COMP4110-Assignment4-JUnit Testing of Trigonometric Functions

Group 11

Lama Khalil

Muhammad Faraz Sohail

Karam Chaban

S M Sadman Sakib Prottay

GitHub Link

https://github.com/LamaKha/Group11-Assign4-TrigsTesting

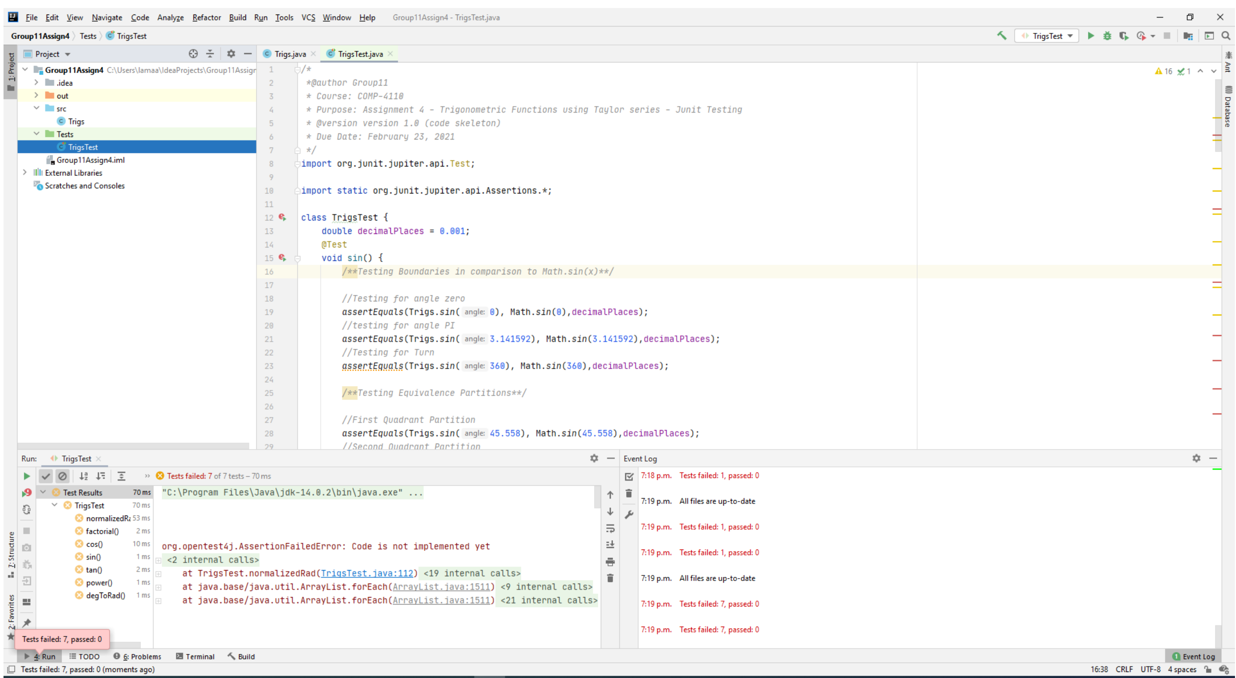
Version 7.0 (final)

Introduction

In this assignment we have implemented the *sin, cos* and *tan* trigonometric functions using Taylor Series in degree and radian. Additionally, we implemented four (4) helper method; namely: *factorial, power, DegtoRad*, and *normalizedRad*. To achieve this, we adapted the Test-Driven Development (TDD) method. Our framework consisted of Java programming language, IntelliJ, Eclipse, Junit5, and GitHub. To complete this task, we had four (4) versions of our code. Please note that the screenshots are taken by our team members from their own IDEs. Thus, you will notice a difference in the appearance.

Version 1.0 code and Testing Results

In *Version 1.0*, we implemented skeleton testcases and skeleton code of our basic functions. Below you will find a few code snippets and screenshots of some test results for version 1.0. You can check the full code in our GitHub repository by checking the commit history.



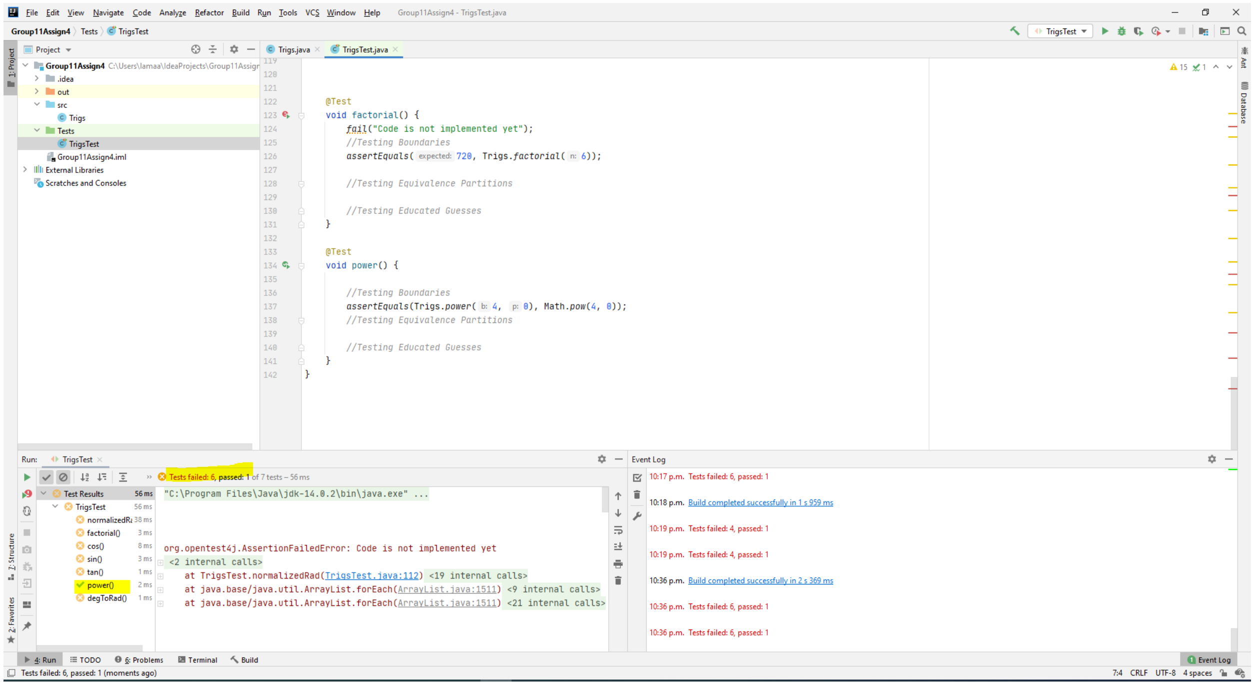
Version 1.0 Testcases skeleton

```java
public class Trigs {

        /**This method calculates sin(x)
         * Input: angle value
         * Return: sin value of x */
        public static double sin(double angle) {
            double sin=0;
            return sin;
        }

        /** This method calculates cos(x)
         * Input: angle value
         * Return: cos value of x */
        public static double cos(double angle) {
            double cos=0;
            return cos;
        }
        /** This method calculates tan(x)
         * Input: angle value
         * Return: tan value of x */
        public static double tan(double angle) {
            double tan=0;
            return tan;
        }

        /** This method coverts Degree to Radian
         * Input: x deg
         * Return: returns x in radian*/

        public static double DegToRad (double x) {
            return x ;
        }
```

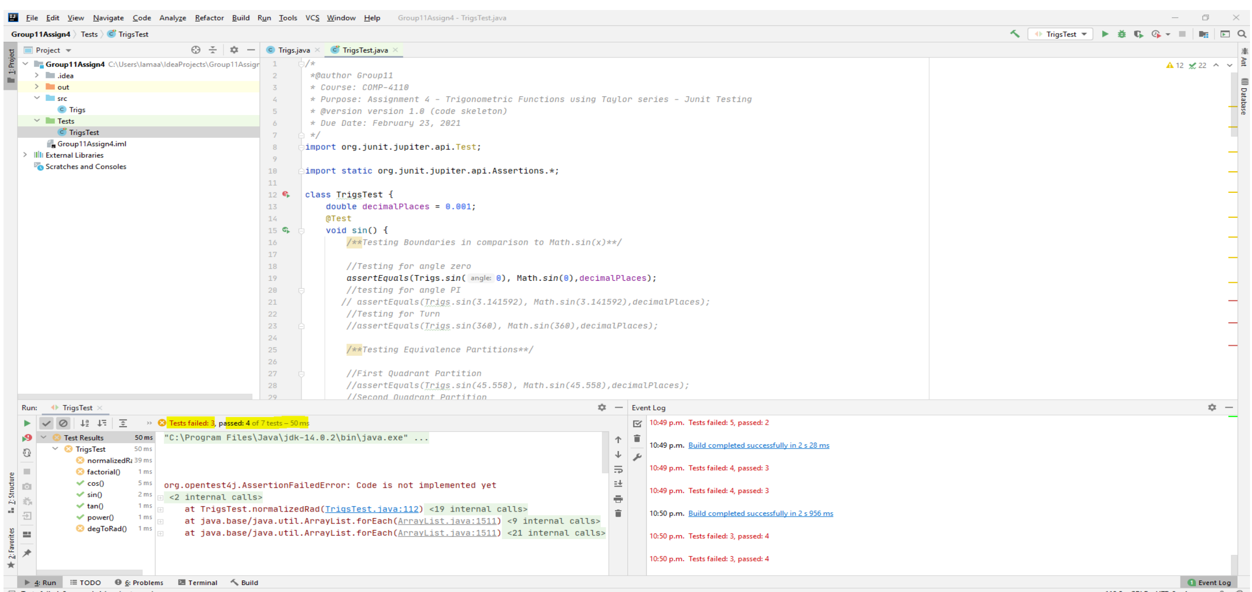
Version 1.0 Code skeleton

Version 2.0 code and Testing Results

In *Version 2.0* we added more test cases and implemented the basic of the functionalities. For example, we implemented our test cases based on the following methodologies: Boundaries Testing and Equivalence Partitions Testing. The according to these cases we developed the *sin(), cos(), tan(), degToRad(), normalizedRad(), factorial(), and power()* functions. Below we will add screenshots of the test case results for this version of the code. You can check the full code in our GitHub repository by checking the commit history.



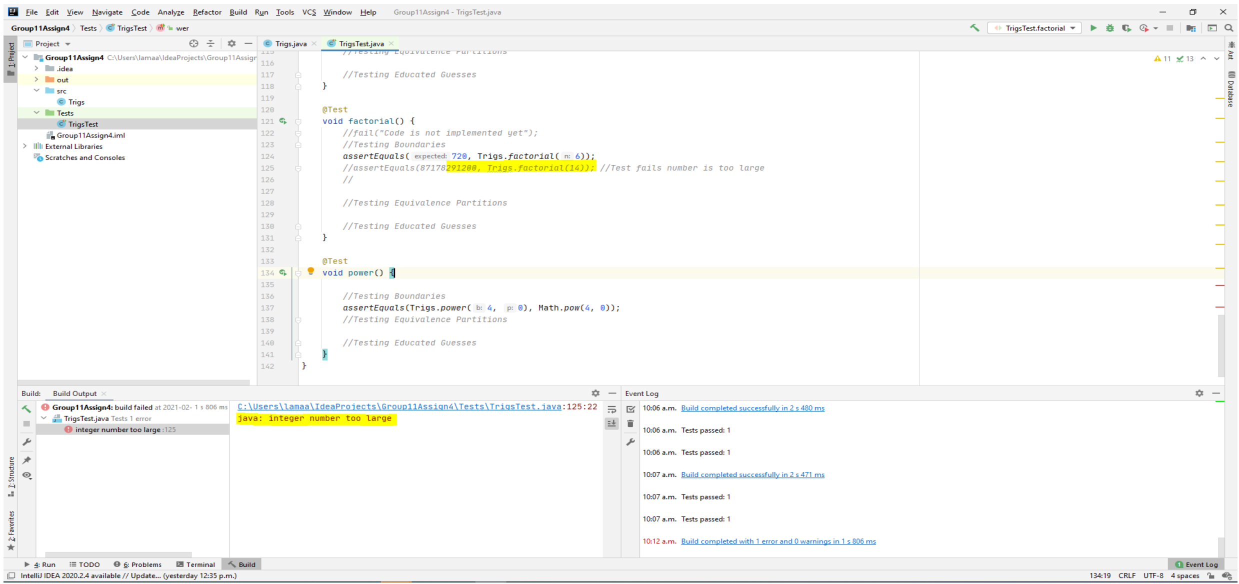
Version 2.0 Test Result - Implementing more test cases (a)



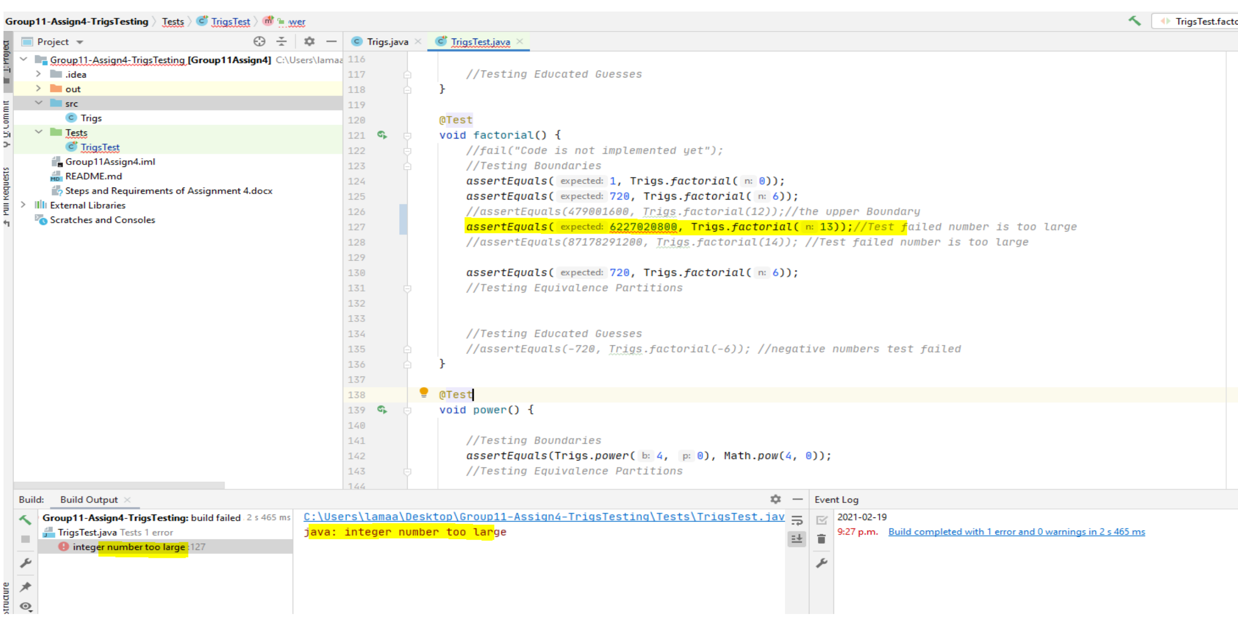
Version 2.0 Test Result - Implementing more test cases (b)

Version 3.0 code and Testing Results

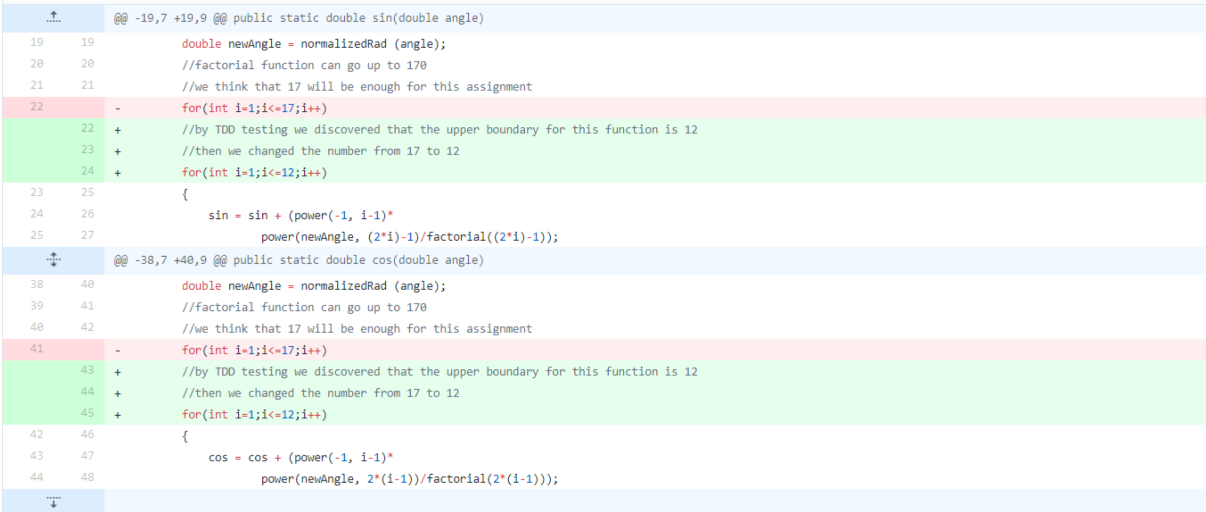
In *Version 3.0* we added more test cases and fixed some of the functions such as sin() and cos() due limitation in the factorial function which resulted from the boundaries testing. When we first coded the *sin, cos and factoria*l function, we checked iPhone and Google's scientific calculator for factorial limitation. It appeared that these calculators could compute up to 170! before going to infinity. Although the calculators where able to compute 170! The resulted number was extremely large. Therefore, we decided to check for the nearest number that the calculator can compute factorial for and give relatively smaller number. That is why we decided on number 17 to be the limit of the for loop inside *sin and cos* functions. However, when we used the boundary testing on the *factorial* function, we found that this code cannot calculate more than 12! Thus, we change the for loop in *cos() and sin()* functions to represent that. To track the changes in the code you can check our code repository on Github.



Version 3.0 Test Result - Implementing more test cases (a)



Version 3.0 Test Result - Implementing more test cases (b)

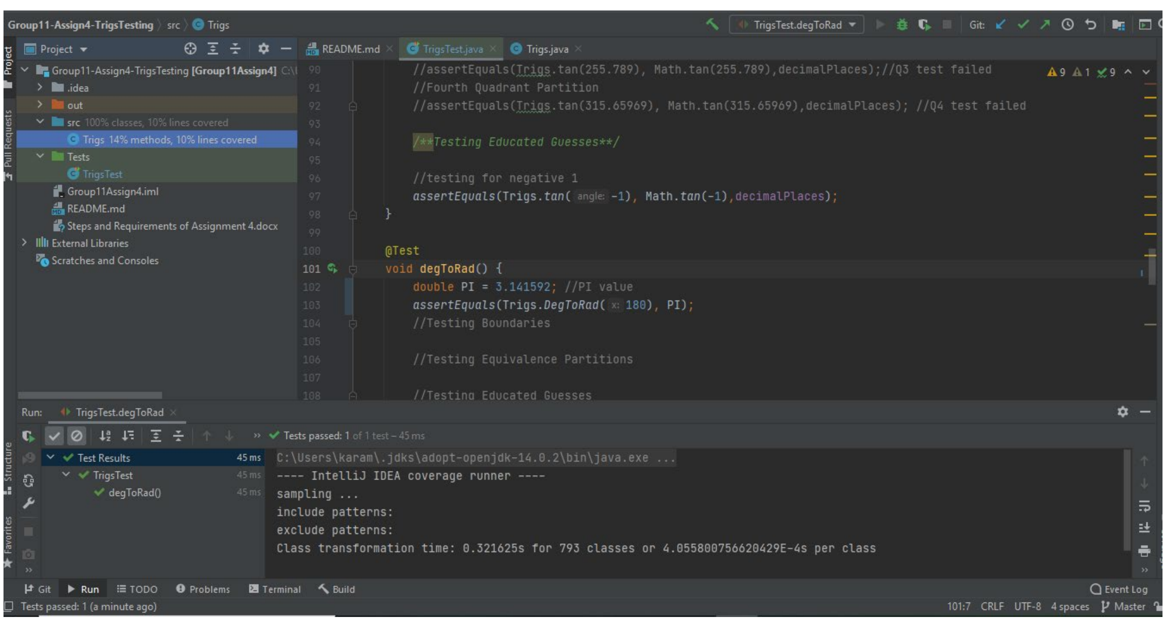


Version 3.0 -GitHub commit history tracing changes to sin() and cos()

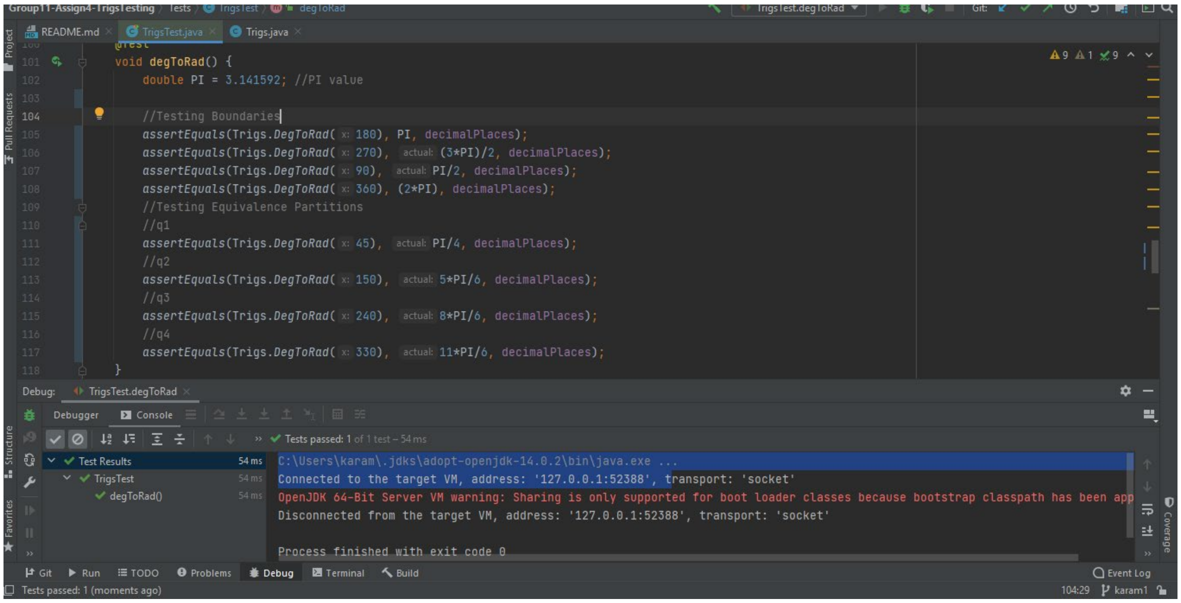
Version 4.0 code and Testing Results

In *Version 4.0,* we added test case to *degToRad(),* fingered out the bounds of our quadrants for the normalize function, and tested the *power()* method, as follows:

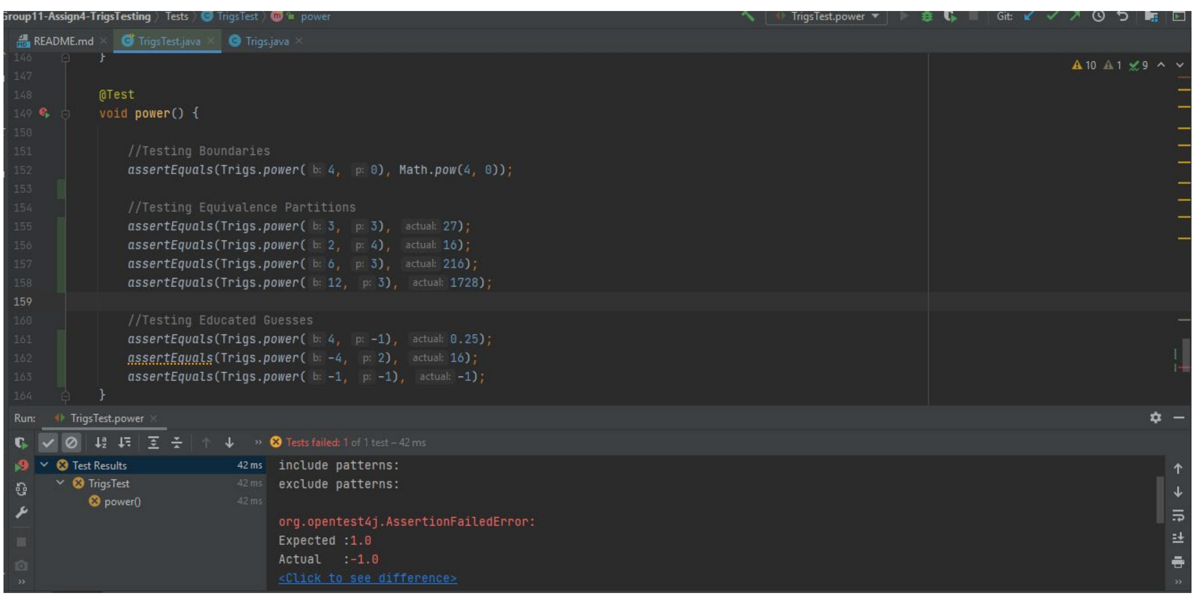
1. Below we have added a test case to the degToRad() method that check if 180 degrees is equal to pie in radians. We can see the test was in fact successful and thus can now develop more testcases.



2. Since all degrees will be normalized between 0 and 360, we are able to create testcases representing out input set from 0 to 360. This gives us the bounds of our quadrants and a value from each quadrant. Below you can see all test have been implemented and tested successfully.



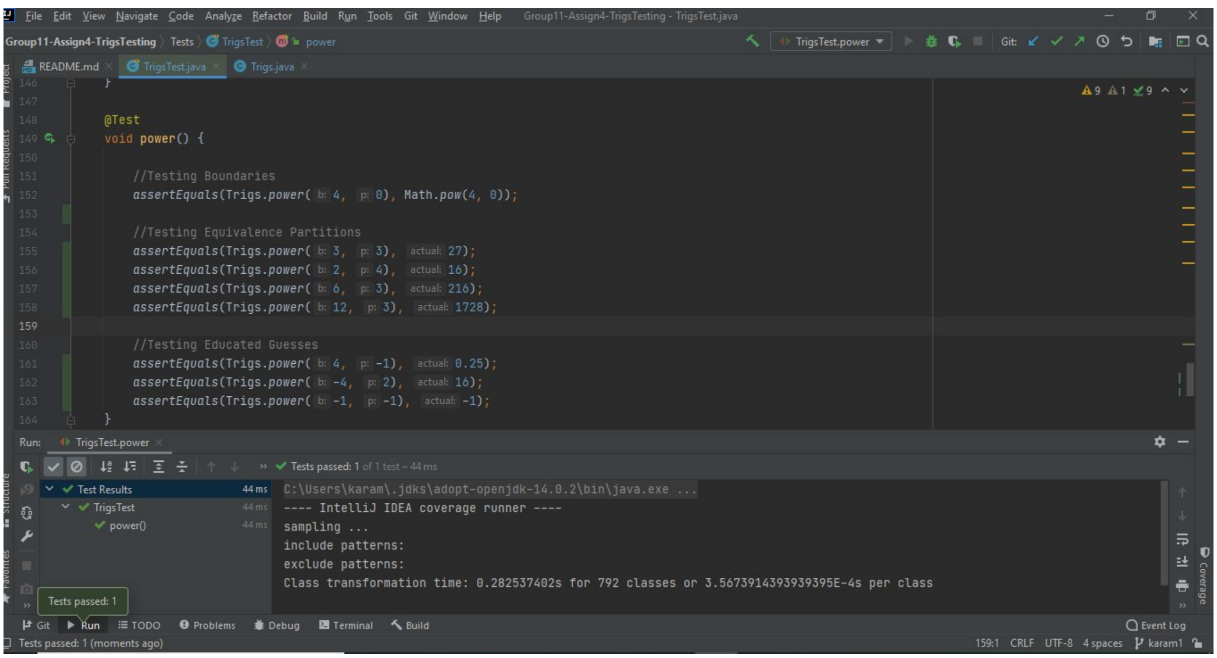
3. Now we move on to testing our power function. We started by creating test cases for equivalence classes for some odd and even powers and bases. Then we did some educated guessing where we picked different values based on properties of the value (odd/even, positive/negative) and then run my tests.



4. An error was detected when dealing with negative powers, so will go back to our code and look.

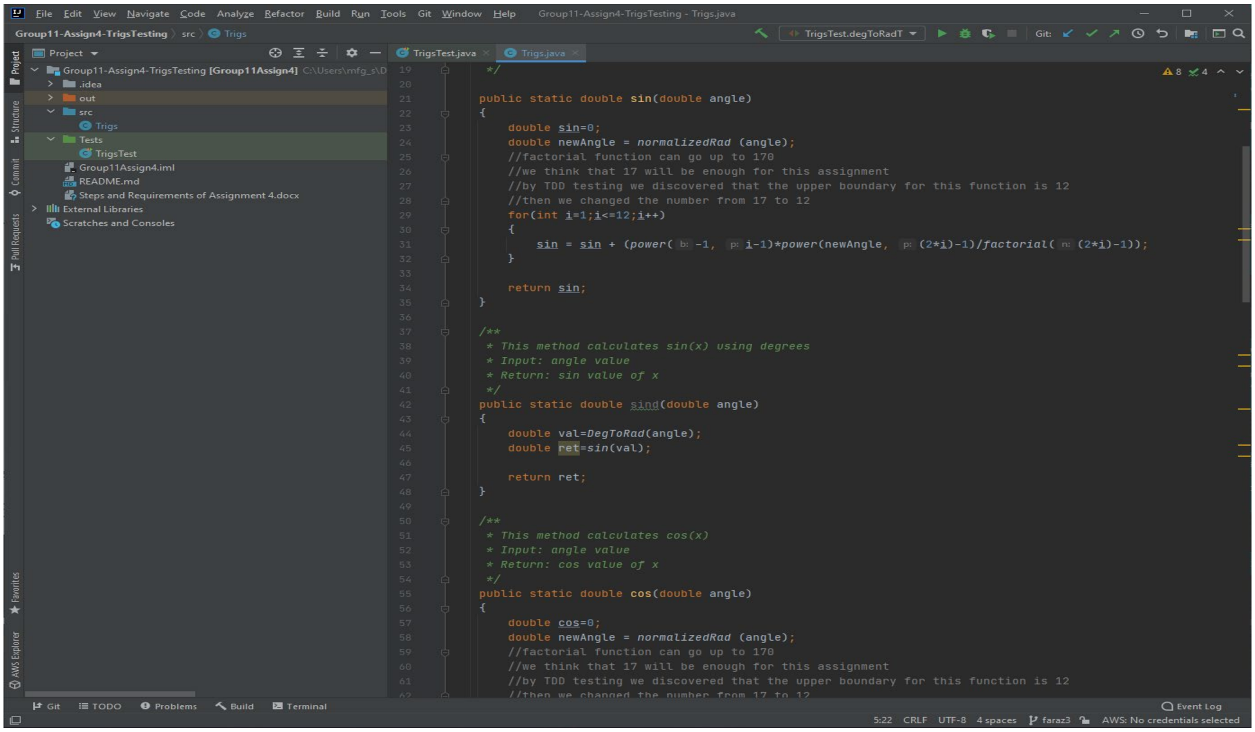
```java
public static double power(double b, int p)
{
    double result=1;

    if(p == 0)
        return result;
    else if ( p > 0)
        for(int i=1; i<=p; i++)
            result = result * b;

    return result;

}
```

5. There it is, we did not set up our function to deal with negative powers. Thanks to equivalence partitioning as I based my guessed test cases on a similar method, we were able to detect a bug. Below are the test results after the fix has been implemented.

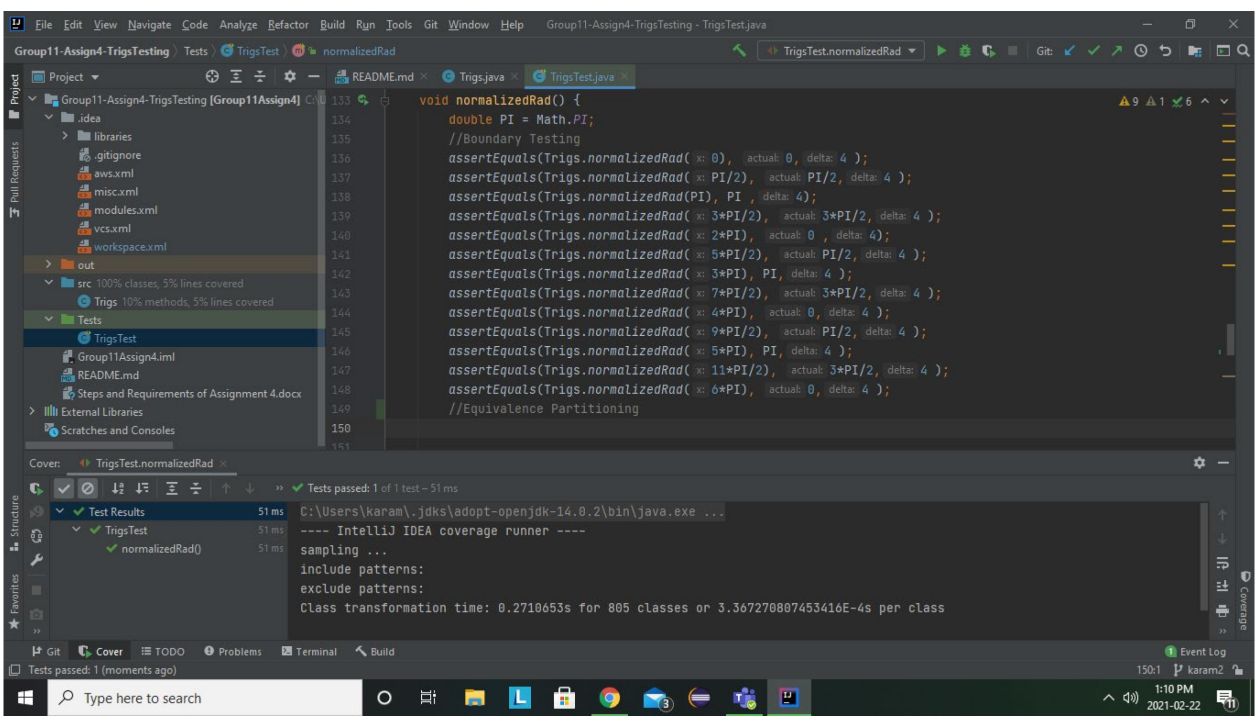


Version 5.0 code and Testing Results

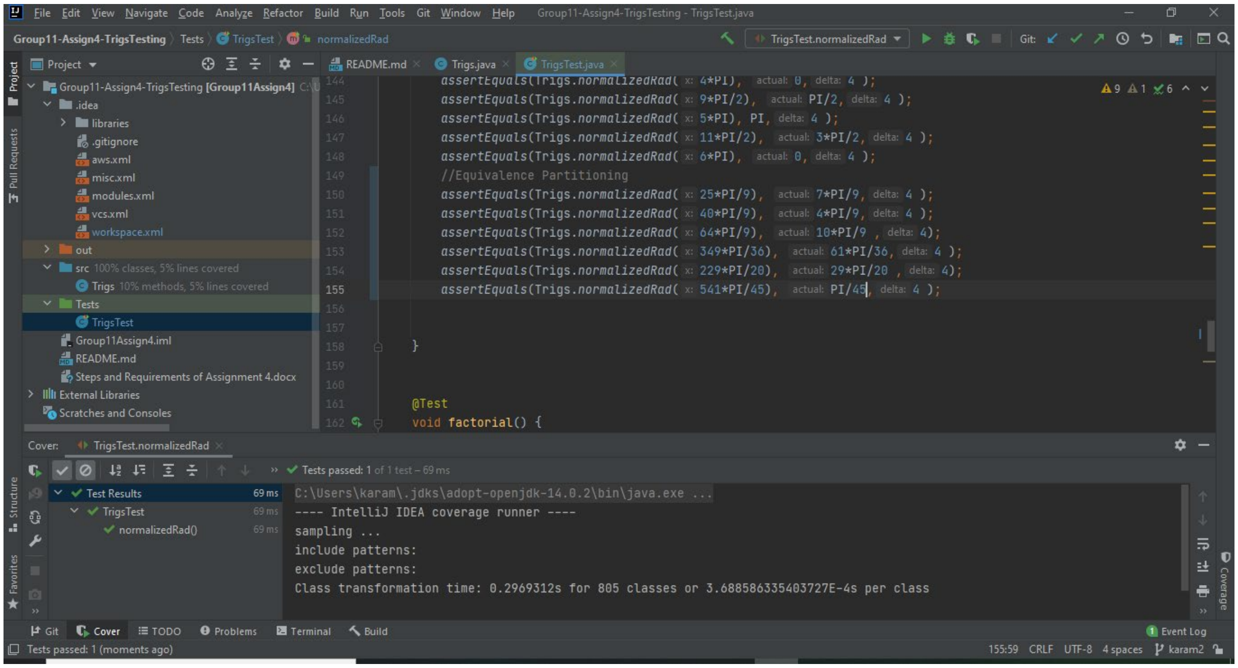
1. In this version we added the methods which calculate the Sin, Cos and Tan using Degree angle. These methods use the original sin methods in radians, first we convert the degrees in radians then we calculate the functions.



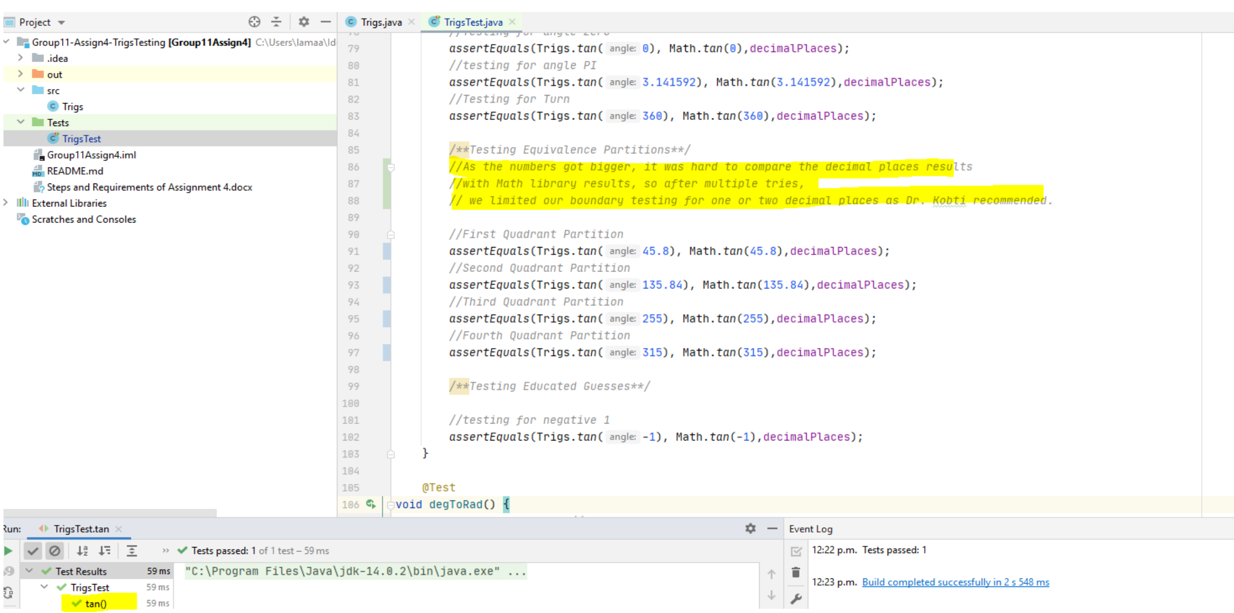
2. Now we have to test out normalizedRad() method which takes any radian value and outputs its radian value on the unit circle. A sample input would be 2π and the output should be 0. We started with boundary testing and tested every quadrant border in the first 3-unit circles. Tests were successful and are shown below: -



3. Now we will focus on equivalence partitioning and we consider each unit circle a partition. There are infinite amounts of unit circles and thus we will choose a random radian angle from the first 6-unit circles after the first. The tests and results are show below: -

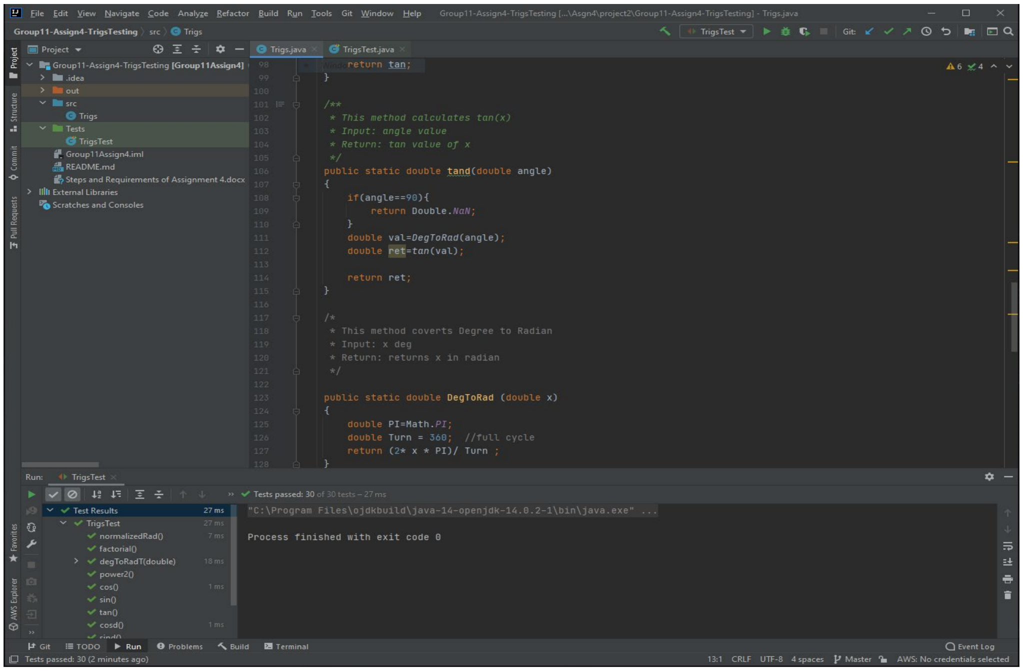


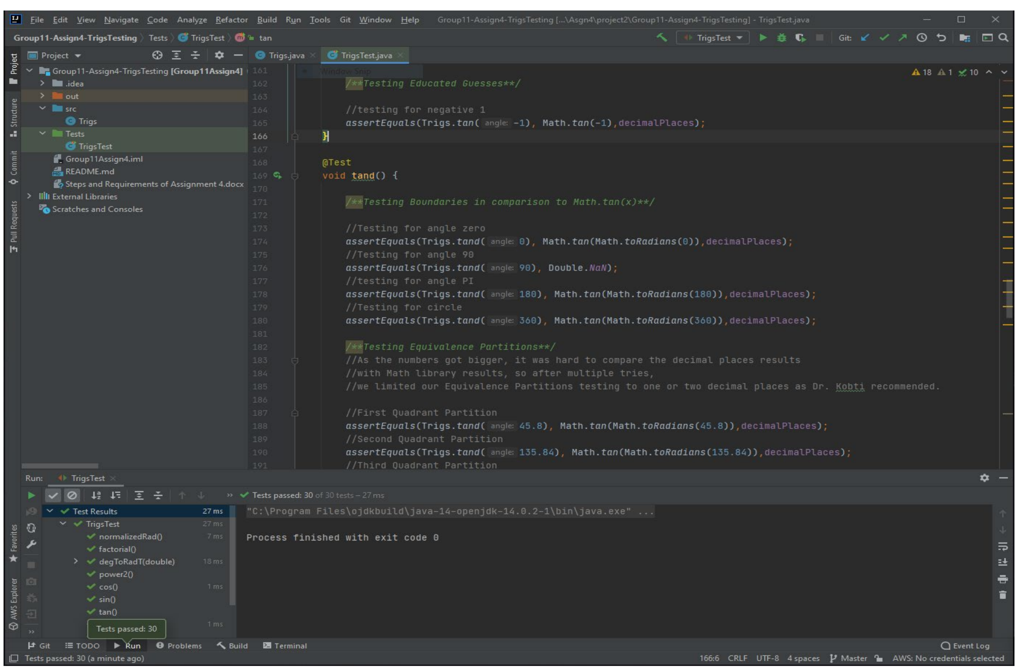
4. We also figured out the issue with our Equivalence Partitions testing in *tan()* function after discussing it with Dr. Kobti. The issue was related to the number of decimal places and how the tan() function in Math library is Java would give different results that our tan() function due to the number of decimal places. Therefore, as per Dr. Kobti's recommendation we limited the test cases to one or two decimal places.



Version 6.0 code and Testing Results

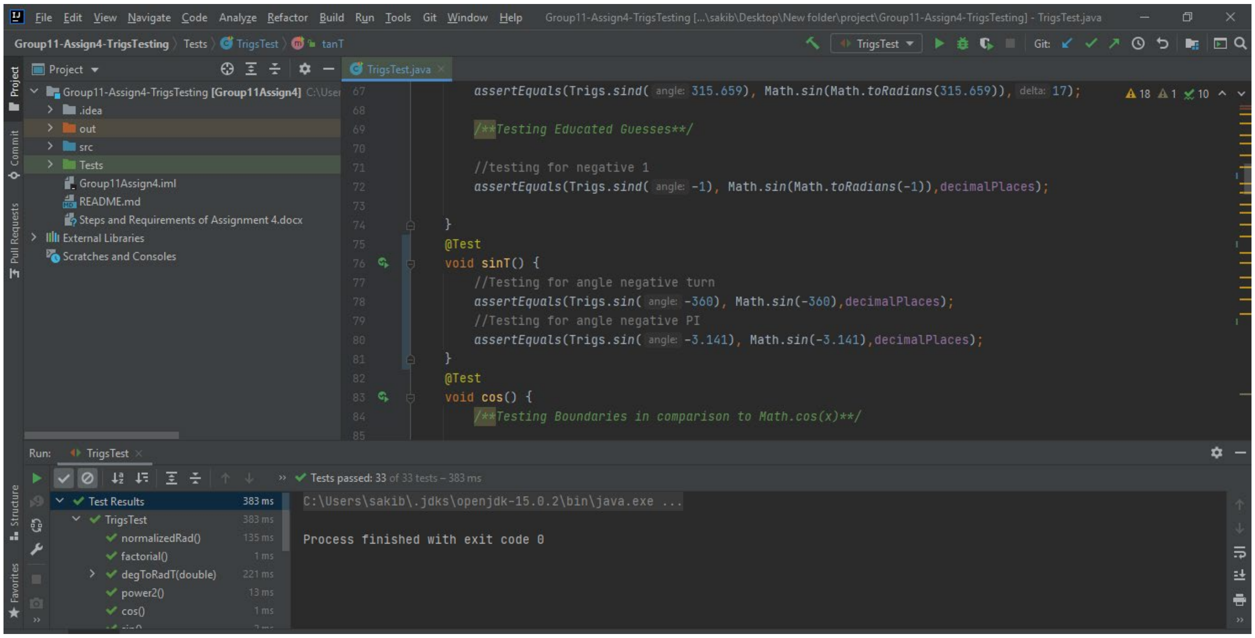
1. Changed tan and tand methods to accommodate 90 degrees and pi/2 respectively which we outputted to double.Nan as tan(90) is 1/0. Moreover, added some test methods to test them.



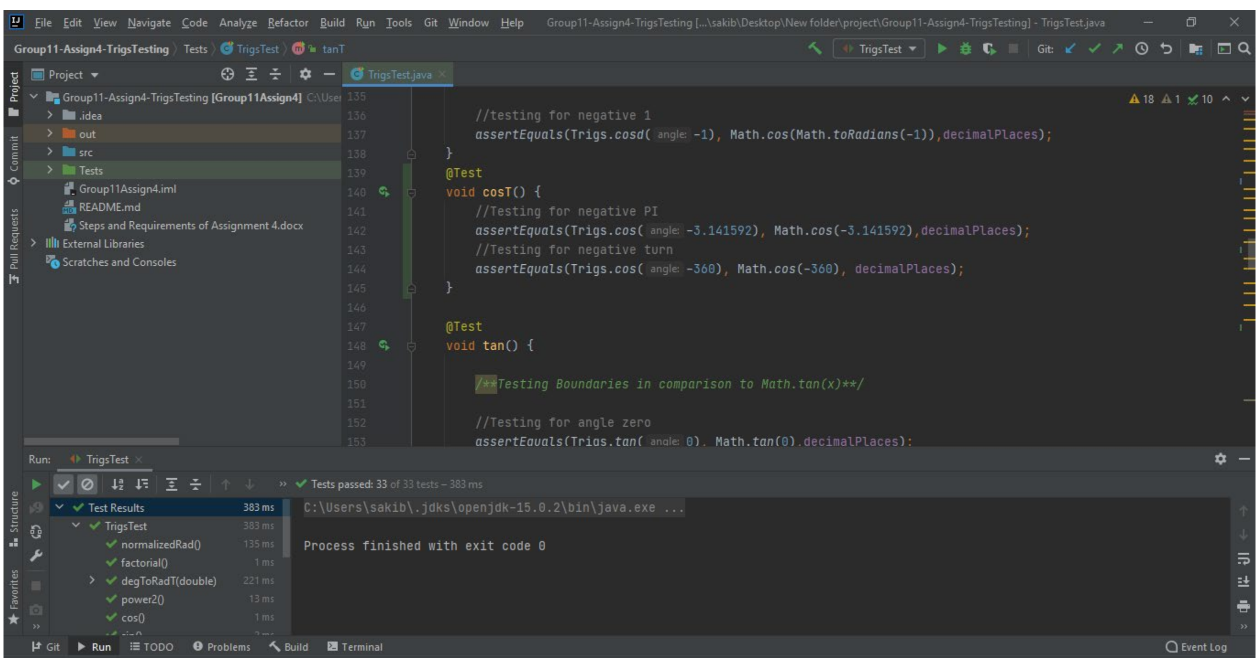


Version 7.0 (FINAL) code and Testing Results

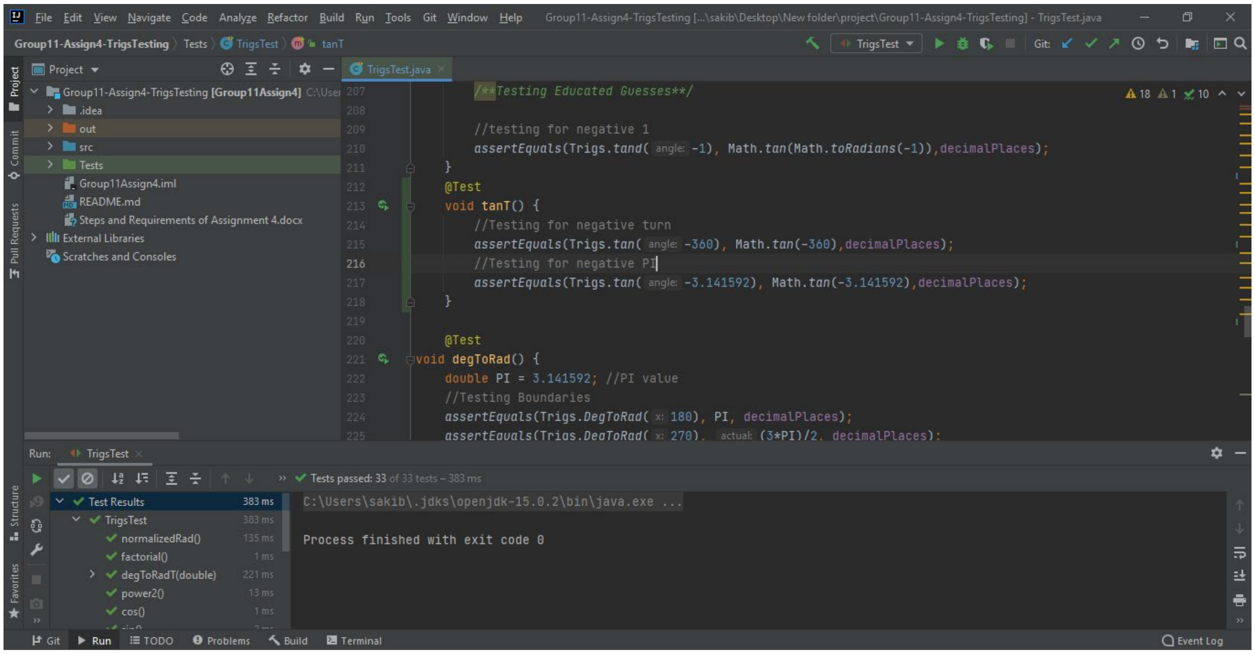
1. Here is sinT() test function, which has been created to test the value of sin function in case of negative degree value



2. Here is cosT() test function, which has been created to test the value of cos function in case of negative degree value



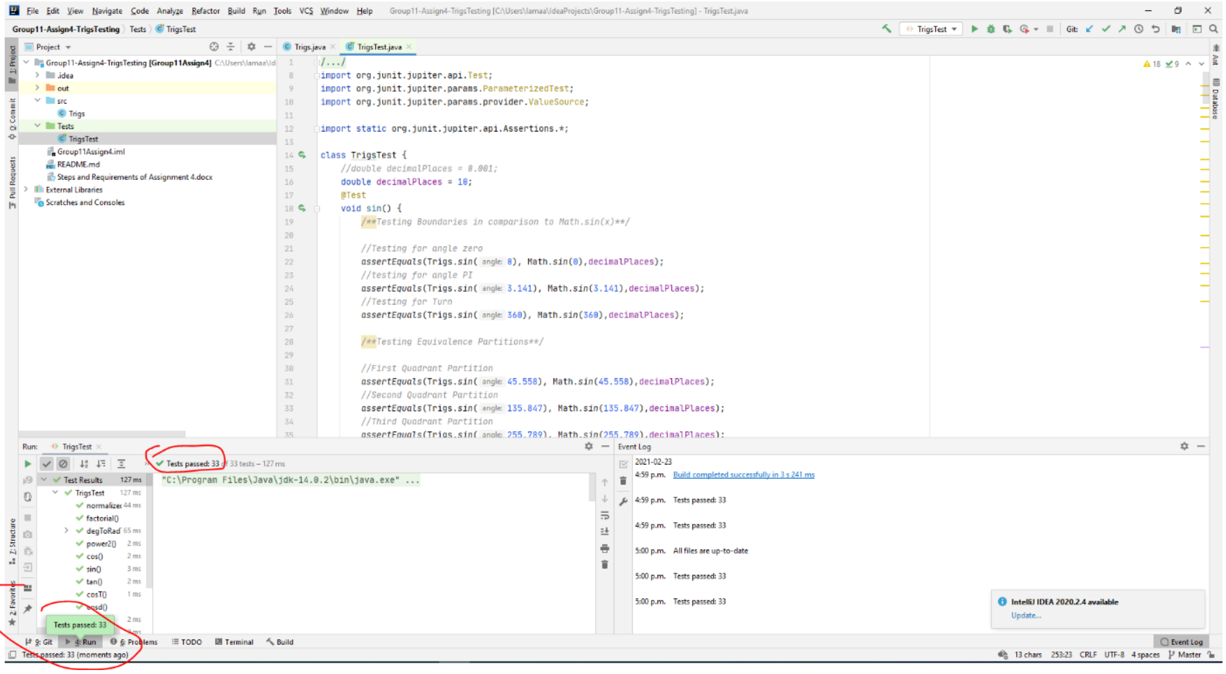
3. Here is tanT() test function, which has been created to test the value of cos function in case of negative degree value



4. We also added Equivalence Partitions testing to factorial(), the partitions were divided to four between 0 to 12. The reason we did not go above 12, is that the boundary testing reveled that the function will not work beyond 12!

```
@Test
void factorial() {
    //fail("Code is not implemented yet");

    //Testing Boundaries
    assertEquals( expected: 1, Trigs.factorial( n: 0));
    assertEquals( expected: 720, Trigs.factorial( n: 6));
    assertEquals( expected: 479001600, Trigs.factorial( n: 12));//the upper Boundary
    //assertEquals(6227020800, Trigs.factorial(13));//Test failed number is too large
    //assertEquals(87178291200, Trigs.factorial(14)); //Test failed number is too large

    //Testing Equivalence Partitions between 1 and 12
    assertEquals( expected: 1, Trigs.factorial( n: 1));
    assertEquals( expected: 6, Trigs.factorial( n: 3));
    assertEquals( expected: 5040, Trigs.factorial( n: 7));
    assertEquals( expected: 39916800, Trigs.factorial( n: 11));
```

5. Finally, all the tests have passed.



Note: Please refer to our GitHub repository or uploaded source code if you need more details.