

## **Developing an Automated Testing Framework for Celestia**

College of Charleston

### **Team International**

Guilherme Costa

Megan Landau

Tony Tang

# Table of Contents

---

<b>Project Summary</b>	<b>2</b>
<b>Chapter 1 - Building Celestia</b>	<b>3</b>
1.1 A rocky start	3
1.2 Building progress	3
1.3 Project compiled	4
<b>Chapter 2 - Detailed test plan</b>	<b>5</b>
2.1 Report	5
2.2 Celestia Application Test Cases	6
<b>Chapter 3 - Automated Testing Framework</b>	<b>13</b>
3.1 Task and Framework Description	13
3.2 Team experience	14
3.3 How-To Documentation of test script	16
<b>Chapter 4 - Creating 25 test cases</b>	<b>17</b>
4.1 The test cases	17
4.1.1. More test cases	17
4.1.2 Finding the last method	18
4.1.3 Editing the HTML page with CSS	18
4.1.4 Analyzing our test results	19
4.2 Problems and Issues	19
<b>Chapter 5 - Fault Injection</b>	<b>21</b>
5.1 First Fault Injection	22
5.2 Second Fault Injection	23
5.3 Third Fault Injection	25
5.4 Fourth Fault Injection	26
5.5 Fifth Fault Injection	27
<b>Chapter 6 - Evaluations</b>	<b>29</b>
6.1 Experiences and lessons learned	29
6.2 Team International self-evaluation	29
6.3 Evaluation of Project	29

# Project Summary

---

This work aims to validate the claim that automated testing frameworks increase the visibility of errors in code. The Humanitarian / Free Open Source Software chosen for research was Celestia, a space simulation application that has been in development since the early 2000s. Celestia uses a custom math library to perform calculations during run time. By developing an automated testing framework that utilizes appropriate test cases, the strengths and weaknesses in Celestia's math library can be exposed easily by comparing expected outcomes to a method's actual outcome. Furthermore, the validity of the testing framework can be demonstrated by injecting fault into the code and then running the framework again: the faults in the new code will be immediately apparent as the pass-fail ratio will be altered dramatically.

The project assigned by Dr. Bowring in CSCI-362-02 Software Engineering required the team to create an automated testing framework for Celestia. Extensive research for building this project was conducted and the overall achievements are reflected and documented here. Each chapter of this document represents a deliverable produced by the team, with an additional chapter (Chapter 6) including evaluations of the team, and the project. The testing framework was developed to run on Ubuntu (Linux/Unix). The results obtained by the testing framework is displayed in table format for immediate evaluation. From this work, a practical assessment of Celestia's coding errors will be collected.

# Chapter 1 - Building Celestia

09.13.2016

## 1.1 A rocky start

Our team spent many hours struggling to download, compile, and run the source code for the mWater app. We worked independently during the week trying to understand the dependencies and permissions aspects of the mWater project and then met as a group on Sunday, September 11 at 8:30 pm to around 12:00 am trying to compile and run the mWater application. Unfortunately, time was lost due to searching for and installing correct versions of dependencies such as Grunt, Cordova, Bower, and Browserify. We were also having trouble with getting Minimatch, Lodash, and Graceful-fs to install successfully. However, we did get the mWater folders called "mWater-common" and "app\_v3" to load to the team's Github repository successfully. After trying to install the mWater application for several more hours, Megan decided to email Dr. Bowring to explain to him that we could not get the application to run. She met with Dr. Bowring around 3:00 pm today, September 12 and he advised our team to choose another project. Megan suggested our team choose Celestia and we all agreed.

## 1.2 Building progress

Update: 9:05pm

We went to the Celestia project website and clicked the download link and after a few minutes of loading the SourceForge website, the download would not run on the Ubuntu virtual machine. We went to this website: <https://sourceforge.net/projects/celestia/>, and downloaded the tar.gz file for the Celestia project. This file is for Celestia version 1.6.1 . After downloading the file, we extracted the file in the home directory and started to follow the instructions in the README file found in the Celestia 1.6.1 folder. Started installing the gtk version by using `./configure --with-gtk make` and `makefile` and then ran into some problems. First, we learned how to use the make file and then some errors were shown on the terminal. Next, we continued to try the other versions of the simulator using KDE3 and GNOME. Currently, we are unsuccessful with getting the simulator running on Linux at 9:05pm, Monday September 12. We will keep you updated...

Update: 2:27 AM

After a few more hours attempting to build the project we finally were able to build and compile Celestia! The main problem we had was probably the repository we were using was outdated.

We found a new repository at <https://github.com/bgodard/celestia-g2>, which had an updated version of Celestia that supposedly was never released. After copying the new repository and trying to build it, it required additional libraries that the old build did not have. The libraries and instructions on how to build it were found at [https://en.wikibooks.org/wiki/Celestia/Development/Qt4#NAIF\\_SPICE](https://en.wikibooks.org/wiki/Celestia/Development/Qt4#NAIF_SPICE). The Cspice package was extremely difficult to install due to the specific path it had to be in order to Celestia to run properly. Cspice package was found at [http://naif.jpl.nasa.gov/naif/toolkit\\_C.html](http://naif.jpl.nasa.gov/naif/toolkit_C.html). After compiling everything we were able to run Celestia. It runs very sluggish on the virtual machine due to the demanding graphics.

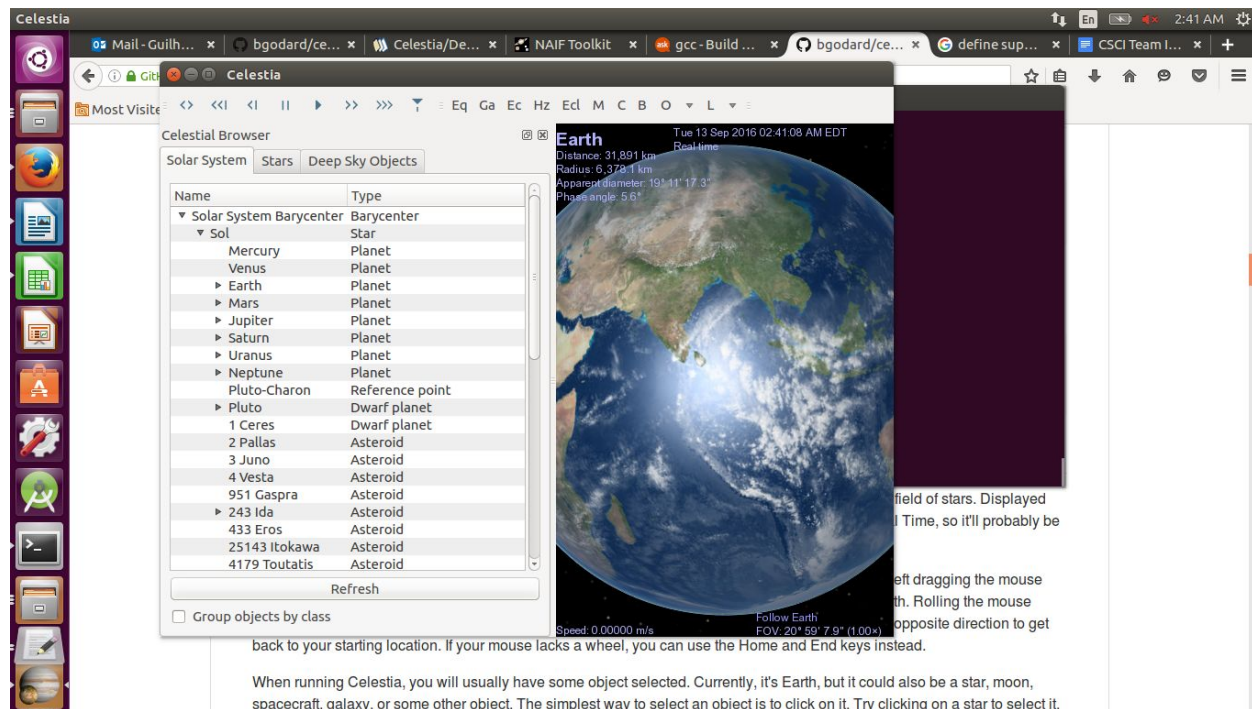


Figure 1: Screenshot of Celestia built and running.

## 1.3 Project compiled

Update: 2:54am

Our team is very pleased that Gui was able to compile and run the source code on his Ubuntu VM tonight. By Googling the Celestia project, we can see that it has been an ongoing project for many years. It had its final update in 2011 at version 1.6.1, which we are using for our project. There are many resources for compiling and running the code (this was a problem we were having with the mWater application - there was not enough information about it online for us to troubleshoot), including an older [Ubuntu Celestia package](#) and a helpful [WikiBooks](#)

[page](#). After many hours searching, both Gui and Megan found a GitHub repository that had the newer source code and much-improved installation instructions.

The Celestia application can be run with one of four different interfaces:

```

10 OK, assuming you've collected all the necessary libraries, here's
11 what you need to do to build and run Celestia:
12
13     ./configure --with-INTERFACE          [*]
14     make
15     make install
16
17 [*] INTERFACE must be replaced with one of "kde", "gtk", "gnome", or "glut".
18
19 Four interfaces are available for Celestia:
20 - glut: minimal interface, barebone Celestia core with no toolbar or menu...
21 - GTK: A full interface with minimal dependencies, adds a menu, a configuration
22     dialog some other utilities.
23 - Gnome: The full GTK interface plus a few Gnome integration goodies, such as
24     preference saving in GConf. This looks and works very much like the
25     Windows interface.
26 - KDE3: brings contextual menus, toolbars, KDE integration,
27     internationalization, bookmarks...
28
29 To build the KDE interface (requires various kde-devel packages):
30     configure --with-kde
31

```

*Figure 2: Interfaces available to run Celestia.*

There are libraries such as the jpeglib, pnglib, glu, and OpenGL to install before configuration. We also have to install the -dev versions as well.

We are very excited that Gui was able to compile and run the program from the command line - we have saved the terminal, but it is much too long to include in this report.

## Chapter 2 - Detailed test plan

09.26.2016

### 2.1 Report

The following are detailed test plans for our current project, Celestia. We have specified 5 eventual test cases that we have developed for our software. Some of our test cases have already been tested, and others are still being built for testing. In order to be concise we have a detailed template for each test. We have a Test Suite ID, which at the moment is TS001; Test

Case IDs which correspond to the test number; a Test Case Summary; Requirements; Components Being Tested; Methods Being Tested; Driver Being Tested; Test Inputs; Prerequisites; Test Procedures; Test Data; Expected Results; Actual Results; Status (pass or fail); Remarks; Created By; and finally the Date Of Creation. Each field mentioned above was carefully recorded with information pertaining to its test.

#### Useful links:

<http://softwaretestingfundamentals.com/test-case/>

<https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h>

## 2.2 Celestia Application Test Cases

<b>Test Case ID</b>	testCase001
<b>Test Case Summary</b>	This test case will compute the output of the square method located in the mathlib.h library of the celestia-g2 project found on github
<b>Requirement being tested</b>	Test the output of the square method
<b>Component being tested</b>	Square method found in mathlib.h library
<b>Method being tested</b>	square(T x)
<b>Driver being tested</b>	testDriverSquare.cpp
<b>Test input(s) including command-line arguments</b>	2
<b>Prerequisites</b>	Mathlib.h must be found in the same directory as the test driver
<b>Test Procedure</b>	<ul style="list-style-type: none"> <li>• First, get the mathlib.h file from the celestia-g2 repository found on github</li> <li>• Next, create testDriverSquare.cpp for the test case driver.</li> <li>• Make sure the mathlib.h file and .cpp file is in the same directory</li> <li>• Type <code>g++ test001 -o testDriverSquare.cpp</code> into the terminal in Linux Ubuntu 16.04</li> <li>• Type <code>./test001</code> to execute the file</li> <li>• Check to see if the expected outcome matches the actual outcome</li> <li>• Record results</li> </ul>
<b>Test Data</b>	<a href="https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b54290">https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b54290</a>

	78304901fa848fec2/celestia/src/celmath/mathlib.h
<b>Expected Result</b>	4
<b>Actual Result</b>	4
<b>Status</b>	Pass
<b>Remarks</b>	Test executed correctly, a was found to be 2 and the square of a is 4. The expected outcome was 4. So, the test passed.
<b>Created by</b>	Gui, Megan, Tony
<b>Date of Creation</b>	September 23, 2016 5:18 pm
<b>Executed by</b>	Gui, Megan, Tony
<b>Date of Execution</b>	September 22, 2016 12:50 pm
<b>Test environment</b>	OS: Linux Ubuntu 16.04

<b>Test Case ID</b>	testCase002
<b>Test Case Summary</b>	Testing that the method can properly determine the mathematical outcome when finding area of a circle.
<b>Requirement being tested</b>	Test the output of the circleArea method
<b>Component being tested</b>	circleArea method found in mathlib.h library
<b>Method being tested</b>	circleArea(T r)
<b>Test driver</b>	testDriverCircleArea.cpp
<b>Test inputs including command-line arguments</b>	10
<b>Prerequisites</b>	Mathlib.h must be found in the same directory as the test driver
<b>Test Procedure</b>	<ul style="list-style-type: none"> <li>• First, get the mathlib.h file from the celestia-g2 repository found on github</li> <li>• Next, create testDriverCircleArea.cpp for the test case driver.</li> <li>• Make sure the mathlib.h file and .cpp file is in the same directory</li> <li>• Type g++ test002 -o testDriverCircleArea.cpp into the terminal in Linux Ubuntu 16.04</li> <li>• Type ./test002 to execute the file</li> <li>• Check to see if the expected outcome matches the actual</li> </ul>



	outcome <ul style="list-style-type: none"> <li>Record results</li> </ul>
<b>Test Data</b>	<a href="https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h">https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h</a>
<b>Expected Result</b>	314.15926...
<b>Actual Result</b>	314.159
<b>Status</b>	Pass
<b>Remarks</b>	The method rounded two decimals.
<b>Created by</b>	Gui, Megan, Tony
<b>Date of Creation</b>	September 23, 2016 5:18 pm
<b>Executed by</b>	Gui, Megan, Tony
<b>Date of Execution</b>	September 23, 2016 6:11 pm
<b>Test environment</b>	OS: Linux Ubuntu 16.04

<b>Test Case ID</b>	testCase003
<b>Test Case Summary</b>	Testing the cube method found in the mathlib.h library.
<b>Requirement being tested</b>	Test the output of the cube method
<b>Component being tested</b>	Use a test driver program created in C++ to test the given method
<b>Method being tested</b>	cube (T x)
<b>Test driver</b>	testDriverCube.cpp
<b>Test inputs including command-line arguments</b>	2
<b>Prerequisites</b>	Mathlib.h must be found in the same directory as the test driver
<b>Test Procedure</b>	<ul style="list-style-type: none"> <li>First, get the mathlib.h file from the celestia-g2 repository found on github</li> <li>Next, create testDriverCube.cpp for the test case driver.</li> <li>Make sure the mathlib.h file and .cpp file is in the same directory</li> <li>Type g++ test003 -o testDriverCube.cpp into the terminal in</li> </ul>

	Linux Ubuntu 16.04 <ul style="list-style-type: none"> <li>• Type ./test003 to execute the file</li> <li>• Check to see if the expected outcome matches the actual outcome</li> <li>• Record results</li> </ul>
<b>Test Data</b>	<a href="https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h">https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h</a>
<b>Expected Result</b>	8
<b>Actual Result</b>	8
<b>Status</b>	Pass
<b>Remarks</b>	Output resulted in 8 which matched the predicted outcome. This method works similarly to the square method.
<b>Created by</b>	Gui, Megan, Tony
<b>Date of Creation</b>	September 23, 2016 5:18 pm
<b>Executed by</b>	Gui, Megan, Tony
<b>Date of Execution</b>	September 23, 2016 5:49 pm
<b>Test environment</b>	OS: Linux Ubuntu 16.04

<b>Test Case ID</b>	testCase004
<b>Test Case Summary</b>	Testing the sphereArea method found in the mathlib.h library
<b>Requirement being tested</b>	Test the output of the sphereArea method
<b>Component being tested</b>	Use a test driver program created in C++ to test the given method
<b>Method being tested</b>	sphereArea(T r)
<b>Test driver</b>	testDriverSphereArea.cpp
<b>Test inputs including command-line arguments</b>	5
<b>Prerequisites</b>	1. Mathlib.h must be found in the same directory as the test driver
<b>Test Procedure</b>	<ul style="list-style-type: none"> <li>• First, get the mathlib.h file from the celestia-g2 repository found on github</li> <li>• Next, create testDriverSphereArea.cpp for the test case driver.</li> </ul>

	<ul style="list-style-type: none"> <li>• Make sure the mathlib.h file and .cpp file is in the same directory</li> <li>• Type <code>g++ test004 -o testDriverSphereArea.cpp</code> into the terminal in Linux Ubuntu 16.04</li> <li>• Type <code>./test004</code> to execute the file</li> <li>• Check to see if the expected outcome matches the actual outcome</li> <li>• Record results</li> </ul>
<b>Test Data</b>	<a href="https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h">https://github.com/bgodard/celestia-g2/blob/89412cd52964b00e7b5429078304901fa848fec2/celestia/src/celmath/mathlib.h</a>
<b>Expected Result</b>	314
<b>Actual Result</b>	310
<b>Status</b>	Pass
<b>Remarks</b>	Found the output for the surface area of a sphere. The expected result matched the actual result.
<b>Created by</b>	Gui, Megan, Tony
<b>Date of Creation</b>	September 23, 2016 6:20 pm
<b>Executed by</b>	Gui, Megan, Tony
<b>Date of Execution</b>	September 23, 2016 7:58 pm
<b>Test environment</b>	OS: Linux Ubuntu 16.04

<b>Test Case ID</b>	testCase005
<b>Test Case Summary</b>	Testing the readChar() method to ensure that it reads input stream and converts it to a char correctly
<b>Requirement being tested</b>	Char conversion and recognizability, exception handling
<b>Component being tested</b>	3dsread.cpp
<b>Method being tested</b>	readChar()
<b>Test driver</b>	testDriverReadChar.cpp
<b>Test inputs including command-line arguments</b>	char t
<b>Prerequisites</b>	1. 3dsread.cpp file has compiled and ran

<b>Test Procedure</b>	<ul style="list-style-type: none"> <li>• First, get the 3dsread.cpp file from the celestia-g2 repository found on github</li> <li>• Next, create testDriverReadChar.cpp for the test case driver.</li> <li>• Make sure the 3dsread.cpp file is in the same directory</li> <li>• Type g++ test005 -o testDriverReadChar.cpp into the terminal in Linux Ubuntu 16.04</li> <li>• Type ./test005 to execute the file</li> <li>• Check to see if the expected outcome matches the actual outcome</li> <li>• Record results</li> </ul>
<b>Test Data</b>	<a href="https://github.com/bgodard/celestia-g2/blob/master/celestia/src/cel3ds/3dsread.cpp">https://github.com/bgodard/celestia-g2/blob/master/celestia/src/cel3ds/3dsread.cpp</a>
<b>Expected Result</b>	t
<b>Actual Result</b>	----- (not yet recorded)
<b>Status</b>	----- (pass/fail not yet recorded)
<b>Remarks</b>	-----
<b>Created by</b>	Gui, Megan, Tony
<b>Date of Creation</b>	September 23, 2016 6:21 pm
<b>Executed by</b>	Gui, Megan, Tony
<b>Date of Execution</b>	-----
<b>Test environment</b>	OS: Linux Ubuntu 16.04

## Screenshots:

The screenshot shows a C++ IDE with a terminal window. The terminal displays the output of the testDriverSphereArea.cpp program. The code in the terminal window is as follows:

```

1 #include <iostream>
2 #include "mathlib.h"
3 int main()
4 {
5
6     using namespace std;
7     cout<< "Testing sphereArea method..."<< endl;
8     int r = 5;
9     int a = sphereArea(r);
10
11     cout << "The sphere area with a radius of 5 is: " << a << endl;
12 }

```

The terminal output shows the following:

```

yo@yo:~/Desktop/362testcases$ g++ testDriverSphereArea.cpp -o test1
yo@yo:~/Desktop/362testcases$ ./test1
Testing sphereArea method...
The sphere area with a radius of 5 is: 300
yo@yo:~/Desktop/362testcases$ cat testDriverSphereArea.cpp
#include <iostream>
#include "mathlib.h"
int main()
{
    using namespace std;
    cout<< "Testing sphereArea method..."<< endl;
    int r = 5;
    int a = sphereArea(r);
    cout << "The sphere area with a radius of 5 is: " << a << endl;
}
yo@yo:~/Desktop/362testcases$ g++ testDriverSphereArea.cpp -o test3
yo@yo:~/Desktop/362testcases$ ./test3
Testing sphereArea method...
The sphere area with a radius of 5 is: 300
yo@yo:~/Desktop/362testcases$

```

Figure 3: Testing the sphereArea() method from mathlib.h.

The screenshot shows a C++ IDE with a terminal window. The terminal displays the output of the testDriverCube.cpp program. The code in the terminal window is as follows:

```

1 #include <iostream>
2 #include "mathlib.h"
3 int main()
4 {
5
6     using namespace std;
7     cout<< "Starting cube method testing..."<< endl;
8     int a = 2;
9     int c = cube(a);
10
11     cout << "The cube of 2 is: " << c << endl;
12 }

```

The terminal output shows the following:

```

yo@yo:~/Desktop/362testcases$ g++ testDriverCube.cpp -o test4
yo@yo:~/Desktop/362testcases$ ./test4
Starting cube method testing...
The cube of 2 is: 8
yo@yo:~/Desktop/362testcases$

```

Figure 4: Testing the cube() method from mathlib.h.

The screenshot shows a terminal window with a code editor interface. The editor has several tabs open: `shell.c`, `hw2.c`, `shell.h`, `mathlib.h`, `testDriverSphereArea.cpp`, `testDriverSphereArea.cpp`, `testDriverCube.cpp`, and `testDriverSquare.cpp`. The active tab is `mathlib.h`, which contains the following C++ code:

```

1 #include <iostream>
2 #include "mathlib.h"
3 int main()
4 {
5
6     using namespace std;
7     cout << "Starting square method testing..." << endl;
8     int a = 2;
9     int s = square(a);
10
11     cout << "The square of 2 is : " << s << endl;
12 }

```

Below the editor, a terminal window shows the execution of the program. The user runs `./test4` and `./test5`, which compile and run the test driver programs. The output shows the square of 2 is 4.

```

yo@yo:~/Desktop/362testcases$ ./test4
Starting cube method testing...
The cube of 2 is : 8
yo@yo:~/Desktop/362testcases$ g++ testDriverSquare.cpp -o test5
yo@yo:~/Desktop/362testcases$ ./test5
Starting square method testing...
The square of 2 is : 4
yo@yo:~/Desktop/362testcases$ cat testDriverSquare.cpp
#include <iostream>
#include "mathlib.h"
int main()
{
    using namespace std;
    cout << "Starting square method testing..." << endl;
    int a = 2;
    int s = square(a);

    cout << "The square of 2 is : " << s << endl;
}
yo@yo:~/Desktop/362testcases$ ./test5
Starting square method testing...
The square of 2 is : 4
yo@yo:~/Desktop/362testcases$

```

Figure 5: Testing the `square()` method from `mathlib.h`.

## Chapter 3 - Automated Testing Framework

10.21.2016

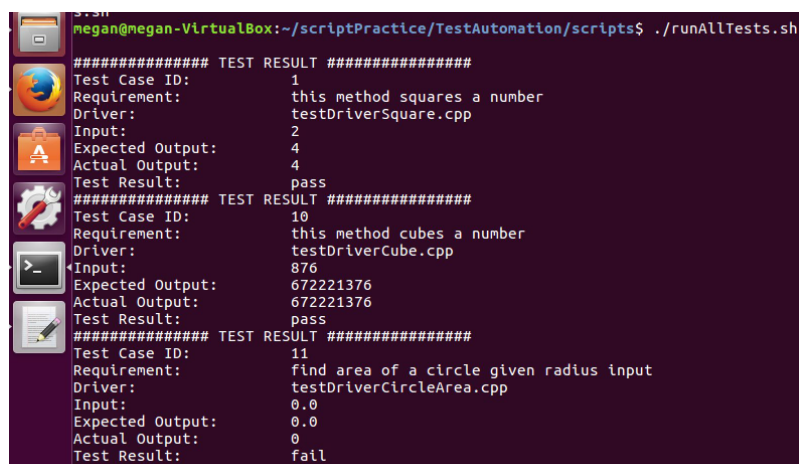
### 3.1 Task and Framework Description

- Description of the task:
  - The task is to build a testing framework using a structure that separates the test case text files into one folder (TestCases); the program executables in another (TestCasesExecutables); and the main script and any helper scripts in other (Scripts). The main script is all that needs to be run via the command line and will output a report in the form of an .html file to the Reports folder.
- Architectural description of project framework:
  - The project's framework was organized by partitioning the test cases, scripts, reports, and executables in different folders. This type of architectural allowed the project to have a uniform structure. The test script inside the scripts folder could

easily navigate through other directories by using the command “../directory” and then perform the task needed in that directory.

## 3.2 Team experience

- The team met in three stages: first, we met to discuss the task and come up with a plan for how the testing framework would be implemented and how we could accomplish the above task. Our first challenge was creating the test case .txt files, which we were unsure how to parse using a bash script. After deciding the format for the test case files, we created a few based on our work from Deliverable #2.
- We had already created a .cpp driver for the square() method and knew how to compile and run it from the command line. So our next step was to write a bash script that would parse a testCase.txt file, make a variable out of line 15 (which is the **Input** value) and then pass that variable to the compiled driver.
- Our next struggle was to receive a return value back from the driver which we could then store into a variable in order to compare the **Actual Output** with the **Expected Output**. We discussed the problem with Dr. Bowring and he suggested we use the driver to create a temporary .txt file and then read that text file and compare the values using this procedure. We believed, however, that there had to be a way to create a variable from the return value of the driver and use the “==” equals comparator to determine any difference in the two values.
- After successfully figuring this out, the team was able to determine whether a single testCase.txt file had a pass / fail outcome.
- With this achieved, the team then used a for loop to run through all of the testCase.txt files within the **testCases** folder of the framework. The script would then output all the stored variables read from the testCase.txt file and the results of the test. This is shown below:



```

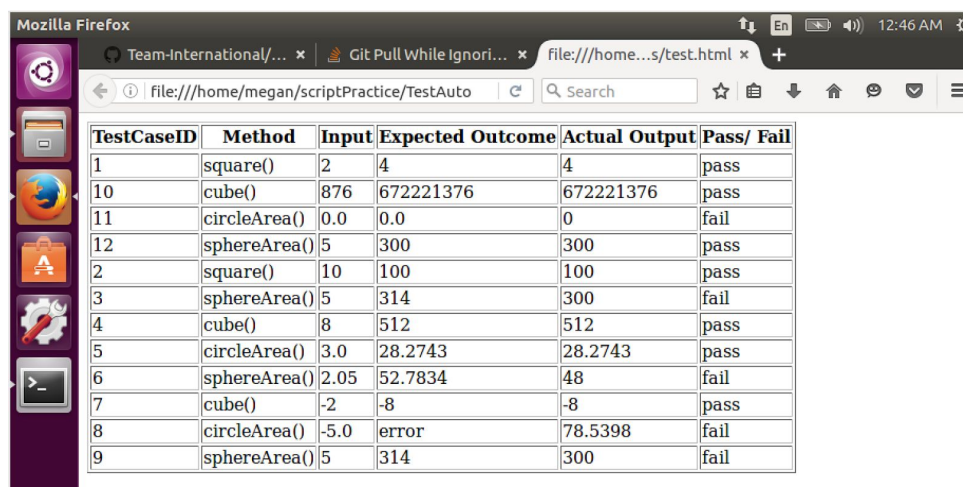
bash
megan@megan-VirtualBox:~/scriptPractice/TestAutomation/scripts$ ./runAllTests.sh

##### TEST RESULT #####
Test Case ID:      1
Requirement:       this method squares a number
Driver:            testDriverSquare.cpp
Input:             2
Expected Output:   4
Actual Output:     4
Test Result:       pass
##### TEST RESULT #####
Test Case ID:      10
Requirement:       this method cubes a number
Driver:            testDriverCube.cpp
Input:             876
Expected Output:   672221376
Actual Output:     672221376
Test Result:       pass
##### TEST RESULT #####
Test Case ID:      11
Requirement:       find area of a circle given radius input
Driver:            testDriverCircleArea.cpp
Input:             0.0
Expected Output:   0.0
Actual Output:     0
Test Result:       fail

```

Figure 6: Test cases results displayed in terminal window.

- We are still having some troubles with conversions between Int and Float values (shown in Test Case ID 11 above) and we hope to resolve this in order to have our successful test cases pass if they should be passing.
- Our next challenge was to append the results of the tests to an .html file which would open a browser window displaying a table of the pass / fail outcomes. First we coded echos and learned how to create a simple table. We then coded a loop so we could append multiple rows to the table and finally, we were able to populate the table with actual values gleaned from the testCase.txt files.
- Then we used what we learned from the first bash script we wrote months ago (myList.sh) to command the .html file to open up in a browser window:



TestCaseID	Method	Input	Expected Outcome	Actual Output	Pass/ Fail
1	square()	2	4	4	pass
10	cube()	876	672221376	672221376	pass
11	circleArea()	0.0	0.0	0	fail
12	sphereArea()	5	300	300	pass
2	square()	10	100	100	pass
3	sphereArea()	5	314	300	fail
4	cube()	8	512	512	pass
5	circleArea()	3.0	28.2743	28.2743	pass
6	sphereArea()	2.05	52.7834	48	fail
7	cube()	-2	-8	-8	pass
8	circleArea()	-5.0	error	78.5398	fail
9	sphereArea()	5	314	300	fail

*Figure 7: Test cases results displayed in web browser using HTML.*

- Success! Our next steps will be to have the test cases output to the .html file in order so that the table is easier to read. We will also be adding text to the .html file so that it explains what the table is showing, gives an explanation of the Celestia project and some information about who we are and why we are testing it. We are also planning on formatting the .html file so that it is more visually appealing. We also have a tentative plan to order the table information by test suites (suites will be based on grouping the test cases together according to the method they will be testing).
- Furthermore, we plan on adding different types of test cases that test other components from the Celestia project besides the mathlib.h library. We also need to add at least 13 more test cases, which should be relatively easy now since we have a system in place. This means we will also be creating two more driver .cpp files in order to test other components or methods.
- Overall, the team is very excited about the distance we have come since beginning this project. We all started out with minimal experience in shell scripting and little to no experience with GitHub and now we have a script that navigates our testing framework and also almost 100 commits to our GitHub repository. We have written in bash and C++ and have been able to compile and run programs from the terminal, which none of us



had been adept at before this semester. So, overall, we are very happy with our progress and seem to be working very efficiently and are content with the amount of collaboration amongst our team members.

### 3.3 How-To Documentation of test script

The following information provides steps for how to run the automated script named **runAllTests.sh**. These steps explain how to download and run this shell script using the command line in a **Linux OS**.

1. First, click on the clone button on the Team-International repository on Github, the button is named [Clone with HTTPS]
2. Copy the file path for the repository
3. Next, enter git clone followed by the name of the file path to clone the Team-International repository from Github

```
git clone https://github.com/CSCI-362-02-2016/Team-International.git
```

4. Then, make a folder and name it something, for example teaminternationalrepo
5. This will be where the Team-International github repository will be located on your computer
6. The following are the instructions needed in order to run the TestAutomation script
7. Next, type git pull to obtain the current working repository from Github

```
git pull
```

8. Type in your username
9. Type in your password
10. Press Enter

Now, you are ready to run the automated script. In order to run **runAllTests.sh**, you will need to navigate to the script folder located in the TestAutomation directory with the command line.

11. Type "ls" in the terminal to display the files in the current directory. You will see a project folder named "scripts"
12. Inside this folder, enter **./runAllTests.sh** to run the script

```
./runAllTests.sh
```

At this point you should see the test cases printed in the terminal as they are run and a browser window should open with the test results displayed in an HTML table.

## Chapter 4 - Creating 25 test cases

11.14.2016

### 4.1 The test cases

#### 4.1.1. More test cases

- For this deliverable, we added more test cases to the initial cases we had from **Deliverable#3** for a total of 25 test cases. Initially, we had already written 12 test cases for Deliverable #3. We began by adding 13 more test cases for **Deliverable #4** and checking to see if our test cases really tested a good variety of inputs. We added 3 more test cases for the **square** method which is found in the **mathlib.h** library of the Celestia repository. Then we added 2 more test cases for the **cube** method. We added 2 more test cases for the **sphere area** method and 3 more test cases for the **circle area** method.

### 4.1.2 Finding the last method

- Lastly, we had to find one more method and create 5 test cases for a complete total of 25 test cases for the entire project.
  - Initially, we wanted to test the method, **readChar**, which is found in the **3dsread.cpp** file. This method is supposed to read an input stream and convert it to a char correctly. The readChar method has inputs that are of variable type char. The return type for the readChar method is a boolean. Unfortunately, the file 3dsread.cpp had too many dependencies and we were unable to create a test driver that could successfully perform the readChar method.
- After failing to use the readChar method, we decided to look into libraries other than the mathlib.h library to find a method that was suitable for testing. First, we found a method that used a timer. We ended up finding a method in the **astro.h** library called **secsToDays**, this method would convert time in seconds to a number of days. The input variable is a double. However, this method also had too many dependencies and we were unable to run the test driver to display the correct results. In the end, we decided to ask for permission from our professor if we could use another method found in the mathlib.h library because we knew what most of the expected outcomes would be and were able to successfully calculate them.
  - We finally settled on a method called **radToDeg** found in the mathlib.h library which converts radians to degrees based on a radian input. This method worked similarly to other methods we have been using. We decided to test the radToDeg method with input variables of 0, 5, 1, -57, and 125,000,000 to test a wide variety of possibilities.

### 4.1.3 Editing the HTML page with CSS

- Next, we decided to add to the aesthetics of our HTML table. Below are a “Before” and “After” screenshot of the table (See *Figure 1* and *Figure 2*).
- First, we wanted to display our team name and logo above the results table because we thought this would make our table look more professional and cohesive with our Wiki page.
- Next, we changed the font for the data in the table to Helvetica because it adds a simple, clean look to the page.
  - We wanted to style the HTML table for it to be easier to read so we decided to alternate the row colors between the rows between gray and white. We made the even rows to be #EAEAEA which is a grayish color.
  - Then, we wanted to write a little description of what the table results show. For example, the table shows the number of the test cases, the name of the method, the expected outcomes, the actual outcome, and whether the test case pass or failed.

- We also added the name of the group members that worked on the project and the original project.
- Next, for additional styling we added a page border in gray. We plan to add links at the bottom of the HTML table to our GitHub account and blogs.

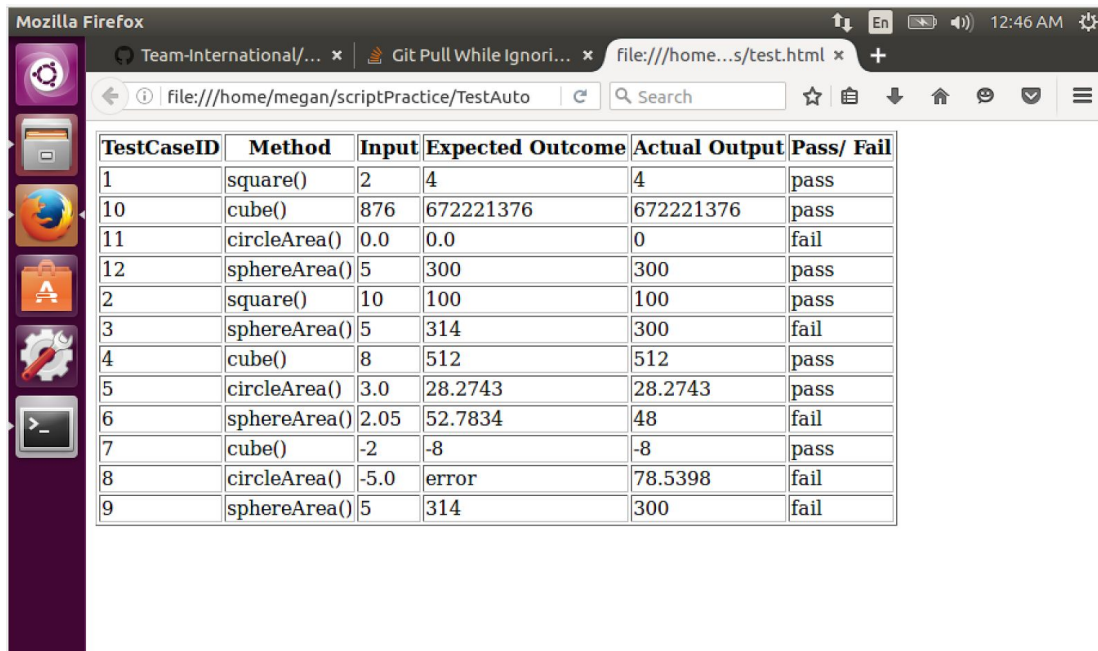
#### 4.1.4 Analyzing our test results

- Lastly, we started looking at the results from the tests. Some of our tests cases were failing due to rounding errors or problems with how the math library defines the number Pi.
- We decided to make a note of these specifications on the HTML page because it is important to note what restrictions the Celestia project places on mathematical formulas.
- We also wanted to make sure all decimals were rounded to four places, which is how the Celestia math library uses decimal point precision.
- Another problem we had was when dealing with the exponential notation. We did not know what was the minimum value that a integer value will be converted into exponential notation, but we think it is to the  $10^5$ th power.

## 4.2 Problems and Issues

- Our script running the drivers had to be executed from inside the /scripts directory. We had to change it to meet the project requirements: the project requires the script to be executed from the /TestAutomation directory using the command ./scripts/runAllTests.sh. It was a learning experience learning how to change directories using Bash, but at this point we are very comfortable with Bash. The team worked together to overcome this problem and we were able to provide a solution to make the script work as required.
- The math library was also generating answers with only 4 decimals. We had to run tests and we determined that, by default, the precision of the custom math library used by celestia was only to 4 decimal places.
- The team spent a lot of time on the HTML report. We felt that it was an important characteristic of the project to provide an exceptional report. We had major difficulties customizing the HTML, since for some of the team members it was their first experience with HTML. We learned how to properly use tags and how to format text to provide a nice visual effect on the report. The HTML report reflects the hard work and determination of the team as we made sure that it is of good quality and easy to understand. Adding a picture to the HTML report proved to be a challenge since we had to make changes to the script itself to apply the picture on the report. After many different combinations of table styles, font sizes, and colors we all agreed that the final HTML report was perfect.

- The team is running into problems adding a pdf file to the GitHub wiki main page. We have converted Deliverables #1-3 into Markdown type but we want to add the actual pdf files to the main wiki page.



TestCaseID	Method	Input	Expected Outcome	Actual Output	Pass/ Fail
1	square()	2	4	4	pass
10	cube()	876	672221376	672221376	pass
11	circleArea()	0.0	0.0	0	fail
12	sphereArea()	5	300	300	pass
2	square()	10	100	100	pass
3	sphereArea()	5	314	300	fail
4	cube()	8	512	512	pass
5	circleArea()	3.0	28.2743	28.2743	pass
6	sphereArea()	2.05	52.7834	48	fail
7	cube()	-2	-8	-8	pass
8	circleArea()	-5.0	error	78.5398	fail
9	sphereArea()	5	314	300	fail

Figure 1. Before screenshot of HTML report presented on deliverable #3.

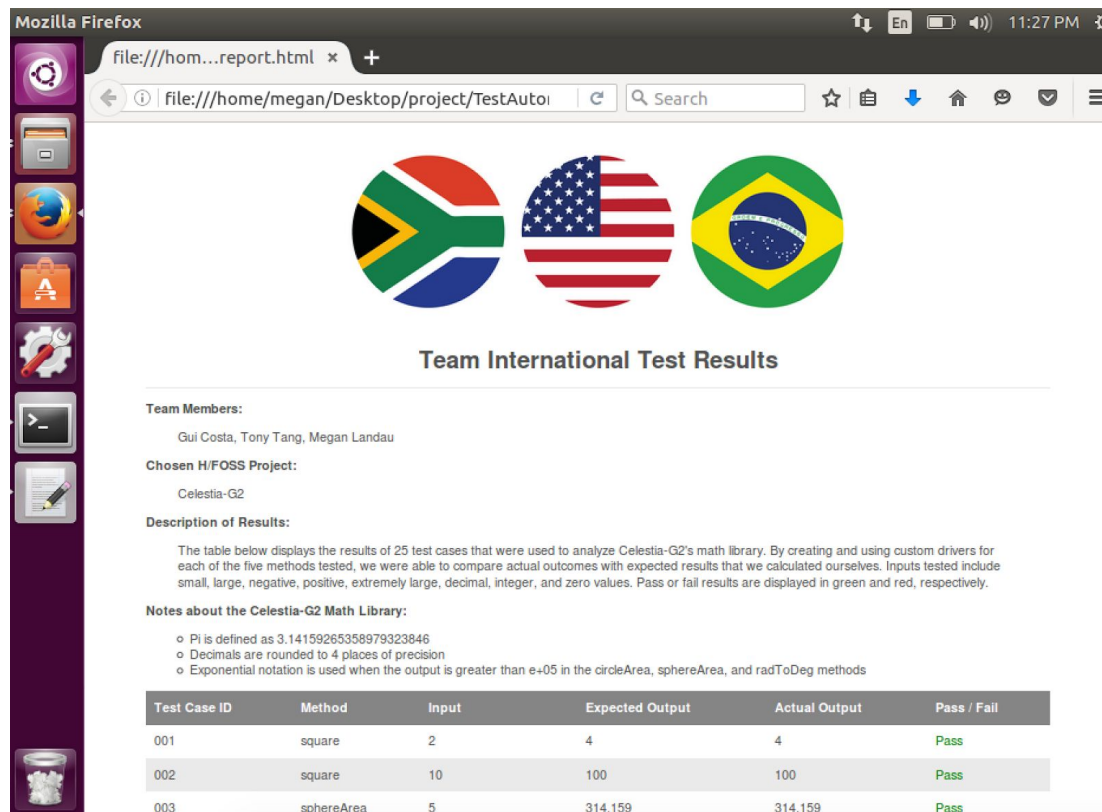


Figure 8. After screenshot of HTML report after it has been altered for deliverable #4.

## Chapter 5 - Fault Injection

11.29.2016

**Task:** Design and inject 5 faults into the code you are testing that will cause at least 5 tests to fail, but hopefully not all tests to fail.

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

Figure 9: Test results before fault injections.

Below are detailed descriptions of what faults were injected, how each method was changed, which test cases were affected by the fault injection, a figure of the Test Results Table after fault injection, and an analysis of the results.

## 5.1 First Fault Injection

- **Description:** Inject a fault into method radToDeg method found in the mathlib.h library. Changed the first multiplication operation in the return statement to division.

- **Original Method:**

```
template<class T> T radToDeg(T r)
{
    return r * 180 / static_cast<T>(PI);
}
```

- **Method with fault injection:**

```
template<class T> T radToDeg(T r)
{
    return r / 180 /static_cast<T>(PI);
}
```

- **Test cases failed:** testCase009, testCase022, testCase023, and testCase025
- **Test results table:**

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	0.00884194	Fail
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	0.00176839	Fail
023	radToDeg	-57	-3265.86	-0.100798	Fail
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	221049	Fail

Figure 10: Test results after first fault injection

- **Analysis:** According to the original results test cases 005, 009, 0022, and 0023 passed while test case 0025 failed. After the fault was injected test case 005 was the only test case to pass while test cases 009, 022, 023, and 025 failed. Test cases 009, 022, and 023 were the only test cases to change from a pass to a fail. The result of test case 005 and test case 025 remained the same.

## 5.2 Second Fault Injection

- **Description:** Inject a fault into the square method found in the mathlib.h library. Changed the multiplication operator to a division operator.
- **Original method:**

```
template<class T> T square(T x)
{
    return x * x;
}
```



- **Method with fault injection:**

```
template<class T> T square(T x)
{
    return x / x;
}
```

- **Test cases failed:** testCase001, testCase002, testCase013, testCase014, testCase015

- **Test results table:**

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	1	Fail
002	square	10	100	1	Fail
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1	Fail
014	square	800000	640000000000	1	Fail
015	square	0	0		Fail
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

*Figure 11: Test results after second fault injection.*

- **Analysis:** Before the fault was injected test cases 001, 002, and 015 passed while test cases 013 and 014 failed. After the fault was injected all test cases which includes 001, 002, 013, 014, and 015 failed. Test cases 013 and 014 resulted in a fail before the fault was injected and after, meaning the fault had no effect on the outcome of the test. In other words, test cases 013 and 014 failed both times. Test cases 001, 002, 013, 014, and 015 all failed after the fault was injected because the division operator was dividing the inputted variable by itself. This resulted in an actual output of 1 for test cases 001,002,013, and 014. Test case 015 resulted in a fail because the actual output could not be calculated due to division by 0.

## 5.3 Third Fault Injection

- **Description:** Inject a fault into sphereArea method found in the mathlib.h library. Changed the multiplication operator to a division operator for the first multiplication operator only.

- **Original method:**

```
template<class T> T sphereArea(T r)
{
    return 4 * (T) PI * r * r;
}
```

- **New method:**

```
template<class T> T sphereArea(T r)
{
    return 4 / (T) PI * r * r;
}
```

- **Test cases failed:** testCase003, testCase006, testCase018, testCase019

- **Test results table:**

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	31.831	Fail
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	5.35079	Fail
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.27324e+10	Fail
019	sphereArea	-300.05	error	114630	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	15555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

Figure 12: Test results after third fault injection.

- Analysis:** Before the fault was injected into the sphereArea method, test cases 003, 006, 012, and 018 passed while test case 019 failed. After the fault was injected into the sphereArea method, test cases 003, 006, 018, and 019 failed while only test case 012 passed. Test cases 012 and 019 did not change the outcome of pass or fail due to the fault injection. Test case 012 resulted in a pass outcome before the fault injection and after the fault injection. Test case 019 resulted in a fail outcome before the fault injection and after the fault injection. Test case 012 passed after the fault injection because the input variable was 0. It is interesting to note that there was no empty value for the actual output for test case 012 and it did not result in a division by zero error.

## 5.4 Fourth Fault Injection

- Description:** Inject a fault into circleArea method found in the mathlib.h library. Changed all multiplication operators to division operators.

- Original method:**

```
template<class T> T circleArea(T r)
{
    return (T) PI * r * r;
}
```

- New method:**

```
//inject a fault into circleArea method
template<class T> T circleArea(T r)
{
    return (T) PI / r / r;
}
```

- **Test cases failed:** testCase008, testCase011, testCase020, testCase021, testCase024
- **After fault injection:**

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	512	Pass
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	-8	Pass
008	circleArea	-5.0	error	0.125664	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	6.72221e+08	Fail
011	circleArea	0.0	0	inf	Fail
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	-1.07374e+09	Fail
017	cube	2.1111	9.4088	9.40863	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	0.0314159	Fail
021	circleArea	155555555555	7.6e+22	1.29831e-22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	1.25664e-05	Fail
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

Figure 13: Test results after fourth fault injection.

- **Analysis:** Before the fault was injected into the circleArea method test cases 011, 020, and 024 passed while test cases 008 and 021 failed. After the fault was injected, test cases 008, 011, 020, 021, and 024. All test cases that tested the circleArea method failed after changing all the multiplication operators to division operators. Also, on testCase 011 after the fault injection, the actual output was found to be inf, shown on the table above. When searching google, this means infinity in C++ and the test case resulted in fail because 0 does not equal infinity.

## 5.5 Fifth Fault Injection

- **Description:** Inject a fault into cube method in the mathlib.h library. Changed all of the multiplication operators to subtraction operators.

- **Original method:**

```
template<class T> T cube(T x)
{
    return x * x * x;
}
```

- **New Method:**

```
template<class T> T cube(T x)
{
    return x - x - x;
}
```

- **Test cases failed:** testCase004, testCase007, testCase010, testCase016, testCase017

- **Test results table:**

Test Case ID	Method	Input	Expected Output	Actual Output	Pass / Fail
001	square	2	4	4	Pass
002	square	10	100	100	Pass
003	sphereArea	5	314.159	314.159	Pass
004	cube	8	512	-8	Fail
005	radToDeg	0	0	0	Pass
006	sphereArea	2.05	52.8102	52.8102	Pass
007	cube	-2	-8	2	Fail
008	circleArea	-5.0	error	78.5398	Fail
009	radToDeg	5	286.479	286.479	Pass
010	cube	876	672221376	-876	Fail
011	circleArea	0.0	0	0	Pass
012	sphereArea	0	0	0	Pass
013	square	-98209	9645007681	1055073089	Fail
014	square	800000	640000000000	49872896	Fail
015	square	0	0	0	Pass
016	cube	-1024	-1073741824	1024	Fail
017	cube	2.1111	9.4088	-2.1111	Fail
018	sphereArea	100000	1.25664e+11	1.25664e+11	Pass
019	sphereArea	-300.05	error	1.13135e+06	Fail
020	circleArea	10	314.159	314.159	Pass
021	circleArea	155555555555	7.6e+22	7.60188e+22	Fail
022	radToDeg	1	57.2958	57.2958	Pass
023	radToDeg	-57	-3265.86	-3265.86	Pass
024	circleArea	500	785398	785398	Pass
025	radToDeg	125000000	7161972439.15	7.16197e+09	Fail

*Figure 14: Test results after fifth fault injection*

- **Analysis:** Before the fault injection of the cube method, test cases 004 and 007 passed while test cases 010, 016, and 017 failed. After the fault injection, test cases 004, 007, 010, 016, and 017 all failed due to changing all of the multiplication operators to subtraction operators. When changing these

operators, all of the test cases would fail because the actual output is vastly different from the expected output. For example, look at test case 004, the expected outcome was 512 but after the fault injection, the actual output was -8. These numbers are really spread apart from being the same value. Test cases 010, 016, and 017 resulted in a fail status before and after the fault injection. Test cases 004 and 007 changed from pass to fail after the fault injection.

## Chapter 6 - Evaluations

### 6.1 Experiences and lessons learned

During the execution of this project the team was introduced to new programming languages and methods which the team was excited to learn. The first major new experience we had was the introduction of Ubuntu using the Linux Operating System. The team had no experience using Ubuntu and it took us the first few weeks of the semester to completely get used to the new system. After learning how to navigate the Linux terminal we faced a new task also unfamiliar to the team. We were asked to create a script using Bash or Python. Despite Bash being a bigger challenge to the team, we agreed to learn and use it as our scripting language. We learned a lot of scripting and we were able to deliver an exceptional script. The project progressed and we were asked to develop a testing framework. During lectures we learned how to design and implement a testing framework before actually implementing the code. After we had the plan for the framework we were able to create one for the Celestia program using our newly learned Bash language.

Working with Celestia opened up opportunities for the team to learn C++, a language not explicitly taught in our major at College of Charleston. Learning the different syntax proved to be a challenge, but we were able to relate some to Java, which we all feel comfortable using. We learned how to compile, and built individual methods from a library being used by Celestia. The Celestia program proved to be a valuable choice for us despite the lack of documentation it presented. To display the test results from Celestia the team had to sharpen our skills in HTML in order to produce a table of results in a browser. During this phase we learned how to combine the bash script, with the results of the Celestia C++ program to produce an HTML file. All this became easier when the team started to actively use GitHub.

Learning version control with Git was one of the most valuable experiences we had during the lifetime of the project. Since most of the project was done together we would all tackle different parts of the task, and then we would use git to merge the changes. We developed important skills with Git by pushing changes, pulling, merging, committing and adding. After we effectively learned how to use Git the code sharing of the project became effortless. We also learned how to write clean code and to document, assuming other programmers would checkout our code and be able to understand it.

Working on this project has taught us how to work in groups and to rely on one another. A project this big would be too difficult and time consuming to be implemented by a single person. We as a group created multiple deliverable documents which expanded our knowledge on how to properly make them and deliver them as presentations in front of an audience. The lessons learned in this project has furthered our knowledge in Computer Science and software testing.

## 6.2 Team International self-evaluation

The team progressed immensely despite a difficult start. Most of the new programming languages and techniques introduced were unknown to the team. The team has faced many challenges and we all put our best attitude forward to overcome them. To accommodate everyone's schedule we sometimes had to meet late nights on the weekends. Despite our schedule conflicts we all had a positive attitude whenever we met. All the members worked the best of their abilities and we respected each other's weaknesses. Our deliverables and presentations are a genuine representation of the team's hard work. We pushed each other to our full potentials even when in some nights working on the project we felt overwhelmed with the task. We all strongly believe that the project have provided us with an large amount of experience for our future careers. The team has learned how to work as a group and it has benefited us tremendously when completing tasks. Most of the work done for this project was completed together, and this facilitated our project when making decisions. The team was able to offer different perspectives when developing as the project progressed. As the project became more difficult, we always managed to get our project goals done in time with pure satisfaction. The team always helped each other if a member was stranded in a specific task, and we offered constructive criticism to each other. The programming skills of each member greatly enhanced throughout the project and it was easily noticeable by the whole team as we evaluated our progress. Upon the completing of the project we as a team are satisfied with our work, and we feel we have achieved all of our project goals.

## 6.3 Evaluation of Project

The team as a whole believe that all of the deliverables were necessary to completing the project. Each deliverable helped us further our progression to finishing the project. We think that Deliverable #1 and Deliverable #2 were simpler to complete than the other deliverables. In our opinion, I think further instruction and details were needed when creating the testing plans or the tables. We were not sure at first what information was needed to be displayed in the tables. Deliverable #3 was the most difficult to complete because it required a fully automated testing framework to work correctly. Learning the syntax of BASH was challenging and also comparing two variables. Once we figure out how to do this then it was easier for us to develop

a working script. Deliverable #4 was just expanding on Deliverable #3 by adding more test cases. Deliverable #5 was also not that difficult to complete because we already had the testing framework completed. In this deliverable, we just had to add 5 fault injections into the existing code of the mathlib.h library. This was simple to do because we just had to change the multiplication operator to a division operator for example.

In our opinion, we thought the most difficult part of the project was completing the Bash script and making sure that it ran how we wanted it to. It was also helpful to create the myList script that outputted the current directory to an HTML page because this was useful when we wanted to output the testing results table to an HTML page. As we continued to work on the project, we enjoyed learning how to use the Linux command line, Bash scripting, HTML/CSS, C++, and GitHub commands. We also think that learning how to use GitHub and its commands are useful not only in this class but when we graduate and start looking for jobs. Our GitHub profile can be used to demonstrate our past programming projects and what we have been working on recently. It is a good way of promoting yourself to potential employers and having a GitHub profile with projects is great on your resume.

Some suggestions for improving the programming projects for the future and to improve the delivery of CSCI 462 lessen our workload when writing blog entries so we can spend more time working on the project. In the beginning of the semester, we had to read several articles and write about them on blog. Some of the articles were very difficult to read being low quality scans. In addition to these articles we had to also read a chapter from the textbook. One suggestion would be to not require us to read both the articles and the chapter from the textbook for the next class. The length of the test should also be reduced, some of the questions that had to be answered we did not have enough time to fully articulate a satisfactory answer. Also, in our opinion, there should be a grading rubric for the final report, the presentation, and the poster.