Parallel Fortran Unit Testing Framework

Installation, Usage, and API

Tom Clune

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

April 31, 2012

Outline

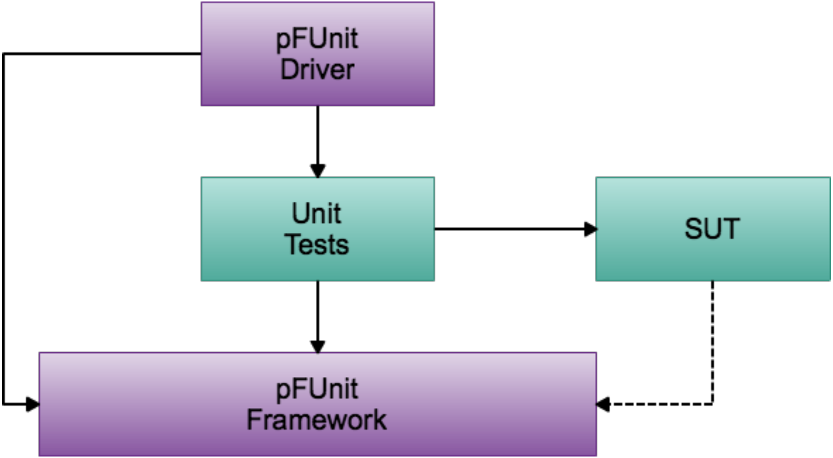
pFUnit fact sheet

- History
  - original "unfunded" development - 2005
  - NASA Open Source Agreement (NOSA) - 2006
  - HEC funding for documentation/tutorial - 2010
  - SBIR grant to Tech-X to integrate within Eclipse/Photran
  - Primary interfaces have been stable for years (too few users?)
- Targeted at technical software written in Fortran
  - Developed using TDD in (almost) standard Fortan
  - Supports testing of parallel software based on MPI
  - Extensive support for multidimensional FP arrays
  - Parameterized tests
- "F2kUnit" - next release, rewrite from scratch in F2003 and OO
  - Very extensible
  - Core cababilities are complete, but need to integrate various little things

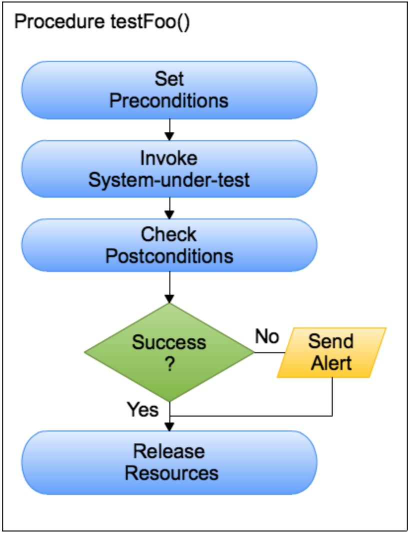
Projects using pFUnit

- Development of pFUnit itself (bootstrapping)
- New implementation of SMVGear chemistry solver
- Large portion of re-engineered DYNAMO (pseudospectral MHD)
- Virtual snowflake simulation
  - Initial implementation serial
  - pFUnit used to develop MPI extension
  - pFUnit used to create entirely new multi-lattice version
- A couple of small packages in GISS modelE
  - hand timers
  - Tracer metadata infrastructure

pFUnit Architecture



Anatomy of a Unit Test



Outline

System requirements

- Unix (Linux, OS X, ...)
- GNU make
- Fortran 95 compiler with F2003 C-Interoperability extensions
  Currently supported compilers:
  - Intel (ifort)
  - GNU (gfortran)
  - NAG (nagfor)
  - IBM (xlf)
  - PGI (pgf)

  Porting to other compilers should be straightforward.
- MPI - optional

Outline

Installation - obtaining source code

pFUnit is maintained in a git repository on sourceforge

- Via git from sourceforge:

```
% git clone git://pfunit.git.sourceforge.net/gitroot/pfunit/pfunit pFUnit
```

- Or use your browser to download nightly snapshot
  http://sourceforge.net/projects/pfunit/files/Source/pFUnit.tar.gz/download

```
% tar -xzf pFUnit.tar.gz
```

Installation - build library and self tests

1. Change directory

```
% cd pFUnit
```

Installation - build library and self tests

1. Change directory

```
% cd pFUnit
```

2. Build library and run self tests

```
% make tests
...
tests/tests.x
...............................
 103 run, 0 failed 0.03 seconds
```

Installation - build library and self tests

1. Change directory

```
% cd pFUnit
```

2. Build library and run self tests

```
% make tests
...
tests/tests.x
...............................
 103 run, 0 failed 0.03 seconds
```

```
% make tests MPI=YES
...
mpirun -np 5 ./mpi_pFUnit.x
...............................
 115 run, 0 failed 0.07 seconds
```

Installation - build library and self tests

1. Change directory

```
% cd pFUnit
```

2. Build library and run self tests

```
% make tests
...
tests/tests.x
...............................
 103 run, 0 failed 0.03 seconds
```

```
% make tests MPI=YES
...
mpirun -np 5 ./mpi_pFUnit.x
...............................
 115 run, 0 failed 0.07 seconds
```

3. Override default compiler

```
% make tests F90_VENDOR=<vendor>
```

Table: Supported compilers

F90_Vendor	Compiler
Intel (default)	ifort
NAG	nagfor
IBM	xlf
PGI	pgf90
GNU/ Gfortran	gfortran

Installation - final step

1. Choose a location (outside pfunit source) in which to install libraries, include files, and Fortran modules.
2. Set the PFUNIT environment variable to the chosen location
   You will want a separate directory for MPI and serial builds of pFUnit.

bash
```
% export PFUNIT=<path>
```

csh,tcsh
```
% setenv PFUNIT <path>
```

3. Use make to perform installation step

```
% make install INSTALL_DIR=$PFUNIT
```

If installation was successful then you should see the following subdirectiories:

```
% ls $PFUNIT
bin include lib mod
```

Outline

- These slides ...
- User guide - distributed with source ($\LaTeX$ document)
- API reference manual
  - http://sourceforge.net/projects/pfunit/files/Documentation
  - PDF and/or HTML

- These slides ...
- User guide - distributed with source (LaTeXdocument)
- API reference manual
  - http://sourceforge.net/projects/pfunit/files/Documentation
  - PDF and/or HTML

Note that documentation is not being actively maintained.

Outline

API - Module pFUnit

The public interfaces to pFUnit are re-exported through a module called "pFUnit".

API - Module pFUnit

The public interfaces to pFUnit are re-exported through a module called "pFUnit".

Thus to access pFUnit data types and procedures, one merely needs to add a F90 USE statement at the beginning of a module/subroutine/function:

```
use pFUnit
```

API: Assertions

API: Assertions

- Caution - not the same as C-style ASSERT macro

API: Assertions

- Caution - not the same as C-style ASSERT macro
- Unit tests specify behavioral costraints with *assertions*.

API: Assertions

- Caution - not the same as C-style ASSERT macro
- Unit tests specify behavioral costraints with *assertions*.
- Test succeeds only if all contained assertions are valid

API: Assertions

- Caution - not the same as C-style ASSERT macro
- Unit tests specify behavioral costraints with *assertions*.
- Test succeeds only if all contained assertions are valid
- When an assertion fails
  - pFUnit logs the test as failing
  - pFUnit accumulates list of failure messages for reporting

API: Assertions

- Caution - not the same as C-style ASSERT macro
- Unit tests specify behavioral costraints with *assertions*.
- Test succeeds only if all contained assertions are valid
- When an assertion fails
  - pFUnit logs the test as failing
  - pFUnit accumulates list of failure messages for reporting
- Support for all intrinsic data types:

Logical	true, false
Integer	equal
Real	equal, within tolerance, (less than, ...)
String	same, optionally ignore differences in white space

API: Assertions

- Caution - not the same as C-style ASSERT macro
- Unit tests specify behavioral costraints with *assertions*.
- Test succeeds only if all contained assertions are valid
- When an assertion fails
  - pFUnit logs the test as failing
  - pFUnit accumulates list of failure messages for reporting
- Support for all intrinsic data types:

Logical	true, false
Integer	equal
Real	equal, within tolerance, (less than, ...)
String	same, optionally ignore differences in white space

- Support for arrays: (Real - 5 dimensions, Integer - 1 dimension)
  - Can compare against scalar or conformable array
  - Reports first location that differs
  - Uses L_∞ norm, but has hooks for other norms (unused)
  - Will be adding interface for *relative* error

API: Assertions (cont'd)

The most common form of assertion is:

```
call assertEqual(<expected>, <found>, <message>)
```

- Test fails if found is diferrent than expected
- Overloaded for integer, real (single and double), and string
- Overloaded for multidimensional arrays

API: Assertions (cont'd)

The most common form of assertion is:

```
call assertEqual(<expected>, <found>, <message>)
```

- Test fails if found is diferrent than expected
- Overloaded for integer, real (single and double), and string
- Overloaded for multidimensional arrays

- Example:

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

API: Assertions (cont'd)

The most common form of assertion is:

```
call assertEqual(<expected>, <found>, <message>)
```

- Test fails if found is diferrent than expected
- Overloaded for integer, real (single and double), and string
- Overloaded for multidimensional arrays

- Example:

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

  Output from a failed assertion looks like:

  ```
  Failure in top::testFactorial – Integer scalar assertion failed:
                   factorial broken
      Expected:   120
      but found:  24
  ```

API: Assertions (cont'd)

- **assertEqual**(expected, found, tolerance , message)
  - Throws exception if difference is larger than tolerance
  - Example:

```
call assertEqual(totalMass, sum(mass(:,:,:)), 0.0001)
```

- **assertTrue**(test , message)
  - Throws exception if logical test is false
  - Example:

```
call assertTrue(pressure < 1100., 'pressure limit')
```

- **assertFalse** (test , message)
  - Throws exception if logical test is true
- Note message argument is *always* optional
  - Appends informative text to default text

API: Assertions (cont'd)

With only the Assertion module, developers can create a variety of complete unit tests. E.g.,

```
subroutine testSumFrom1toN()
  use pFUnit

  call assertEqual(10, sumFrom1toN(4))
end subroutine testSumFrom1toN
```

API: Assertions (cont'd)

With only the Assertion module, developers can create a variety of complete unit tests. E.g.,

```
subroutine testSumFrom1toN()
  use pFUnit

  call assertEqual(10, sumFrom1toN(4))
end subroutine testSumFrom1toN
```

or

```
subroutine testComputeDerivative()
  use pFUnit
  real :: u(3)
  real :: dudx(2)
  real :: dx = 1.

  u(:,1) = [1.,2.,2.,0.]
  call computeDerivative(u, dx, dudx)

  call assertEqual([1.,0.,-2.], dudx)

end subroutine testComputeDerivative
```

API: Exceptions

- Unit tests signal failed assertions by "throwing" an *exception*.

API: Exceptions

- Unit tests signal failed assertions by “throwing” an *exception*.
- The framework “catches” exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

API: Exceptions

- Unit tests signal failed assertions by “throwing” an *exception*.
- The framework “catches” exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions

API: Exceptions

- Unit tests signal failed assertions by “throwing” an *exception*.
- The framework “catches” exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions
- **Kludge:** Global stack of Exception_type variables

API: Exceptions

- Unit tests signal failed assertions by "throwing" an *exception*.
- The framework "catches" exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions
- **Kludge:** Global stack of Exception_type variables
- **Limitations:**

API: Exceptions

- Unit tests signal failed assertions by "throwing" an *exception*.
- The framework "catches" exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions
- **Kludge:** Global stack of Exception_type variables
- **Limitations:**
  - Requires manual return to caller

API: Exceptions

- Unit tests signal failed assertions by "throwing" an *exception*.
- The framework "catches" exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions
- **Kludge:** Global stack of Exception_type variables
- **Limitations:**
  - Requires manual return to caller
  - Errors (as opposed to failures) crash the framework

API: Exceptions

- Unit tests signal failed assertions by "throwing" an *exception*.
- The framework "catches" exceptions
  - Test is recorded as failing
  - Failure messages are aggregated
  - Framework continues to next test

- **Problem:** Fortran lacks native support for exceptions
- **Kludge:** Global stack of Exception_type variables
- **Limitations:**
  - Requires manual return to caller
  - Errors (as opposed to failures) crash the framework
  - Obtaining file & line number of failure is more difficult

API: Exception Class (cont'd)

Primary methods:

- throw() Pushes an exception onto the global stack.

```
type (Exception_type) :: myException
myException = Exception('Another exception')
call throw(myException)
```

Useful shortcut for usual case:

```
call throw('This is an exception')
```

API: Exception Class (cont'd)

Primary methods:

- throw() Pushes an exception onto the global stack.

```
type (Exception_type) :: myException
myException = Exception('Another exception')
call throw(myException)
```

  Useful shortcut for usual case:

```
call throw('This is an exception')
```

- catch() Returns true if specified exception has been thrown
  - Default - delete exception from global stack
  - Override with optional argument preserve=.true.

```
if (catch()) then ! true if global stack is non-empty
if (catch('This is an exception')) then
if (catch(anException)) then
```

API: Exception Class (cont'd)

Primary methods:

- throw() Pushes an exception onto the global stack.

```
type (Exception_type) :: myException
myException = Exception('Another exception')
call throw(myException)
```

  Useful shortcut for usual case:

```
call throw('This is an exception')
```

- catch() Returns true if specified exception has been thrown
  - Default - delete exception from global stack
  - Override with optional argument preserve=.true.

```
if (catch()) then ! true if global stack is non-empty
if (catch('This is an exception')) then
if (catch(anException)) then
```

- catchAny() returns top exception on the stack

```
anException = catchAny()
```

API: Exception Class (cont'd)

Additional methods - should rarely be needed.

- clearAll() - empty the stack
- == - compare two exceptions
- numExceptions()
- getMessage - return string from inside derived type

API: BaseAddress and ProcedurePointer

BaseAddress_type encapsulates a base address for a data entity

- Allows framework to manipulate user-defined data structures
- Only needed for test *fixtures* - discussed elsewhere
- Current implementation uses new F2003 C-interoperability
- Original implementation used semi-portable hack
- Could probably now be replaced by F2003 unlimited polymorphic entities

API: BaseAddress and ProcedurePointer

BaseAddress_type encapsulates a base address for a data entity

- Allows framework to manipulate user-defined data structures
- Only needed for test *fixtures* - discussed elsewhere
- Current implementation uses new F2003 C-interoperability
- Original implementation used semi-portable hack
- Could probably now be replaced by F2003 unlimited polymorphic entities

ProcedurePointer_type encapsulates a base address for a procedure

- Allows framework to aggregate user-defined test procedures
- Uses F2003 C-interoperability
- Could probably now be replaced by F2003 procedure pointers

API: TestResult class

TestResult_type: derived type that accumulates findings from running a sequence of tests.

- How many tests have run
- How many tests have failed
- Accumulate report of failure messages
- Wall clock time passed

API: TestResult (cont'd)

- **newTestResult**() - generate a pristine report object
- **summary**(this) - return 1 line string summarizing results

```
103 run, 2 failed 0.12 seconds
```

- **generateReport**(this) - return a Report_type that contains a list of all failure messages; prepends with test/suite hierarchy
- **setReportMode**(this, mode) - control logging
  - **MODE_USE_BUFFER** (default)
    - output failure messages to internal buffer
  - **MODE_USE_STDOUT**
    - track progress (emit '.', 'x', or 'm') for each test
  - **MODE_USE_LOGFILE**
    - track progress by testname in hidden ".pFUnitLog" file

Advanced Topic - Test Fixtures

Often, several tests require identical initialization steps for a group of input variables.

A test *fixture*:

- Provides a container for the shared input variables
- Provides a setUp() method to allocate resources and/or initialize elements
- Provides a tearDown() method to deallocate resources

Test Fixtures in pFUnit

Fixtures in pFUnit are a bit of a kludge due to lack of polymorphism in F95.

Two "obvious" approaches:

1. Use module variables and/or derived types that are accessed by test procedures
   - Pro: easy to implement and code
   - Con: somewhant easy to accidentally share data between tests
2. Fake polymorphism by passing around a BaseAddress
   - Pro: Framework provides fresh fixture for each test method
   - Con: Requires a wrapper to handle fixture dereference
   - Con: Wrapper complicates makefile

pFUnit supports both approaches. Focus and documentation has been on the latter.

pFUnit simple fixture

```
module myTestModule
   use pfunit
   private
   public :: setUp, tearDown, test1

   real, allocatable :: buffer(:) ! the fixture

contains

   subroutine setUp()
      integer :: i
      allocate(buffer(10))
      buffer = [(i,i=1,10)]
   end subroutine setUp

   subroutine tearDown()
      deallocate(buffer)
   end subroutine tearDown

   subroutine test1()
      call assertEqual(55, sum(buffer))
   end subroutine test1

end module myTestModule
```

pFunit Fixture with Derived Type

```
module myTestModule
   use pfunit
   private
   public :: fixture, setUp, tearDown, test1
   type fixture
      real, allocatable :: buffer(:)
   end type
contains
   subroutine setUp(this)
      type (fixture), intent(inout) :: this
      allocate(this%buffer(10))
      this%buffer = [(i,i=1,10)]
   end subroutine setUp

   subroutine tearDown(this)
      type (fixture), intent(inout) :: this
      deallocate(this%buffer)
   end subroutine tearDown

   subroutine test1(this)
      type (fixture), intent(in) :: this
      call assertEqual(55, sum(buffer))
   end subroutine test1
end module myTestModule
```

pFunit Fixture Wrapper

```
module myTestModule_wrap
   use myTestModule, only: fixture => fixture_private
   use myTestModule, only: setUp => setUp_private
   use myTestModule, only: tearDown => tearDown_private

   type fixture
      type (fixture_private) :: user_fixture
      type (fixture), pointer :: self_reference
   end type


   ! Continue next screen
```

```fortran
contains

   subroutine setUp(this)
      type (fixture) :: this
      call setUp_private(this%user_fixture)
   end subroutine setUp

   subroutine tearDown(this)
      type (fixture) :: this
      call tearDown_private(this%user_fixture)
   end subroutine tearDown

   subroutine test1(this)
      type (fixture) :: this
      call test1_private(this%user_fixture)
   end subroutine test1

end module myTestModule_wrap
```

API - TestCase class

TestMethod_type derived type that binds a procedure pointer with a meaningful name (string)

- Required because Fortran lacks reflection/introspection
- Allows framework to report *which* test ran/failed
- Allows framework to select which tests to run

TestCase_type derived type that contains

- List of related test methods (usually just 1)
- Procedure pointers for setUp() and tearDown()

API - TestCase constructors (overloaded)

- test = **TestCase**() - used internally
- test = **TestCase**(**setUp**, **tearDown**) - used internally fixture
- test = **TestCase**(name, procedure) - 1 Step construction
- test = **TestCase**(**setUp**, **tearDown**, passFixture) - "kludge" fixture
- test = **TestCase**(**setup**, **teardown**, name, procedure) - convenience

Methods are accumulated via

```
call addTestMethod(this, name, procedure)
```

API - TestCase::run()

`call run(this, aTestResult)` performs the following steps for each method

1. Call testStarted()
2. Allocate fixture if any
3. Call setUp() if any
4. Check for exceptions
5. If good so far
   1. Run the test method
   2. Check for exceptions
6. call tearDown() if any

API - MpiTestCase

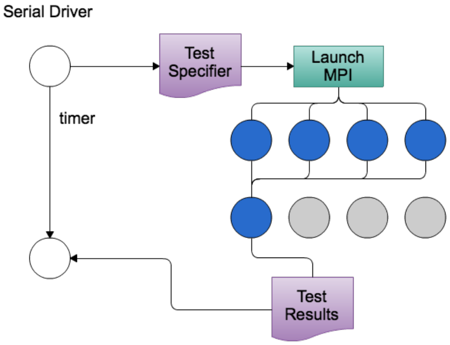
An MPI test case runs a test procedure on a group of MPI processes
Implementation considerations:

- Must allow for tests using varying number of processes
- Need mechanism to specify number of processes to use
- Most MPI implementations are not *reentrant*
- pFUnit self tests need to be able to run MPI test within a serial test

1. Client - Server
   1. Persistent server
   2. Relaunch server for each test
2. Use MPI subcommunicators within executable
   1. With MPI_spawn()
   2. Max NPES determined at launch

API - MpiTestCase Client-Server?

- Serial client interacts with MPI-based Server
- Server can be persistent or relaunched for each test



Pro Pure - MPI can be relaunched for each test

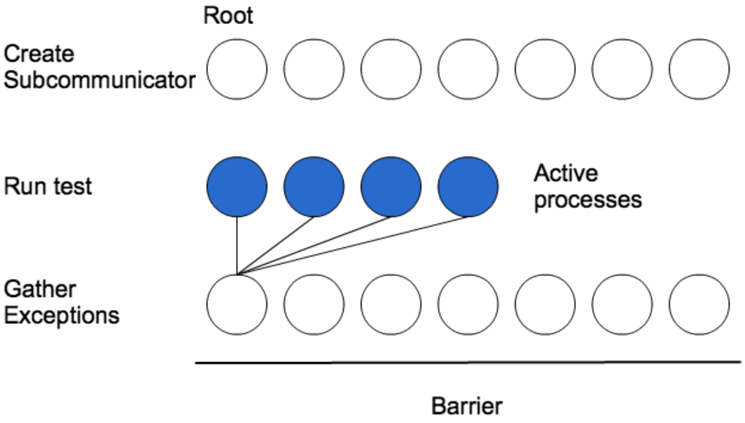
Pro Can support time limits

Con 1 second overhead per test - gets expensive

Con Complex/fragile mechanism

API - MpiTestCase Subcommunicator?

- Pro Low overhead per test
- Pro Relatively simple driver mechanism
- Con Cannot support time limits (Mpi_abort() issues)



API - MpiTestCase (cont'd)

Usage is very similar to the regular **TestCase**, except:

- Constructor - requires extra argument **numProcesses**

```
        MpiTestCase(name, method, numProcesses)
```

- Test method requires extra "info" argument (intent(in))

```
subroutine myMPItest(info)
   type (TestInfo_type), intent(in) :: info
...
```

API - TestInfo

The **TestInfo_type** derived type:

- Passes the mpiCommunicator to be used by the test
  NO MPI_COMM_WORLD

```
comm = mpiCommunicator(info)
```

- Convenient access to other MPI values that are usually needed for setting up an MPI test.

```
npes = numProcesses(info)
rank = processRank(info)
```

- Several other procedures used internally by pFUnit

```
if (amRoot(info)) ...
if (amActive(info)) ... ! participate in test
call barrier(info)
```

TestSuite

The **TestSuite_type** derived type is a container for organizing test cases.

- E.g. fast tests that are *always* run
- Slow tests run overnight, or weekend
- Personal tests vs tests for full application

The primary interfaces are

- Constructors

```
mySuite = TestSuite('mySuiteName') ! empty
mySuite = TestSuite('mySuiteName',suites) ! group of pre-existing
```

- Add a test

```
call add(mySuite, test)
```

Where test is any of:

- **TestCase_type**
- **MpiTestCase_type**
- **TestSuite_type**
- **ParameterizedTestCase_type**

Outline

A simple driver

```
program TestDriver
   use pFUnit
   use TestMyModule_mod

   type (TestSuite_type)   :: suite
   type (TestResult_type)  :: result
   character(len=100)      :: summary_statement
   call pFUnit_init()

   ! Build suite from test procedures:
   suite = TestSuite('My test subroutines')
   call add(suite, TestCase1Step('testMySub', testMySub))
   call add(suite, TestCase1Step('anotherTest', anotherTest))

   result = newTestResult(mode=MODE_USE_STDOUT)
   call run(suite, result)

   summary_statement = summary(result)
   print*, trim(summary_statement)

   call clean(result)
   call clean(suite)
   call pFUnit_finalize()
end program TestDriver
```

Maintaining the List of Tests

3 choices:

- Manual
  - Tedious
  - Error prone - failing test never gets called
- Automation - use preprocessing to find test cases
  - Requires a convention for test names
  - Current mechanism is a bit fragile
- DSO's - not actively supported at this time
  - Developer must create DSO for tests and application
  - No mechanism for spcefying number of MPI processors

Automated assembly of tests

pFUnit includes an automation mechanism - users my wish to improve it.

- Separate directory (or directories) of tests
- Tests are all module procedures
- Tests must all start with the string "test..."

More details

- Each module containing tests is wrapped by an automatically generated module which bundles them into a suite.
- MPI test suites are indicated with *** mpi test cases *** on the 1st line
- A skeleton driver has a master test suite that includes a suite from each of the test modules.
- Developer should have

```
include $PFUNIT/include/pFUnit.makefile
```

in their makefile

Outline

F2kUnit - next release of pFUnit

- Heavily leverages OO features of Fortran 2003
- Complete rewrite from scratch - following design of JUnit
- Superior extensibility to be extended through OO
  - TestListeners - alternate reporting; e.g., Eclipse Photran
  - TestRunners - customize means to select tests
- Basic implementation complete - lacks many bells and whistles
- Upgrade from current release should be relatively easy

Outline

Exercises

You will be attempting 3 exercises that use pFUnit and TDD.
https://modelingguru.nasa.gov/docs/DOC-2222
Each exercise contains a README file with instructions, and is divided into multiple steps. If you get stuck on one step, the solution is the starting point for the next step. E.g. the code in 1-B is the solution for the exercise in 1-A.
Please do not hesitate to ask questions.

The provided Makefile's are designed to work with the Intel compiler. Exercises 1 and 3 should be done with a *serial* build of pFUnit, and exercise 2 should be with a parallel build.

- Exercise 1 is intended to be very simple to allow you to focus on the pFUnit interfaces.
- Exercise 2 uses MPI. Attendees that are not familiar with MPI are encouraged to work with a partner or to proceed to Exercise 3
  - On Janus we recommend:

```
use NCAR-Parallel-Intel
```

- Exercise 3 builds upon the interpolation example from the morning session.