Mutant Accuracy Testing for Assessing the Implementation of Numerical Algorithms*

Ruining (Ray) Wu[1,2] and Ian M. Mitchell[1][0000−0001−7053−441X]

[1] Department of Computer Science,
The University of British Columbia,
Vancouver, Canada
[2] Department of Computer Science,
University of Toronto,
Toronto, Canada
rwu@cs.toronto.edu, ian.mitchell@ubc.ca
http://www.cs.toronto.edu/~rwu/
http://www.cs.ubc.ca/~mitchell

Abstract. Despite their widespread use, implementations of numerical computing algorithms are generally tested manually with fuzzily defined thresholds determining success or failure. Modern software testing methods, such as automated regression testing, are difficult to apply because both test oracles and algorithm output are approximate. Based on the observation that high accuracy numerical algorithms appear to be fragile by design to errors in their parameters, we propose to compare the error of target implementations to mutated versions of themselves with the expectation that the mutants will suffer degraded accuracy. We test the idea on MATLAB implementations of some basic numerical algorithms, and find that most mutants are worse while the few which are better show a distinctive pattern of mutation.

Keywords: mutation testing · numerical algorithms · oracle problem · hypothesis testing

1 Introduction

Testing is a well-refined art in most areas of computing; for example, automated regression and mutation testing are used to ensure that intended code changes do not introduce bugs and that test suites provide sufficient coverage to detect artificially introduced bugs respectively. But much of this infrastructure depends on knowing the correct output for each test; in other words, the test oracle is exact. Although accurate numerical calculation has been a known challenge since before modern computers [3] and has been the subject of serious academic research for more than seventy years (for example, [14]), the state of the art for testing

* This work was supported by National Science and Engineering Research Council of Canada (NSERC) Discovery Grant #298211 and an Undergraduate Student Research Award (USRA).

implementations of numerical algorithms is much more manual [12]. A key reason is that approximation is everywhere in numerical computing. Not only are test oracles approximate, but implementations will generate output containing error when coded correctly, and different implementations may generate different but still correct output. Even the most rigorous categories of testing, such as convergence rate studies, often involve heuristically enforced thresholds; for example, is a fourth order accurate Runge-Kutta scheme for numerically solving initial value ordinary differential equation problems correctly implemented if the measured convergence rate is 3.95? What about 3.89?

Our goal with the technique described in this paper was to develop a testing approach for numerical implementations which:

- Eliminated the need for threshold checks to determine correctness, or at least made the outcome less sensitive to the precise value of that threshold.
- Could be automated to encourage more widespread use of regression testing.

To test a target implementation, we propose to generate a collection of mutant implementations and compare their accuracy against the target, with the hypothesis that the accuracy of mutants of correct targets will degrade significantly, while those of incorrect targets will degrade much less.

We test this idea on MATLAB implementations—both correct and with artificially introduced bugs—of three basic numerical algorithms. Most mutants show degraded accuracy, most buggy implementations show less fragility, and results for a given algorithm do not depend strongly on our choice of threshold. Unfortunately, some mutants have better accuracy, some buggy implementations are more fragile, and different algorithms show quantitatively different responses. Because of this variability, it is unclear how the technique might be automated; however, manual inspection of the mutants which demonstrated better accuracy was enlightening and may prove useful for evaluating the quality of an implementation.

It should be noted that we focus on software designed to produce numerical approximations to mathematical problems. Software for symbolic mathematics (such as the SageMath system, Mathematica, or Maple) is also in widespread use; however, we believe that such implementations are more amenable to traditional software testing approaches because approximation plays a smaller role in typical symbolic mathematics tasks.

2 Background

Numerically approximating the solution of a scientific or other real-world problem is subject to many types of error, so we first categorize these errors and identify which we consider in this paper. We then discuss the limitations of oracles available for numerical computing problems and the types of testing often performed on numerical implementations. We conclude the section with a brief discussion of mutation testing, which is typically used for enterprise software but which has at least once been repurposed for numerical implementations, albeit in a manner different than what we propose.

2.1 Sources of Error in Numerical Computations

Because computer hardware implements only arithmetic and perhaps a few elementary functions, most mathematical tasks on computers involve approximations of various kinds. Demonstrating the correctness of a numerical approximation can be divided into two tasks [12]:

- Validation: Determining whether the mathematical models are sufficiently accurate; in other words, "did you solve the right equations?"
- Verification: Determining whether the implementation produces an accurate approximation of the solution; in other words, "did you solve the equations right?"

Here we will explore only verification, but it is worth keeping in mind that because most real problems involve modeling error, driving implementation error all the way to zero would be wasted effort.

The field of numerical analysis has for decades focused on the construction of algorithms for approximating various mathematical models which are accurate, in the sense that the difference (or error) between the true solution of the equations and the computed approximation are small. Accuracy can be broken into two components [5]. Problem conditioning measures the effect on the solution of perturbing the problem's parameters. Computational error measures the effect on the solution of errors made during computation. We again narrow our focus in this paper to questions of the latter, but we will assume that our target problems are well-conditioned and the primary sources of error are computational.

Finally, we will distinguish two different categories of computational error. Roundoff error arises because we use floating point rather than real numbers to represent values, and at regular intervals during computation results must be rounded to some finite precision. Truncation (or discretization) error arises when we terminate what is mathematically an infinite object or process; for example, represent a continuous function by its value at finitely many points, truncate a Taylor series after finitely many terms, or stop an infinite recurrence after finitely many iterations. While finite precision arithmetic could be considered a form of truncation, we distinguish these two cases because in typical computing environments the user has very limited ability to choose the precision of the arithmetic (for example, choosing either single or double precision floating point numbers), while truncation error can often be tuned more finely with a user controlled parameter (for example, step size in ODE solvers).

2.2 Oracles

An oracle [7] is a mechanism by which we can determine the correctness of a test case. When an exact oracle exists, testing computer programs is relatively straightforward: Run the test case through the program and compare the computed result to the oracle's answer. While it is impractical in most cases to test every possible input, with each correct test case our confidence in the program increases.

Two key challenges arise in testing numerical algorithms. First, an exact oracle may not be available. The class of test cases for which analytic solutions are known can be quite restricted, so testing only within this class may not properly exercise the implementation. In some cases, including the ones considered in section 4, we can work around this constraint using the method of manufactured solutions (MMS) (for example, see [10,12,13]). In MMS, we start from an analytic solution and work backwards to define a problem with that solution; for example, to test a one dimensional quadrature (numerical integration) routine, we start with an arbitrary differentiable function $f(x)$ and suitable endpoints $a, b \in \mathbb{R}$. The test problem is defined as $\int_a^b f'(x)\,dx$, where $f'(x)$ is easily found by analytic differentiation, and the analytic solution is given by $f(b) - f(a)$. However, even if an analytic solution is known, its constituent elements may not be finitely computable and/or representable on the computer; for example, if $f(x)$ in the example above includes a term involving $\sin(x)$. Consequently, we should expect that only approximate oracles are available in practice.

Second, not only do we expect an approximate answer from our implementation, but we typically do not have a precise a priori bound at the time algorithms are designed, implemented and tested for either the expected error in the approximation or the desired accuracy.[3] The former arises because rigorous quantitative analysis of even the computational sources of error is complicated enough that it is rarely done, and the latter because non-computational sources of error (such as modeling and propagated data error) are often poorly quantified, leading to decreased certainty about the level of tolerable computational error.

In summary, for much of the numerical computing domain we cannot use the common approach to testing software implementations which asserts the presence of a bug if the program's output on a test case does not exactly match that of the oracle. Consequently, the practice of testing numerical algorithms has diverged from the rest of the testing community.

2.3 Testing Numerical Algorithms

A number of approaches have been designed to manage or analyze computational error. The first is simply to ensure that the algorithm chosen is stable; in other words, that errors made in early steps of the computation (including in the representation of the input data) do not grow dramatically in later steps [1,5]. More quantitatively, interval arithmetic can be used to bound computational sources of error; for example, VNODE-LP is a software package for computing solutions to ordinary differential equations which produces intervals within which the true solution is known to lie [11]. An alternative quantitative approach focused on estimating the effect of roundoff error is the Contrôle et Estimation STochastic des Arrondis de Calculs (CESTAC) method implemented in the CADNA library [9],

[3] These quantities may be known when it comes time to solve a particular problem, but outside of introductory numerical analysis courses most problems are solved by calling a library routine or legacy implementation; consequently, those who design, implement and test such routines should not assume knowledge of these values.

which essentially repeats the calculation several times under different rounding regimes and thereby provides a probabilistic estimate of both the approximate solution and the magnitude of the roundoff error. However, approaches like these are intended to quantify the computational error incurred during a calculation, not to demonstrate correct implementation: Even if the error between the approximation and oracle for all test cases lies within the computed bounds, there may still be bugs.

In contrast, there has been work in the domain of automatic theorem proving to formally demonstrate implementation correctness; for example, Gappa is a tool which makes it easier to construct formal proof obligations which would demonstrate that for a C implementation of a floating point algorithm the error would lie within a specified interval; these proof obligations can then be passed to an automated proof checker for validation [2]. This level of rigour is wonderful when it can be accomplished, but even Gappa currently requires a careful manual rewriting of the underlying C code and heuristically chosen hints to guide proof procedure, while direct use of theorem provers is beyond the capabilities of most programmers.

While techniques such as those described above bring a pleasing level of quantitative rigour to implementation analysis, in this paper we will focus on testing a much more common pattern encountered in numerical software: The algorithm is designed so that truncation error converges asymptotically toward zero as some tuning parameter is varied, and then implemented in a floating point precision sufficiently high that roundoff error will (hopefully) be negligible compared to other sources of error. For concreteness, we will call the tuning parameter which controls truncation error "h" and assume that the truncation error decreases as h does. In many cases the truncation error can be theoretically bounded as an explicit function of the form $\mathcal{O}(h^p)$ for some $p > 0$, in which case we call p the "order of accuracy" of the approximation algorithm. Four levels of testing for such numerical algorithms have been defined [13]; in order of increasing rigor they are:

1. Expert judgment: The algorithm's output approximation is given to an expert who is asked to determine whether it is sufficiently correct.
2. Error quantification: The error between the approximation and the oracle is computed, and then an expert is asked to determine whether it is sufficiently small.
3. Convergence: The algorithm is run with a sequence of decreasing values of h, and it is checked that the error is decreasing. This level of testing requires the program to converge, but not at a particular order of accuracy.
4. Order of accuracy: As with convergence, but the rate of convergence is checked against the theoretically derived order of accuracy for the algorithm.

While the strongest in this hierarchy, the order of accuracy approach is still surprisingly dependent on expert judgment in practice: The implementation on a test case is run for a series of decreasing values of h, the approximate oracle is used to compute an approximate error for each h, and then the logarithm of the ratio of the errors for two (typically consecutive) values of h is used to estimate

the experimental order of accuracy. Assuming that more than two values of h are used, multiple estimates are produced; some can be discarded as being outside the regime in which the asymptotic truncation error analysis applies, but the remainder are to be compared to the theoretical order of accuracy. How closely must they match? Consider these two prescriptions:

- From [10, p.30]: "In general, one should not expect the trend in the observed order-of-accuracy to match the theoretical order-of-accuracy to more than two or three significant figures..."
- From [12, p.195]: "Note that only for the simplest scientific computing cases (e.g., linear elliptic problems) will the observed order of accuracy match the formal order to more than approximately two significant figures during a successful order verification test."

Beyond the fuzziness of such a procedure (which order of accuracy estimates are compared, how should "significant figures" be mapped into a quantitative threshold), this process is not easily automated; consequently, regression testing of even minor code modifications becomes labour intensive and is often skipped.

In this paper we are not advocating that this approach to testing be abandoned; in fact, we have long found it highly efficient for identifying and correcting bugs during initial design and implementation. Instead, we are seeking to add a subsequent layer of testing once an implementation has satisfied order of accuracy convergence tests such as those described above. We will assume a collection of test cases with approximate oracles such as would be used in these convergence tests, and our goal is not necessarily more rigorous testing, but rather automating the testing so that regression approaches can be easily applied to subsequent code modifications.

2.4 Mutation Testing

Mutation Testing (MT) [8] is a testing technique designed to verify the strength of a test set for some "target" source code once that target passes all of the tests (in the sense that its outputs match the test oracles' output). Hundreds or thousands of "mutant" versions of the target are produced by a source code generator which systematically introduces source code modifications designed to simulate bugs that a programmer might accidentally introduce. The mutants are then run against the same test set, and those whose outputs fail to match the oracles' are "killed." Any mutants which survive represent potential bugs that would not be detected by the test set. A test set could then be strengthened by examining any surviving mutants and designing test cases that would kill them.

MT cannot be directly applied to numerical algorithms: We do not have exact oracles against which to compare for equality, and the standard techniques for testing numerical algorithms described in section 2.3 are not automated; hence it is infeasible to scale them to test hundreds or thousands of mutants. A necessary step in the application of MT to numerical algorithms is therefore a method of automatically evaluating the correctness of a mutant on a test case.

At first glance, it might appear that Mutation Sensitivity Testing (MST) [6] would satisfy this objective. Instead of using exact comparisons, the error of the

mutants' output relative to the oracle is compared against a specified tolerance, and the contribution of the paper is an exploration of how the choice of this tolerance and the type of test cases (random or designed) affected the fraction of mutants which were killed, with the conclusions that:

- A small number of tests with low tolerance is more effective than a large number of tests with high tolerance in killing mutants. Unfortunately, higher accuracy oracles are needed for lower tolerance tests.
- Random and designed tests should be used together for maximum effectiveness.

However, the authors of [6] did not intend for their approach to be used directly to verify numerical code; in fact, they considered the target code itself to be the oracle, and the goal of their study (like traditional MT) was guidance on how to design effective test suites.

In contrast, our goal is a test criterion which does not rely (or at least is only weakly dependent on) the choice of parameters. We note that in pursuit of this goal, we will make a stronger assumption than [6] that we have access to an approximate but reasonably accurate oracle separate from the target code.

3 Mutant Accuracy Testing

While MT is traditionally used to evaluate the strength of a test suite, here we propose to use it to evaluate the correctness of a numerical implementation. We believe that code mutation might yield a useful measure of correctness for numerical algorithms with high orders of accuracy (anything with order of accuracy $p > 1$) because such algorithms appear to be fragile to perturbation by design: They achieve their high order of accuracy by the use of carefully chosen parameters which combine to cancel the lower order terms in the truncation error. If the code mutation process introduces changes to these parameters, then the lower order terms should reappear and cause the error in the output approximation to grow significantly.

3.1 Mutant Generation

MATmute [6] is a freely available mutant generator for MATLAB code. Given a target MATLAB function or file, MATmute generates a collection of mutants by systematically applying *mutation operations* on source code. The operations that MATmute performs are the following:

- Statement deletion: A statement is commented out.
- Branch negation: A branch condition is negated, forcing the opposite decision.
- Constant replacement: A hard-coded constant is replaced with another.
- Operator replacement: A mathematical operator is replaced by another.
- Assignment perturbation: The right hand side of an assignment statement is multiplied by a constant before the assignment is completed.

For each mutant MATmute generates both a mutant source code file and a summary of what mutation operation was applied.

For a given test run, any mutants which do not compile or otherwise fail to generate an intelligible result are removed from consideration. The error for the remaining "viable" mutants is compared to that of the target, and any which are bitwise identical are also removed. These "equivalent" mutants are fairly common because the difference between the target and a given mutant may be in code that is not executed for a given test case; consequently, they are not diagnostically useful for that test case. The remaining viable but nonequivalent mutants and their results are then used to evaluate the target code.

3.2 Mutant Evaluation

We treat the computed error of the approximation produced by a code (either target or mutant) on a test case as an observation of a random process. For a given mutant, we define the hypotheses:

- Null hypothesis: The mutant does not produce approximations with larger error than the target.
- Alternative hypothesis: The mutant produces approximations with larger error than the target.

For a given mutant, we collect all of the test cases for which it was viable and non-equivalent to the target code. We can pair the observation of the mutant's error for each of these test cases with the observation of the target's error for the same test case. We apply a one-sided sign test [15] to the collection of observation pairs to determine the probability of seeing the observations given that the null hypothesis holds; in other words, the p-value. We choose the sign test rather than alternatives like the Wilcoxon signed-rank or paired t-test because it makes the fewest assumptions about the underlying distributions.

Based on our belief that the accuracy of numerical algorithms is fragile, we expect that most mutants will produce approximations with larger error. Therefore, we expect to collect observations that are highly unlikely under the null hypothesis, and the p-value will be low. We can choose a threshold p-value p_k and declare any mutant i with a lower p-value $p_i < p_k$ to be killed.

4 Experiments

We describe the conditions under which we tested our approach.

4.1 Reference Numerical Implementations

We chose some frequently used higher order accurate algorithms for common basic numerical analysis tasks:

- Quadrature (numerical integration) in one dimension: Composite Simpson's rule. Panels of equal size are chosen to cover the interval of integration and h controls the size of these panels.

Table 1. Sample Implementation Statistics. All algorithms were implemented as MATLABm-files.

Algorithm Name	Number of		
	Lines	Test Cases	Buggy Versions
Composite Simpson's	4	3	9
Cubic spline	32	3	9
RK4	25	3	22

- Interpolation in one dimension: Complete cubic spline. Abscissae are chosen equally spaced between the end points of the interpolation, and h controls the space between abscissae.
- Initial value problem for ordinary differential equation: Runge-Kutta fourth order (RK4). Constant stepsize is used and h is the step size.

Additional details of the algorithms are included in appendix A. Coincidentally, the chosen algorithms all have order of accuracy $p = 4$ (so their error should be bounded by $\mathcal{O}(h^4)$), but our proposed approach does not require knowledge of the order of accuracy and hence it should apply to any convergent scheme.

Reference implementations of each algorithm were coded by the first author in MATLAB based on descriptions in [1], and then reviewed for correctness by the second author. Table 1 provides some quantitative measures of the code.

Although versions of all of these algorithms are available in the standard MATLAB libraries, the MATLAB versions are much more general and hence include an enormous amount of code to handle different input and output cases. This kind of data manipulation code is much more amenable to traditional testing strategies and we wanted to focus on testing the core numerical calculation; however, due to limitations with MATmute we could not easily generate mutations only in a subset of lines in a routine. Therefore, we chose to code our own versions which contained only the core numerical calculation.

4.2 Test Cases

MMS was used to design a collection of test cases for each algorithm. For each test case and value of the tuning parameter h, a scalar measure of error can be evaluated:

- Composite Simpson's: Absolute error of the integral.
- Cubic spline: Maximum absolute error of the approximation at the test points.
- RK4: Absolute error at the end of the time interval.

More details can be found in appendix A.

To further build confidence in the implementations, the convergence rate of each algorithm on each test case was qualitatively confirmed experimentally as described in section 2.3.

Table 2. Sample Algorithm Mutation Statistics (reference implementations)

Algorithm Name	Number of Mutants		
	Total	Viable	Nonequivalent
Composite Simpson's	184	122	172
Cubic spline	677	213	287
RK4	531	127	218

4.3 Buggy Versions

When proposing a test criterion for code, we must demonstrate both that it passes correct code and that it fails incorrect or buggy code. For this preliminary exploration, we created buggy versions of our reference implementations by hand based on our experience with incorrect code generated by students in introductory numerical analysis classes, examination of mutants of our reference implementations which did and did not pass the hypothesis test, and careful design to elicit certain error behaviours. We considered only buggy versions which would still be convergent, although not necessarily with the designed order of accuracy, on the basis that non-convergent implementations are easily diagnosed as buggy.

5 Results

Table 2 provides some statistics about the number and type of mutants produced by MATmute for each of the reference implementations. The number of nonequivalent (and in a few cases even viable) mutants does vary slightly depending on the test case. The statistics when using the buggy versions of the algorithms as target code are similar.

Figure 1 shows the distribution of p-values for mutants of the reference composite Simpson's target. Other targets produced similar distributions. The red curve in figure 2 provides similar information in a different form: Threshold p-values p_k against the fraction of mutants which will survive ($p_i \geq p_k$). We observe that the survival rate is dropping slightly between $p_k = 0.2$ and $p_k = 0.4$, but is fairly flat across a wide range of thresholds. The remaining black curves in figure 2 show the corresponding survival rates for mutants of the buggy Simpson's targets. We observe that three of the buggy targets are roughly as fragile as the reference implementation, in the sense that their mutants are unlikely to survive, and two of the buggy targets are even more fragile.

Figures 3 and 4 show the corresponding survival rates for the cubic spline and RK4 implementations.

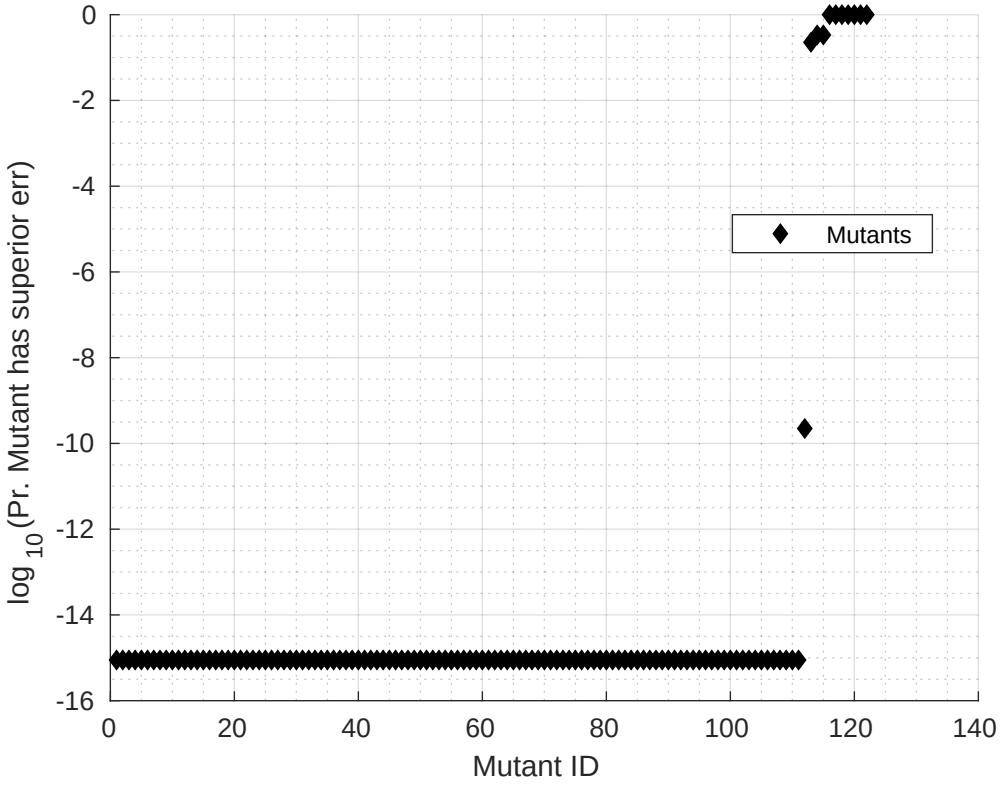

Fig. 1. Distribution of p-values for viable, non-equivalent mutants of the reference implementation of Composite Simpson's. Mutants are ordered by their p-value from smallest on the left to largest on the right. Most mutants have p-values very near zero, and of the rest most are near one. Similar distributions were seen for other target codes (reference and buggy).

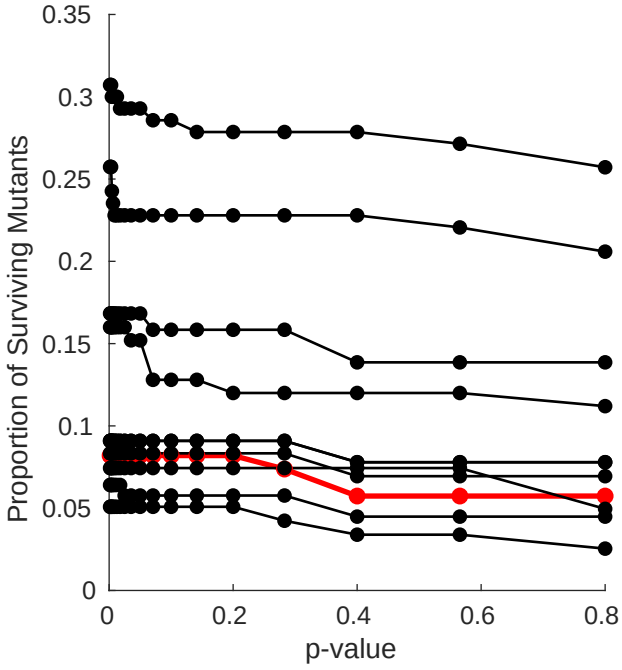

Fig. 2. Survival rate for mutants of reference (red) and buggy (black) versions of composite Simpson's as a function of p-value threshold p_k.

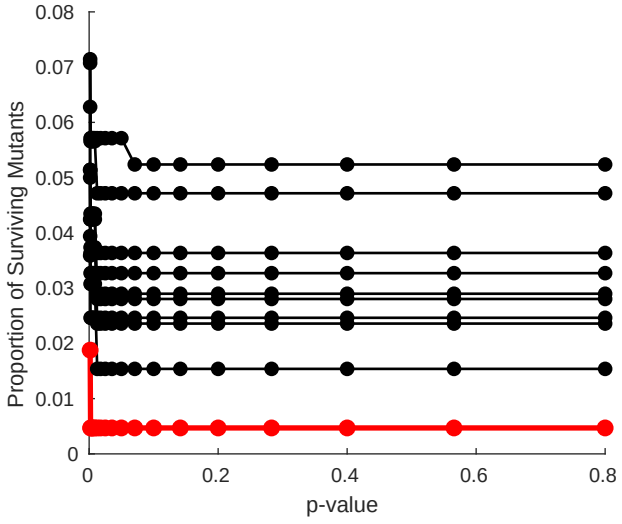

Fig. 3. Survival rate for mutants of reference (red) and buggy (black) versions of cubic spline as a function of p-value threshold p_k.

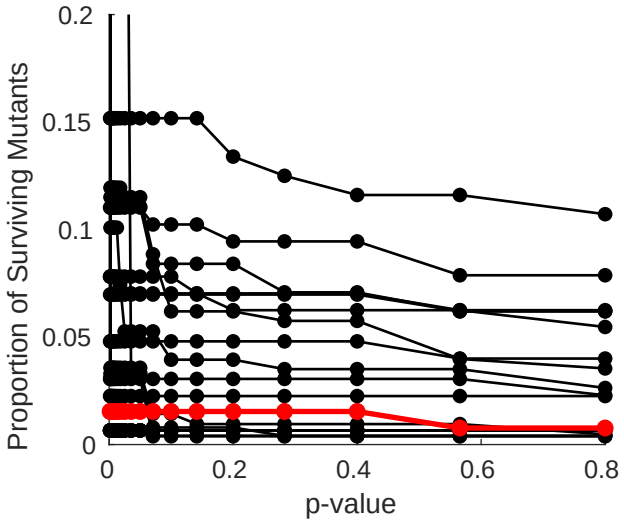

Fig. 4. Survival rate for mutants of reference (red) and buggy (black) versions of RK4 as a function of p-value threshold p_k.

6 Discussion

While the statistical test proposed in section 3.2 still requires choosing a threshold parameter p_k, the results in figures 2, 3 and 4 provide some evidence that the survival rate is relatively insensitive to the precise value of this parameter. Unfortunately, this insensitivity does not extend between algorithms, as the survival rate for the reference code in figure 2 is 5–9% while that in figure 3 is well below 1%. And even once p_k is chosen, the result is a collection of surviving mutants. Consequently, this procedure will not achieve our goal of fully automating the testing process for numerical algorithms.

However, the process of examining the mutants with low p-value proved enlightening. In every case, the mutation operation had inadvertently improved the accuracy of the algorithm, typically by reducing the effective h parameter; for example, replacing h with $0.9h$, $h/2$ or h^2. This increased accuracy is the numerical equivalent of turning your amplifier up to eleven [4], since the user could already adjust the input h parameter to achieve the same outcome.

A similar effect is responsible for the two "buggy" versions of composite Simpson's which proved more fragile than the reference implementation, in the sense that the survival rate of mutants was below that of the reference implementation for all p-value thresholds. In one of these versions the "bug" was to explicitly reduce h, while in the other an error of size $\mathcal{O}(h^4)$ was introduced (the same size as the truncation error expected in the reference implementation). It is clear that such implementations would not be caught by a standard order of accuracy criterion, and perhaps debatable whether they should be considered buggy at all.

7 Conclusions and Future Work

Despite numerical algorithms being an important workload from the earliest days of computing and over sixty years of study, testing their accuracy is still often a manual and qualitative process. Based on the observation that numerical algorithms with higher orders of accuracy are constructed with carefully chosen parameters to cancel out the lower order truncation errors, we hypothesized that their accuracy might be fragile to small changes in the source code. At the same time, tools for automatically generating such small changes in source code have been developed in the domain of mutation testing. Therefore, we proposed to measure the error of mutants and a target implementation over a range of test cases, and use a standard statistical test to estimate the probability of seeing those errors under the null hypothesis that the mutant is as accurate as the target.

For a set of representative but small numerical algorithms implemented in MATLAB and mutated with the MATmute tool, the p-values of most mutants were very small and few mutants had intermediate p-values, at least partially supporting the fragility hypothesis. Moreover, this evaluation process could be fully automated. Although we considered only higher order accurate convergent algorithms, nothing in the evaluation process depends on the order of accuracy, or even that the algorithm is convergent, providing that a sufficient number of test cases could be constructed.

Unfortunately, the distinction in this p-value metric between correct reference implementations and artificially produced buggy versions was algorithm dependent and in some cases lacking; consequently, it does not appear to be directly usable as an automated test of implementation correctness. However, the process may still prove useful if mutants with high p-values can be manually inspected. For our simple reference implementations, it was easy to determine that the mutations in these cases would not reduce (and in many cases would increase) the accuracy of the output. We hypothesize that if a mutant with low p-value were found whose apparent accuracy relative to the target implementation could not be easily explained by code inspection, then there would be strong evidence that the reference implementation was not achieving its design accuracy (or that the mutation engine had stumbled onto an interesting alternative algorithm).

It has become clear that our set of toy implementations and artificially generated bugs is insufficient to properly assess the usefulness of any proposed testing criterion; consequently, we are currently collecting a sample of numerical routines from open source software projects and cataloging wild bugs that have been detected and corrected in them. We plan to make this collection publicly available in the hope that other researchers will take on the challenge of how to better automate the testing of numerical algorithms.

Acknowledgements

The authors would like to thank Kevin Jayamanna for doing related preliminary work in his undergrad thesis, and Daniel Hook for providing MATmute to the community.

References

1. Ascher, U.M., Greif, C.: A First Course on Numerical Methods. SIAM (2011)
2. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: ACM Symposium on Applied Computing. pp. 1318–1322 (2006). https://doi.org/10.1145/1141277.1141584
3. Grcar, J.: John von Neumann's analysis of Gaussian elimination and the origins of modern numerical analysis. SIAM Review **53**(4), 607–682 (2011). https://doi.org/10.1137/080734716
4. Guest, C., McKean, M., Shearer, H., Reiner, R.: This is spinal tap. film (1984), director: Rob Reiner, producer: Karen Murphy
5. Heath, M.T.: Scientific Computing: An Introductory Survey. SIAM (2018)
6. Hook, D., Kelly, D.: Mutation sensitivity testing. Computing in Science & Engineering **11**(6), 40–47 (2009)
7. Howden, W.E.: Theoretical and empirical studies of program testing. In: Proceedings of the 3rd International Conference on Software Engineering. pp. 305–311 (1978)
8. Howden, W.E.: Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering (4), 371–379 (1982)
9. Jézéquel, F., Chesneaux, J.M.: CADNA: a library for estimating round-off error propagation. Computer Physics Communications **178**(12), 933 – 955 (2008). https://doi.org/10.1016/j.cpc.2008.02.003
10. Knupp, P., Salari, K.: Verification of computer codes in computational science and engineering. Chapman & Hall/CRC (2002)
11. Nedialkov, N.S.: VNODE-LP: A validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University (2006)
12. Oberkampf, W.L., Roy, C.J.: Verification and Validation in Scientific Computing. Cambridge University Press (2010)
13. Roy, C.J.: Review of code and solution verification procedures for computational simulation. Journal of Computational Physics **205**(1), 131–156 (2005)
14. Von Neumann, J., Goldstine, H.H.: Numerical inverting of matrices of high order. Bulletin of the American Mathematical Society **53**(11), 1021–1099 (1947)
15. Wikipedia contributors: Sign test — Wikipedia, the free encyclopedia (2019), https://en.wikipedia.org/wiki/Sign_test, [Online; accessed 30-Apr-2019]

A Numerical Algorithms

Our implementations are based on the descriptions in [1]. In this section we briefly describe each of the algorithms, the error measurement for that algorithm, and the test case generation procedure.

A.1 Simpson's method

The definite integral problem is to evaluate

$$I = \int_a^b f(x)\,dx$$

for a specified $a, b \in \mathbb{R}$ and scalar function $f(x)$.

Simpson's method is a fourth order accurate method that is used to approximate definite integrals. It is given by:

$$I_{Simp} = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{b+a}{2}\right) + f(b)\right]$$

Composite Simpson's method involves dividing the domain of integration into subintervals called "panels, " applying Simpson's method to each panel, and summing the result. Let r, the number of panels, be even. The formula is

$$S_{comp} = \frac{h}{3}\left[f(a) + 2\sum_{k=1}^{r/2-1} f(t_{2k}) + 4\sum_{k=1}^{r/2} f(t_{2k-1}) + f(b)\right]$$

where $t_i = a + ih$, $i = \{1, 2, ..., r\}$.

For analysis purposes, we define the error as the absolute value of the difference between the algorithm's output and the (floating point approximation of the) analytic answer. It can be shown that composite Simpson's method has an error bound of

$$\frac{\|f^{(4)}\|_\infty}{180}(b-a)h^4.$$

A typical 3-line implementation of composite Simpson's method in MATLAB generates 180-190 mutants using MATmute, of which roughly 120-130 are viable.

Test cases are generated either using pen-and-paper integration, or by MMS.

A.2 Complete Cubic Spline

A cubic spline is a continuously differentiable, piecewise cubic scalar function $v(x)$ that interpolates points $\{(x_1, f(x_1)), ..., (x_n, f(x_n))\}$, meaning that $v(x_i) = f(x_i)$. We call the spline "complete" because the derivatives at the endpoints $f'(x_1)$ and $f'(x_n)$ are also provided, and we choose $v(x)$ such that $v'(x_1) = f'(x_1)$ and $v'(x_n) = f'(x_n)$.

For analysis purposes, we generate a set of test points between the interpolation points, compute the difference between the value of the interpolant and the value of the original function at these test points, and report the greatest absolute difference across all test points as the error.

MATmute generates roughly 680-700 mutants from our 30-line implementation of a complete cubic spline.

For this problem analytic solutions are trivial to construct: We pick a function f and a set of points $\{x_i\}$. Then, we use $\{x_i\}$ and $\{f(x_i)\}$ as the input to our cubic spline algorithm. The exact answers can be obtained directly from the function f.

A.3 Runge-Kutta schemes

Runge-Kutta methods are used to solve initial value problems for ordinary differential equations. This class of problems is defined by:

$$\frac{dx(t)}{dt} = f(x,t) \text{ such that } x(t_0) = x_0,$$

where $f(x)$ and x_0 are specified. While $x(t)$ may be a vector in general, for our purposes we considered only scalar cases.

We used the classic fourth order accurate Runge-Kutta method. It is given by the following set of update formulae:

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}(k_1 + k_2 + k_3 + k_4), \\ k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_n + h/2, y_n + k_1/2), \\ k_3 &= hf(t_n + h/2, y_n + k_2/2), \\ k_4 &= hf(t_n + h, y_n + k_3), \end{aligned}$$

where y_n is the approximation at time t_n and a fixed stepsize h has been assumed.

For analysis purposes, we define the error in our Runge-Kutta implementation as the absolute value of the difference between the algorithm's output and the analytic answer at a final time.

MATmute generates roughly 480-500 mutants from a typical 4th order Runge-Kutta implementation.

Test cases are generated by MMS: Starting with a function for $x(t)$, supply

$$f(x,t) = \frac{dx(t)}{dt} \text{ and } x_0 = x(t_0)$$

as the inputs.