# Postfix Calculator

## Testing Document

EECS 2311 Group 2 v1.0

***Team 2***

Chauhan, Yash

Jin, Hui

Khan, Akbar

Martintsov, Vladimir

Nguyen, Andrew

# TABLE OF CONTENTS

## 1. Introduction

Testing was a critical part in the process of improving the code for the project as well as fixing a number of bugs that were apparent within the code. The test cases were split up into four classes - **TestCalcModel, TestCalcController, TestCalcView,** and **TestButtonPress**. Each class contains a number of test with inputs that cover a wide variety of test cases sufficient enough to represent all inputs.

## 2. Testing the Model

The Model was tested using the following test cases. Each test case assesses the corresponding method in the Model class and includes all sufficient test coverages. The methods need to yield an appropriate output in order for them to pass the tests. In this case, all the tests passed.

### 2.1 testNumericButton() and testEnter()

The first method to be tested was **numericButton()** (used to enter single numbers or decimal points to the value). One key aspect to testing this method was to make sure every parameter was covered. This was done by calling the method through numbers 0 to 9 and decimal point, and using **getInputValue()** to test if the resulting value is correct. Another aspect was to make sure any extra zeros or decimals were removed in the input. Removing extra zeroes and decimals prevents the number from appearing in an improper format.

testNumericButton()
**Input:** numericButton() used with 0123456.789.
**Expected output:** Value equals 123456.789

For the method **enter()**, its basic function needed to be tested as well as using it on different types of inputs. The basic function was tested by using **enter()** on an input value and checking if the resulting number in history matches by using **getHistoryValue()**. Then, **enter()** was tested on multiple input values in order to see if the history would display each value separated by a comma. Within these multiple input values, a negative number and a decimal number was used to make sure the history would show both these types of numbers. This test also covers cases where the first or last character in the input value is a decimal. When displayed onto the history, zeros should be added to display the number in proper format.

testEnter()
**Input:** 1 [enter] 2 [change sign] [enter] 3 . 5 [enter] . 5 [enter] 6 . [enter]
**Expected output:** History displays 1, -2, 3.5, 0.5, 6.0

### 2.2 testSum() and testSubtract()

Testing the methods **sum()** and **subtract()** required that the resulting sum or difference equaled the expected value and that the history displayed the equation in infix notation. This was done by adding the numbers 1 and -2 and subtracting the numbers 5.5 and 2.5. These tests are important because getting a correct value is crucial in calculators and the infix notation is part of the requirements. Testing two numbers for each has sufficient coverage for all cases.

### testSum()

**Input:** 1 -2 +

**Expected output:** History displays 1 + -2 =, Value is -1

### testSubtract()

**Input:** 5.5 2.5 -

**Expected output:** History displays 5.5 – 2.5 =, Value is 3**,**

## 2.3 testMultiply() and testDivide()

Testing **multiply()** consisted of getting a correct output value and displaying the history in infix notation with correct precedence. Two numbers are multiplied together and the product is compared to the expected value. If this passes, it confirms that **multiply()** will give the correct values for all inputs. Then, an equation involving addition, subtraction, and multiplication is input and is tested for infix notation as well as if multiplication has precedence over addition and subtraction.

### testMultiply()

**Input:** 5 -5 ×

**Expected output:** History displays 5 × -5 =, Value equals -25

**Input:** 2 3 + 2 1 - ×

**Expected output:** History displays (2 + 3) × (2 – 1) =, Value equals 5

The test for **divide()** involved three cases. The first was the basic test of getting the correct value. As stated before, the purpose is to make sure **divide()** gives the correct value for all inputs and testing any two inputs is sufficient for this. The second case tests for precedence over **multiply()**. An equation involving **multiply()** and **divide()** is input and proper infix notation and precedence is checked by looking at the brackets. This is sufficient enough for precedence as it follows from the previous **testMultiply()** that if **divide()** has precedence over **multiply()**, it must have precedence over **sum()** and **subtract()**. The final case involves dividing a number by zero, which the expected output is a **MATH ERROR**.

### testDivide()

**Input:** 7 2 ÷

**Expected output:** History displays 7 ÷ 2 =, Value equals 3.5

**Input:** 1 2 × 1 2 × ÷

**Expected output:** History displays (1 × 2) ÷ (1 × 2) =, Value equals 1

**Input:** 7 0 ÷

**Expected output:** Value equals MATH ERROR

## 2.4 testPi()

The method **pi()** was tested using two cases. The first compared the value the input to the expected value. After **pi()** is called, **getInputValue()** and **getHistoryValue()** are verified. This ensures that **pi()** gives the correct value for calculations and displays the correct symbol in the history. The second case tests a feature that lies within **pi()**. A number is input in the value and then **pi()** is used resulting in a multiplication of the two values. This feature creates easier input of equations involving **pi().**

testPi()

**Input:** π

**Expected output:** History displays π, Value equals 3.1415926536

**Input:** 2 π

**Expected output:** History displays 2 × π =, Value equals 6.2831853072

## 2.5 testSine() and testCosine()

For both of these test cases, **sine()** or **cosine()** were tested to see if they gave the correct values. This was done by first inputting a value into **inputValue**, then calling **sine()** or **cosine()**, then verifying if the resulting value is correct using **getInputValue()**. Just one **inputValue** is sufficient as one pass proves that all other values will give the correct output. This test also validates that the operator is shown in prefix notation with its operand. The history is checked and the test passes if sin or cos appears before its operand i.e. sin goes before the (x) in sin(x) where x is any operand.

testSine()

**Input:** π 2 ÷ sin

**Expected output:** History displays sin(π ÷ 2) =, Value displays 1

testCosine()

**Input:** π cos

**Expected output:** History displays cos(π) =, Value displays -1

## 2.6 testFactorial()

The test case **factorial()** consists of three components. The first part makes certain that the method returns the correct value. After **factorial()** is used, the value in the field **inputValue** is verified to be correct. The next two parts deal with cases of a non-integer input and a negative input. For both of these cases, the **inputValue** displays **MATH ERROR**. Although it may be possible to implement an algorithm to find these values, it is not required so a **MATH ERROR** will suffice.

testFactorial()

**Input:** 5!

**Expected output:** History displays 5! =, Value equals 120

**Input:** 5.8!

**Expected output:** Value equals MATH ERROR

**Input:** -2!

**Expected output:** Value equals MATH ERROR

## 2.7 testChangeSign()

The test for **changeSign()** was split into three parts. The first part verified that the operation changed the sign of the value. This was done by inspecting the value in **getInputValue()** after **changeSign()** was used on a value. The second part tested the capability of continuing an input after **changeSign()** was used. When the user presses **changeSign()**, the user should not be forced to enter the current value. The third part tested to see if an equation involving **changeSign()** gives the correct value. This ensures that the negative value is treated as such in calculations.

testChangeSign()
**Input:** 6 [change sign] [enter]
**Output:** Value equals -6

**Input:** 6 [change sign] . 5
**Output:** Value equals -6.5

**Input:** -9 2 + -3 4 - ×
**Output:** History displays (-9 + 2) × (-3 - 4) =, Value equals 49

### 2.8 testClear()

This test is pretty straightforward. An input value is entered and **clear()** is called. The test checks to see if the **inputValue** and **historyValue** revert back to the initial conditions.

testClear()
**Input:** 8 [enter] [clear]
**Expected output:** History displays Start New Calculation, Value equals 0

### 2.9 testUndo()

The testing of **undo()** comprises of four segments. The first part tests to verify that using **undo()** while in the middle of inputs deletes the most recent input until the **inputValue** is empty. The number 5 will be entered followed by the input of 6.3. When **undo()** is called three times, the **inputValue** should be empty and the **historyValue** untouched. The second part validates that using **undo()** after a number is entered deletes the last value entered in the **historyValue**. The third segment tests that **undo()** works with all binary operators and the fourth part tests that **undo()** works with all unary operators.

testUndo()
**Input:** 5 [enter] 6 . 3 [undo] [undo] [undo]
**Expected output:** History displays 5, Value displays nothing

**Input:** 5 [enter] 6 [enter] 7 [enter] 8 [enter] [undo] [undo]
**Expected output:** History displays 5, 6

**Input:** 1 2 3 4 5 + - × ÷ [undo] [undo] [undo] [undo]
**Expected output:** History displays 1, 2, 3, 4, 5

**Input:** 5! sin cos [undo] [undo] [undo]
**Expected output:** Value equals 5

## 3. Testing the Controller

The tests for the Controller class is limited because the majority of its methods use methods straight from the Model class. Because of this, all test cases within the Model can be applied to the Controller.

The only method within Controller worth testing is **updateView()**, which as its name suggests, updates the fields **display** and **historyDisplay** within the View with the matching **inputValue** and **historyValue**.

### 3.1 testUpdateView()

This test ensures that the fields **display** and **historyDisplay** from the view are set to its correct **inputValue** and **historyValue** from the model. This is done by using the controller to input a sequence of equations and comparing the values in **getDisplayText()** and **getHistoryText()** to its expected value. The test uses every operator within its equation to make sure they all update properly.

testUpdateView()

**Input:** 1 -2 + 3 4 - × π 2 ÷ sin π cos × ÷ 0! ×

**Expected Output:** History displays ((1 + -2) × (3 - 4)) ÷ (sin(π ÷ 2) × cos(π)) × 0! =,
display shows -1

## 4. Testing the View

The view was testing by splitting it up into two categories. The first category is about the methods within the view and the design. The second part simulates button presses and tests to see if they perform the correct operation.

### 4.1 Testing Methods Within View and Design

testSetDisplayText:

| No | Action | Expected Result | Actual Result |
|---|---|---|---|
| 1.1 | Set displaytext as null | Display text field will be null | true |
| 1.2 | Set displaytext as "abc" | Display text field will show abc | true |
| 1.3 | Set displaytext as null | Display text field will be null | true |

Though setting it null in the beginning, after that set it with some value and in the end change it as null to test whether this function can work properly or not.

testSetHistoryText

| No | Action | Expected Result | Actual Result |
|---|---|---|---|
| 2.1 | Set historytext as null | History text field will be null | true |
| 2.2 | Set historytext as "abc" | History text field will show abc | true |
| 2.3 | Set historytext as null | History text field will be null | true |

Same as displaytext, though setting it null in the beginning, after that set it with some value and in the end change it as null to test whether this function can work properly or not.

testBetterViewDesign

| No | Action | Expected Result | Actual Result |
|---|---|---|---|
| 3.1 | Check panel2's background color | Color will be black | true |
| 3.2 | Check historyDisplay's background color | Color will be black | true |
| 3.3 | Check historyDisplay's foreground color | Color will be white | true |
| 3.4 | Check display's background color | Color will be black | true |
| 3.5 | Check display's foreground color | Color will be white | true |
| 3.6 | Check display's border | Border will be null | true |
| 3.7 | Check historyDisplay's border | Border will be null | true |
| 3.8 | Check mainpanel's background color | Color will be (24,29,33) | true |
| 3.9 | Check panel's background color | Color will be (24,29,33) | true |
| 3.10 | Check numbers's background color | Color will be (24,29,33) | true |
| 3.11 | Check operators's background color | Color will be (24,29,33) | true |
| 3.12 | Check buttons's background color | Color will be (24,29,33) | true |

This testcases test that whether the betterview design will generate the view we want.

testButtonColor

| No | Action | Expected Result | Actual Result |
|---|---|---|---|
| 4.1 | Check benter's background color | Color will be (218.100.2) | true |
| 4.2 | Check benter's foreground color | Color will be (250.251.255) | true |
| 4.3 | Check benter is focusable | Focusable will be false | true |
| 4.4 | Check benter's border | Benter's border will be null | true |

This testcases test that whether the ButtonColor will generate the view we want.

## 4.2 Simulating Button Presses

The testing of button presses uses the method **doClick()** to simulate button presses within the view and **getDisplayText()** and **getHistoryText()** to verify the output. Each test case passed and they will be explained in the following.

The button enter was tested first, and a single press had the expected output of **getDisplayText()** to be 0 and **getHistoryText()** to be 0.

The buttons 0 through 9 had expected outputs of **getDisplayText()** and **getHistoryText()** corresponding to the respective value input.

The button dot was tested by expecting a decimal within its input.

The button clear was tested by expecting the fields to be reset to its initial values.

The buttons add, subtract, divide, and multiply were tested by using the method on a two values and expecting a certain sum, difference, quotient, or product.

The button pi was tested by making sure the button gave the correct value of 3.1415926536.

The buttons sin, cos, factorial, and change sign were testing by using the method on an input value and expecting the resultant output value.

The button undo was testing by simulating an undo press after a number has been pressed and expecting that number to be deleted from the input value.