CSE 380: Lecture 7
Getting the Keys to the Coding Car

Dr. Christopher Simmons
csim@ices.utexas.edu

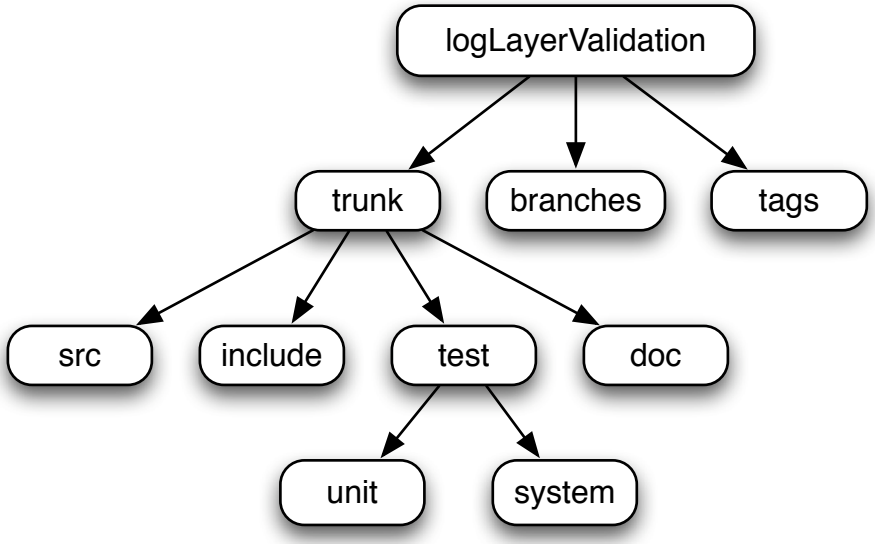
Getting Started – Documentation and Testing

- We have covered…
  - Hardware, Unix, Batch Systems…
  - Version control, compilers and Build systems
- Are we ready to write code? Almost
  - Documentation
  - Requirements
  - Testing/Verification
  - Automation
  - Test Driven Development

We need a place to work…

- To do any work, our project needs a place to live
- Should not just be in your home directory => use version control and work on a local copy
- E.g. : svn co https://collab.tacc.utexas.edu/…
- Start working…
  - Create directories: svn mkdir foo
  - Add files: svn add foo/bar.C
  - Commit to repo: svn ci –m “enlightening commit”

Structure is important



- Trunk: primary development location
- Tags: “tagged” versions (e.g. releases)
- Braches: additional development locations for major refactoring

Model Documentation

- Before you implement anything, you should be able to write down *precisely* what you plan to implement
- Easy to keep track of what you have done and are doing
- Helps to produce papers and thesis after code is done

Model Documentation Cont.

- Ideally, everything necessary to implement your application, including:
  - All model equations, e.g., governing pdes or odes
  - Boundary conditions
  - Detailed discretization approach
  - Solution technique
  - Verification plan
- Implementation details are not required
  - That is, model document is not code documentation, not a developer guide, nor a user guide

LogLayer Model Document

Select External Software

- Know what capability we desire (documented in model document)
- Now we need to decide how to implement that capability
- First question: What do I need to do myself, and what can I outsource?

Select External Software

- First question: What do I need to do myself, and what can I outsource?
- Input parsing: GRVY, Boost program options
- Linear & Non-linear solvers: Petsc, Trilinos
- Finite Elements: Libmesh, Deal II
- General Scientific Comptuing: GSL
- UQ/Statistical Algorithms: QUESO, DAKOTA
- MANY, MANY, MANY MORE!

Spring Mass Damper Example

- Interested in effects of uncertainty and validation philosophy, i.e., not
  - Statistical sampling algorithms
  - Numerical inversion and integration algorithms
  - Input parsing

Spring Mass Damper Example

- Interested in effects of uncertainty and validation philosophy
  - Input parser: GRVY
  - Numerical algorithms: GSL
  - Statistical algorithms: QUESO
- So, I only need to provide
  - Model routines
  - Data handling and uncertainty descriptions

Choose build system

- Have selected some libraries to lighten the workload
- About ready to start coding
- Just need a way to compile & link reproducibly
- Make, Autotools, SCons: you make the call

Unit Testing

- Ready to start writing code, which means we're ready to start writing tests!
- Main idea: Test a "unit" (e.g., a function) of code in isolation from everything else
- Motivation
  - Easier to write tests from small pieces of code rather than entire applications
  - Helps promote modular code design and frequent refactoring
  - Helps identify problem areas when system-level tests fail

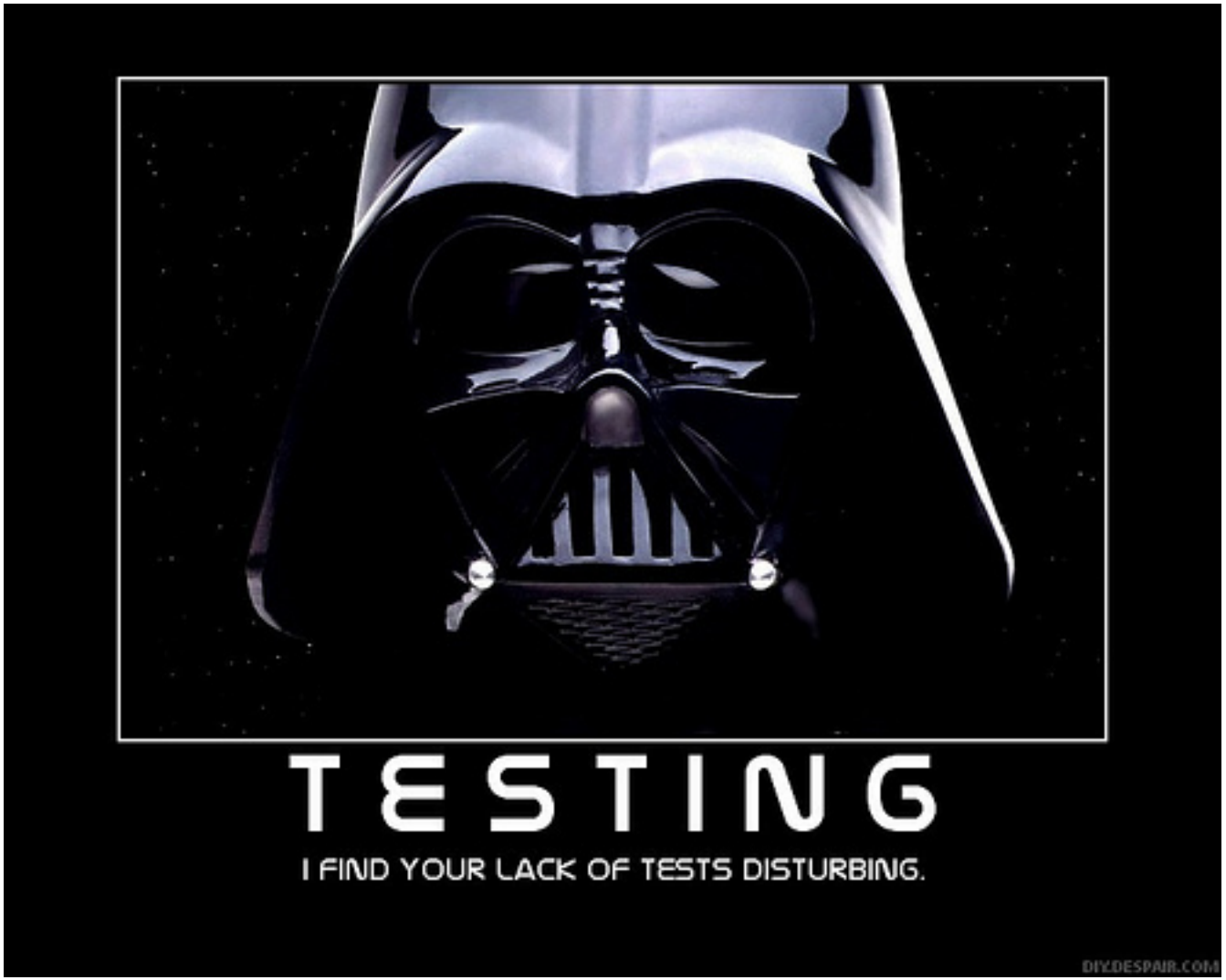
Unit Testing Tools

- Requirements
  - A driver: something to simulate usage of the unit being tested
  - Data stubs: test data the the unit requires
  - Asserts: check that the unit behaves as expected
  - Automation: enable “continuous” testing

Unit Testing Tools

- Many tools (“unit testing frameworks”) available to help with these requirements:
- C/C++: CuTest, CppUnit, Boost Test, FCTX
- Fortran: fUnit, FRUIT
- MATLAB: MlUnit, MATLAB xUnit
- Python: nose
- http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Testing…


TESTING
I FIND YOUR LACK OF TESTS DISTURBING.
DIVEDESPAIR.COM

Trust No One… not even yourself

- Unit Testing
  - Tests individual units of source code
  - A Unit is the smallest testable part of a code
  - Each Unit test should be independent from others
  - Write the unit and the test at the same time

Trust No One… not even yourself

- Regression Testing
  - Example applications, unit tests, benchmark problems
  - Catches unintended consequences of revisions
  - Design of a suite of regressions tests is an art
  - Consider Code and Feature Coverage
  - Test early and test often
  - Automate testing

Verification: Finding the Unknown Unknowns

- Solution Verification: Estimating Numerical Error
  - Discretization Error
  - Iterative Error
  - Round-off Error
- These errors cannot be avoided, but can (and must!) be quantified

Verification: Finding the Unknown Unknowns

- Code Verification
  - Software bugs
  - Numerical algorithm weaknesses
  - Model implementation mistakes
- Codes cannot practically be proven error-free, but can be proven to have errors. So we try our best to do the latter…

Example: Hypersonic Flow App

Math, Software Components

- Discretized Formulation
- Spatial Discretization
- Time Discretization
- System Assembly
- Nonlinear Solver
- Linear Solver
- I/O
- Postprocessing

Approximate LoC

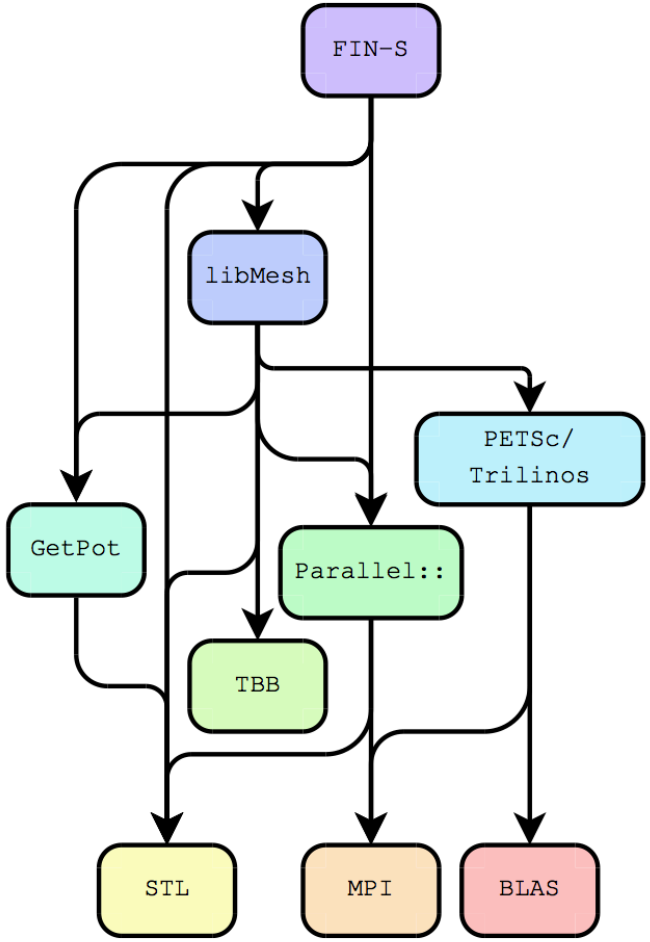
`DPLR`	46,000
Coupling	6,000
Ablation	6,000
Radiation	6,000
`FIN-S`	6,000
`libMesh`	54,000
`PETSc`	170,000
Contrib	56,000

Code Reuse

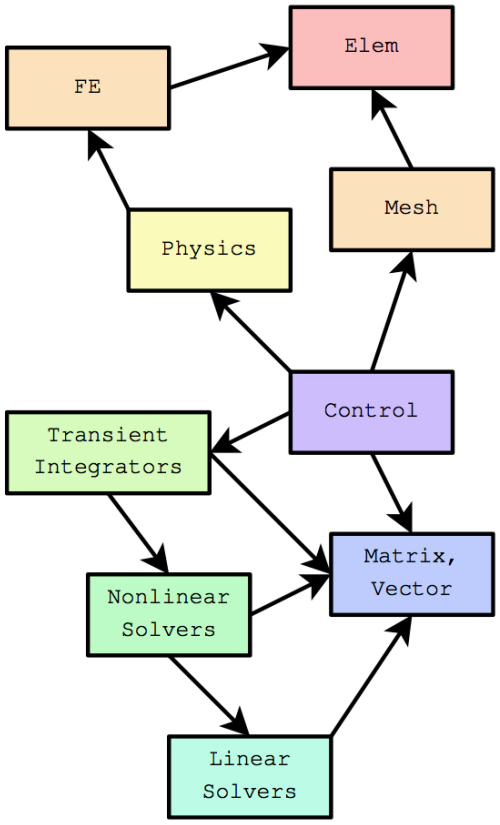
Examples

- MPI, BLAS
- Aztec, MUMPS, PETSc
- deal.II, FEniCS, libMesh

- Don't reinvent the wheel unnecessarily!
- Time spent rewriting something old is time that could have been spent writing something new.
- More eyes == fewer bugs
- Extend existing capabilities where possible.



Modular Programming



Discrete Components, Interfaces

- Linear, nonlinear solvers are discretization-independent
- System assembly, solution I/O & postprocessing can be discretization-independent
- Time, space discretizations can be physics-independent
- Some error analysis, sensitivity methods can be physics-independent

- Reusable components get re-tested
- Errors too subtle to find in complex physics are easy to spot in benchmark problems.

Unit Tests

- Testing One Object At A Time
  - Reusable modules interact with all other code through a limited API
  - That API can be tested directly outside of application code
  - Test one method at a time, isolate problems locally
- Revise Software => Rerun Tests

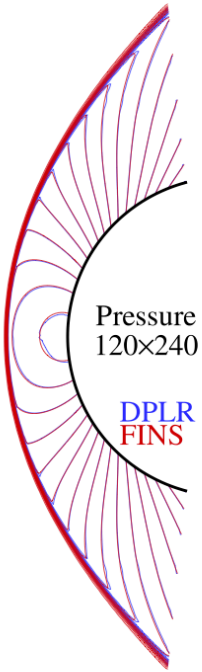
Verification Benchmark Problems

- Choosing Test Problems
- Known solutions
  - Exact solution to discretized problem
  - Limit solution of continuous problem
  - Known quantities of interest
- Known asymptotic convergence rates
- Known residuals

Code to Code Comparisons

"Lumped" Verification

- Differing results from:
  - Different models
  - Different formulations
  - Different discretizations



Hierarchic Models

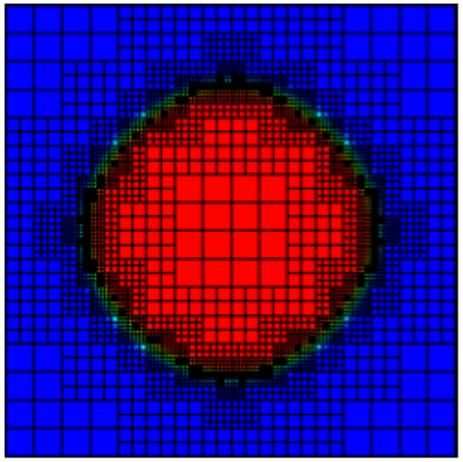
Per-Operator Tesing

- Coefficients selectively “turn off” parts of equations
- Allows easier construction of analytic solutions
- Assists with “narrowing down” other bugs

Model Simplification

- Model linearity can be tested in solver
- Reduce complex physics to simple physics case
- Code-to-code testing

Symmetry Tests



Symmetry In, Symmetry Out

- Mirror, radial symmetries
- Beware unstable solution modes!

Jacobian Verification

Jacobian Verification

Test Analytic vs. Numeric Jacobians

- Relative error in matrix norm
- If match isn't within tolerance, either:
  - The discretization or floating point error has overwhelmed the finite differenced Jacobian
    - Unlikely for good choices of finite difference perturbations
    - Can be investigated
  - The residual is non-differentiable at that iterate
    - Can be checked analytically
  - The Jacobian calculation is wrong
  - The residual calculation is wrong

A Priori Asymptotic Convergence Rates

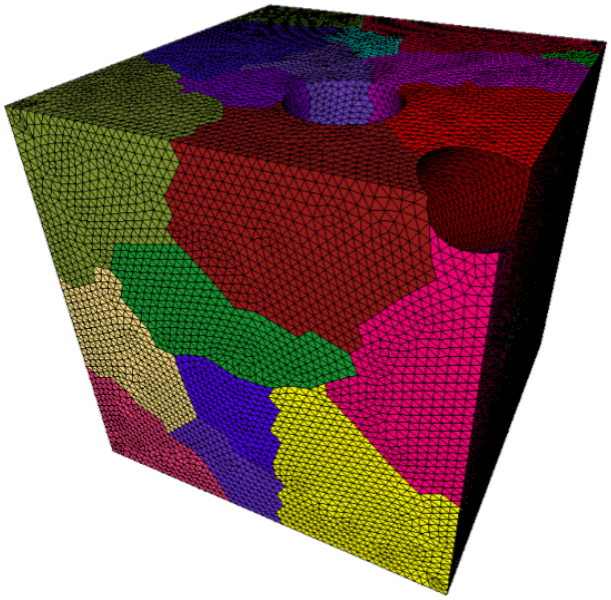
Pros

- Applicable to wide ranges of problems
- Does not require exact, or even manufactured, solution
- Part of solution verification, not just code verification

Cons

- Does not verify physics
- Requires asymptotic assumption
- Part of solution verification, not just code verification

Verification Never Ends



New Problems

- New Optimizations
- New Features
- New Combinations
- New Physics
- New Data
- New Requirements

A test suite that is never run is useless

- Automate testing
  - make check
  - svn hooks
  - crontab
  - Continuous Integration Server
- Standard procedure
  - Grab latest source code
  - Verify fresh configuration and compile
  - Verify solutions via regression tests
  - References solutions (and changelog) included in repository

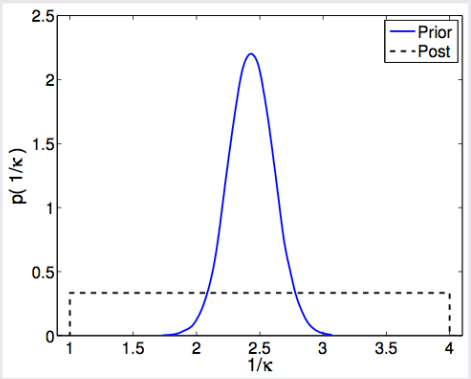
Example output

```
./rtest.sh
[...]
------------------------------------------------------------
PECOS 1D Ablation Model: Version = 0.20
Build Date   = 2009-05-02 11:59
Build Host   = sqa.ices.utexas.edu
Build Arch   = x86_64
Build Rev    = r2650
Build Status = current
------------------------------------------------------------
[...]
Verifying solution differences using a tolerance of: 1e-08

CN           => Identical
O            => Identical
CO           => Identical
Recession    => Identical
Temperature  => Identical
NO           => Identical
C3           => Within tolerance (3.60731569E-19,3.62386249E-19)

Thank you, drive thru...

------------------------------------------------------------------------
(rtest): PASSED: Test 1 (graphite/10species)
------------------------------------------------------------------------
```

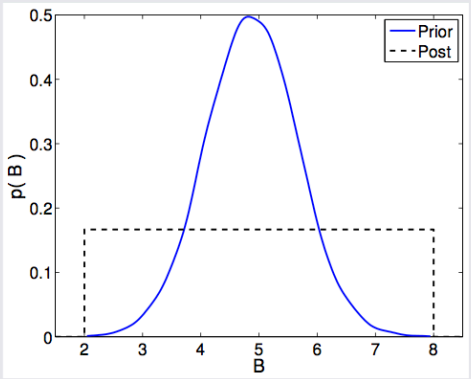
Example loglayer regression test

- Initial results for universal log law with Gaussian likelihood
- Good test case because know a lot about the posterior

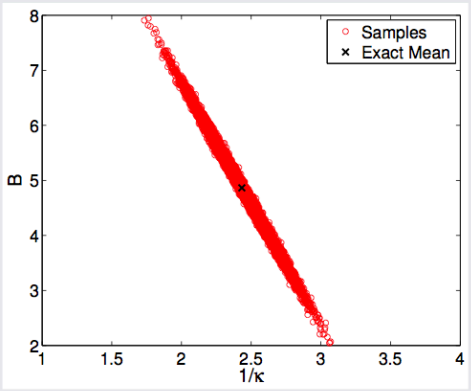
$1/\kappa$ Posterior



B Posterior



Joint Posterior



- Sample mean within 0.05% of exact posterior mean
- Appears to be working, but require more complete statistical analysis

Example loglayer regression test

Main Idea

When you are happy with a result, *generate a new regression test*

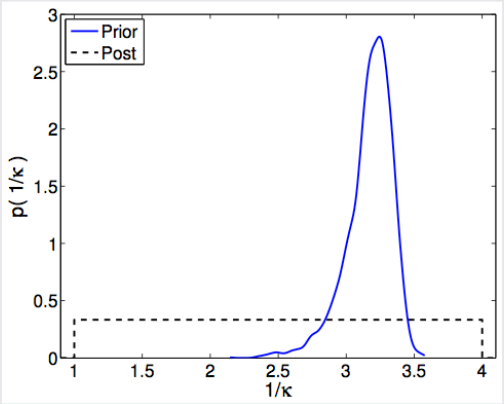
Procedure

- Create place for the test to live, e.g.,
  `cd test/system; svn mkdir universalLog_Gaussian;`
- Store “truth” output—i.e., run the case and save the output
- Generate method to run case and compare output to truth, often a shell script is all you need
- Add case to `make check` (update `TESTS` in `Makefile.am`)

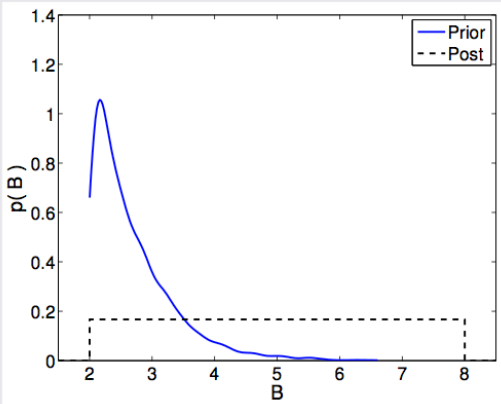
Example loglayer regression test

Initial Results with Non-Gaussian Likelihood

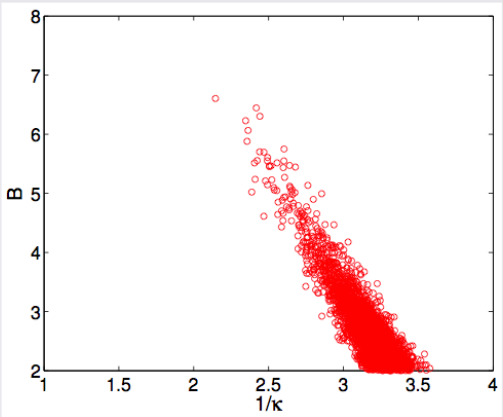
$1/\kappa$ Posterior



B Posterior



Joint Posterior



Do not know exact result (cannot even write likelihood in closed-form), but this does not look right at all

Verification is iterative and ongoing

At least two possibilities:

- The observed phenomenon is real and must be understood
- The observed phenomenon is caused by a bug

Debugging Philosophy

- Be skeptical of odd results—assume a bug and do everything to can to test for it
- View debugging as an opportunity to add new tests
- Once bug is found, add test that caught it

Application to LogLayer

- Only difference between good & bad results is likelihood
- Multiple unit tests in likelihood routines helped to narrow possibilities

The Bug and Updated Test Suite

Broken Code

```
for (unsigned int ii=1; ii<N; ++ii)
  {
    const double yrat = p->expData->yPlus(ii)/p->expData->yPlus(ii-1);
    vp[ii] = modelRoutine<Mod>( *(p->physParams), yrat*zp, p->expData->deltaPlus() );
  }
```

Correct Code

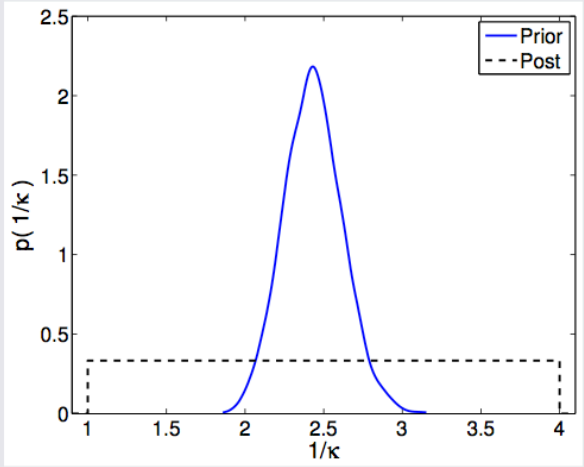
```
for (unsigned int ii=1; ii<N; ++ii)
  {
    const double yrat = p->expData->yPlus(ii)/p->expData->yPlus(0);
    vp[ii] = modelRoutine<Mod>( *(p->physParams), yrat*zp, p->expData->deltaPlus() );
  }
```

New Test

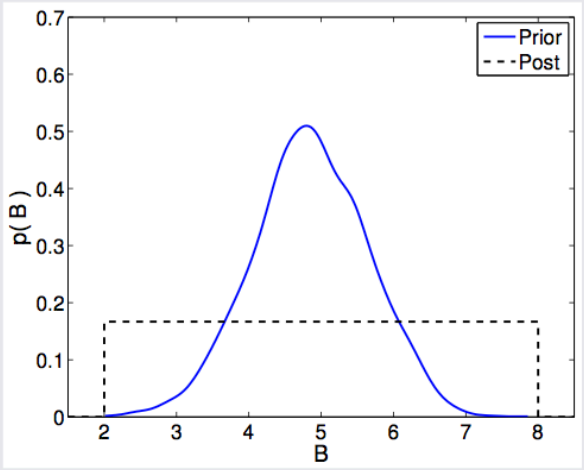
- Created unit test that checks output of function against saved output
- Not perfect, but better than nothing

Updated Results

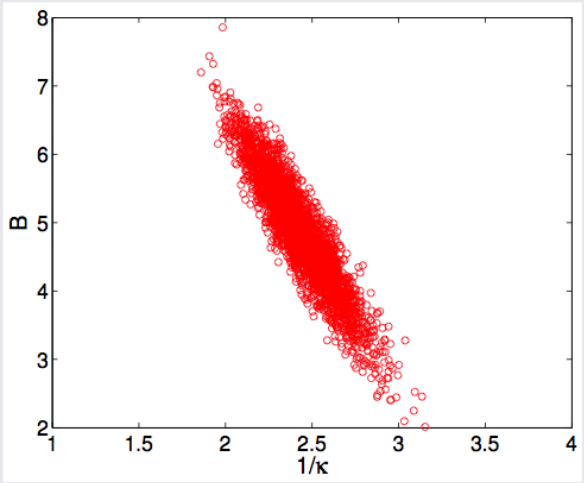
$1/\kappa$ Posterior



B Posterior



Joint Posterior



Appear much more reasonable—no drastic shift from Gaussian result

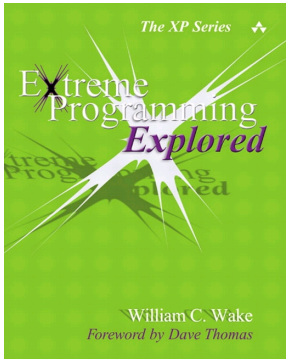
What is TDD?

- TDD is a technique whereby you write your test cases before you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of "intent"
  - Tests provide a specification of "what" a piece of code actually does
  - Some might argue that "tests are part of the documentation"

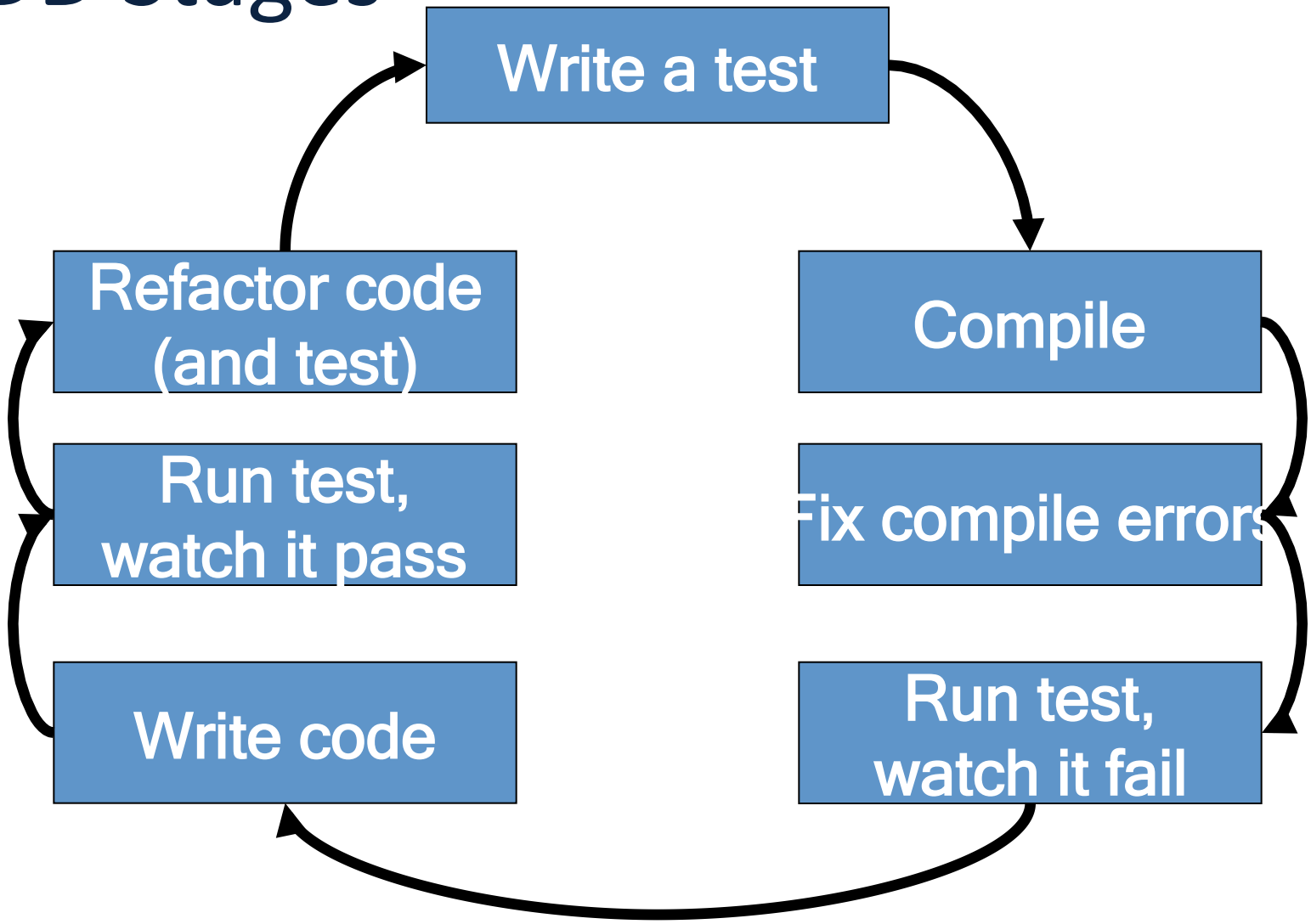
TDD Stages

- In Extreme Programming Explored (The Green Book), Bill Wake describes the test / code cycle:

1. Write a single test
2. Compile it. It should not compile because you've not written the implementation code
3. Implement just enough code to get the test to compile
4. Run the test and see it **fail**
5. Implement just enough code to get the test to pass
6. Run the test and see it **pass**
7. **Refactor** for clarity and "once and only once"
8. Repeat



TDD Stages


Write a test
Compile
Fix compile errors
Run test, watch it fail
Write code
Run test, watch it pass
Refactor code (and test)

Test Complexity

- Do the simplest thing
  - Dave Astels: “Strive for simplicity”
  - Write a test that fails (red)
  - Make the test pass (green)
  - **Refactor** implementation code (change the internal design)
- Use the compiler – let it tell you about errors and omissions
- One assertion (Check/Assert) per test
  - Subject of furious debate on Yahoo’s TDD group

Why TDD?

- Programmers dislike testing
  - They will test reasonably thoroughly the first time
  - The second time however, testing is usually less thorough
  - The third time, well..
- Testing is considered a “boring” task
- Testing might be the job of another department / person
- TDD encourages programmers to maintain an exhaustive set of repeatable tests
  - Tests live alongside the Class/Code Under Test (CUT)
  - With tool support, tests can be run selectively
  - The tests can be run after every single change

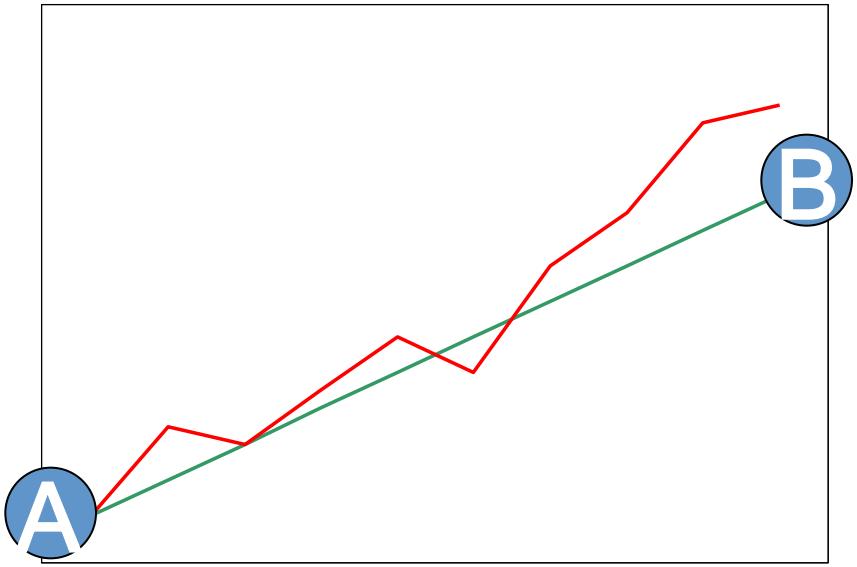
Why TDD?

- Bob Martin:
  - “The act of writing a unit test is more an act of design than of verification”
- Confidence boost
  - By practicing TDD, developers will strive to improve their code – without the fear that is normally associated with code changes
  - Isn’t the green bar a feel good factor?
- Remove / Reduce reliance on the debugger
  - No more “debug-later” attitudes

Who should write the tests?

- The programmers should write the tests
  - The programmers can't wait for somebody else to write tests

- TDD promotes "small steps", and lots of them
  - Small steps: the shortest distance between two points
  - Your destination is closer…



Summary

- TDD does not replace traditional testing
  - It defines a proven way that ensures effective unit testing
  - Tests are working examples of how to invoke a piece of code
    - Essentially provides a working specification for the code
- No code without tests
- Tests determine, or dictate, the code

Summary

- TDD isn't new
  - To quote Kent Beck:
    - "…you type the expected output tape from a real input tape, then code until the actual results matched the expected result…"
- TDD means less time spent in the debugger
- TDD negates fear
  - Fear makes developers communicate less
  - Fear makes developers avoid repeatedly testing code
    - Afraid of negative feedback

Fear is the…

Bugs

TDD

Summary

- TDD promotes the creation of a set of "programmer tests"
  - Automated tests that are written by the programmer
  - Exhaustive
  - Can be run over and over again
- TDD allows us to **refactor**, or change the implementation of a class, without the fear of breaking it
  - TDD and refactoring go hand-in-hand

Closing thoughts…

- Show other people your code early and often
- Reuse good software and evaluate multiple options
- Consider performance versus portability
- Don’t write your own input parser
- Don’t reinvent the wheel
- Don’t write anything until you evaluate options