System Verification and Validation Plan for ROC: Software estimating the radius of convergence of a power series

John M Ernsthausen

December 24, 2020

Contents

1 Symbols, Abbreviations, and Acronyms **iii**

2 Introduction **1**

3 General Information **1**
- 3.1 Objectives . . . 1
- 3.2 Relevant Documentation . . . 1

4 Verification and Validation Plan **1**
- 4.1 Verification and Validation Team . . . 1
- 4.2 SRS Verification Plan . . . 2
- 4.3 Design Verification Plan . . . 2
- 4.4 Implementation Verification Plan . . . 2
  - 4.4.1 Dynamic testing . . . 2
  - 4.4.2 Static testing . . . 3
  - 4.4.3 Continuous integration . . . 4
  - 4.4.4 Parallel testing . . . 4
  - 4.4.5 Comparison with closed-form solution . . . 4
- 4.5 Automated Testing and Verification Tools . . . 5
- 4.6 Software Validation Plan . . . 5

5 System Test Description **5**
- 5.1 Parallel testing . . . 5
- 5.2 Tests for Functional Requirements . . . 5
- 5.3 Tests for Nonfunctional Requirements . . . 5
  - 5.3.1 Timing . . . 5
  - 5.3.2 Accuracy . . . 6
- 5.4 Traceability Between Test Cases and Requirements . . . 7

6 Unit Test Description **7**

List of Tables

1 Test cases . . . 8

Revision History

Date	Version	Notes
29 October 2020	1.0	First submission
24 December 2020	2.0	Second submission

1 Symbols, Abbreviations, and Acronyms

Symbols, abbreviations, and acronyms applicable to ROC are enumerated in Section 1 of the Software Requirements Document (SRS) (Ernsthausen, 2020c).

2 Introduction

This document provides a verification and validation plan for developing ROC. This plan includes dynamic testing and static testing. Functional requirement will be tested using parallel testing techniques, and nonfunctional requirements will test accuracy and timing. Testing on ROC is fully automated up to applying DevOps techniques available through our public repository on GitHub.

3 General Information

The scope of this ROC project includes three term analysis of primary real poles, six term analysis of primary pair of complex conjugate poles, and top line analysis. ROC does not yet include an analysis for essential singularities.

Top-line analysis always applies to any power series (1). It resolves a series anaylsis exhibiting secondary singularities. However top line analysis can be inaccurate.

3.1 Objectives

Our objective is correctness, accuracy, and timing. ROC is an idea for an academic research project. The impression from this software will guide future decisions about this idea as a viable research project.

Every effort will be made to complete every aspect promised in this documentation. However, as a research project, it is more important to discover missing elements in the work of other authors such as Chang and Corliss (1982) than to reproduce a known implementation. As a result, some promised work is expected to be accomplished after the course CSE 741 has completed.

3.2 Relevant Documentation

Relevant documentation includes the authors Software Requirements Specification (SRS) (Ernsthausen, 2020c), the authors Module Guide (MG) (Ernsthausen, 2020a), and the authors Module Interface Specification (MIS) (Ernsthausen, 2020b).

4 Verification and Validation Plan

Verification and Validation of ROC includes automated testing at the module level, the system level, and integration level. This document will additionally propose continuous integration.

4.1 Verification and Validation Team

The ROC team includes author John Ernsthausen, fellow students Leila Mousapour, Salah Gamal aly Hessien, Liz Hofer, and Xingzhi Liu as well as Professors Barak Shoshany, Spencer

Smith, George Corliss, and Ned Nedialkov. The author appreciates the helpful comments and superior guidance on this project.

4.2 SRS Verification Plan

The SRS document Verification and Validation Plan for ROC will be peer-reviewed by domain expert Leila Mousapour and secondary reviewer Salah Gamal aly Hessien. Prof. Spencer Smith the course instructor and my supervisor Ned Nedialkov will review the SRS document.

The SRS document will be published to GitHub. Defects will be addressed with issues on the GitHub platform.

4.3 Design Verification Plan

The Design documents MG and MIS plan for ROC will be peer-reviewed by domain expert Leila Mousapour and secondary reviewer Xingzhi Liu. Prof. Spencer Smith the course instructor and my supervisor Ned Nedialkov will also review these Design documents.

The MG and MIS documents will be published to GitHub. Defects will be addressed with issues on the GitHub platform.

4.4 Implementation Verification Plan

In the development of ROC, all verification will be automatic.

4.4.1 Dynamic testing

Dynamic testing requires the program to be executed. For example, unit test cases are run, and the results are checked against expected behaviour. Dynamic testing of ROC will be through unit tests, memory testing, profiling, and code coverage.

Unit testing (Gtest/GMock): The Implementation Verification Plan for ROC includes automated testing at the module level. For testing at the module level, the author plans to follow the Test Driven Development software development practices as described by Langr (2013) for *test driving*. Test driving is an interactive process for software development where tests are determined as part of the software development process. However, on a high level, some unit tests will be offered in the sequel.

Let us recall test driving Langr (2013). Test driving results in unit tests. A unit test verifies the *behavior* of a code unit, where a *unit* is the smallest testable piece of an application.

A single unit test consists of a descriptive name and a series of code statements conceptually divided into four parts

1. (Optional) statements that set up a context for execution
2. One or more statements to invoke the behavior to be verified

3. One or more statements to verify the expected outcome

4. (Optional) cleanup statements

The first three parts are referred to as *Given-When-Then*. *Given* a context, *When* the test executes some code, *Then* some behavior is verified.

Test Driven Development is used to test drive new behavior into the code in small increments. To add a new piece of behavior into the system, first write a test to define that behavior. The existence of a test that will not pass drives the developer to implement the corresponding behavior. The increment should be the smallest meaningful amount of code, one or two lines of code and one assertion.

Memory testing (valgrind): Memory leaks as result from un-freed dynamically allocated memory. Valgrind is a tool to automatically detect memory leaks and a variety of memory errors. We will engage valgrind to remove all the memory errors this checker flags.

Valgrind will be run on the unit testing test suite.

Profiling (gprof): GCC includes a tool called gprof, which can be used to analyze the performance of ROC. Gprof will list the functions executed by ROC, along with the percentage of total runtime, number of method calls, and time spent in each method. Gprof is used to help performance.

Using gprof, ROC can reveal the parts of itself that take up the most execution time, without needing to do any manual measurements. This will help me to decide which methods to focus on when I attempt to optimize my code.

Gprof will be run on stand alone examples to determine performance for example in using ROC as a stepsize controller in the solution process on an ordinary differential and differential-algebraic equation.

Code coverage (gcov): GCC includes a tool called gcov, which can be used to reveal the lines of code used in the execution of a program. Unit testing is working whenever small increments of code are written when driving into the codebase the tests intended behavior. Gcov tests the assumption that all the code written for that unit test is used for that behavior.

Gcov reveals the code used/not used, and its output indicates the percentage of code used per directory and per file.

Gcov will be run against the test suite.

4.4.2 Static testing

Static testing does not involve program execution. Testing techniques simulate the dynamic environment. Static testing includes syntax checking.

Linting (clang-format): The code itself should be well formatted. Clang-format is a tool that keeps source code formatted to a specification. My specification is in the file ROC/.clang-format. Formats can be customized.

Clang-format will be run against my code base before I push my code to GitHub.

4.4.3 Continuous integration

Continuous integration enables more than one developer to independently drive new behavior into a code base and maintain a working codebase. Without continuous integration and with developers impeccably practicing sound testing philosophies, a code base integration between the developers can fail on merge. Continuous integration asks for frequent merges so that developers can be assured of integration compatibility.

TravisCI offers a working development server environment that will run unit tests, integration tests, static analysis, profile performance, extract documentation, and update the project web-page after developers integrate their code into a shared repo. It is expected that developers will merge their code into the shared repo several times a day. TravisCI offers MacOS, Windows, and Linux development environments for portability testing to eliminate the "it works on my machine" problem.

Continuous integration for ROC is offered through GitHub, TravisCI, and Coveralls. My TravisCI specification is in the file ROC/.travis.yml and a badge on my repo page indicates that my tests are passing/failing. I have not yet figured out how to get the code coverage server Coveralls to work for ROC. However the badge for Coveralls is also on the ROC repo page.

4.4.4 Parallel testing

In parallel testing, one compares to other programs. ROC includes automated testing at the system level and integration level. The code output will be compared with the output of DRDCV developed by Chang and Corliss (1982) for validation at the integration level. More on this topic is in the sequel.

4.4.5 Comparison with closed-form solution

Yes I do this too, in my unit tests.

For example, I expand $f(t) = 1.0/(1.0 - t)$ at $t = -1.0$ into its TS with scaling $h = 0.1$. I then use 3TA to recover the $R_c = 2$ to accuracy of $9.68115e - 14$ and $\mu = 1$ to accuracy of $1.1887e - 12$. See the unit test

$$\text{ThreeTermAnalysisOf.TaylorSeriesAtNegativeOneRealPoleAtOneWithScalingTenthAlphaOne} \tag{1}$$

I do this for a collection of μ and scaling h. I build similar tests for 6TA and TLA.

4.5 Automated Testing and Verification Tools

Automated Testing and Verification Tools will be extensively used in the development of ROC for automation at the module level, the system level, and integration level. These tools include git a distributed version-control system for tracking changes in source code during software development, cmake for build automation, gtest/gmock as a unit testing framework, clang-format for consistent code style formatting, valgrind to identify memory leaks, gcov for code coverage, and gprof as a profiler.

This document will additionally propose to branch out into three new areas for the author. In the DevOps arena, continuous integration will be pursued with TravisCI. The continuous integration tool should send an email confirming the success state of integrating the new code into production.

4.6 Software Validation Plan

Validation is the validation of the requirements. Validation compares experimental data to output from ROC to confirm or reject the problem model. Validation considers the applicability of the equations and assumptions to the problem space. However, ROC is not modelling a physical problem. Thus a Software Validation Plan is not applicable to this project.

5 System Test Description

System tests are about the public interface.

5.1 Parallel testing

ROC includes automated testing at the system level. The code output will be compared with the output of DRDCV developed by Chang and Corliss (1982) for validation at the integration level. Parallel testing will be used in nonfunctional testing.

5.2 Tests for Functional Requirements

ROC does not have System Functional Requirements at this time.

5.3 Tests for Nonfunctional Requirements

5.3.1 Timing

Requirement NFR1 says that ROC should execute as fast as the Chang and Corliss (1982) software DRDCV. The following tests represent a comparison between ROC and DRDCV.

Proportion of time spent finding the stepsize

1. Layne-Watson (Watson, 1979)

   Input: Load a file with the Taylor series solution of the Layne-Watson problem. Each time the local initial value problem was solved there will be a Taylor series of length N and a scale h, which are the required inputs for ROC and DRDCV.

   Output: Difference between time for ROC to solve the problem and time for DRDCV to solve the problem.

   Test Case Derivation: Two techniques resolving the same data.

   How test will be performed: Automatically.

2. Planetary-Motion (Enright and Pryce, 1987)

   Input: Load a file with the Taylor series solution of the Planetary-Motion problem. Each time the local initial value problem was solved there will be a Taylor series of length N and a scale h, which are the required inputs for ROC and DRDCV.

   Output: Difference between time for ROC to solve problem and time for DRDCV to solve problem.

   Test Case Derivation: Two techniques resolving the same data.

   How test will be performed: Automatically.

5.3.2 Accuracy

Requirement NFR2 says that the new top line analysis in ROC should compute a R_c and μ which is close to the values computed with the Chang and Corliss (1982) algorithm DRDCV. This test treats DRDCV as a pseudo oracle. The comparison will be carried out on the process of solving a DAEIVP by the TS method.

Accuracy in finding the stepsize

1. Layne-Watson (Watson, 1979)

   Input: Load a file with the Taylor series solution of the Layne-Watson problem. Each time the local initial value problem was solved there will be a Taylor series of length N and a scale h, which are the required inputs for ROC and DRDCV.

   Output: Each time the local initial value problem was solved, find R_c and μ with ROC and DRDCV. Compare the results. Expect the results to compare within 10%.

   Test Case Derivation: Two techniques resolving the same data.

   How test will be performed: Automatically.

2. Planetary-Motion (Enright and Pryce, 1987)

    Input: Load a file with the Taylor series solution of the Planetary-Motion problem. Each time the local initial value problem was solved there will be a Taylor series of length N and a scale h, which are the required inputs for ROC and DRDCV.

    Output: Each time the local initial value problem was solved, find R_c and μ with ROC and DRDCV. Compare the results. Expect the results to compare within 10%.

    Test Case Derivation: Two techniques resolving the same data.

    How test will be performed: Automatically.

5.4 Traceability Between Test Cases and Requirements

In each test, the requirement supported by the test case is stated.

However I would find a traceability table difficult to present as I have 158 tests, difficult to maintain, and a duplication of work. I wouldn't find this table useful. Here's why.

In the Module Guide there is "Table 2: Trace Between Requirements and Modules". Here I'm thinking of Module as a Parnas Module, a work order. For every work order, I Test Drive the behavior given in the work order into my code base. All of the behaviors required by the module must have corresponding testing code. I then do code coverage on my tests. If I have 100% code coverage of my tests and all the required behaviors are covered in the module. Then by my testing, I automatically have the functionality of the traceability table in discussion making the traceability table unnecessary work.

6 Unit Test Description

To implement each module required (so far) 158 unit tests. The tests are described in natural language through the tests "TestCase.TestName" combination. The tests are grouped by module in the directory roc/test. I enumerate the TestCases used in Table 1 that test all functional requirements. Especially notice the TestNames related to the inputs should the reader run the test suite. These tests demonstrate the exceptions thrown whenever a user error has occurred.

Number of Tests	TestCase Name
7	TestThatDDIST2
5	TestThatDNRM2
1	TestExceptions
3	TestThatIO
6	TestThatMathext
23	TestThatMatrix
10	TestThatCPPQR
10	TestThatHouseholderG
10	TestThatHouseholderL
7	TestThatQRF
8	TestThatQRS
9	TestThatQR
22	SixTermAnalysisOf
21	ThreeTermAnalysisOf
4	TestThatTopLine
12	TestThatVectorF

Table 1: Test cases

References

YF Chang and G Corliss. Solving ordinary differential equations using Taylor series. *ACM TOMS*, 8(2):114–144, 1982.

Wayne H. Enright and John D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Transactions on Mathematical Software*, 13(1):1–27, 1987.

J.M. Ernsthausen. Module guide for ROC: Software estimating the radius of convergence of a power series. `https://github.com/JohnErnsthausen/roc/blob/master/docs/MG/MG.pdf`, 2020a.

J.M. Ernsthausen. Module interface specification for ROC: Software estimating the radius of convergence of a power series. `https://github.com/JohnErnsthausen/roc/blob/master/docs/MIS/MIS.pdf`, 2020b.

J.M. Ernsthausen. Software requirements specification for ROC: Software estimating the radius of convergence of a power series. `https://github.com/JohnErnsthausen/roc/blob/master/docs/SRS/SRS.pdf`, 2020c.

Jeff Langr. *Modern C++ programming with test-driven development.* The Pragmatic Programmers, LLC, 2013.

L.T. Watson. A globally convergent algorithm for computing fixed points of c^2 maps. *Appl. Math. Comput.*, 5:297–311, 1979.