Project Title: System Verification and Validation Plan for Radio Signal Strength Calculator

Xingzhi Liu

December 23, 2020

1 Revision History

Date	Version	Notes
Oct 30, 2020	1.0	First Draft
Dec 20, 2020	1.1	Revision 1

Contents

List of Tables

List of Figures

2 Symbols, Abbreviations and Acronyms

symbol	description
FR	Functional Requirement
MG	Module Guide
MIS	Module Interface Specification
NFR	Non-Functional Requirement
RSSC	Sadio SignalStrength Calculator
SRS	Software Requirement Specification
T	Test
VV	Validation and Verification

This document records and presents the verification and validation plan for RSSC to help ensure the program meets the requirements. General information including a quick recall of RSSC's background is given in section 3. section 4 provides a plan for verification and section 5 describes the system tests, including tests for functional requirements and tests for non- functional requirements.

3 General Information

3.1 Summary

The software to test is Radio Signal Strength Calculator (RSSC). The purpose of RSSC is to simulate the radio signal propagation in the indoor environment defined by the user, and find the analytical signal strength for the user.

3.2 Objectives

The objective of this document is to build confidence in the software correctness. To reach this objective, all functional and non-functional requirements will be tested following the descriptions in this document.

3.3 Relevant Documentation

- SRS

4 Plan

4.1 Verification and Validation Team

- Author: Xingzhi Liu
- Primary Reviewer: Siddharth (Sid) Shinde
- Reviewer: Leila Mousapour
- Reviewer: Shayan Mousavi Masouleh
- Dr. Spencer Smith

4.2 SRS Verification Plan

SRS will be done by team members reviewing the document. Team members can put any comments, suggestions or questions in RSSC's Github repository as issues. The SRS reviewing team includes the following members: domain expert Siddharth (Sid) Shinde, the SRS reviewer Leila Mousapour, Dr. Spencer Smith, and the author Xingzhi Liu. The author will respond to the issues and make modifications when needed.

4.3 Design Verification Plan

Design verification will be done by team members, by reviewing whether the steps of calculation in the software follows the physical model in SRS or not.

4.4 Implementation Verification Plan

Implementation verification will be done by testing all the functional and non-functional requirements. Descriptions of the tests can be found in subsection 5.1 and subsection 5.2. In addition, we will undergo static verification by checking all the codes we build with Pylint. We will also conduct unit testing for modules within the testing scope. Details for unit testing can be found in section 6.

4.5 Automated Testing and Verification Tools

- Python Unittest
- Pylint

4.6 Software Validation Plan

There are no plans for validation. We may be able to set up a simulation of radio frequency signal on existing physics simulation tools and take the simulation result as a reference to evaluate RSSC's output, but different tools makes different assumptions to the indoor signal propagation model and we cannot change their assumptions. It is highly unlikely that we could find a tool that makes the same assumptions as RSSC. Therefore we do not have any reference to evaluate RSSC's correctness.

5 System Test Description

5.1 Tests for Functional Requirements

Functional requirements for RSSC are given in SRS section 5.1. There are 5 functional requirements for RSSC, from R1 to R5. R1 and R2 are corresponding to inputs, while R3 to R5 are corresponding to outputs. subsubsection 5.1.1 describes the input tests for R1 and R2; and subsubsection 5.1.2 describes the output tests for R3, R4 and R5.

5.1.1 Input

This test verifies the following requirements:
R1: RSSC takes input from the user;
R2: RSSC verifies user inputs.

Input Tests

1. Valid inputs

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[10, 0], [-10, 0]]$;
   $[C] = [[1, 1], [2, 2], [1, 3]]$;
   $[D] = [[2, 2], [1, 3], [1, 1]]$;
   $[T] = [0.1, 0.1, 0.1]$;
   $[R] = [0.6, 0.6, 0.6]$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

   Output: Return an input success message in command line.

   Test Case Derivation: RSSC correctly takes the inputs from the user

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

2. Inconsistent input array sizes

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[10, 0], [-10, 0]]$;
   $[C] = [[1, 1], [2, 2], [1, 3]]$;
   $[D] = [[2, 2], [1, 3]]$;
   $[T] = [0.1, 0.1, 0.1]$;
   $[R] = [0.6, 0.6, 0.6]$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

   Output: Return error of inconsistent array size in command line.

   Test Case Derivation: Correct error message displays.

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

3. Out-of-range position coordinates

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [21, 0]$;
   $[Pos_{sp}] = [[10, 0], [-10, 0]]$;
   $[C] = [[1, 1], [2, 2], [1, 3]]$;
   $[D] = [[2, 2], [1, 3], [1, 1]]$;
   $[T] = [0.1, 0.1, 0.1]$;
   $[R] = [0.6, 0.6, 0.6]$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

   Output: Return error of out-of-range position coordinates.

   Test Case Derivation: Correct error message displays.

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

4. Out-of-range transmitter power level

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[10, 0], [-10, 0]]$;
   $[C] = [[1, 1], [2, 2], [1, 3]]$;
   $[D] = [[2, 2], [1, 3], [1, 1]]$;
   $[T] = [0.1, 0.1, 0.1]$;
   $[R] = [0.6, 0.6, 0.6]$;
   $P_{tsm}^{dBm} = 20$;
   $f = 2.48 \times 10^9$;

   Output: Return error of out-of-range transmitter power level.

   Test Case Derivation: Correct error message displays.

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

5. Out-of-range signal frequency

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[10, 0], [-10, 0]]$;
   $[C] = [[1, 1], [2, 2], [1, 3]]$;
   $[D] = [[2, 2], [1, 3], [1, 1]]$;
   $[T] = [0.1, 0.1, 0.1]$;
   $[R] = [0.6, 0.6, 0.6]$;
   $P_{tsm}^{dBm} = 20$;
   $f = 2.48 \times 10^{12}$;

   Output: Return error of out-of-range signal frequency.

   Test Case Derivation: Correct error message displays.

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

5.1.2 Output

This test verifies the following requirements:

R3: RSSC shall find P_{sp}^{dBm};
R4: RSSC shall verify P_{sp}^{dBm};
R5: RSSC shall generate a file to store P_{sp}^{dBm};

Output Test

1. Simple single valid output

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[1, 0]]$;
   $[C] = []$;
   $[D] = []$;
   $[T] = []$;
   $[R] = []$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

   Output: A .txt or .csv file with 1 line, showing the sampling point position (1,0) and P_{sp}^{dBm} that satisfies $P_{sp}^{dBm} \leq 0$.

   Test Case Derivation: RSSC returns the correct output.

   How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

2. Simple multiple valid output

   Control: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[1, 0], [1, 1]]$;
   $[C] = []$;

$[D] = []$;
$[T] = []$;
$[R] = []$;
$P_{tsm}^{dBm} = 0$;
$f = 2.48 \times 10^9$;

Output: A .txt or .csv file with 2 lines. Line 1 shows the sampling point position (1,0) and P_{sp}^{dBm} that satisfies $P_{sp}^{dBm} \leq 0$. Line 2 shows the sampling point position (1,1) and P_{sp}^{dBm} that satisfies $P_{sp}^{dBm} \leq 0$.

Test Case Derivation: RSSC returns the correct output.

How test will be performed: Tester manually feeds the inputs into RSSC and executes RSSC.

5.2 Tests for Non-Functional Requirements

Non-functional requirements are given in SRS section 5.2. There are 5 non-functional requirements: Portable, Maintainable, and Understandable. The rest of this section provides detailed descriptions on how to test them.

5.2.1 Portable

Portability Test RSSC shall be able to run on different OS. We will test execute RSSC on both Windows and Linux Ubuntu.

1. Portability on Windows 10

   Type: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[1, 0]]$;
   $[C] = []$;
   $[D] = []$;
   $[T] = []$;
   $[R] = []$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

Output: A .txt or .csv file with 1 line, showing the sampling point position (1,0) and P_{sp}^{dBm} that satisfies $P_{sp}^{dBm} \leq 0$.

How test will be performed: Tester deploys RSSC on a machine with Windows 10 OS and execute with the input set above. On success, RSSC should provide the output as described.

2. Portability on Linux Ubuntu

   Type: Manual

   Initial State: Pending Input

   Input: $Pos_{tsm} = [0, 0]$;
   $[Pos_{sp}] = [[1, 0]]$;
   $[C] = []$;
   $[D] = []$;
   $[T] = []$;
   $[R] = []$;
   $P_{tsm}^{dBm} = 0$;
   $f = 2.48 \times 10^9$;

   Output: A .txt or .csv file with 1 line, showing the sampling point position (1,0) and P_{sp}^{dBm} that satisfies $P_{sp}^{dBm} \leq 0$.

   How test will be performed: Tester deploys RSSC on a virtual machine with Ubuntu 20.04 and execute with the input set above. On success, RSSC should provide the output as described.

5.2.2 Maintainable

Maintainability Test Proper documents should be included in this project.

1. Maintainability Test

   Type: Manual

   Initial State: none

   Input: none

   Output: none

How test will be performed: Tester manually checks the contents in the Github repo. On success, documents shall be uploaded following the schedule of CAS741 and no issue shall be closed without a proper response.

5.2.3 Understandable

Understandability Test Programs of RSSC should be organized, well commented, and easy to understand.

1. Understandability Test

   Type: Manual

   Initial State: none

   Input: none

   Output: none

   How test will be performed: Tester review the code and complete the survey shown in Table 1.

Item	**Score**
Variable names are rational and follow consistent conventions	{0 - 5}
Functions are well commented on what they do	{0 - 5}
Functions tasks are broken down / No super complicated functions	{0 - 5}
No repeated chunks in the code	{0 - 5}
Code is well organized and follows the order of instance models	{0 - 5}

Table 1: Understandability Survey

	R1	R2	R3	R4	R5	Portable	Maintainable	Understandable
Input	X	X						
Output			X	X	X			
Portable						X		
Maintainable							X	
Understandable								X

Table 2: Traceability Between Test Cases and Requirements

5.3 Traceability Between Test Cases and Requirements

6 Unit Test Description

6.1 Unit Testing Scope

We will conduct unit testings for the following modules:

- Point
- Wall
- Linear Path
- Intersection
- Linear Path Loss
- Specular Reflection

Modules including Control, Input Parameters, Output, and Floor Map have low priorities for verification than others because their correctness can be adequately proven by Input and Output.
Module Received Signal Strength only contains some basic numerical calculations so we do not need to test it separately. Output will be sufficient to verify its functionality.
Module Line-Of-Sight Signal and module First-Order Reflection Signal require module Input Parameters to run first before they work, and cannot be verified separately. However, testings in Output requires these two modules working properly to provide outputs. Therefore, these two modules will be verified by Output.

6.2 Tests for Functional Requirements

6.2.1 Upoint

This section tests the functionality of Point module, including creating a point and reading the coordinates of a point.

1. point-1

   Type: manual

   Initial State: none

   Input:
   x = 0, y = 0;
   x = 3, y = 4;
   x = 1, y = 10;
   x = 3, y = 0;

   Output: 4 points successfully created and their positions printed. Positions of the 4 points should be (0, 0), (3, 4), (1, 10), and (3, 0).

   Test Case Derivation: A point is created with the input position.

   How test will be performed: Call function Point(x, y) 4 times with inputs listed above. Every time after Point() is called, run function get_coordinates() and print the result as well as the expected result.

6.2.2 Uwall

This section tests the functionality of Wall module, including creating a wall and finding the equation and the unit normal vector of itself.

1. wall-1

   Type: manual

   Initial State: none

   Input:
   a = Point (0, 0);

b = Point (1, 10);
T = 0.1;
R = 0.5;

Output:
m1 = -10;
m2 = 1;
k = 0;
$[n1, n2]$ = [0.9950, -0.0995] or [-0.9950, 0.0995]

Test Case Derivation: The equation and unit normal vector are calculated with models shown in SRS.

How test will be performed: Call function Wall(a, b, T, R) and print parameters m1, m2, k, n1, and n2 of the wall object, together with their expected values.

2. wall-2

   Type: manual

   Initial State: none

   Input:
   a = Point (3, 4);
   b = Point (1, 10);
   T = 0.1;
   R = 0.5;

   Output:
   m1 = 3;
   m2 = 1;
   k = 13;
   $[n1, n2]$ = [0.9487, 0.3162] or [-0.9487, -0.3162]

   Test Case Derivation: The equation and unit normal vector are calculated with models shown in SRS.

How test will be performed: Call function Wall(a, b, T, R) and print parameters m1, m2, k, n1, and n2 of the wall object, together with their expected values.

3. wall-3

   Type: manual

   Initial State: none

   Input:
   a = Point (0, 0);
   b = Point (3, 4);
   T = 0.1;
   R = 0.5;

   Output:
   m1 = -1.3333;
   m2 = 1;
   k = 0;
   $[n1, n2]$ = [0.8, -0.6] or [-0.8, 0.6]

   Test Case Derivation: The equation and unit normal vector are calculated with models shown in SRS.

   How test will be performed: Call function Wall(a, b, T, R) and print parameters m1, m2, k, n1, and n2 of the wall object, together with their expected values.

4. wall-4

   Type: manual

   Initial State: none

   Input:
   a = Point (3, 4);
   b = Point (3, 0);
   T = 0.1;

R = 0.5;

Output:
m1 = 1;
m2 = 0;
k = 3;
$[n1, n2]$ = [1, 0] or [-1, 0]

Test Case Derivation: The equation and unit normal vector are calculated with models shown in SRS.

How test will be performed: Call function Wall(a, b, T, R) and print parameters m1, m2, k, n1, and n2 of the wall object, together with their expected values.

6.2.3 Ulp

This section tests the functionality of Linear Path module, including creating a Linear Path and finding the equation and the length of itself.

1. lp-1

   Type: manual

   Initial State: none

   Input:
   a = Point (-2, 2);
   b = Point (13, 2);

   Output:
   m1 = 0;
   m2 = 1;
   k = 2;
   length = 15

Test Case Derivation: The equation is calculated with the linear equation model shown in SRS. The length is the Euclidean distance between the two input points.

How test will be performed: Call function LinearPath(a, b) and print parameters m1, m2, k, and length of the LinearPath object, together with their expected values.

2. lp-2

   Type: manual

   Initial State: none

   Input:
   a = Point (-2, 2);
   b = Point (4, 5);

   Output:
   m1 = -0.5;
   m2 = 1;
   k = 3;
   length = 6.7082

   Test Case Derivation: The equation is calculated with the linear equation model shown in SRS. The length is the Euclidean distance between the two input points.

   How test will be performed: Call function LinearPath(a, b) and print parameters m1, m2, k, and length of the LinearPath object, together with their expected values.

3. lp-3

   Type: manual

   Initial State: none

   Input:
   a = Point (-2, 2);

b = Point (-2, 10);

Output:
m1 = 1;
m2 = 0;
k = -2;
length = 8

Test Case Derivation: The equation is calculated with the linear equation model shown in SRS. The length is the Euclidean distance between the two input points.

How test will be performed: Call function LinearPath(a, b) and print parameters m1, m2, k, and length of the LinearPath object, together with their expected values.

6.2.4 Uint

This section tests the functionality of Intersection module in finding the position and validity of the potential intersection point of any two line segments given.

1. inter-1

   Type: manual

   Initial State: none

   Input:
   seg1 = output of test case wall-1 in Uwall;
   seg2 = output of test case lp-1 in Ulp;

   Output:
   Intersection position = (0.2, 2.0)
   validity = 1

   Test Case Derivation: seg1 is a line segment from (0, 0) to (1, 10); seg2 is a line segment from (-2, 2) to (13, 2). Both of them cross the point

(0.2, 2.0), therefore (0.2, 2.0) is a valid intersection of seg1 and seg2.

How test will be performed: Call Intersection.find_intersection(seg1, seg2) and print the result as well as the expected result. Then Call Intersection.is_valid(seg1, seg2) and print the result as well as the expected result.

2. inter-2

   Type: manual

   Initial State: none

   Input:
   seg1 = output of test case wall-4 in Uwall;
   seg2 = output of test case lp-3 in Ulp;

   Output:
   Intersection position = (not important)
   validity = 0

   Test Case Derivation: seg1 is a line segment from (3, 4) to (3, 0); seg2 is a line segment from (-2, 2) to (-2, 10). seg1 and seg2 are parallel, so they do not have intersection points. Therefore the validity is 0 and the position of the potential intersection point is unessential.

   How test will be performed: Call Intersection.find_intersection(seg1, seg2) and print the result as well as the expected result. Then Call Intersection.is_valid(seg1, seg2) and print the result as well as the expected result.

3. inter-3

   Type: manual

   Initial State: none

   Input:
   seg1 = output of test case wall-3 in Uwall;

seg2 = output of test case lp-3 in Ulp;

Output:
Intersection position = (-2, -2.667)(not important)
validity = 0

Test Case Derivation: seg1 is a line segment from (0, 0) to (3, 4); seg2 is a line segment from (-2, 2) to (-2, 10). extension lines of the two segments intersect at (-2, -2.667), but the two line segments do not cross that point. Therefore the potential intersection point is invalid.

How test will be performed: Call Intersection.find_intersection(seg1, seg2) and print the result as well as the expected result. Then Call Intersection.is_valid(seg1, seg2) and print the result as well as the expected result.

6.2.5 Ull

This section tests the functionality of Linear Path Loss module to determine the power loss of a signal along a given linear path.

1. ll-1

   Type: manual

   Initial State: none

   Input:
   path1 = output of test case lp-1 in Ulp;
   frequency = 2.48×10^9;
   floor_map = a FloorMap object with the following3 walls:
   wall1 = output of test case wall-1 in Uwall;
   wall2 = output of test case wall-2 in Uwall;
   wall2 = output of test case wall-3 in Uwall;

   Output: linear path loss = 4.1185×10^-09.

Test Case Derivation: The linear path loss is calculated with the Line Of Sight Signal Strength model described in SRS.

How test will be performed: Call function LinearLoss.find_linear_path_loss(path1, frequency, floor_map) and print the result as well as the expected result.

6.2.6 Uspec

This section tests the functionality of Specular Reflection in finding the validity and potential path of a potential first-order reflected signal.

1. spec-1

   Type: manual

   Initial State: none

   Input:
   wall = output of test case wall-1 in Uwall;
   start = Point(-2, 2);
   end = Point(-1, 3);

   Output:
   first half of the signal path (path1) = a path from Point(-2, 2) to Point(0.01884, 0.1884)
   first half of the signal path (path1) = a path from Point(0.01884, 0.1884) to Point(-1, 3)
   validity = 1

   Test Case Derivation: the reflected image of the start point (-2, 2) against the given wall is at (2.3564, 1.5644). The virtual signal path from the reflected image of the starting point to the ending point intersects the wall at Point(0.1782, 1.782), which is a valid intersection. These information implies the existence of a first-order reflected signal from Point(-2, 2) to Point(-1, 3), reflected by the input wall at Point(0.1782, 1.782). The module's task is to find such a signal.

   How test will be performed: Call Specular.get_mirrored_paths(wall, start, end)) and print the output as well as the expected results.

2. spec2

   Type: manual

   Initial State: none

   Input:
   wall = output of test case wall-2 in subsubsection 6.2.2;
   start = Point(-2, 2);
   end = Point(-1, 3);

   Output:
   validity = 0

   Test Case Derivation: the reflected image of the start point (-2, 2) against the given wall is at (8.2, 5.4). The virtual signal path from the reflected image of the starting point to the ending point is supposed to intersect the wall at Point(3.1, 3.7), but this is not a valid intersection. These information implies that there is no first-order reflected signals from Point(-2, 2) to Point(-1, 3).

   How test will be performed: Call Specular.get_mirrored_paths(wall, start, end)) and print the output as well as the expected results.

6.3 Traceability Between Test Cases and Modules

	Upoint	Uwall	Ulp	Uint	Ull	Uspec	Input	Output
Control							X	X
Input Parameters							X	
Point	X							
Wall		X						
Floor Map								X
Equation Finder		X						
Linear Signal Path			X					
Intersection				X				
Linear Path Loss					X			
Specular Reflection						X		
Line-Of-Sight Signal								X
First-Order Reflection Signal								X
Received Signal Strength								X
Output								X
Specification Parameters								X

Table 3: Traceability Between Test Cases and Requirements