CFS: Unit Verification and Validation Plan for CFS

Malavika Srinivasan

December 4, 2018

1 Revision History

Date	Version	Notes
Dec 4, 2018	1.0	First draft by Malavika

Contents

2 Symbols, Abbreviations and Acronyms

symbol	**description**
T	Test

Also, see the table of symbols in CA at: https://github.com/Malavika-Srinivasan/CAS741/tree/master/docs/SRS/CA.pdf

This document explains the unit verification and validation plan to improve the quality of CFS. It is organized into different sections which gives a detailed description of the CFS in terms of its goals, objectives, essential qualities and test cases to verify each module listed in the MG.

3 General Information

This section explains the summary of what is being tested in this document, the objectives of this document and references for this document.

3.1 Purpose

This document will summarize the plan for verification and validation of of the modules of CFS in compliance with their functions mentioned in the MIS document found at https://github.com/Malavika-Srinivasan/CAS741/tree/master/docs/Design/MIS/MIS.pdf

The goal statement as found in the CA document is presented below. "Given the set of data points, the choice of software from CFS and the variabilities of the software the CFS should:

1. Compute the parameters of the curve which is the best possible fit through the set of data points. "

3.2 Scope

Some of the modules such as sequence services and plot module are not implemented by CFS but are a part of it. The services offered by these modules are implemented by python and testing them are out of scope for this project.

4 Plan

This section represents the planning phase for unit verification and validation plan in terms of people, tasks and tools involved in this process.

4.1 Verification and Validation Team

The verification and validation team for CFS includes the following people.

- Malavika Srinivasan (Primary tester)
- Dr.Spencer Smith (Verification of this document)
- Hanane Zlitni (Verification of this document)

4.2 Automated Testing and Verification Tools

In this project, we will be using Pytest to verify the modules of CFS. It is a testing framework available in python. The results of the test including the code coverage metrics will be presented in the test report. We will be using cov package which gives the code coverage metrics.

4.3 Non-Testing Based Verification

NA

5 Unit Test Description

The test cases presented here will verify the access programs of the modules listed in MIS. Usually, each access method will have a test case associated with it. If not, the reason for why the method does not have an associated test case is explained. The MIS can be found at `https://github.com/Malavika-Srinivasan/CAS741/tree/master/docs/Design/MIS/MIS.pdf`

5.1 Tests for Functional Requirements

NA

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS] [I have parallel testing, which makes sense as a system test. But not for unit test. —Malavika]

5.1.1 Input Module

In this section , we will present the test cases for each access method. Since all the access methods are tested, the entire module will be tested. The test cases for this module are selected based on the constraints for the input data. The typical constraints on the input data for CFS is given by the data type, length. Some of the inputs such as the degree of the polynomial have mathematical restrictions of being a natural number. The test cases listed below are designed to check whether the input data to CFS violates these constraints.

1. **T1: Test case for verify input - One data point**
   Please refer to test case T1 in section 5.1.1 in System verification and validation plan.

   Test Case Derivation: We need at least two points to draw a straight line which is the lowest order polynomial. Only one data point does not need a curve or numbers to represent itself. The coordinates would be sufficient. Hence, CFS should throw an exception.

   How test will be performed: Pytest

2. **T2: Test case for verify input - Length mismatch**

   Type: Automatic

   Initial State: NA

   Input: t=[1,2,3] , y=[2,4] (length mismatch)

   Output: Exception message ("t and y array must have same length.")

   Test Case Derivation: Basic mathematics rule for curve fitting [Not sure what to write here? —Malavika]

   How test will be performed: Pytest

3. **T3: Test case for verify input - Type error**
   Please refer T2 in section 5.1.1 in System verification and validation plan.

   Test Case Derivation: It is necessary that the input arrays contains only numbers.

   How test will be performed: Pytest

4. **T4: Test case for verify degree - value error**

   Type: Automatic

   Initial State: NA

   Input: deg = 1.5

   Output: Exception message ("Degree cannot be a real number")

   Test Case Derivation: Basic mathematics rule for curve fitting [Not sure what to write here too? —Malavika]

   How test will be performed: Pytest

5.1.2 Interpolation module

In this section, we will represent the test cases for each access program present in the interpolation module. This module has access programs to find the coefficients of the interpolated curve through a set of points (t_i, y_i) for $i = 0$ to n and find the value of the interpolating polynomial at a given t value.

5. **T5: Test case for interpMonomial**

   Please refer to test cases T4 and T5 in section 5.1.2 in System verification and validation plan.

6. **T6: Test case for interpLagrange**

   Please refer to test cases T6 and T7 in section 5.1.2 in System verification and validation plan.

7. **T7: Test case for interpNewton**

   Please refer to test cases T8 and T9 in section 5.1.2 in System verification and validation plan.

8. **T8: Test case for interpHermiteCubic**

   Please refer to test cases T10 and T11 in section 5.1.2 in System verification and validation plan.

9. **T9: Test case for interpBSpline**

   Please refer to test case T12 in section 5.1.2 in System verification and validation plan.

10. **T10: Test case** 1 **for evalMonomial**

    Type: Automatic

    Initial State: NA

    Input: [0,1], 2

    Output: 2

    Test Case Derivation: Please see inputs of T4 in section 5.1.1 in System verification and validation plan.

    How test will be performed: Pytest

11. **T11: Test case** 2 **for evalMonomial**

    Type: Automatic

    Initial State: NA

    Input: [-1, 5, 4], -2

    Output: -27

    Test Case Derivation: Page 314, example 7.1 of **?**.

    How test will be performed: Pytest

12. **T12: Test case** 1 **for evalLagrange**

    Type: Automatic

    Initial State: NA

    Input: [0,1,2], 1

    Output: 1

    Test Case Derivation: Please see inputs of T4 in section 5.1.1 in System verification and validation plan.

    How test will be performed: Pytest

13. **T13: Test case** 2 **for evalLagrange**

    Type: Automatic

    Initial State: NA

    Input: [-27,-1,0], -2

    Output: -27

Test Case Derivation: Page 314, example 7.1 of **?**.

How test will be performed: Pytest

14. **T14: Test case** 1 **for evalNewton**

    Type: Automatic

    Initial State: NA

    Input: [0,1], 2

    Output: 2

    Test Case Derivation: Please see inputs of T4 in section 5.1.1 in System verification and validation plan.

    How test will be performed: Pytest

15. **T15: Test case** 2 **for evalNewton**

    Type: Automatic

    Initial State: NA

    Input: [-1, 5, 4], -2

    Output: -27

    Test Case Derivation: Page 314, example 7.1 of **?**.

    How test will be performed: Pytest

16. **T16: Test case** 1 **for evalHermiteCubic**

    Type: Automatic

    Initial State: NA

    Input:
    $[1, -5.75, 9.5, 2]$, [1,3][Interval to be read as 1 to 3 —Malavika], [1,3][Input x values —Malavika]

    Output: [2,1][Output for each x value —Malavika]

    Test Case Derivation: **?**

    How test will be performed: Pytest

17. **T17: Test case 2 for evalHermiteCubic**

    Type: Automatic

    Initial State: NA

    Input:
    $[[1.0, 1.0, 1.38888889, 2.0]$
    $[4.0, 4.0, 1.0, 1.0]$
    $[3.0, 3.61111111, 4.0, 4.0]]$, [1,2,4,5][Interval to be read as 1 to 2, 2 to 4 and 4 to 5 —Malavika], [1,2,4,5] [Input x values —Malavika]

    Output: [2, 1, 4, 3][Output for each x value —Malavika]

    Test Case Derivation: **?**

    How test will be performed: Pytest

18. **T18: Test case for evalBSpline**

    Type: Automatic

    Initial State: NA

    Input: [[-5.62048630e-18, 2.98780300e+00, -5.74472095e-01, 1.46700914e+01, -1.03253068e+01, 3.10000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00]], t = [0.0, 1.2, 1.9, 3.2, 4.0, 6.5]

    Output: [0.0, 2.3, 3.0, 4.3, 2.9, 3.1]

    Test Case Derivation: **?**

    How test will be performed: Pytest

5.2 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.2.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

5.2.2 Module ?

...

5.3 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

6 Appendix

[This is where you can place additional information, as appropriate —SS]

6.1 Symbolic Parameters

[The definition of the test cases may call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance. —SS]

References