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

John M Ernsthausen

December 20, 2020

Contents

Revision History

Date	Version	Notes
29 October 2020	1.0	First submission
22 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, 2020).

2 Introduction

This document provides a verification and validation plan for developing ROC.

3 General Information

The scope of this ROC project is limited to top-line analysis. Top-line analysis always applies to any power series (1). It resolves situations where secondary singularities are less distinguishable from primary singularities. However it is less accurate. It does have a convergence analysis (Chang and Corliss, 1982).

Also add 3TA and 6TA.

3.1 Objectives

Our objective it to implement Module **findrc** and Module **findrc** via *test driving* in C++. The implementation must satisfy the requirements enumerated in Ernsthausen (2020).

Assume that the assumptions are satisfied. Recall the requirements.

In a nutshell these requirements validate and verify ROC.

3.2 Relevant Documentation

Relevant documentation includes the authors Software Requirements (SRS) Document (Ernsthausen, 2020), the authors Module Guide (MG, to be written), and the authors Module Interface Specification (MIS, to be written).

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

Dynamic testing through unit tests (Gtest/GMock), memory testing (valgrind), profiling (gprof), and code coverage (gcov).

Static testing though linting (clang-format).

Continuous integration through GitHub, TravisCI, and Coveralls.

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.

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): Code free of memory leaks.

Profiling (gprof): Number of method calls and time spent in each method. Used to help performance.

Code coverage (gcov): Reveals the code used/not used. Percentage of code used per directory and per file.

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 should be well formatted. Formats can be customized in XXX.

4.4.3 Continuous integration

- Information available on Wikipedia
- Developers integrate their code into a shared repo frequently (multiple times a day)
- Each integration is automatically accompanied by regression tests and other build tasks
- Build server
  - Unit tests
  - Integration tests
  - Static analysis
  - Profile performance
  - Extract documentation
  - Update project web-page
  - Portability tests
  - etc.

- Avoids potentially extreme problems with integration when the baseline and a developer's code greatly differ
- Eliminates the "it works on my machine" problem
- Package dependencies with your apps
- A container for lightweight virtualization
- Not a full VM

Continuous integration through GitHub, TravisCI, and Coveralls.

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.

4.4.5 Comparison with closed-form solution

Yes I do this too, in my unit tests.

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 as a unit testing framework, clang-format for consistent code style formatting, and valgrind to identify memory leaks.

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. The author will explore the application of linters and metrics for code coverage.

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.

Sample Functional System Testing

- Requirements: Determines if the system can perform its function correctly and that the correctness can be sustained over a continuous period of time
- Regression: Determines if changes to the system do not invalidate previous positive testing results
- Error Handling: Determines the ability of the system to properly process incorrect transactions
- Manual Support: Determines that the manual support procedures are documented and complete, where manual support involves procedures, interfaces between people and the system, and training procedures
- Parallel: Determines the results of the new application are consistent with the processing of the previous application or version of the application

Sample Nonfunctional System Testing

- Stress testing - Determines if the system can function when subject to large volumes
- Usability testing
- Performance measurement

5.1 Tests for Functional Requirements

5.1.1 Parallel testing

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.

And see Nonfunctional tests.

5.2 Tests for Nonfunctional Requirements

5.2.1 Timing

Requirement (R23) 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.

5.2.2 Accuracy

Requirement (R26) 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.

5.3 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.

The Software Development theories of Test Driving recommend running all test and observing that all tests pass. The interest of the third party in "Let's say a third party person is only interested in testing the R3" should be highly discouraged.

6 Unit Test Description

6.0.1 Input testing

User misuses the inputs: Length should be negative. Ensure the program gives an error.

Input testing in these input requirements: (R1) Input acquisition via hardware. (R2) Input acquisition via software. (R3) Validate input format. (R4) Validate input type.

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. 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, 2020.

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.