Test Driven Development of Scientific Models
SEA Conference - April 7-11, 2014
Boulder, CO

Tom Clune

Advanced Software Technology Group
Computational and Information Sciences and Technology Office
NASA Goddard Space Flight Center
Outline

1 Motivations
2 Testing
3 Testing Frameworks
4 Test-driven Development (TDD)
5 What about numerical software?
Motivation 1: Fear/Stress
Motivation 1: Fear/Stress

photomatt7.wordpress.com
Motivation 1: Fear/Stress
Motivation 1: Fear/Stress

Calvin & Hobbes - Bill Waterson
Motivation 2: Productivity
Motivation 2: Productivity

- New feature
- Refactor

Change   Verify
Motivation 2: Productivity

- New feature
- Refactor
- Compiles?

Change
Verify
Motivation 2: Productivity

- New feature
- Refactor
- Compiles?
- Executes?

Change  Verify
Motivation 2: Productivity

- New feature
- Refactor
- Change
- Verify
- Compiles?
- Executes?
- Looks ok?
Motivation 2: Productivity

- New feature
- Refactor

- Compiles?
- Executes?
- Looks ok?
- Really correct?
Motivation 2: Productivity

- New feature
- Refactor

- Compiles?
- Executes?
- Looks ok?
- Really correct?
Motivation 2: Productivity

What is the latency of verification for large scientific models?
Some observations about human behavior:

- Risk of defects scales with magnitude of change per iteration
- Development time per iteration will be comparable to verification time
Some observations about human behavior:

- Risk of defects scales with magnitude of change per iteration
- Development time per iteration will be comparable to verification time

Conclusion:
*Productivity is a nonlinear function of the cost of verification!*
Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:

---

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)
- Implications are politically sensitive/divisive

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)
- Implications are politically sensitive/divisive
- **Scientific integrity is essential**

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)
- Implications are politically sensitive/divisive
- **Scientific integrity is essential!**

Software management and testing have not kept pace

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100$ trillion\(^1\)
- Implications are politically sensitive/divisive
- Scientific integrity is essential!

Software management and testing have not kept pace
- Strong validation against data, but ...
Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)
- Implications are politically sensitive/divisive
- **Scientific integrity is essential!**

Software management and testing have not kept pace
- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems

---
Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100$ trillion\(^1\)
- Implications are politically sensitive/divisive
- **Scientific integrity is essential!**

Software management and testing have not kept pace
- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
- Validation cannot detect certain types of software defects:

---

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100 trillion\(^1\)
- Implications are politically sensitive/divisive
- Scientific integrity is essential!

Software management and testing have not kept pace
- Strong validation against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
- Validation cannot detect certain types of software defects:
  * Those that are only exercised in rare/future regimes

Motivation 3: The Limelight

Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100$ trillion\(^1\)
- Implications are politically sensitive/divisive
- **Scientific integrity is essential!**

Software management and testing have not kept pace
- Strong *validation* against data, but ...  
- Validation is a blunt tool for isolating issues in coupled systems  
- Validation cannot detect certain types of software defects:  
  - Those that are only exercised in rare/future regimes  
  - Those which change results below detection threshold

---

Outline

1 Motivations

2 Testing

3 Testing Frameworks

4 Test-driven Development (TDD)

5 What about numerical software?
Test Harness - work in safety

Collection of tests that constrain system
Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- Localizes defects
Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- Localizes defects
- Improves developer confidence
Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- Localizes defects
- Improves developer confidence
- Decreases risk from change
Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- Localizes defects
- Improves developer confidence
- Decreases risk from change
- Inexpensive compared to application (ideally)
Do you write legacy code?
Do you write legacy code?

“There’s the main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests.”

Michael Feathers

*Working Effectively with Legacy Code*
Do you write legacy code?

“The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests.”

Michael Feathers

*Working Effectively with Legacy Code*

“Fear is the path to the dark side. Fear leads to anger. Anger leads to hate. Hate leads to suffering.” - Yoda
Do you write legacy code?

“The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests.”

Michael Feathers
*Working Effectively with Legacy Code*

- Lack of tests leads to fear of introducing subtle bugs and/or changing things inadvertently.
- Also is a barrier to involving pure software engineers in the development of our models.
Excuses, excuses ...
Excuses, excuses ...

- Takes too much time to write tests
Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
- Don’t know correct behavior
Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
- Don’t know correct behavior

http://java.dzone.com/articles/unit-test-excuses
- James Sugrue

- Numeric/scientific code cannot be tested, because ...
Desirable attributes for tests:
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
Desirable attributes for tests:

- **Narrow/specific**
  - Failure of a test localizes defect to small section of code.
- **Orthogonal to other tests**
  - Each defect causes failure in one or only a few tests.
- **Complete**
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
Desirable attributes for tests:

- **Narrow/specific**
  - Failure of a test localizes defect to small section of code.
- **Orthogonal to other tests**
  - Each defect causes failure in one or only a few tests.
- **Complete**
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- **Independent - No side effects**
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does not terminate execution.
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.
- Automated and repeatable
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.
- Automated and repeatable
- Clear intent
Outline

1. Motivations
2. Testing
3. Testing Frameworks
4. Test-driven Development (TDD)
5. What about numerical software?
Testing Frameworks

Diagram showing the testing process with layers:
- Framework Driver
- User Tests
- Framework Services
- Application

Report:
1271 tests run
2 Failures
Testing Frameworks

- Key services
  - Provide methods to succinctly express expected values
    - Call `assertEqual(120, factorial(5))`
  - Register test procedures with framework
  - Execute test procedures, and summarize success/failure

```
assertEqual(120, factorial(5))
```
Testing Frameworks

- Key services
  - Provide methods to succinctly express expected values
    - call `assertEqual(120, factorial(5))`
  - Register test procedures with framework
  - Execute test procedures, and summarize success/failure
- Generally specific/customized to programming language (xUnit)
  - Java (JUnit)
  - Python (pyUnit)
  - C++ (cxxUnit, cppUnit)
  - Fortran (FRUIT, FUNIT, pFUnit)
Frameworks and IDE’s

Frameworks are often integrated within IDEs for even greater ease of use:
Outline

1. Motivations
2. Testing
3. Testing Frameworks
4. Test-driven Development (TDD)
5. What about numerical software?
Today I am here to sell you something ...
Old paradigm:
- Tests written by separate team (black box testing)
- Tests written after implementation
Old paradigm:
- Tests written by separate team (black box testing)
- Tests written after implementation

Consequences:
- Testing schedule compressed for release
- Defects detected late in development ($$)
Old paradigm:
- Tests written by separate team (black box testing)
- Tests written after implementation

Consequences:
- Testing schedule compressed for release
- Defects detected late in development ($$)

New paradigm - Test-driven development (TDD)
- Developers write the tests (white box testing)
- Tests written before production code
- Enabled by emergence of strong unit testing frameworks
The TDD cycle

- Extend Tests
- Fix/Extend Production Code
- Run Tests
- Refactor

Focus on interface

Focus on algorithm

Success?

Pass

Fail
Benefits of TDD
Benefits of TDD

- High reliability - (excellent test coverage)
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
- Tests act as maintainable documentation
  - Tests show real use case scenarios
  - Tests are continuously exercised (TDD process)
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Tests show real use case scenarios
  - Tests are continuously exercised (TDD process)
- Reduced stress / improved confidence
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
- Tests act as maintainable documentation
  - Tests show real use case scenarios
  - Tests are continuously exercised (TDD process)
- Reduced stress / improved confidence
- Improved productivity
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
- Tests act as maintainable documentation
  - Tests show real use case scenarios
  - Tests are continuously exercised (TDD process)
- Reduced stress / improved confidence
- Improved productivity
- Predictable schedule
Benefits of TDD

- High reliability - (excellent test coverage)
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Tests show real use case scenarios
  - Tests are continuously exercised (TDD process)
- Reduced stress / improved confidence
- Improved productivity
- Predictable schedule
- **High quality implementation?**
  - Emphasis on interfaces
  - Testable code is cleaner code.
Outline

1. Motivations
2. Testing
3. Testing Frameworks
4. Test-driven Development (TDD)
5. What about numerical software?
Unique testing challenges of numerical software
Unique testing challenges of numerical software

- Presence of numerical error (roundoff or truncation)
Unique testing challenges of numerical software

- Presence of numerical error (roundoff or truncation)
- Lack of known (nontrivial) solutions
Unique testing challenges of numerical software

- Presence of numerical error (roundoff or truncation)
- Lack of known (nontrivial) solutions
- Irreducible complexity?
Unique testing challenges of numerical software

- Presence of numerical error (roundoff or truncation)
- Lack of known (nontrivial) solutions
- Irreducible complexity?
- Stability - issues that occur after long integrations
Unique testing challenges of numerical software

- Presence of numerical error (roundoff or truncation)
- Lack of known (nontrivial) solutions
- Irreducible complexity?
- Stability - issues that occur after long integrations
- Emergent properties of coupled systems (including stability)
Numerical error

Testing numerical algorithms requires an \textit{accurate} estimate for tolerance:
Numerical error

Testing numerical algorithms requires an *accurate* estimate for tolerance:

- If too *low*, then test fails for uninteresting reasons.
Numerical error

Testing numerical algorithms requires an accurate estimate for tolerance:

- If too low, then test fails for uninteresting reasons.
- If too high, then the test has no teeth.
Numerical error

Testing numerical algorithms requires an *accurate* estimate for tolerance:

- If too *low*, then test fails for uninteresting reasons.
- If too *high*, then the test has no teeth.

Unfortunately ...

- Error estimates are seldom available for complex algorithms
Numerical error

Testing numerical algorithms requires an _accurate_ estimate for tolerance:
- If too _low_, then test fails for uninteresting reasons.
- If too _high_, then the test has no teeth.

Unfortunately ...
- Error estimates are seldom available for complex algorithms
- Best case scenario is usually some asymptotic form with unknown leading coefficient!
TDD techniques in presence of numerical error
TDD techniques in presence of numerical error

Sources:
TDD techniques in presence of numerical error

Sources:

1. Approximation
TDD techniques in presence of numerical error

Sources:

- Approximation
- Nonlinearity - e.g., small denominators
TDD techniques in presence of numerical error

Sources:

- Approximation
- Nonlinearity - e.g., small denominators
- Composition and iteration
TDD techniques in presence of numerical error

Sources:

- Approximation
- Nonlinearity - e.g., small denominators
- Composition and iteration

Mitigation strategies:
TDD techniques in presence of numerical error

Sources:

- Approximation
- Nonlinearity - e.g., small denominators
- Composition and iteration

Mitigation strategies:

- Approximation:
  - Test the implementation not the math (i.e., duck)
  - Often more appropriate as validation test
TDD techniques in presence of numerical error

Sources:
1. Approximation
2. Nonlinearity - e.g., small denominators
3. Composition and iteration

Mitigation strategies:
1. Approximation:
   ▶ Test the implementation not the math (i.e., duck)
   ▶ Often more appropriate as validation test
2. Nonlinearity - use tailored synthetic inputs:
   ▶ E.g., choose values to make denominators O(1)
TDD techniques in presence of numerical error

Sources:
1. Approximation
2. Nonlinearity - e.g., small denominators
3. Composition and iteration

Mitigation strategies:
1. Approximation:
   ▶ Test the implementation not the math (i.e., duck)
   ▶ Often more appropriate as validation test
2. Nonlinearity - use tailored synthetic inputs:
   ▶ E.g., choose values to make denominators O(1)
3. Composition/iteration: test steps in isolation:
   ▶ Allows choice of tailored synthetic inputs at each step
   ▶ Test iteration logic not accumulation
Example - testing layers in isolation

Consider the main loop of a climate model:

**Do test**
- Proper # of iterations
- Pieces called in correct order
- Passing of data between components

**Do NOT test**
- Calculations inside components

Easier with *objects* than with procedures.
TDD without “known” solutions

Consider the apparent contradiction:
TDD without “known” solutions

Consider the apparent contradiction:
- Complex algorithms yield few nontrivial analytic solutions.
- Implementations are not random keystrokes
TDD without “known” solutions

Consider the apparent contradiction:

- Complex algorithms yield few nontrivial analytic solutions.
- Implementations are not random keystrokes

How can this be?

- Apparently analytic solutions are *unnecessary*!
- Algorithms are only sequences of steps
TDD without “known” solutions

Consider the apparent contradiction:
- Complex algorithms yield few nontrivial analytic solutions.
- Implementations are not random keystrokes

How can this be?
- Apparently analytic solutions are unnecessary!
- Algorithms are only sequences of steps

**Tests should only verify translation, not validity of algorithms**
- Test each step in isolation
- Tailor synthetic inputs to yield “obvious” results for each step
- Separately test that steps are composed correctly
TDD without “known” solutions

Consider the apparent contradiction:
- Complex algorithms yield few nontrivial analytic solutions.
- Implementations are not random keystrokes

How can this be?
- Apparently analytic solutions are unnecessary!
- Algorithms are only sequences of steps

Tests should only verify translation, not validity of algorithms
- Test each step in isolation
- Tailor synthetic inputs to yield “obvious” results for each step
- Separately test that steps are composed correctly

But still use high level analytic solutions as tests when available!
TDD and irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests just repeating logic in the implementation?”
TDD and irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests just repeating logic in the implementation?”

- Short answer: No
TDD and irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests just repeating logic in the implementation?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
TDD and irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests just repeating logic in the implementation?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
  - Unit tests use tailored inputs
  - Implementation handles arbitrary values
TDD and irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests just repeating logic in the implementation?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
  - Unit tests use tailored inputs
  - Implementation handles arbitrary values
  - Models *couple* many components/algorithms ⇒ exponential complexity
  - Tests are *decoupled* ⇒ linear complexity
TDD and emergent properties

- TDD generally does not directly address such issues
TDD and emergent properties

- TDD generally does not directly address such issues
- If long integration gets bad results, (at least) one of the following must hold:
TDD and emergent properties

- TDD generally does not directly address such issues
- If long integration gets bad results, (at least) one of the following must hold:
  - Individual steps have defects $\Rightarrow$ add unit tests
TDD and emergent properties

- TDD generally does not directly address such issues
- If long integration gets bad results, (at least) one of the following must hold:
  1. Individual steps have defects ⇒ add unit tests
  2. Coupling/compositions have defects ⇒ add tests
TDD and emergent properties

- TDD generally does not directly address such issues.
- If long integration gets bad results, (at least) one of the following must hold:
  1. Individual steps have defects ⇒ add unit tests
  2. Coupling/compositions have defects ⇒ add tests
  3. System lacks sufficient accuracy ⇒ increase accuracy
TDD and emergent properties

- TDD generally does not directly address such issues
- If long integration gets bad results, (at least) one of the following must hold:
  1. Individual steps have defects $\Rightarrow$ add unit tests
  2. Coupling/compositions have defects $\Rightarrow$ add tests
  3. System lacks sufficient accuracy $\Rightarrow$ increase accuracy
  4. Insufficient physical fidelity - science issue (testing is not magic)
TDD and emergent properties

- TDD generally does not directly address such issues
- If long integration gets bad results, (at least) one of the following must hold:
  1. Individual steps have defects ⇒ add unit tests
  2. Coupling/compositions have defects ⇒ add tests
  3. System lacks sufficient accuracy ⇒ increase accuracy
  4. Insufficient physical fidelity - science issue (testing is not magic)

- At the very least, TDD can reduce the frequency with which one must perform long integrations
TDD and performance

- TDD emphasizes small fine-grained implementations
- Such implementations are often sub-optimal in terms of performance
- Optimized implementations typically fuse multiple operations
TDD and performance

- TDD emphasizes small fine-grained implementations
- Such implementations are often sub-optimal in terms of performance
- Optimized implementations typically fuse multiple operations
- Solution: bootstrapping
  - Use initial TDD solution as unit test for optimized implementation
  - Maintain both implementations (and tests)
TDD and the burden of legacy code

- TDD was created for developing *new* code, and does not directly speak to testing legacy code.
- Best practice for incorporating new functionality:
  - Avoid *wedging* new logging directly into existing large procedure
  - Use TDD to develop separate facility for *new* computation
  - Just *call* the new procedure from the large legacy procedure
- Refactoring
  - Use unit tests to constrain existing behavior
  - Very difficult for large procedures
  - Try to find small pieces to pull out into new procedures
Acknowledgements

This work has been supported by

- NASA’s High End Computing (HEC) program and
- NASA’s Modeling, Analysis, and Prediction (MAP) Program.
Summary

- TDD can be applied to scientific models
- Tool support exists (unabashed plug for pFUnit tutorial)
- Cost/benefit analysis for numerical software needs further study

Tom Clune
Thomas.L.Clune@nasa.gov
http://pfunit.sourceforge.net
Test-Driven Development: By Example - Kent Beck