Testing of HPC Scientific Software

SIAM CSE17
Atlanta, GA
February 28, 2017

Tutorial slides available at: http://bit.ly/siam-cse17-mt3

Anshu Dubey (ANL) and Alicia Klinvex (SNL)

Acknowledgments

- Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357.
- Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2016-8466 C.

Outline

- Introduction
- Scientific software verification
- How to evaluate needs of a project and devise a testing regime
- Testing during refactoring
- Hands on - Code coverage
- Demo - Continuous integration

Introduction

Why is testing important?

Definitions

Benefits of testing

- Promotes high-quality software that delivers correct results and improves confidence
- Increases quality and speed of development, reducing development and maintenance costs
- Maintains portability to a variety of systems and compilers
- Helps in refactoring
  - Avoid introducing new errors when adding new features
  - Avoid reintroducing old errors

How common are bugs?

Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them.
- Harlan Mills

- Bugs per 1000 lines of code (KLOC)
- Industry average for delivered software
  - 1-25 errors
- Microsoft Applications Division
  - 10-20 defects during in-house testing
  - 0.5 in released product

Code Complete (Steven McConnell)

Why testing is important:
the protein structures of Geoffrey Chang

- Some inherited code flipped two columns of data, inverting an electron-density map
- Resulted in an incorrect protein structure
- Retracted 5 publications
  - One was cited 364 times
- Many papers and grant applications conflicting with his results were rejected

Why testing is important:
the 40 second flight of the Ariane 5

- Ariane 5: a European orbital launch vehicle meant to lift 20 tons into low Earth orbit
- Initial rocket went off course, started to disintegrate, then self-destructed less than a minute after launch
- Seven variables were at risk of leading to an Operand Error (due to conversion of floating point to integer)
  - Four were protected
- Investigation concluded insufficient test coverage as one of the causes for this accident
- Resulted in a loss of $370,000,000.

Why testing is important:
the Therac-25 accidents

- Therac-25: a computer-controlled radiation therapy machine
- Minimal software testing
- Race condition in the code went undetected
- Unlucky patients were struck with approximately 100 times the intended dose of radiation, ~ 15,000 rads
- Error code indicated that no dose of radiation was given, so operator instructed machine to proceed
- Recalled after six accidents resulting in death and serious injuries

Definitions

- Unit tests
  - Test individual functions or classes
- Integration tests
  - Test interaction, build complex hierarchy
- System level tests
  - At the user interaction level

Definitions

- Restart tests
  - Code starts transparently from a checkpoint
- Regression (no-change) tests
  - Compare current observable output to a gold standard
- Performance tests
  - Focus on the runtime and resource utilization

Policies on testing practices

- Avoid regression suites consisting of system-level no-change tests
  - Tests often need to be re-baselined
    - Often done without verification of new gold-standard
  - Hard to maintain across multiple platforms
  - Loose tolerances can allow subtle defects to appear

Policies on testing practices

- Must have consistent policy on dealing with failed tests
  - Issue tracking
    - How quickly does it need to be fixed?
    - Who is responsible for fixing it?
  - Add regression test afterwards (to avoid reintroducing issue later)
- Someone needs to be in charge of watching the test suite

Policies on testing practices

- When refactoring or adding new features, run a regression suite before checkin
  - Be sure to add new regression tests for the new features
- Require a code review before releasing test suite
  - Another person may spot issues you didn't
  - Incredibly cost-effective

Example: Trilinos checkin test script

- Detects which packages were modified by your commits
- Determines which packages you potentially broke
- Configures, builds, and tests those packages
  - On success, pushes to repo
  - On failure, reports why it failed
- Useful for ensuring your changes don't break another package
- May take a while, but many people run it overnight

Maintenance of a test suite

- Testing regime is only useful if it is
  - Maintained
    - Tests and benchmarks periodically updated
  - Monitored regularly
    - Can be automated
  - Has rapid response to failure
    - Tests should pass most of the time

Use of test harnesses

- Essential for large code
  - Set up and run tests
  - Evaluate test results
- Easy to execute a logical subset of tests
  - Pre-push
  - Nightly
- Automation of test harness is critical for
  - Long-running test suites
  - Projects that support many platforms

Jenkins
C-dash
Custom
(FlashTest)

Example: Trilinos automated testing

Login All Dashboards Monday, June 06 2016 08:58:08 MDT

Trilinos

Dashboard Calendar Previous Current Project

Project

Project	Configure			Build			Test		
	Error	Warning	Pass	Error	Warning	Pass	Not Run	Fail	Pass
Trilinos	1	531	530	0	272	257	0	14	3976

SubProjects

Project	Configure			Build			Test		
	Error	Warning	Pass	Error	Warning	Pass	Not Run	Fail	Pass
Teuchos	0	21	21	0	12	9	0	0	227
ThreadPool	0	1	1	0	0	1			
Sacado	0	2	2	0	2	0	0	0	564
RTOp	0	20	20	0	0	20			
Kokkos	0	19	19	0	0	19	0	0	9
Epetra	0	21	21	0	12	9	0	1	244
Zoltan	0	21	21	0	13	8	0	0	135
Shards	0	1	1	0	0	1			
GlobiPack	0	1	1	0	0	1			

SIAM CSE17, Feb 2017

IDEAS productivity

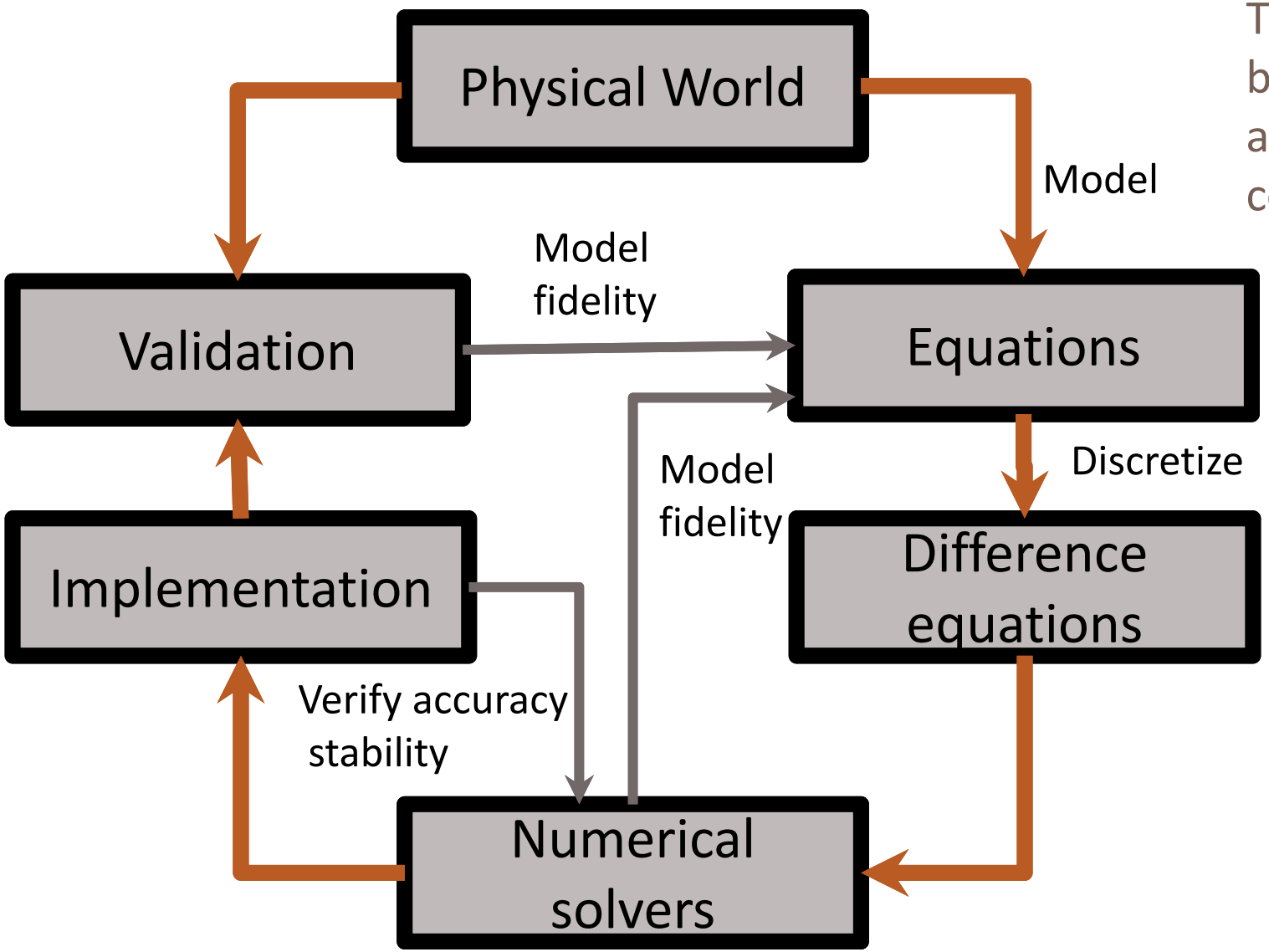
Scientific Software Verification

Challenges specific to scientific software

Verification

- Code verification uses tests
  - It is much more than a collection of tests
- It is the holistic process through which you ensure that
  - Your implementation shows expected behavior,
  - Your implementation is consistent with your model,
  - Science you are trying to do with the code can be done.

Simplified schematic of science through computation



This is for simulations, but the philosophy applies to other computations too.

Many stages in the lifecycle have components that may themselves be under research => need modifications

CSE verification challenges

- Floating point issues
  - Different results
    - On different platforms and runs
    - Ill-conditioning can magnify these small differences
      - Final solution may be different
      - Number of iterations may be different
- Unit testing
  - Sometimes producing meaningful testable behavior too dependent upon other parts of the code
- Definitions don’t always fit

CSE verification challenges

- Integration testing may have hierarchy too
- Particularly true of codes that allow composability in their configuration
- Codes may incorporate some legacy components
  - Its own set of challenges
    - No existing tests of any granularities
- Examples – multiphysics application codes that support multiple domains

Stages and types of verification

- During initial code development
  - Accuracy and stability
  - Matching the algorithm to the model
  - Interoperability of algorithms
- In later stages
  - While adding new major capabilities or modifying existing capabilities
  - Ongoing maintenance
  - Preparing for production

Stages and types of verification

- If refactoring
  - Ensuring that behavior remains consistent and expected
- All stages have a mix of automation and human-intervention

Note that the stages apply to the whole code as well as its components

How to evaluate project needs

And devise a testing regime

Why not always use the most stringent testing?

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high…
  - Team cannot meet code-use objectives
- When is the tax is too low…
  - Necessary oversight not provided
  - Defects in code sneak through

Evaluating project needs

- Objectives: expected use of the code
- Team: size and degree of heterogeneity
- Lifecycle stage: new or production or refactoring
- Lifetime: one off or ongoing production
- Complexity: modules and their interactions

Commonalities

- Unit testing is always good
  - It is unlikely to be sufficient
- Verification of expected behavior
- Understanding the range of validity and applicability is always important
  - Especially for individual solvers

Test Development

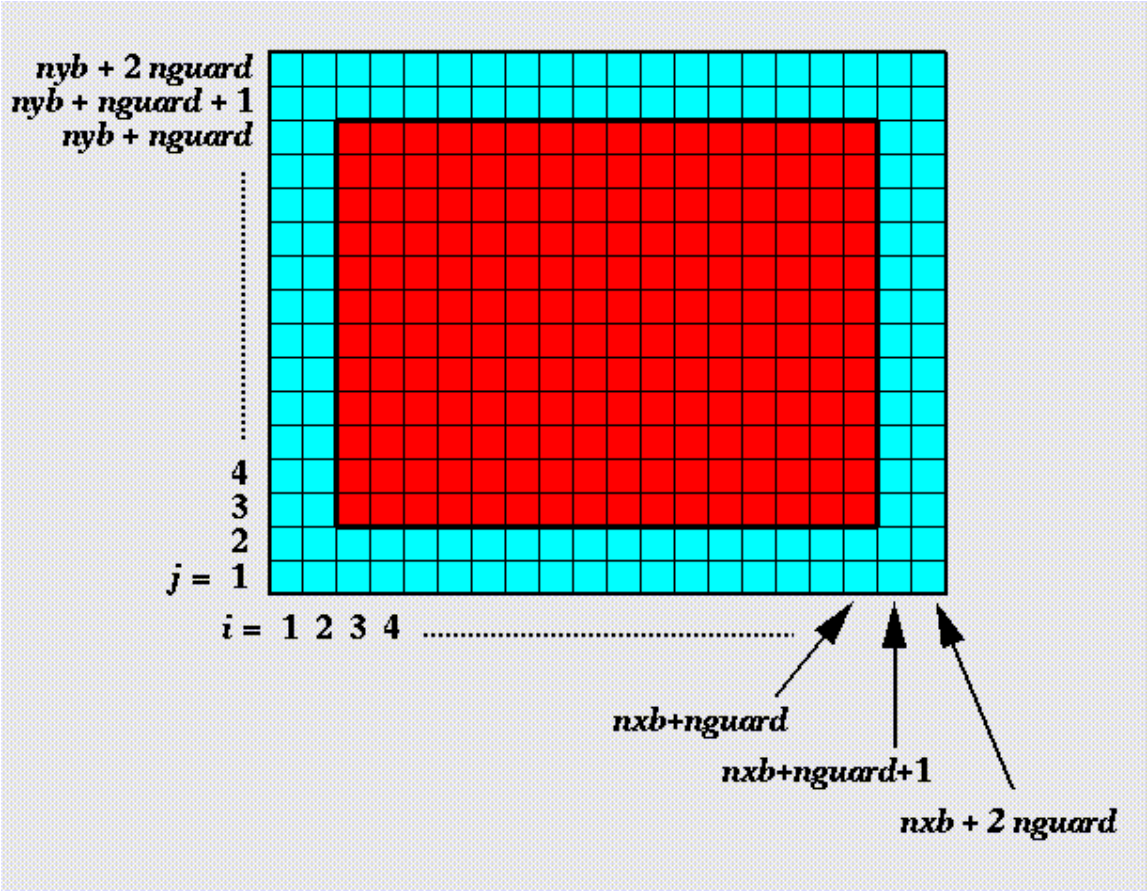
- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
    - They can also form a basis for unit testing
- In addition to testing for “correct” behavior, also test for stability, convergence, or other such desirable characteristics
- Many of these tests go into the test-suite

Example from Flash

- Grid ghost cell fill
  - Use some function to initialize domain
  - Two variables, in one only interior cells initialized, in the other ghost cells also initialized
  - Run ghost cell fill on the first variable – now both should be identical within known tolerance
- Use redundant mechanisms

Against manufactured solution

- Verification of guard cell fill
- Use two variables A & B
- Initialize A including guard cells and B excluding them
- Apply guard cell fill to B

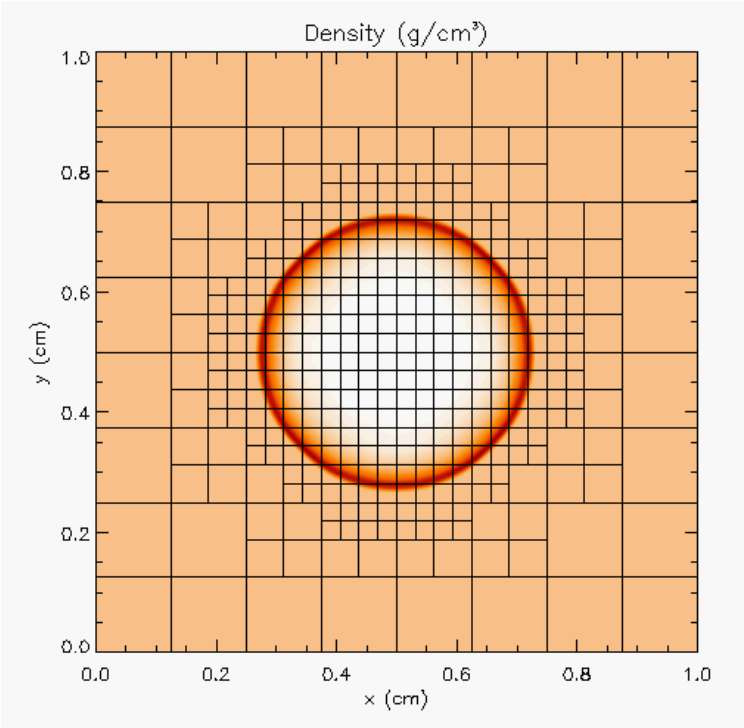


Example from Flash

- Eos
  - Use initial conditions from a known problem
  - Apply eos in two different modes – at the end all variables should be consistent within tolerance
- Hydrodynamics
  - Sedov blast problem has a known analytical solution
  - Runs with UG and AMR

Against analytical solution

- Sedov blast wave
- High pressure at the center
- Shock moves out spherically
- FLASH with AMR and hydro
- Known analytical solution



Though it exercises both mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

Building confidence

- First two unit tests are stand-alone
- The third test depends on Grid and Eos
  - Not all of Grid functionality it uses is unit tested
    - Flux correction in AMR
- If Grid and Eos tests passed and Hydro failed
  - If UG version failed then fault is in hydro
  - If UG passed and AMR failed the fault is likely in flux correction

Development phase – adding on

- Few more steps when adding new components to existing code
  - Know the existing components it interacts with
  - Verify its interoperability with those components
  - Verify that it does not inadvertently break some unconnected part of the code
- May need addition of tests not just for the new component but also for some of the old components
  - This part is often overlooked to the detriment of the overall verification

Selection of tests

- Important to aim for quick diagnosis of error
  - A mix of different granularities works well
    - Unit tests for isolating component or sub-component level faults
    - Integration tests with simple to complex configuration and system level
    - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

Full paper Dubey et al 2015

Approach

- Build a matrix
  - Physics along rows
  - Infrastructure along columns
  - Alternative implementations, dimensions, geometry
- Mark <i,j> if test covers corresponding features
- Follow the order
  - All unit tests – including full module tests
  - Tests representing ongoing productions
  - Tests sensitive to perturbations
  - Most stringent tests for solvers
  - Least complex test to cover remaining spots

Example

	Hydro	EOS	Gravity	Burn	Particles
AMR	CL	CL		CL	CL
UG	SV	SV			SV
Multigrid	WD	WD	WD	WD	
FFT			PT		

Tests	Symbol
Sedov	SV
Cellular	CL
Poisson	PT
White Dwarf	WD

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology

Refactoring

Testing needs during code refactor

Considerations

- Know why you are refactoring
- Know the scope of refactoring
- Know bounds on acceptable behavior change
- Know your error bounds
  - Bitwise reproduction of results unlikely after transition
- Map from here to there
- Check for coverage provided by existing tests
- Develop new tests where there are gaps

Incorporate testing overheads into refactor cost estimates

Challenges with legacy codes

- Legacy codes can have many gotchas
  - Dead code
  - Redundant branches
- Interactions between sections of the code may be unknown
- Can be difficult to differentiate between just bad code, or bad code for a good reason
  - Nested conditionals

Code coverage tools are of limited help

Options

- Test coverings
  - Set of tests used to introduce an invariant
  - Cover a small area of the system
  - Ascertain correct behavior
  - Build the invariant, then refactor to make the code clear
  - Have an on-ramp plan

Code Coverage

How do we determine what other tests are needed?

- Code coverage tools
  - Expose parts of the code that aren't being tested
  - gcov
    - standard utility with the GNU compiler collection suite
    - counts the number of times each statement is executed
  - lcov
    - a graphical front-end for gcov
    - available at http://ltp.sourceforge.net/coverage/lcov.php

How to use gcov/lcov

- Compile and link your code with --coverage flag
  - It's a good idea to disable optimization
- Run your test suite
- Collect coverage data using gcov/lcov
- Optional: generate html output using genhtml

A hands-on gcov tutorial

- https://amklinv.github.io/morpheus/index.html

But I don’t use C++!

- gcov also works for C and Fortran
- Other tools exist for other languages
  - JCov for Java
  - Coverage.py for python
  - Devel::Cover for perl
  - profile for MATLAB
  - etc

Continuous integration

Continuous integration (CI): a master branch that always works

- Code changes trigger automated builds/tests on target platforms
- Builds/tests finish *in a reasonable amount of time*, providing useful feedback when it's most needed
- Immensely helpful!
- Requires some work, though:
  - A reasonably automated build system
  - An automated test system with significant test coverage
  - A set of systems on which tests will be run, and a controller

Continuous integration (CI): a master branch that always works

- Has existed for some time
- Adoption has been slow
  - Setting up and maintaining CI systems is difficult, labor-intensive (typically requires a dedicated staff member)
  - *You have to be doing a lot of things right to even consider CI*

Cloud-based CI is available as a service on GitHub

- Automated builds/tests can be triggered via pull requests
- Builds/tests can be run on cloud systems – no server in your closet. *Great use of the cloud!*
- Test results are reported on the pull request page (with links to detailed logs)
- Already being used successfully by scientific computing projects, with noticeable benefits to productivity
- Not perfect, but *far* better than not doing CI

Travis CI is a great choice for HPC

- Integrates easily with GitHub
- *Free* for Open Source projects
- Supports environments with C/C++/Fortran compilers (GNU, Clang, Intel[?])
- Linux, Mac platforms available
- *Relatively* simple, *reasonably* flexible configuration file
  - Documentation is sparse, but we now have working examples

Travis CI live demo

- https://github.com/amklinv/morpheus

Other resources

Software testing levels and definitions:
http://www.tutorialspoint.com/software_testing/software_testing_levels.htm

Working Effectively with Legacy Code, Michael Feathers. The legacy software change algorithm described in this book is very straight-forward and powerful for anyone working on a code that has insufficient testing.

Code Complete, Steve McConnell. Excellent testing advice. His description of Structure Basis Testing is good, and it is a simple concept: Write one test for each logic path through your code.

Organization dedicated to software testing: https://www.associationforsoftwaretesting.org/

Software Carpentry: http://katyhuff.github.io/python-testing/

Tutorial from Udacity: https://www.udacity.com/course/software-testing--cs258

Papers on testing:
http://www.sciencedirect.com/science/article/pii/S0950584914001232
https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH

Resources for Trilinos testing:
Trilinos testing policy: https://github.com/trilinos/Trilinos/wiki/Trilinos-Testing-Policy
Trilinos test harness: https://github.com/trilinos/Trilinos/wiki/Policies--%7C-Testing