Unit testing in GS2 with pFUnit

Peter Hill[1]

[1] York Plasma Institute, University of York

Why testing?

Why are tests important?

- Code is wrong $\rightarrow$ wrong results $\rightarrow$ bad things happen
  - Need to be confident of results
  - Bugs are a fact of life
- How do you change code?
  - Either *"Edit and Pray"* or *"Cover and Modify"*
- Proper tests will:
  - Catch common bugs
  - Catch edge cases
  - Catch errors early
  - Reduce time to solution
  - Allow you to make changes confidently

If it doesn't have tests, it's wrong!

Unit Testing

Problem

- Integrated tests are slow...
  - so not run as often
- ...and broad
  - hard to pinpoint exactly where error is when test fails
- (although still necessary!)

Solution 1

- Somewhat mitigated by atomic, orthogonal commits: commit should change one thing, with descriptive commit message
- Commit messages are documentation! (possibly only documentation some people will ever write)
- Therefore vital that people write good commit messages!
  - Cultural problem, can't be technically enforced

Unit Testing

Solution 2

- Unit tests:
  - Test the smallest possible bit of code (normally function)
  - Tests should be orthogonal, independent
  - Tests should be fast: "1/10th of a second is *too slow*"
- Added unit test suite using pFUnit framework
  - included as git submodule, allows bundling of code with ease of updating
  - ~54 tests run in ~0.06 seconds
- With mandatory code review at PR stage, can now request new features have tests (preferring unit tests)

What do we want from our unit tests?

- Marginal cost of adding a new test cheap as possible
- Must be fast
- Must be specific
- Must be sensitive!

Difficulties of unit testing

- What's the answer?
  - Scientific codes developed to tell us the answer!
- May be lots of edge cases
  - => Lots of tests
- Important to test failure modes as well
  - Nonsense input should not work!
- Not always possible to test full domain
  - Testing all `reals` is doable
  - Testing all combinations of two `reals` is not

Solutions

- "Golden answer"/regression tests
- Very reduced analytic case
- Artificial modularisation

Golden answers/regression tests

- Call the function with some parameters
- Hard code the result as the expected answer
- May not actually be correct...
- ...but definitely shouldn't change unexpectedly!
- Answer may need updating occasionally

Reduced analytic cases

- Velocity integral of distribution function
- In production, don't know form of g *a priori*, so $\int g dv$ definitely unknown
- But in testing, we have control of g
- So pick an easy form to check the integration
  - Why not $g = \sin(v)$

Reduced analytic cases

- May need to test something more "physical"
- Velocity moment of drift-reduced distribution function at fixed p_ϕ
- But simulation done in (r, θ)
- Implemented as loops over one set of coordinates
- Kernel of loop involves interpolating f in p_ϕ
- This involves physics – form of B, etc.
- But can pick these to be analytic functions and get exact form of integral
  - May need e.g. Mathematica or Sympy

Reduced analytic cases

- Numerical methods are inexact by their nature
  - e.g. finite difference scheme may be $O(h^2)$
- Usually need to specify a tolerance
- Balance between speed of test and accuracy
  - Too few points: need looser tolerance
  - Too many points: tests take too long
  - Tolerance too loose: more false negatives
  - Tolerance too tight: more false positives
- May need second set of tests for scaling
  - e.g. check error really scales as h^2

Artificial modularisation

- Function may just be difficult to test as a whole
- Split into smaller functions and test those!
- Pros:
  - Gives names to parts
  - Parts may be reusable
  - May make whole clearer
- Cons:
  - May reduce performance
  - Parts may not be reusable

Digression: naming parts

What does this do?

```
  long i;
  float x2, y;
  const float threehalfs = 1.5F;
  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;     // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );  // what the ****?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
// 2nd iteration, this can be removed
// y  = y * ( threehalfs - ( x2 * y * y ) );
```

Digression: naming parts

“Fast inverse square root” from Quake

```
float Q_rsqrt(float number) {
  long i;
  float x2, y;
  const float threehalfs = 1.5F;
  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;     // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );  // what the ****?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
// 2nd iteration, this can be removed
// y  = y * ( threehalfs - ( x2 * y * y ) );
  return y
}
```

Artificial modularisation

- Build implementation up out of smaller blocks
- Thoroughly test these blocks
- Once whole is complete, manually inline
- Put regression test on new implementation

Difficulties of unit testing in Fortran

- No reflection/introspection
  - Hard to find all tests
- Necessary separation of declarations/definitions
  - => Either need preprocessor or "boilerplate"
- Lack of utility types in standard (no `map`/`dict`)
  - => Large cost of writing framework
- No exceptions
  - => Difficult to signal errors non-fatally

Enter pFUnit

- Python preprocessor
- Developed by (at) NASA
- Natively understands MPI
- Does most of the heavy lifting
- Essentially only extant, well-developed, parallel choice!

Simple example

Hello world

```
@test
subroutine testHelloWorld()
  use pfunit_mod
  implicit none
  @assertEqual("Hello World!", "Hello World!")
end subroutine testHelloWorld
```

- `@token` is something for python preprocessor
- `@test` denotes a test
- `@assertEqual` checks the two arguments are equal

Floating point numbers

Floating point numbers are a bit strange

```
@test
subroutine test_addition()
  use pfunit_mod
  implicit none
  ! Following fails:
  @assertEqual(0.3, 0.1 + 0.2)
end subroutine test_addition
```

Floating point numbers

```
Failure
 in:
myTests_suite.test_addition
  Location:
[myTests.pf:6]
expected +0.3000000 but found: +0.3000000; \
         difference: |+0.5551115E-16| > tolerance:+0.000000.
```

Floating point numbers

Floating point numbers are a bit strange

```
@test
subroutine test_addition()
  use pfunit_mod
  implicit none
  ! This passes:
  @assertEqual(0.3d0, 0.1d0 + 0.2d0, tolerance=1.d-15)
end subroutine test_addition
```

More useful error messages

message keyword

```
@test
subroutine test_addition()
  use pfunit_mod
  implicit none
  @assertEqual(0.3d0, 0.1d0 + 0.2d0, message="Check associativity")
end subroutine test_addition
```

More useful error messages

```
Failure
 in:
myTests_suite.test_addition
  Location:
[myTests.pf:6]
Check associativity expected +0.3000000 but found: +0.3000000; \
          difference: |+0.5551115E-16| > tolerance:+0.000000.
```

Running tests on multiple rank counts

npes argument to @test

```
@test(npes=[1,2,3])
subroutine test_MPI_thing(this)
  use mp, only : init_mp
  use pfunit_mod
  implicit none
  class (MpiTestMethod) :: this
  integer :: rank

  rank = this%getProcessRank()
  call init_mp(this%getMpiCommunicator())

  ...
end subroutine test_MPI_thing
```

Reusing code between tests

Fixtures

```
module fixtures
  implicit none
  real, dimension(:), allocatable :: data
contains
  @before
  subroutine setup()
    allocate(data(4))
    data = [1, 2, 3, 4]
  end subroutine setup()
  @after
  subroutine teardown()
    if (allocated(data)) deallocate(data)
  end subroutine teardown
  ...
```

pFUnit in GS2

- `make pfunit_tests`

```
tests/
 \-pfunit_tests/
    |- geo/
    |   \- test_leq.pf
    |- Makefile
    \- testSuites.inc
```

`testSuites.inc`:

```
ADD_TEST_SUITE(test_leq_mod_suite)
```

How to add new test module

- New file in `tests/pfunit_tests/<dir>/test_<name>.pf`
- Should contain a module called `test_<name>_mod`
- Add `ADD_TEST_SUITE(test_<name>_mod_suite)`

How to test `private` procedures?

- Lots of modules contain various `private` routines
- Some are useful/necessary for initialising the module
- Slightly gross method: derived type to expose them

leq

```
!> Expose private routines for testing only!
type :: leq_testing_type
contains
  procedure, nopass :: mod2pi
  procedure, nopass :: derm
  procedure, nopass :: set_ntnr
  ...
  procedure, nopass :: eqdbish
  procedure, nopass :: alloc_arrays, dealloc_arrays
end type leq_testing_type
```

How to test `private` procedures?

leq

```
@test
subroutine test_mod2pi()
  use constants, only : pi
  use leq, only : leq_testing_type
  type(leq_testing_type) :: leq_privates
  @assertEqual(0.0, leq_privates%mod2pi(0.0))
end subroutine test_mod2pi
```

Some technical details

- Modules wrapped up with list of tests
- `driver.F90` runs all the tests
- "Robust runner" runs each test in separate process
  - Allows tests to call `mp_abort`

Conclusions

```
module test_leq_mod
  use pfunit_mod
  implicit none
contains
  @test
  subroutine test_mod2pi()
    use constants, only : pi
    use leq, only : leq_testing_type
    type(leq_testing_type) :: leq_privates
    @assertEqual(0.0, leq_privates%mod2pi(0.0))
  end subroutine test_mod2pi
```