ECE 322
Lab Report 5

Arun Woosaree
XXXXXXX

December 6, 2019

1 Introduction

The purpose of this lab was to serve as an introduction to mutation testing techniques, as well as regression testing.

Mutation testing uses the idea of automatic code mutation to determine how well the tests a tester writes actually are at catching errors in the application. The idea is the change, or 'mutate' the source code by changing equality operator, mathematical operators, among other things. Each copy of mutated code is referred to as a 'mutant', and the same set of tests is run on the mutants. In order to do the mutation testing, we must first begin with a test suite where all the tests are green. The idea is that when a mutant fails a test, it is 'killed' and the goal is to make sure that your test suite kills all mutants. Any mutants that survive are a sign that the tests could be improved, since they did not catch the bugs the mutants introduced. In this lab, we test an Array Library program using the mutation testing technique. The program is written in Java, and was tested using JUnit 5, and the PIT testing library was used to generate the mutants and evaluate whether they survive the test suite or not.

With regression testing, we are concerned with testing a system again after an update is made to the code base. For example, when a new feature or 'bug fix' is made, the regression test suite is run to make sure that the change to the code base does not re-introduce bugs into the system. The idea is that for each iteration of the software, when a bug is found, a bugfix is applied, and a test is also written which checks for that bug. That way, when a new feature is applied or some other change is made to the codebase, we can make sure that the new changes do not regress and bring back bugs which were previously already fixed. In this lab, we test a Math library program using the regression testing technique. The program is written in Java and a test suite was created with JUnit 5.

The project was built using Java 13. A **build.gradle** is provided for ease of use, from which an IDE like Intellij or Eclipse should be able to install dependencies from and run the tests for both projects. Alternatively, the command

`./gradlew test` can be run from the command-line. `./gradlew pitest` can be run from the command-line for part 1 of this lab to generate the mutation test report. The test suites for each project are located in `src/main/test/java` relative to the project root.

2 Task 1

For part one of this lab, a simple Array Helper library written in Java was tested using mutation testing. A Java library named PIT test was used to generate mutants of the ArrayLib class. Test cases were created which had 100% line coverage and branch coverage. These are outlined in the table which can be found in Appendix A The failing tests are explained as follows:

1. withoutTestRemoveTwo: This test fails because this method depends on ArrayLib's implementation of indexOf, which only returns the first occurrence of an element. This results in the method only removing one occurrence of a repeated element.

2. withoutTestRemoveFirstElement: This test fails because in the method, there is a check to see if $index > 0$, but it should be $index \geq 0$. Because it strictly checks for $index > 0$, the first element is never considered for removal.

3. intersectionTestDuplicate: This test fails because if there are elements that appear more than once in both arrays, then the method will attempt to increment the index of the intersection array multiple times. However, this array is currently limited to the length of array a. To avoid this error, the size of the intersection array should be the length of array a plus the length of array b.

To do the mutation testing, the three failing tests above were commented out, since mutation testing has a prerequisite, which is that the test suite must be green. With the PIT tool, 37 mutants were created, and 36 were killed. (More on this later).

ArrayLib.java

Mutations

9	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
10	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer subtraction with addition → KILLED
12	1. mutated return of Object value for ArrayLib::reverse to (if (x != null) null else throw new RuntimeException) → KILLED
21	1. negated conditional → KILLED
22	1. Changed increment from 1 to -1 → KILLED
26	1. mutated return of Object value for ArrayLib::unique to (if (x != null) null else throw new RuntimeException) → KILLED
34	1. Changed increment from 1 to -1 → KILLED
36	1. negated conditional → KILLED
37	1. Changed increment from 1 to -1 → KILLED
42	1. mutated return of Object value for ArrayLib::intersection to (if (x != null) null else throw new RuntimeException) → KILLED
49	1. Replaced integer addition with subtraction → KILLED
51	1. Changed increment from 1 to -1 → KILLED
54	1. Changed increment from 1 to -1 → KILLED
57	1. mutated return of Object value for ArrayLib::union to (if (x != null) null else throw new RuntimeException) → KILLED
65	1. negated conditional → KILLED
66	1. Changed increment from 1 to -1 → KILLED
69	1. Replaced integer subtraction with addition → KILLED
72	1. negated conditional → KILLED
73	1. Changed increment from 1 to -1 → KILLED
77	1. mutated return of Object value for ArrayLib::compact to (if (x != null) null else throw new RuntimeException) → KILLED
81	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
82	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
83	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
86	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
91	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
92	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
95	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
104	1. changed conditional boundary → SURVIVED

	2. negated conditional → KILLED
108	1. mutated return of Object value for ArrayLib::without to (if (x != null) null else throw new RuntimeException) → KILLED

Active mutators

- CONDITIONALS_BOUNDARY_MUTATOR
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- ArrayLibTest.withoutTest() (5 ms)
- ArrayLibTest.reverseTest() (5 ms)
- ArrayLibTest.indexOfTest() (147 ms)
- ArrayLibTest.intersectionTest() (7 ms)
- ArrayLibTest.unionTest() (5 ms)
- ArrayLibTest.uniqueTest() (5 ms)

Report generated by PIT 1.4.9

Now, for the one mutant that survived, it is actually directly related to a test failure mentioned above, which we commented out before running PITest. Specifically, it has to do with the if statement which checks if $index > 0$ instead of $index \geq 0$ on line 104.

Before the mutation testing, we found three bugs in the program. After the mutants were created and killed, only one of the bugs was highlighted. By the mutation test report. In this case, the mutation testing did not expose holes in the succeeding test cases, but this was mostly due to the luck of the tester writing good test cases. In other applications, mutation testing can and will help find holes in weak test suites. This is reinforced by the fact that after we commented out the failing tests, the mutation test caught one of the bugs because a mutant survived when the condition on line 104 $index > 0$ was mutated.

I think in the real world mutation testing can be used as an indicator, a sort of 'litmus' test, if you will for the quality of a test suite. It by no means should be the only thing one checks for to evaluate the quality of a testing suite, but on its own, it can reveal some weaknesses in a test suite. Like any other testing strategy, it is not a silver bullet, and the right testing strategies should be used for the right situations. In general, mutation testing excels at checking boundary conditions, which can be extremely useful, since programmers often mix up conditions such as 'less than' or 'less than or equal to'.

3 Task 2

For part 2 of this lab, a second Java application was tested using another testing technique known as regression testing. This application is a Math library which implements some common functions such as max, generating random arrays, standard deviation, and more. A summary of all the test cases discussed can be found in Appendix B.1.

A first iteration of the program was tested, in which two bugs were found, described below. There were a total of nine test cases for this part.

1. <u>normalizeTest</u>: The normalize method almost works, but there is a small bug where it subtracts the i_{th} element from the minimum, when it should actually be $values[i] - min$ In `Commit.java` this fix was implemented.

2. <u>arrayAddTestDifferentLength</u>: This test fails because the method assumes that both arrays given to it are the same length. The test expects an AssertionFailed error. This test case was not fixed in `Commit.java`

Next, there was an incremental upgrade to the code in the Commit.java file. The same test suite was run on it, and also new test cases were created to cover new functionality of the application. These test cases are outlined in Appendix B.1. In this iteration, four new test cases were added for the new functionality. Six out of thirteen test cases failed. Using a diff tool, we see that some boundary conditions were changed in the max and min methods, but they ultimately did not affect the correctness of the program, since the difference is that it now sets the max or min to the element in the array if it is greater than or equal to the maximum instead of strictly greater than, and for minimum, it sets the minimum if the element is less than or equal to the minimum instead of strictly less than. Either way, the global maximum and minimum will be the same.

1. arrayDeviationTest: This method returns null, no matter the input. It should return an array.

2. arrayAddTestDifferentLength: Again, this does not assert that the input arrays have the name length This test fails for the same reason as the previous test did, so nothing changed here.

3. distanceTest: This method returns zero, regardless of the input

4. arraySubtractTestDifferentLength: Again, this does not assert that the input arrays have the name length

5. negateTest: This method now returns a value which is one less than the expected output. This test was passing before, but it now fails due to a new change, which subtracts one from each array element in addition to negating it.

6. arraySubractTestSameLength: Again, this does not assert that the input arrays have the name length

While most of the new functionality failed tests anyways, due to not being fully implemented yet, one regression was caught using the regression testing method. At first glance, the negation method seems to have been inexplicably modified to also subtract one from each element. Because the array subtract method depends on negate, this test also fails.

Finally, for this part of the lab, we applied fixes to the software. The new code is available in Appendix B.2. With the fixes all thirteen test cases now pass. The full summary of test cases and their results can be found in Appendix B.1. Most of the fixes involve using existing functionality that the Java standard library provides, instead of re-inventing the wheel. These library functions have been battle tested by millions of users and tests, and are much less likely to have bugs than anything we write independently.

Regression testing, as the name implies is excellent for making sure one does not regress when adding new features to software. That is, when a bug is found, a test is made, so that when new functionality is added to the program, we ensure the bug does not show up again, and if it is re-introduced, the test will catch it. The lab demonstrated how well this technique works with the test for the negation method catching a bug which was introduced in the new commit where someone made it also subtract one from each element in the array in addition to negating the element.

4 Conclusion

In this lab, we were introduced to mutation and regression testing. We tested two applications written in Java. The first application is an Array library. This was tested with the mutation testing strategy, using a library called PITest. In general, mutation testing seems to be a useful tool for getting a feel for how robust a testing suite is. By generating mutant code, and making sure the mutants are caught and killed by the testing suite, we gain confidence in the robustness of the testing suite for each mutant killed. Like any other testing strategy, it is not a silver bullet, but I can see this technique being used as a 'litmus test' of sorts to give an initial impression for how good a test suite is. After all, if your test suite does not catch the mutants, and the code base is later updated when a new feature is added for example and the feature introduces a bug, the system likely will not work as expected. With a more robust test suite, it is more likely that changes like this will be caught by the tests, and therefore won't be introduced to the system. Because computing mutants takes up a considerable amount of resources, I can see this technique working well for small to medium-sized systems. I think that for larger systems, computing the mutants and running the test suites may be too time-consuming.

We also tested a Math application for this lab using the regression testing strategy. In general, regression testing seems to be really useful for what it is meant for: making sure previously found bugs are not re-introduced into the system. I can see it being useful for small and large systems alike. By making sure known bugs are not reintroduced into the system, developer time and effort is not wasted dealing with issues more than once, since the tests that cover each regression test should do the work of catching the bugs as opposed to the developer trying to trace an issue which was previously fixed before, but some new feature undid the previous fix.

Overall, both mutation testing and regression testing techniques seem to be really useful and practical testing techniques. Mutation testing is more concerned with the quality of the test suite, while regression testing is concerned with not re-introducing old bugs, and to prevent regression as the name suggests. I think that these techniques should not be used alone, but rather together with other testing techniques learned over the course of the labs. Making sure your test suite is robust enough to catch small changes is important, as is the ability to automatically detect and prevent a regression when a new feature is added to an application. Used in compination with other testing techniques, these are very powerful.

A Part 1 Testing Results

	input	expected	actual	
reverseTest	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	PASS
uniqueTest	[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, null]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
intersectionTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9] b: [3, 4, 5]	[3, 4, 5]	[3, 4, 5]	PASS
unionTest	a: [1, 2, 3, 4] b: [5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
indexOfTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: 5	4	4	PASS
	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: null	10	10	PASS
withoutTest	array: [1,2,3,4, 4,5,6,7,8,9], remove: [3, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	PASS
withoutTestRemoveTwo	array: [1,2,3,4,5,6,7,8,9], remove: [3, 4, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 4, 6, 7, 8, 9]	FAIL
withoutTestRemoveFirstElement	array: [1,2,3,4,5,6,7,8,9], remove: [1, 3, 4, 5]	[2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	FAIL
intersectionTestDuplicate	a: [1, 2, 2, 3, 4] b: [2, 2, 3, 4]	[2, 2, 3, 4]	IndexOutOfBoundsException	FAIL

B Part 2

B.1 Test results

	Input	Expected	Actual - MathPackage.java	Actual - Commit.java	Actual - Fixed.java	MathPackage.java	Commit.java	Fixed.java
randomTest:	Generate 1000 arrays using the random function	all values contained within [a,b]	all values contained within [a,b]	all values contained within [a,b]	all values contained within [a,b]	PASS	PASS	PASS
maxTest	Generate 1000 arrays, sort them and compare the last element with the return value of max	last element of array == max	last element of array == max	last element of array == max	last element of array == max	PASS	PASS	PASS
minTest	Generate 1000 arrays, sort them and compare the first element with the return value of min	first element of array == min	first element of array == min	first element of array == min	first element of array == min	PASS	PASS	PASS
normalizeTest	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	[0, -0.1, -0.2, -0.3, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1.0]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	FAIL	PASS	PASS
	[0, 33, 66, 100]	[0, 0.33, 0.66, 1.0]	[0, -0.33, -0.66, -1.0]	[0, 0.33, 0.66, 1.0]	[0, 0.33, 0.66, 1.0]	FAIL	PASS	PASS
sumTest	Generate 1000 arrays, compare the sum against Java's built in method for calculating sum	sum	sum	sum	sum	PASS	PASS	PASS
stddev	[0.8448535275473217, 0.7356655820407797, 0.9431584337138527, 0.9885039386933584, 0.26752295318703023, 0.30118478380150293, 0.5614795756611817, 0.27410185661451103, 0.3727308526619961, 0.9350197461150006, 0.3430515987014403, 0.14325655442623153, 0.09474732578722389, 0.8118925398379901, 0.7889524243808093, 0.6174482765472112, 0.08282552694523304, 0.9906861613488818, 0.1006718689797883, 0.07076982982755198]	0.33383429942515 +/- 1e-10	0.33383429942515 +/- 1e-11	0.33383429942515 +/- 1e-12	0.33383429942515 +/- 1e-13	FAIL	FAIL	FAIL
	[24.351879794282766, -1.2369080960006045, 21.33618933269294, 38.40700888177969, 81.07224984081901, 14.773395426428706, -23.465737641671126, -29.74766797865162, -80.53273274920562, 82.69835022556401, 16.765612733609586, 85.27325420746189, -25.56842729615137, -85.17577250342885, 91.14866661682103, 47.56747687575623, 71.62517118559902, 26.263337092658247, -78.61956099645082, -18.809917650844213, 31.415431556119216, 15.021802528535659, -45.50777659561043, 76.15463305868838, 71.73129701754698, 42.00719896644702, 98.93219072662015, 29.911361379866946, -7.304486822031862, 72.46129116466469, 25.17287827553399, 6.610331737331364, 34.267027445622745, 10.796643684552663, -44.6877977606748, 99.08890435845663, 18.379318834519594, 54.01219466707613, -11.329038630356479, -23.230019858073646, 33.03020694951988, -59.205375268977974, -8.495912401158435, 83.71104890003406, 80.19371195739095, 70.17088288687546, -27.756415190854483, 67.22645204084466, 2.237307845125258, 90.69398038338696, 13.575182881518089, 66.0876889384611, -30.1366615029317, -20.084766272685243, 10.62213430500914, 70.12939720918465, -53.19292944066913, -84.43071331549132, -53.30861375844398, 36.75691360838863, 4.481177488201666, 89.23026437856967, 28.481208229172694, 4.041593290473799, -19.144188062781325, 71.48911820023875, -97.3407210733165, -82.95697886617639, -52.019808046281035, -20.500795201979386, -65.75240723052059, -76.4564498827019, -48.255740133965006, 20.75249609534157, -45.919570907416606, 55.70989509490502, -30.432532466301396, -51.16117165251921, -7.213072379759879, 62.480197271840865, -14.266598956238383, -14.496858737726953, -6.199152015512155, 86.73375309074737, 24.576298742407715, -50.335938636762336, -98.21921344917936, -97.29469852961215, 92.13635661651074, -6.363254915974139, 34.622868801612384, -47.36683311550864, -57.59764296582546, 53.28248250936096, 26.007936605144735, -86.505212849109, -7.68902146399742, -18.99542397736265, -5.991562076158388, 82.91883854718108]	54.00891607	54.00891607	54.00891607	54.00891607	PASS	PASS	PASS
arrayAddTestSameLength	Generate 1000 * 2 arrays with the same length. Compare the result against an alternative way to sum the arrays using Java 8 lambdas	element-wise sum	element-wise sum	element-wise sum	element-wise sum	PASS	PASS	PASS
arrayAddDifferentLength	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	[11, 11, 11, 11, 11, 11]	[11, 11, 11, 11, 11, 11]	Assertion error different lengths	FAIL	FAIL	PASS
	d1: [1, 2, 3, 4, 5, 6] d2: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	Assertion error different lengths	IndexOutOfBoundsException	IndexOutOfBoundsException	Assertion error different lengths	FAIL	FAIL	PASS
negateTest	Generate 1000 arrays, and negate them, compare against return result	array negation	array negation	array negation, but 1 is also subtracted from each element	array negation	PASS	FAIL	PASS

arraySubtractTestSameLength	Generate 1000 * 2 arrays with the same length. Compare the result against an alternative way to subtract the arrays using Java 8 lambdas	element-wise subtraction	n/a	element-wise subtraction	element-wise subtraction		PASS	PASS
arraySubtractTestDifferentLength	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	n/a	[-9, -7, -5, -3, -1, 1]	Assertion error different lengths		FAIL	PASS
	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	n/a	[9, 7, 5, 3, 1, -1]	Assertion error different lengths		FAIL	PASS
distanceTest	d1 or d2 are not of length 2	Assertion error different lengths	n/a	0	Assertion error different lengths		FAIL	PASS
	d1: [-1, -2] d2: [3, 4]	7.211102551	n/a	0	7.211102551		FAIL	PASS
	d1: [6, 7] d2: [-8,-9]	21.26029163	n/a	0	21.26029163		FAIL	PASS
arrayDeviationTest	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[-4.5, -3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5, 4.5]	n/a	null	[-4.5, -3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5, 4.5]		FAIL	PASS
	[4, 8, 1, 3, 9, 5, 10, 2, 7, 6]	[-1.5, 2.5, -4.5, -2.5, 3.5, -0.5, 4.5, -3.5, 1.5, 0.5]	n/a	null	[-1.5, 2.5, -4.5, -2.5, 3.5, -0.5, 4.5, -3.5, 1.5, 0.5]		FAIL	PASS

	input	expected	actual	
reverseTest	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	PASS
uniqueTest	[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, null]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
intersectionTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9] b: [3, 4, 5]	[3, 4, 5]	[3, 4, 5]	PASS
unionTest	a: [1, 2, 3, 4] b: [5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
indexOfTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: 5	4	4	PASS
	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: null	10	10	PASS
withoutTest	array: [1,2,3,4, 4,5,6,7,8,9], remove: [3, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	PASS
withoutTestRemoveTwo	array: [1,2,3,4,5,6,7,8,9], remove: [3, 4, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 4, 6, 7, 8, 9]	FAIL
withoutTestRemoveFirstElement	array: [1,2,3,4,5,6,7,8,9], remove: [1, 3, 4, 5]	[2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	FAIL
intersectionTestDuplicate	a: [1, 2, 2, 3, 4] b: [2, 2, 3, 4]	[2, 2, 3, 4]	IndexOutOfBoundsException	FAIL

B.2 Fixed Code

```java
import java.util.Arrays;
import java.util.Random;

public class Fixed {

    /**
     * Creates an array of n random values in the range [a,b]
     *
     * @param n Number of values to generate
     * @param a lower bound
     * @param b upper bound
     * @return Array of random values
     */
    public static double[] random(int n, double a, double b) {
        double[] values = new double[n];
        for (int i = 0; i < values.length; i++)
            values[i] = a + (Math.random() * (b - a));

        return values;
    }

    /**
     * Returns the maximum values contained in the passed array
     *
     * @param values Array to search in
     * @return Highest value in passed array
     */
    public static double max(double[] values) {
        double max = Double.MIN_VALUE;
        for (int i = 0; i < values.length; i++) {
            if (max < values[i])
                max = values[i];
        }
        return max;
    }

    /**
     * Returns the minimum values of an array
     *
     * @param values Array to search through
     * @return Smallest value in the array
     */
    public static double min(double[] values) {
        double min = Double.MAX_VALUE;
```

```java
        for (int i = 0; i < values.length; i++) {
            if (min > values[i])
                min = values[i];
        }
        return min;
    }

    /**
     * Normalizes the values in the passed array to [0,1]
     *
     * @param values Array to be normalized
     * @return Normalized array
     */
    public static double[] normalize(double[] values) {
        double max = max(values);
        double min = min(values);
        double[] normalized = new double[values.length];

        for (int i = 0; i < values.length; i++)
            normalized[i] = (values[i] - min) / (max - min);

        return normalized;
    }

    /**
     * Calculates the sum of the array elements
     *
     * @param values Array to sum
     * @return summed value
     */
    public static double sum(double[] values) {
        double sum = 0.0;
        for (int i = 0; i < values.length; i++)
            sum += values[i];
        return sum;
    }

    /**
     * Calculates the standard deviation of the values in the array
     *
     * @param values Array to calculate deviation of
     * @return standard deviation
     */
    public static double stddev(double[] values) {
        double mean = sum(values) / values.length;
        double variance = 0;
```

```java
        for (int i = 0; i < values.length; i++)
            variance += Math.pow(values[i] - mean, 2);
        return Math.sqrt(variance / values.length);
    }

    /**
     * Adds two arrays together, element-wise
     *
     * @param d1 first array
     * @param d2 second array
     * @return result of addition
     */
    public static double[] arrayAdd(double[] d1, double[] d2) {
        assert d1.length == d2.length;

        double[] result = new double[d1.length];
        for (int i = 0; i < result.length; i++) {
            result[i] = d1[i] + d2[i];
        }
        return result;
    }

    /**
     * Negates the values in the array
     *
     * @param d values
     * @return result
     */
    public static double[] arrayNegate(double[] d) {
        return Arrays.stream(d).map(x -> -x).toArray();
    }

    /**
     * Subtracts two arrays element-wise
     *
     * @param d1 first array
     * @param d2 second array
     * @return result of subtraction
     */
    public static double[] arraySubtract(double[] d1, double[] d2) {
        assert d1.length == d2.length;
        double[] ret = new double[Math.max(d1.length, d2.length)];
        Arrays.setAll(ret, index -> d1[index] - d2[index]);
        return ret;
    }

```

```java
    /**
     * Calculates the Cartesian distance between points defined by d1 and d2
     *
     * @param d1 first point
     * @param d2 second point
     * @return Cartesian distance
     */
    public static double distance(double[] d1, double[] d2) {
        assert d1.length == 2;
        assert d2.length == 2;

        return Math.sqrt(Math.pow(d2[0] - d1[0], 2) + Math.pow(d2[1] - d1[1], 2));
    }

    /**
     * Calculates an array representing the deviation each value in
     * the array is from the mean value of the set
     *
     * @param d1 input array
     * @return deviation values array
     */
    public static double[] arrayDeviation(double[] d1) {
        double mean = Arrays.stream(d1).sum() / d1.length;

        double[] result = new double[d1.length];
        Arrays.setAll(result, index -> d1[index] - mean);
        return result;
    }

}
```