Unit testing:
Why I write more test code than regular code

Richard E. Turner (ret26@cam.ac.uk)

May 3rd, 2011
Retractions due to software bugs

A Scientist’s Nightmare: Software Problem Leads to Five Retractions

Until recently, Geoffrey Chang’s career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a 2001 Science paper, which described the structure of a protein called MsbA, isolated from the bacterium Escherichia coli. MsbA belongs to a huge and ancient family of molecules that use energy from adenosine triphosphate to transport molecules across cell membranes. These

Retraction: Measures of Clade Confidence Do Not Correlate with Accuracy of Phylogenetic Trees

Barry G. Hall, Stephen J. Salipante

In PLoS Computational Biology, volume 3, issue 3, doi:10.1371/journal.pcbi.0030051:

As a result of a bug in the Perl script used to compare estimated trees with true trees, the clade confidence measures were sometimes associated with the incorrect clades. The error was detected by the sharp eye of Professor Sarah P. Otto of the University of British Columbia. She noticed a discrepancy between the example tree in Figure 1B and the results reported for the gene nuoK in Table 1, and requested that she be sent all ten nuoK Bayesian trees. She painstakingly did a manual comparison of those trees with the true trees, concluded that for that dataset there was a strong correlation between clade confidence and the probability of a clade being true, and suggested the possibility of a bug in the Perl script. Dr. Otto put in considerable effort, and we want to acknowledge the generosity of that effort.

LETTERS

edited by Etta Kavanagh

Retraction

WE WISH TO RETRACT OUR RESEARCH ARTICLE “STRUCTURE OF MsbA from E. coli: A homolog of the multidrug resistance ATP binding cassette (ABC) transporters” and both of our Reports “Structure of the ABC transporter MsbA in complex with ADP-vanadate and lipopolysaccharide” and “X-ray structure of the EmrE multidrug transporter in complex with a substrate” (1–3).

The recently reported structure of Sav1866 (4) indicated that our MsbA structures (1, 2, 5) were incorrect in both the hand of the structure and the topology. Thus, our biological interpretations based on these inverted models for MsbA are invalid.

An in-house data reduction program introduced a change in sign for anomalous differences. This program, which was not part of a conventional data processing package, converted the anomalous pairs (I+ and I−) to (F− and F+), thereby introducing a sign change. As the diffraction data collected for each set of MsbA crystals and for the EmrE crystals were processed with the same program, the structures reported in (1–3, 5, 6) had the wrong hand.
What is unit testing?

Big picture: scientists have little software experience, useful to learn more.

- **Unit** = the smallest piece of testable code
- Write test suite (collection of tests) in parallel with your code
- External software runs the tests, provides reports and statistics

Why should you unit test?
1) Speed and accuracy: Bugs, tortoises and hares

- Isn’t this going to slow me down?
1) Speed and accuracy: Bugs, tortoises and hares

- Isn’t this going to slow me down?

- **Formalises** testing, **breaks it down** into manageable chunks

- **Discover bugs early** whilst you are familiar the code

- Does not eliminate bugs, but **reduce probability**: \( p_{\text{bug}} = p_{\text{bug code}} \times p_{\text{bug test code}} \)
2) Reusing code: Evolving documentation

- You should **reuse code as much as possible**
- Keep testing it in new situations
- Keep improving it
- Need documentation to come back to code after time
- Unit tests are tied to and evolve with the code
3) Easier to alter code: Optimisation and Refactoring

Agile code development

- write tests
- write simplest version of code
- test and debug until passes
- optimise bottleneck

Retain simple code for documentation
3) Easier to alter code: Optimisation and Refactoring

Agile code development

- write tests
- write simplest version of code
- test and debug until passes
- optimise bottleneck

Supporting towers

- test 1 → unit 1
- test 2 → unit 2
- test 3 → unit 3
- test 4 → unit 4
- test 5 → unit 5

- test 6 → test 7
3) Easier to alter code: Optimisation and Refactoring

Agile code development

- write tests
- write simplest version of code
- test and debug until passes
- optimise bottleneck

Supporting towers

- test 1 → unit 1
- test 2 → unit 2
- test 3 → unit 3*
- test 4 → unit 4*
- test 5 → unit 5
- test 6
- test 7
3) Easier to alter code: Optimisation and Refactoring

**Agile code development**
- write tests
- write simplest version of code
- test and debug until passes
- optimise bottleneck

**Supporting towers**
- test 5 → unit 5
- test 4 → unit 4*
- test 3 → unit 3*
- test 2 → unit 2
- test 1 → unit 1
- test 6
- test 7
3) Easier to alter code: Optimisation and Refactoring

**Agile code development**

- write tests
- write simplest version of code
- test and debug until passes
- optimise bottleneck

**Supporting towers**

- test 1 → unit 1
- test 2 → unit 2
- test 3 → unit 3*
- test 4 → unit 4*
- test 5 → unit 5
- test 6 →
- test 7 →
Unit testing in Matlab: xUnit

- Runs whole testing suites or parts of suites automatically
- Errors obvious
- Provides functions to aid test writing (comparing values within a tolerance)
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input

```matlab
1 function test_suite = test_inv
2     initTestSuite;
3     % Demonstration tests of inv.m

4 function test2D
5     % Test expected value for simple 2D matrix
6     A = [1, 2; 3, 4];
7     invA = [-2, 1; 1.5, -0.5];
8     assertElementsAlmostEqual(invA, inv(A), ...
9         'absolute', 1e-6, 0)
```
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute \textbf{expected output} for
  a simple input

- \textbf{boundary analysis}
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input
  
- **boundary analysis**

9 function testScalar

10  % Tests with known scalar input

11  A = pi;  % alternatives are A=[] or A=inf

12  invA = 1/pi;

13  assertElementsAlmostEqual(invA,inv(A),... 'absolute',1e-6,0)
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

• compute expected output for a simple input

• boundary analysis

• compare to alternative computation (e.g. numerical)
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input

- **boundary analysis**

- compare to **alternative computation** (e.g. numerical)

14 function testBackSlash

15  % Test an alternative computation method

16  randn(’seed’,1);  % fix seed to known value
17  A = randn(4);
18  x = randn(4,1);

19  assertElementsAlmostEqual(A\x,inv(A)*x,...
     ’absolute’,1e-6,0)
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input
  
- **boundary analysis**
  
- compare to **alternative computation** (e.g. numerical)
  
- random numbers: coverage (save, display or fix seed)

```matlab
function testBackSlash

% Test an alternative computation method
randn('seed',1); % fix seed to known value
A = randn(4);
x = randn(4,1);

assertElementsAlmostEqual(A\x,inv(A)*x,... 'absolute',1e-6,0)
```
What form should the tests take?

Principle: maximal input coverage through minimal subset of test cases

- compute **expected output** for a simple input

- **boundary analysis**

- compare to **alternative computation** (e.g. numerical)

- random numbers: coverage (save, display or fix seed)

Make it clear what’s being tested, verify 1 condition per test, toy examples
Running Tests

>> runtests
Test suite: /home/rich/Documents/Talks/UnitTesting/Code
03-May-2011 09:30:26

Starting test run with 3 test cases.
...
PASSED in 0.005 seconds.
Running Tests

>> runtests
Test suite: /home/rich/Documents/Talks/UnitTesting/Code
03-May-2011 09:26:26

Starting test run with 3 test cases.
.F.
FAILED in 0.009 seconds.

===== Test Case Failure =====
Location: /home/rich/Documents/Talks/UnitTesting/Code/test_inv.m
Name: testScalar

/home/rich/Documents/Talks/UnitTesting/Code/test_inv.m at line 13

Input elements are not all equal within absolute tolerance: 1e-06

First input:
  -1.2901

Second input:
  0.3183
Summary

• Why you should use unit testing
  – Faster to write bug free code
  – Easier to reuse code (evolving documentation)
  – Easier to change code (optimisation and refactoring)

• xUnit and docTest provide useful tools for organising, running and interrogating unit tests.
Where to find out more information

• Retractions due to software bugs:
  http://www.the-scientist.com/news/display/53289/

• General introduction to unit testing:
  http://en.wikipedia.org/wikui/Unit_testing

• Article on unit testing for scientific computing in matlab:
  http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05337644

• xUnit framework for matlab:
  www.mathworks.com/matlabcentral/fileexchange/22846

• docTest framework for matlab:
  http://bitbucket.org/tgs/doctest-for-matlab/src