# 3. Development

## 3.1 Success Criteria and Testing

### 3.1.0 Version 0.1.0

#### V0.1.0 – Success Criteria

In Version 0.1, I attempted to complete the success criteria:

* “Functionality for the four basic operators, +, -, \*, ÷.”
* “Follow the rules of BODMASS.”
* “Have functionality for using square roots and powers.”

This version covers the success criteria that make up the most basic functionality that a calculator.

For Version 0.1.0 to be successful, the calculator should be able to calculate any basic equation that includes the four basic operators, “+, -, \*, ÷” and powers and roots.

Version 0.1.0 was started by creating an XML layout file which contained all the required parts of the calculator, such as the number input buttons, the calculator display, the operator buttons, a delete button, etc.

The layout was created using a *Constraint Layout*, instead of a *Relative* or *Linear layout*. A *Constraint Layout* is where each view has constraints that act as connections to other views, the parent layout or an invisible guideline.

A *Relative Layout* is where the views are positioned relative to each other, while a *Linear Layout* is where the views are organised into a single horizontal or vertical row.

I have chosen to use a *Constraint Layout* because although it is initially more complicated to understand and implement, you can build much more flexible layouts with it. This will be especially helpful in this project as the *Constraint Layout* will mean the layout will scale and change appropriately depending on different size devices.

In the *MainActvitity* class, variables were linked to their corresponding XML element. For example, the buttons were linked to Button variables and the calculator display was linked to a *TextView* variable.

The buttons variables were set an *OnClickListener* so they called the *onClick* method when tapped.

A switch case statement was used where cases are dependent on the ID of the clicked button. Each button has a unique ID that’s defined in the layout’s XML file.

Each button case adds the input of the button into the expression so it can later be used in the *ShuntingYard* class.

There are also special cases, such as the “AC” button which wipes all data, and the “DEL” button which removes the last character of the expression.

When the equals button is pressed (“=”), a validation is first run to ensure that the expression string is not empty.

This is done to prevent an error occurring when the string is passed to the *ShuntingYard* *infixToPostfix* method.

If the validation check is failed, the stack trace is logged and an error message is shown to the user.

If the validation check is successful, the answer is passed to the *ShuntingYard* method called *infixToPostfix*. This method converts the user’s expression into Reverse Polish Notation.

In this method, a string called *ops* is used to define each operator’s precedence. The precedence is found by the index of the operator divided by 2.

A for-each loop is used to iterate through the user’s expression by each character. If the current character is empty, a *continue* keyword is used to terminate the processing of the current iteration and continue with the for-each loop.

Variables are made to denote the current character and the index of the character in the operator string. If the character is not an operator, it will not exist in the string and therefore getting its index will return “-1”.

If the index is returned as “-1”, and the stack is currently empty, the value of the index is pushed onto the stack.

If the stack is not empty, a while loop is used so that while the stack is not empty, the value of the precedence before the current character is retrieved by using “*stack.peek() / 2”*. The *peek* method is used to get the value of the top of the stack. This will be the index of the previous character in the *ops* string and dividing it by two returns the precedence value. This is stored in a variable called *prec2*.

The precedence of the current character is calculated by dividing the index of the character in the *ops* string by 2. This is stored in a variable called *prec1.*

If the previous operator’s precedence is greater than the second one or they are equal (if the current operator is not a power), the index of the previous operator is popped off the stack and its corresponding operator character is appended to the postfix string.

When the while loop ends (i.e. the expression is empty), the current index is pushed onto the stack.

If the index does not equal “-1”, and the character equals an opening bracket “(“, the value of “-2” is pushed onto the stack to represent the start of the bracket.

If the current character is a closing bracket instead, a while loop is used to add characters to the postfix string. The condition for the while loop is while the previous element in the stack does not equal “-2” (the integer representing an opening bracket).

This means that the part of the expression inside the bracket will take precedence over everything else and therefore obey the rules of BODMAS properly.

If none of these conditions (index != -1, character = “(“ or “)”) are met i.e. the current character is just a digit, the digit is appended to the postfix string.

Finally, the *infixToPostfix* method returns the string it has made.

In the *MainActivity* class, the postfix can be found by calling the *infixToPostfix* method on the user’s input expression string.

The *evaluateRPN* method can then be used on the postfix string to calculate the answer to the user’s expression.

The *evaluateRPN* method works by first creating a stack with a data type of double called “tokens” and then running a for-each loop where the loop iterates over an array created from the postfix that has been split up on the whitespace.

Each iteration, the method uses a HashMap made using a for-each loop which goes over the *enum* S*ign* which contains all the operators and puts each operator into the HashMap.

The *evaluateRPN* can then use the *find* method on the *HashMap* with the parameter of the current token to see if it is an operator. If the token is not an operator, it will return “null”.

If the token is an operator, the operator is applied to the last two elements of the stack using the *calcSign* method and the *enum* data type (which contains all the operators and returns the result for each one).

The *calcSign* method returns the stack after the tokens have been popped off, the result of the two with the operator has been calculated and then the result has been pushed back on.

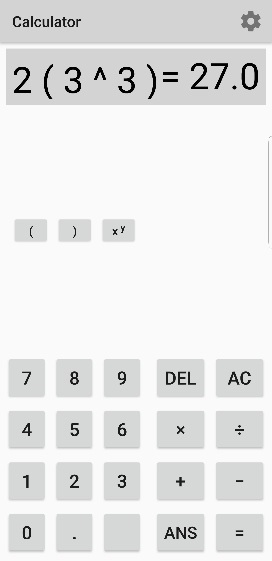
If the current token is not an operator, it must be a number and therefore the token is converted from a string to a double and then pushed onto the stack.

At the end of the iteration, the only thing left in the stack will be a single double data type which is the result of the user’s input expression.

This value is returned where it can be displayed to the user using a *TextView*.

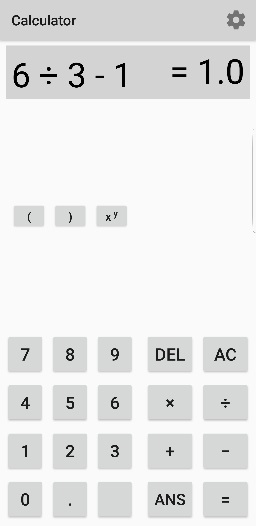
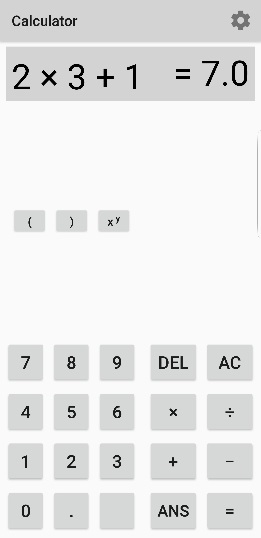
The source code that handles this is shown in Appendix 5.1.1 and 5.1.2.

Version 0.1.0:



#### V0.1.0 - Testing

The first of these success criteria is the four basic operators:

These screenshots show how the calculator properly handles the operators “+, -, x, ÷”. The input takes “2\*3+1” and outputs “7”. This is correct and therefore shows that the calculator is handling multiplication and addition correctly.

The calculator also takes “6÷3-1” and outputs “1”. This is the correct result and there for shows that the calculator is performing division and subtraction correctly.

These screenshots also show that the calculator is properly following BODMAS when completing the calculations. This is because when inputted with “2\*3+1”, the calculator first does “2\*3” and then “+1”.

This is correct because multiplication takes precedence over addition in BODMAS. Therefore, when using the basic operators BODMAS is correctly used. This means that the first and second success criteria aimed to be completed in Version 0.1 were successful.

The last success criteria in Version 0.1 is for the calculator to “have functionality for using square roots and powers”.

These screenshots show how Version 0.1 of the calculator handles powers and how they interact with BODMAS.

In the first screenshot, the calculator interprets the equation as “2x33”. Therefore, it follows BODMAS and calculates “33” to give “27” because powers have precedence over multiplication. It then multiples “27” by 2 to result in “54”. This is an example of the calculator working correctly.

However, in the second screenshot the equation “2(33)” is inputted. This is using a different way of expressing the exact same equation as the first screenshot.

The calculator should be interpreting this expression as “2\*(33)”. However, the calculator does not recognise a number in front of brackets as shorthand for “2\*(33). Therefore, the base and power does not get multiplied by two and the output is incorrect.

Another problem with version 0.1 is that the functionality for roots was not completed in the first iteration of the iterative development process. However, this will be resolved in the next iteration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test basic operators and how they work with BODMAS. | 2\*3+1 | 7  =(2\*3)+1 | 7 | N/A |
| Test basic operators and how they work with BODMAS. | 6/3-1 | 1  =(6/3)-1 | 1 | N/A |
| Test that powers are calculated correctly. | 33 | 27 | 27 | N/A |
| Test BODMAS precedence in an expression with different operators but no brackets to simplify it. | 2\*4 -22 | 4  =(2\*4)-(22) | 4 | N/A |
| Test how brackets are handled when used as shorthand for multiplication. | 2(33) | 54  =2\*(33) | 27  =(33) | Calculator does not recognise a number next to a bracket as shorthand for multiplication and instead ignores the number.  Actions:  When interpreting the user’s input expressions, insert multiplication operators where needed before evaluating the expression. |
| Test that square root function works correctly. | sqrt(4) | 2 | - | Root functions have not yet been created.  Actions:  Create a function that takes two inputs, the base and the root. It will root the base by the root variable and output the result to the user. |
| Test “DEL” button to make sure it deletes one operator/number at a time | Tap “DEL” on any expression.  E.g.:  2-2+ | 2-2 | 2-2+ | When the user inputs operator characters, the application automatically inserts “ “ characters either side of the user’s input. This is done so the Reverse Polish Notation is created correctly.  The “DEL” input does not include the whitespace that was inserted by the application. To delete this as well, the user needs to input “DEL” multiple times.  Actions:  Currently, the “DEL” button works by removing the last character of the expression string.  To fix this, first remove all whitespace in the string.  Then remove the last character.  Finally, the whitespace can be inserted back into the string. |
| Test “AC” button to ensure that it deletes the entire expression and wipes anything in the calculator’s memory that should be non-volatile. | Click “AC” on expression:  2+3 | Wipes current display and everything stored in memory. | Wipes current display and everything stored in memory. | N/A |

### 3.1.1 Version 0.1.1

#### V0.1.1– Success Criteria

V0.1.1 involves completing the success criteria for:

* “Have functionality for using … roots”
* Fix various bugs introduced in version 0.1.0 such as the *DEL* function behaving improperly and correcting how using brackets as shorthand for multiplication is handled.

This version covers the second iteration of the implementation of the first 3 basic success criteria.

This iteration is made up of fixing bugs that were introduced in the first iteration. This includes the errors where the “DEL” button was not working correctly because it only deleted the last character of the expression string variable.

This bug was fixed by removing all whitespace from the expression, then deleting the last character from the string. Then whitespace was inserted back into the string. This was done using the string function *.replaceAll* and the regex expression “*(‘.(?=.)’, ‘$0 ’)”.*

This regex expression replaces all characters in the string with itself and space except the last character.

In this iteration, the bug where using brackets as shorthand for multiplication was not interpreted correctly was also fixed.

This was fixed by creating a function that was called when an opening bracket was entered by the user.

This function checks if the character before it is a digit, e.g. “4(2)” or a closing bracket, e.g. “(2)(2)”. If the check is positive, a *StringBuilder* is made and a multiplication sign and the appropriate whitespace is inserted into the string. This is done using the *StringBuilder* method *replace*.

This iteration also included the implementation of a root function. This was created by adding a button to the main XML file and linking it to the *MainActivity* class so it would call *onClick* when the button is clicked. When the root button is clicked, it inserts the radical sign (√) into the expression string and updates the calculator display to show this.

The radical sign was then included in the *ShuntingYard* class to assign its precedence in BODMAS when creating the Reverse Polish Notation. This ensures that the RPN is formed correctly so BODMAS is followed.

Extra parameters were also created to calculate the user’s inputted roots. The roots are calculated by using Java’s Math library: “*Math.pow(base, 1.0 / root)”*. This is because mathematically √4 is equal to 41/2 through indices laws.

Version 0.1.1 also included the addition of shortcut root and power buttons.

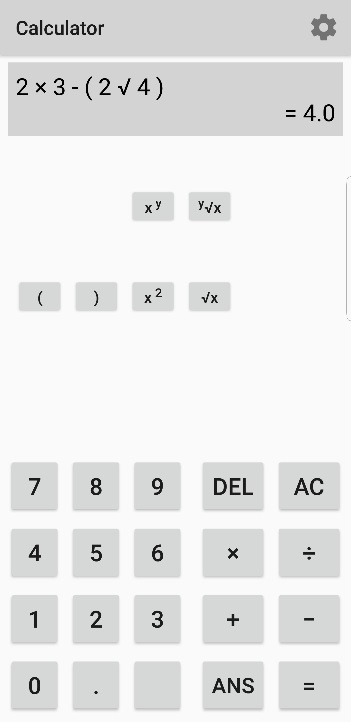
The shortcut power button automatically adds 2 to the user’s base so the user does not need to enter their power as well.

Squaring numbers is the one of most common power used by students and therefore having a shortcut button makes the process of using the calculator much more efficient. This is important as it leaves the user to focus and spend their time and focus on other things.

The square root shortcut button acts similarly to this, it automatically adds a square root because square roots are the most common roots used.

These changes were made to Appendixes 5.1.1 and 5.1.2.

Version 0.1.1:



Another notable change in this version is that the size of the text in the calculator display was decreased to ensure that the user’s entire expression and answer will be displayed clearly to the user.

#### V0.1.1 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test “DEL” button to make sure it deletes one operator/number at a time | Tap “DEL” on any expression.  E.g.:  2-2+ | 2-2 | 2-2 | N/A |
| Test how brackets are handled when used as shorthand for multiplication. | Input an expression where there’s a number adjacent to an opening bracket, e.g.:  3(33) | 3(33)  = 54 | 3\*(33)  = 54 | N/A |
| Test that x/y root function works correctly. | Input any y√x expression to test that the root and base are working correctly. | 3√8  = 2 | 3√8  = 2 | N/A |
| Test square root shortcut button works correctly. | Use the shortcut button to input any 2√x expression. | 2√4  = 4 | 2√4  = 4 | N/A |
| Test square power shortcut button works correctly. | Use the shortcut button to input any x2 expression. | 23  = 8 | 23  = 8 | N/A |

### 3.1.3 Version 0.2.0

#### V0.2.0 – Success Criteria

Version 0.2 is intended to complete the success criteria:

* “Cursor controls that will move the cursor around the equation.”

This means that for Version 0.2 to be considered successful, the app must have functionality that allows the user to traverse through the equation using cursor buttons. This will enable the user to edit their expression without either clearing it using *AC* or deleting part of it.

This was completed by introducing a variable that kept track of the user position in the expression’s string.

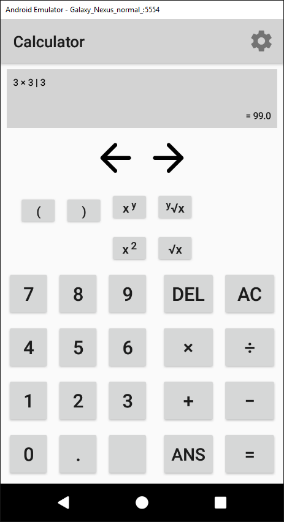
This variable was incremented every time the user added to the expression and was decreased whenever the user deleted something from the expression.

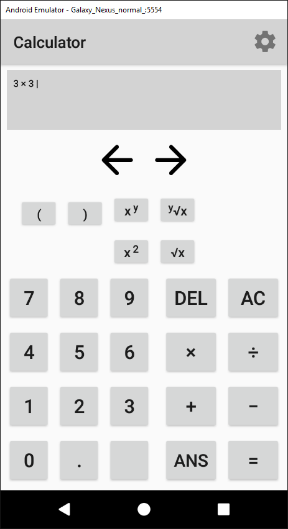
Two buttons were also added which were given the IDs of *shiftLeft* and *shiftRight*. They were both linked to a function that would either decrease or increase the position variable by one depending on which button was clicked.

In this function, a vertical slash character (“|”) is also added to the string in the current position index so the user has a graphical representation of where the position index is.

To finish the implementation of this success criteria, the method that handles the insertion of the user’s input to the expression was changed from adding the input to the end of the string. Instead, it would use a StringBuilder and insert the user’s input at the position index.

To make these changes, source code was added in Appendix 5.1.1, in the *shiftPosition* method.

Version 0.2.0



These screenshots show how the cursor can used to edit the expression by inserting a number and it will give the correct result.

#### V0.2.0 – Testing

Whilst testing Version 0.2.0, I noticed that the foundation of the project was not built as efficiently as it could have been, and this will affect the process of adding further features to the project.

The source of this inefficiency was located in the main calculator Java class which handled the user’s interactions and the GUI. The *ShuntingYard* class (which handles the process of converting the user’s input into an answer) did not experience these same problems.

The logic and structure of the code in the main activity was constructed improperly which created source code that was not easily understandable and therefore was difficult to maintain. This was especially clear as creating new features and adding them to the project became more complex as the project grew larger.

For example, each button the user could tap to enter input into the calculator was linked to an *OnClickListener* in the main activity. This overrode the android superclass *onClick* and handled each button separately using a switch-case statement. This was a very inefficient method to implement functionality for the buttons because it meant each button had to be programmed separately even when similar buttons did very similar things. E.g. tapping the “8” or “9” button does the same thing except enters a different digit.

Another effect of this illogical implementation is that the code being very long and complicated. This made maintaining the code almost impossible and making small changes to fix bugs became needlessly difficult.

Another example of how the codebase was built up incorrectly is a logical error that exists in the app. This logical error is that the variable that tracks the user’s position in the expression was being incremented at the wrong time in the program’s flow. This meant that when the position variable was used to insert input into the expression which often caused an off-by-one error.

This logical error was caused due to the lack of structure in the codebase which reduced the source codes readability and so lead to the code being difficult to understand and maintain.

In order to amend this, the next version will consist largely of refactoring the parts of the source code that handles the graphical user interface and the user’s input.

For example, I will streamline the process of the user entering input. I will remove the many separate buttons and instead of using *OnClickListeners* I will use the *onClick* XML characteristic which will call a specific method.

With this, I can group the buttons into similar groups and each set of groups will call their own method. For example, all the digit buttons (0-9) will call the *inputDigit* method and all operator buttons (+, -, ÷, ×, etc.) will call the *inputOperator* method.

I can use the *tag* XML characteristic to define which input (i.e. which digit or operator) to insert into the string depending on which button was clicked.

### 3.1.4 Version 0.3.0

#### V0.3.0 – Success Criteria

The success criteria for Version 0.3.0 covers refactoring the parts of the source code that are relevant to the calculator’s GUI and handling of the input. This does not include the *ShuntingYard* class.

The project was improved by changing the way the input buttons were handled to input the user’s choices.

In Version 0.2.0, each individual was linked to the overridden *onClick* method using *OnClickListeners*. This led to having a very large function that handled each button specifically. This was incredibly inefficient and hard to manage whenever changes had to be made.

In Version 0.3.0 I made the changes of using the XML characteristic *onClick* to link each group of buttons (e.g. digits or operators) to their own method. In the method, the input was found using the XML characteristic *tag* where the tag was the value to be inputted into the expression.

Changing the project to use this method means that the amount of duplicate code is severely cut down and so the project is made much more efficient to manage and to run. Because each group of buttons links to the same method, less code is required to achieve the same result.

This also means that when I make slight changes to the process, I only have to make one change to the button groups function rather than going through multiple cases and making individual changes.

The source code that handles the digit/operator inputs and the AC button input can be found in Appendix 5.1.1.

Another change made in V0.3.0 was to create new methods called *updateDisplay* and *removePositionMarker.* The *removePositionMarker* method removes all the underscores (character used to denote the user’s position in the expression) in the expression.

This method is used in *updateDisplay* to ensure the string is clean before updating the position marker and then updating the *TextView* that shows the calculators output.

*The method updateDisplay* is called whenever the user makes a change to their expression such as inputting a number to traversing through the expression.

Using a separate method to handle this is improved upon the previous version because it reduces the amount of duplicate code required and makes the projects source code much more understandable.

However, some parts of the previous version did not require refactoring. For example, the method *validateBracketMultiplication* which allows the user to use brackets as shorthand for multiplication (e.g. “5(2) = 5\*(2)”) remained the same. This method already functioned properly and did not require any changes except being ported to the new class.

#### V0.3.0 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test digit inputs work correctly. | Any digit, e.g. “4” | GUI shows “4” | GUI is updated to show “4” | N/A |
| Test operator inputs work correctly. | Any operator e.g. “-“ | GUI is updated to show “-“ | GUI is updated to show “-“ | N/A |
| Test bracket inputs work correctly. | Input “(“ and “)”. | GUI is updated to show brackets and they are interpreted correctly by the *ShuntingYard* class. | GUI is updated to show brackets and they are interpreted correctly by the *ShuntingYard* class. | N/A |
| Test expression to ensure they are being interpreted correctly by the *ShuntingYard* class. | An expression e.g.  “4(3+2^2)” | “28” is outputted. | “28” is outputted. | N/A |
| Test whether shift left/right buttons update position correctly. | Tap left/right button. | Position increases if right button and decreases if left button. | Position increases if right button and decreases if left button. | N/A |
| Test whether shift buttons update the position marker (underscore character) | Tap left or right shift button. | Position marker moves to the character left or right of the previous one. | Position marker moves to the character left or right of the previous one. | N/A |

### 3.1.5 Version 0.4.0

#### V0.4.0 – Success Criteria

Version 0.4.0 aims to complete the success criteria: “store relevant mathematical equations.”

This means that for this version to be considered successful, there must be a way for the user to access a menu where they can select from a range of equations such as the Quadratic Equation, Pythagoras’ Theorem, etc.

This was done by adding a new button to the main calculator layout which had the onClick characteristic set to call the function *onClickEquations* which starts the new activity, *Equations.*

This activity initialises the *RecyclerView* in the activity’s corresponding layout in the overridden method *onCreate*. This method sets the RecyclerView’s LayoutManager and adapter. The adapter is the class *EquationsRAdapter*.

The *RecyclerView* adapter populates the XML *RecyclerView* to show the user a list. In the method *EquationsRAdapter*, an *ArrayList* is created to make an array of strings that hold each equation’s name. This array is later used to populate the RecyclerView.

In the method *OnBindViewHolder*, the strings in the *ArrayList* are set to *TextViews* in the *RecyclerView* and then the method *OnCreateViewHolder* inflates each view so they are displayed to the user.

In *OnCreateViewHolder, OnClickListeners* are also set on each element of the list. This is linked to an *onClick* method that handles the user’s clicks.

In the *onClick* method, a switch-case statement is used on the variable *position*. This variable holds the index value of which element in the *RecyclerView* the user has clicked. Therefore, when the user clicks an element of the *RecyclerView*, it starts a different activity depending on the element clicked.

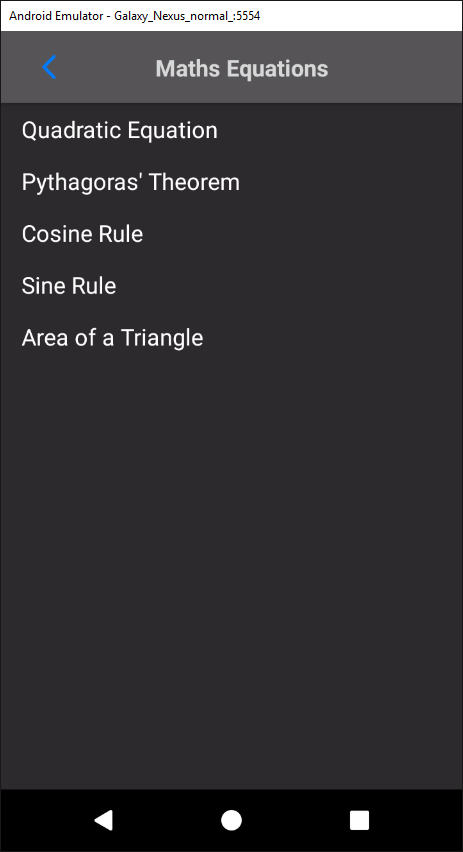
For example, if the user clicks the first element, (i.e.the Quadratic Equation element), it will start the activity that handles all Quadratic Equation.

Also, in this version, a back button was added to the Toolbar to allow the user to return to the main calculator page.

This version has been successful because I have created an activity which displays the choices of the different equations to the user. These choices are “The Quadratic Equation”, “Pythagoras’ Theorem”, “The Cosine Rule”, “The Sine Rule” and the “Area of a Triangle”.

In the next iteration of development, each equation that is relevant to the Maths GCSE will be added with the functionality.

The source code for the *Equations* and *EquationsRAdapter* classes can be found in Appendix 5.1.3 and 5.1.4 respectively.

*The Equations RecyclerView:* 

#### V0.4.0 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Click each element of the *RecyclerView* to ensure they are receiving the clicks. | Tap *RecyclerView* elements. | Activity responds correctly by sending user to the element’s corresponding activity. | Activity responds correctly by sending user to the element’s corresponding activity. | N/A |
| Tap the back button to ensure it send the user to the correct activity. | Tap back button. | User is sent to previous activity (*Calculator)* | Use is sent to previous activity (*Calculator)* | N/A |

### 3.1.6 Version 0.4.1

#### V0.4.1 – Success Criteria

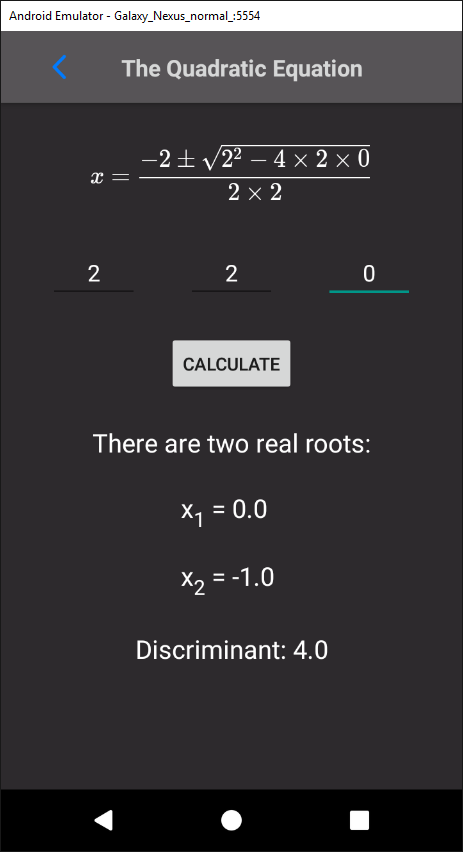
Version 0.4.1 will cover the success criteria off adding the Quadratic Equation. This is the first element in the list in the *Equations* class. When this element is clicked, the user is sent to the *QuadraticEquation* activity.

The quadratic equation is:

The class *QuadraticEquation* handles all the relevant processing for this equation. In the *onCreate* method, the *contentView* is set to the layout that was created for this activity. The layout displays the equations general formula and 3 *EditTexts* where the user can enter their input.

Below this are *TextViews* that are updated with the result when the user clicks the *calculate* button.

*The QuadraticEquation layout:*



In *onCreate*, I added *TextWatchers* to each *EditText*. *TextWatchers* listen for input on the *EditText* they’re attached to. By overriding the *TextWatcher* method *onTextChanged*, I can update the *String* that holds the variable value and call *updateDisplay.*

The method *updateDisplay* creates a new *String* called *quadraticEquation* that is then set as the text for the *MathView*. This process means that when the user changes the content of the *EditText*, the display equation will be updated in real time to display the user’s input.

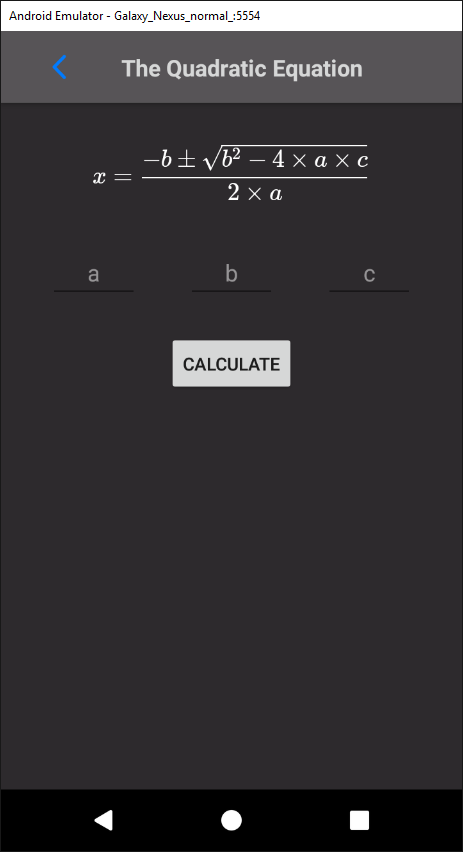
*MathView* is a library for display maths formula in Android apps. It was added to the project by implementing it in the project’s Gradle file.

This library is essentially the same as a *TextView* except it can render *TeX* code into maths formulas. *TeX* code is a typesetting system. Using the *MathView* library is very helpful because it means that special characters like plus-minus signs and fractions can be displayed easily. This is important as it increases the readability of the equations and greatly improves the user experience.

The *TeX* code that was used to render the quadratic equation is:

* String quadraticEquation = String.*format*(  
   "$$\\color{white}{x = \\frac{- %2$s \\pm \\sqrt{%2$s^2 - 4 \\times %1$s \\times %3$s}}{2 \\times %1$s}}$$",  
   aString, bString, cString);
* *$$* is used to signify the start and end of the *TeX* code.
* *\\color{white}{}* is used to change all the text in the empty brackets to the colour in the first brackets.
* *\\frac{a}{b}* is used to turn the content inside each bracket into fractions where the text in the first brackets are the numerator and the second are the denominator.
* *^* is used to put the text following the operator into superscript.
* [*\\times*](file:///\\times)renders a multiplication sign.
* [*\\pm*](file:///\\pm)renders a plus-minus sign.
* *%1$s* and *aString* is string substitution where *%1$s* is replaced by the *String* stored in the *aString* variable.

This is rendered as:



When the user clicks the *Calculate* button, it calls the method *calculateQuadratic*. This method handles all the processing required to calculate the answer for the user.

The method starts by closing Android’s virtual keyboard so the user doesn’t need to do it themselves. This makes the process smoother and improves the user’s experience.

The following code is surrounded by a try-catch statement so that any *NumberFormatExceptions* are caught and an error message is displayed instead of the app crashing. *NumberFormatExceptions* are caused when the program tries to run the method *parseDouble* from the *Double* class on an empty string. In this case, that would mean that the user hasn’t entered their input and so the error informs the user that they need to input values.

Inside the try-catch statement, firstly the discriminate is calculated. The discriminant is equal to “b2-4ac” and the quadratic equation will give different outputs depending on the value of the discriminant.

After calculating the discriminant, an if-else statement is run where if the discriminant is greater than zero then the quadratic equation will have two unique roots. The two roots are then calculated inside this if statement and the results are displayed to the user.

If the discriminant is equal to zero, then the quadratic equation must have two repeated roots. If this is true, then only one root is calculated and then displayed to the user.

If the discriminant is not greater than or equal to zero, then it must be negative. In the quadratic equation, the discriminant is square rooted. This cannot be done without using imaginary numbers which are not part of the GCSE specification. Therefore, an error is outputted to the user to inform them of this.

The *QuadraticEquation* is found in Appendix 5.1.5.

#### V0.4.1 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test if back button sends user to correct activity. | Click back button | User is sent to *Equations* class. | As expected. | N/A |
| Enter input into the *EditTexts* to ensure display equation is being updated in real time. | Enter values into each *EditText.* | The display equation is updated to show the user’s input. | As expected. | N/A |
| Calculate a quadratic equation that should result in two real roots. | a=2  b=2  c=0 | Layout displays both roots and the discriminant.  X1=0 and X2=-1.  Discriminant = 4 | As expected. | N/A |
| Calculate a quadratic equation that should result in one repeated root. | a=9  b=12  c=4 | Layout displays one root and the discriminant  X = -2/3 | As expected | N/A |
| Calculate a quadratic equation that should that have no real roots. | a=4  b=2  c=4 | Layout shows message saying there are no real roots and that the discriminant equals “-60”. | As expected. | N/A |

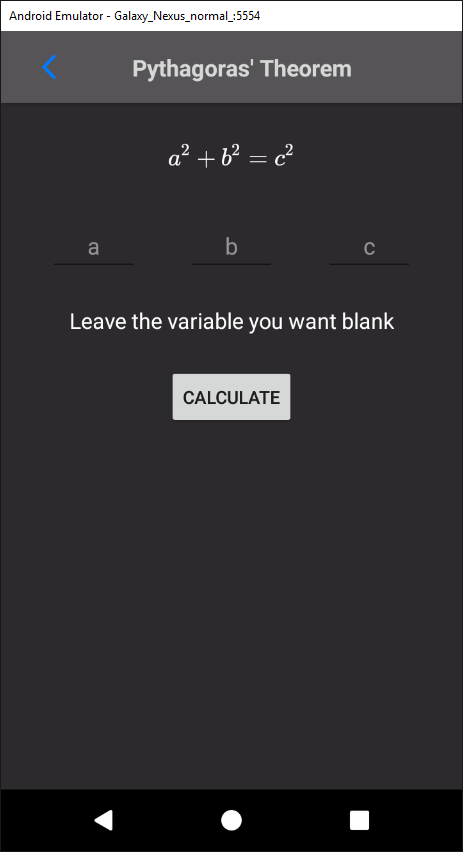
### 3.1.7 Version 0.4.2

#### V0.4.2 – Success Criteria

Version 0.4.2 has the aim of adding the next equation to the calculator, Pythagoras’ Theorem. When the Pythagoras’ Theorem element is clicked in the *Equations* *RecyclerView*, it sends the user to the *PythagorasTheorem* activity

Pythagoras’ Theorem is: . In *onCreate* of *PythagorasTheorem*, a *MathView* is set to display the equation in its general formula. Much like *QuadraticEquation, EditTexts* are added with *TextWatchers* to update the display when the user enters input.

*Pythagoras’ Theorem layout:*



When the user clicks the *calculate* button, it runs the method *calculatePythagoras*. In this method, the Android virtual keyboard is closed and a try-catch statement surrounds the code where the answer is calculated. This statement catches *NumberFormatExceptions,* which occur when *parseDouble* is run on a *String* that cannot be parsed. In context, this means that the user has not entered correct input into the *EditTexts.*

Pythagoras’ Theorem has two different versions. The first is where the two sides are known and the hypotenuse is calculated (). The second is where one side and the hypotenuse is known and the second side is calculated ().

To implement this in the app, an if statement was included which tested if the *EditText* for the “c” variable was empty. If so, this means the user wants to calculate the hypotenuse and so the values for each side were found by running *parseDouble* on *aString* and *bString*. These variables hold the values for sides “a” and “b” respectively.

After this, the hypotenuse was calculated using Pythagoras’ Theorem and displayed to the user.

If the input boxes for the variables “a” or “b” were empty, then it must mean that the user wants to calculate one of the sides. An exclusive or operator must be used here to ensure that only one variable was empty because Pythagoras’ Theorem wouldn’t work without at least one side.

If it is the variable “a” that was empty, the values for “b” and “c are fetched. A validation check must be run to ensure that the value for “b” is greater than the value for “c”. This is because the hypotenuse must always be the largest side and if it is not, then the user must have made an error. Therefore, an error message is displayed to the user informing them that their values are incorrect.

If the validation check is passed, then the side “a” is calculated using Pythagoras’ Theorem and displayed to the user.

If it is the variable “b” that was empty, the values for “a” and “c” are fetched. A similar validation check is run to ensure that the hypotenuse is greater and an error message is displayed if it is not.

If the validation check is passed, then the side “b” is calculated using Pythagoras’ Theorem and displayed to the user.

The *PythagorasTheorem* class source code can be found in Appendix 5.1.6.

#### V0.4.2 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Calculate the hypotenuse (“c”). | a=4  b=3 | Pythagoras’ Theorem gives the result:  5.0 | As expected. | N/A |
| Calculate side “a”. | b=5  c=12 | Using Pythagoras’ Theorem gives the result:  12.0 | As expected. | N/A |
| Calculate side “b”. | a=5  c=12 | Using Pythagoras’ Theorem gives the result:  12.0 | As expected. | N/A |
| Test back button to ensure it takes the user back to the *Equations* activity. | Click the back button. | The *Equations* class is loaded and the current activity is destroyed. | As expected. | N/A |

### 3.1.8 Version 0.4.3

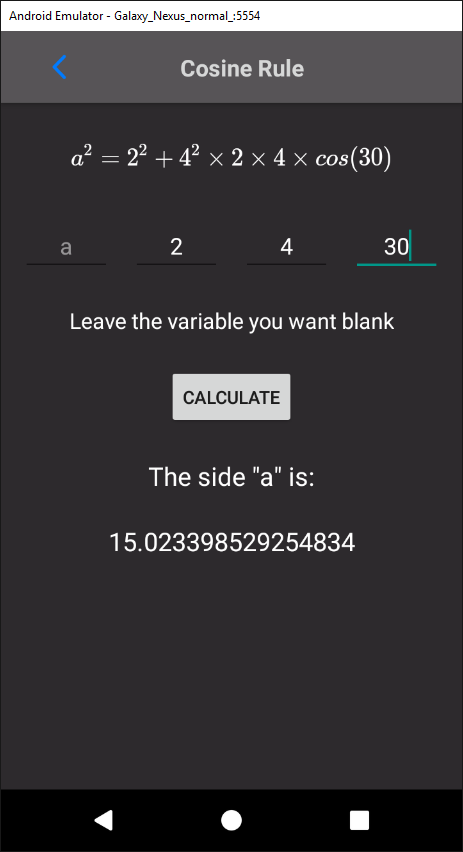
#### V0.4.3 – Success Criteria

Version 0.4.3 will aim to complete activity that handles the Cosine Rule. This equation’s activity will be started when the user clicks the corresponding element in the *Equations* *RecyclerView*.

In the *onCreate* method of *CosineRule*, the *MathView* that displays the equation to the user is set to display the Cosine Rule. This looks like:

The activity’s layout is created in a similar way to the other equations, where *EditTexts* are used for the user to enter their input and edit the display equation.

*The CosineRule layout:*



When the user clicks the “calculate” button in the activity’s layout, it calls the *caclulateCosineRule* method in the *CosineRule* class. First, this method hides Android’s virtual keyboard to clear up space on the screen.

The code required to calculate the Cosine Rule is surrounded by a try-catch statement. This will catch the *NumberFormatException* error which is caused by trying to parse a double from a string that doesn’t contain a number.

The Cosine Rule has two different forms. The normal form is as shown above and is used to calculate the side of the triangle. This can be rearranged to:

This form of the equation is used to calculate a specific angle of the triangle.

Both forms are required for Version 0.4.3 to be successful. To implement this, an if statement is run to check the *EditText* holding the input for the variable “a” is empty. If it is empty, then the user must want to calculate the side a, and therefore the values for the two other sides and the angle.

The angle values must first be converted to radians because as the Java library *Math* uses radians instead of degrees in the trigonometric functions.

The result can then be calculated using:

**double** a = Math.*sqrt*((b \* b) + (c \* c) \* 2.0 \* b \* c \* Math.*cos*(angle));

This is then displayed to the user.

If the user does not want to calculate the side “a”, then an if statement is used to see if the angle input is empty. If it is, then the user must want to calculate the angle. Therefore, the values for all the sides are parsed from the member *String* variables. The result is then calculated using:

**double** angle = Math.*toDegrees*(Math.*acos*(((a \* a) - (b \* b) - (c \* c)) / (2 \* b \* c)));

The calculated result is then displayed to the user.

If the user does not leave the *EditText* for either the “a” or the angle input then an error is displayed to the user informing them that their input is invalid.

The source code for the *CosineRule* class activity can be found in Appendix 5.1.7.

#### V0.4.3 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Calculate side “a” | b=3  c=5  Angle A = 90 | Calculator outputs side “a” = 3.0 | As expected. | N/A |
| Calculate angle “A”. | a=3  b=4  c=5 | Calculator outputs angle “A” = 143.13 degrees. | As expected. | N/A |
| Test back button to ensure it sends the user to the correct activity. | User taps back button. | The previous activity, *Equations* is started and *CosineRule* is destroyed. | As expected. | N/A |

### 3.1.9 Version 0.4.4

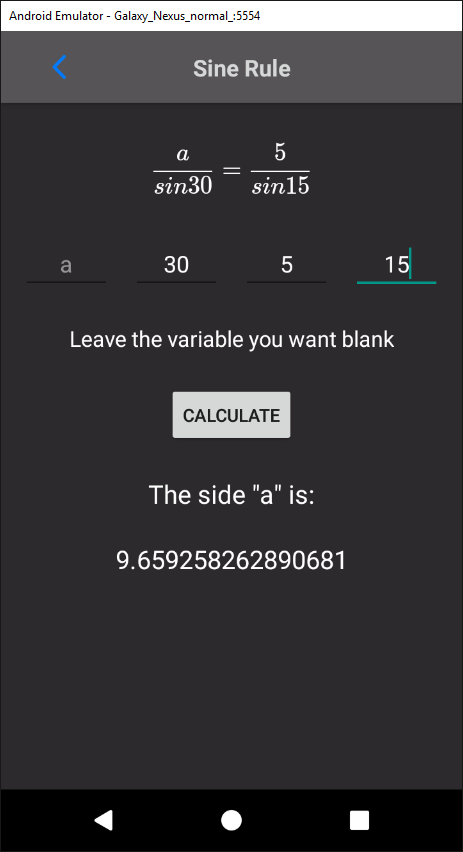
#### V0.4.4 – Success Criteria

Version 0.4.4 aims to complete the *SineRule* activity. This activity is opened when the user taps on the corresponding element in the *RecyclerView* in the *Equations* activity.

In the *onCreate* method of the *SineRule* activity, the text in the *MathView* in the layout is set to the Sine Rule equation:

A similar layout is used with this activity as with the other equation layouts. This means that the *EditTexts* can be used to update the display equation in real time.

*The SineRule layout:*



When the user clicks the “calculate” button, the virtual keyboard is closed and, inside a try-catch statement to prevent crashes and output errors instead, the result is calculated.

The result is calculated differently depending on which variable the user left empty (i.e. which variable the user wants to calculate).

The different variables can be split up into two groups, the sides and the angles. If either of the side inputs (“a” or “b”) are left empty, then the angle variables must be known and so they are retrieved and converted to radians.

An if statement is then run to determine if the “a” or “b” input was left empty. If it was the “a” input, then the value for side “b” is retrieved and the variable “a” is calculated. This is then outputted to the user using the *TextViews* in the layout.

If the “b” input is left empty, the value for side “a” is retrieved and used to calculate the result, which is then displayed to the user.

The sides are calculated with the expression:

**double** a = Math.*sin*(angleA) \* (b / Math.*sin*(angleB));

If the user wants to calculate an angle instead, the values for the sides are retrieved. Another if statement is then used to decide whether to calculate angle “A” or “B”.

If angle “A” is chosen, then the value for angle “B” is retrieved, converted to radians and then used in the Sine Rule to calculate the result. This answer is then displayed to the user.

The same thing is done if angle “B” is chosen, but with reversed variables.

The angles are calculated with:

**double** aAngle = Math.*toDegrees*(Math.*asin*(a \* Math.*sin*(bAngle) / b));

If neither the angles nor sides are left empty, then an error is displayed to the user. This error informs the user that they need to leave one input box blank.

The source code for the *SineRule* activity can be found in Appendix 5.1.8.

#### V0.4.4 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Calculate side “a” | Angle A = 15  Angle  Angle B = 15  Side b = 5 | Side a = 5 | As expected. | N/A |
| Calculate side “b” | Angle A=15  Angle B=15  Side a = 5 | Side b = 5 | As expected. | N/A |
| Calculate angle A. | Side a = 2  Side b = 3  Angle A = 30 | Angle A = 19.47 | As expected. | N/A |
| Calculate angle B. | Side a = 2  Side b = 3  Angle A = 30 | Angle B = 48.59 | As expected. | N/A |
| Tap back button to ensure it sends user back to correct activity. | Tap back button. | User is sent to previous activity – *Equations.* | As expected. | N/A |

### 3.1.8 Version 0.4.5

#### V0.4.5 – Success Criteria

Version 0.4.5 has the aim of adding the Area of a Triangle to the list of equations. This is done by first adding the Area of a Triangle to the *Equations* activity *RecyclerView.* When the user taps this element, it sends the user to the *AreaTriangle* activity.

In the *onCreate* method of *AreaTriangle*, a *MathView* is used to display the general equation:

A similar layout to the ones used in the other equations is used here.

*Area of a Triangle layout:*

// put image of layout here

When the user clicks the “calculate” button, it runs the method *calculateAreaTriangle*. In this method, the virtual keyboard is closed to free up screen space and a try-catch statement is used to prevent crashes from *NumberFormatExceptions*.

The variables for the sides and angle are retrieved and the angle is converted to radians for use in Java’s Math library. The area is then calculated using the code:

double area = 0.5 \* a \* b \* Math.sin(cAngle);

// replace code exercepts with copy&paste from github

The result from this is then displayed to the user.

// haven’t actually finished this, doesn’t calculate sides or angle.

#### V0.4.5 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Calculate area “A”. | a = 3  a = 5  C = 30 | A = 3.75 | As expected. | N/A |
| Tap back button to ensure it sends user to correct activity. | Tap back button. | User is sent back to *Equations* activity. | As expected. | N/A |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |