of the cycle can be under research
- ❑ Requirements change throughout the lifecycle as knowledge grows
- ❑ Verification complicated by floating point representation
- ❑ Real world is messy, so is the software

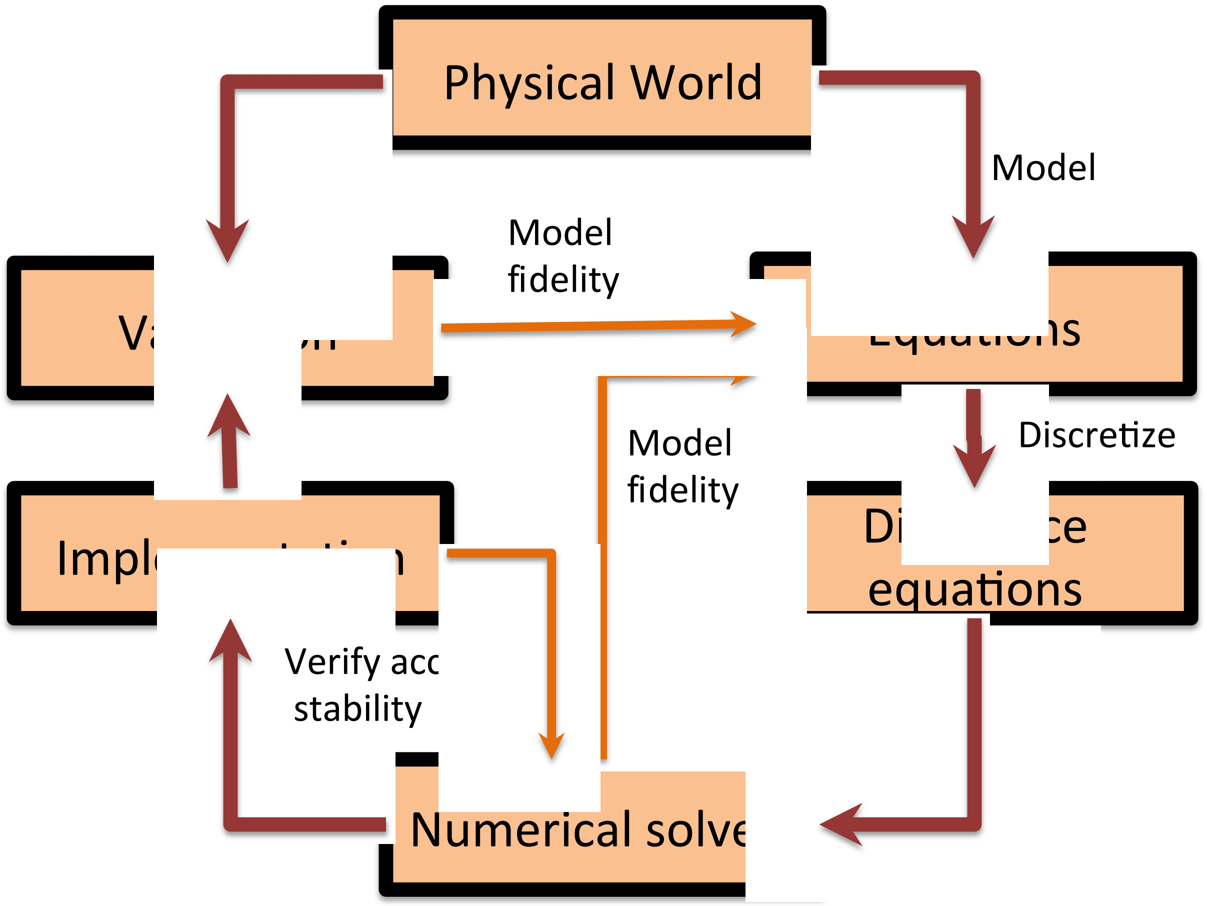
Sociological

- ❑ Competing priorities and incentives
- ❑ Limited resources
- ❑ Perception of overhead without benefit
- ❑ Need for interdisciplinary interactions

CUSTOMIZATIONS

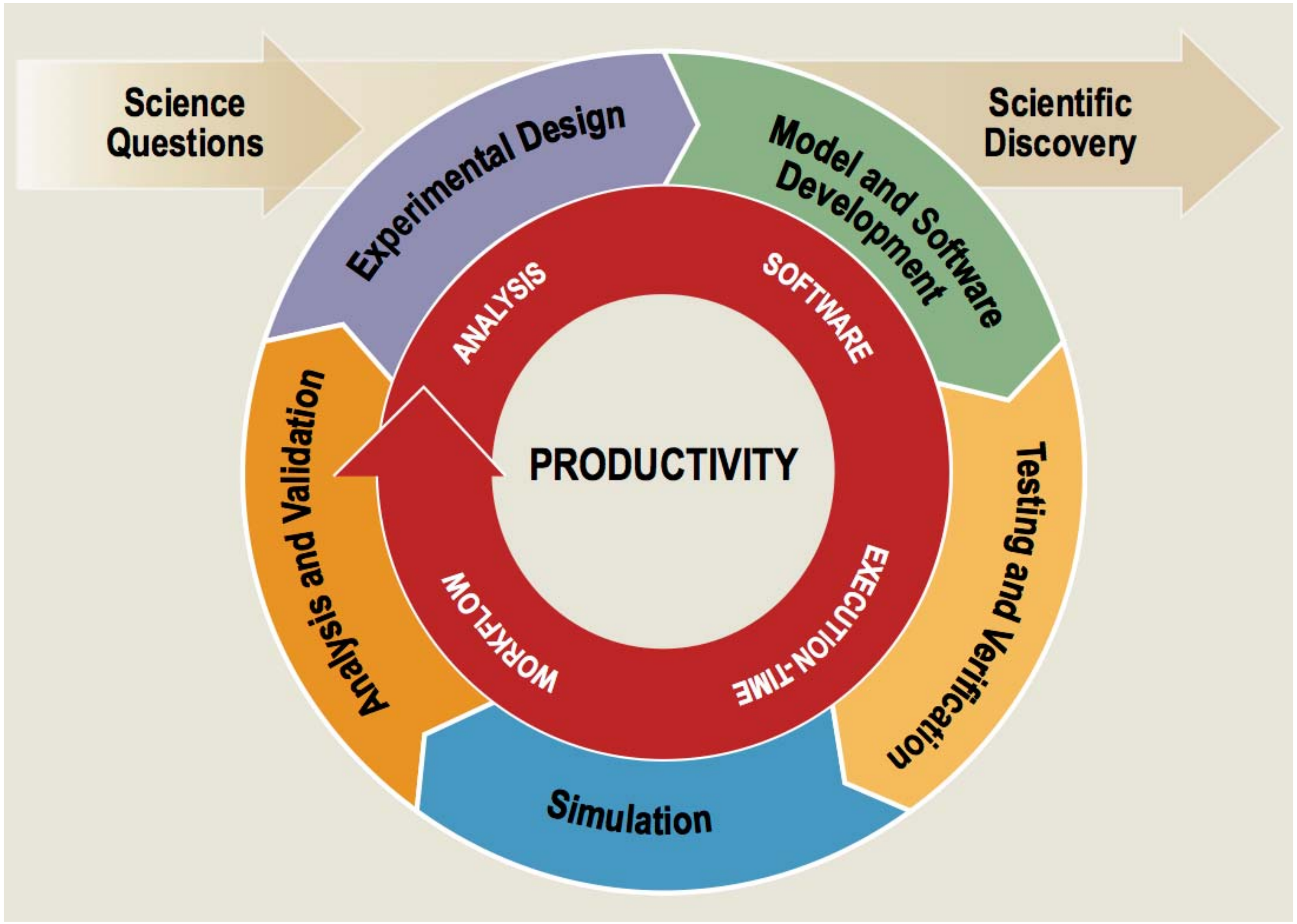
- ❑ Testing does not follow specific methods as understood by the software engineering research community
  - ❑ The extent and granularity reflective of project priorities and team size
  - ❑ Larger teams have more formalization
- ❑ Lifecycle – closer to the figure in the next slide
- ❑ Development model
  - ❑ Mostly ad-hoc, some are close to agile model, but none follows it explicitly
  - ❑ Much more responsive to the needs of the lifecycle

LIFECYCLE



- ❑ Modeling
  - ❑ Approximations
  - ❑ Discretizations
  - ❑ Numerics
    - ❑ Convergence
    - ❑ Stability
- ❑ Implementation
  - ❑ Verification
    - ❑ Expected behavior
  - ❑ Validation
    - ❑ Experiment/observation

SOFTWARE PRODUCTIVITY CYCLE



http://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf

OUTLINE

- ❑ Motivation
- ❑ Customization
- ❑ **Best Practices**
- ❑ Community Development

SOFTWARE PROCESS

Baseline

- ❑ Invest in extensible code design
- ❑ Use version control and automated testing
- ❑ Institute a rigorous verification and validation regime
- ❑ Define coding and testing standards
- ❑ Clear and well defined policies for
  - ❑ Auditing and maintenance
  - ❑ Distribution and contribution
  - ❑ Documentation

Desirable

- ❑ Provenance and reproducibility
- ❑ Lifecycle management
- ❑ Open development and frequent releases

Many of these practices will be covered in much greater detail later in the series

A USEFUL RESOURCE

https://ideas-productivity.org/resources/howtos/

- **‘What Is’ docs**: 2-page characterizations of important topics for SW projects in computational science & engineering (CSE)
- **‘How To’ docs**: brief sketch of best practices
  - Emphasis on ``bite-sized'' topics enables CSE software teams to consider improvements at a small but impactful scale
- We welcome feedback from the community to help make these documents more useful

OTHER RESOURCES

http://www.software.ac.uk/

http://software-carpentry.org/

http://flash.uchicago.edu/cc2012/

http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=4375255

http://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6171147

CONSIDERATIONS FOR CUSTOMIZATION

- ❑ There is no “all or none”
- ❑ Focus on improving productivity rather than purity of process
  - ❑ There is danger of being too dismissive too soon
  - ❑ Examine options with as little bias as possible
- ❑ Fine balance between getting a buy-in from the team and imposing process on them
  - ❑ Many skeptics get converted when they see the benefit
  - ❑ First reaction usually is resistance to change and suspicion of new processes

INTERDISCIPLINARY INTERACTIONS

A partnership model that works

- ❑Science users treat the code as a research instrument that needs its own research
- ❑Developers and computer scientists interested in a product and the science being done with the code
  - ❑Helps to have people with multidisciplinary training
- ❑Comparable resources and autonomy for the developers
  - ❑And recognition of their intellectual contribution to scientific discovery
- ❑Careful balance between long term and short term objectives

OUTLINE

- ❑ Motivation
- ❑ Customization
- ❑ Best Practices
- ❑ **Community Development**

WHY COMMUNITY CODES ?

- Scientists can focus on developing for their algorithmic needs instead of getting bogged down by the infrastructural development
- Graduate students do not start developing codes from scratch
  - Look at the available public codes and converge on the ones that most meet their needs
  - Look at the effort of customization for their purposes
  - Select the public code, and build upon it as they need

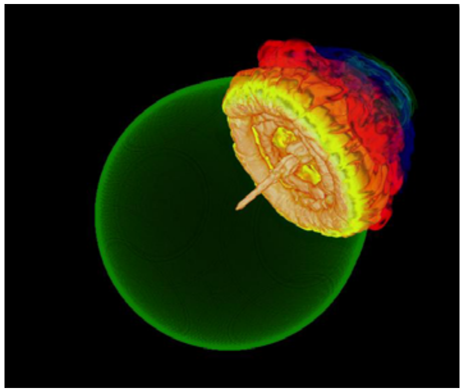
Important to remember that they still need to understand the components developed by others that they are using, they just don't have to actually develop everything themselves. And this is particularly true of pesky detailed infrastructure/solvers that are too well understood to have any research component, but are time consuming to implement right

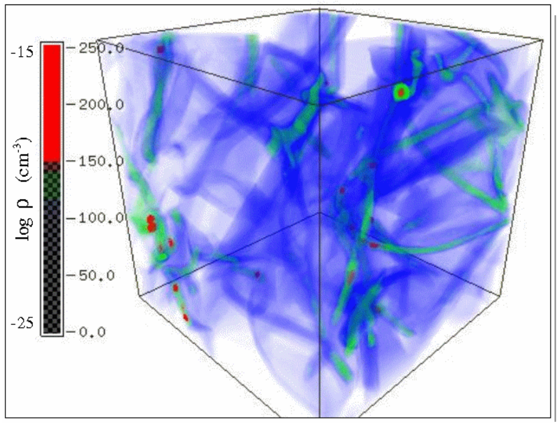
- Researchers can build upon work of others and get further faster, instead of reinventing the wheel
  - Code component re-use
  - No need to become an expert in every numerical technique
- More reliable results because of more stress tested code
  - Enough eyes looking at the code will find any errors faster
  - New implementations take several years to iron out the bugs and deficiencies
  - Different users use the code in different ways and stress it in different ways
- Open-source science results in more reproducible results
  - Generally good for the credibility

THE ASTROPHYSICS COMMUNITY

- ❑ Had an early culture of releasing research software starting in the early eighties
  - ❑ N-body codes for many-body gravitational interactions
    - ❑ Nbody_x went from Nbody_1 to Nbody_6
    - ❑ Barnes and Hut tree code
  - ❑ Hydrodymanics with ZEUS-2D, and later ZEUS-3D
  - ❑ SPH codes such as Hydra and Gadget
- ❑ Over time public codes became more sophisticated
  - ❑ AMR appeared in FLASH is early 2000
  - ❑ Shock-capturing MHD and radiation hydro also started to appear

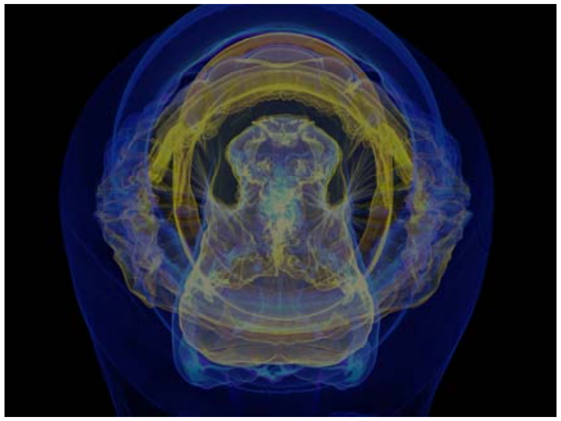
ASTROPHYSICS NEEDS MULTI-PHYSICS AND MULTI-SCALE

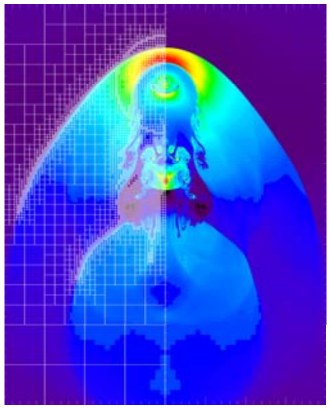

Type !a Supernova

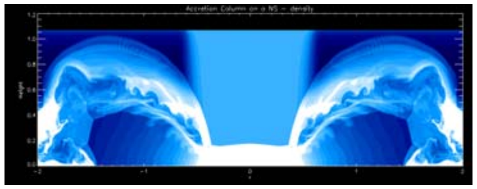


Gravitational collapse/Jeans instability

- ❑ Mesh methods: Explicit (gas dynamics), semi-Implicit (gravitational potential), and implicit (radiation)
- ❑ Particle methods: tracers, massive, charged
- ❑ Point-wise calculations: EOS, source terms
- ❑ AMR for data and computation compression


Galaxy Cluster Merger


Intracluster interactions


Shortly: Relativistic accretion onto NS

Developing and maintaining such complex codes is beyond the resources of capabilities of individuals or even small groups: Community codes are the solution

WHAT ABOUT OTHER COMMUNITIES ?

- ❑Community/open-source approach more common in areas which need multi-physics and/or multi-scale
- ❑A visionary sees the benefit of software re-use and releases the code
- ❑Sophistication in modeling advances more rapidly in such communities
- ❑Others keep their software close for perceived competitive advantage
  - ❑Repeated re-invention of wheel
  - ❑General advancement of model fidelity slower

Let us examine what does it take to build a community code

COMMUNITY BUILDING

- ❑ Popularizing the code alone does not build a community
- ❑ Neither does customizability – different users want different capabilities

So what does it take ?

- ❑ Enabling contributions from users and providing support for them
- ❑ Including policy provisions for balancing the IP protection with open source needs
- ❑ Relaxed distribution policies – giving collective ownership to groups of users so they can modify the code and share among themselves as long as they have the license

More inclusivity => greater success in community building

An investment in robust and extensible infrastructure, and a strong culture of user support is a pre-requisite

EXAMPLES : FLASH

- ❑ Under sustained funding from the ASC alliance program
- ❑ One of the expected outcomes was a public code
  - ❑ Use the same code for many different applications
    - ❑ All target applications were for reactive flows
- ❑ Diverging camps from the beginning
  - ❑ Camp 1: Produce a well architected modular code
  - ❑ Camp 2: Let’s build what can be used for science soon
- ❑ Both goals hard to meet in the near term
- ❑ Two parallel development paths started
  - ❑ Not enough resources to sustain both
  - ❑ Camp 2 won out
- ❑ Took three iterations of code refactoring to get robust framework built

FLASH'S COMMUNITY

- ❑ Originally designed for thermo-nuclear flashes
    - ❑ Expanded to include N-body capabilities through particles
    - ❑ Over the years many other physics capabilities got added
- ❑ Now serves many communities in Astrophysics, Cosmology, Solar physics, HEDP and CFD/FSI
    - ❑ Over 1100 publications in a self reporting database
- ❑ Very little modification to the basic infrastructure needed to accommodate these capabilities
- ❑ Additions typically prove to be synergistic for all the communities
- ❑ Follows a "Cathedral" model, code managed by the Flash Center with gatekeeping for external contributions

(http://www.catb.org/esr/writings/cathedral-bazaar/)

COMMUNITY BUILDING

- ❑ Took several years
- ❑ Started with collaborations with the Center scientists
- ❑ Alumni of the center took the culture and the code with them
  - ❑ Their students and post-docs adopted the code
- ❑ Tutorials on-site and at scientific conferences to promote
  - ❑ Tutorials had hands-on sessions and help for user's specific problems
- ❑ Easy customizability built into the infrastructure helped
  - ❑ As did the included ready to run examples
- ❑ Increasing capabilities enable tackling more complex and higher fidelity modeling

The greatest impact in popularizing the code though was relative ease in getting started, easy customizability and reliability

ENZO : TRANSITIONED FROM CLOSE TO OPEN SOURCE

- Started as a closed code
  - From 1996-2003
- First public release in March 2004
  - Mostly cathedral model
- Has now moved very close to a “bazaar” model
  - 25 contributors (~12 active developers) at >10 institutions
  - ~200 people on enzo-users mailing list (~50% active?)
  - Financial support from NSF (AST, OCI, PHY), NASA, and DOE

Complementary community: **yt** (http://yt-project.org)

DEVELOPMENT MODEL

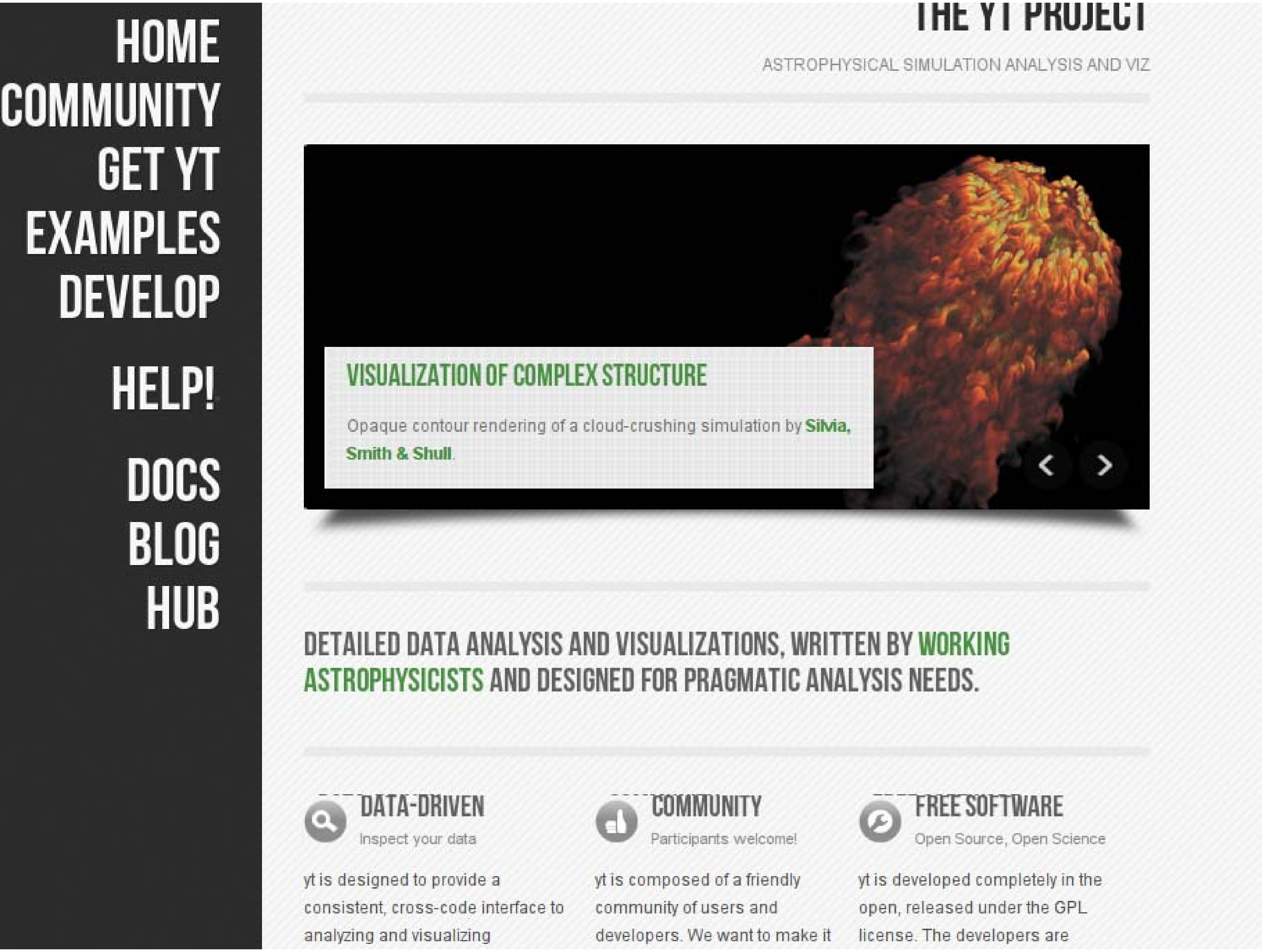
- Entirely distributed development model
  - Small number of developers per institution
- Use code forks / pull requests to move features from development branches to the main branch
- Almost all discussion on archived public mailing lists
  - And on Google docs

COMMUNITY

- ❑ Most developers are astrophysicists “scratching their own itch”
- ❑ Development spurred by ~1.5 workshops/year
  - ❑ And periodic task-oriented “code sprints”
  - ❑ Many streams of funding
- ❑ Enthusiastic and heavily involved user/developer community

Challenges:

- ❑ No leader => hard to make major code revisions
- ❑ Part-time developers: distractions, less incentive to do “boring but important” infrastructure development
- ❑ Significant work required to build consensus and keep community together



HOME
COMMUNITY
GET YT
EXAMPLES
DEVELOP
HELP!
DOCS
BLOG
HUB

THE YT PROJECT

ASTROPHYSICAL SIMULATION ANALYSIS AND VIZ

VISUALIZATION OF COMPLEX STRUCTURE

Opaque contour rendering of a cloud-crushing simulation by Silvia, Smith & Shull.

DETAILED DATA ANALYSIS AND VISUALIZATIONS, WRITTEN BY WORKING ASTROPHYSICISTS AND DESIGNED FOR PRAGMATIC ANALYSIS NEEDS.

DATA-DRIVEN

Inspect your data

yt is designed to provide a consistent, cross-code interface to analyzing and visualizing

COMMUNITY

Participants welcome!

yt is composed of a friendly community of users and developers. We want to make it

FREE SOFTWARE

Open Source, Open Science

yt is developed completely in the open, released under the GPL license. The developers are

COMMUNITY CODES: SUMMARY

- ❑ Open source with a governance structure in place
- ❑ Trust building among teams
- ❑ Commitment to transparent communications
- ❑ Strong commitment to user support
- ❑ Either an interdisciplinary team, or a group of people comfortable with science and code development
- ❑ Attention to software engineering and documentation
- ❑ Understanding the benefit of sharing as opposed to being secretive about the code

CONTRIBUTION POLICIES

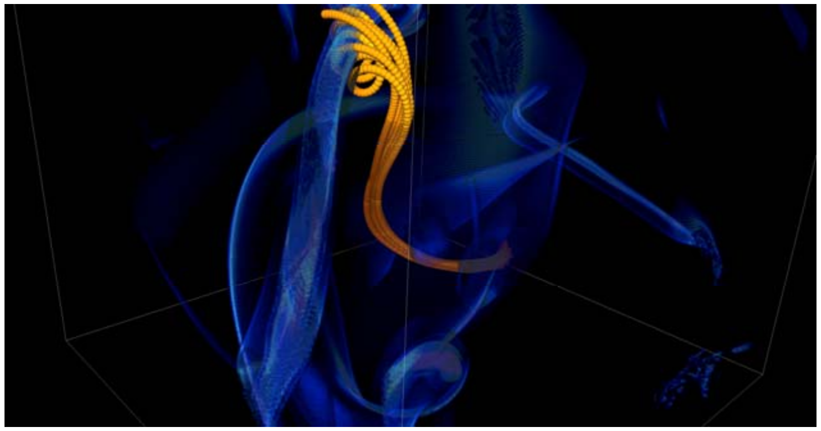
- ❑ Balancing contributors and code distribution needs
    - ❑ Contributor may want some IP protection
- ❑ Maintainable code requirements
    - ❑ The minimum set needed from the contributor
        - ❑ Source code, build scripts, tests, documentation
- ❑ Agreement on user support
    - ❑ Contributor or the distributor
- ❑ Add-ons: components not included with the distribution, but work with the code

CONCLUSIONS

- There are many reasons why software engineering practices are good and should be encouraged
  - Science and engineering by simulation needs more scrutiny into the methods and software
  - There is no need to keep reinventing the wheel
    - This is especially true of book-keeping work
    - Reuse infrastructural components
- The days of heroic programming are past, collaborative efforts are more productive

It is extremely important to recognize that science through computing is only as good as the software that produces it

LAST THOUGHT: WHAT CAN HAPPEN WHEN PROCESS IS IGNORED



- In 2005 BG/L was made available at short notice
- Quick and dirty development of particles
- Many in-flight corrections of defects
- One was adding tags to track individual particles
  - **Got many duplicated tags due to round-off**
- Had to develop post-processing tools to correctly identify trajectories

FLASH had a software process in place. It was tested regularly. This was one instance when the full process could not be applied because of time constraints. We got ready for the run in less than a month, the run went for 1.5 weeks, and it took over 6 months before we could trust the processed results.