Automated Software Testing for MATLAB

Software testing can improve software quality. To test effectively, scientists and engineers should know how to write and run tests, define appropriate test cases, determine expected outputs, and correctly handle floating-point arithmetic. Using MATLAB xUnit automated testing framework, scientists and engineers using MATLAB can make software testing an integrated part of their software development routine.

Although they wouldn't describe themselves as programmers, most science and engineering researchers routinely write software to perform their work. Indeed, in one survey, scientists reportedly spent about 30 percent of their work time writing software.[1] Software bugs, therefore, have serious implications for the reliability of scientific and engineering research. One study comparing nine seismic data processing packages showed that seismic traces were frequently shifted incorrectly by off-by-one indexing errors. Even after these faults were corrected, the various program outputs sometimes agreed to within only a single significant digit, mostly because of bugs.[2] Further, several journal papers have been retracted recently because of bugs.[3,4] For example, authors had to retract a widely cited protein-structure paper in *Science* (as well as several related papers) because of a column-swapping bug in a data-analysis program.[5] The paper's incorrect results confused other researchers and apparently even some grant panels.

In professional programming practice, software testing is a fundamental tool for improving software quality. From my own experience interacting with MATLAB users, people often understand the need for software testing, but might not know how to go about it. This observation agrees with other reports that scientists and engineers tend to pick up computing knowledge on their own and have relatively little software engineering experience.[6–8]

The practice of unit testing can help improve the quality of science and engineering software. Unit tests focus on small units of code, such as a class, module, method, or function. To be most effective, unit tests should be automated so that entire test suites can be run quickly and easily and so that the developer doesn't have to review the program output to determine whether the tests passed or failed. Here, I illustrate these ideas and how to practically apply them using MATLAB xUnit, a testing framework available for free download at www.mathworks.com/matlabcentral/fileexchange/22846.

Test Writing Basics

To get started with MATLAB xUnit, you first create a test file containing one or more test cases. Figure 1 shows a test file, `test_upslope_area.m`, that tests the function `upslope_area`, which belongs to a collection of hydrology functions available on the MATLAB Central File Exchange. This function collection includes a test suite written using MATLAB xUnit. (Conventional practice is to write tests after the software is written. Increasingly, professional software developers are turning this convention on its head; see the "Test-Driven Development" sidebar.)

1521-9615/09/$26.00 © 2009 IEEE
COPUBLISHED BY THE IEEE CS AND THE AIP

STEVEN L. EDDINS
The MathWorks, Inc.

Test-Driven Development

In Microsoft researcher Simon Peyton Jones' presentation about writing research papers,[1] he provocatively reverses the commonsense notion that you should perform the research first and then write the paper. He proposes instead that you write the paper first—after all, writing forces you to think clearly about your ideas and also clarifies the research needed to support them.

Many software developers similarly reverse the pattern of coding and then testing. In *test-driven development*,[2] developers first write the tests. Naturally, the tests fail until the necessary code is written. This process is repeated in small, incremental steps until they've implemented all the desired functionality. Writing tests first forces you to think clearly about your program's desired behavior.

In test-driven development, then, test writing is an integral part of the code construction process rather than an activity performed later and intended solely to find bugs. Test-driven development can take some getting used to, but it can also significantly improve code quality and maintainability. Once you become familiar with the basic mechanics of writing and running tests, you might want to give it a try.

References

1. S.P. Jones, "How to Write a Great Research Paper," Microsoft Research, 2004; http://research.microsoft.com/en-us/um/people/simonpj/papers/giving-a-talk/writing-a-paper-slides.pdf.
2. K. Beck, *Test Driven Development: By Example*, Addison-Wesley, 2002.

Test Files

As lines 1-2 of Figure 1 show, the first function in a test file always has the same form. Line 1 shows the function name, which begins with the prefix `test`, and the output argument name, which is always `suite`. The next line calls the helper script `initTestSuite`, which examines the rest of the test file, identifies all test cases in it, and bundles them up into a test suite, which is returned as the output argument. Other functions in the file that begin with `test` are individual test cases. Their names make it easy for the test code reader to understand the test case's purpose.

Test Cases

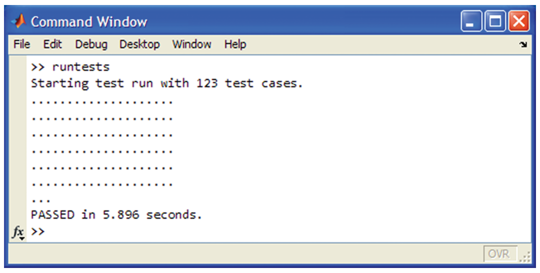
Test-case functions follow a simple pattern (see Figure 1, lines 4–22). First, you initialize a set of input values to pass to the function you're testing. Next, you call the function, capturing the output results. Finally, you use an assertion utility function—such as `assertElementsAlmostEqual` on line 22—to state what you expect to be true about the results. If A equals B within some tolerance, then `assertElementsAlmostEqual(A, B)` returns immediately (and quietly) and the test case passes. Otherwise, it throws an exception; the test framework then catches the exception and notifies you that the test case has failed.

Running Tests

To make test running easy, MATLAB xUnit provides a main driver function, `runtests`. This function finds all the test cases in all the test files in your current working directory, gathers them into a test suite, runs the test suite, and displays the results. Figure 2 shows the output of `runtests` for MATLAB xUnit's own test suite. As the figure shows, `runtests` displays the number of

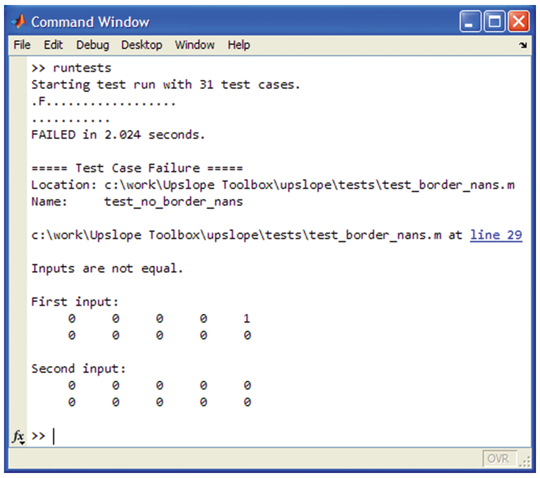
```
 1   function suite = test_upslope_area
 2   initTestSuite;

 3

 4   function test_normalCase
 5   E = [ ...
 6        2  2  2  2  2  3  NaN
 7        2  1  2  2  2  3  NaN
 8        2  1  2  2  2  3  NaN
 9        2  2  2  2  2  3  NaN];

10

11   % Expected result hand-computed.
12   exp_A = [ ...
13        1,  1,   31/9,  8/3,   2,  1,  0
14        1,  12,  41/9,  10/3,  2,  1,  0
15        1,  12,  41/9,  10/3,  2,  1,  0
16        1,  1,   31/9,  8/3,   2,  1,  0];

17

18   R = dem_flow(E);
19   T = flow_matrix(E, R);
20   A = upslope_area(E, T);

21

22   assertElementsAlmostEqual(A, exp_A);
23
24   function test_emptyInput
25   E = [];
26   R = dem_flow(E);
27   T = flow_matrix(E, R);
28   A = upslope_area(E, T);
29
30   assertEqual(A, []);
31
32   function test_constantInput
33   E = ones(5, 10);
34   R = dem_flow(E);
35   T = flow_matrix(E, R);
36   A = upslope_area(E, T);
37
38   assertEqual(A, ones(5, 10));
```

Figure 1. Sample test file. The top function, `test_upslope_area`, initializes the test suite. The functions `test_normalCase`, `test_emptyInput`, and `test_constantInput` each form one test case. (Used with permission from The MathWorks, Inc. All rights reserved.)



```
>> runtests
Starting test run with 123 test cases.
....................
....................
....................
....................
....................
....................
...
PASSED in 5.896 seconds.
>>
```

Figure 2. Running tests. The function `runtests` automatically runs all tests and determines whether they pass or fail. (Used with permission from The MathWorks, Inc. All rights reserved.)



```
>> runtests
Starting test run with 31 test cases.
.F..................
...........
FAILED in 2.024 seconds.

===== Test Case Failure =====
Location: c:\work\Upslope Toolbox\upslope\tests\test_border_nans.m
Name:     test_no_border_nans

c:\work\Upslope Toolbox\upslope\tests\test_border_nans.m at line 29

Inputs are not equal.

First input:
     0     0     0     0     1
     0     0     0     0     0

Second input:
     0     0     0     0     0
     0     0     0     0     0

>> |
```

Figure 3. A failing test. When a test case fails, `runtests` shows the failing test case's location and name and provides clickable links that open the code in the editor at the point of failure. (Used with permission from The MathWorks, Inc. All rights reserved.)

test cases found, how long the test suite ran, and whether the suite passed or failed.

When test cases fail, `runtests` displays additional information about each failure (see Figure 3). The display includes the name and location of each failing test case, as well as clickable links that load the code directly into the MATLAB editor at the point of failure.

Constructing Test Cases

Gathering test cases for your programs' functions and methods is something of an art. You want to find a minimal set of cases that covers the meaningful variations in input data and that also covers the code logic.

Data Test Cases

To gather your test cases, start with the nominal case—that is, choose a typical set of input values and determine the expected outputs. For an example, see Figure 1 lines 4–22. Next, consider inputs at and near the end of the allowed data range (that is, conduct a *boundary analysis*[9]). These test cases help identify off-by-one errors, such as writing `N` instead of `N-1` or vice versa.

With more experience, you can gain insight into coding-error patterns and add test cases that are likely to expose such errors. Experienced MATLAB quality engineers, for example, routinely write test cases with empty input matrices or `NaN` and `Inf` values. From my own experience with image processing software, I anticipate possible coding errors for constant-valued images or those that have a single row or column. Lines 24–38 of Figure 1 show a couple of error-guessing[9] test cases for the `upslope_area` function.

Logic Test Cases

In addition to considering input data variations, you should also think about the various logic paths through your code. Structured basis testing[9] is a straightforward way to count the minimum number of test cases you need to fully exercise a particular function's logic. You start by counting one for the straight-line path through the code; that is, for the path without loops or branches. You then count one for each appearance of the following MATLAB keywords and operators: `if`, `elseif`, `while`, `for`, `case`, `otherwise`, `try`, `&&`, and `||`.

Figure 4 shows a code fragment of the `run` method of the MATLAB xUnit `TestSuite` class. Minimally exercising this code's logic paths requires at least four test cases corresponding to the highlighted code fragments. The test cases might include

- two input arguments (`nargin == 2`) and no test components (`numel(self.testComponents) == 0`), a combination that tests the straight-line path;
- one input argument (`nargin == 1`);
- multiple test components that all pass (`numel(self.testComponents) > 0`); and
- some failing and some passing test components.

There are more sophisticated techniques for selecting test cases,[9] but this straightforward method will get you started and ensure basic test coverage of all the code paths.

Determining the Expected Values

Determining expected values can be difficult, especially for complex scientific and engineering research simulations. Full verification and validation requires that we consider factors such as the appropriateness of physical models and approximations, error estimates, and numerical solution stability.[7,8]

As a practical matter, to deal with testing complex models, I recommend that you start by focusing on the "unit" in "unit test." Even complex simulations comprise simpler computational units that you can individually verify. Verifying individual computational units' behavior is a helpful step toward verifying overall system behavior. Focusing on the testability of small computational units also encourages the development of modular, more maintainable code.

When it's infeasible to hand compute expected results—even for small computational units—there are several strategies you can try.

Compare with alternate computation method. The Matlab `fft` function computes the discrete Fourier transform (DFT) using a fast algorithm. Some `fft` test cases compare the function's output against the output computed directly using the standard DFT mathematical equation. The direct method's relative slowness doesn't matter for testing purposes.

Check the output's expected properties. Sometimes, you can check program output against the solution's expected mathematical or physical properties. For example, the Matlab backslash operator (\) solves linear systems of equations. You can check its output, `x = A\b`, by ensuring that its residual is sufficiently close to zero—that is, `res = A*x - b` should be very small.

Compare with baseline results. You might be able to independently verify your program's output. If so, you can save the verified output to a data file that the test-case code can load as the expected value. Baseline tests can have some value even when independent verification isn't available. In particular, baseline tests help you catch unintended changes in program behavior. Writing baseline tests is a good way to get started when you face an existing body of untested code.

Writing Clear Test Cases

Test code is subject to the same problems as any other type of code. But, when test code becomes hard to debug and maintain, it tends to fall into disuse. Therefore, you should write your test code so that it's trivially easy to track down and

```
 1  function did_pass = run(self, monitor)
 2  if nargin < 2
 3    monitor = CommandWindowTestRunDisplay();
 4  end
 5
 6  monitor.testComponentStarted(self);
 7  did_pass = true;
 8  for k = 1:numel(self.TestComponents)
 9    this_component_passed = ...
10      self.TestComponents{k}.run(monitor);
11    did_pass = did_pass && this_component_passed;
12  end
13
14  monitor.testComponentFinished(self, did_pass);
```

Figure 4. Structured basis testing. We count one each for the straight-line path, the `if` statement on line 2, the `for` loop on line 8, and the `&&` operator on line 11. The total test-case count is four. (Used with permission from The MathWorks, Inc. All rights reserved.)

```
function test_iccWhitePoint
exp_ICCwhite = ...
    [hex2dec('7b6b')/hex2dec('8000') 1 ...
    hex2dec('6996')/hex2dec('8000')];
assertEqual(whitepoint('ICC'), exp_ICCwhite);

function test_d50WhitePoint
exp_d50 = [0.96419865576090, 1.0, ...
    0.82511648322104];
assertEqual(whitepoint('d50'), exp_d50);

function test_d65WhitePoint
exp_d65 = [.9504 1.0 1.0889];
assertEqual(whitepoint('d65'), exp_d65);

function test_caseInsensitivity
assertEqual(whitepoint('ICC'), whitepoint('icc'));

function test_default
assertEqual(whitepoint, whitepoint('ICC'));
```

Figure 5. Test file for the Image Processing Toolbox's `whitepoint` function. Each of the five test cases verifies just one aspect of program behavior and is easy to understand at a glance. (Used with permission from The MathWorks, Inc. All rights reserved.)

understand test failures. Here are two suggested ways to do this.

Avoid conditional logic. Using conditional logic makes it hard to quickly understand the code. Also, when different test code executes during different runs, tracking down test failures can be frustrating. Strive therefore to write test code that has no conditional logic.

Verify one condition per test case. Figure 5 shows a portion of a test file for the Matlab Image Processing Toolbox's `whitepoint` function. As the figure shows, there are five different test cases and each verifies a single, specific aspect of the program behavior. The first three test cases verify the correct output for three different inputs (`ICC`, `d50`, and `d65`). The fourth test case (lines 16-17) verifies the inputs' case insensitivity, and the fifth test case (lines 19-20) verifies the correct default behavior. The five tests cases are easy to understand at a glance and have specific names that clearly communicate the programmer's intent.

Using Random Values Appropriately

Novice test writers often use randomly generated input data for their test cases. As Gerard Meszaros notes, this kind of nondeterministic test case can be difficult to debug because it can be hard to reproduce the failure.[10] In the worst case, a particular test failure might never appear again despite numerous test runs. To avoid frustration when using randomly generated input data, write your test to use the same set of randomly generated input data for every test run by using a fixed "seed" for the random number generator. Alternatively, if you want to use a different seed for every test run, display or save the seed state so that if you have a test failure, you can reproduce and debug it.

Comparing Floating-Point Values

For numerical code, checking that the values in your test cases are correct can be surprisingly difficult. The problem is illustrated by an oft-heard complaint on the Usenet newsgroup comp.soft-sys.matlab: "In MATLAB, why is 0.1 + 0.1 + 0.1 not equal to 0.3?!" That is,

```
>> 0.1 + 0.1 + 0.1 == 0.3
ans =
       0
```

The explanation is that MATLAB, like most other numeric computation languages today, computes using IEEE floating-point numbers and arithmetic.[11,12] The decimal fractions above, 0.1 and 0.3, cannot be represented exactly in IEEE floating point because 1/10 and 3/10 don't have a finite representation as a binary fraction. Also, floating-point arithmetic operations are approximate, not exact, and are thus subject to round-off error. As a surprising consequence, floating-point arithmetic systems don't strictly obey associativity for multiplication and addition. You can see this readily in MATLAB:

```
>> 3/14 + (3/14 + 15/14) ...
   == (3/14 + 3/14) + 15/14
ans =
       0
```

The actual difference between the two expressions is very small, but not zero:

```
>> (3/14 + (3/14 + 15/14)) ...
   - ((3/14 + 3/14) + 15/14)
ans =
       -2.2204e-016
```

Because the order of operations matters, two mathematically equivalent computational procedures can produce different answers when implemented using floating-point arithmetic. Even identical code can produce different answers on different computers, or when compiled using different compilers, because of optimization variations or differing versions of underlying runtime libraries, such as the basic linear algebra subprograms (BLAS).

For testing, then, we should usually avoid exact equality tests when checking floating-point results and instead use some sort of tolerance. There are two types of floating-point tolerance tests commonly used: absolute and relative. Absolute tolerance tests for two values a and b:

$$|a-b| \le T,$$

where T is the tolerance. The relative tolerance test is

$$\frac{|a-b|}{\max(|a|,|b|)} \le T.$$

To understand the difference between these tests, consider the values 10.1 and 11.2 and the tolerance 10^{-3}. The absolute difference between these values is 1.1; given the specified tolerance, they clearly fail the absolute tolerance test. Now, consider the values 134,712.5 and 134,713.6. Their absolute difference is also 1.1, so they also fail the absolute tolerance test. However, they pass the relative tolerance test:

$$\frac{|134712.5-134713.6|}{\max(|134712.5|,|134713.6|)} = 8.1655 \cdot 10^{-6} <= 10^{-3}.$$

Specifying a relative tolerance of 10^{-n} is roughly equivalent to saying you expect two values to differ by no more than one unit in the n^{th} significant digit.

When you compare two vectors, as opposed to two scalars, you should consider whether to apply the tolerance test to each vector element independently or whether to apply it using a vector norm. For example, suppose you compared the two vectors [1 100,000] and [2 100,000] using a relative tolerance of 10^{-3}. If you compare the two vectors elementwise, they clearly fail the tolerance test because the relative difference between 1 and 2 is much higher than the tolerance value. However, another way to compare vectors is using a vector norm, such as L2:

$$\frac{\| v_1 - v_2 \|}{\max(\| v_1 \|, \| v_2 \|)} \le T.$$

The two vectors `[1 100,000]` and `[2 100,000]` pass this form of relative tolerance test because they differ in the L2-norm sense by only about 1 part in 100,000.

To compare floating-point values, MATLAB xUnit provides the functions `assertElementsAlmostEqual` and `assertVectorsAlmostEqual`, either of which can use a relative or an absolute tolerance.

When choosing a tolerance, it helps to consider the machine precision, or ε. This quantity is the spacing of floating-point numbers between successive powers of two, relative to the lower power of two. For example, the next floating-point number higher than 1 is $1 + \varepsilon$, while the next floating-point number higher than 2 is $2 + 2\varepsilon$. The MATLAB function `eps` returns the machine precision. For double-precision floating point, ε is approximately $2.2 \cdot 10^{-16}$, and for single precision it's approximately $1.2 \cdot 10^{-7}$.

For computations involving relatively little floating-point arithmetic, a small multiple of ε might be appropriate, such as 100ε. For computations involving coarse approximations, we might use much higher tolerances. The default relative tolerance for the MATLAB xUnit assertion functions is approximately 10^{-8}, corresponding to eight significant digits. In general, your choice of tolerance depends on the particular domain and the nature of the your models and approximations.

MATLAB xUnit Design and Architecture

MATLAB xUnit was designed to achieve several objectives:

- Typical MATLAB users should be able to write and run tests easily.
- Tests should run automatically.
- The framework should support the development of other programs—such as graphical user interfaces—that can control test execution and reporting.
- The framework should be scalable, handling several tests on a simple, short program as well as extensive test suites for large applications.
- The framework should borrow familiar, proven design and usage concepts from test frameworks created for other languages.

To help satisfy these goals, we modeled the framework on xUnit, a class of testing frameworks.[10]

xUnit

The xUnit family of testing frameworks was popularized both by the JUnit Java test framework and the article "Test Infected—Programmers Love Writing Tests" by Erich Gamma and Kent Beck.[13] You can find xUnit-style test frameworks for most widely used programming languages. Although different xUnit frameworks vary in the details, most share several common design elements. I'll now describe the xUnit design elements that are most relevant to the workings of MATLAB xUnit.

First, MATLAB xUnit uses test-case objects and methods. Each test case is instantiated as a full object. The test framework instantiates one object of the test subclass for each of the test methods it contains. The corresponding MATLAB xUnit base class is `TestCase`.

Second, running a particular test case proceeds in four phases:

1. Perform setup steps, such as opening a file or connecting to a database.
2. Call the function being tested.
3. Verify that the expected outcome has occurred.
4. Perform any required teardown steps, such as closing a file or database connection.

Third, test suites are hierarchical. A MATLAB xUnit test suite is a collection of individual test cases as well as other test suites. This hierarchical nature allows test suites to scale from small to large applications.

Four, most xUnit frameworks support automatic test-case discovery so that the programmer doesn't have to create and update an explicit list of test cases. The MATLAB xUnit function `runtests` discovers test cases automatically by scanning a directory for test files and then it automatically runs all the discovered test cases.

Adapting xUnit for Procedural Programming

Most MATLAB code today is procedural, and most scientists and engineers who use MATLAB prefer procedural programming. Although implemented in a fully object-oriented xUnit fashion, MATLAB xUnit was designed from the start to be used by procedural programmers and its documentation focuses on procedural test writing.

The `FunctionHandleTestCase`—which is a `TestCase` subclass—supports the procedural testing style. The helper script `initTestSuite` (see Figure 1 line 2) automatically turns each test function in the file into a `FunctionHandleTestCase` object and returns the collection of them as a `TestSuite` object.

If you're a scientist or engineering researcher, you probably don't regard yourself as a professional software developer. Still, you probably write software to get your work done. That software's quality significantly affects the quality of your research. MATLAB is widely used for research in many science and engineering disciplines; the availability of an industry-standard testing approach for software created with MATLAB can help you incorporate automated software testing into your regular practice. By doing so, you can increase your confidence in both your software and in the accuracy of the research results based upon it. CiSE

Acknowledgments

Special thanks to Greg Wilson and Gerard Meszaros for their helpful comments on MATLAB xUnit's design and architecture, as well as to Mara Yale, Rob Comer, and Bill McKeeman for their suggestions on this article. I also thank the anonymous reviewers for their helpful and constructive comments. MATLAB is a registered trademark of The MathWorks, Inc.

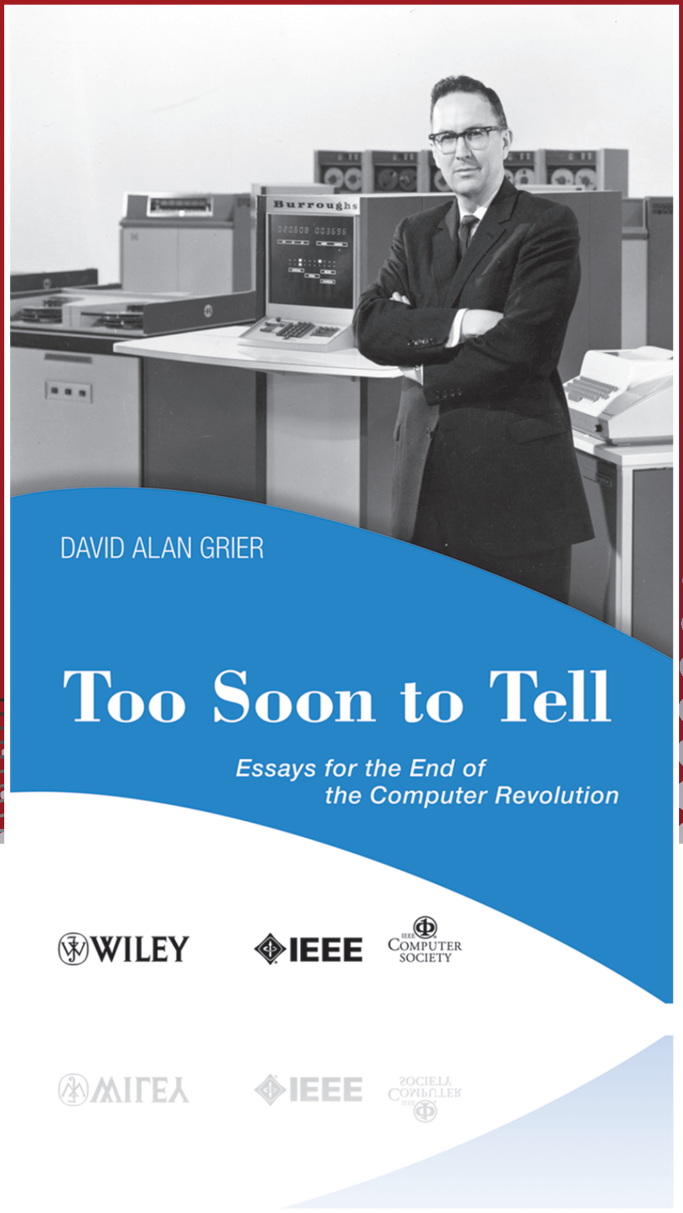
References

1. J.E. Hannay et al., "How Do Scientists Develop and Use Scientific Software?" *Proc. 2nd Int'l Workshop Software Eng. Computational Science and Eng.*, IEEE CS Press, 2009, pp. 1–8.
2. L. Hatton, "The T Experiments: Errors in Scientific Software," *IEEE Computational Science & Eng.*, vol. 4, no. 2, 1997, pp. 27–38.
3. G. Chang et al., "Retraction of Pornillos et al., *Science* 310 (5756) 1950–1953. Retraction of Reyes and Chang, *Science* 308 (5724) 1028–1031. Retraction of Chang and Roth, *Science* 293 (5536) 1793–1800," *Science*, vol. 314, no. 5807, 2006, p. 1875; www.sciencemag.org/cgi/content/full/314/5807/1875b.
4. B.G. Hall and S.J. Salipante, "Retraction: Measures of Clade Confidence Do Not Correlate with Accuracy of Phylogenetic Trees," *PLoS Computational Biology*, vol. 3, no. 3, 2007; www.ploscompbiol.org/article/info:doi%2F10.1371%2Fjournal.pcbi.0030051.
5. G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, 2006, pp. 1856–1857.
6. G. Wilson, "What Should Computer Scientists Teach to Physical Scientists and Engineers?" *IEEE Computational Science & Eng.*, vol. 3, no. 2, 1996, pp. 46–65.
7. D.E. Stevenson, "A Critical Look at Quality in Large-Scale Simulations," *Computing in Science & Eng.*, vol. 1, no. 3, 1999, pp. 53–63.
8. D. Kelly and R. Sanders, "The Challenge of Testing Scientific Software," *Proc. Conf. Assoc. Software Testing: Beyond the Boundaries* (CAST 2008), Assoc. Software Testing, 2008; www.associationforsoftwaretesting.org/documents/cast08/DianeKellyRebeccaSanders_TheChallengeOfTestingScientificSoftware_paper.pdf.
9. S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed., Microsoft Press, 2004.
10. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Pearson Education, 2007.
11. C. Moler, "Floating Points: IEEE Standard Unifies Arithmetic Model," Cleve's Corner, The MathWorks, 1996; www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf.
12. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *Computing Surveys*, vol. 23, no. 1, 1991, pp. 5–48.
13. E. Gamma and K. Beck, "JUnit Test Infected: Programmers Love Writing Tests," Source Forge, 1998; http://junit.sourceforge.net/doc/testinfected/testing.htm.

***Steven L. Eddins** manages the image processing and geospatial computing software development team at The MathWorks, Inc. He is coauthor of* Digital Image Processing Using MATLAB *(Gatesmark Publishing, 2009). Eddins has a PhD in electrical engineering from the Georgia Institute of Technology. He's a senior member of IEEE and a member of SPIE. Contact him at steve.eddins@mathworks.com.*

cn *Selected articles and columns from IEEE Computer Society publications are also available for free at http://ComputingNow.computer.org.*

NEW TITLE FROM WILEY & IEEE CSPress



A rich collection of essays exploring enduring themes of the computer era

- Based on David Alan Grier's popular monthly column "In Our Time" for *Computer* magazine, with twenty entirely new pieces
- Collects poignant and accessible portraits of the computer revolution and the people and families it has affected

978-0-470-08035-1 • March 2009
Paperback • 238 pages • $29.99
A Wiley-IEEE Computer Society Press Publication

To Order

15% Off for CS Members

North America
1-877-762-2971

Rest of the World
+ 44 (0) 1243 843291

Meet the Author
computingnow.computer.org/grier

WILEY Publishers Since 1807

IEEE Celebrating 125 Years of Engineering the Future

IEEE computer society
www.computer.org