Testing for (Code) Trustworthiness in Scientific Software

Daniel Hook[1] Diane Kelly[2]

[1]Queen's University

[2]The Royal Military College of Canada

The SECSE Workshop at ICSE 2009

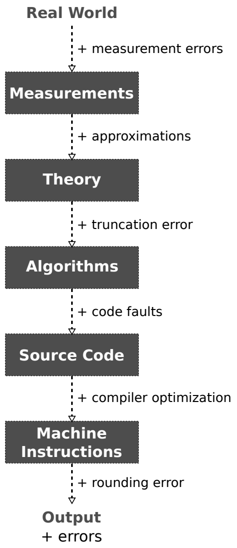
How should scientists mitigate code fault related risks?

Errors and Faults

Error is a measure of the difference between a measured or calculated value of a quantity and what is considered to be its actual value.

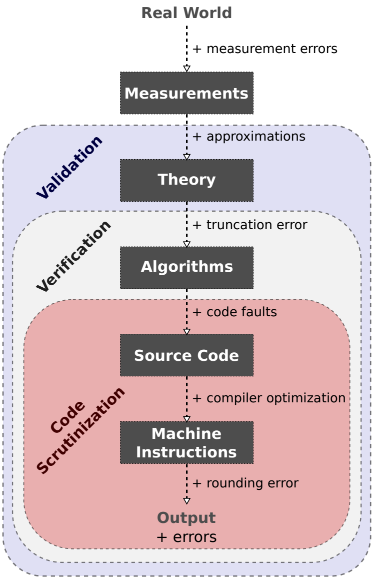
Code faults are mistakes made when abstract algorithms are implemented in code.

Faults are not errors, but they frequently lead to errors.



Scientific software development involves a number of model refinements in which errors may be introduced.

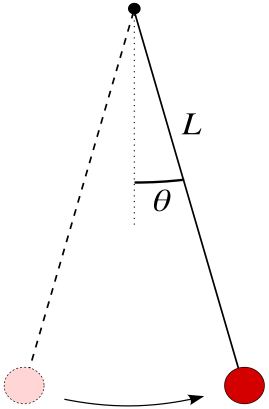
Program outputs include the accumulation of all these errors.



Validation and verification activities are applied in an attempt to identify or bound these errors.

In addition to validations and verifications, we also suggest that computational scientists should conduct *code scrutinizations*.

The Pendulum Example



Consider the development of a code that is used to model the swinging of a pendulum.

Pendulum Physics

We simplify the problem by assuming the pendulum can be modeled as a point mass suspended from a massless string. We will also assume that there are no damping forces other than some constant friction at the pivot.

The net force on this pendulum can then be calculated using

$$F_{net} = mg \sin\theta - f$$

Applying Newton's 2nd Law yields a differential equation:

$$\frac{d^2\theta}{dt^2} + C_1 \sin\theta + C_2 = 0$$

Pendulum Equations

Note that $\sin\theta \approx \theta$ for small angles so we can simplify the ODE to

$$\frac{d^2\theta}{dt^2} + C_1\theta + C_2 = 0$$

Using certain boundary conditions, the solution to this ODE takes the form

$$\theta = A - B\sin(2\pi\omega t)$$

where A, B, and ω are constants calculated from the physical properties of the pendulum.

Pendulum Algorithms

A straightforward algorithm can be used to evaluate this equation. A, B, ω, and t are taken as inputs and used in the calculation of the output θ.

Note that $\sin(2\pi\omega t)$ is usually evaluated using a truncated series. For example, if one used four terms from a Taylor series the following calculation would be used:

$$\theta = A - B(2\pi\omega t - \frac{(2\pi\omega t)^3}{3!} + \frac{(2\pi\omega t)^5}{5!} - \frac{(2\pi\omega t)^7}{7!})$$

Pendulum Code

Coding this algorithm into a MATLAB function is fairly straightforward. The following code can be used:

```
function theta = pendulum(A, B, omega, t)
    theta = A - B*sin(2*pi*omega*t);
```

Before execution, this code will compiled into some form of machine code that will be executed on finite precision hardware.

Pendulum Errors

Acknowledged error in the outputs of this program result from:

- Errors in the measurement of physical properties (e.g., mass).
- Simplifying assumptions (e.g., ignoring the mass of the string).
- Small-angle approximation used to simplify the ODE.
- Truncation error resulting from the finite number of terms used to evaluate the sin function.
- Round-off errors from the finite precision calculations.

In addition to these errors, *unacknowledged errors*—e.g., code faults and other conceptual or concrete mistakes—will also reduce output accuracy.

Scientific software testing is complicated, at a fundamental level, by two problems.

1) *The Oracle Problem*
In general, scientific software testers only have access to approximate and/or limited oracles. This limits a tester's ability to detect failures and restricts the set of tests that they can apply.

2) *The Tolerance Problem*
If an output exhibits acknowledged error then a tester cannot conclusively determine if that output is free from unacknowledged error: i.e., it can never be said that the output is "correct." This complicates output evaluation and means that many test techniques cannot be (naively) applied.

In our recent work, we have shown that a generalized version of mutation testing allows us to circumvent the oracle and tolerance problems so that we can begin to effectively explore testing strategies.

Mutation Testing Basics

Mutations are syntactic changes to program statements.

Mutation operators are rules that are applied to program code to generate mutations.

A *mutation target* P_0 is a program that is to be mutated.

A *mutant* P_m is a program that is syntactically identical to some P_0 except that one of its statements contains a mutation.

A test set T for some target P_0 can be improved by *traditional mutation testing* as follows:

1. Mutants of P_0 are generated by applying some mutation operators.
2. P_0 and its mutants are run using T. If the outputs from some mutant P_m are *strictly equal* to the outputs from P_0 then P_m is a *survivor*; otherwise, P_m is said to have been *killed*.
3. If there are any non-equivalent survivors then T is augmented in an attempt to kill these survivors.
4. Steps 2 and 3 may be repeated using augmented versions of T until the tester is satisfied with the results.

A Flaw in Traditional Mutation Testing

Traditional mutation testing should not be used to study scientific software test sets. To understand why, consider the 2nd line from the pendulum code:

P_0: `theta = A - B*sin(2*pi*omega*t);`

This line can be mutated to produce

P_m: `theta = A - B*sin(1*pi*omega*t);`

For `A` = `B` = `omega` = 1.

P_0: `theta = 1 - sin(2*pi*t);`
P_m: `theta = 1 - sin(1*pi*t);`

Using `t` = 1:

P_0: `theta` $\approx 1 + 2 \times 10^{-16}$
P_m: `theta` $\approx 1 - 1 \times 10^{-16}$
$\Delta_1 \approx 3 \times 10^{-16}$

Using `t` = 0.7:

P_0: `theta` ≈ 1.95
P_m: `theta` ≈ 0.19
$\Delta_{0.7} \approx 1.76$

The fact that `t` = 1 kills the mutant is dangerous for two reasons:

1. The difference between the two results is almost negligible: using `t` = 0.7 makes the mutation is 10^{15} times more obvious.
2. The mutant is only killed because of round-off and truncation errors: P_0 and P_m would both exhibit 0 error if exact math were used.

Mutation Sensitivity Testing

MST is a generalized form of traditional mutation testing. Boolean evaluation of output equivalence is replaced by a sensitivity measure—e.g., relative error—and mutant killing is replaced with mutant detection.

Relative error γ is given by

$$\gamma(P_m, t) = \frac{|P_m(t) - P_0(t)|}{|P_0(t)|}$$

where t is a test.

A *detected mutant* is a mutant that has exhibited an error greater than a specified *detection boundary* γ_d.

Given a set of mutants P_M and a detection boundary γ_d, the *detection score* of a test set is the percentage of mutants in P_M that it detects.

To explore and demonstrate some potential uses for MST, a series of experiments were conducted. 8 mutation targets were mutated—using 4 classes of mutation operators—to produce 1492 mutants; these mutants were then tested using 1155 tests.

The following eight targets were used:

- `binSearch`: uses a binary search to search through a vector.
- `gaussQuad`: uses Gaussian quadrature to integrate a function.
- `GEPiv`: uses Gaussian elimination to solve a system of equations.
- `nwtsqrt`: uses Newton's method to find a number's square root.
- `odeRK4`: uses the 4th-order Runge-Kutta method to solve a single, first-order ODE.
- `powerit`: uses the power method to find the largest eigenvalue of a matrix.
- `simpson`: uses Simpson's rule to integrate a function.
- `sphereFnet`: computes the forces on a sphere falling through a fluid.

These are small functions. The smallest contains 6 statements while the largest contains 24. However, they are all, with the exception of `sphereFnet`, meant to be used as components or units in larger scientific codes, and they all contain code statements of the type that are essential to numerical analysis.

1492 mutants were generated by applying code mutation modules from the MATmute package[1]. These modules were configured to use four classes of mutation operators:

- *Statement Deletion*: deletes a statement.
- *Conditional Negation*: logically negates the conditional part of an `if` or `while` statement.
- *Constant Replacement*: replaces a hard-coded constant C with 0, $-C$, $C-1$, $C+1$, $0.9C$, and $1.1C$.
- *Operator Replacement*: replaces an arithmetic (`+`, `-`, `*`, `/`, `\`, `^`), relational (`<`, `<=`, `>`, `>=`, `==`, `~=`), or logical (`&&` and `||`) operator with another operator of the same class.

[1] Available at `matmute.sourceforge.net`.

Popperian tests are designed to encourage novel computations that push the boundaries of a program in an attempt to falsify its code, i.e., they are designed to expose faults analogously to the way that some experiments are designed to expose flaws in a theory.

Pseudo-random tests are randomly selected from intervals or lists of reasonable inputs. In the thesis experiments, only numerical inputs were randomly selected—Popperian inputs were used for non-numerical inputs.

1155 tests were selected in total: 105 Popperian tests (T_{Pop}) and 1050 pseudo-random tests (T_{rnd}). T_{cmb} indicates the combination of these two sets. Tests were executed and analyzed in MATLAB by using MATLAB functions included in the MATmute package.

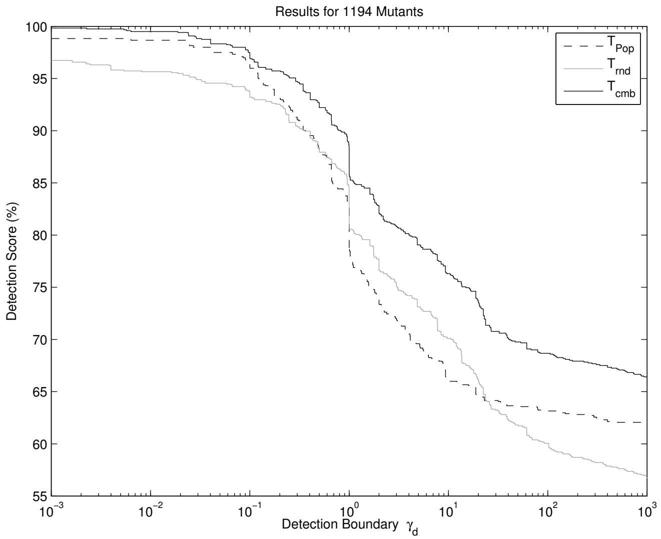


Figure: Detection scores accumulated from all targets.

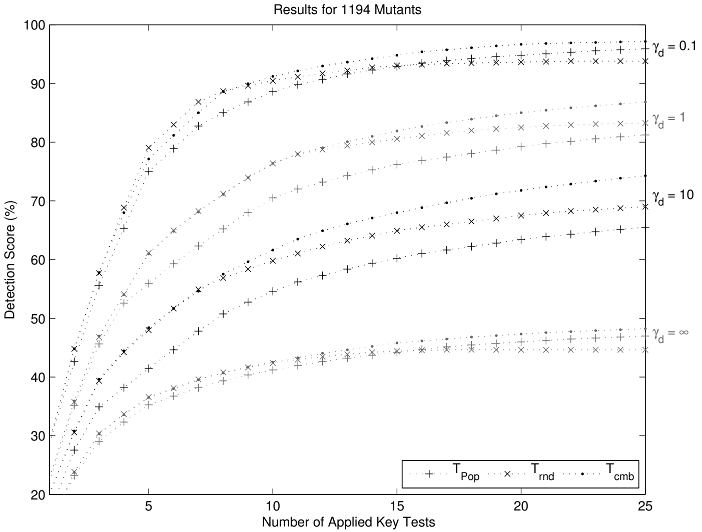


Figure: Key detector efficiency.

Our results show that *code falsification can be effectively used to build trust in scientific codes*.

To elaborate: a small number of “key” tests can detect many faults if sufficiently accurate oracles are available. Therefore, a code that successfully executes a set of these key tests is deserving of some trust. Note however, that this requires two conditions to be met:

- The oracle and tolerance problems must be acknowledged, and uncertainties must be understood.
- Testers must be provided with testing techniques that will allow them to choose effective tests.

A Final Question

In the *Proceedings of Foundations 2002*, Oberkampf et al. write, "There are no straightforward methods for estimating, bounding, or ordering the contributions of unacknowledged errors."

How can testing researchers help scientists deal with unacknowledged errors in their software?

(`matmute.sourceforge.net`)

Table: Counts of categorized mutants.

Target	Not Viable	Not Revealed	Revealed	Terminal	Total
binSearch	17	5	13	47	82
gaussQuad	60	4	120	82	266
GEPiv	31	17	39	138	225
nwtsqrt	15	2	6	42	65
odeRK4	61	1	169	193	424
powerit	15	2	9	6	32
simpson	36	12	106	27	181
sphereFnet	13	7	151	46	217
Total	248	50	613	581	1492

Table: Counts of minimally equivalent tests and key detectors.

Target	$\lvert T_{cmb} \rvert$	$\lvert T_Q \rvert$	$\lvert T_K \rvert$ at γ_d of			
			0.1	1.0	10.0	Inf
binSearch	121	6	3	4	4	4
gaussQuad	198	16	2	3	5	2
GEPiv	44	12	5	5	6	5
nwtsqrt	55	3	2	2	2	2
odeRK4	264	21	2	5	8	5
powerit	66	3	2	2	2	2
simpson	198	16	2	4	4	3
sphereFnet	209	19	5	8	6	7
Total	1155	96	23	33	37	30

NB: T_Q is a minimal subset of T_{cmb} that yields identical Γ results to the full set. T_K is a minimal set of tests that detects all mutants detected by T_{cmb} for a given detection boundary.