FUN3D v12.4 Training

Session 7:
Code Development

Mike Park

Motivation

- Development of NASA Langley CFD solvers was traditionally lead by one or two people
  - Resulted in tools that are vulnerable to the loss of a single person
- Rigorous and repeatable testing environments were rare
- Standardization, portability, and performance was often overlooked
- Informal or no source code version control
- Low-level collaboration within NASA Langley very difficult; off-site collaborations near impossible

Software	Caretaker(s)	Lines of Source Code
CFL3D	Rumsey, Biedron	153,000
OVERFLOW	Buning	160,000
LAURA	Gnoffo	75,000
VULCAN	Baurle, White	180,000
TLNS3D	Vatsa	60,000

Motivation

- Needed expertise from a large number of subject matter experts to perform the required simulation for NASA projects
  - "Opportunities for Breakthroughs in Large-Scale Computational Simulation and Design" NASA TM-211747 (2002)
- These experts have tasks other than part-time code development
- There is a strong dislike for ceremonial software processes
- The number of developers and the amount of source code required is almost an order of magnitude larger than previous projects
- Previous software development methods were seen as problematic at this scale

Software	Caretaker(s)	Lines of Source Code
CFL3D	Rumsey, Biedron	153,000
OVERFLOW	Buning	160,000
LAURA	Gnoffo	75,000
VULCAN	Baurle, White	180,000
TLNS3D	Vatsa	60,000
FUN3D	~10 Developers	800,000

Motivation

- High Energy Flow Solver Synthesis (HEFSS) effort was created as an element of the Faast Adaptive AeroSpace Tools (FAAST) project for the whitepaper vision
  - Unstructured-grid analysis and design across speed range
    - Incompressible
    - Compressible
    - Hypersonic reacting gas
  - Combine and extend the strengths of multiple tools
    - FUN3D unstructured-grid, incompressible and compressible analysis and design
    - LAURA structured-grid, external hypersonics
    - VULCAN structured-grid, internal hypersonics
  - Reduce time from concept to application for vehicles and algorithms
  - Mobility to respond to unforeseen challenges and increase software lifespan

Production and Research

- We are pulled in multiple directions by simultaneously combining research and production needs in the same code base
- Similar to “Google’s Hybrid Approach to Research” Communications of the ACM (2012) we strive to do research in a production environment
  - Provides stability to suit time-sensitive application needs and to release to outside customers
  - Speeds the use of research ideas on large scale problems and allows evaluation in relevant situations
- Disadvantages
  - May slow research to some degree
  - Can frustrate applications engineers due to rapid pace of change and occasional instability
  - Often requires extra code to support research that is not always pruned when a research avenue is no longer explored

Modern Software Practices

- "Sabbatical" by Bil Kleb and Bill Wood
  - Sponsored a local lecture series entitled "Modern Programming Practices."
  - Prototyped in "Exploring XP for Scientific Research" IEEE Software (2003)
- Processes can be loosely categorized as
  - Ad hoc
    - "Code and Fix"
  - Plan-driven
    - Predictive, "Big up front design"
    - Delivering to the original contract
    - Capability Maturity Model (CMM), CMMI
  - Agile
    - Adaptive, "Evolutionary design"
    - Recognizes software development an empirical process that can not always be defined
    - Extreme Programming (XP)

Modern Software Practices

- Approach is summarized in two versions
  - "Team Software Development for Aerothermodynamic and Aerodynamic Analysis and Design" NASA TM-212421 (2003)
  - "Collaborative Software Development in Support of Fast Adaptive AeroSpace Tools (FAAST)" AIAA-2003-3978 (2003)

XP

- Extreme Programing (XP) appeared lighter weight and more flexible than alternatives like CMMI
- Values
  - Communication
  - Simplicity
  - Feedback
  - Courage
- Only adopted the parts that fit due to challenges
  - Geography
  - Part-time developers
  - We view ourselves differently than a full-time software development group
  - Wide range of interest in trying and developing new processes
- The list of XP practices will be used as a framework to describe our current process

XP Practices

- XP has twelve practices that are interconnected
- We implemented the parts that fit nicely with our team and adapted or omitted the parts that did not fit
- These practices are still in place after 10 years
  - It was not obvious at the time that this effort would last beyond a year or two
  - It has created a coalition of the willing
  - There have been some small tweaks over the years
  - Some things we think we can do better now (indicates learning)

XP Practices (In Order of Adoption)

- Sustainable pace
- Metaphor
- Coding standards
- Collective ownership
- Continuous integration
- Small releases
- Test-driven development
- Refactoring
- Simple design
- Pair programming
- On-site customer
- Planning game

Sustainable Pace (Full Adoption)

- Pushing workers too hard is possible for short bursts, but is not sustainable and has far reaching negative consequences in the long term
- Compulsory overtime is simply not part of the 40-hour week working environment that we operate under

Metaphor (Full Adoption)

- A consistent metaphor facilitates communication both within the code and within the team
- Our CFD jargon fits nicely
  - `rho` for density
  - `u`, `v`, `w` for velocity
- Metaphor is often refined in pair programming sessions
  - Having a metaphor that two people can understand goes a long way towards being understandable by the entire team

Coding Standard (Full Adoption)

- Coding standard encourages readability through uniformity and enables
  - Code translation and modification via scripting
  - Portability
- Code is read many more times than it is written
- You know you have succeeded when you can't tell who wrote it
- Enables complex-step derivatives and parsing to extract user manual sections
- Partly enforced during SVN commits via a sentinel and with compiler warnings
- Some languages provide an automated formatter (gofmt, Uncrustify)
  - It makes code easier to write, read, and maintain
  - Uncontroversial!
  - We didn't like the Fortran products available a decade ago

Collective Code Ownership (Full)

- Change any line of code at anytime
  - No single developer claims code ownership
  - All developers share responsibility
- Requires automated testing to allow changes outside of a developer's expertise
- Requires a coding standard to eliminate the distraction of changing formatting
- Communication is key to avoiding problems and minimizing frustrations due to "Who Moved My Subroutine?"
- Requires a version control system

Version Control

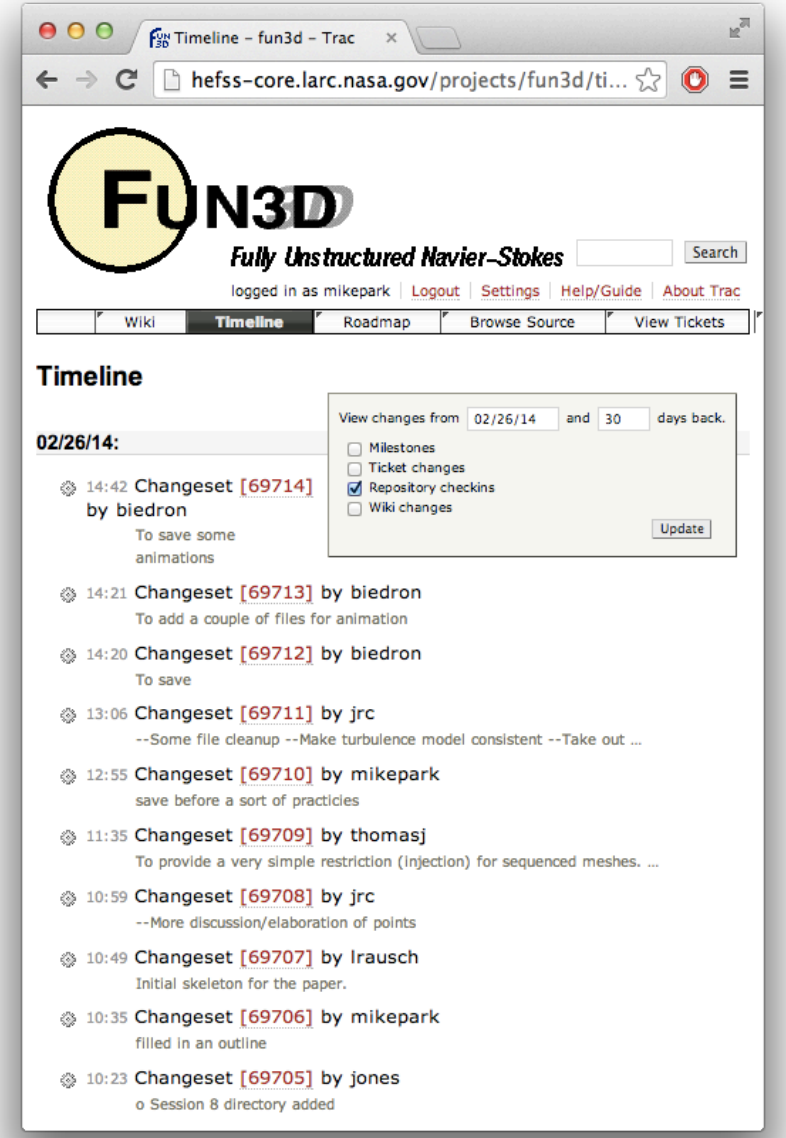
- Often overlooked or under emphasized
- “Zeroth” principle of software engineering
- Learning to work with it and not against it is key to team programming (glue)
- Safety net or “Save Button”
- Convenient for accounts on multiple machines
- Not just for code: test cases, build scripts, presentations, papers, website, user manual, etc.
- Required for automated testing and continuous integration

Version Control

- Formal version control using Subversion (SVN); repository sits outside LaRC firewall protected with two-factor authentication
  - Enables any authorized developer to work directly on the source code in real-time
  - Frequent commits encouraged to prevent conflicts
    - typically 10 a day to “trunk” and 30 a day in total
  - Integrates all capabilities in one centralized place
    - Avoids fracturing or divergence of the code
    - Integration problems are addressed early
  - Allows arbitrary version of the code to be recalled
    - Facilitates user support and isolation of bugs
  - Pre-commit script to validate commits (sentinel)
  - Commit logs are posted to an email list
  - Real-time view on available on a Trac website

Trac Website for SVN History

- Website lists commits
- Each commit has a URL for reference
- Formatted, side-by-side differences between committed versions



Testing

- Programmer's (or Unit)
  - I want a function that adds vectors, does f([1,2],[3,4]) return [4,6]?
- Integration
  - Does my whole system compile and work together?
- Regression
  - My code gave answer *x* yesterday, does it give answer *x* today?
- Verification
  - Am I solving the governing equations correctly?
  - My code is supposed to be second-order accurate in space; what happens when I change the element size?
- Validation
  - Does my model implemented in the code give the same answer as a wind tunnel or flight test measurement?
- Performance
- Acceptance
  - Will the code work for my application

Regression Tests

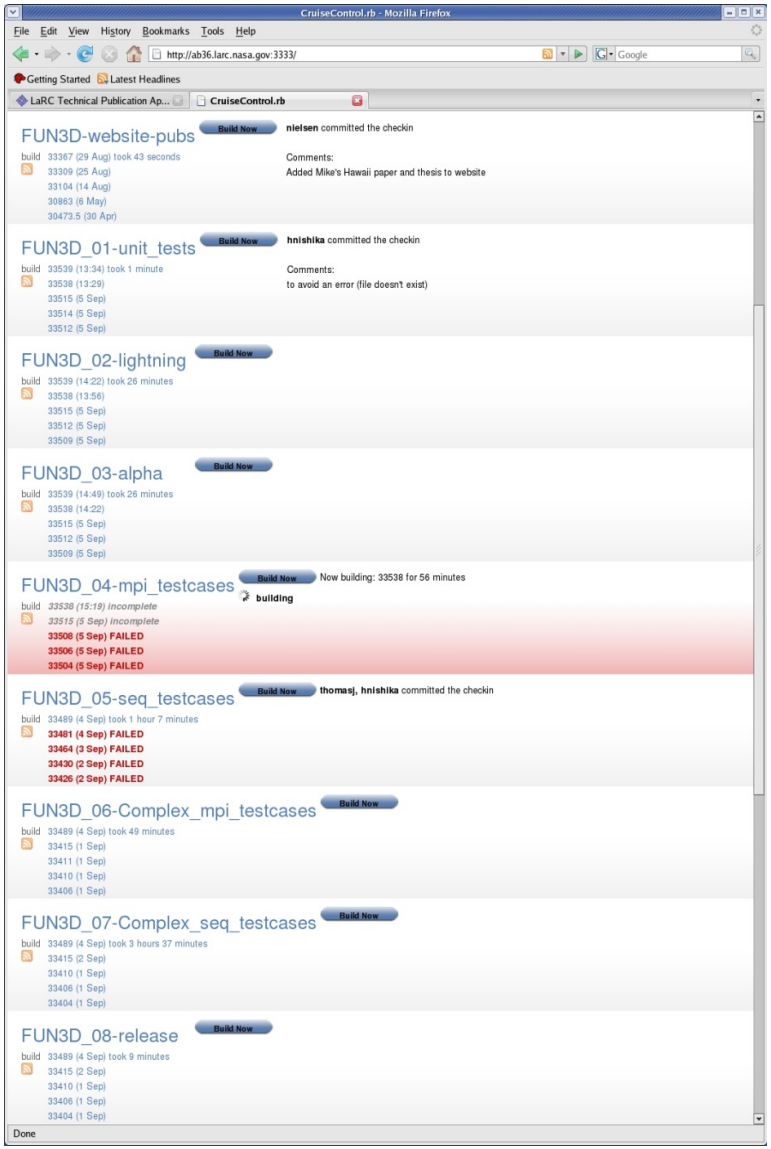
- Have added value and confidence, but we have become overly relent on this form of testing
- Become too easy to add overly complicated, long running tests that are costly to maintain
- Currently takes hours to run
  - Running in minutes allows you to remember what you are doing
- Often combines too many compatibles, making it difficult to tell the cause of a change in results
- Requires the subject matter expert to “recertify” changes to expected results (golden files)
  - Multiple examples of accepting bugs due to signal to noise ratio
  - Subject to confusion due to round off
  - Trying to move to golden file free tests
  - Could use the coarsest mesh of a verification test gain confidence in “recertification”

Continuous Integration (Full Adoption)

- Hundreds of tests performed continuously around-the-clock
  - Unit tests of small self contained functionality
  - Regression tests to ensure the code gives the same result today as yesterday Performance tests for execution speed
  - Validation tests (Turbulence Modeling Resource Website)
  - Code quality analysis
- Failures reported to team members immediately via email, SMS
  - Creates peer pressure on developers to compile and run a range of tests before committing changes because it hinders the entire team
- Bugs confined to a few lines of code and identified within an hour, rather than thousands of lines developed over months
  - The cost to fix a bug raises exponentially in time: Boehm, “Software Engineering Economics” (1981)
- Wide range of compilers/hardware: Linux/Mac/Alpha/HPC/etc
  - Compiler bugs identified routinely

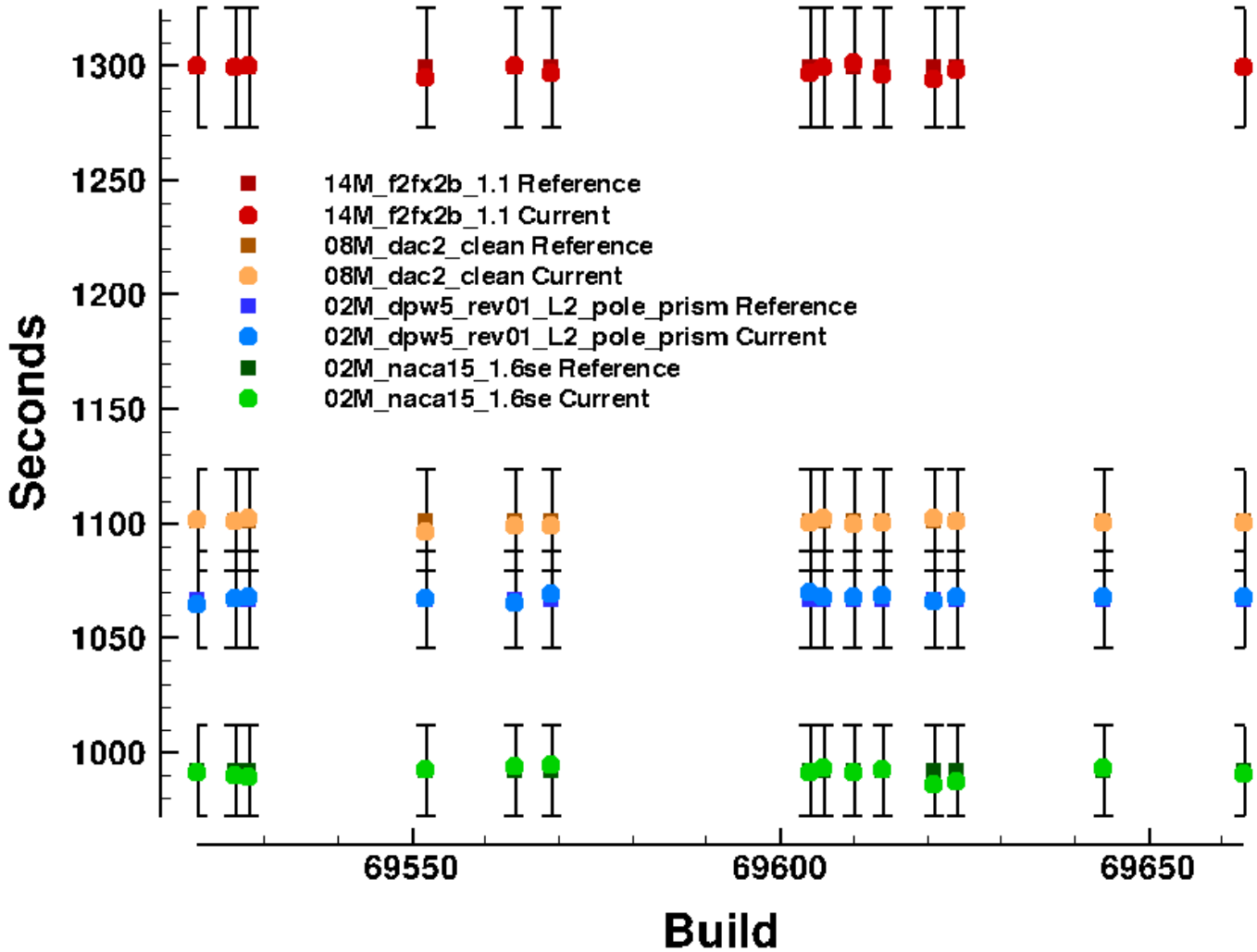
Continuous Integration

- Hierarchy of test on a CruiseControl.rb website
  - Unit test of small self contained functionality
  - Compile checks (10 compilers) with warnings
  - Regression tests to ensure the code gives the same result today as yesterday
  - Performance tests for execution speed
  - Typeset and spell check user's manual
  - Code quality analysis
  - Execute tutorials
  - Create code distribution tar ball



Continuous Integration

- Performance tests for a number of parallel cases with a 2 percent tolerance



Code Quality Analysis

- Embrace and automate code analysis: Static compile checks, run-time checks, Valgrind heap analysis
  - Memory leaks kill big expensive simulations and are 20+ times more costly to repair by hand
  - Uninitialized variables produce unrepeatable results
  - Floating point exceptions (Inf, NaN)
- Code base must be clean to target specific problem as they arise, preventing “Yak Shaving” (overcoming an intermediate difficulty, which allows you to solve a larger problem)

Small Releases (Partial Adoption)

- The goal is to increase feedback and shorten learning cycles
- Developers tend to use the latest version for applications
- Every version of the code that passes all of the automated tests is tagged
  - Typically used by internal application engineers
- Formal external releases are created irregularly, every 6-8 months
  - Driven by an external customers critical need
  - Requires a software usage agreement
  - Always a tagged version that has passed all automated tests
- Two packages available
  - Perfect gas capability is export controlled
  - High-energy generic gas capability is ITAR restricted

Test-Driven Development (Partial)

- The time to fix a software defect (aka “bug”) scales exponentially with the time lag between introduction and detection: Boehm, “Software Engineering Economics” (1981)
- Ideally no functionality is added to the code before an automated test verifies its correctness
- Unit testing preferred by some developers, but limited adoption by team as a whole
  - Tough to retrofit legacy code
  - Released open source “FUnit” framework for unit testing Fortran
- Overly rely on regression tests

Test-Driven Development (Partial)

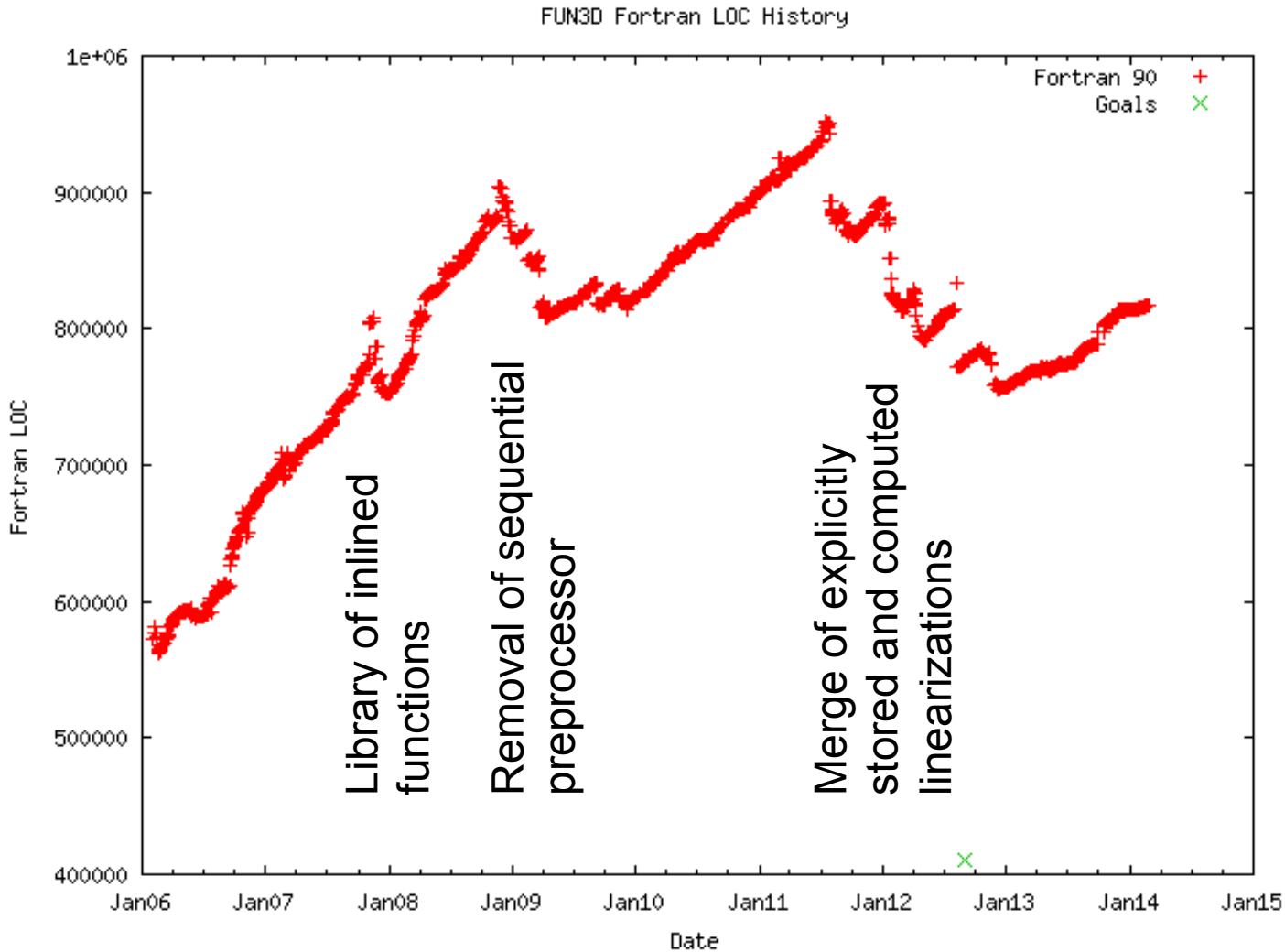
- Seems trivial at first
- Hard to imagine benefit until the first major refactoring or code simplification is experienced
- Gains power as the number of tests and their coverage increases
- Produces your own custom debugger
- Provides a clear completion to an implementation task
- Code with a failing test is much easier to fix or extend
- Inventing the tests required is generally harder
- Code that is easy to test is often simpler and easier to read, understand, and extend
- Creating tests brings the design to the forefront; design is difficult, but it is easiest in small increments
- Testing framework
  - Must allow for the easy creation and management of tests with a minimal additional effort over writing the actual code
  - Enable programmers to experience the benefits of test-first programming as soon as possible

Refactoring (Partial Adoption)

- Simplify and remove duplication while maintaining
- Critical to the long term survival of software and team happiness
- Currently only done when things are critically in the way
  - Limited test converge can erode confidence
  - Limited developer time and confidence in understanding the code
  - Lack of target high-level architecture plan or vision
  - No time or resources explicitly dedicated to the refactoring task in our project funded environment
- The refactoring deficit creates a technical debt that incurs additional cost by deferring maintenance

Refactoring (Partial Adoption)

- Can be seen as reduction on lines of code



Simple Design (Partial Adoption)

- Ideally evolved from refactoring, pair programming, and test first
  - You aren't gonna need it (YAGNI)
  - Do the simplest thing that could possibly work
  - Design for assured requirements not "expected" requirements
- The current highly coupled architecture makes it hard to make progress
- Requires courage to delete half implemented or outdated approaches

Pair Programming (Partial Adoption)

- Two people working on the same computer
- Pilot and navigator roles that change continuously
- Great communication and mentoring
- Great for building the metaphor (variable and function names)
- Bug fixing at the intersection between two disciplines
- Counterintuitive, but research shows it is more efficient
- Some people like doing it and some people don't
- Remote pair programming works very well
- Rule of pair programming non-refusal
  - Helps wall flowers

On-site Customer (Partial Adoption)

- Intended to bring the developers and the customers as close together as possible
  - Not separated by a contract
- We lack customers in the classic definition
- Developers are their own customers
  - Requires diligent role playing to keep technical and business needs separated
  - Minor high level input from project stakeholders and users

The Planning Game (No Adoption)

- Should be responsive to customer by balancing time, cost, quality, and scope
- Informal developer by developer decisions on priorities are made
- No formal FUN3D project
- Funded by projects that are typically more interested in applications
  - No budget for refactoring and documentation
- Without high-level planning we have inefficiencies
- Sandia National Laboratories has a strong focus in this area “How We Successfully Adapted Agile for a Research-Heavy Engineering Software Team” AGILE (2013)

Communication

- Most important element in team software development, but expensive!
- Weekly “scrum-style” meeting foster collaboration and communication with low overhead
- Management/observers encouraged to attend, but only workers allowed to talk
- One at a time, each member reports
  - What they did this week
  - What they will next week
  - What is in the way (impediment)
- Scrum master notes impediments to progress, strives to remove them
- Goal is 15 minutes; Law of Two Feet applies
- No “Death by PowerPoint”
- Tendency of engineers to always fix stuff is suppressed, clarification questions only, more involved discussions are tabled until after the scrum
- Additional discussion via mailing list, wiki

Retrospectives

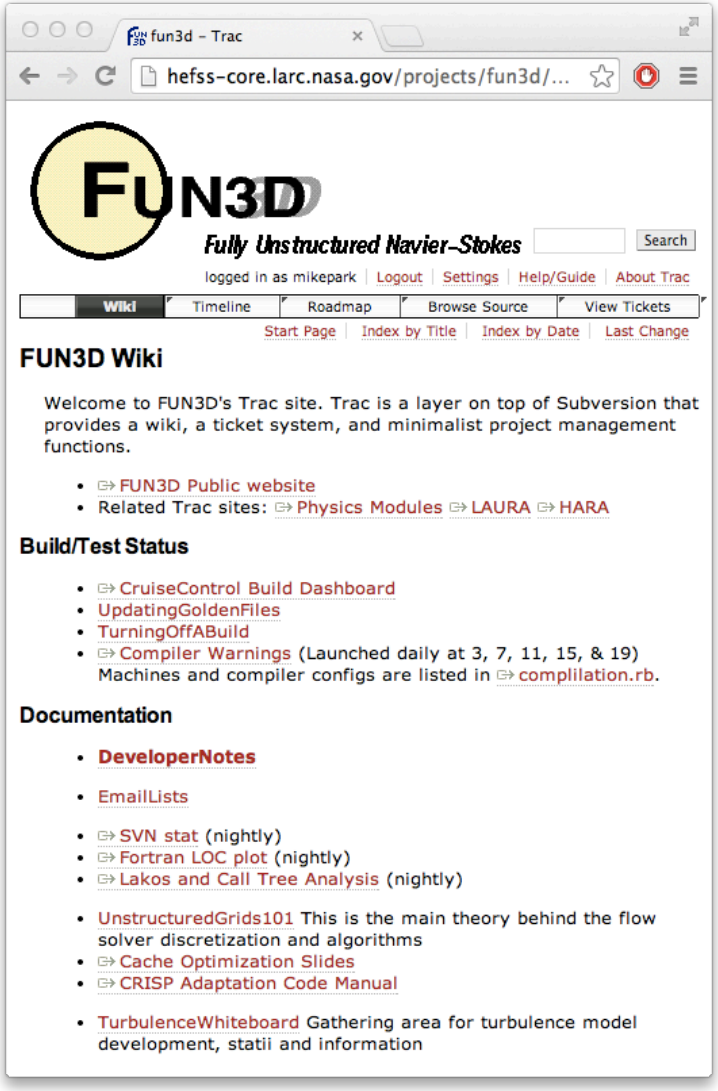
- Too few and far between, every 2-4 years
- Prime Directive
  - Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given what was known at the time, his or her skills and abilities, the resources available, and the situation at hand.
- Tend to be a useful catharsis of pent up frustrations
- Shining light on things tends to make them “fix themselves”
- Unfortunately the “hard” problems are identified at multiple retrospectives

War Room

- All developers meet for a single day of discussion and code development
  - Valuable to part-time or new team members
- Certain weakness reduced its utility
  - The planning phase was difficult because it is rarely performed by the group
  - Overlapped with the retrospective in some ways (i.e., catharsis of pent up frustrations)
  - Projects initiated but not completed in a single day had to be completed by other developers

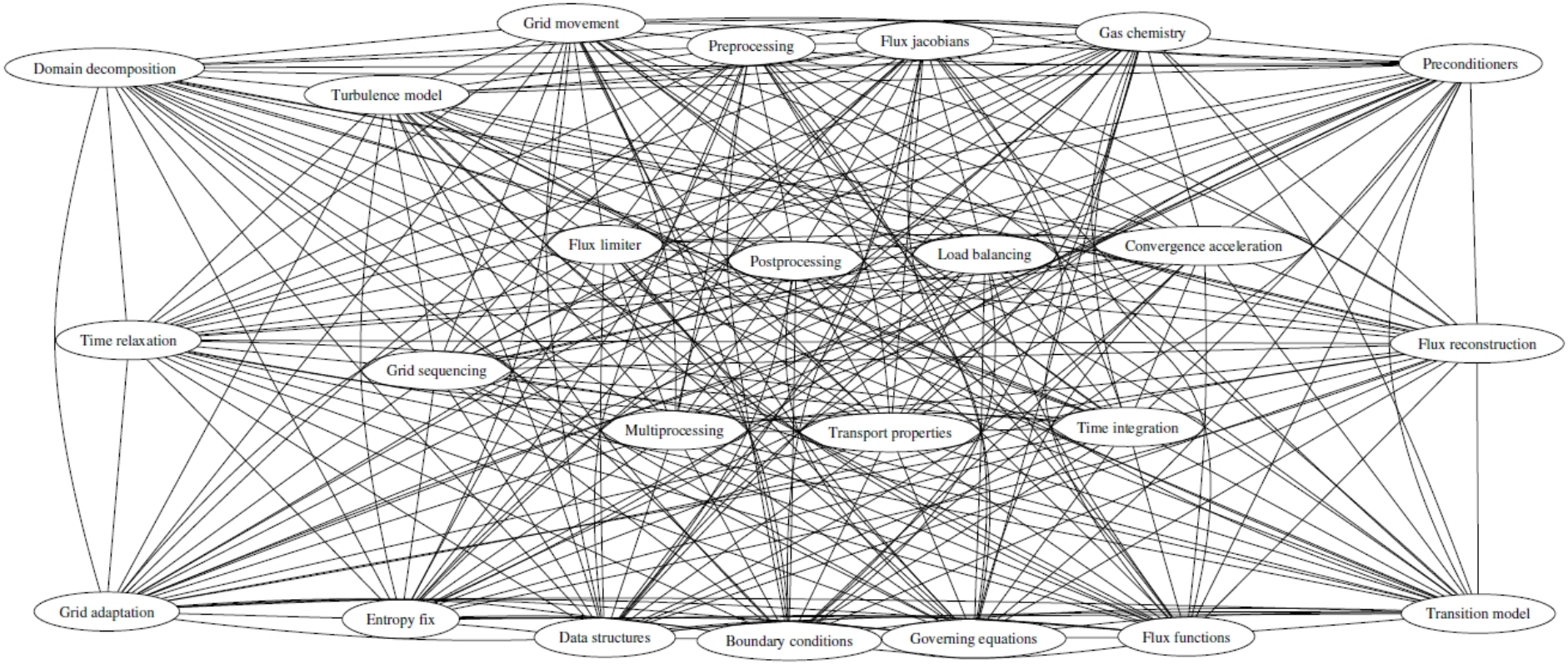
Documentation

- Website also kept under Subversion, maintained collectively by entire team
- Automatically generated and placed on server whenever text files in repository are updated
- Team need not know fancy HTML to contribute
- LaTeX manual sections parsed from comments in namelist code
- Wiki also hosted with Trac



Room for Improvement

- Could be described as “Ball of Mud” (Foote) architecture
- Too many interconnections that difficult to visualize navigate
- Localized pockets of organization, but they are different and lack universal design



Room for Improvement

- Fear of changing working code, disrupting other's progress, or communicating still results in the copy-paste-change pattern
  - Multiple optimized or simplified versions of the same code that lack features
  - Requiring multiple copies to be maintained or merged
- Identified need for more specialized computer science expertise
  - Average developer is self-taught in this regard
  - Benefit from dedicated computer science personnel

Room for Improvement

- Complexity is the enemy
- Distributed version control system
  - Great time to organize code base and revisit policy
  - Enable quarantine builds
  - Facilitate distributed development
- Reduce over reliance on regression testing with
  - Verification tests
  - Complex-step and adjoint consistency tests
- Code generation for linearization and hybrid architectures
  - Overloading for complex step and derivative derived types (slow run and compile)
- Refactoring to expose APIs
  - Enable collaboration
  - Make it easier to plug into the proliferation of frameworks
  - “FUN3D as a subroutine”
- Quicken feedback loop to reduce the cost (time) of change
  - compile, test, deploy

Other Resources

- CFD Vision 2030
- CREATE
- DOE National Labs
- DLR TAU
- ONERA CEDRE
- Universities