College of Charleston

Software Engineering:
Term Project

Spatiotemporal Epidemiological Modeler (STEM) Automated Testing Framework

Team Fantastic Four

Katherine Sweeney
Ashanti Long
Clare Clever

Supervised by

Dr. Jim Bowring

Fall 2020

Table of Contents

Return to Table of Contents

Introduction

This project was assigned and completed in the College of Charleston's Software Engineering class; CSCI 362. It was a semester-long team project, where progress was meticulously documented, presented, and subsequently evaluated via the professor, Dr. Jim Bowring.

The goal of this project was to select an H/FOSS project and build it. Throughout the process, we were to document our progress and evaluate our experiences building the project and executing the specified tasks. Such tasks necessitated finding testable methods within the project that had no dependencies such that they could be compiled and ran on the command line. We were to find five testable methods, and make five test cases for each method. Following that we needed to implement these test cases via a script which would pull and run the test case, and then compile the results into a testing report.

We made several mistakes attempting to meet the specifications of the project. These mistakes remain in the report, and will be pointed out and further explained. Comments made after a portion of the project was completed are differentiated by a set of asterisks.

This report details our progress from the first deliverable to the last with in-depth explanations, examples, and reflections.

Return to Table of Contents

Chapter One

First Evaluation of Spatiotemporal Epidemiological Modeler (STEM)

Why We Chose STEM

The Spatiotemporal Epidemiological Modeler, or (STEM) for short, provides, as defined on the STEM FAQ page, a graph based spatiotemporal simulation engine. This allows researchers to produce models on infectious diseases. Such models can provide a plethora of information about diseases, which can assist scientists in learning more about them. Such pieces of information could include infection rates, potentially allowing for the ability to mitigate or prevent infection. It additionally, as stated on the About STEM page, allows for models to be made consisting of climate data.

Furthermore, there is extensive documentation surrounding all aspects of the application, from building it, to constructing models. This solidified our choice to select this project.

STEM FAQ
About STEM
STEM Main Page ~ Eclipsepedia (Hyperlinked Resources)

Thoughts on Building the Application

Because there is an abundance of documentation, it was fairly easy to set the project up without issue. Following the documentation provided support throughout every step of the process. There was a small hiccup with a required plug-in development tool not being explicitly referenced in the documentation, but we were able to resolve the issue. While there were dependencies, it was not complex to address them. Regardless, for the scope of our project, we are more interested in specific methods in the application rather than the application as a whole.

STEM Installation Guide

Final Thoughts

Ultimately, we are incredibly excited to jump into what is uncharted territory for all of us. While we have our work cut out for us, we look forward to making progress as we resolve issues along the way.

Return to Table of Contents

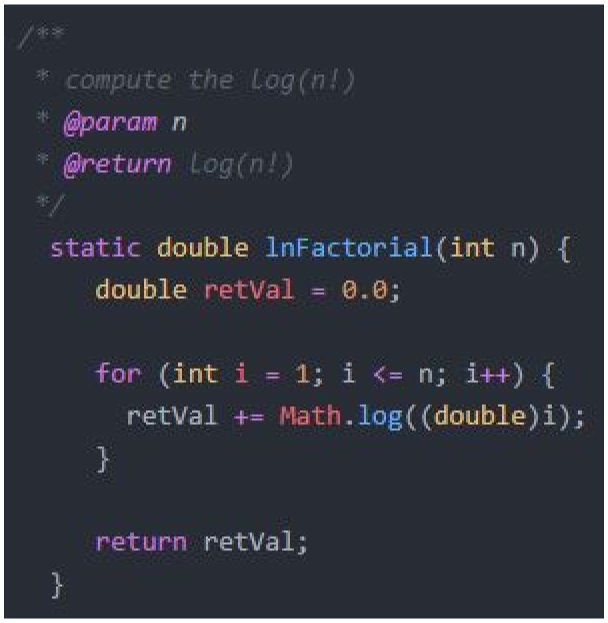
Chapter Two

Test Plan Foundations

Preface to Method Selection

The STEM engine manipulates inputted data to produce spatio-temporal models, therefore it must contain a plethora of mathematical functions needed to compute related statistical values for modeling. Thus, most of the methods that we are interested in testing are going to be math related methods.

The Selected Initial Method

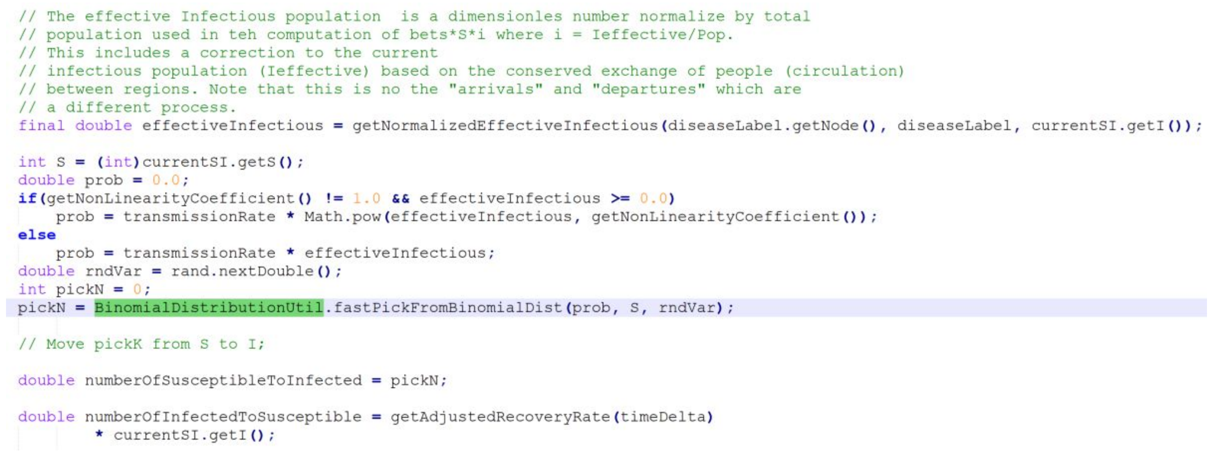


```java
/**
 * compute the log(n!)
 * @param n
 * @return log(n!)
 */
 static double lnFactorial(int n) {
    double retVal = 0.0;

    for (int i = 1; i <= n; i++) {
      retVal += Math.log((double)i);
    }

    return retVal;
 }
```

This genre of method is relatively easy to test because we can compute these values outside of the method, and produce an oracle based on that information. Because of the nature of the method, it is easy to recognize what sort of input *should* be invalid, and which input is reasonable.

Method Context

The lnFactorial() method is used in the calculation of binomial distributions within the StochasticPoissonSIDiseaseModelImpl class, for modeling purposes.



```
// The effective Infectious population  is a dimensionles number normalize by total
// population used in teh computation of bets*S*i where i = Ieffective/Pop.
// This includes a correction to the current
// infectious population (Ieffective) based on the conserved exchange of people (circulation)
// between regions. Note that this is no the "arrivals" and "departures" which are
// a different process.
final double effectiveInfectious = getNormalizedEffectiveInfectious(diseaseLabel.getNode(), diseaseLabel, currentSI.getI());

int S = (int)currentSI.getS();
double prob = 0.0;
if(getNonLinearityCoefficient() != 1.0 && effectiveInfectious >= 0.0)
    prob = transmissionRate * Math.pow(effectiveInfectious, getNonLinearityCoefficient());
else
    prob = transmissionRate * effectiveInfectious;
double rndVar = rand.nextDouble();
int pickN = 0;
pickN = BinomialDistributionUtil.fastPickFromBinomialDist(prob, S, rndVar);

// Move pickK from S to I;

double numberOfSusceptibleToInfected = pickN;

double numberOfInfectedToSusceptible = getAdjustedRecoveryRate(timeDelta)
        * currentSI.getI();
```

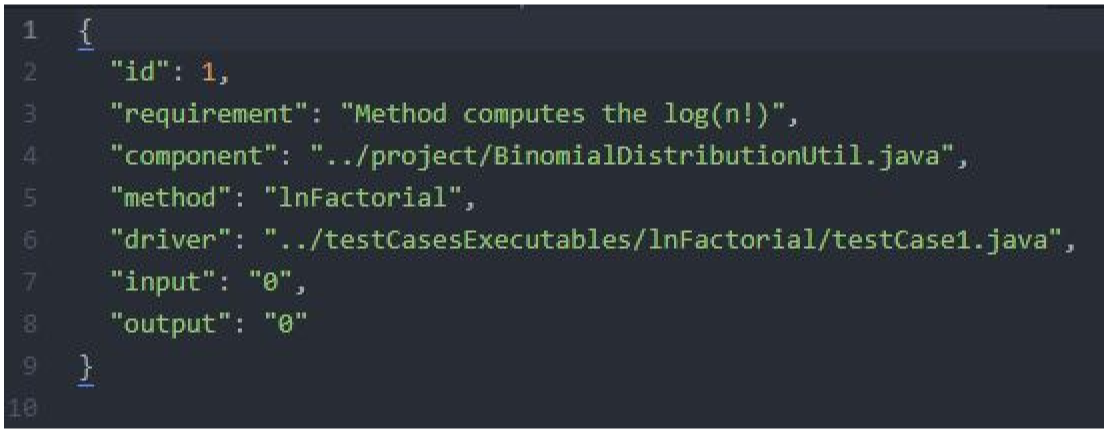
System Architecture Relating to Method Behavior

The system architecture of the machine that might run this method will have no bearing on how the method runs. This is because the method is written in a high-level language, and by definition, this means that system architecture has no effect on the behavior of the method, so long as the system has Java installed on it.

Test Case Considerations For Initial Method

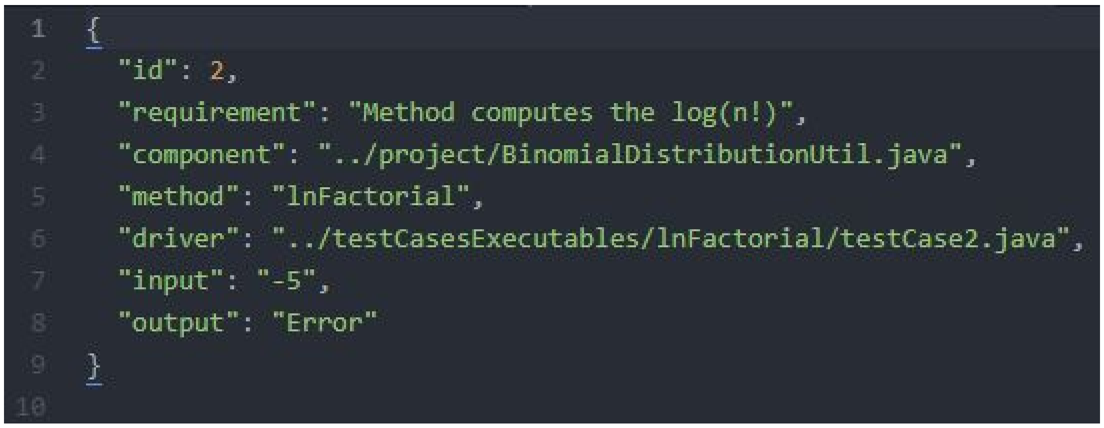
Firstly, we know that this method should not take anything other than an integer, as is specified by its parameter. Second, this method should not compute the value of a negative integer, due to the nature of factorials. Third, the method should not compute the value of an out of bounds integer; thus -2,147,483,648 <= n <= 2,147,483,647. Fourth, the method should not produce valid output if there is white space between an integer, for example, passing in 1 2, should not return valid output. Lastly, the method should produce a valid computation of ln(n!), given valid input; a positive integer, in the range of $2^{32} - 1$.

Test Case 1:



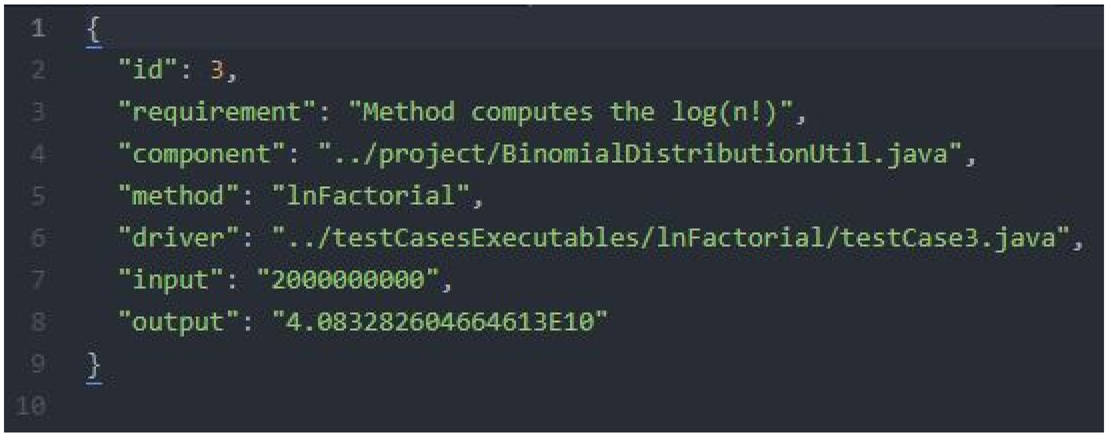
```
{
  "id": 1,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase1.java",
  "input": "0",
  "output": "0"
}
```

Test Case 2:



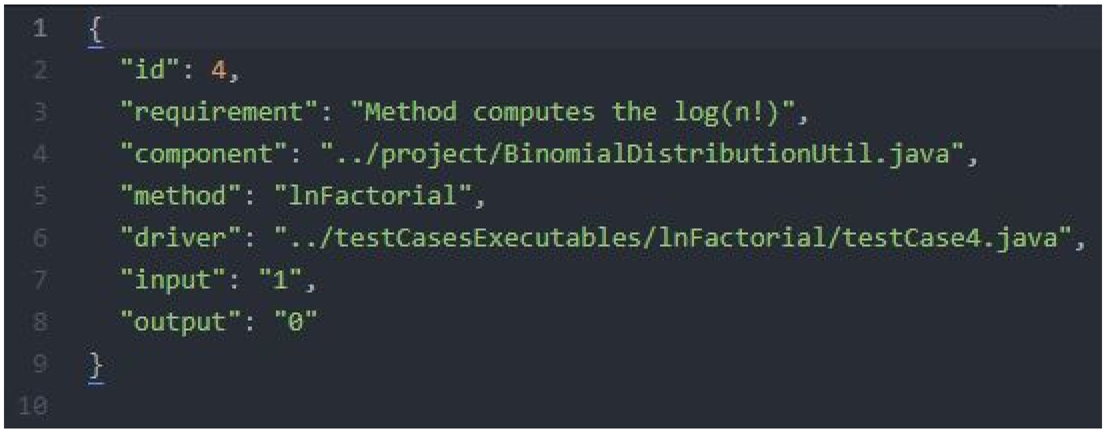
```
{
  "id": 2,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase2.java",
  "input": "-5",
  "output": "Error"
}
```

Test Case 3:



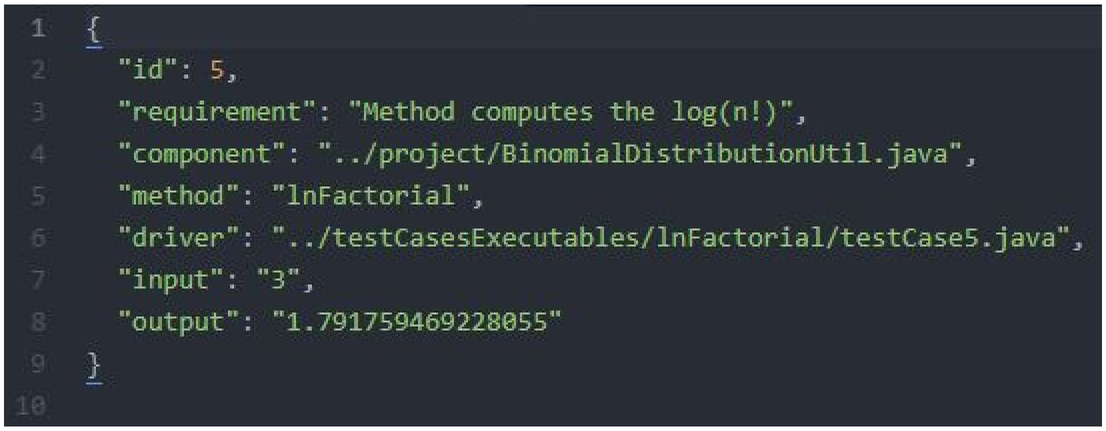
```
{
  "id": 3,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase3.java",
  "input": "2000000000",
  "output": "4.083282604664613E10"
}
```

Test Case 4:



```
{
  "id": 4,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase4.java",
  "input": "1",
  "output": "0"
}
```

Test Case 5:



```
{
  "id": 5,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase5.java",
  "input": "3",
  "output": "1.791759469228055"
}
```

Return to Table of Contents

Chapter Three

Developing the Automated Testing Framework

Architectural Description

An extensive architectural description of the automated testing framework can be found on our repo via the link above.

*This is based off of the first incorrect version of the testing framework.

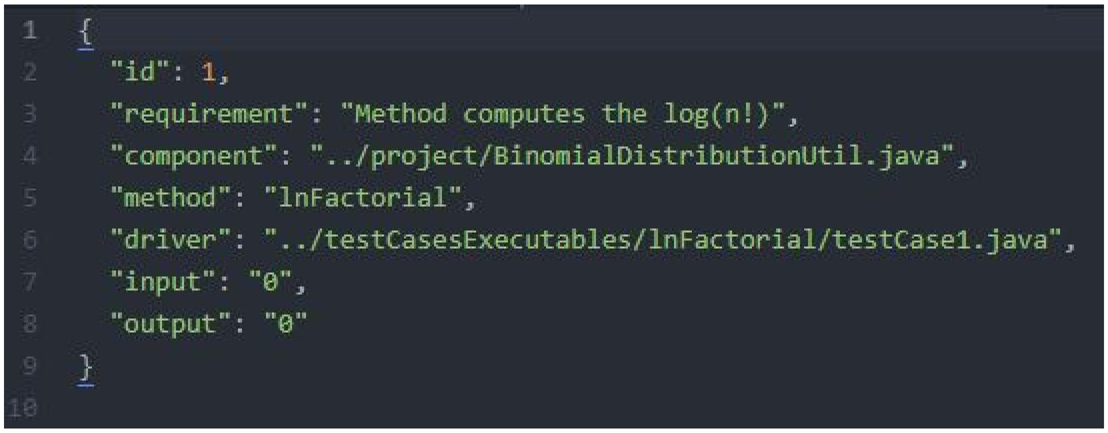
We updated our architectural description after we had properly implemented the script. This version can be found here.*

How-To (Run the Framework)

1. Install Oracle VM VirtualBox.
   https://www.virtualbox.org/wiki/Downloads
2. Download a Linux ISO file.
3. Import the ISO file into Virtual Box.
4. Start the Linux VM.
5. Install the operating system.
6. Open a browser and navigate to the Fantastic 4 repo page.
7. Save the repository as a zip file.
8. Unzip the repo to a location on the computer.
9. Open a terminal.
10. Type the following commands…
    a. sudo apt-get update
    b. sudo apt-get install python3.9
    c. sudo apt-get install openjdk-8-jdk
11. Navigate to the folder that houses the downloaded repo.
12. Type the following commands…
    a. cd TestAutomation/scripts
    b. python3.9 runAllTests.py

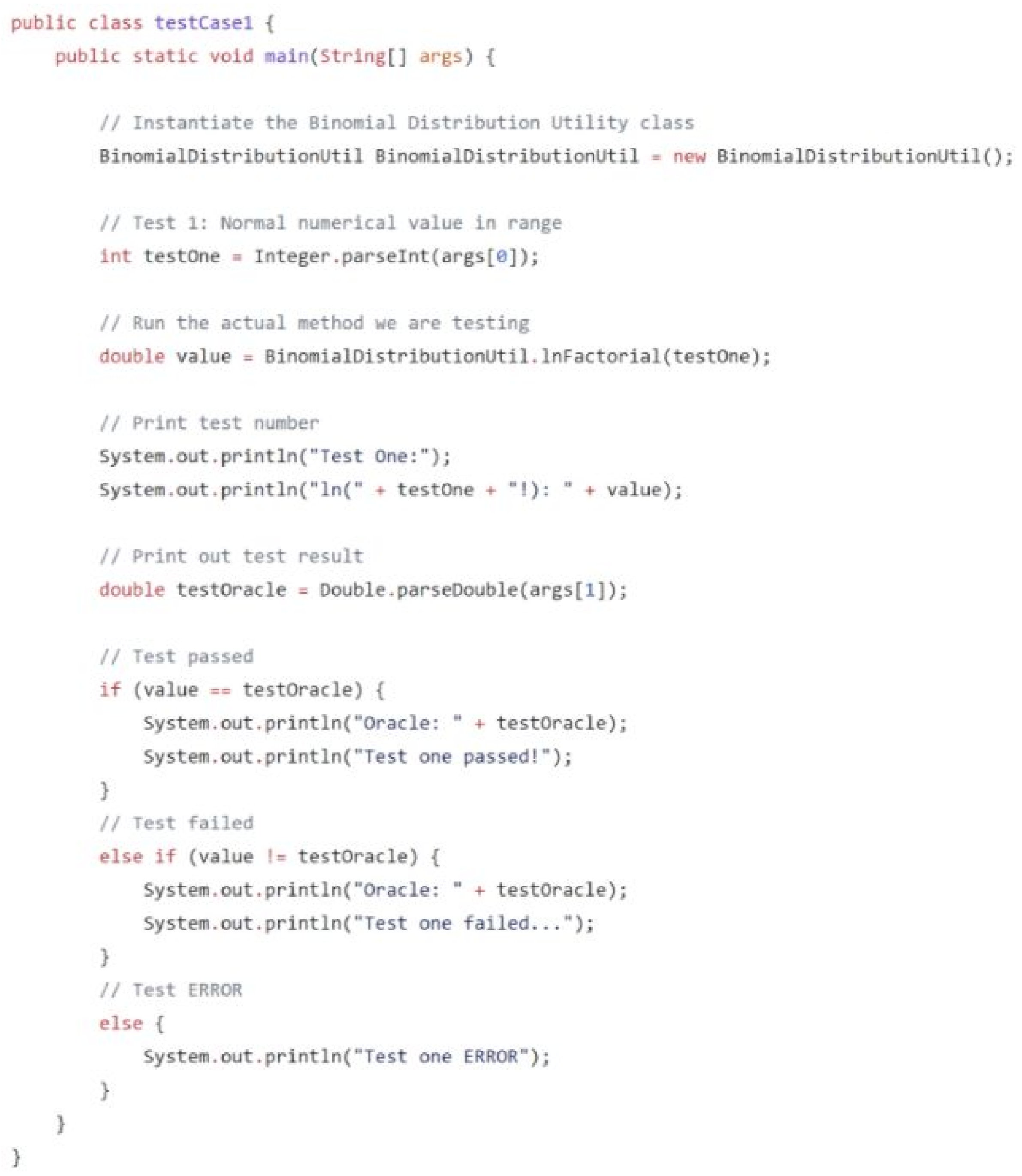
Test Cases (lnFactorial)

In the previous chapter we specified five test cases for the method lnFactorial, which is used in the calculation of binomial distributions within the StochasticPoissonSIDiseaseModelImpl class.



```json
{
  "id": 1,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase1.java",
  "input": "0",
  "output": "0"
}
```

Because we had originally implemented our file structure incorrectly, our original JSON test case files had an incorrect path to the driver, which can be seen above. The proper, and final JSON files have the path for the driver without the unnecessary division of test case by method type. So for all test cases, the proper driver path is testCasesExecutables/testcase_.java

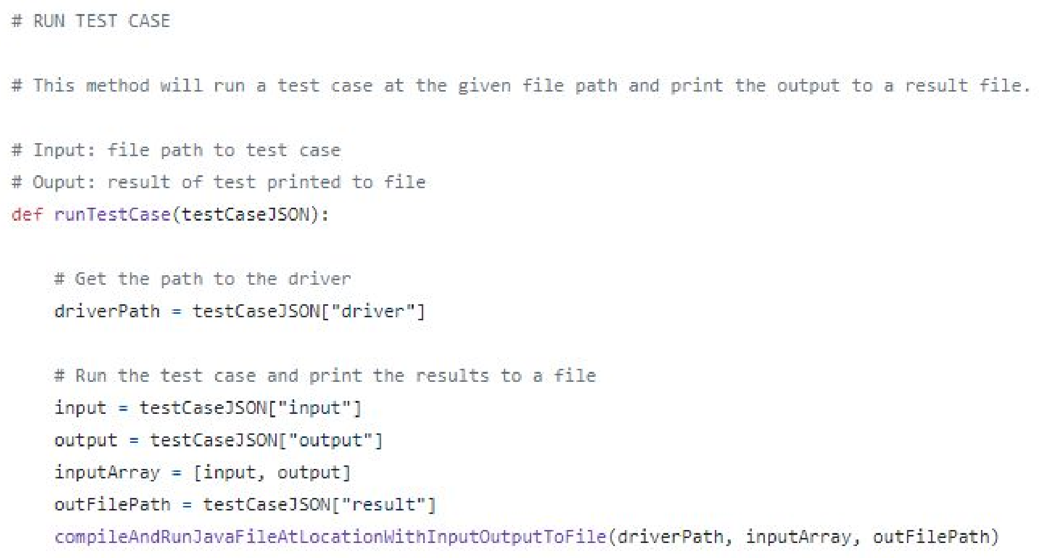

```java
public class testCase1 {
    public static void main(String[] args) {

        // Instantiate the Binomial Distribution Utility class
        BinomialDistributionUtil BinomialDistributionUtil = new BinomialDistributionUtil();

        // Test 1: Normal numerical value in range
        int testOne = Integer.parseInt(args[0]);

        // Run the actual method we are testing
        double value = BinomialDistributionUtil.lnFactorial(testOne);

        // Print test number
        System.out.println("Test One:");
        System.out.println("ln(" + testOne + "!): " + value);

        // Print out test result
        double testOracle = Double.parseDouble(args[1]);

        // Test passed
        if (value == testOracle) {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Test one passed!");
        }
        // Test failed
        else if (value != testOracle) {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Test one failed...");
        }
        // Test ERROR
        else {
            System.out.println("Test one ERROR");
        }
    }
}
```

It is important to note that we initially executed this incorrectly, as will be discussed in a later chapter. As can be seen in the above screenshot, we initially built the driver for each test case, which was a misunderstanding, and lack of scrutinization of the specifications. We fixed this later. In addition, the file structure of our project was implemented incorrectly, as the test cases were in individual folders, as discussed above.

Important methods from the runAllTests Script (Incorrect Approach)

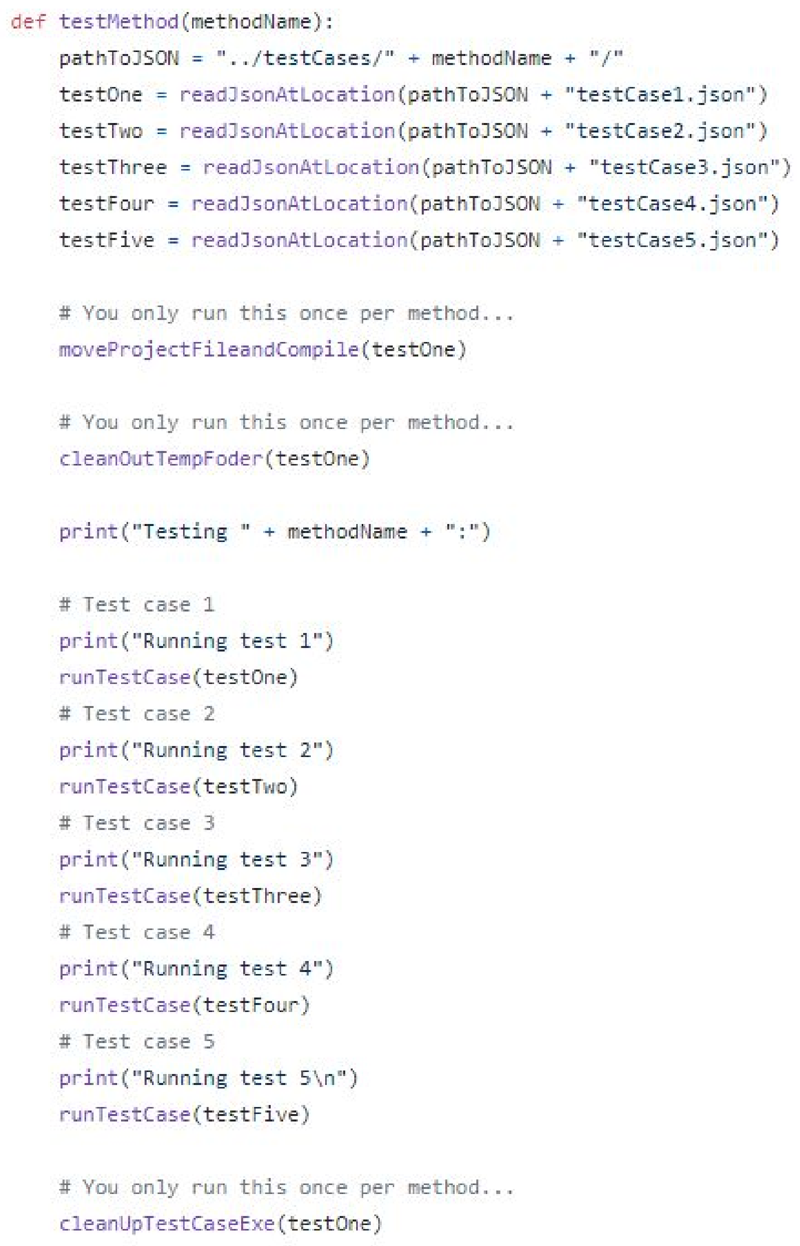
runAllTests.py (runTestCase)

Takes a test case, runs it, and stores the results



```
# RUN TEST CASE

# This method will run a test case at the given file path and print the output to a result file.

# Input: file path to test case
# Ouput: result of test printed to file
def runTestCase(testCaseJSON):

    # Get the path to the driver
    driverPath = testCaseJSON["driver"]

    # Run the test case and print the results to a file
    input = testCaseJSON["input"]
    output = testCaseJSON["output"]
    inputArray = [input, output]
    outFilePath = testCaseJSON["result"]
    compileAndRunJavaFileAtLocationWithInputOutputToFile(driverPath, inputArray, outFilePath)
```

runAllTests.py (testMethod)

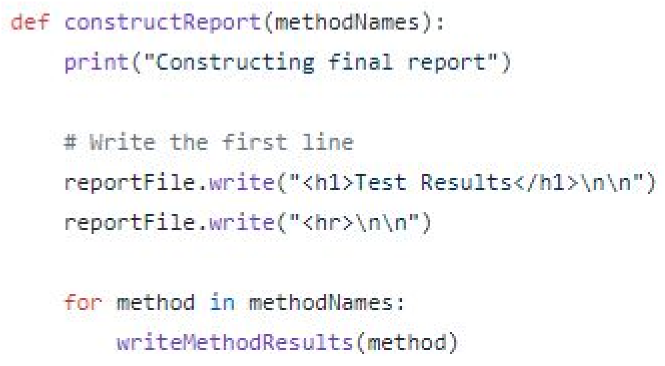


```
def testMethod(methodName):
    pathToJSON = "../testCases/" + methodName + "/"
    testOne = readJsonAtLocation(pathToJSON + "testCase1.json")
    testTwo = readJsonAtLocation(pathToJSON + "testCase2.json")
    testThree = readJsonAtLocation(pathToJSON + "testCase3.json")
    testFour = readJsonAtLocation(pathToJSON + "testCase4.json")
    testFive = readJsonAtLocation(pathToJSON + "testCase5.json")

    # You only run this once per method...
    moveProjectFileandCompile(testOne)

    # You only run this once per method...
    cleanOutTempFoder(testOne)

    print("Testing " + methodName + ":")

    # Test case 1
    print("Running test 1")
    runTestCase(testOne)
    # Test case 2
    print("Running test 2")
    runTestCase(testTwo)
    # Test case 3
    print("Running test 3")
    runTestCase(testThree)
    # Test case 4
    print("Running test 4")
    runTestCase(testFour)
    # Test case 5
    print("Running test 5\n")
    runTestCase(testFive)

    # You only run this once per method...
    cleanUpTestCaseExe(testOne)
```

The initial implementations of these script methods are not correct, as they are executing via method names, instead of test cases. We fixed this later on.

Return to Table of Contents

runAllTests.py (constructReport);



```python
def constructReport(methodNames):
    print("Constructing final report")

    # Write the first line
    reportFile.write("<h1>Test Results</h1>\n\n")
    reportFile.write("<hr>\n\n")

    for method in methodNames:
        writeMethodResults(method)
```

This method constructs a report detailing the results of the test cases.

Because of our initial misfire with the script, our way of outputting the results had to be changed at a later time, because the script was executing by method instead of by test case.

Chapter Four

Completing the Testing Framework

Changes Implemented From Previous Chapter

There were three issues with our partial testing framework:

- Unnecessary number of drivers
- The script knowing the test case names
- Formatting the test result output

The first issue was resolved incredibly easy, by simply making a generalized driver for each method. Thus, we have five methods, and their respective drivers.

The script knowing the test case names was resolved by relying on the static nature of the file structure associated with our project. While the script should not know anything about the actual test cases, it can find the location of the test cases, and iterate through the files present in the testCases directory, which are the test cases, and temporarily store them in an array methodNames.

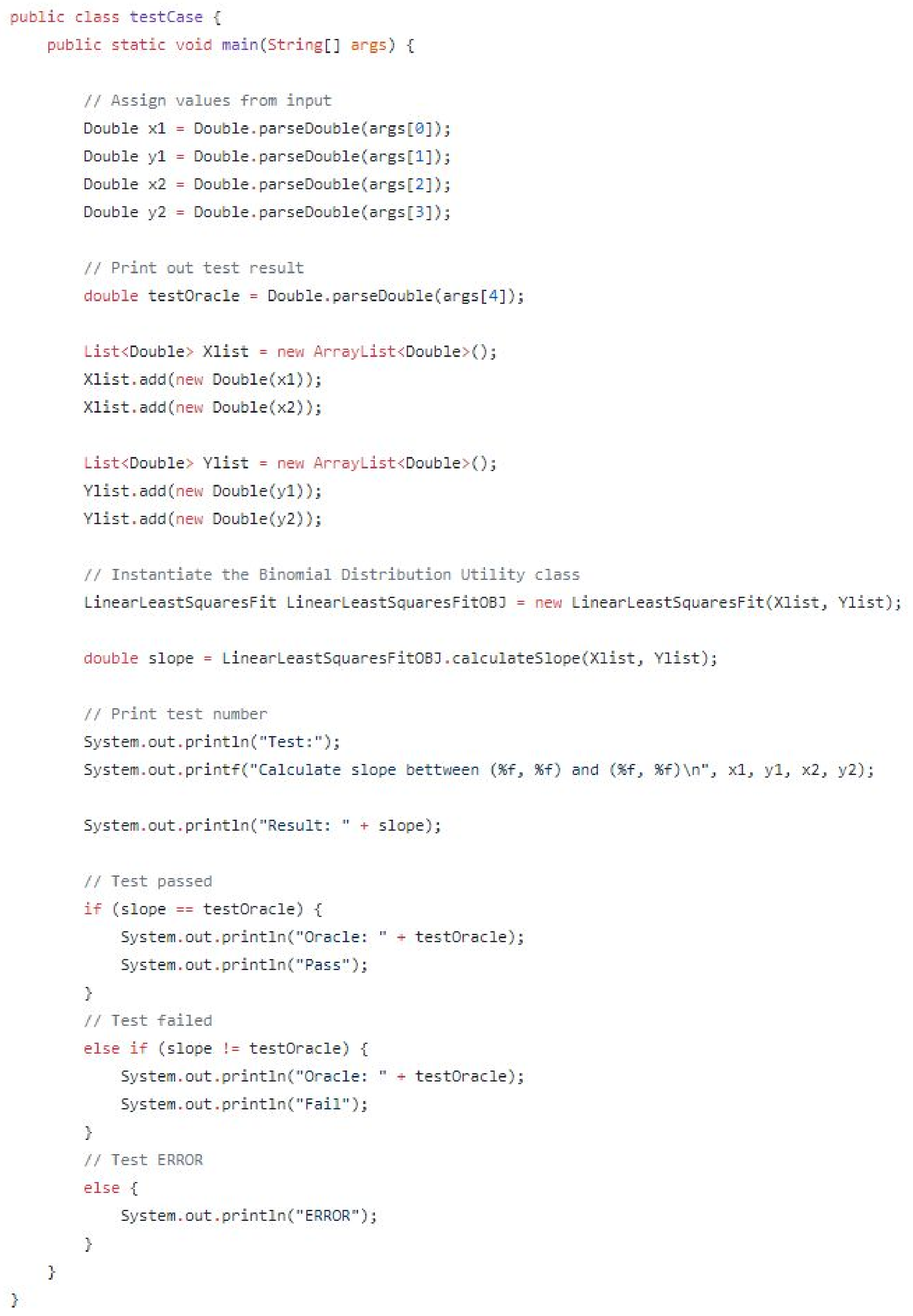
returnJsonFiles takes a methodName, and gets the path to the testCase for that specific method name, based on the fact that all the testCases are located in testCases/*method-name*/.

It then runs ls on the ./testCases/ directory and stores the results in a temporary text file called temp.txt. It then will iterate through the temporary file and read the Json files by line. Finally it will remove the temporary file and return an array of the jsonFiles.
Thus, all other functions included in the script rely on reading in methodName for testing, which is now retrieved by way of traversing the file structure and looking for specific files rather than those files being hardcoded, which defeats the purpose of the automation.

At the end of the chapter, it is explained that this is still incorrect. Keep this in mind while finishing up this chapter, as there are some glaring errors in parts of the script that were eventually resolved. For example, the script is not supposed to execute via the method names as is shown in the screenshots below, and referenced above.

Return to Table of Contents

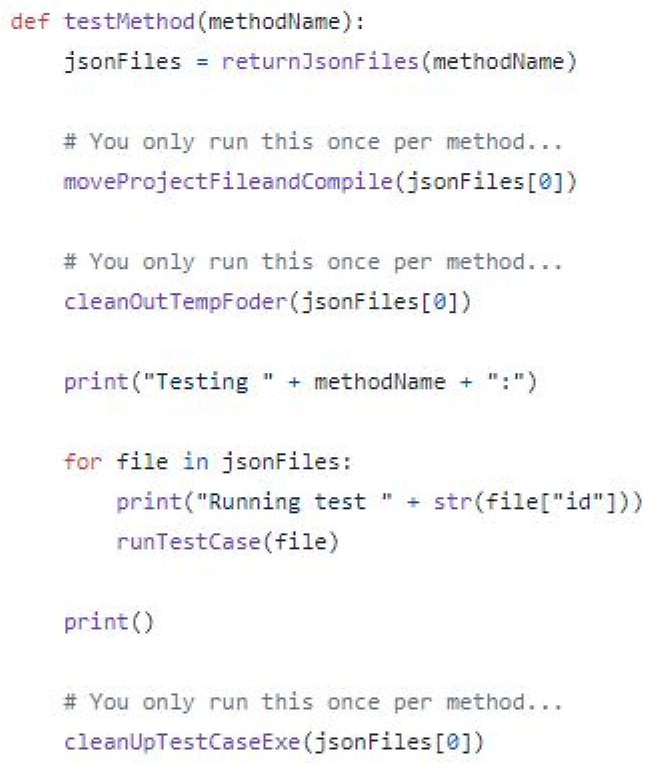
Example Driver for Method calculateSlope (Incorrect Approach)



```java
public class testCase {
    public static void main(String[] args) {

        // Assign values from input
        Double x1 = Double.parseDouble(args[0]);
        Double y1 = Double.parseDouble(args[1]);
        Double x2 = Double.parseDouble(args[2]);
        Double y2 = Double.parseDouble(args[3]);

        // Print out test result
        double testOracle = Double.parseDouble(args[4]);

        List<Double> Xlist = new ArrayList<Double>();
        Xlist.add(new Double(x1));
        Xlist.add(new Double(x2));

        List<Double> Ylist = new ArrayList<Double>();
        Ylist.add(new Double(y1));
        Ylist.add(new Double(y2));

        // Instantiate the Binomial Distribution Utility class
        LinearLeastSquaresFit LinearLeastSquaresFitOBJ = new LinearLeastSquaresFit(Xlist, Ylist);

        double slope = LinearLeastSquaresFitOBJ.calculateSlope(Xlist, Ylist);

        // Print test number
        System.out.println("Test:");
        System.out.printf("Calculate slope bettween (%f, %f) and (%f, %f)\n", x1, y1, x2, y2);

        System.out.println("Result: " + slope);

        // Test passed
        if (slope == testOracle) {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Pass");
        }
        // Test failed
        else if (slope != testOracle) {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Fail");
        }
        // Test ERROR
        else {
            System.out.println("ERROR");
        }
    }
}
```

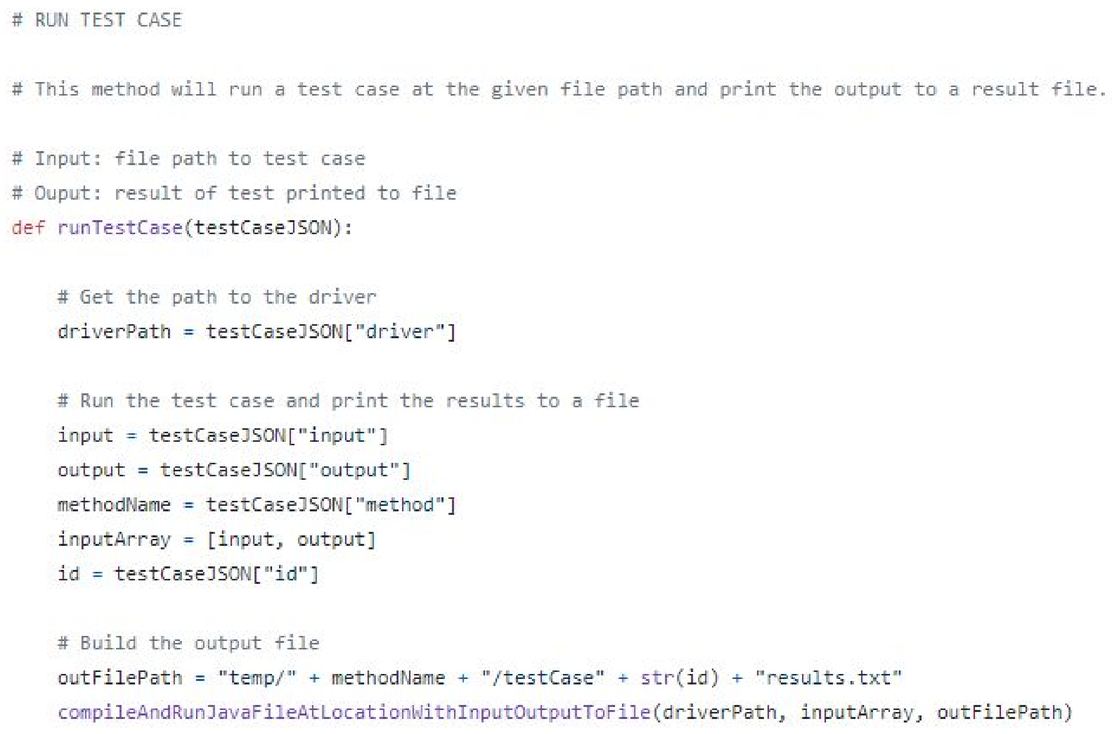
Return to Table of Contents

Test Method



```
def testMethod(methodName):
    jsonFiles = returnJsonFiles(methodName)

    # You only run this once per method...
    moveProjectFileandCompile(jsonFiles[0])

    # You only run this once per method...
    cleanOutTempFoder(jsonFiles[0])

    print("Testing " + methodName + ":")

    for file in jsonFiles:
        print("Running test " + str(file["id"]))
        runTestCase(file)

    print()

    # You only run this once per method...
    cleanUpTestCaseExe(jsonFiles[0])
```

Executing a Test Case

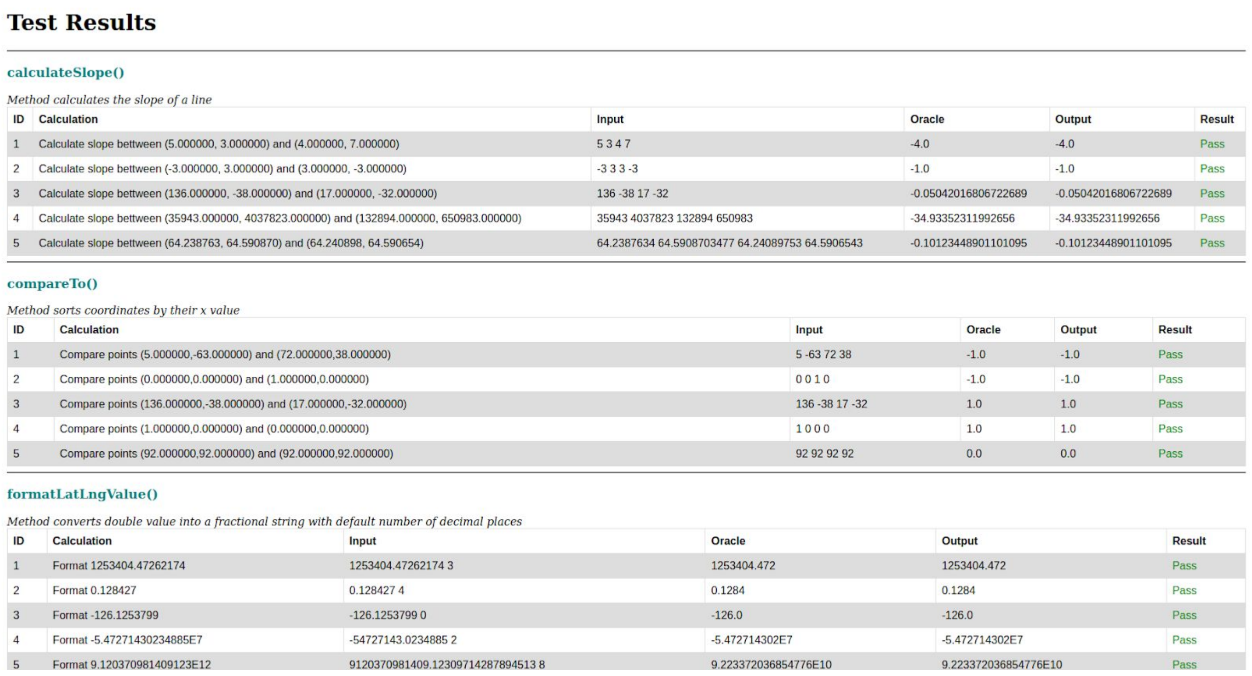


```
# RUN TEST CASE

# This method will run a test case at the given file path and print the output to a result file.

# Input: file path to test case
# Ouput: result of test printed to file
def runTestCase(testCaseJSON):

    # Get the path to the driver
    driverPath = testCaseJSON["driver"]

    # Run the test case and print the results to a file
    input = testCaseJSON["input"]
    output = testCaseJSON["output"]
    methodName = testCaseJSON["method"]
    inputArray = [input, output]
    id = testCaseJSON["id"]

    # Build the output file
    outFilePath = "temp/" + methodName + "/testCase" + str(id) + "results.txt"
    compileAndRunJavaFileAtLocationWithInputOutputToFile(driverPath, inputArray, outFilePath)
```

Formatting Test Results

The last issue was the quality of the test result output, which was vastly improved between last deliverable and now. Addition of color coding specific to test cases pass/fail, cleaning up the tables, and generally making it more attractive.

HTML Output



Test Results

calculateSlope()

Method calculates the slope of a line

ID	Calculation	Input	Oracle	Output	Result
1	Calculate slope bettween (5.000000, 3.000000) and (4.000000, 7.000000)	5 3 4 7	-4.0	-4.0	Pass
2	Calculate slope bettween (-3.000000, 3.000000) and (3.000000, -3.000000)	-3 3 3 -3	-1.0	-1.0	Pass
3	Calculate slope bettween (136.000000, -38.000000) and (17.000000, -32.000000)	136 -38 17 -32	-0.05042016806722689	-0.05042016806722689	Pass
4	Calculate slope bettween (35943.000000, 4037823.000000) and (132894.000000, 650983.000000)	35943 4037823 132894 650983	-34.93352311992656	-34.93352311992656	Pass
5	Calculate slope bettween (64.238763, 64.590870) and (64.240898, 64.590654)	64.2387634 64.5908703477 64.24089753 64.5906543	-0.10123448901101095	-0.10123448901101095	Pass

compareTo()

Method sorts coordinates by their x value

ID	Calculation	Input	Oracle	Output	Result
1	Compare points (5.000000,-63.000000) and (72.000000,38.000000)	5 -63 72 38	-1.0	-1.0	Pass
2	Compare points (0.000000,0.000000) and (1.000000,0.000000)	0 0 1 0	-1.0	-1.0	Pass
3	Compare points (136.000000,-38.000000) and (17.000000,-32.000000)	136 -38 17 -32	1.0	1.0	Pass
4	Compare points (1.000000,0.000000) and (0.000000,0.000000)	1 0 0 0	1.0	1.0	Pass
5	Compare points (92.000000,92.000000) and (92.000000,92.000000)	92 92 92 92	0.0	0.0	Pass

formatLatLngValue()

Method converts double value into a fractional string with default number of decimal places

ID	Calculation	Input	Oracle	Output	Result
1	Format 1253404.47262174	1253404.47262174 3	1253404.472	1253404.472	Pass
2	Format 0.128427	0.128427 4	0.1284	0.1284	Pass
3	Format -126.1253799	-126.1253799 0	-126.0	-126.0	Pass
4	Format -5.47271430234885E7	-54727143.0234885 2	-5.472714302E7	-5.472714302E7	Pass
5	Format 9.120370981409123E12	9120370981409.12309714287894513 8	9.223372036854776E10	9.223372036854776E10	Pass

Return to Table of Contents

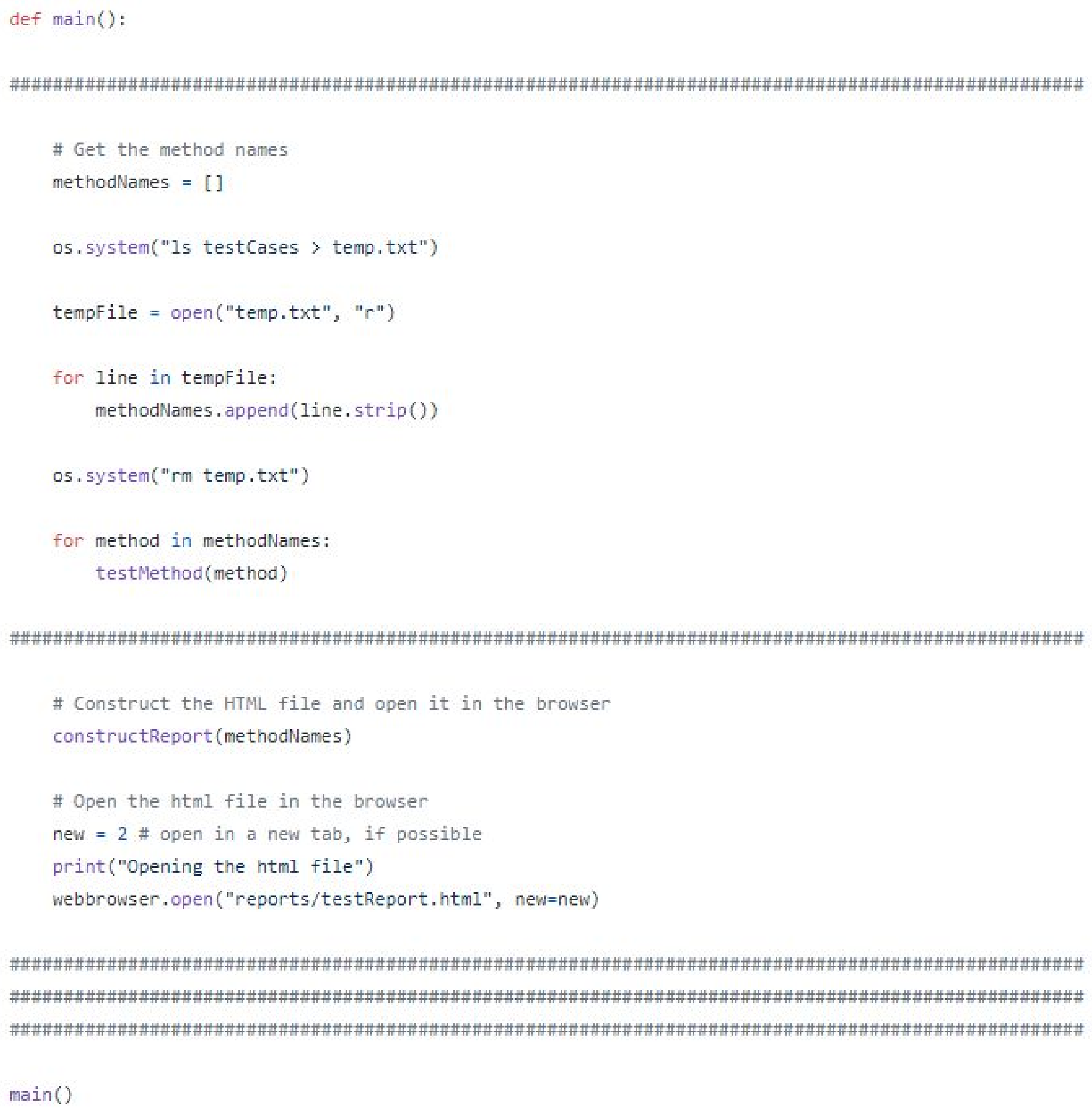
Extending the Testing Framework to the Rest of the Methods

After resolving the issues above, extending the framework to include the four additional methods was simple, because each method is treated exactly the same with its respective driver being called in the script.

The majority of the time spent between chapter three, and the current chapter was working on generalizing the code, because it was too specific (having more than one driver, or the script having the method names hard coded), as well as optimizing the output file in terms of appearance and data presentation.

Return to Table of Contents

Second (Incorrect) Script Attempt



```
def main():

####################################################################################################

    # Get the method names
    methodNames = []

    os.system("ls testCases > temp.txt")

    tempFile = open("temp.txt", "r")

    for line in tempFile:
        methodNames.append(line.strip())

    os.system("rm temp.txt")

    for method in methodNames:
        testMethod(method)

####################################################################################################

    # Construct the HTML file and open it in the browser
    constructReport(methodNames)

    # Open the html file in the browser
    new = 2 # open in a new tab, if possible
    print("Opening the html file")
    webbrowser.open("reports/testReport.html", new=new)

####################################################################################################
####################################################################################################
####################################################################################################

main()
```

Executing all 25 test cases for the 5 methods is then done by iterating through the methodNames that have been pulled from the .json files, and executing each respective driver (5 total), gathering the data from these test cases, and then formatting and outputting the results via testReport.html.

* We missed the mark again. We had the right idea, but our execution was still off, due to the nature of how our files existed in the project (unnecessary folders, misunderstandings with file names). The script was still implemented incorrectly. It was still running via method names, instead of test cases. This in turn affected the implementation of building the report, as well as other minor things within the script. How we addressed this is discussed in the next chapter. *

Return to Table of Contents

Chapter Five

Fault Injection

Final Changes

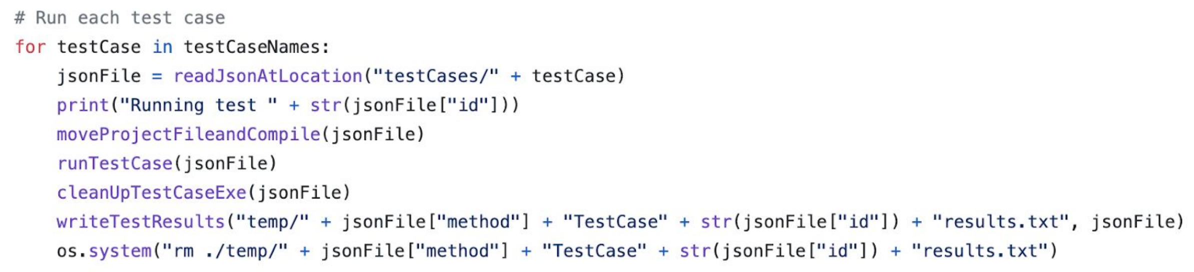
In the last chapter we had attempted to resolve the issue of the script having information on the test cases. We had approached the problem from the perspective of the methods, instead of the test cases, so we were preloading the method names in and building the script around the methods. Luckily, the third time's the charm and we were able to take the fundamentals of how our script ran and use that to implement the testing framework correctly. We just needed to practice our reading comprehension; it is a learning experience after all.

Final Script Update: Test Case Loading

```
testCaseNames = []

# Call the ls command on the testCases directory
os.system("ls testCases > temp.txt")

# Open the temp file
tempFile = open("temp.txt", "r")
```

Unlike the last version of our script, the new and improved, and most importantly, correct, version, gets the test cases from their directory. The only potential problem with this approach of preloading the test cases, is the potentiality of having a *lot* of test cases. This could slow the script down to preload all of those test cases and then execute each one individually. For the 25 test cases, we felt that this approach was acceptable.

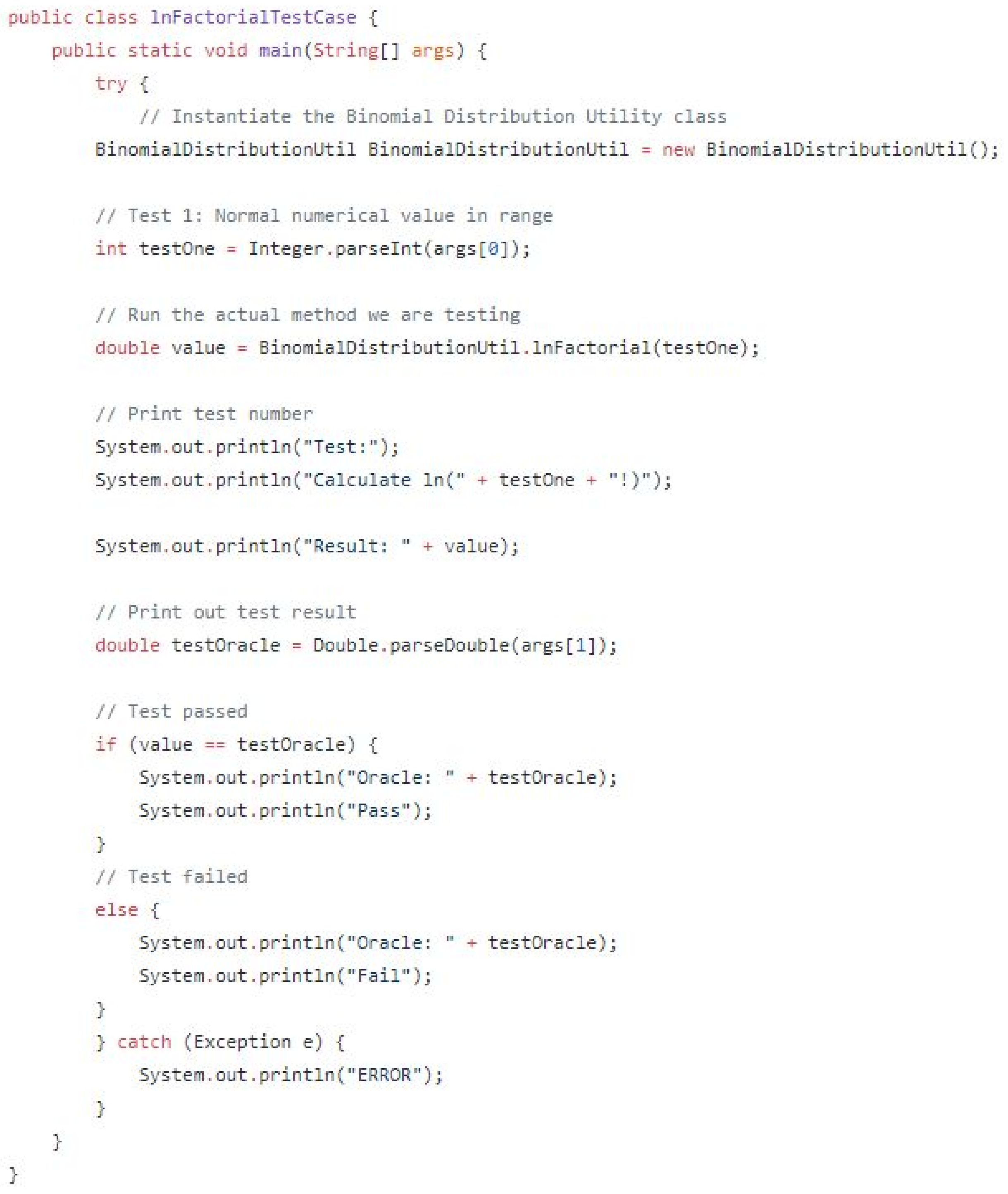
Final Script Update: Test Case Execution



```
# Run each test case
for testCase in testCaseNames:
    jsonFile = readJsonAtLocation("testCases/" + testCase)
    print("Running test " + str(jsonFile["id"]))
    moveProjectFileandCompile(jsonFile)
    runTestCase(jsonFile)
    cleanUpTestCaseExe(jsonFile)
    writeTestResults("temp/" + jsonFile["method"] + "TestCase" + str(jsonFile["id"]) + "results.txt", jsonFile)
    os.system("rm ./temp/" + jsonFile["method"] + "TestCase" + str(jsonFile["id"]) + "results.txt")
```

Because our script was executing in terms of the method names, the output was being grouped by method name, due to how the test cases were organized in their folder. The problem with this approach is, if the order of the test cases changed the output would not be grouped correctly. Thus, when we fixed the script and it met the specifications, we also had to reimplement how the output would be organized. So, we decided to order it in ascending order by test ID. For now, the test cases for each method are grouped together, just because that is how the test cases exist in their directory, but if test cases were added it would remain sorted by test case ID but the method grouping might not stay as orderly. A look at the updated testing report can be seen later in the chapter.

Final Driver Example



```java
public class lnFactorialTestCase {
    public static void main(String[] args) {
        try {
            // Instantiate the Binomial Distribution Utility class
        BinomialDistributionUtil BinomialDistributionUtil = new BinomialDistributionUtil();

        // Test 1: Normal numerical value in range
        int testOne = Integer.parseInt(args[0]);

        // Run the actual method we are testing
        double value = BinomialDistributionUtil.lnFactorial(testOne);

        // Print test number
        System.out.println("Test:");
        System.out.println("Calculate ln(" + testOne + "!)");

        System.out.println("Result: " + value);

        // Print out test result
        double testOracle = Double.parseDouble(args[1]);

        // Test passed
        if (value == testOracle) {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Pass");
        }
        // Test failed
        else {
            System.out.println("Oracle: " + testOracle);
            System.out.println("Fail");
        }
        } catch (Exception e) {
            System.out.println("ERROR");
        }
    }
}
```

Return to Table of Contents

Here we have our final implementation of a driver for one of the methods. It grabs a test case with this specified driver, and executes it. There is a try-catch to ensure that an error does not completely break the script.

Fault Injection Preface

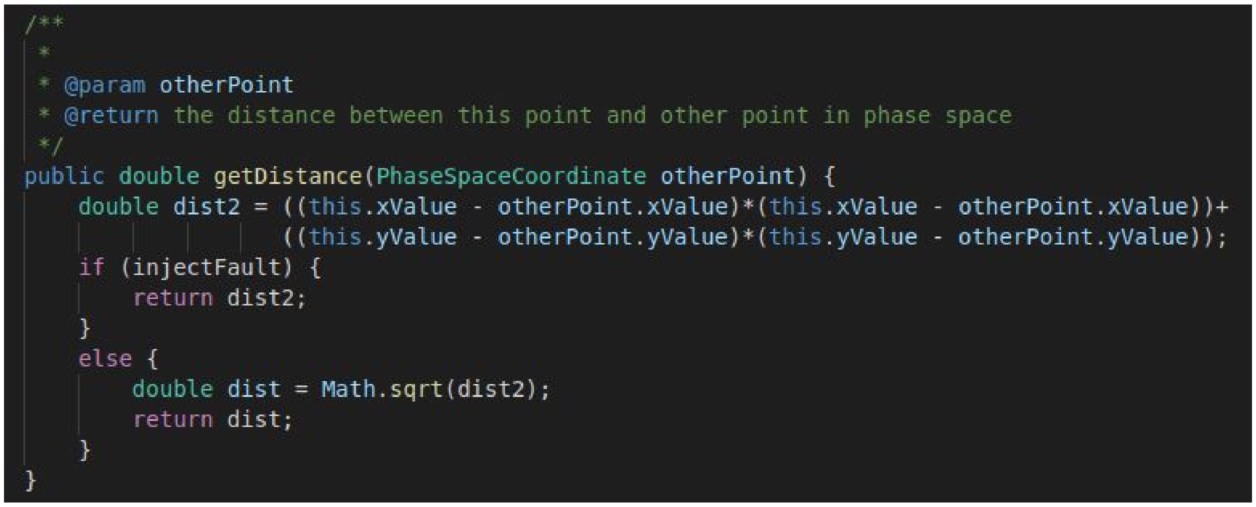
This portion of the project was injecting faults into the methods that were being tested, and attempting to make changes that would not cause all test cases to fail.

When we were first approaching this problem, we had to consider what types of faults would be realistic. eg., a person could make a mistake in computing an equation, which might not break the code, and perhaps it might even work for some test cases, but not all of them. Thus, we scrutinized the methods to come up with realistic mistakes/faults that someone could make during implementation. We used a boolean variable to implement the faults, so that if it is set to true, then the faulted code will execute, otherwise, the original code will execute.

Return to Table of Contents

Fault Injection Explanations

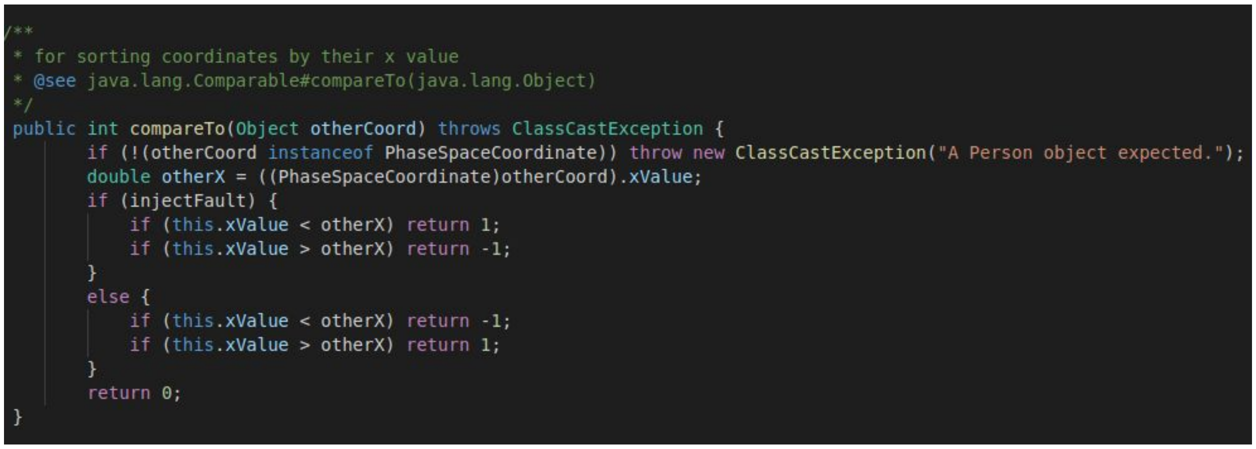
Method: getDistance



```
/**
 *
 * @param otherPoint
 * @return the distance between this point and other point in phase space
 */
public double getDistance(PhaseSpaceCoordinate otherPoint) {
    double dist2 = ((this.xValue - otherPoint.xValue)*(this.xValue - otherPoint.xValue))+
                   ((this.yValue - otherPoint.yValue)*(this.yValue - otherPoint.yValue));
    if (injectFault) {
        return dist2;
    }
    else {
        double dist = Math.sqrt(dist2);
        return dist;
    }
}
```

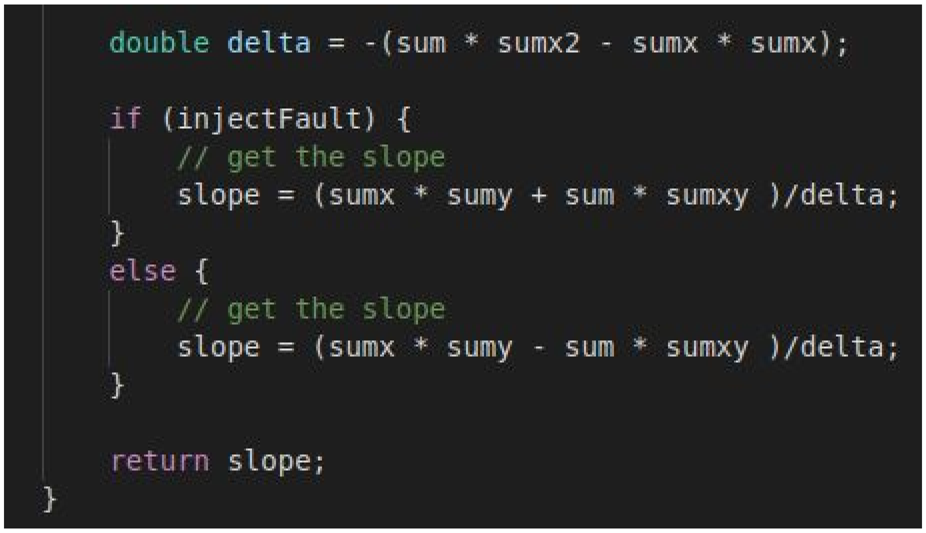
When considering what could go wrong with implementing the Euclidean distance formula, we thought that it might be realistic for someone to compute the ‘inside’ of the distance formula, and then after computing it they would return that value, instead of taking the square root of the computed value. This fault resulted in failing 3/5 of the test cases. The two test cases that passed were when the points were all the same, or they were all 0.

Method: compareTo


```
/**
 * for sorting coordinates by their x value
 * @see java.lang.Comparable#compareTo(java.lang.Object)
 */
public int compareTo(Object otherCoord) throws ClassCastException {
        if (!(otherCoord instanceof PhaseSpaceCoordinate)) throw new ClassCastException("A Person object expected.");
        double otherX = ((PhaseSpaceCoordinate)otherCoord).xValue;
        if (injectFault) {
            if (this.xValue < otherX) return 1;
            if (this.xValue > otherX) return -1;
        }
        else {
            if (this.xValue < otherX) return -1;
            if (this.xValue > otherX) return 1;
        }
        return 0;
}
```

Here we decided that it could be possible for a programmer to accidentally switch the proper returns for the conditionals. This failed all test cases besides one, where the inputted points were all the same.

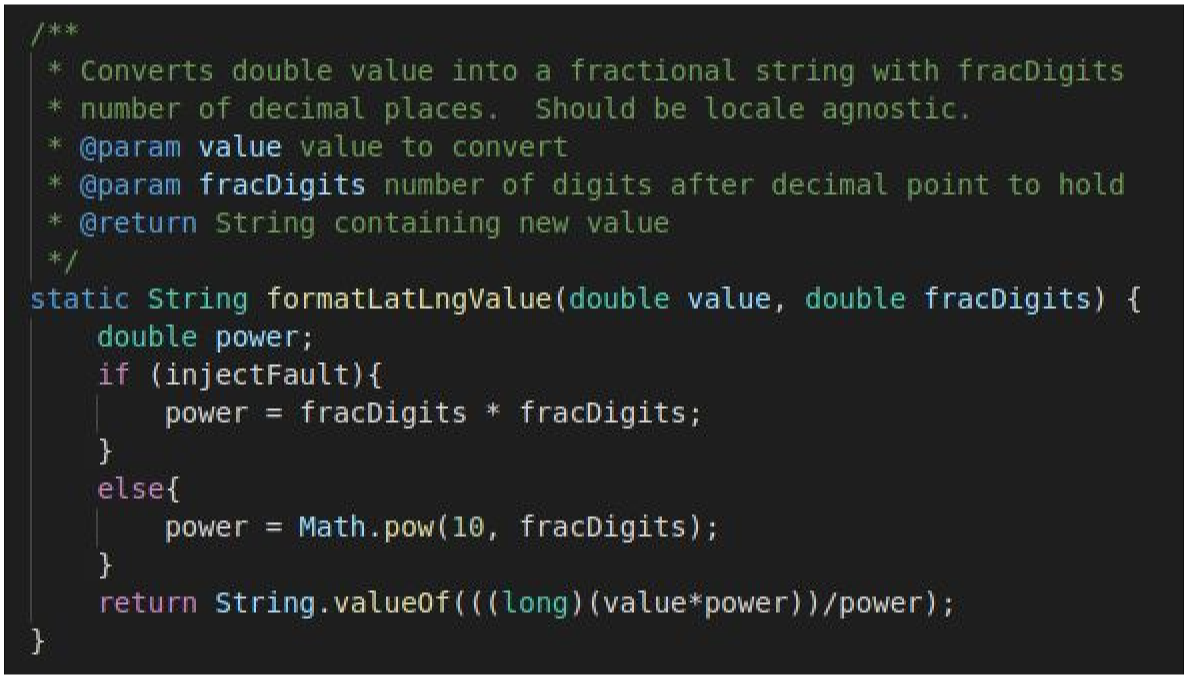
Method: calculateSlope


```
        double delta = -(sum * sumx2 - sumx * sumx);

        if (injectFault) {
            // get the slope
            slope = (sumx * sumy + sum * sumxy )/delta;
        }
        else {
            // get the slope
            slope = (sumx * sumy - sum * sumxy )/delta;
        }

        return slope;
    }
```

For this method, we used subtraction where there was supposed to be addition. It failed all test cases. We could not think of a change that would result in some test cases passing, so we had to decide on a fault that could be realistic, not break the code, but unfortunately fail all test cases.

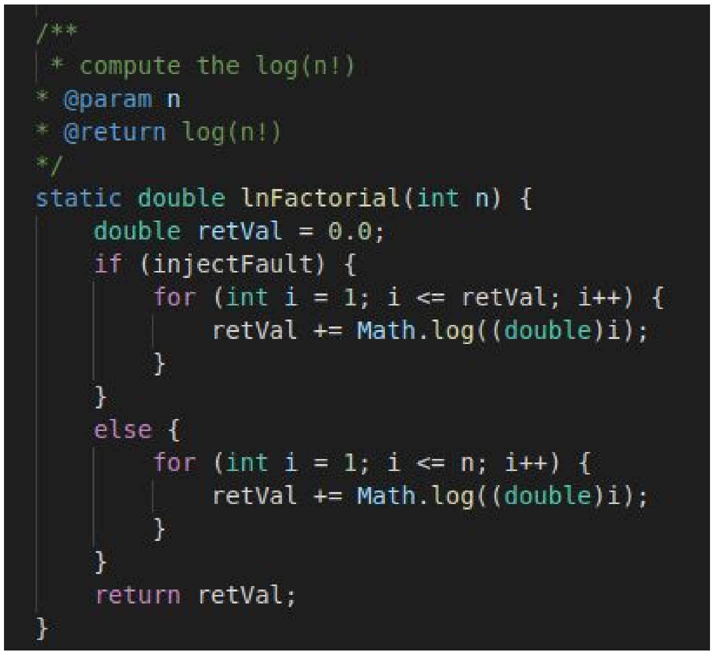
Return to Table of Contents

Method: formatLatLngValue


```
/**
 * Converts double value into a fractional string with fracDigits
 * number of decimal places.  Should be locale agnostic.
 * @param value value to convert
 * @param fracDigits number of digits after decimal point to hold
 * @return String containing new value
 */
static String formatLatLngValue(double value, double fracDigits) {
    double power;
    if (injectFault){
        power = fracDigits * fracDigits;
    }
    else{
        power = Math.pow(10, fracDigits);
    }
    return String.valueOf(((long)(value*power))/power);
}
```

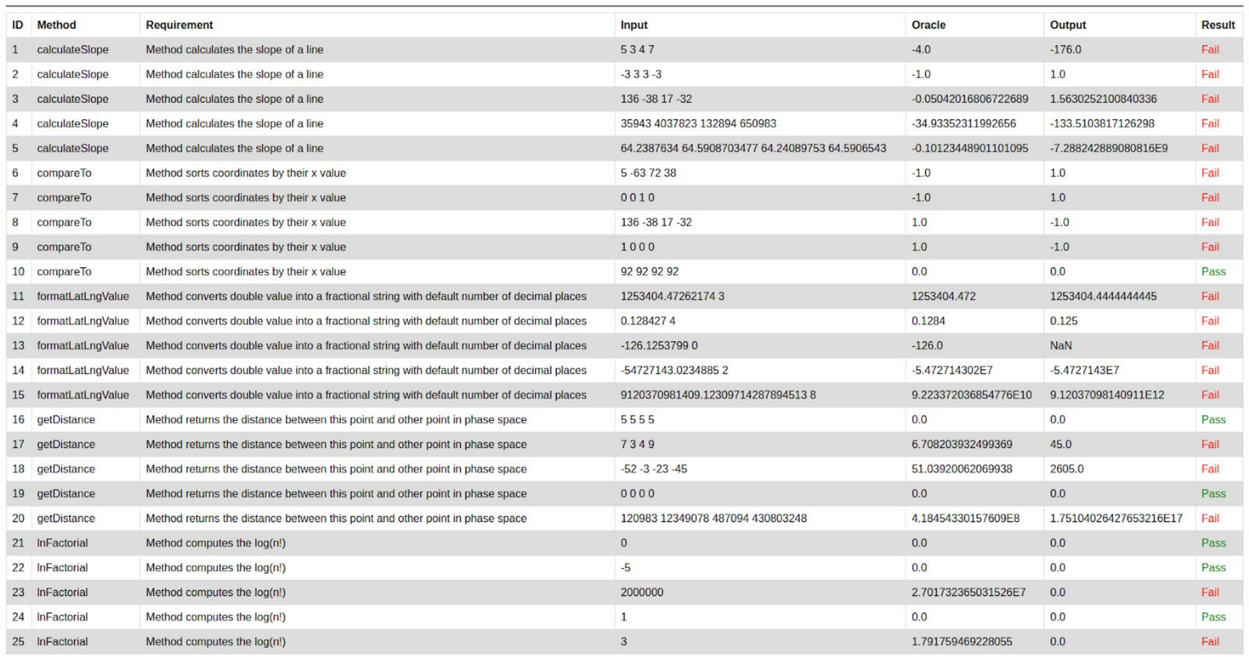
For this method, we could not think of something that would still pass some of the test cases, but we decided that someone could make a mistake when formatting the number of decimal places by computing the power as fracDigits * fracDigits, instead of taking 10^fracDigits.

Method: lnFactorial


```
/**
 * compute the log(n!)
* @param n
* @return log(n!)
*/
static double lnFactorial(int n) {
    double retVal = 0.0;
    if (injectFault) {
        for (int i = 1; i <= retVal; i++) {
            retVal += Math.log((double)i);
        }
    }
    else {
        for (int i = 1; i <= n; i++) {
            retVal += Math.log((double)i);
        }
    }
    return retVal;
}
```

For this method, we changed how many times the for loop would iterate for computing the log(n!). We thought it might be possible for someone to make a mistake and set the for loop to run retVal times, instead of n times. This passed three test cases, and failed two. It passed the test cases where the inputted value was -5, 0, and 1.

Final Testing Report (With Faults Live)



ID	Method	Requirement	Input	Oracle	Output	Result
1	calculateSlope	Method calculates the slope of a line	5 3 4 7	-4.0	-176.0	Fail
2	calculateSlope	Method calculates the slope of a line	-3 3 3 -3	-1.0	1.0	Fail
3	calculateSlope	Method calculates the slope of a line	136 -38 17 -32	-0.05042016806722689	1.5630252100840336	Fail
4	calculateSlope	Method calculates the slope of a line	35943 4037823 132894 650983	-34.93352311992656	-133.5103817126298	Fail
5	calculateSlope	Method calculates the slope of a line	64.2387634 64.5908703477 64.24089753 64.5906543	-0.10123448901101095	-7.288242889080816E9	Fail
6	compareTo	Method sorts coordinates by their x value	5 -63 72 38	-1.0	1.0	Fail
7	compareTo	Method sorts coordinates by their x value	0 0 1 0	-1.0	1.0	Fail
8	compareTo	Method sorts coordinates by their x value	136 -38 17 -32	1.0	-1.0	Fail
9	compareTo	Method sorts coordinates by their x value	1 0 0 0	1.0	-1.0	Fail
10	compareTo	Method sorts coordinates by their x value	92 92 92 92	0.0	0.0	Pass
11	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	1253404.47262174 3	1253404.472	1253404.4444444445	Fail
12	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	0.128427 4	0.1284	0.125	Fail
13	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	-126.1253799 0	-126.0	NaN	Fail
14	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	-54727143.0234885 2	-5.472714302E7	-5.4727143E7	Fail
15	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	9120370981409.12309714287894513 8	9.223372036854776E10	9.12037098140911E12	Fail
16	getDistance	Method returns the distance between this point and other point in phase space	5 5 5 5	0.0	0.0	Pass
17	getDistance	Method returns the distance between this point and other point in phase space	7 3 4 9	6.708203932499369	45.0	Fail
18	getDistance	Method returns the distance between this point and other point in phase space	-52 -3 -23 -45	51.03920062069938	2605.0	Fail
19	getDistance	Method returns the distance between this point and other point in phase space	0 0 0 0	0.0	0.0	Pass
20	getDistance	Method returns the distance between this point and other point in phase space	120983 12349078 487094 430803248	4.18454330157609E8	1.75104026427653216E17	Fail
21	lnFactorial	Method computes the log(n!)	0	0.0	0.0	Pass
22	lnFactorial	Method computes the log(n!)	-5	0.0	0.0	Pass
23	lnFactorial	Method computes the log(n!)	2000000	2.701732365031526E7	0.0	Fail
24	lnFactorial	Method computes the log(n!)	1	0.0	0.0	Pass
25	lnFactorial	Method computes the log(n!)	3	1.791759469228055	0.0	Fail

This displays our updated test case report following our fixes to the script, with the faults live.

Thoughts on Fault Injection

Ultimately, we found that it was really important to fully test methods, because faults can exist that are not going to cause errors to be thrown, but could cause the method to behave in a way that it is not supposed to. This deliverable has been enlightening to us in the importance of double/triple/quadruple checking things, and making sure that the test cases are catching a wide range of potentialities of method uses. This shines a light on the fact that errors are not going to be caught by the compiler 100% of the time, because logic errors need to be found via exhaustive testing or additional human scrutinization of the code.

Chapter Six

Closing Thoughts

At the beginning of the semester, this project seemed insurmountable. Between the lack of experience with GitHub, scripts, and overall naivete, the prospect of tackling this project was daunting. In addition to the unfamiliar territory we were entering, our team consisted of a group of people who had no prior experience with each other. As was stressed again and again throughout the course, planning is everything. Planning requires communication, and overall team cohesion. So not only was the project introducing us to new aspects of computer science, we were also introduced to an environment in which we needed to adapt, and maintain proper communication in order to reach our milestones, and ultimately to deliver a successful project. Luckily, communication was not lacking. While we made several mistakes along the way, we were able to fix them, even though it took more than one try, due to all team members' willingness to provide solutions and ideas to resolve problems. Every mistake was a team mistake, and every resolution was arrived at due to team effort.

Thoughts on Project Execution

We thought that the timing between deliverables was reasonable. We never felt rushed to complete something, even though we had a lot of errors. The feedback from Dr. Bowring was exponentially helpful, as we began trending in the proper direction. The amount of guidance we were given was just enough to be able to implement the project, but we had a lot of independent learning we had to do to gap the final mile. The specifications were clear and concise. The only problems we ran into were a direct consequence of misinterpreting or misreading the specifications.

Return to Table of Contents