Test Driven Development of Scientific Models

Tom Clune

Software Systems Support Office
Earth Science Division
NASA Goddard Space Flight Center

June 5, 2012
1 Motivations

2 Testing

3 Testing Frameworks

4 Test-Driven Development

5 What about scientific/technical software?
The development cycle and productivity

Conventional software verification for modeling is slow.
The development cycle and productivity

Conventional software verification for modeling is slow.
The development cycle and productivity

- Extend
- Fix

Conventional software verification for modeling is slow.
The development cycle and productivity

- Extend
- Fix
- Port

Conventional software verification for modeling is slow.
The development cycle and productivity

- Extend
- Fix
- Port

Conventional software verification for modeling is **slow**.
The development cycle and productivity

- Extend
- Fix
- Port

Implement

Verify

Compiles?

Executes?

Conventional software verification for modeling is slow.
The development cycle and productivity

Conventional software verification for modeling is **slow**.
The development cycle and productivity

- Extend
- Fix
- Port

Conventional software verification for modeling is slow.
Some observations

- Risk grows with magnitude of implementation step
- Magnitude of implementation step grows with cost of verification/validation
Some observations

- Risk grows with magnitude of implementation step
- Magnitude of implementation step grows with cost of verification/validation

Conclusion:
Optimize productivity by reducing cost of verification!
Climate modeling has grown to be of extreme socioeconomic importance:
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 \text{ trillion}$
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100$ trillion
- Implications are politically sensitive/divisive
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100$ trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100\ trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace
Climate modeling has grown to be of extreme socioeconomic importance:
- Adaptation/mitigation strategies easily exceed $100$ trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace
- Strong *validation* against data, but ...
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- *Scientific integrity is crucial*

Software management and testing have not kept pace

- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100$ trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

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:
Climate modeling has grown to be of extreme socioeconomic importance:

▶ Adaptation/mitigation strategies easily exceed $100$ trillion
▶ Implications are politically sensitive/divisive
▶ Scientific integrity is crucial

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
Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100$ trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

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
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
- **Implements 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
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.

Programming on a tightrope

This is also a barrier to involving pure software engineers in the development of our models.

Tom Clune (SSSO)
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*
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.

- Programming on a tightrope
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.

- Programming on a tightrope

This is also a barrier to involving pure software engineers in the development of our models.
Excuses, excuses ...

- Takes too much time to write tests

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

-James Sugrue

Numeric/scientific code cannot be tested, because ...

Tom Clune (SSSO)
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

"Correct" behavior is unknown

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

James Sugrue

TDD - Testing
Tom Clune (SSSO)
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
- “Correct” behavior is unknown
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
- “Correct” behavior is unknown

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

- James Sugrue
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
- “Correct” behavior is unknown

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

- Numeric/scientific code cannot be tested, because …
Just what is a test anyway?

Tests can exist in many forms

- **Conditional termination:**
  
  ```
  IF (PA(I,J)+PTOP.GT.1200.) &
  call stop_model('ADVECM: Pressure diagnostic error',11)
  ```

- **Diagnostic print statement**
  
  ```
  print *, 'loss of mass = ', deltaMass
  ```

- **Visualization of output**

  ![Temperature and Difference Plots](image-url)
Analogy with Scientific Method?

Scientists ought to like TDD:

- **Objective reality** → **Requirements**
- **Constraints: theory and data** → **Constraints: existing tests**
- **Formulate hypothesis** → **Select a feature**
- **Design experiment** → **Write a test**
- **Run experiment** → **Run tests**
- **Refine hypothesis** → **Refine implementation**

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality $\rightarrow$ Requirements
Analogy with Scientific Method?

Scientists ought to like TDD:

- Objective reality $\rightarrow$ Requirements
- Constraints: theory and data $\rightarrow$ Constraints: existing tests
Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality $\rightarrow$ Requirements
Constraints: theory and data $\rightarrow$ Constraints: existing tests
Formulate hypothesis $\rightarrow$ Select a feature

Analogy with Scientific Method?

Scientists ought to like TDD:

- Objective reality $\rightarrow$ Requirements
- Constraints: theory and data $\rightarrow$ Constraints: existing tests
- Formulate hypothesis $\rightarrow$ Select a feature
- Design experiment $\rightarrow$ Write a test
Analogy with Scientific Method?

Scientists ought to like TDD:

- Objective reality $\rightarrow$ Requirements
- Constraints: theory and data $\rightarrow$ Constraints: existing tests
- Formulate hypothesis $\rightarrow$ Select a feature
- Design experiment $\rightarrow$ Write a test
- Run experiment $\rightarrow$ Run tests

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality $\rightarrow$ Requirements
Constraints: theory and data $\rightarrow$ Constraints: existing tests

Formulate hypothesis $\rightarrow$ Select a feature
Design experiment $\rightarrow$ Write a test
Run experiment $\rightarrow$ Run tests
Refine hypothesis $\rightarrow$ Refine implementation

Properties of good tests

- Isolating
  - Test failure indicates location in source code

- Orthogonal
  - Each defect results in failure of small number of tests

- Complete
  - Each bit of functionality covered by at least one test

- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution

- Frugal
  - Run quickly
  - Small memory, etc.

- Automated and repeatable
- Clear intent
Properties of good tests

- Isolating
  - Test failure indicates location in source code
Properties of good tests

- Isolating
  - Test failure indicates location in source code

- Orthogonal
  - Each defect results in failure of small number of tests

Tom Clune (SSSO)
TDD - Testing
June 5, 2012
Properties of good tests

- **Isolating**
  - Test failure indicates location in source code

- **Orthogonal**
  - Each defect results in failure of small number of tests

- **Complete**
  - Each bit of functionality covered by at least one test
Properties of good tests

- **Isolating**
  - Test failure indicates location in source code
- **Orthogonal**
  - Each defect results in failure of small number of tests
- **Complete**
  - Each bit of functionality covered by at least one test
- **Independent**
  - No side effects
  - Test order does not matter
  - Corollary: *cannot terminate execution*
Properties of good tests

- Isolating
  - Test failure indicates location in source code

- Orthogonal
  - Each defect results in failure of small number of tests

- Complete
  - Each bit of functionality covered by at least one test

- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution

- Frugal
  - Run quickly
  - Small memory, etc.
Properties of good tests

- **Isolating**
  - Test failure indicates location in source code

- **Orthogonal**
  - Each defect results in failure of small number of tests

- **Complete**
  - Each bit of functionality covered by at least one test

- **Independent**
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution

- **Frugal**
  - Run quickly
  - Small memory, etc.

- **Automated and repeatable**
Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test
- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution
- Frugal
  - Run quickly
  - Small memory, etc.
- Automated and repeatable
- Clear intent
Anatomy of a Software Test Procedure

```python
def testTrajectory(a, t):
    s = trajectory(a, t)
    assertEqual(9., s)
    assertEqual(9., trajectory(2., 3.))
```

Procedure `testFoo()`

1. Set Preconditions
2. Invoke System-under-test
3. Check Postconditions
4. Success?
   - Yes: Release Resources
   - No: Send Alert
Anatomy of a Software Test Procedure

\[
testTrajectory() \quad ! \quad s = \frac{1}{2} at^2
\]
testTrajectory() \[ s = \frac{1}{2} at^2 \]

\[ a = 2.; t = 3. \]
Anatomy of a Software Test Procedure

testTrajectory() \! s = \frac{1}{2} at^2

a = 2.; t = 3.

s = trajectory(a, t)
Anatomy of a Software Test Procedure

```
\text{testTrajectory()} ! \ s = \frac{1}{2} at^2
```

\[ a = 2.; \ t = 3. \]

\[ s = \text{trajectory}(a, t) \]

\text{call } \text{assertEqual} (9., s)
Anatomy of a Software Test Procedure

```
Procedure testFoo()

Set Preconditions
Invoke System-under-test
Check Postconditions

Success ?
   No   Send Alert
      Yes Release Resources

assertEqual (9., s)
```

testTrajectory() ! $s = \frac{1}{2} at^2$

a = 2.; t = 3.

$s = \text{trajectory}(a, t)$

call `assertEqual` (9., s)

! no op
Anatomy of a Software Test Procedure

**Procedure testFoo()**

1. **Set Preconditions**
2. **Invoke System-under-test**
3. **Check Postconditions**

- **Success?**
  - No → Send Alert
  - Yes → **Release Resources**

**testTrajectory()**

\[ s = \frac{1}{2} at^2 \]

Call **assertEqual**(9., trajectory (2.,3.))
Outline

1. Motivations
2. Testing
3. Testing Frameworks
4. Test-Driven Development
5. What about scientific/technical software?
Testing Frameworks

- Provide infrastructure to radically simplify:
  - Creating test routines (Test cases)
  - Running collections of tests (Test suites)
  - Summarizing results

- Key feature is collection of assert methods
  - Used to express expected results

```
call assertEqual(120, factorial(5))
```

- Generally specific to programming language (xUnit)
  - Java (JUnit)
  - Python (pyUnit)
  - C++ (cxxUnit, cppUnit)
  - Fortran (FRUIT, FUNIT, pFUnit)
GUI - JUnit in Eclipse

![JUnit GUI in Eclipse]

- **JUnit** Test class name: `com.jcorporate.espresso.core.ExpressoTestSuite`
- **Reload classes every run**
- **Run Test Suite**
  - Enter the name of the Test Case class: `MoneyTest`
  - Progress: 18 Runs, 0 Errors, 1 Failure
  - Errors and Failures:
    - Failure: `MoneyTest.testMixedSimpleAdd.expected = "([1 2 3])`
Outline

1. Motivations

2. Testing

3. Testing Frameworks

4. Test-Driven Development

5. What about scientific/technical software?
(Somewhat) New Paradigm: TDD

Old paradigm:
- Tests written by separate team (black box testing)
- Tests written *after* implementation
(Somewhat) New Paradigm: TDD

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 ($$)
(Somewhat) New Paradigm: TDD

Old paradigm:
- Tests written by separate team (black box testing)
- Tests written \textit{after} implementation

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

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

1. Focus on interface
   - Extend Tests
2. Fix/Extend Production Code
3. Run Tests
4. Success or Fail
   - Refactor
5. Pass

Focus on algorithm
Benefits of TDD

- High reliability
- Excellent test coverage
- Always "ready-to-ship"
- Tests act as maintainable documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
- Porting
Benefits of TDD

- High reliability
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
  - Test shows real use case scenario
  - Test is maintained through TDD process
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
- Porting
Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
- Porting
- **Quality implementation?**
Outline

1 Motivations

2 Testing

3 Testing Frameworks

4 Test-Driven Development

5 What about scientific/technical software?
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
  - Irreducible complexity
    - Test would require the same redundant logic
    - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
    - More generally - emergent properties of coupled systems

General mitigation strategy:
- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation

Stability/Nonlinearity
- Problems that occur only after long integrations
- More generally - emergent properties of coupled systems

General mitigation strategy:
- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases

General mitigation strategy:
- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?

General mitigation strategy:
- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation

- Insufficient analytic cases

- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?

- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems

General mitigation strategy:
Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems

General mitigation strategy:
- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation
Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:
Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

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

For testing numerical results, a good estimate for the tolerance is necessary:

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

For testing numerical results, a good estimate for the tolerance is necessary:

- 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
For testing numerical results, a good estimate for the tolerance is necessary:

- 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 - usually asymptotic form with unknown leading coefficient!
Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Mitigation

▶ Tailored synthetic inputs: eliminate/minimize roundoff from nonlinearity
▶ Test layers in isolation: circumvent growth from composition
▶ Put iteration logic in separate layer: circumvent growth from iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.
Numerical tolerance (cont’d)

Sources of roundoff

- Ordinary arithmetic - machine epsilon (not a concern)
- Nonlinearity - esp. small denominators
- Composition and iteration

Mitigation

▶ Tailored synthetic inputs: eliminate/minimize roundoff from nonlinearity
▶ Test layers in isolation: circumvent growth from composition
▶ Put iteration logic in separate layer: circumvent growth from iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Mitigation
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Mitigation

- Tailored synthetic inputs:
  - eliminate/minimize roundoff from nonlinearity
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

- **Tailored synthetic inputs:**
  eliminate/minimize roundoff from nonlinearity
- **Test layers in isolation:**
  circumvent growth from composition
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Mitigation

- **Tailored synthetic inputs**: eliminate/minimize roundoff from nonlinearity
- **Test layers in isolation**: circumvent growth from composition
- **Put iteration logic in separate layer**: circumvent growth from iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.

Tom Clune (SSSO)  TDD - What about scientific/technical software  June 5, 2012 28 / 38
Numerical tolerance (cont’d)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. Composition and iteration

Mitigation

- Tailored synthetic inputs: 
  eliminate/minimize roundoff from nonlinearity
- Test layers in isolation: 
  circumvent growth from composition
- Put iteration logic in separate layer: 
  circumvent growth from iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.
Test layers in isolation

Example: Procedure that does too much

\[
\begin{align*}
    a &= \langle \text{complex expression} \rangle \\
    b &= \langle \text{complex expression} \rangle \\
    c &= \langle \text{complex expression} \rangle \\
    \text{return } a + \sqrt{b/c}
\end{align*}
\]
Test layers in isolation

Example: Procedure that does too much

```python
... 
a = <complex expression>
b = <complex expression>
c = <complex expression>
return a + sqrt(b/c)
```

Same capability, but split into two decoupled levels

```python
... 
a = f1(...) 
b = f2(...) 
c = f3(...) 
return g(a, b, c)
```
Test layers in isolation

Example: Procedure that does too much

\[
\ldots
a = \langle \text{complex expression} \rangle \\
b = \langle \text{complex expression} \rangle \\
c = \langle \text{complex expression} \rangle \\
return a + \sqrt{b/c}
\]

Same capability, but split into two decoupled levels

\[
\ldots
a = f1(\ldots) \\
b = f2(\ldots) \\
c = f3(\ldots) \\
return g(a, b, c)
\]

Higher level test ensures proper coupling, but not fully expanded arithmetic.
Test layers in isolation (cont’d)

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

Much easier to do in practice with *objects* than with procedures.
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions

Mitigation:
- Test algorithmic steps in isolation
- Tailor synthetic inputs to yield "obvious" results for each step
- Use integration tests to verify that steps are composed correctly
- But still use high level analytic solutions as tests whenever possible

Consider Newton's three-body problem - no analytic solution
- Test generation of pairwise forces
- Test time integration (e.g., RK4)
- Use special cases that have solutions as additional tests
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?

Mitigation:

- Test algorithmic steps in isolation
- Tailor synthetic inputs to yield “obvious” results for each step
- Use integration tests to verify that steps are composed correctly
- But still use high-level analytic solutions as tests whenever possible

Consider Newton’s three-body problem - no analytic solution

- Test generation of pairwise forces
- Test time integration (e.g., RK4)
- Use special cases that have solutions as additional tests
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration

Mitigation:
- Test algorithmic steps in isolation
- Tailor synthetic inputs to yield "obvious" results for each step
- Use integration tests to verify that steps are composed correctly
- But still use high level analytic solutions as tests whenever possible

Consider Newton’s three-body problem - no analytic solution
- Test generation of pairwise forces
- Test time integration (e.g., RK4)
- Use special cases that have solutions as additional tests
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield “obvious” results for each step
  - Use integration tests to verify that steps are composed correctly

But still use high level analytic solutions as tests whenever possible

Consider Newton’s three-body problem - no analytic solution
Test generation of pairwise forces
Test time integration (e.g., RK4)
Use special cases that have solutions as additional tests
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield “obvious” results for each step
  - Use integration tests to verify that steps are composed correctly
- *But still use high level analytic solutions as tests whenever possible*

Consider Newton’s three-body problem - no analytic solution
TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield “obvious” results for each step
  - Use integration tests to verify that steps are composed correctly
- But still use high level analytic solutions as tests whenever possible

Consider Newton’s three-body problem - no analytic solution

- Test generation of pairwise forces
- Test time integration (e.g., RK4)
- Use special cases that have solutions as additional tests
Irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”
Irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”

- Short answer: No
Irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”

- Short answer: No
- Long answer: Well, they shouldn’t be ...
Irreducible complexity

“ Aren’t my tests as complex as the implementation?”
“ Aren’t my tests doing redundant calculations (tautological)?”

• Short answer:  No
• Long answer: Well, they shouldn’t be ...
  ▶ Unit tests use tailored inputs - implementation handles generic case
“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*
Irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*
  - Tests are *decoupled* - low complexity
Irreducible complexity

“Aren’t my tests as complex as the implementation?”
“Aren’t my tests doing redundant calculations (tautological)?”

- Short answer: **No**
- Long answer: Well, they shouldn’t be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*
  - Tests are *decoupled* - low complexity
  - Actual model *couples* layers - huge complexity
Long integration and emergent properties

- TDD generally does not directly address such issues

If long integration gets incorrect results, one of the following holds:

1. Individual steps have defects - add tests
2. Integration has a defect - add tests
3. Component steps lack necessary accuracy - need tests and improved algorithm
4. Insufficient physical fidelity - genuine science challenge

At the very least, TDD can reduce the frequency at which long integrations are needed/performed.
Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

At the very least, TDD can reduce the frequency at which long integrations are needed/performed.
Long integration and emergent properties

- TDD generally does not directly address such issues.
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

At the very least, TDD can reduce the frequency at which long integrations are needed/performed.
Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests

At the very least, TDD can reduce the frequency at which long integrations are needed/performed

Tom Clune (SSSO)  TDD - What about scientific/technical software  June 5, 2012  35 / 38
Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm

At the very least, TDD can reduce the frequency at which long integrations are needed/performed.
Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

At the very least, TDD can reduce the frequency at which long integrations are needed/performed
Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

- At the very least, TDD can reduce the frequency at which long integrations are needed/performed
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
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
TDD and the legacy burden

- TDD was created for developing new code, and does not directly speak to maintaining legacy code.

- Adding 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
References

- pFUnit: http://sourceforge.net/projects/pfunit/
- Tutorial materials
  - https://modelingguru.nasa.gov/docs/DOC-1982
  - https://modelingguru.nasa.gov/docs/DOC-1983
  - https://modelingguru.nasa.gov/docs/DOC-1984
- TDD Blog
  - https://modelingguru.nasa.gov/blogs/modelingwithtdd
- Test-Driven Development: By Example - Kent Beck
- Refactoring: Improving the Design of Existing Code - Martin Fowler
- JUnit http://junit.sourceforge.net/