



pFUnit 3.0 Tutorial Advanced

Tom Clune

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

April 10, 2014

View metadata, citation and similar papers at core.ac.uk

Outline



- 1 Introduction
 - Overview
- 2 API - Advanced
- 3 Test-driven development

Outline



- 1 Introduction
 - Overview
- 2 API - Advanced
- 3 Test-driven development

Class Overview



Primary Goals:

- Learn how to use pFUnit 3.0 to create and run unit-tests
- Learn how to apply test-driven development methodology

Prerequisites:

- Access to Fortran compiler supported by pFUnit 3.0
- Familiarity with F95 syntax
- Familiarity with MPI¹

Beneficial skills:

- Exposure to F2003 syntax - esp. OO features
- Exposure to OO programming in general

¹MPI-specific sections can be skipped without impact to other topics.

Syllabus



- **Thursday PM - Introduction to pFUnit**
 - ▶ Overview of pFUnit and unit testing
 - ▶ Build and install pFUnit
 - ▶ Simple use cases and exercises
 - ▶ Detailed look at framework API
- **Friday AM - Advanced topics (including TDD)**
 - ▶ User-defined test subclasses
 - ▶ Parameterized tests
 - ▶ Introduction to TDD
 - ▶ Advanced exercises using TDD
- **Friday PM - Bring-your-own-code**
 - ▶ Incorporate pFUnit within the build process of your projects
 - ▶ Apply pFUnit/TDD in your own code
 - ▶ Supplementray exercises will be available

Materials



- ① You will need access to one of the following Fortran compilers to do the hands-on portions
 - ▶ gfortran 4.9.0 (possibly available from cloud)
 - ▶ Intel 13.1, 14.0.2 (available on jellystone)
 - ▶ NAG 5.3.2
- ② Last resort - use AWS
 - ▶ ssh keys are at <ftp://tartaja.com>
 - ▶ user name: `pfunit@tartaja.com` passwd: `iuse.PYTHON.1969`
 - ▶ login: `ssh -i user1 user1@54.209.194.237`
- ③ You will need a copy of the exercises in your work environment
 - ▶ Browser: <https://modelingguru.nasa.gov/docs/DOC-2529>
 - ▶ Jellystone:
`/picnic/u/home/cacruz/pFUnit.tutorial/Exercises.tar`
- ④ These slides can be downloaded at
<https://modelingguru.nasa.gov/docs/DOC-2528>

Outline



- 1 Introduction
- 2 API - Advanced
 - API: pFUnit test Hierarchy
 - API: Misc
 - Parser: Advanced
- 3 Test-driven development

Peeking under the hood - what is inside pFUnit?



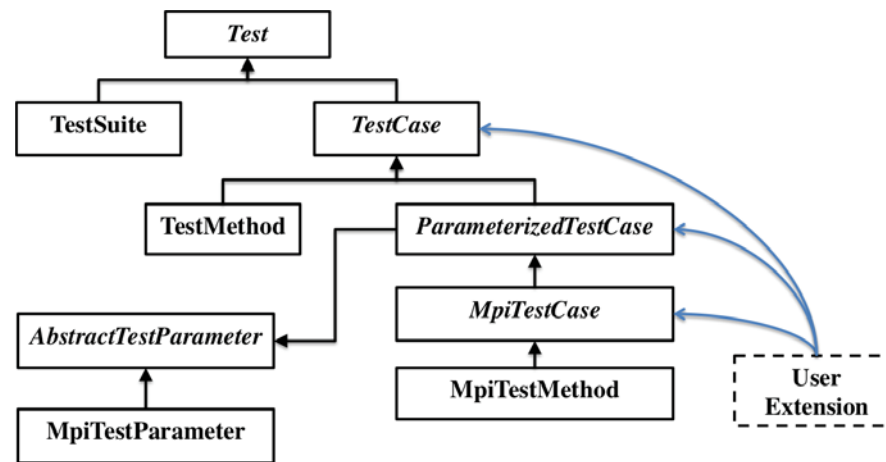
www.shescribes.com

Outline



- 1 Introduction
- 2 API - Advanced
 - API: pFUnit test Hierarchy
 - API: Misc
 - Parser: Advanced
- 3 Test-driven development

Hierarchy of Test Classes



Test



Role: Abstract base class for all test objects.

Implementation: Framework provides various subclasses for common/generic cases. Users can define custom subclasses for specific purposes. Provided subclasses include:

- *TestCase*
- `TestMethod`
- *MpiTestCase*
- `MpiTestMethod`
- `TestSuite`

TestSuite



Role: Aggregates collection of tests into single entity.

Implementation: TestSuite objects are simultaneously Test objects *and* collections of tests. Run() method applies run() to each contained test.

TestCase class



Role: *Abstract* Test subclass that provides some services that are common to most Test subclasses.

Implementation:

TestMethod class



Role: Simple concrete Test subclass that supports the common case where test procedure receives no arguments.

Implementation: Constructor stores a procedure pointer to vanilla Fortran subroutine with no arguments. A restricted form of test fixture is permitted by specifying `setUp()` and `tearDown()` methods that also have no arguments. (I.e. fixture is not encapsulated.)

TestMethod API



Constructor:

```
function TestMethod(name, method[, setUp, tearDown])  
  character(len=*), intent(in) :: name  
  procedure(empty) :: method  
  procedure(empty) :: setUp  
  procedure(empty) :: tearDown
```

Methods:

ParameterizedTestCase class



Role: Allows a single test procedure to be executed multiple times with different input values.

Implementation: *ParameterizedTestCase* objects contain an *AbstractTestParameter* object that encapsulates input. Subclasses of *ParameterizedTestCase* must generally also subclass *AbstractTestParameter*.

MpiTestCase class



Role: (*Abstract*) Extends *ParameterizedTestCase* with support for MPI.

Implementation: *MpiTestCase* modifies the `runBare()` launch mechanism to create an appropriately sized MPI group and corresponding subcommunicator. Processes within that group then call the user's test procedure, while any remaining processes wait at a barrier.

MPI based tests *must not* use `MPI_COMM_WORLD`, and must instead obtain MPI context from the passed test object.

The following convenient type-bound procedures are provided:

```
getProcessRank() ! returns rank within group  
getNumProcesses() ! returns size of group  
getMpiCommunicator() ! returns the bare MPI com.
```

MpiTestMethod class



Role: Simple concrete Test subclass that supports common MPI cases that just need basic MPI context.

Implementation: Analogous to the vanilla TestMethod, except that user test procedures are now passed an object which must be queried for any MPI context that the test needs.

MpiTestMethod API



Constructor:

```
function MpiTestMethod(name, method, numProcesses, [, setUp]
  character(len=*) , intent(in) :: name
  procedure(empty) :: method
  integer :: numProcesses ! requested
  procedure(empty) :: setUp
  procedure(empty) :: tearDown
```

Outline



- 1 Introduction
- 2 API - Advanced
 - API: pFUnit test Hierarchy
 - API: Misc
 - Parser: Advanced
- 3 Test-driven development

TestResult class



Role: “Scorecard” – accumulates information about tests as they run.

Implementation: Each `run()` method for `Test` objects has a mandatory `TestResult` argument. The *Visitor* pattern is used to allow the `TestResult` object to manage and monitor the test as it progresses.

Note: *Visitor* is a somewhat advanced pattern and uses OO capabilities in a nontrivial manner. Users should not need to be aware of this, but developers of framework extensions likely will.

Abstract BaseTestRunner class



Role: Runs a test (usually a TestSuite).

Implementation: Run() method constructs and configures a TestResult object, then runs the passed Test object.

TestRunner class



Role: Default Runner for pFUnit.

RobustRunner class



Role: Runner subclass that executes tests within a separate process.

Implementation: Collaborates with SubsetRunner. RobustRunner restarts SubsetRunner if it detects a hang or a crash. Currently a bit unreliable.
(Irony)

Outline



- 1 Introduction
- 2 API - Advanced
 - API: pFUnit test Hierarchy
 - API: Misc
 - Parser: Advanced
- 3 Test-driven development

Annotations: @testCase



```
@testCase  
@testCase(<options>)
```

- Indicates next line defines a new derived type which extends TestCase.
- All test procedures in file must accept a single argument of that extended type.
- Accepts the following options:
 - ▶ `constructor=<name>` Specifies the name of the function to construct corresponding test object. Default is a constructor with same name as derived type²
 - ▶ `npes=[<list-of-integers>]` Indicates that extension is a subclass of `MpiTestCase`, and provides a default set of values for NPES for all test procedures in the file. Individual tests can override.
 - ▶ `esParameters={expr}` Indicates that extension is a subclass of `ParameterizedTestCase`, and provides a default set of parameters for all tests in the file. Can be overridden by each test.
 - ▶ `cases=[<list-of-integers>]` Alternative mechanism for specifying default test parameters where a single integer is passed to the test constructor.

²This E9002 feature is somewhat unreliable, even prior to 14.0.2

Annotations: @testParameter



Encapsulated test fixture



```
. module SomeTests_mod
.   use pFUnit_mod
.   implicit none
.   @testCase
.   type, extends( TestCase ) :: MyTestCase
.     real, allocatable :: xInitial(:)
.   contains
.     procedure :: setUp
.     procedure :: tearDown
.   end type MyTestCase
. contains
.
.   subroutine setUp(this)
.     class (MyTestCase), intent(inout) :: this
.     xInitial = [1.,3.,5.,3.,1.]
.   end subroutine setUp
.
.   subroutine tearDown(this)
.     class (MyTestCase), intent(inout) :: this
.     deallocate(this%xInitial)
.   end subroutine tearDown
.
```

...

Encapsulated test fixture (cont'd)



```
...
:
: @test
: subroutine anotherTest(this)
:   class (MyTestCase), intent(inout) :: this
:
:   real, allocatable :: x(:)
:
:   x = oneStep(this%xInitial)
:   @assertEqual(...)
:
: end subroutine anotherTest
:
: end module MyTests_mod
```

Encapsulated test fixture (cont'd)



What you need to know:

- Declare derived type that EXTEND's `TestCase`
- Annotate TestCase extension with `@testCase`
- Declare TYPE-BOUND procedures: `setUp` and `tearDown`
- Annotate test procedure in usual way with `@test`
- Declare single test procedure argument as

```
class (<your type>), intent(inout) :: <dummy>
```

MPI test fixture



```
module SomeMpiTests_mod
:   use pFUnit_mod
:   implicit none
:
:   @testCase(npes=[1,3,5])
:   type, extends(MpiTestCase) :: MyTestCase
:       integer :: rank, npes
:       integer :: peEast, peWest
:   contains
:       procedure :: setUp
:       procedure :: tearDown
:   end type MyTestCase
:
: contains
:
:   subroutine setUp(this)
:       class (MyTestCase), intent(inout) :: this
:       integer :: rank, npes
:       this%rank = this%getProcessRank()
:       this%npes = this%getNumProcesses()
:       this%peWest = mod(this%rank + this%npes - 1, this%npes)
:       this%peEast = mod(this%rank + 1, this%npes)
:   end subroutine setUp
:
: ...
```

MPI test fixture (cont'd)



```
...
:
: @test
: subroutine anotherTest(this)
:   class (MyTestCase), intent(inout) :: this
:
:   integer :: comm
:   real :: x(0:2)
:
:   comm = this%getMpiCommunicator()
:
:   call someMpiProcedure(comm,x)
:
:   @mpiAssertEqual(this%peWest, x(0))
:   @mpiAssertEqual(this%rank, x(1))
:   @mpiAssertEqual(this%peEast, x(2))
:
:   end subroutine anotherTest
:
: end module MyTests_mod
```


MPI test fixture (cont'd)



What you need to know:

- Declare derived type that EXTEND's `MpiTestCase`
- Annotate TestCase extension with `@testCase`
 - ▶ Optionally specify default npes list: `(npes=[...])`
- Declare TYPE-BOUND procedures: `setUp` and `tearDown`
- Annotate test procedure in usual way with `@test`
- Declare single test procedure argument as

```
class (<your type>), intent(inout) :: <dummy>
```

- Use `@mpiAssert*` to synchronize returns

Parameterized tests



Suppose you want to test an interface using variant input data:

Parameterized tests



Suppose you want to test an interface using variant input data:
E.g. sorting a list ...

```
list = sort([1,2,3,4])  
list = sort([4,3,2,1])  
list = sort([1,4,2,3])  
list = sort([1,2,3,1])
```

Parameterized tests



Suppose you want to test an interface using variant input data:
E.g. sorting a list ...

```
list = sort([1,2,3,4])  
list = sort([4,3,2,1])  
list = sort([1,4,2,3])  
list = sort([1,2,3,1])
```

or varying boundary conditions...

```
call solve(x, BC='dirichlet')  
call solve(x, BC='neumann')
```

Parameterized tests (cont'd)



One simple strategy is to just duplicate tests:

```
@test
subroutine test1()
  @assertEquals([1,2,3,4], sort([1,2,3,4]))
end subroutine test1

@test
subroutine test2()
  @assertEquals([1,2,3,4], sort([4,3,2,1]))
end subroutine test2
...
```

Parameterized tests (cont'd)



One simple strategy is to just duplicate tests:

```
@test
subroutine test1()
    @assertEquals([1,2,3,4], sort([1,2,3,4]))
end subroutine test1

@test
subroutine test2()
    @assertEquals([1,2,3,4], sort([4,3,2,1]))
end subroutine test2
...
```

This can be quite tedious if there are many cases and/or the tests are more complex.

Parameterized tests (cont'd)



Another approach is to loop within a test

```
@test
subroutine test()
  real, allocatable :: x(:)

  call checkDeriv(x, x**0)
  call checkDeriv(x**2, 2*x)
  call checkDeriv(x**3, 3*x**2)
  ...

contains

  subroutine checkDeriv(fx, dfx)
    real, intent(in) :: fx
    real, intent(in) :: dfx
    @assertEqual(dfx, deriv(fx))
  end subroutine checkDeriv
end subroutine test1
```

Parameterized tests (cont'd)



Another approach is to loop within a test

```
@test
subroutine test()
  real, allocatable :: x(:)

  call checkDeriv(x, x**0)
  call checkDeriv(x**2, 2*x)
  call checkDeriv(x**3, 3*x**2)
  ...

contains

  subroutine checkDeriv(fx, dfx)
    real, intent(in) :: fx
    real, intent(in) :: dfx
    @assertEqual(dfx, deriv(fx))
  end subroutine checkDeriv
end subroutine test1
```

Here we lose information about which case(s) failed.

Parameterized tests (cont'd)



pFUnit provides custom support for parameterized tests:

- Exercise tests across list of user-defined parameters
- User EXTEND's two classes:
 - ▶ `ParameterizedTestCase` (analog of `TestCase`)
 - ▶ `AbstractTestParameter`
- Annotation argument: `testParameters={<expr>}`
 - ▶ Specifies default parameter list for `@testCase`
 - ▶ Override with argument to `@test`
- Failures indicate parameter caused failing assert.
 - ▶ Provided through type-bound interface `toString()` on `AbstractTestParameter`

Example: Parameterized test



...

```
7  @testParameter
8  type, extends(AbstractTestParameter) :: StringTestParameter
9      character(:), allocatable :: string
10     character(:), allocatable :: lowerCase
11     character(:), allocatable :: upperCase
12     contains
13     procedure :: toString
14 end type StringTestParameter
```

...

```
66 function toString(this) result(string)
67     class (StringTestParameter), intent(in) :: this
68     character(:), allocatable :: string
69
70     string = '{' // this%string // ',' // this%lowerCase //
71             ',' // this%upperCase // '}'
72 end function toString
```

Example: Parameterized test (cont'd)



```
...
16  @testCase(testParameters = {getParams()}, constructor=
    newTest_StringUtilities)
17  type, extends(ParameterizedTestCase) :: Test_StringUtilities
18  character(:), allocatable :: string
19  character(:), allocatable :: lowerCase
20  character(:), allocatable :: upperCase
21  end type Test_StringUtilities

24
25  function getParams() result(params)
26  type (StringTestParameter), allocatable :: params(:)
27
28  params = [ &
29  StringTestParameter('a', 'a', 'A'), &
30  StringTestParameter('b', 'b', 'B'), &
31  StringTestParameter('A', 'a', 'A'), &
32  StringTestParameter('1', '1', '1'), &
33  StringTestParameter('+', '+', '+'), &
34  StringTestParameter('a1B2c3D4', 'a1b2c3d4', 'A1B2C3D4')
    &
35  ]
36
37  end function getParams
```

Example: Parameterized test (cont'd)



```
...
48  @test
49  subroutine test_toLowerCase(this)
50      class (Test_StringUtilities), intent(inout) :: this
51
52      @assertEqual(this%lowerCase, toLowerCase(this%string))
53
54  end subroutine test_toLowerCase
55
56
57  @test
58  subroutine test_toUpperCase(this)
59      class (Test_StringUtilities), intent(inout) :: this
60
61      @assertEqual(this%upperCase, toUpperCase(this%string))
62
63  end subroutine test_toUpperCase
```

Example: Parameterized test (cont'd)



To specify a variant list of parameters:

```
.. @test (testParameters={getOtherParams()})  
.. subroutine test_toUpperCase(this)  
..   class (Test_StringUtilities), intent(inout) :: this  
..  
..   @assertEqual(this%upperCase, toUpperCase(this%string))  
..  
.. end subroutine test_toUpperCase
```

Combining MPI and Parameterized Test



Combining MPI and Parameterized Test



Good news:

`MpiTestCase` is a **subclass** of `ParameterizedTest`

Combining MPI and Parameterized Test



Good news:

`MpiTestCase` is a **subclass** of `ParameterizedTest`

- Extend `MpiTestCase`

Combining MPI and Parameterized Test



Good news:

`MpiTestCase` is a **subclass** of `ParameterizedTest`

- Extend `MpiTestCase`
- Extend `MpiTestParameter` (invisible with simple MPI)

Combining MPI and Parameterized Test



Good news:

`MpiTestCase` is a **subclass** of `ParameterizedTest`

- Extend `MpiTestCase`
- Extend `MpiTestParameter` (invisible with simple MPI)
- Framework augments `toString()` to ensure that rank/npes is always included in failure messages

Outline



- 1 Introduction
- 2 API - Advanced
- 3 Test-driven development

TDD





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

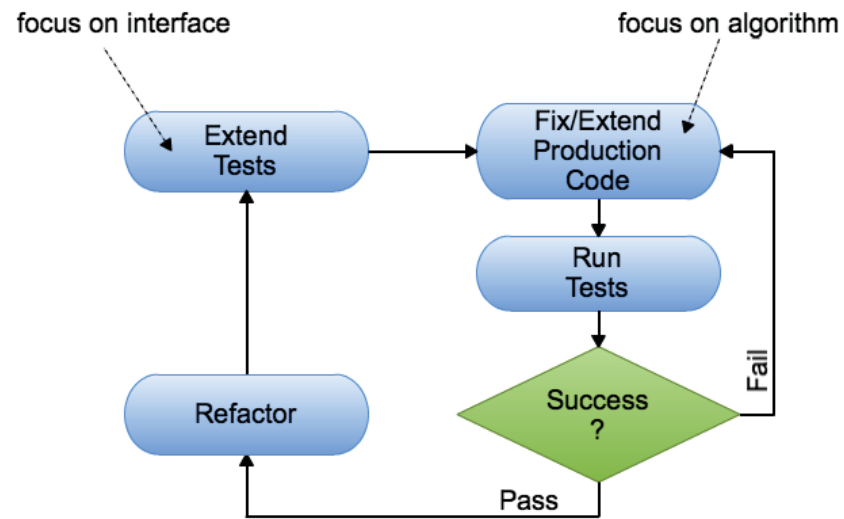
Consequences:

- 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



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?!!

Experience to date



TDD has been used heavily within several projects at NASA

- Mostly for “infrastructure” portions - relatively little numerical
- pFUnit itself
- Snowfake - virtual snowflakes; Multi-lattice Snowfake
- DYNAMO - spectral MHD code on spherical shell
- GTRAJ - offline trajectory integration (C++)
- SpF - OO parallel spectral framework

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

TDD - Talking Points

- How large of a step at each cycle?



TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors
- Commit/backup *frequently*

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors
- Commit/backup *frequently*
- Use synthetic data to make results *obvious*

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors
- Commit/backup *frequently*
- Use synthetic data to make results *obvious*
- Private vs testable

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors
- Commit/backup *frequently*
- Use synthetic data to make results *obvious*
- Private vs testable
 - ▶ One module has everything PUBLIC
 - ▶ 2nd module is default private - just export the things you want PUBLIC
 - ▶ Tests use first module; application uses 2nd.

TDD - Talking Points



- How large of a step at each cycle?
 - ▶ Gauge by time
 - ▶ If steps are going quickly try larger changes
 - ▶ If iteration > 10 min, start iteration over (repository is your friend)
- Triangulation
 - ▶ Start with simple tests
 - ▶ Add tests that probe weaknesses in existing implementation
 - ▶ Stop when it is apparent that new tests will all pass
- Don't test constructors and accessors
- Commit/backup *frequently*
- Use synthetic data to make results *obvious*
- Private vs testable
 - ▶ One module has everything PUBLIC
 - ▶ 2nd module is default private - just export the things you want PUBLIC
 - ▶ Tests use first module; application uses 2nd.
- *Think* when writing tests; *autopilot* when writing implementation

TDD - process reminder



- 1 Extend test (new test procedure, new assert, etc)
- 2 *Verify test fails* **Red Light**
- 3 Alter implementation to pass test
- 4 Refactor to eliminate redundancy **Green Light**
- 5 Repeat

TDD Demonstration: Factorial



Instructions:

Use TDD to implement factorial function

To make it interesting, we'll add tests to guard against illegal inputs and overflow.



- Change into the directory `./Exercises/TDD_Warmup`
- Set PFUNIT for a serial build
- `% make tests` (ensure that make is working for you)

TDD Demonstration: Dynamical System



Instructions:

We are going to build a set of classes that will integrate a simple dynamical system:

- State of system is specified by a scalar, t , and 2 vectors: x and v
- Denote timestep with h
- Force (F) on system is any function of x, v, t
- Initial integration will be via forward Euler: $Y_{n+1} = Y_n + hF(Y_n, t)$
- Then we will “upgrade” to RK4

Possible unit tests for Dynamical System



Forward Euler integration

- $F(t) = 0, v(t = 0) = 0$ leaves $x_{n+1} = x_0$
- $F(t) = 0, v(t = 0) = v_0$ has $x_{n+1} = nhv_0$
- $F(t) = 0, v(t = 0) = v_0$ has $v_{n+1} = v_n$
- $F(t) = F(t = 0) = a, v(t = 0) = x(t = 0) = 0$ has $v_{n+1} = v_n + ha$
- $v_{n+1} = v_n + hF(t_n)$
- $x_{n+1} = x_n + hv_n$
- If $h = 0, x_n = x_0$ and $v_n = v_0$ for any F
- Vary number of dimensions



- Change into the directory `./Exercises/TDD_DynamicalSystem`
- Set PFUNIT for a serial build
- `% make tests` (ensure that make is working for you)

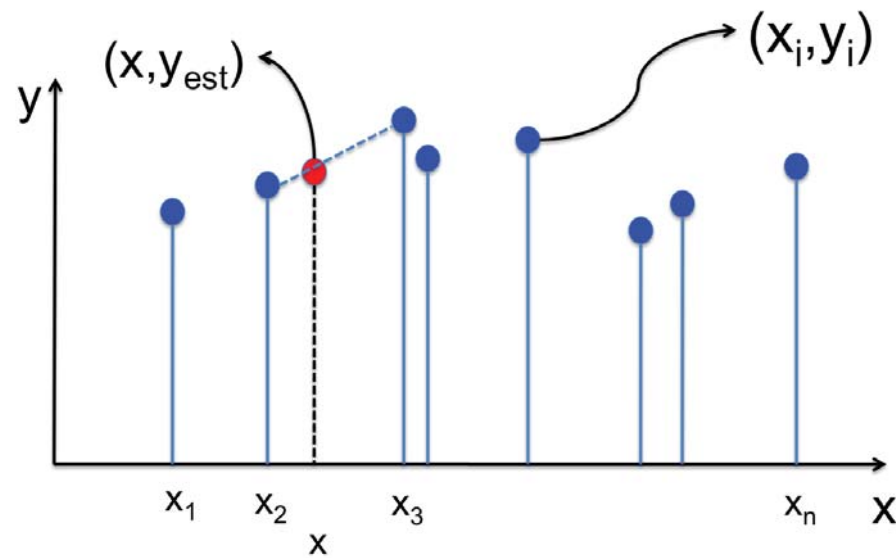
Runge-Kutta (RK4)



$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$
$$t_{n+1} = t_n + h$$

$$k_1 = f(t_n, y_n)$$
$$k_2 = f\left(t_n + \frac{1}{2}h, y_n + \frac{h}{2}k_1\right)$$
$$k_3 = f\left(t_n + \frac{1}{2}h, y_n + \frac{h}{2}k_2\right)$$
$$k_4 = f(t_n + h, y_n + hk_3)$$

Demo: Build a Linear 1D Interpolator



Interpolation ...



What are some potential tests?

Interpolation ...



What are some potential tests?

- Bracket: Find i such that $x_i \leq x < x_{i+1}$

Interpolation ...



What are some potential tests?

- Bracket: Find i such that $x_i \leq x < x_{i+1}$
- Computing weights:

$$w_a = \frac{x_{i+1} - x}{x_{i+1} - x_i}$$
$$w_b = 1 - w_a$$

Interpolation ...



What are some potential tests?

- Bracket: Find i such that $x_i \leq x < x_{i+1}$
- Computing weights:

$$w_a = \frac{x_{i+1} - x}{x_{i+1} - x_i}$$
$$w_b = 1 - w_a$$

- Combining weighted sum: $y = w_a y_i + w_b y_{i+1}$

Tests for finding enclosing bracket



| $\{x_1, x_2, x_3\}$ | x | Expect | Comment |
|---------------------|-----|------------|-------------------------|
| {1.,2.,3.} | 1.5 | $i = 1$ | vanilla |
| {1.,2.,3.} | 2.5 | $i = 2$ | vary x |
| {1.,2.,4.} | 3.0 | $i = 2$ | irregular spacing |
| {1.,2.,4.,5.} | 2.5 | $i = 2$ | vary # of nodes |
| {1.,2.,3.} | 2.0 | $i = 2$ | edge case |
| {1.,2.,3.} | 1.0 | $i = 1?$ | edge case |
| {1.,2.,3.} | 3.0 | $i = 2?$ | edge case |
| {1.,2.,3.} | 0.5 | exception? | out-of-bounds |
| {3.,2.,1.} | 1.5 | exception? | support inverted order? |

Tests for compute weights



| x_i | x_{i+1} | x | expected | Comment |
|-------|-----------|-----|--------------|----------------|
| 1. | 2. | 1.0 | $w_a = 1.0$ | left end |
| 1. | 2. | 2.0 | $w_a = 0.0$ | right end |
| 1. | 2. | 1.5 | $w_a = 0.5$ | middle |
| 1. | 3. | 1.5 | $w_a = 0.75$ | vary interval |
| 1. | 2. | 0.0 | $w_a = ?$ | out-of-bounds |
| 1. | 1. | 1.0 | ? | duplicate node |

Tests for combine weights



| w_a | y_a | y_b | expected | Comment |
|-------|-------|-------|-----------|-----------|
| 1. | 1. | 2. | $y = 1.0$ | left end |
| 0. | 1. | 2. | $y = 2.0$ | right end |
| 0.5 | 1. | 2. | $y = 1.5$ | middle |
| 0.5 | 3. | 2. | $y = 2.5$ | vary data |



Live Demo: Cross Fingers

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/>

Acknowledgements



- This work has been supported by NASA's High End Computing (HEC) program and Modeling, Analysis, and Prediction Program.
- Many thanks to team members Carlos Cruz and Mike Rilee for helping with implementation, regression testing and documentation.
- Special thanks to members of the user community that have made contributions.
 - ▶ Sean Patrick Santos
 - ▶ Matthew Hambley
 - ▶ Evan Lezar