Modern Fortran: Useful tools and techniques

Bálint Aradi

2018 Workshop on Fortran Modernization for Scientific Application (ForMoSA)
Outline

Development workflow

- **Unit testing**: pFUnit

- **Coverage testing**: lcov & genhtml

- **API-documentation**: doxygen / ford
Testing your code (pFUnit)
Program testing

1. Write program
2. Check correctness for selected cases
3. Extend / Refactor program
   - Check correctness of new functionality for selected cases
4. Are old features still functional?!
Program testing

When to test?

- Package functionality/integrity must be tested after each (relevant) change
- Package functionality/integrity must be tested whenever it is used in a different environment

How to test?

Effort needed to carry out tests must be as low as possible

- It should be possible to run all (or selected tests) with one command
- Tests should be reasonably fast
- Correctness of the results should be checked automatically (you definitely won’t catch a sign change in a 100 x 100 matrix at midnight by visual comparison)

Automated testing (with test protocol) is an essential part of the development work-flow
Unit testing vs. regression testing

Unit tests – white box testing
- Each program unit (e.g. function) is tested independently
- Check whether for given input the right output is returned
- Test driven development (e.g. agile programming)
  - First write the tests for a given functionality, then implement the functionality and make sure it passes the test
  - If a bug is found, add it first as test (improve coverage) and then fix it so that it passes the test

Regression tests – black box testing
- Testing the package functionality as a whole
- Testing whether for given input (e.g. user supplied data) expected output is generated
- All possible code paths should be tested (coverage!)
- Could also include stress-tests or scaling tests
Fortran unit testing frameworks

The ones I am aware of. The list may not be complete..

- **ftnunit** (written by Arjen Markus)
- **FRUIT** (Fortran Unit Test Framework by Andrew Hang Chen)
- **pFUnit** (written by NASA developers)

I’ll cover pFUnit
(just because this is the only one I have practical experience with)

**Before implementing any algorithm you should ask following questions:**

- How do I test it?
- How do I implement it?

In some work-flows you even write the tests first and only later the implementation (see agile programming & test driven development)
Writing a simple test in pFUnit

- Let’s implement some functionality first

```fortran
module mymath
  implicit none

contains

  function factorial(nn) result(fac)
    integer, intent(in) :: nn
    integer :: fac

    ! Put factorial implementation here
  end function factorial

end module mymath
```

- Compile it into an object file

```
gfortran -c -o mymath.o mymath.f90
```
Writing a simple test in pFUnit

- pFUnit uses a **meta-programming** approach
- Special test-file must be processed with pFUnits preprocessor first

```fortran
module test_mymath
    use mymath
    use pfunit_mod
    implicit none
contains
    @test
    subroutine test_factorial_5()
        @assertEqual(120, factorial(5))
    end subroutine test_factorial_5
end module test_mymath
```

Indicates a test routine

Checks for a condition:

- **if fulfilled**, nothing happens, test continues
- **if not fulfilled**, test executions is stopped (test failed)

Checks for equality

Test succeds if subroutine finishes (without stopping due failing assert)
Writing a simple test in pFUnit

- Transform your meta-Fortran test into standard Fortran

```
env F90_VENDOR=GNU /opt/pfunit/bin/pFUnitParser.py test_mymath.pf test_mymath.F90
```

- Compile it

```
gfortran -I/opt/pfunit/include -I/opt/pfunit/mod -c -o test_mymath.o test_mymath.f90
```
Writing a simple test in pFUnit

- Create a special file `testSuites.inc` to register your test

```plaintext
TestSuites.inc
ADD_TEST_SUITE(test_mymath_suite)
```

- Then, you have to **compile the test driver**, which will execute the tests
- The driver is **provided by pFUnit**. Unfortunately, it must be preprocessed by another preprocessor (fpp)

```plaintext
gfortran -E -I. /opt/pfunit/include/driver.F90 > pfunit_driver.f90
gfortran  -I/opt/pfunit/include -I/opt/pfunit/mod -c pfunit_driver.f90
```

- **Link everything together** (cross your fingers to avoid unresolved refs)

```plaintext
gfortran  -o pfunit_driver pfunit_driver.o test_mymath.o mymath.o -L../../opt/pfunit/lib -lpfunit
```
Now RUN your TEST

- Run your test
  
  
  ```
  ./pfunit_driver
  ```

  Time: 0.000 seconds

  OK
  (1 test)

  No failures!
Now, extend the tests

- Make sure you also **check for special cases**

```plaintext
@test subroutine test_factorial_0()
    @assertEqual(1, factorial(0))
end subroutine test_factorial_0

@test subroutine test_factorial_negative()
    @assertEqual(1, factorial(-1))
end subroutine test_factorial_negative
```

Probably, our function should return some other values for negative inputs...
Randomized consistency checks

- Sometimes, **consistency checks** can be done with random numbers:
- Make sure you **print the random number in case of failure**, so that **failing test** can be **reproduced**.

```fortran
@test
subroutine test_random_consistency()

  real :: rand
  integer :: nn
  character(100) :: str

  call random_number(rand)
  nn = int(10.0 * rand) + 1
  write(str, '(A,I0)') 'Failed with argument value ', nn
  @assertTrue(factorial(nn) == nn * factorial(nn - 1),&
                & trim(str))

  Checks for .true.
end subroutine test_random_consistency
```
Parameterized tests

- When **repeating** the **same test** again and again just **with different data**

Parameterized tests in pFUnit

- You need to define a **separate module for each parameterized test**

```plaintext
module test_mymath_param
    use mymath
    use pfunit_mod
    implicit none
```

```plaintext
test_mymath_param.pf
```
The module must contain a test parameter object containing a specific value of the test parameter(s).

```plaintext
@testParameter
type, extends(AbstractTestParameter) :: TestParam
   integer :: number, factorial
contains
   procedure :: toString => TestParam_toString
end type TestParam
```

The test parameter should contain the number we want to calculate the factorial of, and the result.

You have to overwrite the `toString()` method of the abstract base class (deferred).

It should return a string representation of the given test parameter values (for error messages).
ParametrizedTestCase object

- The module must contain a **test case object** which is created with the individual instances of the test parameters
- It should be **derived from the ParameterizedTestCase class** for parameterized tests

```pseudocode
@TestCase(testParameters={get_test_params()},
constructor=get_factorial_test)
type, extends(ParameterizedTestCase) :: FactorialTest
type(TestParam) :: testpar
end type FactorialTest
```

- Function returning a test case for a given value of the test parameter(s)
- Function returning an array with all possible test parameters
- The test case instance should keep a copy of the actual test parameters
All possible test parameter values

- The individual test parameters are returned by looping over the result of the specified **test-parameter construction function**.
- The test parameter function should return an **array with test parameter instances**.

```fortran
contains

function get_test_params() result(testparams)
  type(TestParam), allocatable :: testparams(:,)

  testparams = [&
    & TestParam(2, 2),&
    & TestParam(5, 120)&
    & ]

end function get_test_params
```

Initializing derived type TestParam instances
1\textsuperscript{st} arg = number
2\textsuperscript{nd} arg = factorial

*test_mymath_param.pf*
Test constructor

- Individual test instances are initialized using the specified “constructor” function.
- The constructor function receives an instance of the actual test parameters as argument.
- The constructor function must return the initialized test case as result.

```plaintext
function get_factorial_test(testpar) result(test)
    type(TestParam), intent(in) :: testpar
    type(FactorialTest) :: test

    test%testpar = testpar

end function get_factorial_test
```

_test_mymath_param.pf_
Test function(s)

- The test function receives the initialized test instance as argument

```plaintext
@test
subroutine test_factorial_calc(this)
    class(FactorialTest), intent(inout) :: this
    @assertEqual(this%testpar%factorial, factorial(this%testpar%number))
end subroutine test_factorial_calc
```

The preprocessor turns it into a type-bound procedure, argument must be a class, not a type
At last, the test case needs to override a routine, which returns a string representation of the parameters (for printing error messages)

```plaintext
function TestParam_toString(this) result(string)
   class(TestParam), intent(in) :: this
   character(:), allocatable :: string

   character(100) :: buffer

   write(buffer, "((A,I0,A,I0)) \"in: \",&
       & this%number, \", out: \", this%factorial
   string = trim(buffer)

end function TestParam_toString
end module test_mymath_param
```
Build your test driver

- Transform your meta-Fortran test into standard Fortran

```bash
env F90_VENDOR=GNU /opt/pfunit/bin/pFUnitParser.py test_mymath.pf test_mymath.F90
```

- Compile it

```bash
gfortran  -I/opt/pfunit/include -I/opt/pfunit/mod -c -o test_mymath.o test_mymath.f90
```
Build your test driver

- Create a special file `testSuites.inc` which registers your test

  ```
  ADD_TEST_SUITE(test_mymath_suite)
  ADD_TEST_SUITE(test_mymath_param_suite)
  ```

- Then, you have to re-create and re-compile the test driver

  ```
gfortran -E -I. /opt/pfunit/include/driver.F90 > pfunit_driver.f90
f90
f90 -I/opt/pfunit/include -I/opt/pfunit/mod -c pfunit_driver.f90
  ```

  ```
gfortran -o pfunit_driver pfunit_driver.o test_mymath.o
test_mymath_param.o mymath.o -L../opt/pfunit/lib -lpfunit
  ```

- The driver will execute the parameterized tests for all test parameters

  ```
  ./pfunit_driver
  ```
Test fixtures

- Tests, which need the **same initialization or finalization**
- The TestCase object may define a `setUp()` and a `tearDown()` method, which are **executed before / after each test**.

```plaintext
@TestCase
type, extends(TestCase) :: MyTest
  integer :: someVariable
contains
  procedure :: setUp
  procedure :: tearDown
end type Test_LinearInterpolator
```

- The test subroutines will receive the initialized TestCase object as argument and can use any variables which have been set up during initialization
When pFUnit fails

When a test fails, pFUnit will report it:

```
@test
subroutine test_factorial_5()
    @assertEqual(125, factorial(5))
end subroutine test_factorial_5
```

Failure in:
test_mymath_suite.test_factorial_5
Location:
[test_mymath.pf:11]
exected 125 but found: 120; difference: |5|.

FAILURES!!!
Tests run: 6, Failures: 1, Errors: 0
ERROR STOP *** Encountered 1...
Final unit-test notes

- pFUnit has many more features
- Especially, pFUnit also supports testing MPI-parallelized routines

See the pFUnit project page for further details

And finally once more the main message:

ALWAYS DELIVER AN AUTOMATED TEST FRAMEWORK WITH YOUR SOFTWARE
Test coverage
Do you test really test all parts of your code?

Test coverage

- Indicates which amount of the total code lines have been executed at least ones during the tests.
- Desirable: 100%
- Note: **100% coverage does not mean bug free code!** It only means, that each line has been reached at least once during some tests. The code still can misbehave, if given line is executed with different (non-tested) data.

In order to record coverage information

- Recompile your code with the according special option (GNU: `--coverage`, you need the option for both, compiling and linking)
- Run your automated test framework
- Coverage information will be stored in `.gcda` files
Visualizing coverage statistics

- Collect coverage information
  
  `lcov -t 'mymath_coverage' -o mymath.covinfo -c -d .`

  - Report title name
  - Store info in mymath.covinfo
  - Only include coverage about source files in current folder
  - Capture coverage info

- Visualize coverage information

  `genhtml -o covreport mymath.covinfo`

  Folder with HTML-pages

  Processing file factorial/mymath.f90
  Writing directory view page.
  Overall coverage rate:
  
  - lines.......: 100.0% (5 of 5 lines)
  - functions..: 100.0% (1 of 1 function)
Visualizing coverage statistics

- Open the `index.html` file with your browser for the graphical coverage report

### LCOV - code coverage report

<table>
<thead>
<tr>
<th>Directory</th>
<th>Line Coverage</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial</td>
<td>100.0 %</td>
<td>100.0 %</td>
</tr>
</tbody>
</table>

**Filename**

<table>
<thead>
<tr>
<th>File</th>
<th>Line Coverage</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mymath.f90</td>
<td>100.0 %</td>
<td>100.0 %</td>
</tr>
</tbody>
</table>

Number indicate how often a line was executed.

---

**Line data**

```plaintext
1  module mymath
2    implicit none
3    contains
4
5  function factorial(nn) result(fac)
6      integer, intent(in) :: nn
7      integer :: fac
8      integer :: ii
9
10   fac = 1
11  do ii = 2, nn
12     fac = fac * ii
13  end do
14
15  end function factorial
16
17  end module mymath
```
When coverage is not 100%

Processing file linsolve/solver.f90
Writing directory view page.
Overall coverage rate:
  lines......: 93.5% (29 of 31 lines)
  functions..: 100.0% (2 of 2 functions)

```fortran
33 : integer :: nvar, imax, ii, jj
34 : real(dp), allocatable :: buffer(:)
35 : real(dp) :: tmp
36
37 1 : nvar = size(aa, dim=1)
38 1 : allocate(buffer(nvar))
39
40
41 1 : ipiv(:) = [(ii, ii = 1, nvar)]
42
43 4 : do ii = 1, nvar
44 3 :   imax = maxloc(abs(aa(ii:, ii)), dim=1) + ii - 1
45 3 :   if (abs(aa(imax, ii)) < TOLERANCE_DP) then
46 0 :     info = ii
47 0 :     return
48
49 :   end if
50
51 3 :   if (imax /= ii) then
52 2 :     buffer = aa(ii, :)
53 2 :     aa(ii, :) = aa(imax, :)
54 8 :     aa(imax, :) = buffer
```
Final test coverage note

- You should aim for 100% coverage, although that is often not possible.
- Even 100% test coverage does not warranty bug-free code.
- If you pre-process your source file, you should keep the pre-processed when building for coverage testing, as genhtml needs them.
API-documentation
Application Programming Interface (API)

- All public routines of your project
- They could be called by other projects / scripts by importing modules from this project (**reusability**!)

API-documentation

- Description of the purpose and input/output arguments of the API
- Description of the modules containing the API
- The modules / subroutines should be documented with specially placed and formatted comments.

Extracting API-documentation

- Documentation is extracted from the comments in the source code
- Generated documentation can be inspected without looking into the code (e.g. HTML-pages, PDF-document, etc.)
- Modules can be reused without knowing the internal code details
Wide-spread documentation systems for Fortran

- Doxygen
- Ford
- I think, some projects use RoboDoc

Doxygen

- Originally written for C++ (and the de facto standard tool for C++)
- Sort of works for Fortran
- Has problems with very Fortran specific constructs (e.g. interface)
- Robust, mature, well maintained, has been around for quite a while

Ford

- Written specially for Fortran
- Can deal with Fortran specific constructs
- A rather young project
Where to put the documentation?

- Into the source code directly before (or in Ford by default after) the documented entity

```fortran
!> Decomposes the matrix aa into the LU-form.
subroutine ludecompose(aa, ipiv, info)

!> Matrix to be decomposed on entry
!> LU-decomposed matrix on exit.
real(dp), intent(inout) :: aa(:, :)

!> Contains the row pivots made during the
!> LU-decomposition.
integer, intent(out) :: ipiv(:)

!> Info flag. Non-zero value indicates error.
!> In that case all other returned quantities are
!> meaningless
integer, intent(out) :: info
```
How to extract the documentation?

**Doxygen**

- Create config file with default settings
  
  ```
  doxygen -g
  ```

- Edit following fields in the generated config file (**Doxyfile**)

  - `PROJECT_NAME`: Set the name of your project
  - `PROJECT_BRIEF`: Brief (one-liner) description of your project
  - `OPTIMIZE_FOR_FORTRAN`: Set it to YES
  - `INPUT`: Set directory with source files, if not current
  - `HAVE_DOT`: Set to YES, if you installed graphviz

- Run `doxygen` to generate documentation
  
  ```
  doxygen
  ```

- Open the index.html file in the html/ directory in your browser
How to extract the documentation?

LinSolve
Didactic solver for linear system of equations

solver Module Reference

Contains necessary routines to solve linear system of equations via LU-decomposition. More...

Functions/Subroutines

- subroutine, public ludecompose (aa, ipiv, info)
  Decomposes the matrix aa into the LU-form. More...

- subroutine, public lusolve (aa, ipiv, bb, info)
  Solves a linear system of equation represented by an LU-decomposed matrix A. More...

Detailed Description

Contains necessary routines to solve linear system of equations via LU-decomposition.

The linear system of equations $A \times x = b$ can be solved via LU-decomposition in two steps. First the matrix $A$ is brought into the LU-decomposed form (routine ludecompose) and then arbitrary number of b-vectors on the right hand side of the equation (routine lusolve())

Function/Subroutine Documentation

- ludecompose()
How to extract the documentation?

Ford

- Set up your projects config file (e.g. `project-file.md`)

With this setting, Ford will be able to process Doxgen-styled comments which are **before** the documented object

Switch it on if you have graphviz installed

- Run Ford

  `ford project-file.md`

- Open the index.html file in the html/ folder with your browser

```ini
project: my project
summary: some summary
author: Me
src_dir: ./
output_dir: html
extensions: f90
predocmark: >
display: public
protected
proc_internals: false
source: false
graph: false
search: false
license: by-sa
warn: true
```
How to extract the documentation?

solver Module

Contains necessary routines to solve linear system of equations via LU-decomposition.

The linear system of equations $A \times x = b$ can be solved via LU-decomposition in two steps. First the matrix $A$ is brought into the LU-decomposed form (routine ludecompose()) and then the linear system of equations is solved for arbitrary number of b-vectors on the right hand side of the equation (routine lusolve())

Subroutines

**public subroutine ludecompose(aa, ipiv, info)**

Decomposes the matrix $aa$ into the LU-form.

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Intent</th>
<th>Optional</th>
<th>Attributes</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(kind=dp), intent(inout)</td>
<td>:: aa(:,:)</td>
<td>Matrix to be decomposed on exit. LU-decomposed matrix on exit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer,</td>
<td>intent(out)</td>
<td>:: ipiv(:)</td>
<td>Contains the row pivots made during the LU-decomposition.</td>
<td></td>
</tr>
<tr>
<td>integer,</td>
<td>intent(out)</td>
<td>:: info</td>
<td>Info flag. Non-zero value indicates error. In that case all other returned quantities are meaningless</td>
<td></td>
</tr>
</tbody>
</table>

solver.fpp  /  solver