A Fourier Series Library: System Verification and Validation Plan for FSL

Bo Cao

December 14, 2019

1 Revision History

Date	Version	Notes
Oct. 28, 2019	0.99	First draft.
Dec. 5, 2019	1.0	Final version.

Contents

2 Symbols, Abbreviations and Acronyms

Some symbols, abbreviations and acronyms are defined in the Software Requirements Specification(SRS) document[1]. For simplicity and maintainability, they are not redefined here. Readers shall refer to the CA documents when a certain item is not defined here.

symbol	**description**
T	Test

[1]This document is available at https://github.com/caobo1994/FourierSeries/blob/master/docs/SRS/SRS.pdf.

3 General Information

This document provides an overview of the Verification and Validation (VnV) plan for the Fourier Series Library (FSL). It lays out the purpose, methods, and test cases for the VnV procedure.

3.1 Summary

The library to be tested is called the Fourier Series Library (FSL). This library performs a set of computations, transformations, and/or input/output at the request of the library user.

3.2 Objectives

The intended objective of the VnV procedure is to verify that this library has generally met the requirements described in the CA document. These requirements include the functional requirements (FRs) and the non-functional requirements (NFRs).

Note that if a small part of the NFRs has not been met, the library is still acceptable when the not-met NFRs' impact has been analyzed and deemed non-essential.

3.3 Relevant Documentation

As we said before, this document relies on the CA document. This document is also the base of the Unit Test Plan document.

4 Plan

This section lists the plan of the VnV of the FSL library.

- subsection 4.1 introduces the Validation and Verification Team of this library, including the team members and their duties.
- subsection 4.2 outlines the plan for the verification of the SRS.
- subsection 4.3 outlines the plan for the verification of the design of this libary.

- subsection 4.4 outlines the plan for the verification of the library's implementation.

- subsection 4.5 outlines the validation plan of the library based on the pseudo-oracle provided by MATLAB.

[There should usually be text between section headings. In this case, a "roadmap" of the subsections in this section would be appropriate. —SS] [Contents added here —Author]

4.1 Verification and Validation Team

The major member of the team is the author himself. Other contributors might assist in the VnV procedure, but their contributions are not guaranteed. In detail, the other contributors include the following people and their github accounts.

- Dr. Spencer Smith smiths and Deema Alomair deemaalmair1 review the whole library, including all documents and codes.

- Ao Dong Ao99 reviews the SRS.

- Peter Michalski peter-michalski reviews the VnV plans.

- Sasha Soraine sorainsm reviews the design.

[You should specifically list the class mates that are assigned to review your documents. You should also list the course instructor. —SS][Done. —Author]

4.2 SRS Verification Plan

The verification of the SRS document mainly consists of the checking of all the mathematical expressions of the system. Especially, we will use some simple mathematical functions and their related results to check the theories related to operations.

Some functions are reversions of other functions. For example, `Function Value` is the reversion of `Transform`, `Addition` is the reversion of `Subtraction`, `Division` is the reversion of `Multiplication`. For each pair of function `A` and `B`, we will see if functions `A(B())` and `B(A())`'s output is close to its input with some examples.

We will also learn from the feedback from reviewers, and the author's experience in developing and verifying this library. [A task directed verification plan could be much more effective than this ad hoc approach. —SS][Some ways added before the original content. —Author]

4.3 Design Verification Plan

The design of this library will be verified by checking the library against all the requirements. The functional requirements shall be checked first, followed by the performance-related parts of the non-functional requirements. [This is not a very detailed plan. Did you consider other options? Some of your colleagues, like Peter, have a detailed plan for design verification. —SS][Amended. —Author]

4.4 Implementation Verification Plan

The verification of the implementation of this library is mainly done by unit testing. The detail of unit testing can be found in the Unit Test Plan document. Mainly, the unit test will be done by first testing the basic functions in this library, and then testing the advanced functions. **Please note that the test result of any function in this library is acceptable, if and only if the reliant functions of this function is tested to be right.**

4.5 Software Validation Plan

The transformation part of this library will be validated by comparing its result with the MATLAB pseudo-oracle.

The MATLAB pseudo-oracle is computed by the implemented Fourier Transformation function within the MATLAB math libraries. We will choose several groups of $f(t), \omega, n$, compute the Fourier Transformation of $f(t)$ with frequency ω, and compare the leading terms with $CFS(f(t), n, \omega)$ [Where does the Matlab pseudo-oracle come from? This is a good idea, but you should take it a step further and explain the source of the pseudo-oracle. —SS][Elaborated here. —Author]

5 System Test Description

5.1 Tests for Functional Requirements

All tests in this section will be done by unit testing, the detail of which will be covered in the Unit Test VnV Plan document.

Since these tests involve comparing floating point numbers, and there are intrinsic errors regarding floating point computations, we have to set a tolerance, so that two floating numbers with difference smaller than this tolerance are considered equal. We use ϵ to represent this tolerance, and choose its value as 10^{-6} for all tests. This value is chosen because it is large enough to include all computation-related error, but small enough to detect any error in the result of the library. This value can be changed if we have good reasons to support this change. [Yes, using a symbolic constant for this is a good idea. I suggest that you put this constant in a table in the Appendix to this report, so that it is easy to maintain. I see that you are varying ϵ throughout this document. You should say this here, and not follow my advice to add a table of constants at the end, since the value is not constant. —SS]

5.1.1 Module 1: Basic comparison function

The tests here are selected to cover the tolerated comparison function and its base, subtraction operation and amplitude function. The tolerated comparison function will be used in later tests to compare the resulted CFS of the tested function with its (pseudo-)oracle counterpart, so we need to test this function first to ensure that the following test results are reliable.

NOTE: Do not proceed with other modules unless you have succeed in this module. For each round of test, the result of other modules is trustworthy if and only if the test results of this module are all successes.

1. Test of subtraction:

   Type: Automatic

   Initial State: The subtract function of the loaded FSL library.

   Input: $CFSf = [n = 2, \omega = 1.0, A = \{1.0, 0.0, 0.0\}, B = \{0.0, 1.0\}]$ and $CFSg = [n = 2, \omega = 1.0, A = \{0.0, 2.0, 1.0\}, B = \{1.0, 0.0\}]$

Output: Evaluated result of $CFSf - CFSg$, which should be a CFS object $[n = 2, \omega = 1.0, A = \{1.0, -2.0, -1.0\}, B = \{-1.0, 1.0\}]$

Test Case Derivation: Feed input, get output, and compare value-to-value automatically via Unit Test. [Why do this manually? This is a good test for automation. A unit testing framework can take care of the details. I see that many of the following tests are manual. Please consider making all of your tests automatic. —SS][I misunderstood the meaning of 'Test Case Derivation'. I thought it means how I find a set of input/outputs. Now, I found out that it means how we compare our output with the standard output. —Author]

How test will be performed: Unit Test framework will feed function with the aforementioned input, and compare the function output to the aforementioned standard output by comparing its n, ω, A_i's and B_i's variable-by-variable. Called 'compute and compare later'.

2. Test of amplitude function:

   Type: Automatic

   Initial State: None.

   Input: $CFSf = [n = 2, \omega = 1.0, A = \{1.0, 2.0, 2.0\}, B = \{2.0, 2.0\}]$ and $\epsilon = 10^{-6}$.

   Output: Evaluated result of $|Amp(CFSf) - 3.0| \leq \epsilon$, which should be `True`.

   Test Case Derivation: Feed input, get output, and compare value-to-value automatically via Unit Test.

   How test will be performed: Compute and compare.

3. Test of tolerated equality function `TolEq` (`True` result):

   Type: Automatic

   Initial State: Verified amplitude function.

   Input: $CFSf = [n = 2, \omega = 1.0, A = \{1.0, 0.0, 0.0\}, B = \{0.0, 1.0\}]$, $CFSg = [n = 2, \omega = 1.0, A = \{0.0, 2.0, 1.0\}, B = \{1.0, 0.0\}]$, and error $\epsilon = 10.0$.

Output: Return value of this function, which should be `True` [What is being compared? This is not clear. —SS][I mean the return value of this function —Author]

Test Case Derivation: Feed input, get output, and compare value-to-value automatically via Unit Test.

How test will be performed: Compute and compare.

4. Test of tolerated comparison function `TolEq` (`False` result):

   Same as *Test of tolerated comparison (`True` result)*, but with $\epsilon = 1.0$ and Output as `False`.

5.1.2 Module 2: Fourier transformation and approximation

This module tests the functions that compute Fourier transformation and approximated values of functions.

1. Test of coefficient (even function):

   Type: Automatic

   Initial State: Verified tolerated equality function `TolEq`.

   Input: $f(t) = t^2$, $\omega = 1$, $n = 2$, $\epsilon = 10^{-6}$, $CFSstd = [n = 2, \omega = 1, A = \{\pi^2/3, -4.0, 1.0\}, B = \{0.0, 0.0\}]$

   Output: Evaluated result of $TolEq(CFSf, CFSstd, \epsilon)$, which should be `True`. [Is this TolEq function defined somewhere? —SS][In SRS —Author]

   Test Case Derivation: Feed input, get output, and compare with `TolEq` and ϵ automatically via Unit Test.

   How test will be performed: Compute and compare, but instead of compare one-by-one, use `TolEq` function with the aforementioned tolerance ϵ. (Called 'Compute and compare with tolerance' later.)

2. Test of coefficient (odd function):

   Type: Automatic

Initial State: Verified tolerated equality function.

Input: $f(t) = t$, $\omega = 1$, $n = 2$, $\epsilon = 10^{-6}$, $\mathit{CFSstd} = [n = 2, \omega = 1, A = \{0.0, 0.0, 0.0\}, B = \{-2.0, 1.0\}]$

Output: $\mathit{TolEq}(\mathit{CFSf}, \mathit{CFSstd}, \epsilon)$, which should be `True`.

Test Case Derivation: Feed input, get output, and compare by `TolEq` and ϵ automatically via Unit Test.

How test will be performed: Compute and compare with tolerance.

3. Test of approximated function value:

   Type: Automatic

   Initial State: None.

   Input: $\mathit{CFSf} = [n = 2, \omega = 1, A = \{0.0, 2.0, 0.0\}, B = \{-2.0, 1.0\}]$, $t = \pi/4$, $\epsilon = 10^{-6}$.

   Output: Evaluated result of $|\mathit{App}(\mathit{CFSf}, t) - (2 + \sqrt{2}/2)| \leq \epsilon$, which should be `True`.

   Test Case Derivation: Feed input, get output, and compare by `TolEq` and ϵ automatically via Unit Test.

   How test will be performed: Compute and compare with tolerance.

5.1.3 Module 3: Operations and functions

1. Test of addition:

   Type: Automatic

   Initial State: Verified tolerated equality function.

   Input: $\mathit{CFSf} = [n = 2, \omega = 1, A = \{0.0, 2.0, 0.0\}, B = \{-2.0, 0.0\}]$, $\mathit{CFSg} = [n = 2, \omega = 1, A = \{1.0, 0.0, 2.0\}, B = \{0.0, 1.0\}]$, $\mathit{CFSstd} = [n = 2, \omega = 1, A = \{1.0, 2.0, 1.0\}, B = \{-2.0, 1.0\}]$, $\epsilon = 10^{-6}$

   Output: Evaluated result of $\mathit{TolEq}(\mathit{CFSf} + \mathit{CFSg}, \mathit{CFSstd}, \epsilon)$, which should be `True`.

   Test Case Derivation: Manual Computation.

   How test will be performed: Compute and compare.

2. Test of multiplication:

   Type: Automatic

   Initial State: Verified tolerated equality function.

   Input: $\mathit{CFSf} = [n = 2, \omega = 1, A = \{1.0, 0.0, 1.0\}, B = \{1.0, 0.0\}]$, $\mathit{CFSg} = [n = 2, \omega = 1, A = \{1.0, 1.0, 0.0\}, B = \{1.0, 0.0\}]$, $\mathit{CFSstd} = [n = 2, \omega = 1, A = \{2.0, 2.0, 0.0\}, B = \{2.0, 1.0\}]$, $\epsilon = 10^{-6}$

   Output: Evaluated result of $\mathit{TolEq}(\mathit{CFSf} * \mathit{CFSg}, \mathit{CFSstd}, \epsilon)$, which should be `True`.

   Test Case Derivation: Manual Computation.

   How test will be performed: Compute and compare.

3. Test of division:

   Same as *Test of multiplication*, but swap *CFSf* and *CFSstd*, while change $*$ to $/$.

4. Test of function of CFS:

   Type: Automatic

   Initial State: Verified tolerated equality, addition and multiplication function.

   Input: $\mathit{CFSf} = [n = 2, \omega = 1, A = \{1.0, 0.0, 1.0\}, B = \{1.0, 0.0\}]$, $g(t) = e^t$, and $\epsilon = 10^{-6}$.

   Output: Evaluated result of $\mathit{TolEq}(g(\mathit{CFSf}), 1 + \mathit{CFSf} + 0.5\mathit{CFSf}^2, \epsilon)$, which should be `True`.

   Test Case Derivation: Manual Computation.

   How test will be performed: Compute and compare.

5.1.4 Test of conversion

1. Test of conversion from other data format:

   Type: Automatic

Initial State: Verified tolerated equality function

Input: $n = 2$, $\omega = 1$, $A = 1.0, 0.0, 2.0$, $B = 2.0, 0.0$.

Output: constructed CFS

Test Case Derivation: None

How the test will be performed: convert and compare the result with a standard CFS using tolerated equality and $\epsilon = 10^{-6}$.

2. Test of conversion to other data format:

   Same as *Test of conversion from other data format*, while exchanging input and output, and compare each coefficients with tolerance as ϵ.

5.2 Test for input constraints

We design input test for detecting mismatch n and ω for functions that accept two CFS's as inputs. For each test in the following list, we derive tests for input constraints based on it.

- Test of addition
- Test of subtraction
- Test of multiplication
- Test of division
- Test of tolerated equality

For each test in this list, we derive two tests. One is to change the ω in the second CFS to the half of its original value, and the other is to change the n in the second CFS to 1, and remove A_2 and B_2 of the second CFS accordingly. Error message indicating a pair of mismatched CFS's shall appear.

Additionally, we do input constraint check on the tests of conversion from other data formats. We make n in the original test from 2 to 3, and error message indicating mismatched n, number of $A_i (i \neq 0)$'s, and number of B_i's.

5.3 Tests for Nonfunctional Requirements

The tests in this section will also be done by unit testing.

5.3.1 Speed evaluation

We test the speed of IM2, IM3, IM4, IM5, IM6, IM7, IM8 and IM9.

For each test, generate random CFS's with same ω and various n, as well as A_i's and B_i's from the same random number generator. [be more specific on what is involved in generating the random CFS's. What is the range of values that are allowed? —SS][The range of the values (fully decided by the random number generator) is the same within each test case, but may vary among test cases in the future, so I choose not to mention the detail of the range here, but within the test case. —Author] clock the execution time with the generated CFS's as input, and check whether the relationship between n and the execution time follows the requirements in the NFR.

We will demonstrate how the test is performed by showing a template of tests, and each test's values of template parameters (shown in the following font `Parameter`).

1. Test the speed of `Operation`

   Type: Automatic

   Initial State: `Operation`

   Input/Condition: Various pair of CFS's, with the same ω, $n = 100 : 100 : 1000$ respectively, and A_i's and B_i's generated from a random number generator. In this test, this generator is chosen as the uniform random number generator on $[-1.0, 1.0]$.

   Output/Result: A plot of average execution time versus n, together with the related `Speed Rule` regression coefficient r (if the value of `Speed Rule` is unknown, we will report the r, a, and b of the log-log regression $\log(y) = a\log(x) + b$ instead) [I think the output should be the plot of the time versus n. —SS]

   How test will be performed: For each n, generate 10 pairs of CFS's, clock their execution time to take average value as the execution time for CFS's of size n, plot as said before, find and show the r value of a `Speed Rule` fitting between n and the execution time. Usually, this test is considered a success when $|r| > 0.9$, but it can also be considered a success if the tester has found a good explanation of this coefficient.

Here is a list of the values of parameters in each test.

IM#	`Operation`	`Speed Rule`
IM2	Approximate Function Value	Linear
IM3	Addition	Linear
IM4	Subtraction	Linear
IM5	Multiplication	Squared
IM6	Division	Unknown
IM7	Function	Unknown
IM8	Amplitude	Linear
IM9	Tolerated Equality	Linear

Table 1: Table of parameter values for each test.

[You should explain all of the tests, but you can use a table to summarize just the deltas between the base test and the individual tests. —SS][Done accordingly. —Author]

[What about other nonfunctional tests? Usability could be a valuable thing to assess. Portability? (I think this would be a fairly easy one to assess.) Installability? —SS] [The usability test is deemed unnecessary, due to the fact that the interface of this library is pretty simple, and given my experience of using several libraries, I think that it is quite easy to use if the user have basic knowledge of this language. —Author] [Initially when I think about my library at this stage, I have not decided about the language, test tools, and other things. I only knows that I will distribute the source code. In the case of source code distribution, the common practice among the open-source software area is that the users shall fully taken care of the portability and installability part, thus I did not plan to test them here. —Author]

[Furthermore, I anticipate that I will be writing codes strictly following basic standards with popular supporting libraries (like boost), and the portability of these standards and libraries have been tested by lots and lots of people on different kinds of platforms, so the portability of my library is guaranteed. —Author]

[I will also organize my library in a popular way, and when used by the users, the library will not contain any codes to be compiled when being installed, just some source codes, and the users will compile my source code together with theirs. Thus, the install procedure only involves that the user copy/link my library files to their intended location, which is pretty easy and straightforward on the platforms that I know of, so the installability is guaranteed. —Author]

[However, after I have considered the actual unit test part, I think I can do a little bit to the two test just to provide proof to my arguments. The contents I added is shown as follows. —Author]

5.3.2 Test of portability and installability

The portability and installability test will be done by doing the unit tests on several platforms, with different compilers and different versions of language standards.

The platforms include mac OS X (10.11 10.15), Windows 10 (1909 as a representative, since Windows has a good reputation of compatibility between versions), and Linux distributions including Ubuntu 18.04, CentOS 7.7-1908, Fedora 30, and Archlinux 2019-12-01 version.

At this stage, I have anticipated that I will be using C++ to implement this library. The standards I choose to test for compatibility are C++03, C++11, C++14, C++17, and C++20 (if it becomes official in the lifespan of this library).

I will also be checking the compatibility with the following C++ compilers: gcc (version 5.0˜8.0), clang (version 5.0, 6.0), msvc (the versions distributed with vs2017 and vs2019), and icc (16.0˜19.0)

5.4 Traceability Between Test Cases and Instance Modules

We show each instance module's covering tests. [You don't have modules. I think you mean requirements? —SS][Actually, I mean instance modules. Each set of tests covers the related requirements to this instance module, and how the requirements apply to this instance module. —Author]

- IM1: *Test of transformation (even function)* and *Test of transformation (odd function)*.

- IM2: *Test of approximated function value*
- IM3: *Test of addition*
- IM4: *Test of subtraction*
- IM5: *Test of multiplication*
- IM6: *Test of division*
- IM7: *Test of function of CFS*
- IM8: *Test of amplitude*
- IM9: *Test of tolerated equality*
- IM10: *Test of conversion from other data format*
- IM11: *Test of conversion to other data format*

[We discussed that your unit verification and validation plan will be identical to the system plan. I agree there is overlap, but I expect that you will have unique tests for the unit case. For instance, if you introduce a data structure for the CFSs, then you want to be able to check each of the methods for the CFSs. —SS][Indeed. The contents you mentioned will be elaborated in UnitVnV. —Author]

References