Test Driven Development of Scientific Models

Tom Clune

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

May 1, 2012

Outline

The Tightrope Act

Software development should not feel like this

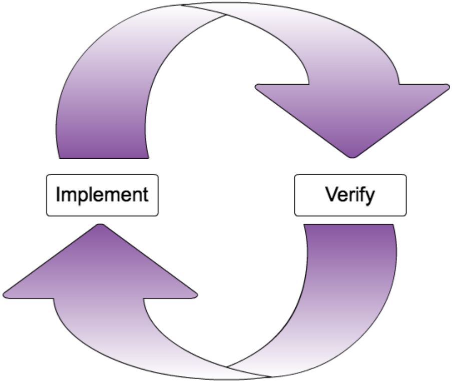
The Tightrope Act

... or even like this

The Tightrope Act

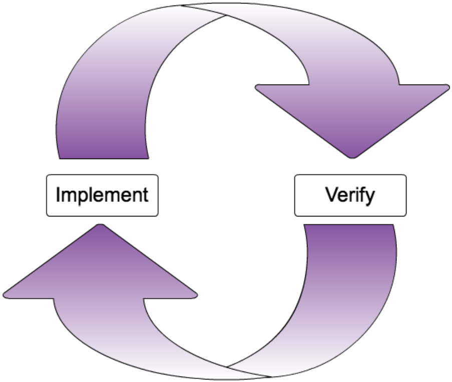
Hopefully something more like this

The Development Cycle



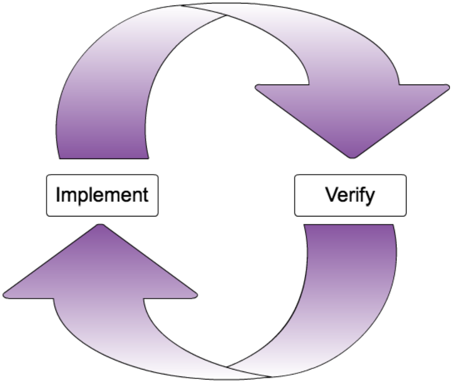
The Development Cycle

- **Extend**



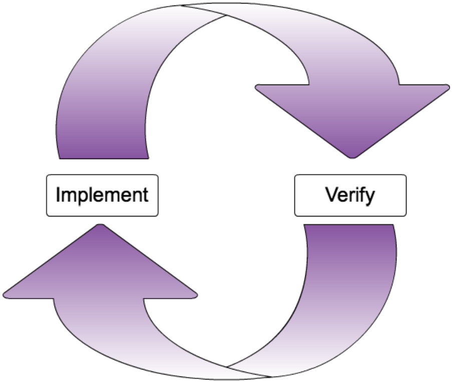
The Development Cycle

- Extend
- **Fix**



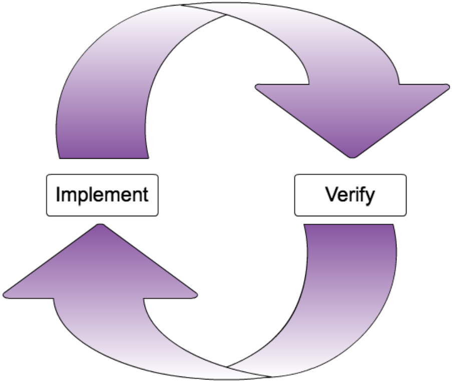
The Development Cycle

- Extend
- Fix
- **Port**



The Development Cycle

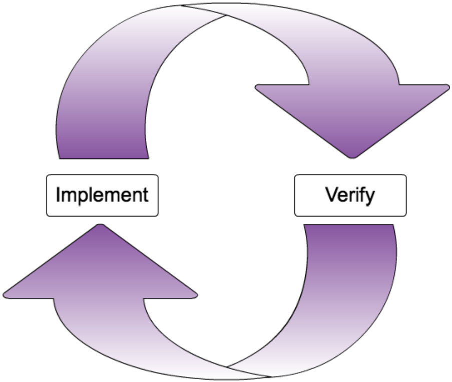
- Extend
- Fix
- Port



- **Compiles?**

The Development Cycle

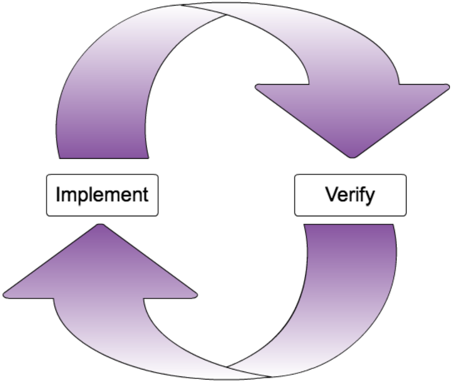
- Extend
- Fix
- Port



- Compiles?
- **Executes?**

The Development Cycle

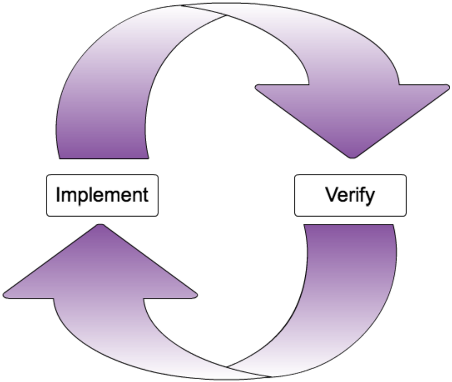
- Extend
- Fix
- Port



- Compiles?
- Executes?
- **Looks ok?**

The Development Cycle

- Extend
- Fix
- Port



- Compiles?
- Executes?
- Looks ok?
- **Correct?**

Natural Time Scales

- Design
- Edit source
- Compilation
- Batch
  waiting in queue
- Execution
- Analysis

Some observations

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

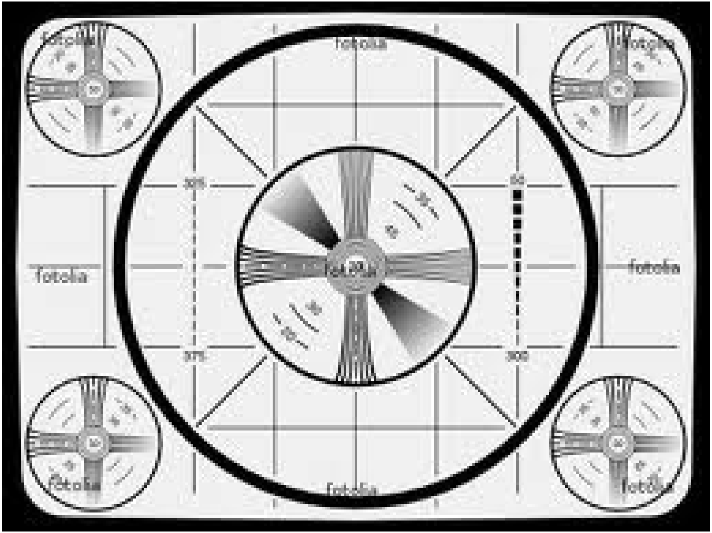
Some observations

- 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!

Outline

Testing



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

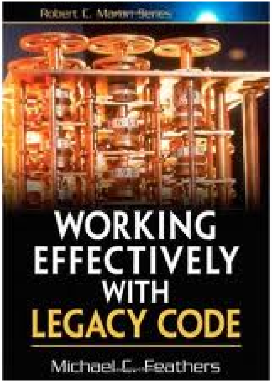
- Detects unintended changes
- Localizes defects
- Improves developer confidence
- **Decreases risk from change**

Do you write 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."

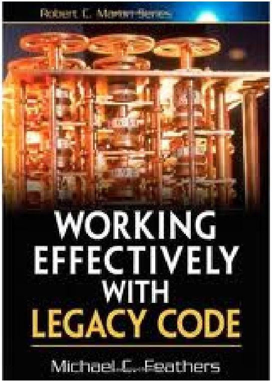
Michael Feathers
Working Effectively with Legacy Code



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



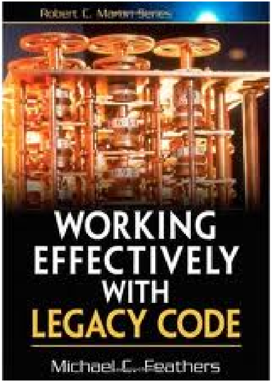
Lack of tests leads to fear of introducing subtle bugs and/or changing things inadvertently.

- 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

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
- "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

Just what is a test anyway?

Tests can exist in many forms

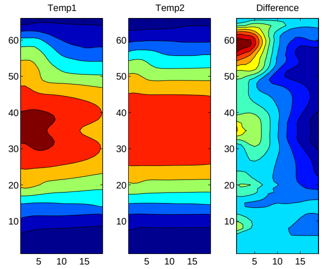
- Conditional termination:

```
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



Analogy with Scientific Method?

Reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: tests
Formulate hypothesis	$\longrightarrow$	Trial implementation
Perform experiment	$\longrightarrow$	Run tests
Refine hypothesis	$\longrightarrow$	Refine implementation

Properties of good tests

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

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.

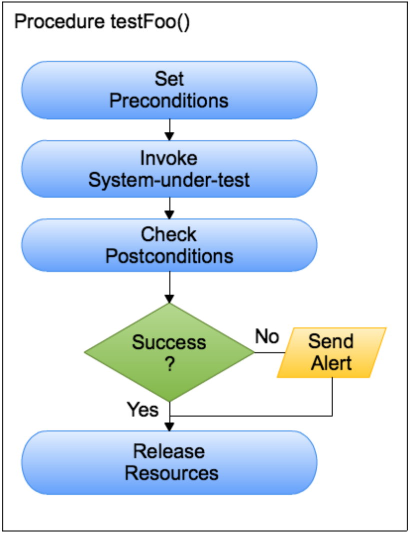
Properties of good tests

- 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

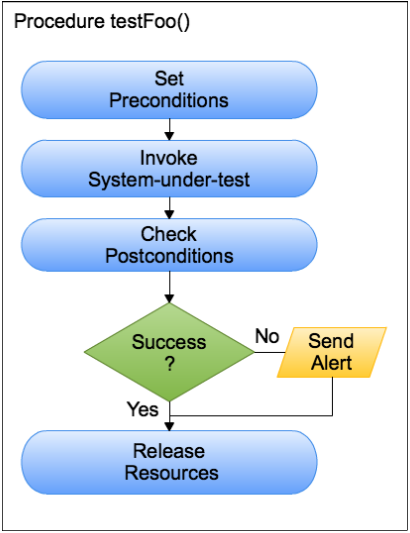
Properties of good tests

- 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

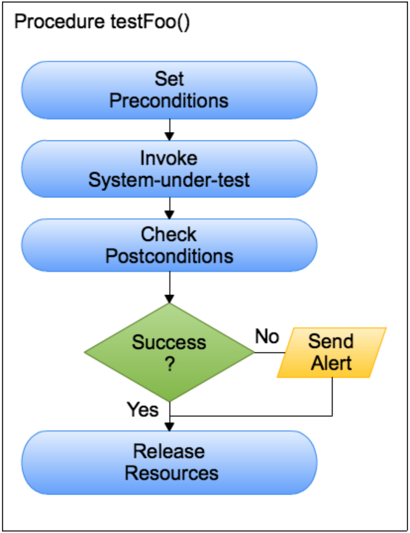


Anatomy of a Software Test Procedure



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

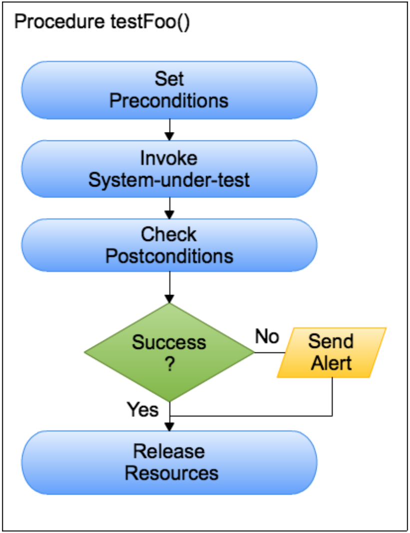
Anatomy of a Software Test Procedure



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

a = 2.; t = 3.

Anatomy of a Software Test Procedure

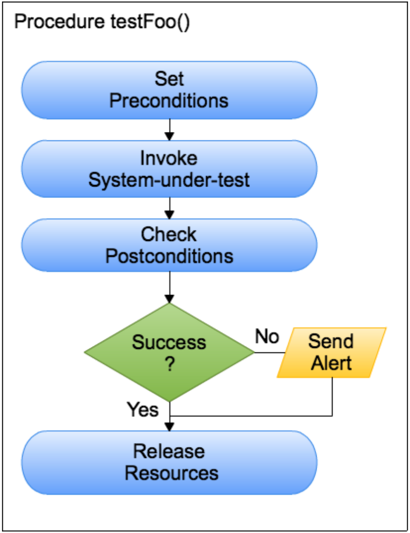


testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

Anatomy of a Software Test Procedure



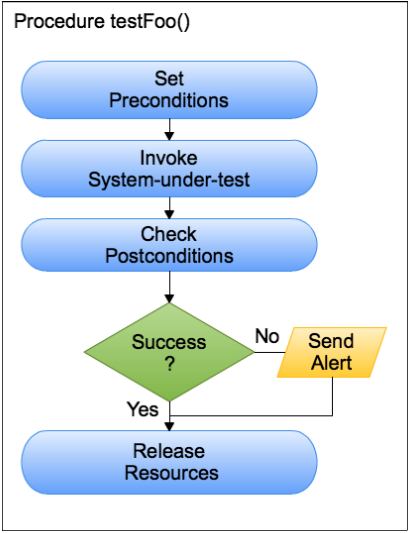
testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

call **assertEqual** (9., s)

Anatomy of a Software Test Procedure



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

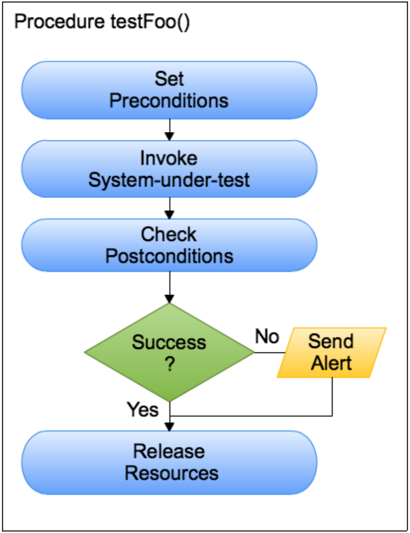
a = 2.; t = 3.

s = trajectory(a, t)

call **assertEqual** (9., s)

! no op

Anatomy of a Software Test Procedure



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

```
call assertEqual (9., trajectory (2.,3.))
```

Outline

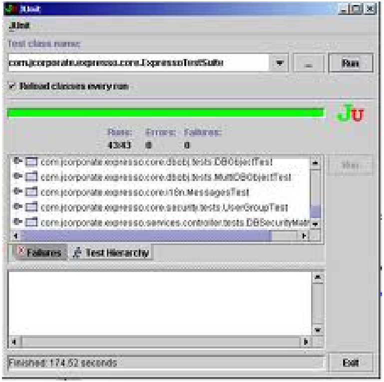
Testing Frameworks

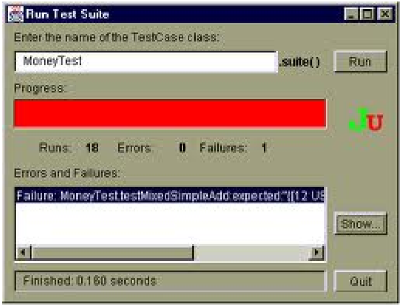
- Provide infrastructure to radically simplify:
  - Creating test routines (Test cases)
  - Running collections of tests (Test suites)
  - Summarizing results
- Key feature is collection of assert methods
  - Used to express expected results

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

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

GUI - JUnit in Eclipse





Outline

(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 *after* implementation

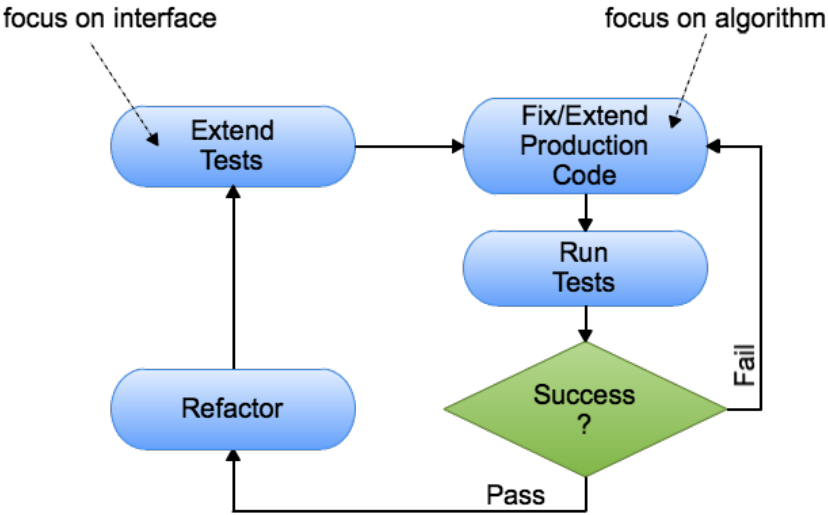
Consequences:

- 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



Benefits of TDD

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?**

Anecdotal Testimony

- Many professional SEs are initially skeptical
  - High percentage refuse to go back to the old way after only a few days of exposure.
- Some projects drop bug tracking as unnecessary
- Often difficult to sell to management
  - "What? More lines of code?"

Not a panacea

Not a panacea

- Requires training, practice, and discipline

Not a panacea

- Requires training, practice, and discipline
- Need strong tools (framework + refactoring)

Not a panacea

- Requires training, practice, and discipline
- Need strong tools (framework + refactoring)
- Does not invent new algorithms (e.g. FFT)
  - No such thing as magic

Not a panacea

- Requires training, practice, and discipline
- Need strong tools (framework + refactoring)
- Does not invent new algorithms (e.g. FFT)
  - No such thing as magic
- Maintaining tests difficult during a major re-engineering effort.

Not a panacea

- Requires training, practice, and discipline
- Need strong tools (framework + refactoring)
- Does not invent new algorithms (e.g. FFT)
    - No such thing as magic
- Maintaining tests difficult during a major re-engineering effort.
    - But isnt the alternative is even worse?!!

Outline

The Challenge of Technical Software

- Serious objections have been raised:

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?
- These concerns largely reveal

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?
- These concerns largely reveal
  - Lack of experience with *software* testing

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?
- These concerns largely reveal
  - Lack of experience with *software* testing
  - Confusion between roles of *verification* vs *validation*

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?
- These concerns largely reveal
  - Lack of experience with *software* testing
  - Confusion between roles of *verification* vs *validation*
  - Burden of legacy software (long procedures; complex interfaces)

The Challenge of Technical Software

- Serious objections have been raised:
  - Difficult to estimate error
    - Roundoff
    - Truncation
  - Stability/Nonlinearity
    - Problems that occur only after long integrations
  - Insufficient analytic cases
  - Test would just be re-expression of implementation
    - Irreducible complexity?
- These concerns largely reveal
  - Lack of experience with *software* testing
  - Confusion between roles of *verification* vs *validation*
  - Burden of legacy software (long procedures; complex interfaces)

Software Testing vs Science/Validation

Software tests should only check *implementation*.

- Only a subset tests will express external requirements (i.e. implementation independent)
- Other tests will reflect implementation choices
- Use "convenient" input values - **not** *realistic* values

Consider tests for an ODE integrator implemented with RK4

- A generic test may be for a constant flow field - any integrator should get an "exact" answer
- A RK4 specific test may provide an artificial "flow field" that returns the values 1.,2.,3.,4. on subsequent calls *independent* of the coordinates

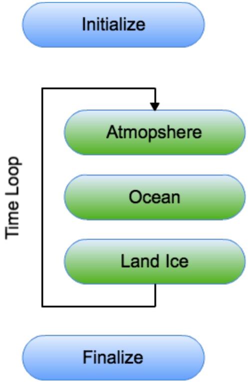
Test by Layers

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.

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 the tolerance is too *low*, then the test may fail for uninteresting reasons.

Numerical Tolerance

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

- If the tolerance is too *low*, then the test may fail for uninteresting reasons.
- If the tolerance is too *high*, then the test may have no teeth

Numerical Tolerance

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

- If the tolerance is too *low*, then the test may fail for uninteresting reasons.
- If the tolerance is too *high*, then the test may have no teeth

Unfortunately ...

- Error estimates are seldom available for complex algorithms

Numerical Tolerance

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

- If the tolerance is too *low*, then the test may fail for uninteresting reasons.
- If the tolerance is too *high*, then the test may have no teeth

Unfortunately ...

- Error estimates are seldom available for complex algorithms
- And of those, usually we just have an asymtotic form with unknown leading coefficient!

Numerical tolerance (cont'd)

Numerical tolerance (cont'd)

Observations

Numerical tolerance (cont'd)

Observations

1. machine epsilon is a good estimate for most short arithmetic expressions

Numerical tolerance (cont'd)

Observations

1. machine epsilon is a good estimate for most short arithmetic expressions
2. large errors arise in small expressions in fairly obvious places ($1/\Delta$)

Numerical tolerance (cont'd)

Observations

1. machine epsilon is a good estimate for most short arithmetic expressions
2. large errors arise in small expressions in fairly obvious places $(1/\Delta)$
3. larger errors are generally a result of composition of many operations

Numerical tolerance (cont'd)

Observations

1. machine epsilon is a good estimate for most short arithmetic expressions
2. large errors arise in small expressions in fairly obvious places $(1/\Delta)$
3. larger errors are generally a result of composition of many operations

Conclusion: If we write software as a composition of distinct small functions and subroutines, the errors can be reasonably bounded at each stage

TDD and long integration

- TDD does not directly relate to issues of stability
- If long integration gets incorrect results:

TDD and long integration

- TDD does not directly relate to issues of stability
- If long integration gets incorrect results:
  1. Software defect: missing test

TDD and long integration

- TDD does not directly relate to issues of stability
- If long integration gets incorrect results:
  1. Software defect: missing test
  2. Genuine science challenge
- TDD can reduce the frequency at which long integrations are needed/performed

TDD and Lack of Analytic Results

- Keep in mind: "How can you implement it if you cannot say what it should do?"
- Split into pieces - often each step has analytic solution
- Choose input values that are convenient

Consider a trivial case:

```
call assertEqual(3.14159265, areaOfCircle(1.))
call assertEqual(6.28..., areaOfCircle(2.))
```

What if instead the areaOfCircle() function accepted 2 arguments: "π" and r.

```
call assertEqual(1., areaOfCircle(1., 1.))
call assertEqual(4., areaOfCircle(1., 2.))
call assertEqual(2., areaOfCircle(2., 1.))
```

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**
- Long answer: Well, they shouldn't be ...

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use specific inputs - implementation handles generic case

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use specific inputs - implementation handles generic case
  - Each layer of algorithm is tested separately

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use specific inputs - implementation handles generic case
  - Each layer of algorithm is tested separately
  - Layers of the production code are *coupled* - huge complexity

TDD and irreducible complexity

- Are the tests as complex as the implementation?
- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use specific inputs - implementation handles generic case
  - Each layer of algorithm is tested separately
  - Layers of the production code are *coupled* - huge complexity
  - Tests are *decoupled* - low complexity

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 loging 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

TDD Best Practices

TDD Best Practices

- Small steps - each iteration $\ll$ 10 minutes

TDD Best Practices

- Small steps - each iteration $\ll$ 10 minutes
- Small, readable tests

TDD Best Practices

- Small steps - each iteration $\ll$ 10 minutes
- Small, readable tests
- Extremely fast execution - 1 ms/test or less

TDD Best Practices

- Small steps - each iteration $\ll$ 10 minutes
- Small, readable tests
- Extremely fast execution - 1 ms/test or less
- *Ruthless* refactoring

TDD Best Practices

- Small steps - each iteration $\ll$ 10 minutes
- Small, readable tests
- Extremely fast execution - 1 ms/test or less
- *Ruthless* refactoring
- Verify that each test initially **fails**

TDD and Performance

- Optimized algorithms may require many steps within a single procedure
- TDD emphasizes small simple procedures
- Such an approach may lead to slow execution
- Solution: Bootstrapping
  - Use initial solution as unit test for optimized solution
  - Maintain *both* implementations

Experience to date

TDD has been used heavily within several projects at NASA

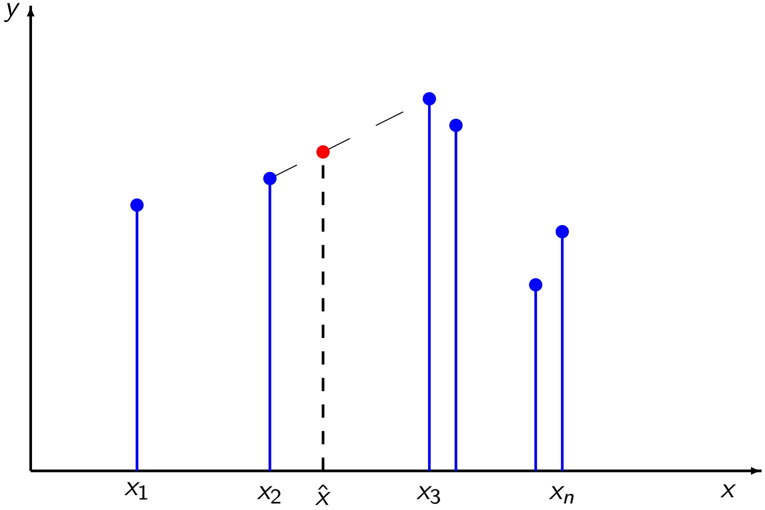
- Mostly for "infrastructure" portions - relatively little numerical alg.
- pFUnit
- DYNAMO - spectral MHD code on shperical shell
- GTRAJ - offline trajectory integration (C++)
- Snowfake - virtual snowfakes; Multi-lattice Snowfake

Observations:

- $\sim$ 1:1 ratio of test code to source code
- Works very well for *infrastructure*
- Learning curve
  - 1-2 days for technique
  - Weeks-months to wean old habits
  - Full benefit may require some sophistication

Outline

Linear Interpolation



Potential Tests

Potential Tests

- Bracketing: Find i such that $x_i <= \hat{x} < x_{i+1}$

Potential Tests

- Bracketing: Find i such that $x_i <= \hat{x} < x_{i+1}$
- Computing node weights:

$$
\begin{aligned}
w_a &= \frac{x_{i+1} - \hat{x}}{x_{i+1} - x_i} \\
w_b &= 1 - w_a
\end{aligned}
$$

Potential Tests

- Bracketing: Find i such that $x_i <= \hat{x} < x_{i+1}$
- Computing node weights:

$$
\begin{aligned}
w_a &= \frac{x_{i+1} - \hat{x}}{x_{i+1} - x_i} \\
w_b &= 1 - w_a
\end{aligned}
$$

- Compute weighted sum: $\hat{y} = w_a f(x_i) + w_b f(x_{i+1})$

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	nodes	x	return

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$
at node	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.0$	$i = 2$ (?)

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$
at node	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.0$	$i = 2$ (?)
at edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.0$	$i = 1$ (?)

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$
at node	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.0$	$i = 2$ (?)
at edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.0$	$i = 1$ (?)
other edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 3.0$	$i = 2$ (????)

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	**Preconditions** nodes	x	**Postcondition** return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$
at node	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.0$	$i = 2$ (?)
at edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.0$	$i = 1$ (?)
other edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 3.0$	$i = 2$ (????)
out-of-bounds	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	out-of-bounds error

Bracketing Tests

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	nodes	x	return
interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	$i = 1$
other interior	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.5$	$i = 2$
at node	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 2.0$	$i = 2$ (?)
at edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.0$	$i = 1$ (?)
other edge	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 3.0$	$i = 2$ (????)
out-of-bounds	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	out-of-bounds error
out-of-order	$\{x\} = \{1, 2, 3\}$	$\hat{x} = 1.5$	out-of-order error

Example: Bracketing Test 1

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 1.5$
- Postcondition: return 1

Example: Bracketing Test 1

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 1.5$
- Postcondition: return 1

```
subroutine testBracket1()
   nodes = [1.,2.,3.]
   index = getBracket(nodes, 1.5)
   call assertEqual(1, index)
end subroutine
```

Example: Bracketing Test 1

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 1.5$
- Postcondition: return 1

```
subroutine testBracket1()
   call assertEqual(1, getBracket([1.,2.,3.], 1.5))
end subroutine
```

Example: Bracketing Test 1

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 1.5$
- Postcondition: return 1

```
subroutine testBracket1()
   call assertEqual(1, getBracket([1.,2.,3.], 1.5))
end subroutine
```

```
function getBracket(nodes, x) result(index)
   index = 1
end function
```

Example: Bracketing Test 2

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 2.5$
- Postcondition: return 2

```
subroutine testBracket2()
   nodes = [1.,2.,3.]
   index = getBracket(nodes, 2.5)
   call assertEqual(2, index)
end subroutine
```

Example: Bracketing Test 2

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 2.5$
- Postcondition: return 2

```
subroutine testBracket2()
   nodes = [1.,2.,3.]
   index = getBracket(nodes, 2.5)
   call assertEqual(2, index)
end subroutine
```

```
function getBracket(nodes, x) result(index)
   if (x > nodes(2)) then
      index = 2
   else
      index = 1
   end if
end function
```

Example: Bracketing Test 2

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 2.5$
- Postcondition: return 2

```
subroutine testBracket2()
   nodes = [1.,2.,3.]
   index = getBracket(nodes, 2.5)
   call assertEqual(2, index)
end subroutine
```

```
function getBracket(nodes, x) result(index)
   if (x > nodes(2)) then
      index = 2
   else
      index = 1
   end if
end function
```

Generalize ...

Example: Bracketing Test 2

- Preconditions: $\{x\} = \{1, 2, 3\}, \hat{x} = 2.5$
- Postcondition: return 2

```
subroutine testBracket2()
  nodes = [1.,2.,3.]
  index = getBracket(nodes, 2.5)
  call assertEqual(2, index)
end subroutine
```

```
function getBracket(nodes, x) result(index)

  do i = 1, size(nodes)     1
    if (nodes(i+1) > x) index = i
  end do

end function
```

Tests for Computing Weights

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	interval	x	weights

Tests for Computing Weights

```
index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$

Tests for Computing Weights

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$
upper bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [0.0, 1.0]$

Tests for Computing Weights

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$
upper bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [0.0, 1.0]$
interior	$[1., 2.]$	$\hat{x} = 1.5$	$w = [0.5, 0.5]$

Tests for Computing Weights

```
    index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$
upper bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [0.0, 1.0]$
interior	$[1., 2.]$	$\hat{x} = 1.5$	$w = [0.5, 0.5]$
big interval slope	$[1., 3.]$	$\hat{x} = 1.5$	$w = [0.75, 0.25]$

Tests for Computing Weights

```
index = bracket(nodes, x)
```

Case	**Preconditions**		**Postcondition**
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$
upper bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [0.0, 1.0]$
interior	$[1., 2.]$	$\hat{x} = 1.5$	$w = [0.5, 0.5]$
big interval slope	$[1., 3.]$	$\hat{x} = 1.5$	$w = [0.75, 0.25]$
degenerate	$[1., 1.]$	$\hat{x} = 1.0$	degenerate error

Tests for Computing Weights

```
    index = bracket(nodes, x)
```

Case	Preconditions		Postcondition
	interval	x	weights
lower bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [1.0, 0.0]$
upper bound	$[1., 2.]$	$\hat{x} = 1.0$	$w = [0.0, 1.0]$
interior	$[1., 2.]$	$\hat{x} = 1.5$	$w = [0.5, 0.5]$
big interval slope	$[1., 3.]$	$\hat{x} = 1.5$	$w = [0.75, 0.25]$
degenerate	$[1., 1.]$	$\hat{x} = 1.0$	degenerate error
out-of-bounds	$[1., 2.]$	$\hat{x} = 0.5$	out-of-bounds error

Example: Weights Test 1

- Precondition: $[a, b] = [1., 2.], \hat{x} = 1.0$
- Postcondition: $w = \{1.0, 0.0\}$

```
subroutine testWeight1()
   real :: interval(2), weights(2)
   real :: x
   interval = [1.,2.]
   weights = computeWeights(interval, 1.0)
   call assertEqual([1.0,0.0], weights)
end subroutine testWeight1
```

```
real function computeWeights(interval, x) result(weights)
   real, intent(in) :: interval(2)
   real, intent(in) :: x
   weights = [1.0,0.0]
end function
```

Example: Tying it together

- Precondition:
  - $\{(x,y)_i\} = \{(1,1),(2,1),(4,1)\}$
  - $\hat{x} = 3$
- Postcondition: $\hat{y} = 1$.

```
subroutine testInterpolateConstantY()
   real :: nodes(2,3)
   nodes = reshape([[1,1],[2,1],[4,1]], shape=[2,3])
   call assertEqual(1.0, interpolate(nodes, 3.0))
end subroutine testInterpolate1
```

```
function interpolate(nodes, x)
   real, intent(in) :: nodes(:,:)
   y = 1
end function interpolate
```

Example: Tying it together

- Precondition:
  - $\{(x, y)_i\} = \{(1, 1), (2, 3), (4, 1)\}$
  - $\hat{x} = 3$
- Postcondition: $\hat{y} = 2$.

```
subroutine testInterpolate1()
   real :: nodes(2,3)
   nodes = reshape([[1,1],[2,3],[4,1]], shape=[2,3])
   call assertEqual(1.0, interpolate(nodes, 3.0))
end subroutine testInterpolate1
```

```
function interpolate(nodes, x) result(y)
   integer :: i
   real :: weights(2), xAtEndPoints(2), yAtEndpoints(2)

   i = getBracket(nodes(1,:), x)

   xAtEndPoints = nodes(1,i)   ! used derived type?
   yAtEndpoints = nodes(2,i)
   weights = computeWeights(nodes(1,[i,i+1]), x)

   y = sum(weights * yAtEndpoints)
end function interpolate
```

Outline

pFUnit - Fortran Unit testing framework

- Tests written in Fortran
- Supports testing of parallel (MPI) algorithms
- Support for multi-dimensional array assertions
- Written in standard F95 (plus a tiny bit of F2003)
- Developed using TDD

Tutorial in the afternoon sessioon

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
- Mller and Padberg,"About the Return on Investment of Test-Driven Development," http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf
- *Refactoring: Improving the Design of Existing Code* - Martin Fowler
- JUnit http://junit.sourceforge.net/
- These slides https://modelingguru.nasa.gov/docs/DOC-2222