System and Unit Test Report: Fourier Series Library

Bo Cao

December 18, 2019

1 Revision History

Date	Version	Notes
Dec. 17, 2019	1.0	First and final draft.

2 Symbols, Abbreviations and Acronyms

symbol	description
T	Test

[symbols, abbreviations or acronyms – you can reference the SRS tables if needed —SS]

Contents

List of Tables

List of Figures

This document introduces the result of the Unit VnV procedure.

3 Functional Requirements Evaluation

All the functional requirements have been met, except for the division function.

4 Nonfunctional Requirements Evaluation

Generally, all the requirements have been met.

4.1 Usability

During implementation, we find out that several functions' names are too long to use. For the sake of compatibility, we will introduce shorter function names in parallel. These functions with shorter names are essentially wrappers to the original ones, and the structures of these functions are too simple, so we do not test them.

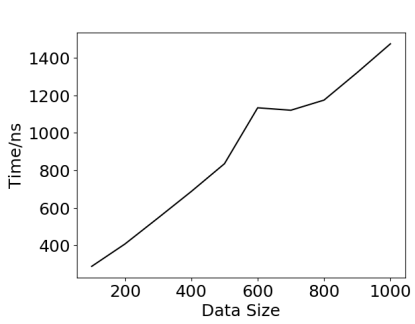
We show a list of old function names versus new function names.

Old Function Name	New Function Name
`Addition`	`Add`
`Subtraction`	`Sub`
`Multiplication`	`Mul`
`Division`	`Div`
`Amplitude`	`Amp`
`ToleratedEquality`	`TolEq`

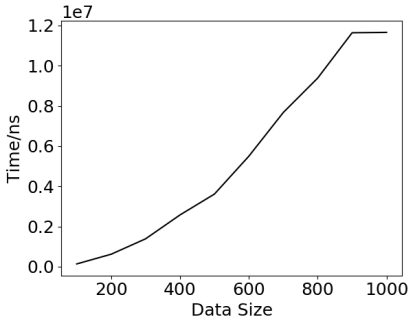
4.2 Performance

We have done performance tests on three functions, `Amplitude`, `Multiplication`, and `TransformTo`. Each of these tests represents tests on functions with one kind of speed rule. The tests on other functions will be done in the future.

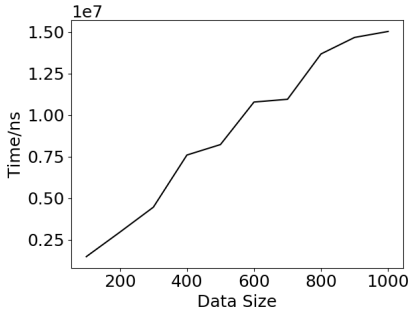
We give the plots for these functions. The extrapolation will be done in the future, since these plots are already convincing. In these figures, Figure 1a



(a) Function `Amplitude`



(b) Function `Multiplication`



(c) Function `TransformTo`

Figure 1: Selected functions: Data Size versus Exectution time

and Figure 1c shows that the execution time of functions `Amplitude` and `TransformTo` is generally proportional to the data size, while Figure 1b shows that the execution time of function `Multiplication` is generally proportional to the square of the data size. These speed rules are consistent with what we expect.

4.3 Portability and Install-ability

The unit testing has been performed on Ubuntu 18.04.3 and Mac OS X 10.11, with compilers gcc-7 and clang++-6.0. Based on the compatibility reports by the developers and users of these operating systems and compilers, we have confidence that this library should work on all *nix-based systems, with a C++ compiler supporting C++-17.

We also tested the installation and configuration of this library. We found out that the installation is a simple copy of all the header files, and the

configuration only involves designating the location of these library during the user's building of their own programs. However, we found out that users have to include these headers one-by-one, and we think that an all-inclusive header might facilitate this process. We have implemented headers for each module, as well as an all-inclusive header.2

5 Unit Testing

The Unit Testing has been done with Catch2 in accordance with Unit VnV. All tests except these for function `Division` succeeds. We will fix this function in the future.

6 Changes Due to Testing

Most changes are grammar related. We list those changes that are not trivial.

1.
   - Function: `CFSData::setn`
   - Problem: Old data still exists after resizing the sequences.
   - Solution: `std::fill(A.begin(), A.end(), 0)`, `std::fill(B.begin(), B.end(), 0)`
2.
   - Function: `Addition` and `Subtraction`
   - Problem: Last element not computed.
   - Solution: `for(size_t i = 0; i<n; ++i)<` $\rightarrow$ `<=`
3.
   - Function: `Multiplication`
   - Problem: Computation part wrong, unable to debug.
   - Solution: Computation part changed from the mathematical expression in SRS to a more basic function. Easy to implement now.

7 Automated Testing

Test can now be done via the `r` script file. This file contains commands for building the test program based on the test case, as well as executing the test program.

8 Code Coverage Metrics

The following lines of code are reported by `gcov` as being not covered.

1. All the exception-throwing lines in `CFSData.hpp`
2. All the `return false;` in `CFSMatch.hpp`. This is strange, since there are test cases that requires the function to return `false`, and these test cases are success. Later, we change the compiler flag from "-O3" meaning maximum optimization to "-O0" meaning no optimization, and these codes are shown covered in our test. We thus conclude that these codes are optimized when "-O3" flag is used.

These codes are examined manually by us to be right, and we will produces test cases for these un-covered codes in the future.

References