Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.03.04 «Программная инженерия» –

Системное и прикладное программное обеспечение

Отчёт

По лабораторной работе №2

по предмету

Тестирование программного обеспечения

Вариант: 7531

Выполнили:

студенты 3 курса

Батманов Даниил Евгеньевич

Разинкин Александр Владимирович

Группа: P3307

Принял:

Гаврилов Антон Валерьевич

Отчёт принят «__»_____2025 г.

г. Санкт-Петербург, 2025

Оглавление

Задание:

Провести интеграционное тестирование программы, осуществляющей вычисление системы функций (в соответствии с вариантом).

$$\begin{cases} \left(\left(\left(\left(\frac{\tan(x)}{\sec(x)}\right)\cdot\cot(x)\right)-\sec(x)\right)-\csc(x)\right)\cdot\left(\frac{\csc(x)+\sin(x)}{\sec(x)}\right)\right) & \text{if} \quad x \le 0 \\ \left(\left(\left(\frac{(\log_2(x)+\ln(x))^3}{\log_3(x)}\right)^3\right)\cdot(\log_{10}(x)-\ln(x))\right) & \text{if} \quad x > 0 \end{cases}$$

x <= 0 : (((((tan(x) / sec(x)) * cot(x)) - sec(x)) - csc(x)) * ((csc(x) + sin(x)) / sec(x)))

x > 0 : (((((log_2(x) + ln(x)) ^ 3) / log_3(x)) ^ 3) * (log_10(x) - ln(x)))

Правила выполнения работы:

- Все составляющие систему функции (как тригонометрические, так и логарифмические) должны быть выражены через базовые (тригонометрическая зависит от варианта; логарифмическая - натуральный логарифм).
- Структура приложения, тестируемого в рамках лабораторной работы, должна выглядеть следующим образом (пример приведён для базовой тригонометрической функции sin(x)):
- Обе "базовые" функции (в примере выше - sin(x) и ln(x)) должны быть реализованы при помощи разложения в ряд с задаваемой погрешностью. Использовать тригонометрические / логарифмические преобразования для упрощения функций ЗАПРЕЩЕНО.
- Для КАЖДОГО модуля должны быть реализованы табличные заглушки. При этом, необходимо найти область допустимых значений функций, и, при необходимости, определить взаимозависимые точки в модулях.
- Разработанное приложение должно позволять выводить значения, выдаваемое любым модулем системы, в csv файл вида «X, Результаты модуля (X)», позволяющее произвольно менять шаг наращивания X. Разделитель в файле csv можно использовать произвольный.

Порядок выполнения работы:

- Разработать приложение, руководствуясь приведёнными выше правилами.

- С помощью JUNIT4 разработать тестовое покрытие системы функций, проведя анализ эквивалентности и учитывая особенности системы функций. Для анализа особенностей системы функций и составляющих ее частей можно использовать сайт https://www.wolframalpha.com/.
- Собрать приложение, состоящее из заглушек. Провести интеграцию приложения по 1 модулю, с обоснованием стратегии интеграции, проведением интеграционных тестов и контролем тестового покрытия системы функций.

Ход выполнения:

Разложение sin(x) в ряд Тейлора:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Разложение ln(x) в ряд Тейлора:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n}$$

Масштабирование y для для ln(x) и повышения точности значений при x большем 1 по модулю:

$$\ln(y) = \ln\left(y \cdot \frac{1}{a}\right) + \ln(a)$$

$$\ln(y) = \ln\left(\frac{y}{2^k}\right) + k \cdot \ln(2)$$

Листинг программы на GitHub: https://github.com/DecafMangoITMO/st_lab2

Исходный код:

```
package function.impl;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class CosFunction implements Function {

    private final Function sin;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        double sinX = sin.calculate(normalizedX, epsilon);
        double cosX = Math.sqrt(Math.abs(1 - sinX * sinX));
```

```
        if (normalizedX > Math.PI / 2 && normalizedX < Math.PI * 3 / 2) {
            cosX = -cosX;
        }

        return cosX;
    }
}
```

```
package function.impl;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class CotFunction implements Function {

    private final Function sin;
    private final Function cos;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        if (normalizedX == 0 || normalizedX == Math.PI)
            throw new ArithmeticException("cot is not defined for 0 and PI");

        return cos.calculate(normalizedX, epsilon) /
sin.calculate(normalizedX, epsilon);
    }
}
```

```
package function.impl;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class CscFunction implements Function {

    private final Function sin;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        if (normalizedX == 0 || normalizedX == Math.PI)
            throw new ArithmeticException("csc is not defined for 0 and PI");
```

```
        return 1 / sin.calculate(normalizedX, epsilon);
    }
}
```

```
package function.impl;

import function.Function;

public class Log2Function implements Function {

    private final Function ln;

    public Log2Function(Function ln) {
        this.ln = ln;
    }

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        if (x <= 0) {
            throw new ArithmeticException("y must be greater than 0");
        }

        return ln.calculate(x, epsilon) / ln.calculate(2, epsilon);
    }

}
```

```
package function.impl;

import function.Function;

public class Log3Function implements Function {

    private final Function ln;

    public Log3Function(Function ln) {
        this.ln = ln;
    }

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        if (x <= 0) {
            throw new ArithmeticException("y must be greater than 0");
        }

        return ln.calculate(x, epsilon) / ln.calculate(3, epsilon);
    }

}
```

```
package function.impl;

import function.Function;
```

```java
public class Log10Function implements Function {

    private final Function ln;

    public Log10Function(Function ln) {
        this.ln = ln;
    }

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        if (x <= 0) {
            throw new ArithmeticException("y must be greater than 0");
        }

        return ln.calculate(x, epsilon) / ln.calculate(10, epsilon);
    }

}
```

```java
package function.impl;

import function.Function;

public class LogEFunction implements Function {

    @Override
    public double calculate(double y, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        if (Double.isNaN(y) || Double.isInfinite(y)) {
            throw new IllegalArgumentException("no NaN or Infinity here!");
        }

        if (y <= 0) {
            throw new ArithmeticException("x must be greater than 0");
        }

        int k = 0;
        while (y >= 2) {
            y /= 2;
            k++;
        }
        while (y < 1) {
            y *= 2;
            k--;
        }

        double x = y - 1;
        double result = 0;
        double term = x;
        int n = 1;

        while (Math.abs(term) > epsilon) {
            result += term;
            n++;
            term = Math.pow(-1, n + 1) * Math.pow(x, n) / n;
        }
```

```
        return result + k * Math.log(2);
    }
}
```

```
package function.impl;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class SecFunction implements Function {

    private final Function cos;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        if (normalizedX == Math.PI / 2 || normalizedX == 3 * Math.PI / 2)
            throw new ArithmeticException("sec is not defined for PI/2 and
3*PI/2");

        return 1/cos.calculate(x, epsilon);
    }
}
```

```
package function.impl;

import function.Function;

public class SinFunction implements Function {

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        if (Double.isNaN(x) || Double.isInfinite(x)) {
            throw new IllegalArgumentException("no NaN or Infinity here!");
        }

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        double sum = 0.0;
        double term = normalizedX;
        int n = 1;

        while (Math.abs(term) >= epsilon) {
            sum += term;
            term = -term * normalizedX * normalizedX / ((2 * n) * (2 * n +
1));
            n++;
        }
```

```
        return sum;
    }

}
```

```
package function.impl;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class TanFunction implements Function {

    private final Function sin;
    private final Function cos;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        double normalizedX = (x % (2 * Math.PI) + 2 * Math.PI) % (2 *
Math.PI);

        if (normalizedX == Math.PI / 2 || normalizedX == Math.PI * 3 / 2)
            throw new ArithmeticException("tan is not defined for PI/2 and
3*PI/2");

        return sin.calculate(x, epsilon) / cos.calculate(x, epsilon);
    }
}
```

```
package function.task;

import function.Function;
import function.impl.*;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class LogarithmicFunction implements Function {

    private final Function ln;
    private final Function log_2;
    private final Function log_3;
    private final Function log_10;

    @Override
    public double calculate(double x, double epsilon) {
        if ((x <= 0) || (x == 1)) {
            throw new ArithmeticException("x must be greater than 0 and not
equal to 1");
        }

        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        return ((Math.pow(((Math.pow((log_2.calculate(x, epsilon) +
ln.calculate(x, epsilon)), 3)) / log_3.calculate(x, epsilon)), 3)) *
(log_10.calculate(x, epsilon) - ln.calculate(x, epsilon)));
    }
}
```

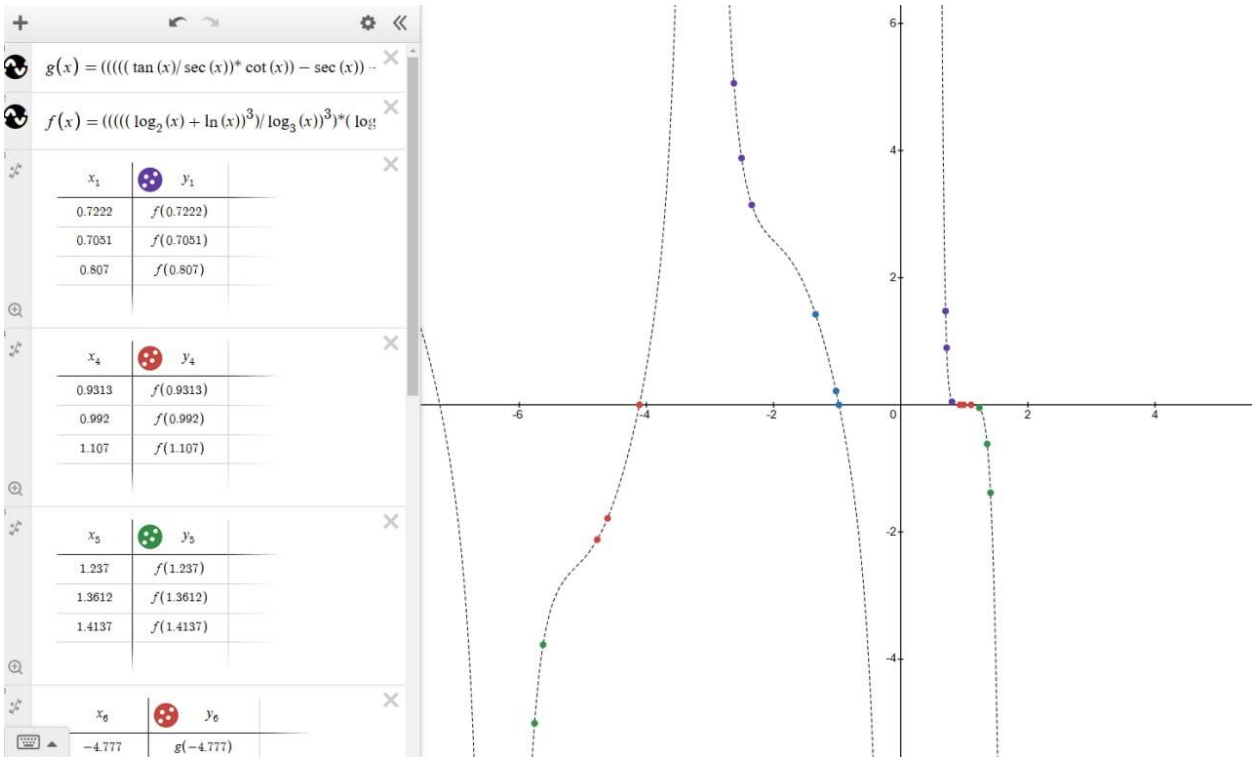
```java
package function.task;

import function.Function;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class TotalFunction implements Function {

    private final Function trigFunction;
    private final Function logFunction;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0)
            throw new IllegalArgumentException("epsilon must be positive");

        if (x <= 0)
            return trigFunction.calculate(x, epsilon);
        else
            return logFunction.calculate(x, epsilon);
    }
}
```

```java
package function.task;

import function.Function;
import function.impl.*;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class TrigonometricFunction implements Function {

    private final Function sin;
    private final Function cos;
    private final Function tan;
    private final Function cot;
    private final Function sec;
    private final Function csc;

    @Override
    public double calculate(double x, double epsilon) {
        if (epsilon <= 0d)
            throw new IllegalArgumentException("epsilon must be positive");

        return (((((tan.calculate(x, epsilon) / sec.calculate(x, epsilon)) *
cot.calculate(x, epsilon)) - sec.calculate(x, epsilon)) - csc.calculate(x,
epsilon)) * ((csc.calculate(x, epsilon) + sin.calculate(x, epsilon)) /
sec.calculate(x, epsilon)));
    }

}
```

```java
package function;

public interface Function {

    double calculate(double x, double epsilon);

}
```

```java
package util;

import function.Function;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;

public class CalculationRecorder {

    public static void record(Function function, double x, double epsilon,
double step, int stepCount, String filepath) {
        if (function == null)
            throw new NullPointerException("function is null");
        if (step <= 0)
            throw new IllegalArgumentException("step <= 0");
        if (stepCount <= 0)
            throw new IllegalArgumentException("stepCount <= 0");
        if (filepath == null)
            throw new NullPointerException("filepath is null");
        if (!filepath.endsWith(".csv")) {
            throw new IllegalArgumentException("File must have an extension:
.csv");
        }

        File file = new File(filepath);
        if (!file.exists()) {
            try {
                if (!file.createNewFile())
                    throw new RuntimeException();
            } catch (Exception e) {
                System.out.println("Failure during file creation: " +
filepath);
                return;
            }
        }

        try (BufferedWriter bw = new BufferedWriter(new FileWriter(file))) {
            bw.write("X:Result\n");

            double y;
            for (int i = 0; i < stepCount; i++) {
                try {
                    y = function.calculate(x, epsilon);
                    bw.write(String.format("%f:%f\n", x, y));
                } catch (ArithmeticException e) {
                    bw.write(String.format("%f:%s\n", x, e.getMessage()));
                } finally {
                    x += step;
                }
            }
        } catch (Exception e) {
            System.out.println("failure during writing file: " + filepath);
            return;
        }

        System.out.println("Record completed");
    }


}
```

Анализ эквивалентности:



Подробнее: https://www.desmos.com/calculator/cokkdji7go

```java
package function.task;

import function.Function;
import function.impl.*;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.ValueSource;

public class TotalFunctionUnitTest {

    private static final double EPSILON = Double.MIN_VALUE;
    private static final double DELTA = 10e-5d;

    private static Function sin;
    private static Function cos;
    private static Function tan;
    private static Function cot;
    private static Function sec;
    private static Function csc;

    private static Function trigFunction;

    private static Function ln;
    private static Function log2;
    private static Function log3;
```

```
    private static Function log10;

    private static Function logFunction;

    private static Function totalFunction;

    @BeforeAll
    public static void init() {
        sin = new SinFunction();
        cos = new CosFunction(sin);
        tan = new TanFunction(sin, cos);
        cot = new CotFunction(sin, cos);
        sec = new SecFunction(cos);
        csc = new CscFunction(sin);

        trigFunction = new TrigonometricFunction(sin, cos, tan, cot, sec,
csc);

        ln = new LogEFunction();
        log2 = new Log2Function(ln);
        log3 = new Log3Function(ln);
        log10 = new Log10Function(ln);

        logFunction = new LogarithmicFunction(ln, log2, log3, log10);

        totalFunction = new TotalFunction(trigFunction, logFunction);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "-5.8485:-6.51712",
            "-5.764:-5.01322",
            "-5.63:-3.77661"
    }, delimiter = ':')
    public void testFirstIntervalNonPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "-4.777:-2.12107",
            "-4.615:-1.78571",
            "-4.11362:0"
    }, delimiter = ':')
    public void testSecondIntervalNonPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "-2.626:5.06171",
            "-2.505:3.88417",
            "-2.345:3.14729"
    }, delimiter = ':')
    public void testThirdIntervalNonPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }
```

```
    @ParameterizedTest
    @CsvSource(value = {
            "-1.34:1.42589",
            "-1.018:0.21748",
            "-0.97203:0"
    }, delimiter = ':')
    public void testFourthIntervalNonPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @ValueSource(doubles = {
            -2 * Math.PI,
            -3 * Math.PI / 2,
            -Math.PI,
            -Math.PI / 2,
            0
    })
    public void testSpecialNonPositiveXPoints(double x) {
        Assertions.assertThrows(ArithmeticException.class, () -> {
totalFunction.calculate(x, EPSILON);});
    }

    @ParameterizedTest
    @ValueSource(doubles = {
            -5.8485d,
            -4.777d,
            -2.626d,
            -1.34d
    })
    public void testPeriodNonPositiveXPoints(double x) {
        double yCalculated1 = totalFunction.calculate(x, EPSILON);
        double yCalculated2 = totalFunction.calculate(x - 2 * Math.PI,
EPSILON);

        Assertions.assertEquals(yCalculated1, yCalculated2, DELTA);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "0.7051:1.47687",
            "0.7222:0.89819",
            "0.807:0.04841"
    }, delimiter = ':')
    public void testFirstIntervalPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "0.9313:0.00002",
            "0.992:0d",
            "1.107:-0.00026"
    }, delimiter = ':')
    public void testSecondIntervalPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
```

```
    }

    @ParameterizedTest
    @CsvSource(value = {
            "1.237:-0.04573",
            "1.3612:-0.61577",
            "1.4137:-1.38461"
    }, delimiter = ':')
    public void testThirdIntervalPositiveX(double x, double yExpected) {
        double yCalculated = totalFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @ValueSource(doubles = {
            0,
            1
    })
    public void testSpecialPositiveXPoints(double x) {
        Assertions.assertThrows(ArithmeticException.class, () -> {
totalFunction.calculate(x, EPSILON);});
    }

    @ParameterizedTest
    @ValueSource(doubles = {
            Double.NEGATIVE_INFINITY,
            Double.POSITIVE_INFINITY,
            Double.NaN
    })
    public void testIllegalXPoints(double x) {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
totalFunction.calculate(x, EPSILON);});
    }
}
```

```
package function.task;

import function.Function;
import function.impl.*;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.ValueSource;
import org.mockito.Mockito;

public class TrigonometricFunctionIntegrationTest {

    private static final double EPSILON = Double.MIN_VALUE;
    private static final int TO_ROUND = 5;
    private static final double DELTA = Math.pow(10, -TO_ROUND);

    private static Function sin;
    private static Function cos;
    private static Function tan;
    private static Function cot;
    private static Function sec;
    private static Function csc;

    private static Function trigFunction;
```

```
    @BeforeAll
    public static void init() {
        sin = Mockito.mock(SinFunction.class);
        cos = Mockito.mock(CosFunction.class);
        tan = Mockito.mock(TanFunction.class);
        cot = Mockito.mock(CotFunction.class);
        sec = Mockito.mock(SecFunction.class);
        csc = Mockito.mock(CscFunction.class);

        trigFunction = new TrigonometricFunction(sin, cos, tan, cot, sec,
csc);
    }

    @ParameterizedTest
    @CsvSource(value = {
            "-5.8485:-6.51712",
            "-4.777:-2.12107",
            "-2.505:3.88417",
            "-0.97203:0"
    }, delimiter = ':')
    public void testNormalNonPositivePoints(double x, double yExpected) {
        Mockito.when(sin.calculate(x, EPSILON)).thenReturn(Math.sin(x));
        Mockito.when(cos.calculate(x, EPSILON)).thenReturn(Math.cos(x));
        Mockito.when(tan.calculate(x, EPSILON)).thenReturn(Math.tan(x));
        Mockito.when(cot.calculate(x, EPSILON)).thenReturn(1 / Math.tan(x));
        Mockito.when(sec.calculate(x, EPSILON)).thenReturn(1 / Math.cos(x));
        Mockito.when(csc.calculate(x, EPSILON)).thenReturn(1 / Math.sin(x));

        double yCalculated = trigFunction.calculate(x, EPSILON);

        Assertions.assertEquals(yExpected, yCalculated, DELTA);
    }

    @ParameterizedTest
    @ValueSource(doubles = {
            -5.8485d,
            -4.777d,
            -2.626d,
            -1.34d
    })
    public void testPeriod(double x) {
        Mockito.when(sin.calculate(x, EPSILON)).thenReturn(Math.sin(x));
        Mockito.when(sin.calculate(x - 2 * Math.PI,
EPSILON)).thenReturn(Math.sin(x));

        Mockito.when(cos.calculate(x, EPSILON)).thenReturn(Math.cos(x));
        Mockito.when(cos.calculate(x - 2 * Math.PI,
EPSILON)).thenReturn(Math.cos(x));

        Mockito.when(tan.calculate(x, EPSILON)).thenReturn(Math.tan(x));
        Mockito.when(tan.calculate(x - 2 * Math.PI,
EPSILON)).thenReturn(Math.tan(x));

        Mockito.when(cot.calculate(x, EPSILON)).thenReturn(1 / Math.tan(x));
        Mockito.when(cot.calculate(x - 2 * Math.PI, EPSILON)).thenReturn(1 /
Math.tan(x));

        Mockito.when(sec.calculate(x, EPSILON)).thenReturn(1 / Math.cos(x));
        Mockito.when(sec.calculate(x - 2 * Math.PI, EPSILON)).thenReturn(1 /
Math.cos(x));

        Mockito.when(csc.calculate(x, EPSILON)).thenReturn(1 / Math.sin(x));
        Mockito.when(csc.calculate(x - 2 * Math.PI, EPSILON)).thenReturn(1 /
```

```
Math.sin(x));

        double yCalculated1 = trigFunction.calculate(x, EPSILON);
        double yCalculated2 = trigFunction.calculate(x - 2 * Math.PI,
EPSILON);

        Assertions.assertEquals(yCalculated1, yCalculated2, EPSILON);
    }

}
```

Заключение

В ходе выполнения данной лабораторной работы нам удалось разработать приложение, руководствуясь указанными в задании правилами. С помощью JUNIT5 разработали тестовое покрытие системы функций, проведя анализ эквивалентности и учитывая особенности системы функций. Для анализа особенностей использовали Десмос. Провели интеграцию приложения по 1 модулю с использованием Mockita.