Version of Record: https://www.sciencedirect.com/science/article/pii/S1877750321000442
Manuscript_1abf18b3c66c3987a6acfe6765f00964

Unit and Regression Tests of Scientific Software: A Study on SWMM

Zedong Peng[a], Xuanyi Lin[a], Michelle Simon[b], Nan Niu[a,*]

[a] *Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221, USA*
[b] *U.S. EPA Office of Research and Development, Center Water Infrastructure Division, Cincinnati, OH 45268, USA*

Abstract

Testing helps assure software quality by executing a program and uncovering bugs. Scientific software developers often find it challenging to carry out systematic and automated testing due to reasons like inherent model uncertainties and complex floating-point computations. Extending the recent work on analyzing the unit tests written by the developers of the Storm Water Management Model (SWMM) [32], we report in this paper the investigation of both unit and regression tests of SWMM. The results show that the 2,953 unit tests of SWMM have a 39.7% statement-level code coverage and a 82.4% user manual coverage. Meanwhile, an examination of 58 regression tests of SWMM shows a 44.9% statement-level code coverage and a near 100% user manual coverage. We also observe a "getter-setter-getter" testing pattern from the SWMM unit tests, and suggest a diversified way of executing regression tests.

Keywords:
scientific software, unit testing, regression testing, user manual, test coverage, Storm Water Management Model (SWMM)

*Corresponding author
Email addresses: pengzd@mail.uc.edu (Zedong Peng), linx7@mail.uc.edu (Xuanyi Lin), simon.michelle@epa.gov (Michelle Simon), nan.niu@uc.edu (Nan Niu)

1. Introduction

Scientific software is commonly developed by scientists and engineers to better understand or make predictions about real world phenomena. Without such software, it would be difficult or impossible for many researchers to do their work. For example, in nuclear weapons simulations, code is used to determine the impact of modifications as these weapons cannot be field tested [34]. Scientific software includes both software for end-user researchers (e.g., climate scientists and hydrologists) and software that provides infrastructure support (e.g., message passing and scheduling). Because scientific software needs to produce trustworthy results and function properly in mission-critical situations, rigorous software engineering practices should be adopted to assure software qualities.

Testing, which is important for assessing software qualities, has been employed extensively in business/IT software. However, developers of scientific software have found it more difficult to apply some of the traditional software testing techniques [20]. One chief challenge is the lack of a test oracle. An *oracle* in software testing refers to the mechanism for checking whether the program under test produces the expected output when executed using a set of test cases [15]. Many testing techniques—especially unit testing commonly carried out in business/IT software development projects—require a suitable oracle to set up the expectation with which the actual implementation (e.g., sorting inventory items or calculating tax returns) can be compared.

In addition to unit testing, regression testing is important to ensure that the new changes of the scientific software do not break what already works. There exist many approaches to regression testing [2]. Therefore, regression tests can be used to check the correctness of not only a computational unit (e.g., a method or a function), but also the software as a whole. In regression testing, the output from a previous version of the software can serve as an oracle for the current version. Our ongoing collaborations with the U.S. Environmental Protection Agency's Storm Water Management Model (SWMM) team suggests that both unit and regression tests are developed throughout the project's five decades history [42]. To comply with the recent movements toward improving public access to data [39], these tests are released in GitHub, oftentimes together with the source code of SWMM. However, little is known about the characteristics of the SWMM tests.

To shorten the knowledge gap, we report in this paper the tests that are publicly available for the SWMM software. In recent work [32], we performed

coverage analysis of SWMM's 1,458 unit tests and uncovered a "getter-setter-getter" pattern from the unit tests. In this paper, we extend the unit test analysis by not only updating the unit tests to 2,953 but also incorporating regression testing practices manifested in SWMM repositories. Altogether, we provide a detailed look at how many tests were written in what environments, and further analyze the coverage of the unit and regression tests from two angles: how much they correspond to the user manual and to the codebase.

The contributions of our work lie in the qualitative characterization and quantitative examination of the tests written and released by the scientific software developers themselves in the context of SWMM. Our results clearly show that oracle *does* exist in scientific software testing, and that the numerical regression tests could support the development of test oracles as well as support the better writing of units tests via the "getter-setter-getter" pattern. In what follows, we provide background information and introduce SWMM in Section 2. Section 3 presents our search of SWMM tests, Section 4 analyzes the test coverage, and finally, Section 5 draws some concluding remarks and outlines future work.

2. Background

2.1. Testing Scientific Software

Testing is a mainstream approach toward software quality, and involves examining the behavior of a system in order to discover potential differences. Kanewala and Bieman [20] performed a systematic literature review of 62 relevant studies. Although scientific software developers and testers conducted testing at different levels: unit testing, integration testing, system testing, acceptance testing, and regression testing, many challenges remain. For example, several studies described the use of regression testing to compare the current output to previous outputs to identify faults introduced during code modifications [13, 18, 36]. However, testers must manually specify the input variables' values to run the different versions of the scientific simulation; they must also define the tolerances for output comparisons [20].

The most significant obstacle identified in Kanewala and Bieman's literature review is the oracle problem that impacts unit testing directly [20]. Given an input for the system under test, the *oracle problem* refers to the challenge of distinguishing the corresponding desired, correct behavior from observed, potentially incorrect behavior [4]. The oracle of desired and correct

behavior of scientific software, however, can be difficult to obtain or may not be readily available. Kanewala and Bieman [20] listed five reasons.

- Some scientific software is written to find answers that are previously unknown; a case in point is the program computing a large graph's shortest path of any arbitrary pair of nodes.
- It is difficult to determine the correct output for software written to test scientific theory that involves complex calculations, e.g., the large, complex simulations are developed to understand climate change [12].
- Due to the inherent uncertainties in models, some scientific programs do not give a single correct answer for a given set of inputs; Hinsen [17] depicted the approximation tower in computational science ranging from physical reality through numeral analysis to software implementation.
- Requirements are unclear or uncertain up-front due to the exploratory nature of the software, e.g., the unprecedented exploration recently made by NASA's solar probe has evolving requirements [14].
- Choosing suitable tolerances for an oracle when testing numerical programs is difficult due to the involvement of complex floating point computations [21, 22, 33].

Barr *et al.* [4] showed that test oracles could be explicitly specified or implicitly derived. In scientific software testing, an emerging technique to alleviate the oracle problem is metamorphic testing [20, 38]. For example, Ding and colleagues [10] tested an open-source light scattering simulation performing discrete dipole approximation. Rather than testing the software on each and every input of a diffraction image, Ding *et al.* systematically (or metamorphically) changed the input (e.g., changing the image orientation) and then compared whether the software would meet the expected relation (e.g., scatter and textual pattern should stay the same at any orientation).

While we proposed hierarchical and exploratory ways of conducting metamorphic testing for scientific software [24, 25], our work is similar to that of Ding *et al.*'s [10] by gearing toward the entire application instead of checking the software at the unit testing level. Unit tests are especially useful for guarding the developers against programming mistakes and for localizing the errors when they occur. Similarly, regression tests are valuable for guarding

against mistakes during code changes. Thus, we are interested in the unit and regression tests written and released by the scientific software developers themselves, and for our current work, the focus is on SWMM.

2.2. Storm Water Management Model (SWMM)

The Storm Water Management Model (SWMM) [42], created by the U.S. Environmental Protection Agency (EPA) and others, is a dynamic rainfall-runoff simulation model that computes runoff quantity and quality from primarily urban areas. The development of SWMM began in 1971 and since then the software has undergone several major upgrades.

We studied the version 5.1.014 of SWMM which was released in February 2020. Both the model's structure and its user interface (UI) have been modernized through the years. The top of Figure 1 shows a screenshot of SWMM running as a Windows application. The two main parts of SWMM are the computational engine written in C/C++ with about 46,300 lines of code, and the UI written using Embarcadero's Delphi 10.3. Note that the computational engine can be compiled either as a DLL under Windows or as a stand-alone console application under both Windows and Linux. The bottom of Figure 1 shows that running SWMM in the command line takes three parameters: the input, report, and output files.

The users of SWMM include hydrologists, engineers, and water resources management specialists who are interested in the planning, analysis, and design related to storm water runoff, combined and sanitary sewers, and other drainage systems in urban areas. Thousands of studies worldwide have been carried out by using SWMM, such as generating the spatial distribution of precipitation for the Ballona Creek Watershed (a large urban catchment in California) [3], predicting the pollution in rainy weather in a combined sewer system catchment in Santander, Spain [41], modeling the hydrologic performance of green roofs in Wrocław, Poland [8], and simulating a combined drainage network located in the center of Athens, Greece for preparing events like pluvial flooding [23].

Despite the global adoptions of SWMM, the United States EPA maintains and releases the official version of the software. Our collaborations with the SWMM team involve the creation of a connector allowing for the automated parameter calibration [19], and through developing this software solution, we recognize the importance of testing in assuring quality and contribute hierarchical and exploratory methods of metamorphic testing [24, 25]. In addition, we release our metamorphic tests in the connector's GitHub repository [26],

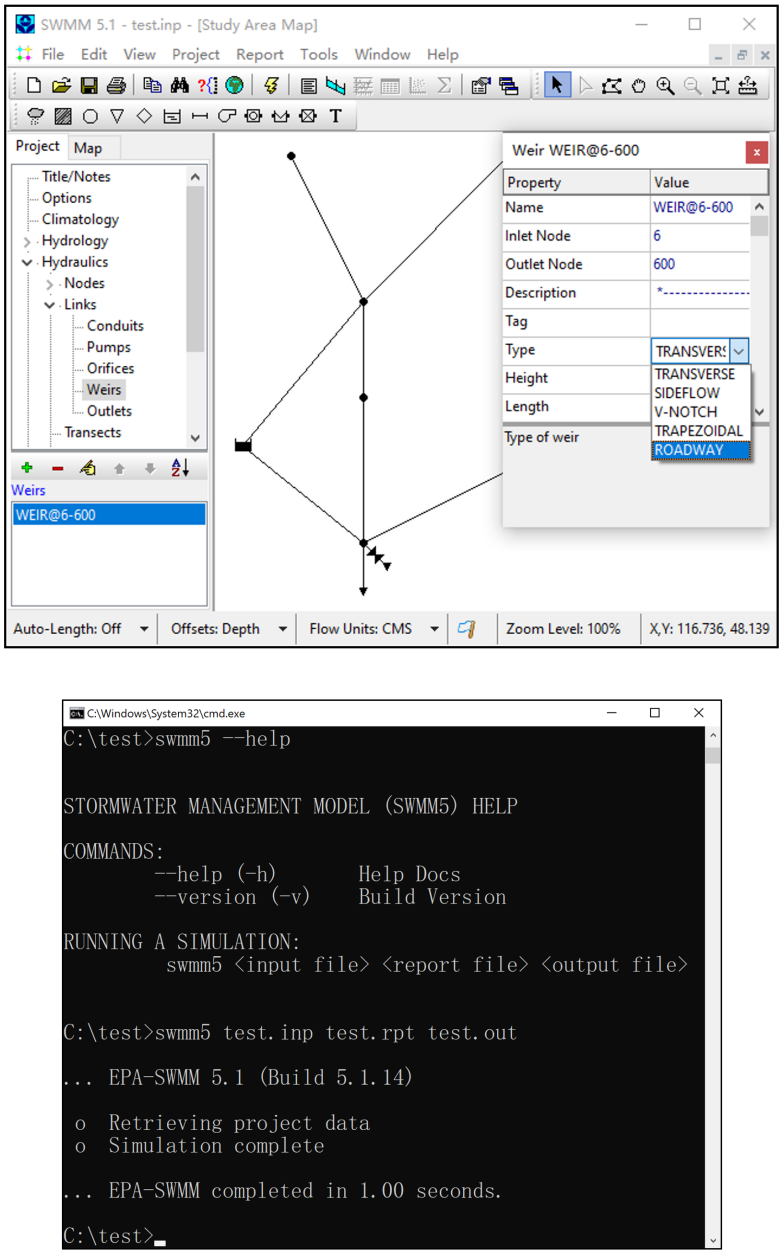


Figure 1: SWMM running as a Windows application (top) and the computational engine of SWMM running as a console application (bottom).

promoting the open access to data and research results [39]. For similar purposes, we realize that the SWMM team has released their own tests in

Table 1: SWMM tests in two repositories.

Source	Author (#; Role)	# of Tests	Type	Method	Language
https://github.com/SWMM-Project/swmm-nrtestsuite/tree/dev/public	(3+; EPA and non-EPA developers)	58	numerical regression testing	numpy. allclose	Python, json
https://github.com/OpenWaterAnalytics/Stormwater-Management-Model	(3+; EPA and non-EPA developers)	2,953	unit testing	boost test	C++

publicly accessible repositories. Understanding these tests is precisely the objective of our study.

3. Identification and Characterization of SWMM Tests

We performed a survey analysis of the SWMM tests released in publicly accessible repositories. Our search was informed by the SWMM team members and also involved using known test repositories to find additional ones. Table 1 lists the two repositories that we identified, as well as the characteristics of the testing data. It is worth noting that Table 1 updates our earlier work [32] where five regression-test sources and one unit-test source were reported. The main reason was that, by using a more authoritative GitHub link, we were able to identify the tests in different branches. In addition, some links reported in our earlier work [32] were no longer available.

The tests of Table 1 can be classified in two categories: numerical regression testing and unit testing. The top of Figure 2 illustrates how Python's numpy.allclose() function is used in SWMM's regression testing. The two variables, test and ref, represent the outputs from two different versions of the software. The if condition of lines 48–49 checks these outputs are of the same length, i.e., they contain the same number of output items. Then, lines 52–64 check if each pair of values is equivalent (numpy.array_equal() returns true) or sufficiently close (numpy.testing.assert_allclose() returns true) for the absolute and relative tolerances specified by comp_args[0] and comp_args[1]. For regression testing, we count each SWMM input (an .inp file) as a test, i.e., a single unit for different code versions to check equivalence. As shown in Table 1, there are 58 regression tests in total, organized in 6 folders. Table 2 lists these .inp files, their folder information, and the number of input

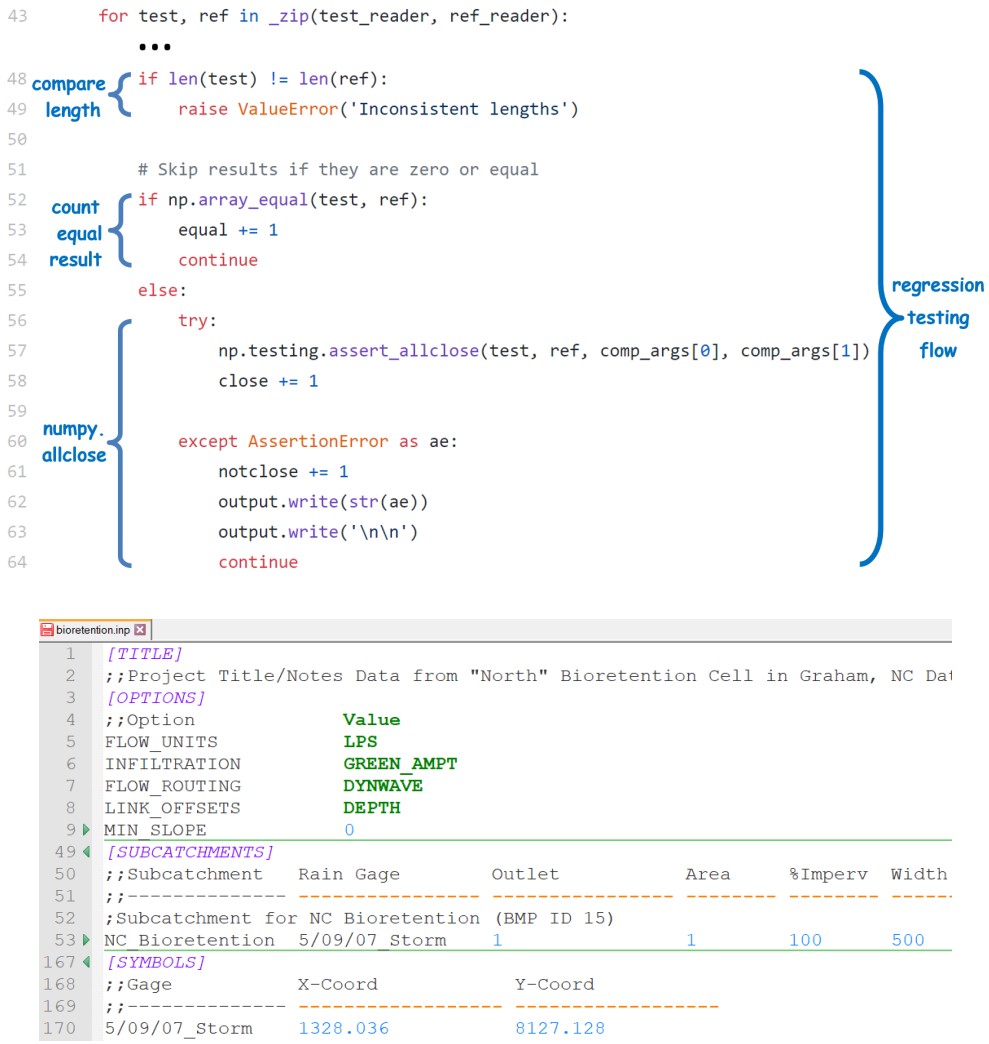


Figure 2: Illustration of numpy.allclose() used in SWMM regression testing (top) and an numerical input file (bottom).

parameters in each .inp file.

The bottom of Figure 2 shows the content of one regression test, namely bioretention.inp. This file has 170 lines in total, and contains 86 parameters, illustrating the large input space of the SWMM simulation. Concrete values are given to the input parameters, allowing SWMM, or different versions of SWMM, to run.

In contrast, unit testing does not compare different versions of SWMM

Table 2: SWMM's 58 regression tests organized in 6 folders and the number of parameters in each test.

Folder	Regression Test	# of Parameters	Folder	Regression Test	# of Parameters
examples	Example1.inp	121	swc	swc1.inp	31
	Example2.inp	65		swc2.inp	31
	Example3.inp	69		swc3.inp	36
	Example4.inp	74		swc4.inp	31
	Example5.inp	105		swc5.inp	31
extran	extran1.inp	76		swc6.inp	32
	extran2.inp	66		swc7.inp	31
	extran3.inp	68		swc8.inp	31
	extran4.inp	71		swc9.inp	32
	extran5.inp	72		swc10.inp	31
	extran6.inp	68		swc11.inp	32
	extran7.inp	68		swc12.inp	31
	extran8a.inp	69		swc13.inp	31
	extran9.inp	55		swc14.inp	32
	extran10.inp	63		swc15.inp	31
routing	test1.inp	63		swc16.inp	32
	test2.inp	72		swc17.inp	39
	test3.inp	63		swc18.inp	36
	test4.inp	63		swc19.inp	36
	test5.inp	63		swc20.inp	36
user	user1.inp	91		swc21.inp	36
	user2.inp	109		swc22.inp	36
	user3.inp	102		swc23.inp	36
	user4.inp	100		swc24.inp	36
	user5.inp	103		swc25.inp	36
update _v5111	bioretention.inp	86		catchment_as_outfall.inp	93
	events_example.inp	121		gate_control_2.inp	80
	gate_control_3.inp	75		ncdc_format.inp	71
	porous_pavement.inp	83		rain_garden.inp	71

but focuses on the specific computations of the software. The second row of Table 1 shows that the GitHub repository contains 2,953 tests written by a group of EPA and non-EPA developers by using the boost environment [9]. In particular, Boost test library is used in SWMM, and Boost.Test provides both the interfaces for writing and organizing tests and the controls of their executions. Figure 3 uses a snippet of test_toolkitapi_lid.cpp to explain the three different granularities of SWMM unit tests. At the fine-grained level are the assertions, e.g., line #334 of Figure 3 asserts "error == ERR_NONE". The value of "error" is obtained from line # 333. As shown in Figure 3, we define a *test* in our study to be one instance that triggers SWMM execution and the associated assertions with that triggering. In Figure 3, three tests are shown. A group of tests forms a *test case*, e.g., lines #311–616 encapsulate many tests into one BOOST_FIXTURE_TEST_CASE. Finally, each file

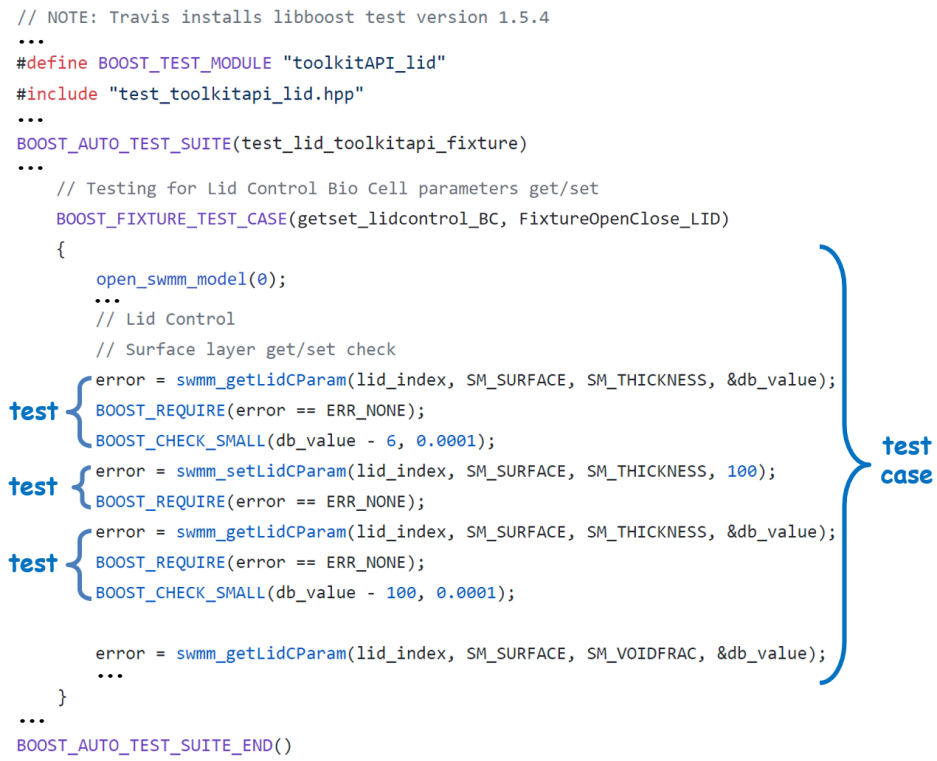


Figure 3: Illustration of SWMM tests and test cases written in the boost environment.

corresponds to a *test suite* containing one or more test cases. Table 3 lists the 16 test suites, and the number of test cases and tests per suite. Averagely speaking, each test suite has 7.9 test cases, and each test case has 23.4 tests.

4. Coverage of SWMM Tests

Having characterized how many SWMM tests were developed in what environments, we turn our attention to the unit and regression tests for quantitative analysis. When tests are considered, *coverage* is an important criterion. This is because a program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage [6]. Practices that lead to higher testing coverage have therefore received much attention. For example, test-driven development (TDD) advocates test-first over the traditional test-last

Table 3: SWMM unit tests: test suites, number of test cases, and number of tests.

Test Suite	# of Test Cases	# of Tests
test_canonical.cpp	11	11
test_coupling.cpp	2	45
test_gage.cpp	1	11
test_lid.cpp	17	677
test_lid_results.cpp	8	555
test_output.cpp	15	61
test_pollut.cpp	1	15
test_solver.cpp	1	2
test_swmm.cpp	11	11
test_toolkit.cpp	14	144
test_toolkitapi.cpp	12	128
test_toolkitapi_coupling.cpp	6	35
test_toolkitapi_gage.cpp	1	11
test_toolkitapi_lid.cpp	17	677
test_toolkitapi_lid_results.cpp	8	555
test_toolkitapi_pollut.cpp	1	15
Σ	126	2,953

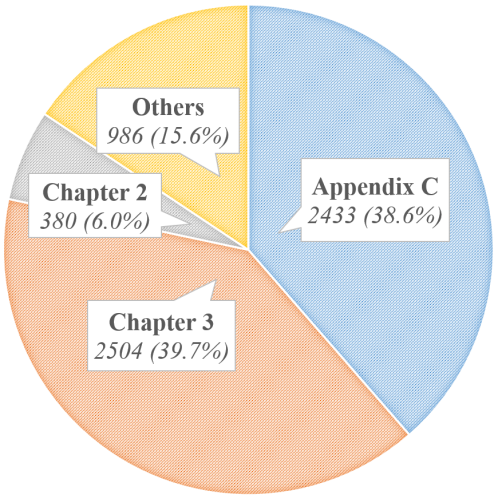
approach, and the studies by Bhat and Nagappan [5] show that the block coverage reached to 79–88% at unit test level in projects employing TDD. While Bhat and Nagappan's studies were carried out at Microsoft, some scientific software demands even higher levels of test coverage. Notably, the European Cooperation for Space Standardization requires a 100% test coverage at software unit level, and Prause *et al.* [35] collected experience from a space software project's developers who stated that 100% coverage is unusual and brings in new risks (e.g., additional costs). Nevertheless, the space software developers acknowledged that 100% coverage is sometimes necessary. For regression tests, the coverage to the code changes is of particular interest and importance.

Our work analyzes the coverage of SWMM unit and regression tests not only from the source code perspective, but also from the viewpoint of the user manual. Compared to business/IT software, scientific software tends to release authoritative and updated user manual intended for the software system's proper operation. The rest of this section reports the test coverage and discusses our study's limitations.

4.1. SWMM User Manual Coverage

Carver and his colleagues' study [43] reveals one of the pain points in scientific and engineering software development is the expertise gap caused by the complexity of the underlying domain's code and difficulty. Expert review is a method for uncovering usability issues and includes many aspects of the user interface that cause problems. The author of the SWMM user manual [37] is an environmental scientist who worked at the U.S. EPA. Consequently, SWMM user manual provides a partial expert view on the domain. The test cases, on the other hand, represent concrete ways that the software is executed. Mapping the tests to the user manual (partial expert view) may uncover the domain complexity not matched by tests, which in turn helps increase the diversity of the tests. Users may find issues not covered in the User's Manual and the this testing can help the software developers add this missing material to the manual. This is in line with Feldt *et al.*'s perspective [44] advocating that test diversity is important for software quality.

We manually mapped the SWMM tests to its version 5.1 user manual [37], and for validation and replication purposes, we share all our analysis data in the institutional digital preservation site Scholar@UC [31]. The 353-page



(a)

Others	**# of Tests**
Chapter 5	194(3.1%)
Chapter 11	149(2.4%)
Chapter 7	125(2.0%)
Appendix B	118(1.9%)
Chapter 6	111(1.8%)
Chapter 9	77(1.2%)
Chapter 4	75(1.2%)
Chapter 8	70(1.1%)
Chapter 1	32(0.5%)
Appendix D	23(0.4%)
Appendix A	12(0.2%)

(b)

Figure 4: **(a)** Mapping 2,953 unit tests to user manual chapters/appendices, and **(b)** Explaining the "Others" part of **(a)**.

user manual contains 12 chapters and 5 appendices.

Unit Tests: The method that we mapped unit tests to the user manual is keyword matching. We relied on the textual information in the test code statements and the developer's comments in the tests as a basis for such mappings. We performed the mappings manually because the textual information in the code statements and the developer's comments do not match 100% to the user manual, which needs human judgment. For example, in Figure 3, variables like "SM_SURFACE" and "SM_THICKNESS" in line #330 do not exist in the SWMM user manual. The comments in line # 310 indicate the tests belong to Lid (or LID meaning "low impact development") Control Bio Cell parameters, and yet keyword matching "bio cell" in the SWMM user manual returns no result. Nevertheless, the parameter "bio-retention Cells" exists in the SWMM user manual on page 69, one of the generic types of LID controls. This allows us to build a match between the tests to the SWMM user manual.

Our analysis shows that 14 chapters /appendices, or 82.4% ($\frac{14}{17}$), are covered by at least one of the 2,953 unit tests. Figure 4 shows the distributions of the unit tests over the 14 user manual chapters/appendices. Because one unit test may correspond to many chapters/appendices, the test total of Figure 4 is 6,303. The uncovered chapters are: "Printing and Copying" (Chapter 10), "Using Add-In Tools" (Chapter 12), and "Error and Warning Messages" (Appendix E). The error and warning messages are descriptive in nature, and printing, copying, and add-in tools require the devices and/or services external to SWMM. Due to these reasons, it is understandable that no unit tests correspond to these chapters/appendices.

Figure 4(a) shows that the unit tests predominantly cover "SWMM's Conceptual Model" (Chapter 3) and "Specialized Property Editors" (Appendix C). The close percentages, 38.6% and 39.7%, of these two parts are not accidental to us. In fact, they share 2,431 unit tests. We present a detailed look at these parts in Figure 5. Chapter 3 describes not only the configuration of the SWMM objects (e.g., conduits, pumps, storage units, etc.) but also the LID controls that SWMM allows engineers and planners to represent combinations of green infrastructure practices and to determine their effectiveness in managing runoff. The units presented in §3.2 ("Visual Objects"), §3.3 ("Non-Visual Objects"), and §3.4 ("Computational Methods") thus represent some of the core computations of SWMM. Consequently, unit tests are written for the computations except for the "Introduction" (§3.1) overviewing the

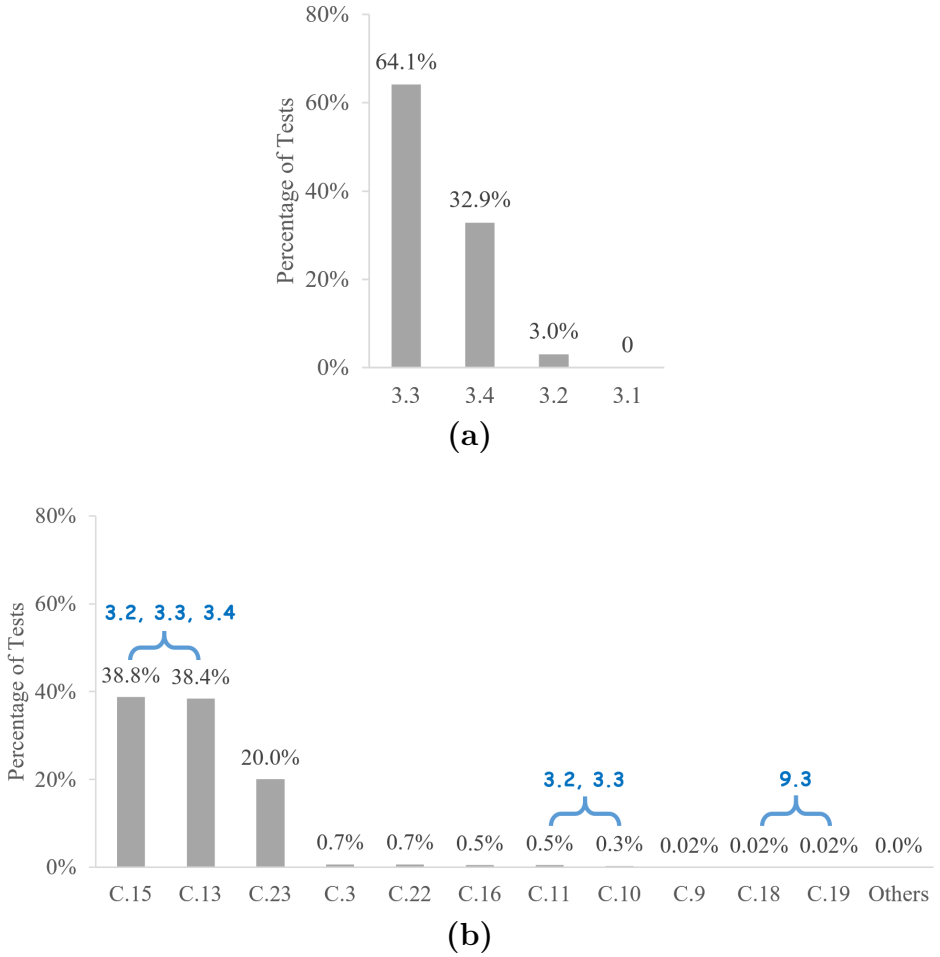


Figure 5: **(a)** Breakdowns of unit tests into Chapter 3 ("SWMM's Conceptual Model") of the user manual, and **(b)** Breakdowns of unit tests into Appendix C ("Specialized Property Editors") of the user manual.

Atmosphere, Land Surface, Groundwater, and Transport compartments of SWMM. Surprisingly, more tests are written for the non-visual objects than the visual objects, as shown in Figure 5(a). The visual objects (rain gages, subcatchments, junction nodes, outfall nodes, etc.) are those that can be arranged together to represent a stormwater drainage system, whereas non-visual objects (climatology, transects, pollutants, control rules, etc.) are used to describe additional characteristics and processes within a study area. One reason might be the physical, visual objects (§3.2) are typically combined, making unit tests (e.g., single tests per visual object) difficult to construct.

The non-visual objects (§3.3), on the other hand, express attributes of, or the rules controlling, the physical objects, which makes unit tests easier to construct. For example, two of the multiple-condition orifice gate controls are RULE R2A: "IF NODE 23 DEPTH $>$ 12 AND LINK 165 FLOW $>$ 100 THEN ORIFICE R55 SETTING $=$ 0.5" and RULE R2B: "IF NODE 23 DEPTH $>$ 12 AND LINK 165 FLOW $>$ 200 THEN ORIFICE R55 SETTING $=$ 1.0". For units like RULE R2A and RULE R2B, tests could be written to check whether the orifice setting is correct under different node and link configurations. Under these circumstances, the test oracles are *known* and are given in the user manual (e.g., orifice setting specified in the control rules).

During our manual mappings of the unit tests, we realize the interconnection of the user manual chapters/appendices. One example mentioned earlier is the connection between Chapter 3 and Appendix C. It turns out that such interconnections are not one-to-one, i.e., Appendix C connects to not only Chapter 3 but also to other chapters. In Figure 5(b), we annotate the interconnections grounded in the SWMM unit tests. For instance, §3.2, §3.3, and §3.4 are linked to §C.10 ("Initial Buildup Editor"), §C.11 ("Land Use Editor"), §C.13 ("LID Control Editor"), and §C.15 ("LID Usage Editor"), indicating the important role of LID plays in SWMM. Although only a very small number of unit tests connects §9.3 ("Time Series Results") with §C.18 ("Time Pattern Editor") and §C.19 ("Time Series Editor"), we posit more tests of this core time-series computation could be developed in a similar way as LID tests (e.g., by using the boost environment illustrated in Figure 3). A more general speculation that we draw from our analysis is that if some core computation has weak links with the scientific software system's parameters and properties (e.g., Appendix C of the SWMM user manual), then developing unit tests for that computation may require other environments and frameworks like CppTest or CPPUnit; investigating these hypotheses is part of our future research collaborations with the EPA's SWMM team.

Regression Tests: We performed an in-depth analysis of the 58 regression tests of SWMM, and provided the relevant analysis results of the eight regression tests in one folder in Table 4. The results of other folders' regression tests are similar to those reported in Table 4, and can be found in our data repository [31]. At the chapter level, the user manual coverage of six regression tests in Table 4 is 100%. Two tests, gate_control_2.inp and gate_control_3.inp, have 94% user manual coverage, and in both cases, the one chapter that is not covered is "Printing and Copying" (Chapter 10). Compared to the

Table 4: User manual coverage and parameter analysis of SWMM regression tests in the "update_v5111" folder; the statistics of the regression tests in the other five folders ("examples", "extran", "routing", "swc", and "user") are similar to the ones reported in this table. All our analysis data are accessible in [31].

Regression Test (User Manual Coverage)	**# (%) of Parameters in Code**	**# (%) of Input Parameters in User Manual**
bioretention.inp (100%)	66 (66 / 86 = 77%)	59 (59 / 66 = 89%)
catchment_as_outfall.inp (100%)	72 (72 / 93 = 77%)	61 (61 / 72 = 85%)
events_example.inp (100%)	84 (84 / 121 = 69%)	72 (72 / 84 = 86%)
gate_control_2.inp (94%)	64 (64 / 80 = 80%)	51 (51 / 64 = 80%)
gate_control_3.inp (94%)	62 (62 / 75 = 83%)	53 (53 / 62 = 85%)
ncdc_format.inp (100%)	57 (57 / 71 = 80%)	53 (53 / 57 = 93%)
porous_pavement.inp (100%)	65 (65 / 83 = 78%)	62 (62 / 65 = 95%)
rain_garden.inp (100%)	53 (54 / 71 = 75%)	48 (48 / 53 = 91%)

unit tests where the user manual coverage is 82.4%, the regression tests of SWMM have also covered "Using Add-In Tools" (Chapter 12) and "Error and Warning Messages" (Appendix E). The higher coverage indicates that more parameters, including add-in tools (e.g., Format, Rain Gage, etc.) and exception handling (e.g., Interval, Source, etc.), appear in regression tests as the focus is to compare different versions of the software. When defining unit tests, however, the SWMM developers would either face the oracle problem for specifying the expected outcome of add-in tools, or focus more exclusively on normal behaviors of the software than systematically and automatically checking exceptions. Taken all the 58 regression tests in our study together, the user manual coverage is 100%.

Table 4 also reports our further analysis of SWMM regression tests' parameters to examine how many actually appear in the non-comment, source code and how many appear as input parameters in the user manual. From the middle column of Table 4, 69–83% of the parameters are included in the code of SWMM's version 5.1.014. We believe the percentages here represent the realistic status of scientific software's regression testing. On one hand, the code evolution will unavoidably make some parameters in a large input space become obsolete, yet keeping them around may help to ensure the successful execution of older versions of the software. On the other hand, the 17–31% parameters not found in the code will not negatively impact regression testing, as the superfluous information presented by these parameters will simply be ignored when a newer version of SWMM is executed.

The rightmost column of Table 4 shows the extent to which the parameters appearing in the code are classified as input parameters in the user manual. The analysis here is based on our manual classification of the SWMM user manual where we separate the parameters into input and output groups. The manual classifications are part of our shared data [31] to facilitate validation and replication. Overall, Table 4 shows that 80–95% of the parameters are input parameters. Surprisingly, the other 5–20% are not output parameters, but do not appear in the user manual at all. We speculate that one reason may be the user manual [37], which was released in September 2015, becoming outdated; however, these parameters remain in the version 5.1.014 of SWMM. Another reason might be that a small proportion of the parameters defined in regression tests are *internal*, *non-user-facing* parameters, but are important for running the simulations and for comparing different versions of the software. Understanding these internal parameters is part of our future work. The analysis presented in Table 4 indicates the importance of considering the source code. Next, we turn our attention to code coverage of SWMM's unit and regression tests.

4.2. SWMM Codebase Coverage

There are a number of coverage measures commonly used for test-codebase analysis, e.g., Prause *et al.* [35] compared statement coverage to branch coverage in a space software project and showed that branch coverage tended to be lower if not monitored but could be improved in conjunction with statement coverage without much additional effort. For our analysis, We apply OpenCppCoverage, an open source test coverage tool [30], to compute code coverage at the statement level. For the code coverage analysis, the scope is focused on SWMM's computational engine (about 46,300 lines of code written in C/C++). Like the test-to-user-manual data, we also share our test-to-codebase analysis data in Scholar@UC [31].

Unit Tests: OpenCppCoverage measures code coverage for each file at the statement level when the test is executed. We run each unit test against each source code file of SWMM's computational engine. In total, there are 6,595 covered statements over the 16,611 statements in the tested code files. Thus, the statement-level code coverage of the 2,953 SWMM unit tests is 39.7%. Figure 6 includes five files with the highest code coverage. The coverage of "odesolve.c", "kinwave.c", and "hash.c" is higher than 90%. We speculate that the high code coverage reflects these computations' responsibility for the

File name	Covered Statement	Uncovered Statement	Total Statement	Coverage in Single File
surfqual.c	111	26	137	81%
project.c	534	95	629	85%
hash.c	61	6	67	91%
kinwave.c	82	4	86	95%
odesolve.c	106	5	111	95%

Figure 6: Code coverage for the top five SWMM files.

essential functions of the software: "odesolve.c" is integrated with adaptive step size control, "kinwave.c" is the wave flow routing function, and "hash.c" is used to create a hash table for string storage. Surprisingly, the coverage of "lid.c" is only 72%. It is a module that handles all data processing involving the Low Impact Development practices used to treat runoff for individual subcatchments within a project. As introduced in Section 4.1, LID is the primary part of Chapter 3 and Appendix C, which account for 78% of the user manual measurement. Hilton and his colleagues' study [16] on 47 projects show the average of code coverage is 75%, so unit tests in SWMM have relatively low code coverage (39.7%). One possible reason is that scientific software may be moderate in size; however, the complexity may be higher.

In line with our user manual analysis results, the code corresponding to the greatest number of unit tests involves runoff, including toposort.c, treatmnt.c, and runoff.c. Different from our user manual analysis where we speculated that control rules, such as RULE R2A and RULE R2B, would be among the subjects of unit testing, the actual tests have a strong tendency toward getters and setters. For each instance variable, a getter method returns its value while a setter method sets or updates its value. This is illustrated in Figure 3. Interestingly, we also observe a pattern of "getter-setter-getter" in the tests. In Figure 3, the test of lines #330–332 first gets swmm_getLidCParam, ensures that there is no error in getting the parameter value (line #331), and compares the value with the oracle (line #332). A minor change is made in the next test where the new "&db_value" is set to be 100, followed by checking if this instance of parameter setter is successful (line #334). The last test in the "getter-setter-getter" sequence immediately gets and checks the parameter value (lines #335–337). Our analysis confirms many instances of this "getter-setter-getter" pattern among the 2,953 unit tests.

It is clear that oracle exists in SWMM unit tests, and as far as the "getter-

setter-getter" testing pattern is concerned, two kinds of oracle apply: whether the code crashes (e.g., lines #331, #334, and #336 of Figure 3) and if the parameter value is close to pre-defined or pre-set value (e.g., lines #332 and #337 of Figure 3). One advantage of "getter-setter-getter" testing lies in the redundancy of setting a value followed immediately by getting and checking that value, e.g., swmm_setLidCParam with 100 and then instantly checking swmm_getLidCParam against 100. As redundancy improves reliability, this practice also helps with the follow-up getter's test automation. However, a disadvantage here is the selection of the parameter values. In Figure 3, for example, the oracle of 6 (line #332) may be drawn from SWMM input and/or observational data, but the selection of 100 seems random to us. As a result, the test coverage is low from the parameter value selection perspective, which can limit the bug detection power of the tests.

It is worth noting that the purpose of the "getter-setter-getter" pattern is not to test if these core computations results are invalid or out of range, but to test that these computation methods are normally operated and executable. For example, Figure 3 shows the parameter "SM_THICKNESS" in the surface layer which was being tested under open_swmm_model (0) (line #313). The same test code (line #330–337) was used in other test cases but under different SWMM models (e.g., open_swmm_model (1), open_swmm_model (2), etc.) in the same test file. This implies that the primary purpose of unit tests in SWMM is to test a certain parameter under different SWMM models, which can be applied normally.

A post from the SWMM user forum [1] provides a concrete situation of software failure related to specific parameter values. In this post, the user reported that: "The surface depth never even reaches 300 mm in the LID report file" after explicitly setting the parameters of the LID unit (specifically, "storage depth of surface layer" = 300 mm) to achieve the effect [1]. The reply from an EPA developer suggested a solution by changing: "either the infiltration conductivity or the permeability of the Surface, Soil or Pavement layers". Although these layers are part of "LID Controls", and even have their descriptions in §3.3.14 of the SWMM user manual [37], the testing coverage does not seem to reach "storage depth of surface layer" = 300 mm under different value combinations of the Surface, Soil or Pavement layers. We believe the test coverage can be improved when the design of unit tests builds more directly upon the SWMM user manual. There are 130 parameters with default values specified in the SWMM user manual, but only 53 parameters (40.8%) are included in the current unit tests. In Figure 7,

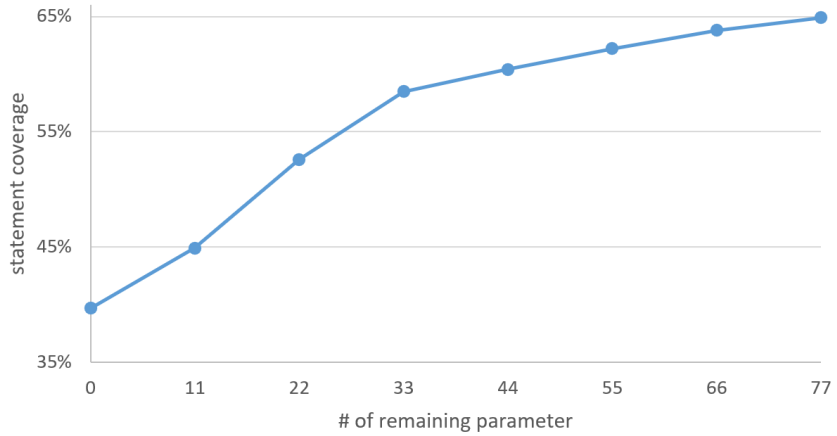


Figure 7: Unit test of the parameters with default values specified in the SWMM user manual

Table 5: Fifty-eight regression tests grouped by folders and then ranked by the folder-level Jaccard distance.

Rank	Folder	# of .inp Files	Average Jaccard Distance
1	swc	25	0.52
2	routing	5	0.41
3	extran	10	0.38
4	examples	5	0.37
5	user	5	0.36
6	update_v5111	8	0.28

we apply the "getter-setter-getter" pattern to create new unit tests with the remaining 77 parameters that are not currently included. The order of 77 parameters selection is based on the number of times they appear in the user manual (from greater to less). The result of Figure 7 shows the statement-level code coverage of SWMM would have been increased to 63.8%.

Regression Tests: The statement-level code coverage for the 58 .inp files ranges from 21.0–41.4%, with the average and medium coverage being 31.4% and 27.7% respectively. Compared to the 39.7% code coverage of the 2,953 SWMM unit tests, the regression tests have jointly covered 44.9% of the codebase. Similar to the user manual coverage analysis, regression tests have a higher coverage than unit tests over the source code as well.

In Table 5, we group the 58 regression tests by their folder structure,

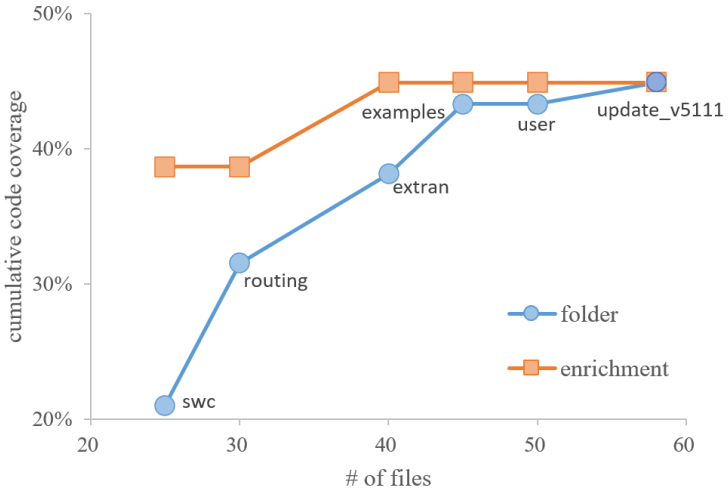


Figure 8: Comparing cumulative code coverage by regression tests' folder structure and by enrichment.

and further rank the folders by the average Jaccard distance [40] defined by the executed statements in a pair of tests (.inp files). To illustrate the Jaccard distance of A.inp and B.inp, let us assume when A.inp is executed, four statements in SWMM are covered: s1, s2, s3, and s4, and when B.inp is executed, three statements in SWMM are covered: s1, s2, and s5. Jaccard distance is defined by dividing the difference of the sizes of the union and the intersection of two sets by the size of the union [40]. Here, the executed statements' union of A.inp and B.inp is {s1, s2, s3, s4, s5}, and the intersection is {s1, s2}. Thus, the Jaccard distance of A.inp and B.inp is (5−2) / 5 = 0.60.

The Jaccard distance serves as a practical guide in terms of selecting and prioritizing regression tests. When testing resources are limited, one may choose the most diverse set of tests to run first, followed by less diverse sets. Therefore, the ranking presented in Table 5 can be used to determine which folder of regression tests to run, and which other folders to follow. We plot the cumulative code coverage by following this folder ranking in Figure 8. Running the 25 tests from the swc folder, for example, covers 21.0% of the code. Adding the 5 tests from the routing folder improves the cumulative code coverage to 31.5%, and so on.

Inspired by information foraging theory [28, 29], we could enrich the re-

ID	File	# of statement changes	Covered by the 58 regression tests
C1	dwflow.c	3	no
C2	swmm5.def	14	no
C3	surfqual.c	3	no
C4	dynwave.c	5	yes
C5	runoff.c	5	yes
C6	flowrout.c	1	no
C7	reports.c	5	yes
C8	funcs.h	2	yes
C9	kinwave.c	1	yes
C10	rdii.c	4	no
C11	rain.c	1	no
C12	lid.c	4	yes
C13	output.c	1	no
C14	node.c	3	no
C15	lidproc.c	5	no
C16	link.c	15	yes

```
        …
123     void updateNodeFlows(int i)
        {
        …
548     +     uniformLossRate *= barrels;
        …
        if ( q >= 0.0 )
        {
554     –     Node[n1].outflow += q + uniformLossRate;
554     +     Node[n1].outflow += q;
555     –     Node[n2].inflow  += q;
555     +     Node[n2].inflow  += q – uniformLossRate;
        }
        else
        {
559     –     Node[n1].inflow   -= q;
559     +     Node[n1].inflow   -= q + uniformLossRate;
560     –     Node[n2].outflow  -= q – uniformLossRate;
560     +     Node[n2].outflow  -= q;
561         …
```

Figure 9: Summarizing the statement-level code changes from SWMM's version 5.1.013 to version 5.1.014 (top) and an illustration of the specific changes of C4 (bottom).

gression tests by rearranging them based on parameter diversity as measured by the Jaccard distance. In Figure 8, we plot the effect of enrichment by grouping the same number of .inp files as the folder structure; however, the actual .inp files are reordered from the greatest to the least Jaccard distance. For example, the first 25 regression tests (and their average Jaccard distances) in the enriched environment are: swc3.inp (0.56), swc2.inp (0.51),

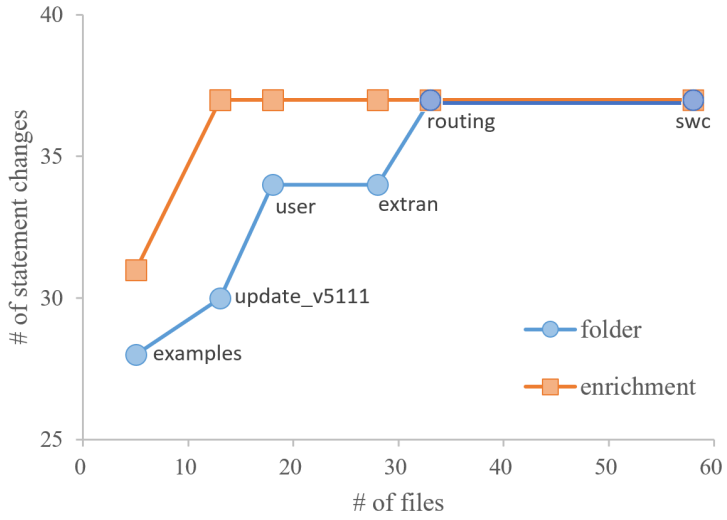


Figure 10: Comparing the average number of covered code changes by regression tests' folder structure and by enrichment.

test2.inp (0.50), ..., and extran3.inp (0.49). This group of regression tests has a cumulative code coverage of 38.7%. Adding the next 5 tests, as shown by the "enrichment" curve in Figure 8, does not lead to any change in the cumulative code coverage. The maximum coverage of 44.9% is achieved by the enrichment curve at the third group, i.e., after the first 40 regression tests are executed. In contrast, the folder structure achieves the 44.9% coverage at the six group after the 58 .inp files are tested.

While the results of Figure 8 show the enrichment effect over the *entire* codebase of SWMM, we are also interested in the specific changes made in consecutive SWMM versions. After all, it is these statement changes that regression testing is intended to check. We therefore identified 72 statement changes at the statement level between SWMM versions 5.1.013 and 5.1.014. The top of Figure 9 summarizes the size of each statement changes in SWMM files, and the bottom of Figure 9 illustrates a specific change (C4) where additions and deletions are annotated. It can be seen from the top of Figure 9 that 37 statement changes (51.4%) from version 5.1.013 to version 5.1.014 are covered by one or more regression tests.

When prioritizing regression tests for the actual statement changes, we believe the enrichment mechanism shall be different from the entire codebase. In particular, we shall find the most similar set of .inp files to a specific set

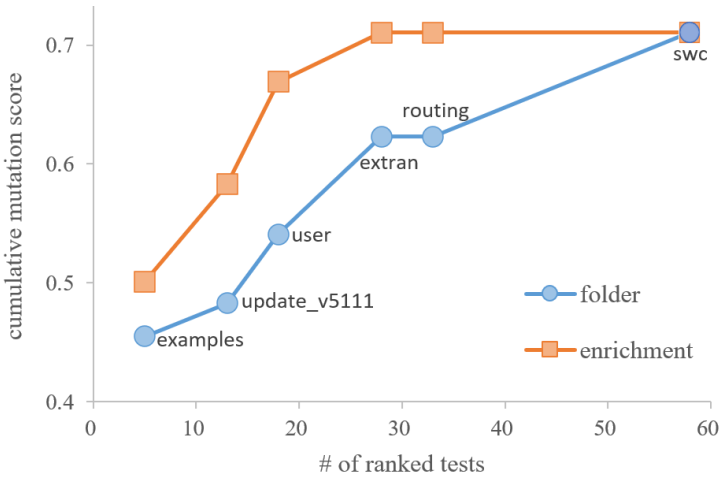


Figure 11: Cumulative mutation score by regression tests' folder structure and by enrichment.

of statement changes and execute that set first, followed by the second most similar set, and so forth. The use of Jaccard distance is tailored to the specific statement changes, and the sets of regression tests are ordered from the most similar to the least similar as far as test selection is concerned. There are 37 statement changes covered by the 58 regression tests. We find the closely related set of .inp files by measuring the closest Jaccard distance of the 37 statement changes and set of .inp files' covered statements. In Figure 10, the examples folder has the shortest Jaccard distance with the 37 statement changes, so it will be executed first, followed by the second similar set, and so forth. Although the enrichment effect of Figure 10 is not as prominent as that of Figure 8, the difference can be attributed to the relatively small number of code changes (72 statements) and the covered changes (37 statements).

Lu and Sireci [45] also point out most standardized tests are required by being executed within a specified time limit because sometimes the developer does not fully consider all test items. Our enhancement focus is to prioritize existing regression tests and achieve better test effects as early as possible. We use a cumulative mutation score to measure when the set of tests reach the maximum test effectiveness. We apply the mutation operators (arithmetic operator and relational operator replacement) in Visual Studio to generate the mutants, each containing a single fault. In total, 150 mutants are created by applying the mutation operators on the 37 changing statements from version 5.1.013 to version 5.1.014 covered by one or more

regression tests. In Figure 11, We compute an average cumulative mutation score by regression tests' folder structure and enrichment structure. The enrichment curve achieves the maximum cumulative mutation score of 0.71 at the fourth group, and the folder structure achieves the same mutation score at the sixth group after the full test files are executed.

Our analysis shows that enriching regression tests based on Jaccard distance has better coverage and test effectiveness than the tests' current folder structure. A practical implication is on prioritizing the .inp files to test: if broadly scoped changes are made, the diverse tests shall be executed first, but if only local changes are made, the tests closely related to the specific changes shall be run first.

4.3. Threats to Validity

We discuss some of the important aspects of our study that one shall take into account when interpreting our findings. A threat to construct validity is how we define tests. Our work on SWMM unit tests is influenced by the boost environment, as illustrated in Figure 3. While our units of analysis—tests, test cases, and test suites—are consistent with what boost defines and how the developers apply boost, the core construct of "tests" may differ if boost evolves or the SWMM developers adopt other test development environments. Within boost itself, for instance, BOOST_AUTO_TEST_CASE may require different ways to define and count tests than BOOST_FIXTURE_TEST_CASE shown in Figure 3.

An internal validity threat is our manual mapping of the SWMM unit and regression tests to the user manual. Due to the lack of traceability information from the SWMM project, our manual effort is necessary in order to understand the coverage of the considered tests. Our current mapping strategy is driven mainly by keywords, i.e., we matched keywords from the tests with the user manual contents. Two researchers independently performed the manual mappings of a randomly chosen 200 tests and achieved a high level of inter-rater agreement (Cohen's κ=0.77). We attributed this to the comprehensive documentation of SWMM tests and user manual. The disagreements of the researchers were resolved in a joint meeting, and three researchers performed the mappings for the remaining tests.

Several factors affect our study's external validity. Our results may not generalize to other kinds of SWMM testing (integration testing, acceptance testing, etc.), to the tests shared internally among the SWMM developers, and to other scientific software with different size, complexity, purposes, and

testing practices. As for conclusion validity and reliability, we believe we would obtain the same results if we repeated the study. In fact, we publish all our analysis data in our institution's digital preservation repository [31] to facilitate reproducibility, cross validation, and future expansions.

5. Conclusions

Testing is one of the cornerstones of modern software engineering [11]. Scientific software developers, however, face challenges like the oracle problem when performing testing [20]. In this paper, we report our analysis of the unit tests and the regression tests written and released by the EPA's SWMM developers. For the 2,953 SWMM unit tests that we identified, the statement-level code coverage is 39.7% and the user manual coverage is 82.4%. We further analyzed 58 regression tests and found their code coverage is 44.9% and the user manual coverage is almost 100%.

Our results show that oracle *does* exist in at least two levels: whether the code crashes and if the returned value of a computational unit is close to the expectation. In addition to relying on historical data to define the test oracle [24, 25], our study uncovers a new "getter-setter-getter" testing pattern, which helps alleviate the oracle problem by setting a parameter value and then immediately getting and checking it. This practice, though innovative, can be further improved by incorporating the user manual in developing tests and by automating parameter value selection to increase coverage. The increased code coverage can also be obtained by diversifying regression tests. Our work suggests that, when designing a set of regression tests, greater distance is preferred when broad changes are made and even superfluous information may not have negative impacts on regression testing.

Our future work will explore parameter dependencies as they relate to unit and regression testing. In addition to source code analysis like control flow or data flow, the dependencies might be identified by the co-appearance patterns in a rich set of tests. We also plan to build initial tooling to interrelate tests and user manual or other software documentation on the basis of keyword matching drawn from our current operational insights. Our goal is to better support scientists in improving testing practices and software quality.

Disclaimer
The U.S. Environmental Protection Agency, through its Office of Research and Development, partially funded and collaborated in, the research de-

scribed herein. It has been subjected to the Agency's peer and administrative review and has been approved for external publication. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the Agency, therefore, no official endorsement should be inferred. Any mention of trade names or commercial products does not constitute endorsement or recommendation for user.

Acknowledgment
This work is supported in part by the U.S. National Science Foundation (Award CCF 1350487) and the U.S. Environmental Protection Agency.

References

[1] B. Adei, R. Dickinson, and L. A. Rossman. Some observations on LID output. https://www.openswmm.org/Topic/4214/some-observations-on-lid-output Last accessed: January 2021.

[2] G. Baradhi and N. Mansour. A comparative study of five regression testing algorithms. In *Australian Software Engineering Conference*, pages 174–182, 1997.

[3] J. Barco, K. M. Wong, and M. K. Stenstrom. Automatic calibration of the U.S. EPA SWMM model for a large urban catchment. *Journal of Hydraulic Engineering*, 134(4): 466–474, 2008.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, 2015.

[5] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *International Symposium on Empirical Software Engineering*, pages 356–363, 2006.

[6] L. Brader, H. Hilliker, and A. C. Wills. Chapter 2 Unit Testing: Testing the Inside. *Testing for Continuous Delivery with Visual Studio 2012*, Microsoft Patterns & Practices, 2013.

[7] S. S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16(2): 238–247, 1990.

[8] E. Burszta-Adamiak and M. Mrowiec. Modelling of green roofs' hydrologic performance using EPA's SWMM. *Water Science & Technology*, 68(1): 36–42, 2013.

[9] B. Dawes and D. Abrahams. Boost C++ libraries. https://www.boost.org Last accessed: January 2021.

[10] J. Ding, D. Zhang, and X-H. Hu. An application of metamorphic testing for testing scientific software. In *International Workshop on Metamorphic Testing*, pages 37–43, 2016.

[11] P. F. Dubois. Testing scientific programs. *Computing in Science & Engineering*, 14(4): 69–73, 2012.

[12] S. Easterbrook and T. C. Johns. Engineering the software for understanding climate change. *Computing in Science & Engineering*, 11(6): 65–74, 2009.

[13] P. E. Farrell, M. D. Piggott, G. J. Gorman, D. A. Ham, C. R. Wilson, and T. M. Bond. Automated continuous verification for numerical simulation. *Geoscientific Model Development*, 4(2): 435–449, 2011.

[14] R. Garner. NASA's Parker Solar Probe sheds new light on Sun. https://www.nasa.gov/feature/goddard/2019/nasas-parker-solar-probe-sheds-new-light-on-the-sun Last accessed: January 2021.

[15] R. M. Hierons. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, 38(3): 629–641, 2012.

[16] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *International Conference on Automated Software Engineering*, pages 53–63, 2018.

[17] K. Hinsen. The approximation tower in computational science: why testing scientific software is difficult. *Computing in Science & Engineering*, 17(4): 72–77, 2015.

[18] L. Hochstein and V. R. Basili. The ASC-alliance projects: a case study of large-scale parallel scientific code development. *Computer*, 41(3): 50–58, 2008.

[19] S. Kamble, X. Jin, N. Niu, and M. Simon. A novel coupling pattern in computational science and engineering software. In *International Workshop on Software Engineering for Science*, pages 9–12, 2017.

[20] U. Kanewala and J. M. Bieman. Testing scientific software: a systematic literature review. *Information & Software Technology*, 56(10): 1219–1232, 2014.

[21] D. Kelly, R. Gray, and Y. Shao. Examining random and designed tests to detect code mistakes in scientific software. *Journal of Computational Science*, 2(1): 47–56, 2011.

[22] D. Kelly, S. Thorsteinson, and D. Hook. Scientific software testing: analysis with four dimensions. *IEEE Software*, 28(3): 84–90, 2011.

[23] I. M. Kourtis, G. Kopsiaftis, V. Bellos, and V. A. Tsihrintzis. Calibration and validation of SWMM model in two urban catchments in Athens, Greece. In *International Conference on Environmental Science and Technology*, 2017.

[24] X. Lin, M. Simon, and N. Niu. Exploratory metamorphic testing for scientific software. *Computing in Science & Engineering*, 22(2): 78–87, 2020.

[25] X. Lin, M. Simon, and N. Niu. Hierarchical metamorphic relations for testing scientific software. In *International Workshop on Software Engineering for Science*, pages 1–8, 2018.

[26] X. Lin, M. Simon, and N. Niu. Releasing scientific software in GitHub: a case study on SWMM2PEST. In *International Workshop on Software Engineering for Science*, pages 47–50, 2019.

[27] J. Mayer. On testing image processing applications with statistical methods. In *Software Engineering*, pages 69–78, 2005.

[28] N. Niu, X. Jin, Z. Niu, J.-R. C. Cheng, L. Li, and M. Y. Kataev. A clustering-based approach to enriching code foraging environment. *IEEE Transactions on Cybernetics*, 46(9): 1962–1973, 2016.

[29] N. Niu, A. Mahmoud, and G. Bradshaw. Information foraging as a foundation for code navigation. In *International Conference on Software Engineering*, pages 816–819, 2011.

[30] OpenCppCoverage. An Open Source Code Coverage Tool for C++ Under Windows. https://github.com/OpenCppCoverage Last accessed: January 2021.

[31] Z. Peng, X. Lin, and N. Niu. Data of SWMM Unit and Regression Tests. http://dx.doi.org/10.7945/zpdh-7a44 Last accessed: January 2021.

[32] Z. Peng, X. Lin, and N. Niu. Unit tests of scientific software: a study on SWMM. In *International Conference on Computational Science*, pages 413–427, 2020.

[33] J. Pitt-Francis, M. O. Bernabeu, J. Cooper, A. Garny, L. Momtahan, J. Osborne, P. Pathmanathan, B. Rodriguez, J. P. Whiteley, and D. J. Gavaghan. Chaste: using agile programming techniques to develop computational biology software. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366: 3111–3136, 2008.

[34] D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: lessons learned from ASCI. *International Journal of High Performance Computing Applications*, 18(4): 399–416, 2004.

[35] C. R. Prause, J. Werner, K. Hornig, S. Bosecker, and M. Kuhrmann. Is 100% test coverage a reasonable requirement? Lessons learned from a space software project. In *International Conference on Product-Focused Software Process Improvement*, pages 351–367, 2017.

[36] H. Remmel, B. Paech, P. Bastian, and C. Engwer. System testing a scientific framework using a regression-test environment. *Computing in Science & Engineering*, 14(2): 38–45, 2012.

[37] L. A. Rossman. Storm Water Management Model User's Manual Version 5.1. https://www.epa.gov/sites/production/files/2019-02/documents/epaswmm5_1_manual_master_8-2-15.pdf Last accessed: January 2021.

[38] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9): 805–824, 2016.

[39] J. Sheehan. Federally funded research results are becoming more open and accessible. https://digital.gov/2016/10/28/federally-funded-research-results-are-becoming-more-open-and-accessible/ Last accessed: September 2020.

[40] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Pearson, 2005.

[41] J, Temprano, Ó. Arango, J. Cagiao, J. Suárez, and I. Tejero. Stormwater quality calibration by SWMM: a case study in Northern Spain. *Water SA*, 32(1): 55–63, 2005.

[42] United States Environmental Protection Agency. Storm Water Management Model (SWMM). https://www.epa.gov/water-research/storm-water-management-model-swmm Last accessed: January 2021.

[43] J. Carver, RP. Kendall, SE. Squires, and DE. Post. Software development environments for scientific and engineering software: A series of case studies. In *29th International Conference on Software Engineering*, pages 550-559, 2007.

[44] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation*, pages 223-233, 2016.

[45] Y. Lu and SG. Sireci. Validity issues in test speededness. *Educational Measurement: Issues and Practice,* 26(4): 29-37, 2007.