# Graphing - Programming Project

Shashi Balla

February 24, 2019

# Contents

# III Prototype II 94

# Part I

# Analysis

# Chapter 1

# Introduction

For my project I will attempt to make a graphing software. Graphing software is incredibly important in Linear Algebra and a lot of maths taught in schools is to do with Linear Algebra. Linear Algebra also links to many other aspects of Maths and hence is very important to understand. My stakeholders will consist of teachers, who will use my software to show graphs of functions, and to students who will use it to practise their graph sketching or to help them with their homework. There are many graphing tools out there, some of them shown below, however they have many downsides. I hope to make a piece of software that has as much functionality as possible, while retaining simplicity and reducing the number of downsides to an absolute minimum.
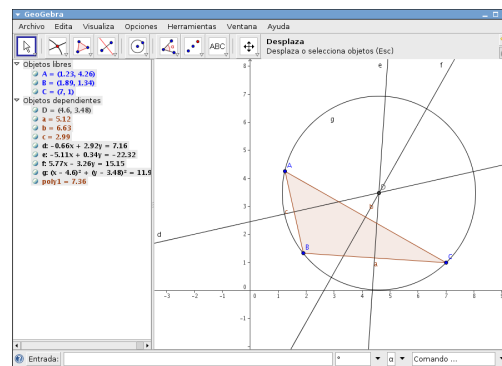


Figure 1.1: GeoGebra
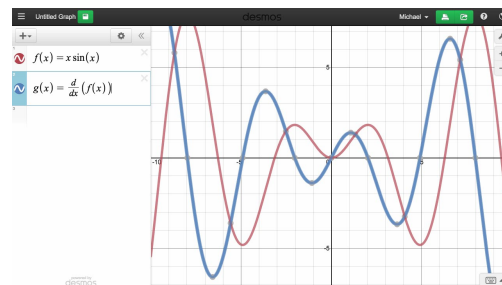


Figure 1.2: Desmos

# Chapter 2

# Stakeholders

My stakeholders will be comprised of teachers and students in year 10 and above. This is because graph sketching as a skill becomes very important from GCSE onwards and as such the tool will be aimed towards students in the upper years and the teachers themselves. I have interviewed several of my potential stakeholders about the software that they use and their opinions on it.

## 2.1 Teacher Interview

I interviewed Miss Naguthney who is a Maths teacher who teaches pupils from year 7 to 13. She uses multiple tools to aid her teaching. I asked her a couple of questions to get an idea of what she finds important in a graph drawing software.

| Question | Answer | Analysis |
|---|---|---|
| What graph drawing software have you used? | I have used Desmos, MATLAB and GeoGebra. | These three programs are very different, ranging from a simple graph drawing software, Desmos, to a professional data presentation software, MATLAB. |
| Which do you think is the best and why? | I think all 3 are good. In terms of pure graph drawing I think that Desmos is the best due to its simplicity. | From this response, it is important that my program is as simple as possible so that anyone can use. I think that to make program simple it must be responsive as well, and hence I should make my solution as efficient as possible so that it can be responsive on most devices. |
| What do you think is the most important aspect of a graph drawing software? | The most important part of a graph drawing software are obviously the graphs themselves and the most important part of the graphs are of course their intersections with axes, turning points and any asymptotes. | I agree with all these points made and I think that my program should at least be able to identify the turning points and any intersections with the axes. |

## 2.2   Student Interview

I find that during my sixth form studies, that graph sketching is a must have skill, and hence a graph drawing tool is important to verify that you have drawn a graph accurately. I interviewed Matthew, who is a sixth form student studying similar subjects, about graph drawing tools that he has used and his opinions about them.

| Question | Answer | Analysis |
|---|---|---|
| What graph drawing software have you used? | GeoGebra, Desmos and Kalgebra (KDE Application) | These three programs are quite similar as they have many of the same features. |
| Which do you think is the best and why? | I would say GeoGebra because:<br><br>• Native client so it is more responsive than a web-based app<br><br>• Versatile<br><br>• Easily Adjustable Axes<br><br>• Very easy to focus on a part of the graph<br><br>• Multiple function support<br><br>However the UI looks ugly and has no dark mode to reduce strain on the eyes. | I think that my program, should have the ability to be themed, through using CSS or a text file and while I may not be able to make my program versatile, it should still be able to draw multiple functions at once and navigating the graph must be fluid. |
| What do you think is the most important aspect of a graph drawing software? | The two most important aspects for me are ease of use and flexible input. Specifically this would be stuff like being able to input complex functions such as sums of multiple rational functions [1] (Not complex as in $x \in \mathbb{C}$) and the ability for the software to automatically adjust the scale. An example of this is for trigonometric functions sine and cosine. They don't have high $y$ values but they always appear very small on GeoGebra because the scale is wrong. A way to automatically set a suitable scale would be nice. Finally, I feel performance is also a must - it's frustrating to navigate the graph and have the application lag a lot. | Matthew's answer can be summarized into two points: Responsiveness and Flexible Input. I think that both of these points are valid and I should aim to make sure my application satisfies both these points. |

---

[1]A function $f(x) = \frac{P(x)}{Q(x)}$, where $P(x)$ and $Q(x)$ are functions of $x$

# Chapter 3

# Similar Programs

During my interviews with my stakeholders, they mentioned several pieces of software that they have used. I will be analyzing 3 of those programs in particular (or similar in the case of MATLAB, since MATLAB is extremely expensive), which are:

1. Desmos [1]

2. GeoGebra [2]

3. GNU Octave[3] (MATLAB[4] Clone)

## 3.1 Desmos

Desmos is a closed source graphing calculator written in HTML5, which allows it to be used on many devices. It has dedicated apps on Android and on Apple devices, but has no dedicated application on a desktop environment.

The real strength of Desmos comes from its ability to create activities that students can then complete.
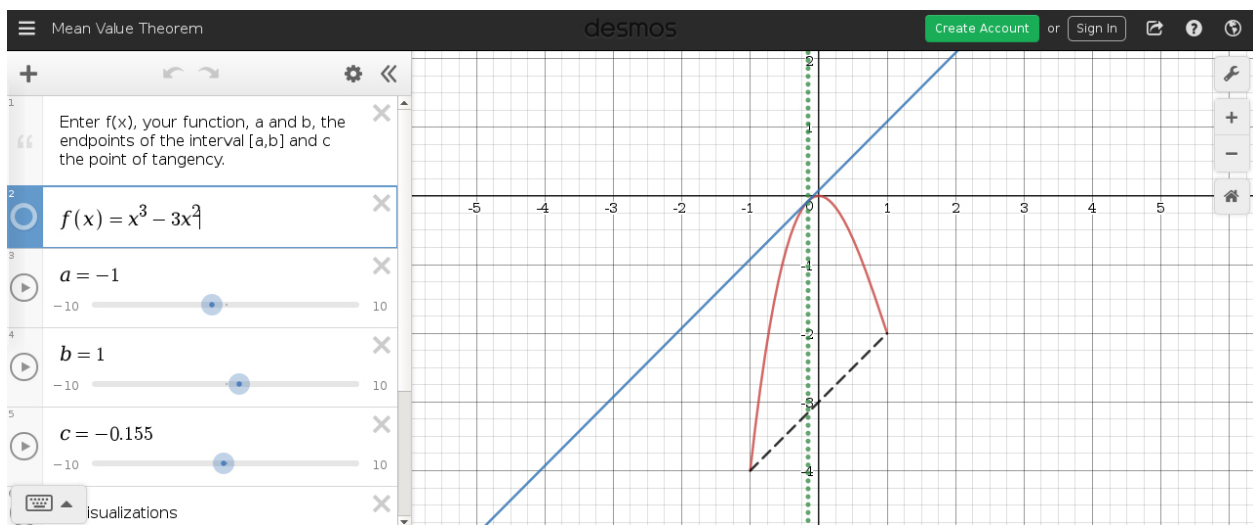


Figure 3.1: Mean Value Theorem example that students can interact with

The problem however is that it is web based and as such, can be clunky to use. The mobile app and website are especially unresponsive when zooming in or out and when panning around. Another pitfall is that Desmos tries to make itself more secure in a way by restricting input. This however makes input very difficult to do and along with the unresponsiveness, it can be impossible to input the required equation.



Figure 3.2: The Input

On the other hand, the actual graph drawing is excellent, with many features such as inequalities, modulus functions and the ability to draw polar functions. It also highlights turning points, intersections with axes, inflection points and in the case of trigonometric functions, it expresses them in terms of $\pi$ instead of a numerical values.

Overall as a graphing calculator it does an excellent job, but has poor optimization issues and input, which I have realized are both important and need to be at the top of my list of requirements for my program.

In terms of other features, you can print, save and share your graphs to use later. I think that my program should have at least 1, if not all 3, of these features.

## 3.2 GeoGebra

GeoGebra is an interactive geometry, algebra, statistics and calculus application under the GNU General Public License, meaning that it is open source.

The main advantage to GeoGebra is that it can do a lot of things and is all of its features integrate well with each other. For example you can create a shape on the graph and create lines intersecting it, linking geometry with algebra. Individual line equations can also be moved around allowing for a very interactive experience. Like Desmos, you can create activities that you can interact with.

The mobile app for GeoGebra has a similar input to Desmos and as such suffers from the same problems. However it is more responsive than Desmos but only slightly. The desktop version however does not suffer from any of these problems.

As a pure graphing software, it does not automatically show the points of interest of a curve and is instead confined in a tool called the "Function Inspector". This tool is awkward to use as it only shows local maximums and minimums within a certain range (the highlighted red region), and if there are multiple roots, it does not acknowledge what the values of the roots are. Desmos does better than GeoGebra in this regard. Like Desmos it supports many functions that you can plot such as modulus and inequalities.
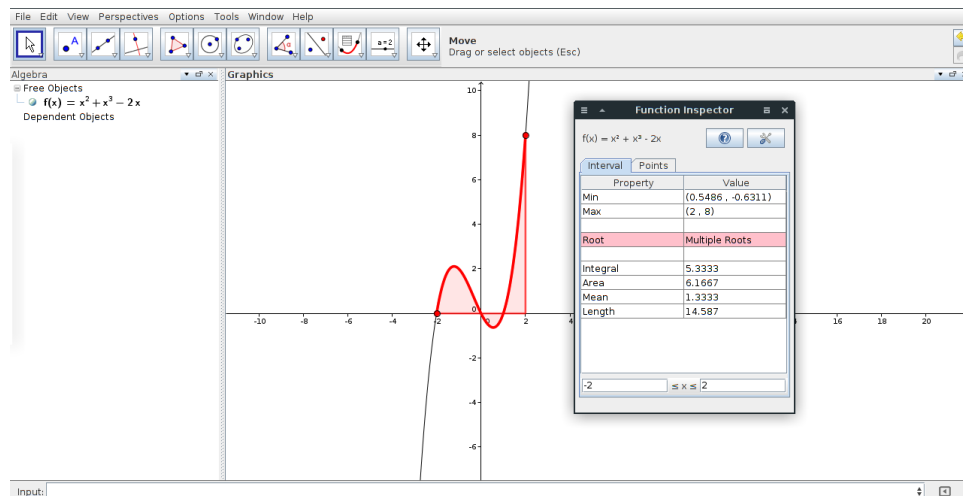


Figure 3.3: GeoGebra Function Inspector Tool

9

The panning and zooming in/out in GeoGebra is very fluid, with the zoom function being centered around the cursor. This means that you zoom in towards your cursor which feels more responsive than if it zooms in/out from the center of the graph. The only criticism is that you can accidentally move already drawn graphs when trying to pan, which can be frustrating.

In terms of other features, like Desmos, you can save and print your work as well as being able to export it into other formats such as JPG, PNG or even convert it into a PSTricks or TikZ environment which can be used in LaTeX documents for excellent graphs.

## 3.3   GNU Octave

GNU Octave is essentially an interpreter for a high level language centered around numerical calculation. GNU Octave is designed to be an open source software clone of MATLAB (MATrix LABatory).

Octave is different from GeoGebra and Desmos in the fact that it plots points that you tell it to not lines. For example if you wanted to plot $y = x^2$, you would define an array for your range in $x$ by doing:

$$x = -10 : 0.1 : 10;$$

This creates an array that starts at $-10$ increments by 0.1 until 10. You would then define $y$ by:

$$y = x. \wedge 2$$

You would then plot $x$ and $y$ by doing $plot(x, y)$;.

As a graphing tool Octave is not very impressive, it has limited support for implicit functions and will not highlight points of interest of a curve. Realistically this is expected since Octave is primarily made for data presentation. This means that it can produce excellent plots, including meshes and surfaces [1] in 3 dimensions. [2]
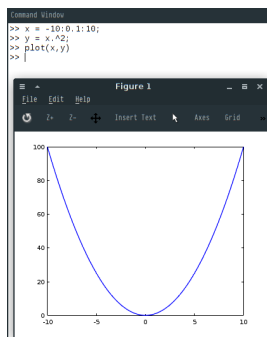
Figure 3.4: Creating a simple plot in GNU Octave
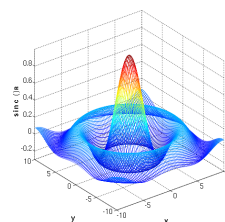
Figure 3.5: Wireframe 3D plot of the two-dimensional unnormalized sinc function

The way Octave plots (using the plot function at least) is by joining a line between consecutive points that it plots. I may use this in my program, since it seems more efficient to approximate a curve by making linear lines and joining them up than to check each pixel on the plot to check if it satisfies the given relation ship.

When inputting an algebraic expression in Octave it can be quite awkward, for example in Figure 3.4, a period is before the carat. This is unnatural for us to input and as such can cause errors. Natural input is

---

[1]A mesh is when a set of points in 3D space are joined up with lines to look like wire mesh. A surface is a mesh but adjacent points are used to make a solid shape instead, hence creating a surface.

[2]To clarify, a plot is where we explicitly define data that we then represent graphically, while a graph is where we define a relationship between 1 or more variables and represent this graphically.

something that I think I should focus on since it is means that the program is easier to use and requires no further reading unlike Octave.

Octave is split into two views, the main editor view and a dockable plot view (you can also have multiple plots open at once). Dockable plots, or multiple plots could be something I implement later into my project.

Octave, like the other programs, can save plots and additionally the scripts you have made. It also has a feature that saves the current state when you close it, and resumes this state when you open it again. This could also be a feature I implement later into the project.

## 3.4   Comparison

Overall I think that I will be using aspects of all three of the programs that I have analyzed in my program. I think I should try and make my program as simple to use as Desmos. As mentioned, the input is a problem, and hence I will try and implement an input system like GeoGebras'. All three of the programs analyzed have some form of saving their plots, so I think that this is a feature that I should implement within my own program.

In terms of UI design, both Desmos and GeoGebra have a similar layout. They have an equation input box to the left, that can be minimized and the actual graph in the center left. I think that this layout is easy to use for the user so I will implement a similar design in my own program.

| Main-Ribbon | |
|---|---|
| Input-Equations<br><br>$y = x^2$<br><br>New-Equation | Plot |
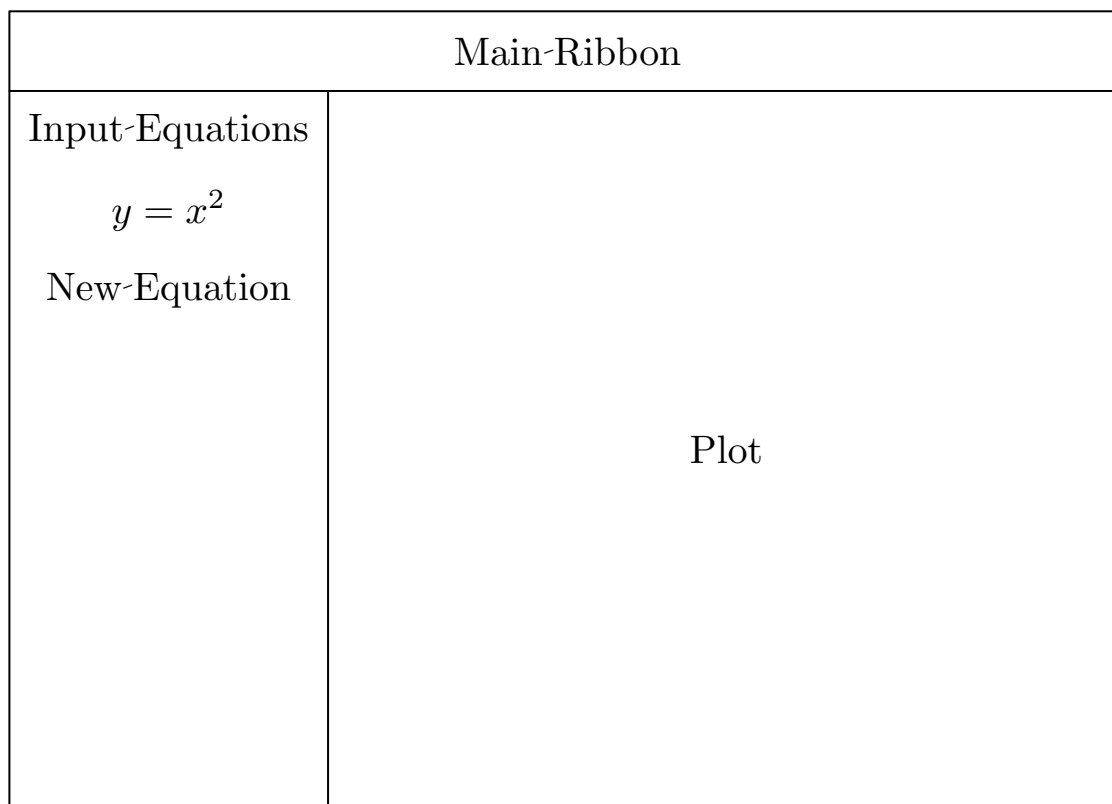
Figure 3.6: UI Design

# Chapter 4

# Computational Methods

## 4.1   Abstraction, Heuristics and Computability

Abstraction is the process of creating a model of a real-life problem by separating and highlighting important details. There are two main details that I can abstract away:

1. The input of expressions, and their conversion into a form that the computer can then manipulate.

2. The drawing of the function onto the screen.

By analyzing both parts of these details separately, we can determine if our problem is intractable.

The first part of the problem is solvable by converting the expression into an abstract syntax tree or into a post-fix stack, which is easily implementable.

The second part of the problem is solvable using heuristic methods. Most APIs, such as `OpenGL[5]` and `Vulkan`, only allow the drawing of primitives, which are points, lines and triangles. [1] This means that our curve cannot be drawn straight onto the screen and must be approximated by either using points or lines, hence there are two ways we can approximate the curve:

1. Let $x$ be our starting value, $f(x)$ our function and $dx$ be a small value. Now draw a line between the points $(x, f(x))$ and $(x + dx, f(x + dx))$; $(x + dx, f(x + dx))$ and $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$ and $(x + (n+1) \cdot dx, f(x + (n+1) \cdot dx))$. Here we are effectively creating line segments and joining them together to approximate the curve. As $dx \to 0$ our approximation tends towards the real curve. However as $dx \to 0$ the time taken tends to infinity and therefore we only need to make $dx$ small enough so that the human eye can not see individual line segments. Also as a practical note, as $dx \to 0$, the pixel density required to view the curve tends to infinity, therefore $dx$ needs to be big enough so that the $dx$ is not less than the individual pixel width.

2. In the method above we approximated a curve by creating line segments, here we will approximate a curve by using individual points. Let $x$ be our starting value, $f(x)$ our function and $dx$ be a small value. Now draw a point at $(x, f(x))$; $(x + dx, f(x + dx))$; $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$. This can take a lot of time since calculate where to draw a pixel $w$ number of times, where $w$ is the pixel width. In some cases, for complicated functions, it may be intractable.

By using the first method we can model a curve to good accuracy within good time, therefore our second part of the problem is computationally possible albeit, using a heuristic method.

---

[1]Strictly this isn't true, but most other things we can draw using these APIs usually consist of a combination of primitives, usually triangles.

## 4.2   Logical Approach

Many parts of this problem require clear decision making, making sure that user inputs are valid and making my solution as efficient as possible. I could validate and standardize user input by using `RegEx[6]`. I will definitely use standard structures in my solutions, and therefore will be able to logical traverse these structures to get the required solution.

## 4.3   Decomposition

Decomposition is where a problem is broken down into smaller sub-problems that are manageable and are solved individually and combined to make the complete solution. As stated in the abstraction section, I can break down the problem into two main parts. These parts could then be broken down further into classes and functions such as:

1. A function class which consists of:

    (a) Converting an expression into a form the computer can manipulate.

    (b) Evaluating the expression for a value $x$, where $x \in \mathbb{R}$.

    (c) Calculating the roots and turning points of the function.

2. A plot class which consists of:

    (a) Drawing functions onto the screen

    (b) Heuristically adjusts the scale so the graph looks good

    (c) Drawing the axes

This is just an example of how I can decompose the problem and therefore the problem will be decomposed more.

## 4.4   Divide and Conquer and Concurrent Processing

Divide and Conquer is where a problem is recursively broken down into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. While this may seem similar to decomposition, Divide and Conquer relates to algorithms. My solution may be using trees, and as such I can use a Divide and Conquer algorithm to convert an expression into a tree by looking at sub-expressions within the expression. This can then be further improved by using Concurrent Processing to process each sub-expression individually. I can also use Concurrent Processing to draw multiple lines or functions on the screen at once.

# Chapter 5

# Requirements

From my research and analysis I have created a set of requirements that I aim to fulfill. I have split these requirements into development iterations that they will be linked with. Each iteration has a theme that most of the requirements will follow.

- Iteration I - Emphasis on Core Functionality

  - Plot a explicit function in $x$
  - Plot multiple functions on the same plot

- Iteration II - Emphasis on the User

  - Let the user input functions
  - Zoom in/out of the graph
  - Pan around the graph
  - Plot special functions such as trigonometric functions, logarithms, modulus functions, etc.
  - Identify intersection with the axes
  - Multiple plots that you can switch between
  - Save plots as pictures
  - Save workspace to resume later
  - Dark Theme

- Iteration III - Advanced Features

  - Identify intersections between functions
  - Differentiate explicit continuous functions with respect to $x$
  - Polar equations
  - Implicit equations
  - Parametric equations
  - LaTeX equation support
  - Identify turning points

I will use `Java` to accomplish this task. `Java` is platform independent meaning that I do not have to compile for many different systems. It also means that if I want to create a mobile version I will only have to recreate the UI. Within `Java` I will use `JavaFX` for my UI. `JavaFX` allows for customisation of UI elements through `css` files, as well as an easy interface to draw objects onto through the screen, through its `Canvas` class. I am also comfortable with the language which makes it the most suitable language for this project.

# Part II

# Prototype I

# Chapter 6

# Design

## 6.1 Decomposition

My program will be decomposed into two main parts the GUI and the parsing of functions. The GUI will consist of the input of functions and the drawing of the functions. Many parts of this project can be achieved in a cleaner fashion using an OOP approach rather than a procedural approach.

### 6.1.1 OOP vs Procedural Approach Justification

An Object Oriented approach for my program is better than a procedural approach due to its complexity. An OOP approach allows parts of a program to be independent, and this means that these parts can be tested individually, allowing for simple debugging.

Specific to my project, Java is wholly based around the OOP approach and hence it makes more sense to use an OOP approach rather than a procedural one. Also many parts of my program can use the OOP concepts of inheritance and polymorphism. An example of this is the layer system I will be using later on to draw functions onto the screen. There are many different types of functions (again which can use inheritance), and each of these will have a coresponding layer. Each type of layer will draw, have very simlar attributes and other similar features which will be very slightly different for each type of function. By making all these different layers inherit a base layer class, it means that all of them can be accessed generally, especially when drawing. They can also all be stored in an array of the type of the superclass, and accessed (outside of the class) in a standard way making the solution cleaner and easier to test.

## 6.2   Function Class

Our basic `Function` class will contain two main methods:

- Parse - This method will convert the user's input into a data structure that we can use to evaluate. This data structure will be stored as a private attribute.

- Evaluate - This method will take a value of $x$ and input it into our function and return the value $f(x)$.

Our class will contain more methods and attributes later (colour of the line, roots, turning points etc.) but these can be considered later as these are quite small parts. There are two distinct methods to parse a function:

- We can convert the input into a data structure, which we can then use to evaluate a value $x$.

- We can convert the input into its equivalant in a scripting language, such as `Lua[7]`, and then use the scripting language to evaluate a value $x$.

An example of the second is shown below:

```
1   import org.luaj.vm2.*;
2   import org.luaj.vm2.lib.jme.*;
3   public double evaluate(double x) {
4           ScriptEngineManager mgr = new ScriptEngineManager();
5           ScriptEngine e = mgr.getEngineByName("luaj");
6           e.put("x", x);
7           e.eval("y = math.sqrt(x)");
8           return e.get("y");
9   }
```

The problem with using scripts is that they are take a huge hit on performance as you are effectively creating a virtual machine during your program. Especially in `Java` where a VM is used to run your programs, creating a VM inside a VM is not very effiecient. Also we may make upto 10000 evaluate calls to render multiple lines, and therefore we will need to be as effiecient as possible. While the first method is harder to implement, it will have better performance, and it will be easier to debug as we are not using external tools. Therefore we will use the first method.

There are two main structures we can convert a mathematical input into:

- Binary Trees

- Stack Based Programming

both of which we will analyze in detail.

### 6.2.1 Analysis of Algebraic Expressions

Algebraic Expressions are quite important in this project and as such it is important to break them down and understand what they really mean. The most basic idea in evaluating expressions is **BIDMAS**, which stands for **Brackets**, **Indices**, **Division**, **Multiplication**, **Addition** and **Subtraction**. It signifies the order that we must do operations on an expression (It is sometimes called **BODMAS** where the **O** stands for **Orders**) with the order of significance going from the first letter to the last. For example, let us use BIDMAS with the following expression:

$$3(4^2 + 2)$$

According to BIDMAS we will first look for brackets. There are brackets! The expression inside those brackets is $4^2 + 2$. We then apply BIDMAS again. There are no brackets but we do hae indices. We apply the indice function which produces $16 + 2$. Applying BIDMAS again we see that we must do addition which produces 18. Finally we have the expression $3(18)$ which is multiplication and it produces 54. Here we repeatedly took the most significant operator and applied its function until we fully evaulated our expression. Later we will instead take the least significant operator as this will allow us to split our expression down in a more structured manner.

In the example above there was nuance that we ignored. This was when we identified $3(18)$ actually meant $3 * (18)$. This implicitly represented multiplication and while as humans it is easy for us to process it, it is impossible for a computer to know this. It is therefore important that we remove these inconsistencies before we properly convert our input into a structure. It is also important that we strip away all whitespace before we start as this will allow our input to be more consistent. Here is a list of all these inconsistencies that we need to remove:

- Any instance of $ax$ where $a \in \mathbb{R} : a \neq 0$ is to be converted to $a * x$.[1]

- Any instance of $a($ and $)a$ where $a$ is not an operator, is to be converted to $a * ($ and $) * a$ respectively..[2]

- Any instance of $(f(x))(g(x))$ is to be converted to $(f(x)) * (g(x))$.[3]

- Any instance of $! - f(x)$ where $!$ is to be any operator (e.g. $*$ or $/$) is to be converted to $!(-f(x))$.[4]

- Any instance of $-f(x)$ at the start or next to an opening bracket is to be converted to $0 - f(x)$.[5]

---

[1]For example $4x$ is to be converted to $4 * x$

[2]For example $4(x + 1)$ is to be converted to $4 * (x+!)$

[3]For example $(x + 4)(x - 3)$ is to be converted to $(x + 4) * (x - 3)$

[4]If there is a negate symbol next to another operator, we need to make sure that the negate symbol is not treated as an operator (even though we treat it like an operator in certain situations in the next step)

[5]Both these expressions are equivalent but the second allows us to reduce ambiguity if there is a negate symbol at the start of an expression e.g. $-x + 4$ would be treated as $0 - x + 4$ and $(-x)$ is $0 - x$.

### 6.2.2 Binary Trees

The second stage that a compiler goes through is called Syntax Analysis and this is where the code is transformed into an Abstract Syntax Tree. What we are trying to achieve is very similar to that (albeit on a smaller scale). If we take a function $f(x) = x + 4$, this would be transformed into the binary tree:

$$+$$
$$4 \quad x$$

If we take something more complicated such as $f(x) = (x + 4)(x - 3)$, we need to first remove consistencies as discussed at the start of this section. Therefore the function would become $f(x) = (x + 4) * (x - 3)$. Now when we convert this into a syntax tree, we find the least significant operator (the operation we would do last), make it our root node and split the parameters that it is operating and make those parameters the child nodes of the root node. Here $*$ is the least significant operator and $x+4$ and $x-3$ become the child nodes.

$$*$$
$$(x + 4) \quad (x - 3)$$
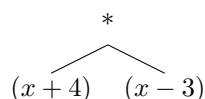
We can then repeat what we did above on the child nodes. In $x + 4$, $+$ is the least significant operator and in In $x - 3$, $-$ is the least significant operator. Therefore we now get:

$$*$$
$$+ \qquad -$$
$$x \quad 4 \quad x \quad 3$$
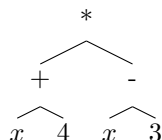
We now stop as we cannot split this down any further. What we notice here is that only the bottommost nodes are actually values, the rest are operators. This is significant, because what we just did was a recursive algorithm, where we split each node down, until a node does not contain any operators.

Now if we want to get $f(1)$, we simply replace every instance of $x$ with 1 and then perform the recursive algorithm outlined below.

| **Algorithm 1:** Evaluate: Binary Tree Version |
| --- |
| **1 function** *evaluate(tree)***:** |
| **2**     **if** *tree.height = 0* **then** |
| **3**        **return** tree.root.value |
| **4**     **else** |
| **5**        **return** *evaluate*(tree.leftsubtree) tree.root.value *evaluate*(tree.leftsubtree) |
| **6**     **end** |
| **7 end** |

This recursive algorithm has base case "tree.height = 0", which essentially checks if are at the bottom of our tree. We know that the bottommost nodes are actually values, therefore we can return this value as we cannot go deeper into our tree. Our recursive case is where we are not at the bottom of our tree. We know that at this point, the root node is an operator [6]. Therefore we operate on the nodes below it.

---

[6]It is important to realize that tree.root.value is just an operator, and in theory we can simply just write it as shown above but in implementation this will require us to use if statements to check which operator it actually is as the operators will most likely be of data type string

First let us replace every instance of $x$ with 1.

```
              *
           ╱     ╲
          +        -
         ╱ ╲      ╱ ╲
        1   4    1   3
```
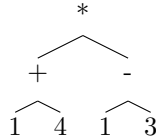
Now the root node has value "+", i.e. an operator therefore we take the left subtree which is,

```
          +
         ╱ ╲
        1   4
```

and again the root node is an operator so we take the left subtree again, which is 1. This is not an operator, so we return this value, going back to our previous call which is,

```
          +
         ╱ ╲
        1   4
```

Now we take the right subtree, which is 4. Again, this is not an operator so we return this value, going back to our previous call. Now we have **return** $1 + 4$, therefore we return 5 to our previous call which is,

```
              *
           ╱     ╲
          +        -
         ╱ ╲      ╱ ╲
        1   4    1   3
```
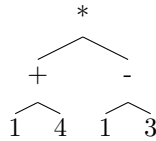
We now repeat the process for the right subtree,

```
          -
         ╱ ╲
        1   3
```

which returns $-2$. We then go back to our original call to get $5 * -2$ which returns $-10$.
The recursive tree for this is exactly the same as the original tree that we started with.

```
              *
           ╱     ╲
          +        -
         ╱ ╲      ╱ ╲
        1   4    1   3
```
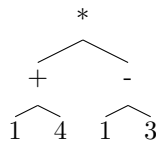
This is significant as this means that the number of recursive calls made is the number of edges, which is $n - 1$ where $n$ is the total number of operands and operators combined.

### 6.2.3 Stack Based Programming

Stack Based Programming[8] is all about using a stack data structure, and manipulating the items within it to get the desired result. Firstly we have to convert our *Infix Notation* into *Reverse Polish Notation*. Infix notation is the "normal" way of writing algebra, where the operator is inbetween its operands. Reverse Polish notation or Postfix notation is where the operator is after its operands. The advantage of this notation is that no brackets are required. For example $4 + 3$ in infix notation would be 4   3   + in postfix notation. To evaluate this expression we push each individual operator/value, from left to right, one by one, to a stack. If the value is an operator then it pops however many inputs it would normally take off the stack, perform the operation, on the values we just popped off, and then add that new value back to the stack. For example if we take the infix expression,

$$(((6 * (8 - 3)/3))/2) + 1$$

we can convert this to the equivalent postfix expression,

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2 \quad / \quad 1 \quad +$$

The way we get this is by looking for the least significant operator (the one we would consider last) and putting it to the end. We then take the operands of the operator which just removed, and repeat until we are left with only values. This seems very similar to the binary tree solution. In fact, if we convert this into a binary tree,
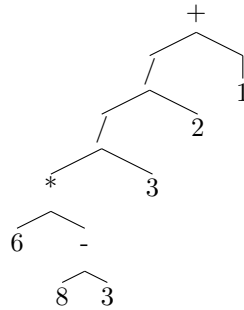


and if we then perform post-order depth traversal. So first we go all the way down the left hand side of the tree to get 6. We then visit its sibling, −, which has more children. So we visit 8, 3 then −. So far we have

$$6 \quad 8 \quad 3 \quad -$$

We then move up to ∗ and add this to our list. We then visit its sibling 3 and add this to our list. We have now got

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3$$

We then go up to / and add this to our list. We then visit its sibling 2 and add this to our list. We have now got

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2$$

We then go up to / and add this to our list. We then visit its sibling 1 and add this to our list. We then go up to the root node, +, and add this to our list. Finally we have,

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2 \quad / \quad 1 \quad +$$

This is identical to our post-fix notation. This is important as later on we can use this fact to make our solution as effiecient as possible.
Now if we start to evaluate this expression we get,

$$[6] \qquad \begin{bmatrix} 8 \\ 6 \end{bmatrix} \qquad \begin{bmatrix} 3 \\ 8 \\ 6 \end{bmatrix}$$

we have reached an operator, $-$, so we now pop 2 items of the stack, 3 and 8, and perform the operation, $8 - 3 = 5$. We now push this value onto the top off the stack and resume our evaluation.

$$\begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Again we have reached a operator so we repeat what we did before. Take 5 and 6 off the stack, push $5 * 6 = 30$ onto the stack.

$$\begin{bmatrix} 30 \end{bmatrix} \begin{bmatrix} 3 \\ 30 \end{bmatrix}$$

Take 3 and 30 off the stack, push $30/3 = 10$ onto the stack.

$$\begin{bmatrix} 10 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

Take 2 and 10 off the stack, push $10/2 = 5$ onto the stack.

$$\begin{bmatrix} 5 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

Finally Take 1 and 5 off the stack, push $5 + 1 = 6$ onto the stack. We are left with 6 and have now got our answer. Just like with the binary trees, if we have an unknown $x$, we can simply replace the x with a value when we want. The algorithm to evaluate post-fix notation is shown below,

---

**Algorithm 2:** Evaluate: Stack Based Version

```
1 function evaluate(list):
2     Stack stack
3     Array temp
4     Real out
5     Real x
6     foreach i in list do
7         if i is an operator then
8             pop = i.numberOfInputs
9             temp = new Array[pop]
10            for j=0 to pop by 1 do
11                temp[j] = stack.pop()
12            end
13            x = i.input(temp[1],temp[2],...,temp[n])
14            stack.push(x)
15        else
16            stack.push(i)
17        end
18    end
19    out = stack.pop()
20    return out
21 end
```

---

Here we are treating the operator to be an object for simplicity. In practice, all these values will be of data type String, therefore we will probably use some sort of selection contruct, either *if* or *switch-case* statements, to determine if $i$ is an operator and if it is, which one is it.

### 6.2.4 Analysis of the two Methods

When inititially parsing the infix expression $f(x)$, the binary tree version will have time complexity $O(n)$ where $n$ is the number of operators ($n$ will be the number of operators, and $m$ will be the number of operands in $f(x)$) in $f(x)$. The stack based version will convert the infix expression into a binary tree, and then perform depth-first traversal to convert the tree into post-fix. Depth-first traversal has a time complexity of $O(m + n)$ (every traversal visits every node, and there are $n + m$ nodes). This means that to initially parse the infix expression, the binary tree version has a smaller time complexity.

When evaluating $f(x)$ for a specific value of $x$, both versions have a time complexity of $O(m + n)$. This is because the binary tree version visits every node, $m + n$ (what we do is essentially post-order depth-first traversal but applying an operation each time), and the stack based version has $m + n$ items in the list that it goes through. Here we are assuming that each operation between operands takes equal time. This is a reasonable assumption to make because we are comparing two algorithms and they have the same input, $f(x)$, therefore we can remove the steps that it takes to complete the operations.

Now from a pure time complexity point of view, we can say that the binary tree version is better as the initial parsing is quicker. However when it comes to space complexity this is not true. When evaluating $f(x)$, the binary tree version uses a lot more memory because it creates subtrees every time it calls a recursive function. On the otherhand, with the stack based version, the maximum space that could be occupied is $(m + n) + m$. $(m + n)$ is the size of the orignal list that we pull values from, and $m$ is the maximum size our stack could ever get (we never add an operator to the stack). From a memory point of view a binary tree is not very effieciently stored. Most languages don't have a binary tree construct, and implementing your own with primitive arrays (each child is an array) is ineffiecient. For example the binary tree,
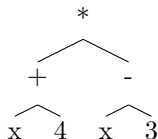


would be represented as (actual implementation would have pointers),

$$[*[+[x, 4]], [-[x, 3]]]$$

If we then stored this in contiguous memory,

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ * & + & x & 4 & - & x & 3 \end{bmatrix}$$

Now when doing post-order depth-first traversal, we jump between the memory locations,

$$\begin{bmatrix} 2 & 3 & 1 & 5 & 6 & 4 & 0 \\ x & 4 & + & x & 3 & - & * \end{bmatrix}$$

This is horribly ineffiecient because our stride is not consistent. This is unlike our stack based version, where we always have a stride of 1, no matter what. While this is not that big of a problem for one evaluate call, when 2000 evaluate calls are made, this makes a massive impact. We could, in theory, modify the order that we store our tree into one which is effiecient when doing a depth-first traversal but this is what we did when we converted our tree into post-fix notation. Therefore using stacks to process input is best for performance even though, the initial parsing takes longer. However this does not mean that we are not going to consider binary trees. This is because we need binary trees to initially parse our infix expression, therefore we will use stacks with binary trees to parse our user input. We need to inplement trees and stacks in Java before we can start parsing our infix expression.

### 6.2.5 Implementing Stacks and Trees

Since stacks and trees are a prerequisite to our function class, we will design them now.

Our stack does not need to become infinitely big, as the size of the `Stack`, as discussed in the last section, is $m$ where $m$ is the number of operands. Therefore we can initialize an array of size $m$ and manipulate the array to make it act like a `Stack`.

Our binary tree will be quite simple, containing only the bare minimum of what our binary tree needs, with an added traversal function which returns the post-order depth-first traversal of the tree. For simplicity we will use a private static helper function, which will take a tree and a stack as parameter. This stack which is passed in by reference will be edited in every recursive call. We will also make our left and right tree attributes public. This is because our root tree will be public, and as the left and right sub-trees are simply different copies of the root tree, it isn't consistent to make the right and left private while keeping the root tree public.

| Stack |
|---|
| ▬ Stack(int maxSize) |
| ▬ double pop() |
| ▬ void push(double n) |
| ▬ int getHeight() |
| 🔒 double[] stack |
| 🔒 int maxSize |
| 🔒 int height = 0 |
| 🔒 int pointer = -1 |

| Tree |
|---|
| ▬ Tree(String value) |
| ▬ String getValue() |
| ▬ void setValue(String value) |
| ▬ Stack traverse() |
| 🔒 Stack traverseHelper(Tree tree, &Stack order) |
| ▬ Tree right = null |
| ▬ Tree left = null |
| 🔒 String value |
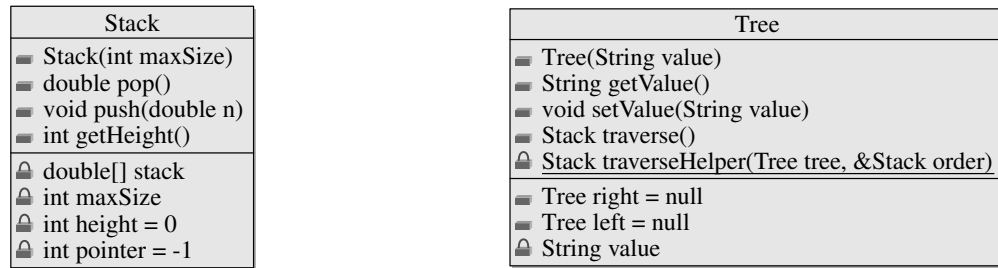
Figure 6.1: Class Diagrams

---

**Algorithm 3:** Post-Order Depth-First Traversal Helper

```
1 function traverseHelper(Tree tree, Stack &order):
2     if tree == null then
3         return;
4     else
5         traverse(tree.left,order)
6         traverse(root.right,order)
7         order.add(root.data)
8     end
9 end
```

---

**Algorithm 4:** Post-Order Depth-First Traversal

```
1 function traverse():
2     Stack order
3     traverseHelper(this,order)
4     return order
5 end
```

### 6.2.6 Parsing Algorithm

#### 6.2.6.1 Least Significant Operator

An important part of the parsing stage, is the ability to find the least significant operator within an expression. If there are no operators $-1$ will be returned. If an operator is inbetween within a pair of brackets it is automatically discounted. We do this by respectively incrementing or decrementing a variable when we reach an opening or closing bracket to monitor when we are within a pair of brackets.

---

**Algorithm 5:** Least Significant Operator Position

```
1  function leastSigOperatorPos(String input):
2      Integer parenthesis = 0
3      Integer leastSigOperatorPos = -1 /* stores the position of the least significant
           operator so far                                                              */
4      Integer leastSigOpcode = 1000
5      Character[] operators = ["+",' "-", "/", " * ", " ^'] /* stores each operator in order of
           increasing significance                                                      */
6      Integer currentOpcode /* the current index of the operator in the array operators */
7      Character currentChar
8      for i=0 to (input.size - 1) by 1 do
9          currentChar = input[i]
10         if currentChar in operators then
11             currentOpcode = operators.find(currentChar)
12             if (currentOpcode ≤ leastSigOpcode) and (parenthesis == 0) then
13                 leastSigOperatorPos = i
14                 leastSigOpcode = currentOpcode /* Update the least significant operator so
                       far, as it is now the current character                           */
15         else if currentChar == "(" then
16             parenthesis++
17         else if currentChar == ")" then
18             parenthesis−−
19         end
20     end
21     return leastSigOperatorPos
22 end
```

---

#### 6.2.6.2 Removing Brackets

The function above assumes that there is no whitespace and that there are no brackets enclosing the entire expression (e.g. $(x - 4)$). We can deal with our whitespace issue in our constructor[7] however we need to make another function check for and remove any brackets surrounding an expression.

However algorithm 6 has some issues. For example if we have the expression $\frac{x+1}{x+2}$, this would be input as, $(x + 1)/(x + 2)$. Now if we apply algorithm 6 which removes enclosing brackets we get $x + 1)/(x + 2$. This is completely wrong, as in this case we do not want to any remove brackets at all. The significant issue here is that we only want to remove the enclosing brackets, if they are ***matching***.

To do this the algorithm 7 is more suited. This algorithm is based on 6 but checks for matching brackets. It does this by using a variable that increments everytime there is a opening bracket and decrements every time there is a closing bracket. When the variable becomes 0 the matching bracket has been found. If this

---

[7]In our function class we will store the original input so we can show the user, therefore we do not need to remove whitespace here

happens at the end the opening and closing brackets are removed, else nothing happens. This algorithm will also throw an exception if there are unequal number of opening and closing brackets. This is so that we can inform the user later of the error that they have made and so that we can kill the process instantly rather than letting this error have consequences later on (probably during the evaluation of a value).

---

**Algorithm 6:** Check for and remove any Brackets surrounding an input

**1 function** *checkBracket(***String** *input)***:**
**2**     **Boolean** done = ***False***
**3**     **while** *!done* **do**
**4**         done = ***True***
**5**         **if** *(input[0] == '(')* **and** *(input[input.size - 1] == ')')* **then**
**6**             done = ***False***
**7**             input = input.substring(1, input.size - 2)
**8**         **end**
**9**     **end**
**10**     **return** input
**11 end**

---

**Algorithm 7:** Check for and remove any Matching Brackets surrounding an input

**1 function** *checkBracket(***String** *input)***:**
**2**     **Boolean** done = ***False***
**3**     **if** *input.count('(') != input.count(')')* **then**
**4**         **throw** "There is an unequal number of opening and closing brackets"
**5**     **while** *!done* **do**
**6**         done = ***True***
**7**         **if** *input[0] == '(' **and** *input[input.size - 1] == ')'* **then**
**8**             **Integer** countMatching = 1
**9**             **for** *i=1* **to** *(input.size - 2)* **by** *1* **do**
**10**                 **if** *countMatching == 0* **then**
**11**                     **return** input
**12**                 **else if** *input[i] == '('* **then**
**13**                     countMatching++
**14**                 **else if** *input[i] == ')'* **then**
**15**                     countMatching–
**16**                 **end**
**17**             **end**
**18**             **if** *countMatching == 1* **then**
**19**                 done = ***False***
**20**                 input = input.substring(1, input.size - 2)
**21**     **end**
**22**     **return** input
**23 end**

### 6.2.6.3   RegEx

Another important part of the parsing stage is to standardize the input. This is where we convert any inconsistencies discussed in section 6.2, page 17. The easiest way to do this is to use `RegEx[6]`. `RegEx` stands for regular expression and is a standardized form of pattern recognition in strings, usually used during syntax analysis during compilation of software. Many languages support `RegEx` in some form or another and Java is no exception. These were our 5 inconsistencies that we needed to fix:

1. Any instance of $ax$ where $a \in \mathbb{R} : a \neq 0$ is to be converted to $a * x$.

2. Any instance of $a($ and $)a$ where $a$ is not an operator, is to be converted to $a * ($ and $) * a$ respectively.

3. Any instance of $(f(x))(g(x))$ is to be converted to $(f(x)) * (g(x))$.

4. Any instance of $! - f(x)$ where ! is to be any operator (e.g. $*$ or $/$) is to be converted to $!(-f(x))$.

5. Any instance of $-f(x)$ at the start or next to an opening bracket is to be converted to $0 - f(x)$.

For the first and second examples, we look around every instance of $x$, ( or ) and if the adjacent characters are not operators or brackets then we replace with $*x$ or $x*$. Therefore we can combine the first and second examples to use two separate `RegEx` expressions to deal with the case where we have $x$ after and where we have $x$ before.

The `RegEx` expression for the first case is **"([∧\(\))\+\-\*\/\∧])([\(a-z])"** with the replacement expression being **"$1*$2"**. If we take the `RegEx` expression, it creates two capture groups, *"$1"* and *"$2"*, which are *"([∧\(\))\+\-\*\/\∧])"* and *"([\(a-z])"* respectively. A capture group stores a set of characters for each match that is made, so that we can perform actions on it later. The first capture group checks if the first character, in the substring that is currently being checked, is not any of the operators or brackets[8]. The not is signified by the first $\wedge$. The second capture group checks if the second character, in the substring that is currently being checked, is a constant/variable [9] , signified by the *"a-z"* or is an opening bracket *"("*. If both capture groups return true then a match is found and the match is replaced with *"$1*$2"* where *"$1"* is the first capture group, *"$2"* is the second capture group and the * asterisk between them signifying the multiply.

The `RegEx` expression for the second case is **"([\)a-z])([∧\(\))\+\-\*\/\∧])"** with the replacement expression being **"$1*$2"** again. This expression does the same as the first but checks for the reverse order i.e. $xa$ and instead checks for a ) instead of ( as we are checking the back of a substring instead of the start.

For the third inconsistency, the `RegEx` expression is **"\)\("** with the replacement expression being **")*("**. This expression returns a match if it finds a ) followed by a (. If it finds a match it then replaces the entire match with *")*("*.

For the fourth inconsistency, the `RegEx` expression is **"([\+\-\*\/\∧])-([∧\+\-\*\/\(\)\∧]*)"** with the replacement expression being **"$1(-$2)"**. The `RegEx` expression returns a match when there is an operator followed by a minus sign followed by any number of characters that are not operators or brackets. There are two capture groups. The first is *"([\+\-\*\/\∧])"* and this captures the operator. The second is *"([∧\+\-\*\/\(\)\∧]*)"* and this captures the expression after the minus sign. The match is then replaced by the first capture group, followed by an opening bracket, a minus sign, the second capture group, then a closing bracket.

For the fifth inconsistency, the `RegEx` expression is **"(∧|\()-"** with the replacement expression being **"$10-"**. The `RegEx` expression returns a match when it is either the start of a line, signified by the $\wedge$, or[10] a ( followed by a minus sign. The start of the line of bracket is captured and is used in the replacement expression, when the match is replaced with a 0- preceded by either a bracket or nothing depending on if the start of a line or an opening bracket was captured.

---

[8]the reason there are so many backslashes is because a lot of the operators are actually key characters in `RegEx` and a blackslash is an escape character which means that it signifies to treat the next character as a pure character

[9]While we are only dealing with the variable $x$ at the moment, this allows for constants to be used in the future and be dealt with correctly.

[10]the or keyword in `RegEx` is signified by |

### 6.2.6.4 Creating the Tree and the Stack

Using the functions above we can create a syntax tree from our expression. Before we dive into this, remember that we will convert this into a `Stack` later to process, since it is more efficient (section 6.2.4). To create the `Tree` we will use the Divide and Conquer methodology by recursively splitting the original expression until it becomes a single constant or variable. We can know if we should split the expression and if so where we should split the function by using our Least Significant Operator function. Our recursive case will return a `Tree` where the left and right nodes are made up of the trees of the two sub-expressions and the root node will be the operator that we split our original expression with. Our base case will return a `Tree` containing the constant or variable that is remaining. For example if we have the expression "$x^2 + 4$", we will first split this expression by the least significant operator which is the "+". We then make this our root node and our left and right trees will be what is left and right of that operator. This will be repeated for each sub-tree until only no operators remain. So our bottom nodes will be $x$, 2 and 4.

---

**Algorithm 8:** Create a Binary Tree for an Algebraic Expression

1 **function** *createTree(* **String** *expression)*:
2     expression = checkBracket(expression)
3     **Integer** leastSigOperatorPos = leastSigOperatorPos(expression)
4     **if** *leastSigOperatorPos == -1* **then**
5         **return new** Tree(expression)
6     **else**
7         **String** operator = expression[leastSigOperatorPos]
8         **String** a = expression.substring(0, leastSigOperatorPos)
9         **String** b = expression.substring(leastSigOperatorPos+1,expression.length - 1)
10        **return new Tree** (operator,createTree(a),createTree(b))
11     **end**
12 **end**

---

Within this algorithm I can use concurrent processing to process each sub-expression individually. To do this I will create 2 threads to create the 2 individual sub-trees. I will then start these threads and on the root thread wait for these threads to finish (usually done using a method called "join"), and then return the new `Tree` object made up of the 2 individual sub-tree.

---

**Algorithm 9:** Create a Binary Tree for an Algebraic Expression (Using Concurrent Processing)

1 **function** *createTree(* **String** *expression)*:
2     expression = checkBracket(expression)
3     **Integer** leastSigOperatorPos = leastSigOperatorPos(expression)
4     **if** *leastSigOperatorPos == -1* **then**
5         **return new** Tree(expression)
6     **else**
7         **String** operator = expression[leastSigOperatorPos]
8         **String** a = expression.substring(0, leastSigOperatorPos)
9         **String** b = expression.substring(leastSigOperatorPos+1,expression.length - 1)
10        **Thread** threadA = **new Thread** (createTree(a))
11        **Thread** threadB = **new Thread** (createTree(b))
12        threadA.join()
13        threadB.join()
14        **return new Tree** (operator,createTree(a),createTree(b))
15     **end**
16 **end**

---

### 6.2.7   Substitute Algorithm

To actually plot the functions we need to be able to substitute values into the actual functions to know where to draw our points. Eventually we should be able to substitute multiple variables/constants into our function, but for now we will only substitute an $x$ value into our function. This can be an extension for my next iteration.

Now algorithms 3 and 4 on page 24 traverse a tree and output a list (we will implement this using a stack, within the constructor). However the most significant operations and values, i.e. the actions we would do first according to **BIDMAS** [11] , are at the front of the list. This means that in a stack they would be at the bottom, which means they would be popped off last which is not what we want when we evaluate a function. While we can reverse the stack, this is inefficient and also defeats the purpose of using a stack.

However this is fixed by the need of this `substitute` algorithm. We will need to traverse this stack to replace the variables/constants with actual values anyway, so while we do this we can reverse the stack, fulfilling to purposes.

To accomplish this we will create a new stack, of the same height as the original, and then pop each value of the original stack, checking if it is a variable, and if it is substituting the value required. The value will then be pushed onto the new stack. This new stack will then be returned. We also will need to create a copy of the old stack and manipulate that so that we retain the original stack for other substitution calls.

In order to make my function class more generic, I will introduce a new attribute of type `character` called `parameter`. In all our examples our parameter has been $x$, but we may eventually use different parameters. Within this algorithm, instead of checking for the character $x$, I will instead check for the character `parameter`. Within this algorithm I have assumed that a `character` is equivalent to a `String` of

---

**Algorithm 10:** Substitute into a function

**1 function** *substitute(***Double** *x***):**
**2**     **Stack** copy = postFixStack                                          // Copy the Post-Fix Stack
**3**     **Stack** substituteStack = new **Stack** (this.postFixStack.getHeight())  /* Create new Stack the
          same size to reverse it into */
**4**     **for** *i=0* **to** *copy.getHeight()* **do**
**5**         **String** pop = copy.pop()
**6**         **if** *pop == parameter* **then**
**7**             | pop = x
**8**         substituteStack.push(pop)
**9**     **end**
**10**     **return** substituteStack
**11 end**

---

length 1 (on line 6). This is obviously not the case, therefore during implementation I will need to cast the character to a `String` or do another method that allows me to compare the two values.

---

[11]**BIDMAS** stands for **Brackets**, **Indices**, **Division**, **Multiplication**, **Addition** and **Subtraction**. It signifies the order that we must do operations on an expression. It is sometimes called **BODMAS** where the **O** stands for **Orders**.

### 6.2.8 Evaluate Algorithm

The `evaluate` algorithm is very similar to algorithm 2, the stack-based evaluate algorithm. However we will pass in the inputs as parameters and use the `substitute` algorithm within our `evaluate` algorithm to replace our variables with the actual values. We will also hardcode the selection and verification of which operator to use instead of assuming that an operator is an object we can manipulate. A precondition of this algorithm is that we are assuming that the original expression is valid, otherwise this algorithm may not behave as intended.

---

**Algorithm 11:** Evaluate the Expression for a Value of $x$

**1 function** *evaluate(***Double** *x***):**
**2**     **Stack** substituteStack = substitute(x)
**3**     **Stack** evaluateStack = **new Stack** (postFixStack.getHeight())
**4**     **for** *i=0* **to** *substituteStack.getHeight()* **do**
**5**         **String** pop = subStack.pop()
**6**         **Double** a,b
**7**         **switch** *pop* **do**
**8**             **case** "+" **do**
**9**                 b = evaluateStack.pop()
**10**                 a = evaluateStack.pop()
**11**                 evaluateStack.push($a + b$)
**12**                 **break**
**13**             **case** "−" **do**
**14**                 b = evaluateStack.pop()
**15**                 a = evaluateStack.pop()
**16**                 evaluateStack.push($a - b$)
**17**                 **break**
**18**             **case** "∗" **do**
**19**                 b = evaluateStack.pop()
**20**                 a = evaluateStack.pop()
**21**                 evaluateStack.push($a ∗ b$)
**22**                 **break**
**23**             **case** "/" **do**
**24**                 b = evaluateStack.pop()
**25**                 a = evaluateStack.pop()
**26**                 evaluateStack.push($a/b$)
**27**                 **break**
**28**             **case** "∧" **do**
**29**                 b = evaluateStack.pop()
**30**                 a = evaluateStack.pop()
**31**                 evaluateStack.push($a^b$))
**32**                 **break**
**33**             **default do**
**34**                 evaluateStack.push(**Double** (pop))
**35**                 **break**
**36**             **end**
**37**         **end**
**38**     **end**
**39**     **return** evaluateStack.pop()
**40 end**

---

### 6.2.9  Constructor and the Class as a Whole

The constructor for the `Function` class will be quite simple. We take an expression and the primary parameter as inputs and:

1. Remove Whitespace from the expression

2. Standardize the expression (using the RegEx rules)

3. Create the Binary Tree

4. Create the Post-Fix Stack from the Binary Tree

5. Set the Primary Parameter

---

**Algorithm 12:** Function Class Constructor

```
1 function Function(String expression, String parameter):
2     expression = expression.strip()
3     expression = standardize(expression)
4     this.expression = expression this.tree = createTree(this.expression)
5     this.postFixStack = this.binaryTree.traverse()
6     this.parameter = parameter
7 end
```

---

The class as a whole will look something like the diagram below.

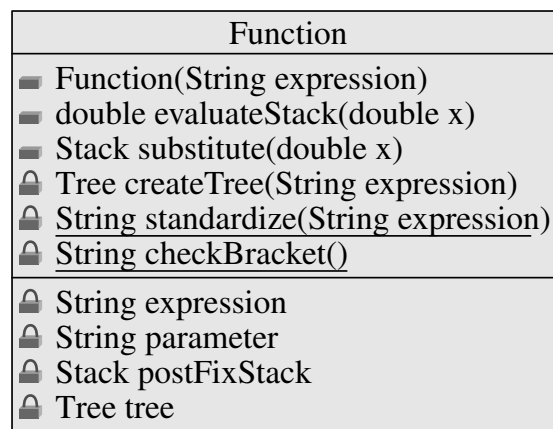| Function |
| --- |
| ▬ Function(String expression) |
| ▬ double evaluateStack(double x) |
| ▬ Stack substitute(double x) |
| 🔒 Tree createTree(String expression) |
| 🔒 <u>String standardize(String expression)</u> |
| 🔒 <u>String checkBracket()</u> |
| 🔒 String expression |
| 🔒 String parameter |
| 🔒 Stack postFixStack |
| 🔒 Tree tree |

Figure 6.2: `Function` Class

## 6.3 Graphical User Interface

The Graphical User Interface will be achieved using one the many toolkits within `Java`. Specifically I will be using `JavaFX[9]`. JavaFX is unique in the fact that the GUI can be generated using two approaches:

1. Within the code itself, we can define how the GUI[10] is comprised. This is done by instantiating Objects of specific component Classes, such `Pane`, `Label`, `Button` etc. all inheriting a top class called `Node`, and using their methods to join them together. The advantage of this approach is that since these are simply classes, we can create child classes from them allowing us to create custom components that we can control.

2. The UI is defined using `.xml` files (aptly named `FXML[11]` files). A controller class is defined alongside it. This controller class is referenced within the the `FXML` file and when the `FXML` file is loaded, the controller class is initialized along with it. This controller allows us to "control" the UI nodes. While this approach is very simple to implement and the design of the UI looks clean, it can be somewhat convoluted to allow the controller classes to interact with other controller classes.

The real strength of `JavaFX` however comes from the fact that these two approaches can be intertwined as needed, making solutions unique and versatile.

The GUI can be decomposed into smaller parts as shown by the hierarchical diagram shown below. The parts in **bold** are the parts to be implemented and designed in Prototype I and the rest will be completed in Prototype II.
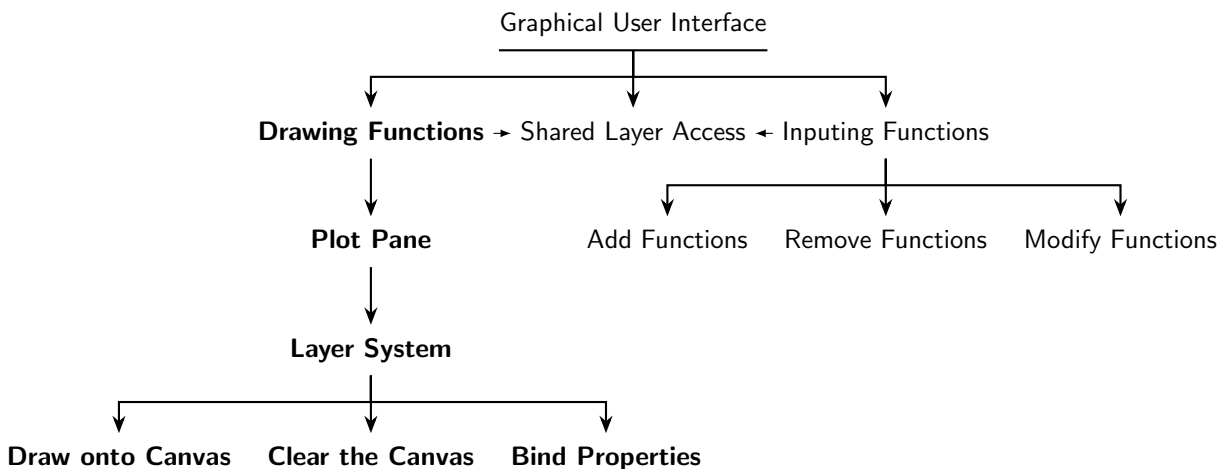


Figure 6.3: Graphical User Interface Hierarchical Diagram

Many of the features above will be discussed in greater detail later, however the Shared Layer Access and Properties will be touched upon now.

As mentioned before it can be hard to make two controller classes interact, as such one solution[12] is an object that both controller classes can reference. This class will contain attributes and methods that both classes need to interact with each other.

However, how do these classes know when something changes and hence know when to update their corresponding UI elements. The answer is through a class called `Property[13]`. A `Property` can bind another `Property`, one updating when the other one changes. Objects called `Listener` can also be added to properties. A `Listener` can be programmed to do a specific action when a `Property` is updated. This action could be a function, or updating a variable. Using the class `Property` allows for a responsive UI, updating whenever the user does an action.

### 6.3.1 Overview of the Graphical User Interface

The GUI will look something like:

| Main-Ribbon | |
|---|---|
| Input-Equations<br><br>$y = x^2$<br><br>New-Equation | Plot |
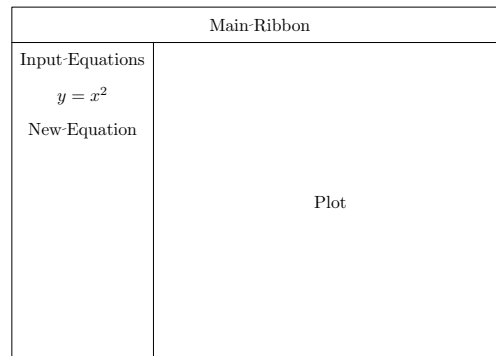
Figure 6.4: UI Design

    as showcased in the analysis.

Within `JavaFX` there is a hierarchy[14] of a standard GUI components. This is shown below:

Graphical User Interface

↓

Stage

↓

Scene

↓

Node 1    Node 2    Node 3    ...
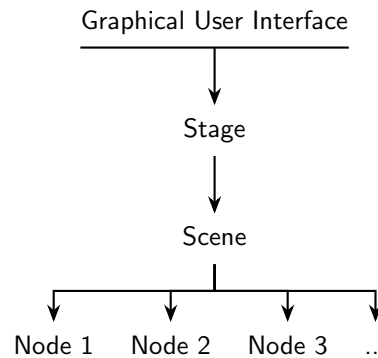
Figure 6.5: `JavaFX` Components Hierarchical Diagram

    Now we can manipulate the the nodes beneath `Scene`. It is standard to make the first `Node` beneath `Scene` some kind of `Pane`. For this project a `BorderPane` is most suited since it has multiple objects of type `Region`, that can hold one `Node` each. A diagram of this is shown to the right showcases how a `BorderPane` is arranged[15] and way it behaves.

    The center `Region` can consist of the plots themselves, and the left `Region` can consist of the input. The top `Region` could be used for the ribbon and other buttons to manipulate the plot.
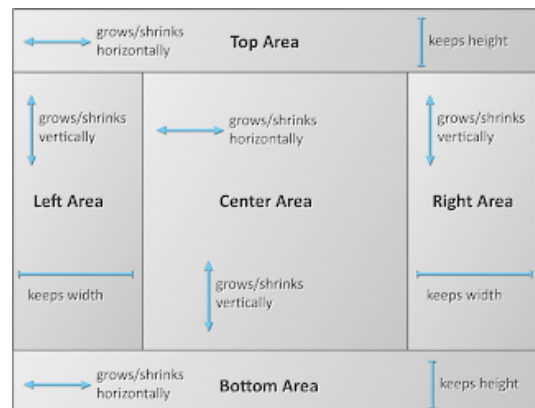
Figure 6.6: `BorderPane` Architecture

### 6.3.2 Layer System

Drawing onto the screen using `JavaFX` can be achieved using the `Canvas[16]` object. This is a node, which allows us to draw shapes, lines, arcs, on to the screen. As discussed in the analysis, we can approximate a curve by drawing many lines however one of our requirements was to draw multiple functions. From the article cited above, there are two ways to achieve this:

1. Draw every function onto one `Canvas` and add this to the root `Pane`.

2. Create a `Canvas` for every function and draw each function onto the corresponding canvasses and add all these canvasses onto the root `Pane`. One canvas will be paired with one function and bundled `Layer`. This only works because canvasses are by default transparent.

The advantage of the first is that it is simple to implement. However if there are multiple types of functions, i.e. an explicit function and an implicit function, then each one of these functions must have a draw function associated with it. This can become slightly convoluted, because then we must check what type of function something is, before we draw it.

Although the second is harder to implement, it is by far the superior. A `Canvas` is essentially an abstracted `OpenGL` view, and hence it is almost impossible[17] to draw multiple things on the same canvas at once. However it is possible to draw onto multiple canvasses at once, which boosts performance on multicore system. This also means that we can create a class for each type of function (explicit, implicit, parametric, etc.), making this system more modular. It is important to realize that only the topmost `Canvas` receives input (keyboard and mouse events) and hence we can create a `Layer` solely to handle input.

However before we handle the canvasses themselves, we must concern ourselves with how we can draw a line at a specific place.

#### 6.3.2.1 Coordinate Systems

The coordinate system that we are familiar with is the Cartesian Coordinate System. Cartesian Coordinates take a fixed point as reference, usually called the origin (symbolized by $O$), and specify a point by taking the distance from the origin in $n$ directions, where $n$ is the dimension of the system. The directions are vectors, that are all perpendicular to each other, and have magnitude 1, i.e. unit vectors. This graphing software will be dealing with a 2-Dimensional Cartesian Coordinate System and in this system, up and down are defined as the $y$ direction, with up being positive and down being negative; whereas right and left are defined as the $x$ direction, with right being positive and left being negative. Within our graphing program we will only show a section of the $x$–$y$ plane. This is because:

1. It is intractable to show the entire $x$–$y$ planesince the plane goes on for an infinite distance in both the $x$ and $y$ directions.

2. It is not helpful to the user to see the entire plane even if they could. A graphing tool should enable the user to highlight parts of the graph that is important to them.

However a `JavaFX Canvas` doesn't exactly use a Cartesian Coordinate System. It specifies a position on itself taking the top-left corner as the origin, right as the positive $x$ direction and down as the positive $y$ direction. It is important to realize that negative coordinates do not exist in this system. The maximum $x$ and $y$ values are the width and height, respectively, in pixels.

We must therefore find a way to convert between these two coordinate systems. Since our viewport is predefined, let us label what the maximum and minimum $x$ and $y$ values in our viewport (with respect to our original Cartesian Coordinate System). Let these values be `minX`, `maxX`, `minY` and `maxY`.
Now let us define how much the width and height of each pixel on the canvas is ''`worth`''. This *"worth"* specifies how much going across or down a pixel width or height, is in terms of our original Cartesian Coordinate System. This is best demonstrated with an example. If our canvas shows the $x$–$y$ planein the $x$ direction from 0 to 10, and our canvas is 1000 pixels wide, then our pixel worth is the amount we have

traversed in the $x$ direction, divided by the number of pixels, in this case $\frac{(10-0)}{1000} = 0.01$. The algorithm for this is shown below:

| **Algorithm 13:** Calculate the *"Worth"* of each Pixel |
|---|
| **1 function** *updatePixelWorth()*: |
| **2**     pixelWorthX = (maxX - minX)/canvas.getWidth() |
| **3**     pixelWorthY = (maxY - minY)/canvas.getHeight() |
| **4 end** |

Now that we have the pixel worth, we can convert between our coordinate systems. To do this we take the vector from the origin, $O = (x_{min}, y_{max})$, to the point, $P = (x, y)$ in the $x$ or $y$ direction. We then calculate how many pixel *"worths"* this is and this gives us the pixel on the canvas that this coordinate is at. It is important to realize that going down in a `JavaFX Canvas` is actually positive, relative to the canvas, hence our two algorithms are slightly different. This means our vector is actually

$$\vec{d} = \begin{bmatrix} (x - x_{min}) \\ (y_{max} - y) \end{bmatrix}$$

Hence our position within the canvas is:

$$\vec{p} = \begin{bmatrix} \left( \frac{x - x_{min}}{x_{worth}} \right) \\ \left( \frac{y_{max} - y}{y_{worth}} \right) \end{bmatrix}$$

The algorithms below convert between $x$ and $y$ coordinates [12] respectively.

| **Algorithm 14:** Convert the $x$ Coordinate |
|---|
| **1 function** *convertX(***Double** *x)*: |
| **2**     x = x - minX /* Calculate the horizontal distance between the point and origin    */ |
| **3**     x = x / pixelWorthX /* Calculate the number of pixel ``worths'' this is    */ |
| **4**     **return** x |
| **5 end** |

| **Algorithm 15:** Convert the $y$ Coordinate |
|---|
| **1 function** *convertX(***Double** *y)*: |
| **2**     y = maxY - y /* Calculate the vertical distance between the point and origin    */ |
| **3**     y = y / pixelWorthY /* Calculate the number of pixel ``worths'' this is    */ |
| **4**     **return** y |
| **5 end** |

---

[12]Here we assume the coordinate is inside the viewport. Realistically we should check if the coordinates are in the viewport, i.e. bigger than the width or height of the canvas, or in some cases negative. However it does not matter since we can make this check later. Even then a `JavaFX Canvas` will check if the coordinates are in the canvas viewport anyway, so it does not matter to us since this detail is abstracted away from us.

### 6.3.2.2 Drawing the Function

From the analysis (Section 4.1) I briefly described how I could approximate a curve by creating a finite number of line segments.

Firstly we take the minimum value of $x$, which in the viewport is $x_{min}$. If our function is defined by $f(x)$ then let the coordinate of the function at this value be $P_1 = (x_{min}, f(x_{min}))$. We then take a second coordinate a small value $dx$ away, so let the coordinate be $P_2 = (x + dx, f(x + dx))$. We now draw a line between these two points. We then make $P_1 = P_2$ and make $P_2$ with $x + 2 \cdot dx$. We repeat this process until $x + n \cdot dx \geq x_{max}$. We obviously need to convert the $x$ and $y$ values from the Cartesian coordinates into the canvas coordinates do we can do this when drawing the line. $dx$ will also be a finite value that is very small. We will define this by taking the number of line segments we want, and working out $dx$ for that number of steps. This is simply done by finding out how much each step is "worth" within the viewport, i.e. $dx = (\mathtt{maxX} - \mathtt{minX})/\mathtt{steps}$. The algorithm for this is shown below.

---

**Algorithm 16:** Draw a Function in the Viewport

1 **function** *draw()*:
2     **Double** x1 = minX
3     **Double** y1 = f.evaluate(x1)
4     **Double** dx = (maxX - minX)/steps
5     **for** *x2=minX + dx* **to** *maxX* **by** *dx* **do**
6         y2 = f.evaluate(x2)
7         drawLine(convertX(x1), convertY(y1), convertX(x2), convertY(y2))
8         x1 = x2
9         y1 = y2
10     **end**
11 **end**

---

A precondition of this is that f is a continuous function and as such we need to make sure we do not draw an undefined part of the graph, but this can be sorted out in the implementation.

### 6.3.2.3 Input Layer

While I will not design nor implement the responsive user input in this prototype, I will implement a basic input layer class with simple attributes. This is because it controls the other layers since if it zooms or pans around the viewport changes and this affects the properties defined in methods in the previous sections. Properties such as the minimum and maximum values of $x$ and $y$ for the canvas, and the pixel *"worth"*. So we should create a basic input layer so I can bind the properties of other layers to the input layer, which means that it is easier to build on in prototype 2.

This class will be quite bare, it will contain the methods to calculate pixel *"worth"* and update it whenever the canvas changes size. It will be updated in other scenarios when we actually start manipulating the viewport in prototype 2, but at the moment I will simply add a property called `changeViewport` that will eventually be connected to the `PlotPane` to notify it when any input is registered. The class diagram for this is included in the next section with all the rest of the layer classes.

#### 6.3.2.4    Constructor and Class as a Whole

The layer class we have described in the previous sections, describe a layer for an explicit function, in terms of $x$, in a Cartesian Coordinate System. To take full advantage of an OOP approach, as described at the start of this section, we can use layers for input, drawing axes etc. Hence we will create a superclass called `Layer` and then make our other types of layers inherit this class. This superclass will never be instatiated, in OOP terms it is abstract. The superclass will have some basic methods and attributes which we have used during this design, such as `draw()`, the attributes of the viewport (`minX`, `maxX`, etc.), a getter for the canvas (so it can be accessed by the `PlotPane`) and the functions to convert between coordinate systems, where some of the functions such as `draw()` will be overriden. Our `Explicit Function` and `Axes` classes are the only layers we will implement (apart from the superclass layer) in prototype 1. The input layer will be partially implemented readying it for prototype 2. While we haven't talked about the axes layer, this will simply be drawing the lines $y = 0$ and $x = 0$. This is done just for reference so that we can verify that the functions are drawn somewhat accurately.

The constructor for the Explicit Function Layer is shown below.

---
**Algorithm 17:** Explicit Function Layer Class Constructor

---
**1 function** *ExplicitFunctionLayer(* **String** *function)* **:**
**2**  |  super()
**3**  |  this.f = **new** Function(function)
**4 end**

---

The constructor for the Input Layer will be slightly different. Since the pixel *"worth"* needed to be updated if the window is resized, I will add a listener to update the values if the canvas changes size.

---
**Algorithm 18:** Input Layer Class Constructor

---
**1 function** *InputLayer()* **:**
**2**  |  this.canvas.heightProperty().addListener(event $\rightarrow$ updatePixelWorth())
**3**  |  this.canvas.widthProperty().addListener(event $\rightarrow$ updatePixelWorth())
**4 end**

---

While the superclass will never be instantiated, it can still contain a constructor. Here it will be used to initialize the canvas and its `OpenGL` context, in a consistent manner, so that we do not have repeated, redundant code for no reason.

---
**Algorithm 19:** Layer Class Constructor

---
**1 function** *Layer()* **:**
**2**  |  this.canvas = **new** Canvas()
**3**  |  this.gc = canvas.getGraphicsContext2D()
**4 end**

---

The class diagrams for the `Layer` [13] classes are shown on the next page:

---
[13]The inherited methods and attributes are protected so that they can actually be viewed and accessed by the child classes.
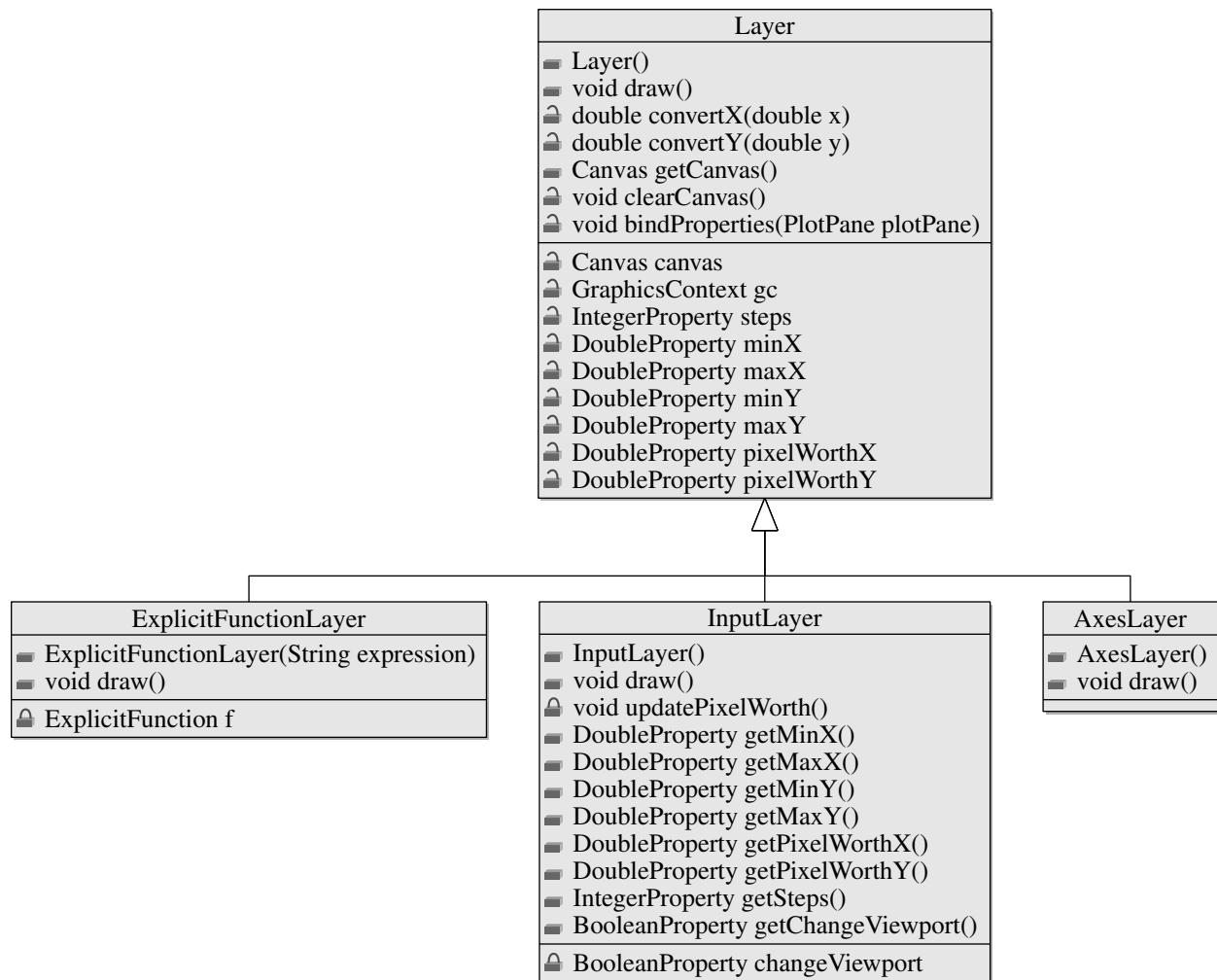
Figure 6.7: Layer Classes

### 6.3.3 Plot Pane

The purpose of the `PlotPane` is to manage the layers. The `PlotPane` will tell them when to draw and notify them when something changes through the use of properties. Since the `InputPane` will tell when something needs to be drawn, the `PlotPane` will contain a property which will be bound to the `InputPane` and we will draw whenever this property changes. This property, from the `InputPane` section, will be called `changeViewport`.

Our `PlotPane` will inherit the `JavaFX Pane`. While we could have used `FXML`, `FXML` is more applicable when there are complicated UI components that are hard to arrange and manage. In this situation we only need to have one `Pane` hence there is no need to use `FXML`. Inheriting `Pane` also allows it to integrate well within the `JavaFX` hierarchy.
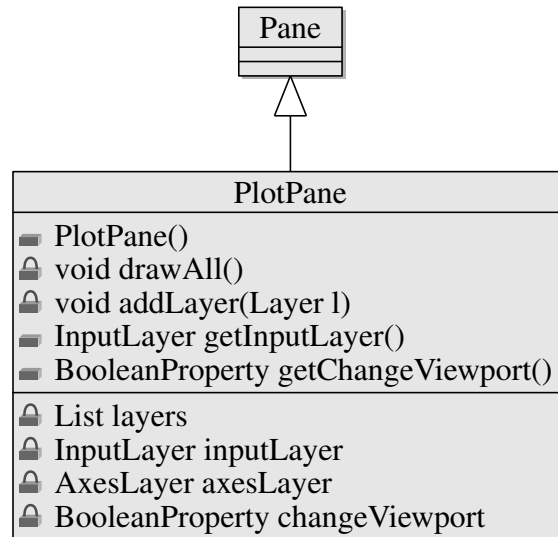


Figure 6.8: `PlotPane` Class

The `drawAll()` procedure will call the draw function for every `Layer`. We will start by doing this on a single thread, and then we will implement it using multiple threads. The `addLayer()` function will add the `Layer` to the list, and bind the layers' properties to the `InputLayer` attribute of `PlotPane`.

## 6.4 Testing Strategy

There are two main types of tests that I will do Unit and Overall tests.

Unit Tests will be done for most functions or classes that I create. Naturally these will be white box tests since I know how the unit works and am trying to find flaws within them. These will be based on testing the requirements that are defined in the design section. I will not create a detailed test table here since the tests are simply testing the requirements for each unit, but at the end of each unit implementation I will provide some code to test the unit, work out the expected result and compare the expected result to the actual output. If required I will modify the code to acquire the correct output. I will also define whether the data is valid, invalid or extreme.

Overall Tests will be done at the end of implementing the first prototype. Since I will not have defined any input mechanism to add functions in this prototype I will have to inject code, probably in a constructor somewhere, to add the functions I want to test. Naturally these will be black box tests since I am going to test them as if i were a stakeholder in the sense that I am inputting something and expecting an output without knowing the internal workings. I will include very little of the stakeholders at this stage since this prototype is only a foundation for user related features in the next one. Also all the tests will be for **valid data** we since only need to test the user cannot input anything yet. This means there will be no validation of input yet and hence there is no need to test the nonexistent validation with invalid data.
The overall testing table is below:

| Test Number | Function being Tested | Sub-Test Number | Input |
|---|---|---|---|
| 1 | Horizontal Lines | a | 4 |
| | | b | $-2$ |
| 2 | Linear | a | $2x - 5$ |
| | | b | $-0.5x$ |
| 3 | Positive Integer Factorised Polynomials | a | $x(x-1)$ |
| | | b | $x \wedge 2(x \wedge 2 + 1)$ |
| | | c | $-x(3x+1)$ |
| 4 | Positive Integer Unfactorised Polynomials | a | $x \wedge 2 + 2x + 1$ |
| | | b | $x \wedge 2 - 1$ |
| | | c | $x \wedge 3 + 4x \wedge 2 + x - 6$ |
| 5 | Non-Integer Polynomials | a | $x \wedge (1/2)$ |
| | | b | $x \wedge (1/3)$ |
| 6 | Exponentials | a | $e \wedge x$ |
| | | b | $2 \wedge x$ |
| | | c | $x \wedge x$ |
| 7 | Asymptotal | a | $1/(x-1)$ |
| | | b | $1/(x-1) + 1/(x+1)$ |
| 10 | Multiple and Coloured Functions | N/A | |

Table 6.1: Prototype 1: Testing Table

I have not included the expected output within this table. This is because I will generate these just before doing the test. These will most likely be in the form of an image from Desmos, another graphing software. Also you may notice how I have not included any details from the test for multiple functions. This is because I will draw all sub tests within the same test at the same time, testing the multiple functions feature at the same time. This allows me to see a function with respect to another function which makes the verification of whether the function is drawn correctly easier to do, I will also make each of the functions different colours, which allows me to test another feature and distinguish between the different functions.

# Chapter 7

# Implementation

When beginning to implement my solution, I made a few decisions. I realized a couple of important ideas which I did not address in my Design:

1. My function class designed was more of a class for an expression instead of a function. This is significant because different functions will use expressions in different ways. For example, an explicit function will use one expression to define the relationship, whereas an implicit function will use two expressions to define this relationship. Therefore to make our solution expandable in the future it would make more sense to have an expression class with multiple other classes for the different types of functions. These function classes can then inherit each other, for example if we make classes for different probability distribution, they are simply explicit functions with a specific standard structure, which makes them perfect child classes.

2. The layer system only relates to the Cartesian Coordinate System. This is a problem if I expand my program later to contain the Polar Coordinate System. So what I will do to fix this problem, is to take the `Layer` class that I created, rename it to `CartesianLayer` and make this class inherit a new class called `Layer` which will be all the attributes some key methods that all layers will have. I will also make another class called `ExplicitFunctionLayer` which will inherit from `CartesianLayer`. This allows me to split up each one of my layers into the types of function that will be used.

3. `Java` allows projects to be split up into packages. To organize my implementation I will split it up into the following packages:

   (a) `application` - Contains the UI classes, controllers, `PlotPane`, etc.
   (b) `layers` - Contains the `Layer` classes.
   (c) `structures` - Contains the data structure classes, stacks, trees, expressions, functions, etc.
   (d) `exceptions` - Contains custom exceptions.

## 7.1  Stacks

I will be implementing Stacks using a concept called *Generic Programming[18]*. This is best explained with an example.

Imagine we have a data structure which will only ever contain one type of object but we will use this data structure in different scenarios, each with different types of object. The structure must also be independent of the objects it is storing.

Now we can use an OOP approach where we have a base abstract class called `Structure` and then we create other classes that are related to a certain type of object and make it inherit this class, e.g. `IntegerStructure` or `StringStructure`. However this can become unfeasible to do this for every single object that we use this structure for.

A more elegant approach is to use *Generics*. This is where we link a type of object with the structure. Let this type of object be of type `T`. Now we make our structure of type `T` and make all of its methods take or return an object of type `T`. This allows the structure to be versatile and be used for any object.

Here is the implementation I have created in `Java` using *Generics[19]*:

```java
package structures;

import exceptions.StackOverflowException;
import exceptions.StackUnderflowException;

public class Stack<T> {

        // attributes
        // pointer variables
        private int maxHeight;
        private int height = 0;
        private int pointer = -1;
        // stack
        private T[] stack;

        // methods
        // constructor
        @SuppressWarnings("unchecked")
        public Stack(int maxHeight) {
                this.maxHeight = maxHeight; // set this max height
                // create an array of this size of type "T"
                this.stack = (T[]) new Object[this.maxHeight];
        }

        // push an item on to the stack
        public void push(T push) throws StackOverflowException {
                if (height == maxHeight) { // check that the stack isn't full
                        throw new StackOverflowException();
                } else {
                        // make the element one above the top, the new value
                        this.stack[this.pointer + 1] = push;
                        this.pointer++; // increment pointer variables
                        this.height++;
                }
        }
```

```
36          // pop an item off the stack
37          public T pop() throws StackUnderflowException {
38                  if (this.isEmpty()) { // check that the stack isn't full
39                          throw new StackUnderflowException();
40                  } else {
41                          // get the value of the top element
42                          // no need to make it null, it is a waste of an instruction
43                          // it also releases no memory since we are using
44                          // an array to store our stack which is a static structure
45                          T pop = this.stack[this.pointer];
46                          this.pointer--; // decrement pointer variables
47                          this.height--;
48                          return pop; // return the popped value
49                  }
50          }
51
52          // check if stack is empty, return true if so
53          public boolean isEmpty() {
54                  return this.height == 0 ? true : false;
55          }
56
57          // return the height of the stack
58          public int getHeight() {
59                  return height;
60          }
61
62          // allow for a visualization of the stack by listing it
63          // with the top element being encapsulated in square brackets
64          @Override
65          public String toString() {
66                  if (this.isEmpty()) { // if empty notify the user
67                          return "Stack is Empty";
68                  } else {
69                          // highlight the top element
70                          String out = "[" + this.stack[this.pointer] + "]";
71                          // loop through the rest of the stack backwards
72                          // concatenating each element to a string
73                          for (int i = this.pointer - 1; i >= 0; i--) {
74                                  out = out + ", " + this.stack[i].toString();
75                          }
76                          return out; // return this string
77                  }
78          }
79
80  }
```

There are two custom `Exception` classes that I have created for this class, `StackOverflowException` and `StackUnderFlowException`, in a separate package. These simply output a different message to the standard `Exception` class. The code for this is at the end of the documentation.

To just make sure that my code is working as intended I tested 4 things:

1. Popping and pushing items

2. Overflowing the stack

3. Underflowing the stack

4. Copying a stack (this is needed for the substitute algorithm)

Here is the code for test 1:

```java
// add two items, pop one off, add another one
Stack<String> stack = new Stack<String>(4);
stack.push("item 1");
System.out.println("Stack -> "+stack); // expecting item1
stack.push("item 2");
System.out.println("Stack -> "+stack); // expecting items 1 and 2
String pop = stack.pop();
System.out.println("Popped item -> "+pop); // expecting items 2
stack.push("item 3");
System.out.println("Stack -> "+stack); // expecting items 1 and 3
```

Here was the output for test 1:

```
Stack -> [item 1]
Stack -> [item 2], item 1
Popped item -> item 2
Stack -> [item 3], item 1
```

Which is exactly what I expected.

To spice things up for test 2 I made a stack of integers instead of strings just to make sure the generics part is working properly. Here is the code for test 2.

```java
// create a stack of max height 4
Stack<Integer> stack = new Stack<Integer>(4);
// push 4 items, the max amount
stack.push(1);
stack.push(2);
stack.push(3);
stack.push(4);
// show that the stack has 4 items
System.out.println("Stack -> "+stack);
// push another item
stack.push(5);
```

Here was the output for test 2:

```
Stack -> [4], 3, 2, 1
Exception in thread "main" exceptions.StackOverflowException:
Stack has reached its max height
        at structures.Stack.push(Stack.java:40)
        at structures.Stack.main(Stack.java:107)
```

Which is exactly what I expected.

Again for test 3 I made a stack of doubles. Here is the code for test 3.

```
// create a stack
Stack<Double> stack = new Stack<Double>(4);
// try and pop an item off
stack.pop();
```

Here was the output for test 3:

```
Exception in thread "main" exceptions.StackUnderflowException:
Stack has no items for you to pop
        at structures.Stack.pop(Stack.java:52)
        at structures.Stack.main(Stack.java:100)
```

Which is exactly what I expected.

For test 4 I made a stack of booleans. Here is the code for test 4.

```
// create a stack
Stack<Boolean> stack = new Stack<Boolean>(4);
// pop some items
stack.push(true);
stack.push(false);
// assign the value of the old stack to the copy stack
Stack<Boolean> copy = stack;
// pop an item off the copy stack to test that they are independent
copy.pop();
// output both stacks
System.out.println("Stack -> "+stack);
System.out.println("Copy -> "+copy);
```

Here was the output for test 4:

```
Stack -> [true]
Copy -> [true]
```

This is not what I expected. I expected the original stack to have an extra item. It appears as if they are mimicking each other, like they are sharing the same memory address. When I researched this issue[20] it appears that this is true. In Java every variable is a reference so when you assign a variable the value of an object, you actually assign the variable a reference. The exceptions to this are when you pass variables into functions, this is done by value not reference [1] , and with primitives [2]. The suggested solution was to use Serialization[21]. This is where an object is serialized into a byte array and then deserialized into an object again. This works because we are effectively converting an object into a primitive and then assigning this to a new value, which bypasses the reference issue. However this can be a hit on performance. So why don't I use the same principle in my code. Since all my attributes are primitives, if I just create a constructor which assigns the new stacks' attributes' values the original stacks' attributes, the problem should be solved.

---

[1] A function cannot have parameters passed by reference[22] in Java

[2] A language's base types of object that are inherently part of the language. In Java the primitives are int, double, float, boolean, char, byte, short, long.

I made a constructor that takes an existing stack as a parameter to solve the issue. In many languages this is called a *Copy Constructor*. The code for this is below:

```java
// copy constructor
public Stack(Stack<T> copy) {
        // assign the new stacks' attributes' values the original stacks' attributes
        this.pointer = copy.pointer;
        this.height = copy.height;
        this.maxHeight = copy.maxHeight;
        this.stack = copy.stack;
}
```

For my new test I slightly modified it from the original by using the copy constructor. Here is the code for new test:

```java
// create a stack
Stack<Boolean> stack = new Stack<Boolean>(4);
// pop some items
stack.push(true);
stack.push(false);
// assign the value of the old stack to the copy stack
Stack<Boolean> copy = new Stack<Boolean>(stack);
// pop an item off the copy stack to test that they are independent
copy.pop();
// output both stacks
System.out.println("Stack -> "+stack);
System.out.println("Copy -> "+copy);
```

Here was the output for the test:

```
Stack -> [false], true
Copy -> [true]
```

This is what I expected.

## 7.2 Trees

Our trees will be simpler to implement than our stacks. This is because we do not need to use generics with our trees since we will only be storing strings within them. The only hard part of this implementation was the fact that you cannot pass by reference[22] in `Java`. I got around this issue by simply passing this list in, and returning it each time. Here is the implementation I have created in `Java`:

```java
package structures;

import exceptions.StackOverflowException;

public class BinaryTree {

        // attributes
        // children
        public BinaryTree left;
        public BinaryTree right;
        // node value
        private String value;

        // methods
        // constructor - create a null tree
        public BinaryTree() {
                this.left = null;
                this.right = null;
                this.value = null;
        }

        // constructor - create leaf tree i.e. no children
        public BinaryTree(String value) {
                this.value = value;
                this.left = null;
                this.right = null;
        }

        // constructor - create a tree with the children defined
        public BinaryTree(String value, BinaryTree left, BinaryTree right) {
                this.value = value;
                this.left = left;
                this.right = right;
        }

        // return the value of the current node
        public String getValue() {
                return value;
        }

        // set the value of the current node
        public void setValue(String value) {
                this.value = value;
        }
```

```java
45
46          // count the number of nodes beneath this tree + this tree
47          public int countNodes() {
48                  // set the number of nodes counted to 0
49                  int count = 0;
50
51                  // recursive case, counts the number of nodes in its children
52                  // if they are not null
53                  if (this.left != null) {
54                          count = count + this.left.countNodes();
55                  }
56                  if (this.right != null) {
57                          count = count + this.right.countNodes();
58                  }
59                  // return the count of the children
60                  // but increment it to count it self
61                  return count + 1;
62          }
63
64          // post-order depth-first traversal wrapper function
65          public Stack<String> traverse() throws StackOverflowException {
66                  // create a stack of max height
67                  // which is the number of nodes in the entire tree
68                  // this is to save memory
69                  Stack<String> order = new Stack<String>(this.countNodes());
70                  // traverse the tree
71                  order = traverseHelper(this, order);
72                  return order;
73          }
74
75          // the actual recursive traversal function
76          private static Stack<String> traverseHelper(BinaryTree tree, Stack<String> order)
     ↪    throws StackOverflowException {
77                  // base case, if the tree is null just return the unmodified stack
78                  if (tree == null) {
79                          return order;
80                  } else { // recursive case search its children if not null
81                          // since this is post-order, the order is left, right then root
82                          // traverse left
83                          order = traverseHelper(tree.left, order);
84                          // traverse right
85                          order = traverseHelper(tree.right, order);
86                          // finally push the root value into the stack
87                          order.push(tree.value);
88                          return order;
89                  }
90          }
91
92  }
```

Just to test my binary tree implementation I tried to recreate the tree below:
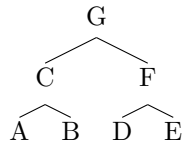


If this graph has post-order, depth first traversal performed on it then the output should be:

$$\begin{bmatrix} A & B & C & D & E & F & G \end{bmatrix}$$

Here is the test code:

```
BinaryTree root;
BinaryTree left = new BinaryTree("C", new BinaryTree("A"), new BinaryTree("B"));
BinaryTree right = new BinaryTree("F", new BinaryTree("D"), new BinaryTree("E"));
root = new BinaryTree("G",left,right);
// print the traversal stack
System.out.println(root.traverse());
```

Here is the output:

```
[G], F, E, D, C, B, A
```

While this appears to be different to our expected stack it isn't. If we look closely the items are actually reversed, which makes sense since a stack is FILO (First In Last Out) structure and we made the visual representation of our stack implementation reflect this property. This means that $A$ was pushed into the stack first, which means our tree implementation is working as intended.

## 7.3    Expressions

My `Expression` class, the one I designed as `Function`, is quite a complex and large, therefore I will implement each method independently and test them before I use them within my other methods.

### 7.3.1    Standardize Expression

I created a method to standardize my expression using the `RegEx` expressions that I made. This method will be `static` since it isn't related to the object, but instead to the class as a whole. I also added support for a few constants such as $e$ and $\pi$ so that we can use them later on. Here is the code:

```java
// standardize an expression to remove ambiguity
private static String standardize(String expression) {
        // remove whitespace
        expression = expression.replace(" ", "");
        // make pi one character for faster regex
        expression = expression.replace("pi", "");
        // inconsistency 1
        String regex1 = "([^\\(\\)\\+\\-\\*\\/\\^])([\\(a-z])";
        String replace1 = "$1*$2";
        // inconsistency 2
        String regex2 = "([\\)a-z])([^\\(\\)\\+\\-\\*\\/\\^])";
        String replace2 = "$1*$2";
        // inconsistency 3
        String regex3 = "\\)\\(";
        String replace3 = ")*(";
        // inconsistency 4
        String regex4 = "([\\+\\-\\*\\/\\^])-([^\\+\\-\\*\\/\\(\\)\\^]*)";
        String replace4 = "$1(-$2)";
        // inconsistency 5
        String regex5 = "(^|\\()-";
        String replace5 = "$10-";
        // perform regex
        expression = expression.replaceAll(regex1, replace1);
        expression = expression.replaceAll(regex2, replace2);
        expression = expression.replaceAll(regex3, replace3);
        expression = expression.replaceAll(regex4, replace4);
        expression = expression.replaceAll(regex5, replace5);
        // replace constants with numerical values
        expression = expression.replaceAll("e",Double.toString(Math.E));
        expression = expression.replaceAll("",Double.toString(Math.PI));
        // return the standardized expression
        return expression;
}
```

To test my module I tested each of the inconsistencies and then a complex expression just to make sure everything is working. Here is the code with comments for what I expected:

```
System.out.println(standardize("2x"));          // 2*x
System.out.println(standardize("(x)3"));          // (x)*3
System.out.println(standardize(")("));          // )*(
System.out.println(standardize("*-x"));                    // *(0-x)
System.out.println(standardize("3x(-x+1)(x+2)"));          // 3*x*(0-x+1)*(x+2)
```

Here are the results:

```
2*x
(x)*3
)*(
*(0-x)
3*x(0-x+1)*(x+2)
```

All the tests passed except the last one. It seems as though, the `RegEx` is ignoring the second instance of the multiplication for some reason. To see if it is a one time thing I ran the same `RegEx` expression again, as in I executed the `replaceAll()` functions again. For some reason this seemed to fix it as I got the expected output of $3 * x * (0 - x + 1) * (x + 2)$. To account for this, within my `standardize()` function, I will add a loop which make the `RegEx` checks are run until my expression has not changed between two consecutive `RegEx`. Here is the code[3] that I modified:

```
1   // loop until check and expression are equal
2   String check = "";
3   while(!expression.equals(check)) {
4           check = expression;
5           // perform regex
6           expression = expression.replaceAll(regex1, replace1);
7           expression = expression.replaceAll(regex2, replace2);
8           expression = expression.replaceAll(regex3, replace3);
9           expression = expression.replaceAll(regex4, replace4);
10          expression = expression.replaceAll(regex5, replace5);
11  }
```

While making this modification I originally checked if the previous and current versions of the expressions by doing `check!=expression`. This caused an infinite loop. This is because the `String` object in `Java` is quite unique in the fact that they act like primitives when assigning values (as shown in line 4 in the code above), but act like references when checking for equivalency[23]. In the example before, we were checking if they were the same reference i.e. the same object, not if they have the same contents, therefore it always evaluated as true (never same object hence true). Hence `!expression.equals(check)` is more suitable.

I tested this new function with the same arguments as before and here are the results:

```
2*x
(x)*3
)*(
*(0-x)
3*x*(0-x+1)*(x+2)
```

These are the results I expected.

---

[3]The full code is at the end of the section

### 7.3.2 Check Brackets

Like the `standardize()` function, this method is related to the class but not to a specific object hence it will be `static`. Here is the code:

```java
// check for and remove enclosing matching brackets
private static String checkBracket(String input) throws UnequalBracketsException,
    StringIndexOutOfBoundsException {
        boolean done = false;
        // check if there are an equal number of opening and closing brackets
        if((input.length() - input.replace("(", "").length())
                        !=(input.length() - input.replace(")", "").length())) {
                throw new UnequalBracketsException(input);
        }
        while(!done) {          // repeat until there are no enclosing brackets
                done = true;// assume no enclosing matching brackets until otherwise found
                // check if there are enclosing brackets (not necessarily matching)
                if((input.charAt(0) == '(') && (input.charAt(input.length()-1) == ')')) {
                        int countMatching = 1;          // initialize the bracket count as 1 to
                        ↪  count for the opening bracket
                        // loop from the second character to one before the end
                        for(int i=1; i<input.length() - 1; i++) {
                                if (countMatching == 0) {        // if we find the matching
                                ↪  bracket before the end terminate and return the input
                                        return input;
                                } else if(input.charAt(i) == '(') {
                                        // increment if we find an opening bracket
                                        countMatching++;
                                } else if(input.charAt(i) == ')') {
                                        // decrement if we find an closing bracket
                                        countMatching--;
                                }
                        }
                        // if the last character is the matching bracket keep the outer while
                        ↪  loop going and and remove the enclosing bracket
                        if(countMatching == 1) {
                                input = input.substring(1, input.length() - 1);
                                done = false;
                        }
                }
        }
        return input;
}
```

While this algorithm is explained in the design section, the confusing part of this is the checking if there are an equal number of opening and closing brackets. There is no builtin method for this, but we can use builtin methods to achieve the same result. To do this[24] we replace all of the character that we want with empty characters and then take this away from the length of the original String.[4]

---

[4]Let $l$ be the original length of the String, $x$ the number of instances of the character and $c$ the length of the reduced String:

$$l = x + c \implies x = l - c$$

To test this method I used the following code with comments for what I expected:

```
System.out.println(checkBracket("2x"));          // 2x - nothing changes
System.out.println(checkBracket("((x))"));        // x - 2 sets of brackets removed
System.out.println(checkBracket("(x-1)(x+1)"));     // (x-1)(x+1) - nothing changes
System.out.println(checkBracket("((x-1)(x+1))")); // (x-1)(x+1) - outer set of bracket removed
System.out.println(checkBracket("((x)"));         // throws an exception
```

Here are the results:

```
2x
x
(x-1)(x+1)
(x-1)(x+1)
Exception in thread "main" exceptions.UnequalBracketsException: There are an unequal number of
↪   opening and closing brackets in the expression: (()
        at structures.Expression.checkBracket(Expression.java:133)
        at structures.Expression.main(Expression.java:274)
```

Which is exactly what I expected.[5]

---

[5]Like with the Stack class I created a custom exception for this.

### 7.3.3 Least Significant Operator

Again this method is not linked an object, hence it will be static. The code is shown below:

```java
// find the least significant operator in an expression
private static int leastSigOperatorPos(String input) {
        int parenthesis = 0;
        int leastSigOperatorPos = -1;
        int leastSigOpcode = 1000;           // make the opcode super high so every operator is
        ↪  more significant than it
        // used a string to store the operators as it is essentially a char array but with
        ↪  added utility such as finding elements
        final String operators = "+-*/^";
        int currentOpcode;
        // loop through the entire input
        for(int i=0; i<input.length(); i++) {
                char currentChar = input.charAt(i);
                currentOpcode = operators.indexOf(currentChar);
                // check if it is an operator
                if(currentOpcode>=0) {
                        // check if it the same or more significant and if it is not enclosed
                        ↪  in parenthesis
                        if((currentOpcode <= leastSigOpcode) && (parenthesis == 0)) {
                                // update the significance and the position of the operator
                                leastSigOperatorPos = i;
                                leastSigOpcode = currentOpcode;
                        }
                } else if(currentChar == '(') {
                        // increment to signify we have entered an enclosing bracket
                        parenthesis++;
                } else if(currentChar == ')') {
                        // decrement to signify we have exited an enclosing bracket
                        parenthesis--;
                }
        }
        return leastSigOperatorPos;
}
```

To test this method I used the following code with comments for what I expected:

```java
System.out.println(leastSigOperatorPos("3+x^2"));          // should return 1 for the +
System.out.println(leastSigOperatorPos("(x+3)"));          // should return -1 since it is
↪  enclosed in brackets
System.out.println(leastSigOperatorPos("x*(x+3)^3"));       // should return 1 for the *
```

Here are the results:

```
1
-1
1
```

Which is exactly what I expected.

### 7.3.4 Create the Abstract Syntax Tree

I first implemented the normal sequential function and later I will implement the multi-threaded approach. Again like the methods before it this is a static function, since it relates to the class not the object itself.

```java
// create the abstract syntax tree for the expression - single-threaded
private static BinaryTree createTree(String expression) throws UnequalBracketsException {
        // remove enclosing matching brackets
        expression = checkBracket(expression);
        // find the least significant operator, if an operator remains
        int leastSigOperatorPos = leastSigOperatorPos(expression);
        if (leastSigOperatorPos == -1) { // base case - no operators remain
                return new BinaryTree(expression);
        } else { // recursive case - recurse on the sub-expressions
                // locate and hold the operator
                String operator = String.valueOf(expression.charAt(leastSigOperatorPos));
                // split the expression into sub-expressions by the operator and recurse on
                ↪   them
                String a = expression.substring(0, leastSigOperatorPos);
                String b = expression.substring(leastSigOperatorPos + 1);
                // return the new tree containing the trees of the sub-expressions and the
                // operator as the root value
                return new BinaryTree(operator, createTree(a), createTree(b));
        }
}
```

The problem with creating a tree is that I can't actually display it in a way to check if my algorithm works. To get around this issue I can just use my traversal algorithm. I have tested my traversal algorithm and know that it works hence I can just output the stack from traversing it. I will standardize my expressions before I create the trees since this is what will happen in the constructor later.

```java
//  create the standardized expressions
String a = standardize("2^x");
String b = standardize("(x+1)(x-1)");
// create the trees
BinaryTree treeA = createTree(a);
BinaryTree treeB = createTree(b);
// show the conversion between the expressions and trees
// the traverse function uses a stack so remember the output will be reversed
        // valid data
        // [2, x, ^] but it will be reversed therefore [^, x, 2]
        System.out.println(a + " -> " + treeA.traverse());
        // valid extreme data
        // [x, 1, +, x, 1, -, *] but it will be reversed therefore [*, -, 1, x, +, 1, x]
        System.out.println(b + " -> " + treeB.traverse());
```

Here are the results:

```
2^x -> [^], x, 2
(x+1)*(x-1) -> [*], -, 1, x, +, 1, x
```

Which is exactly what I expected.

When I designed this algorithm I wrote about how I could implement it using multiple threads. While multi-threading can be very powerful, like everything in Computer Science there is a cost. This is especially true where in `Java` the cost of instantiating a thread[25] (on the fly at least) is incredibly expensive. In the article it talks about three main things:

1. Allocating Memory to the thread

2. Create the call stack for the thread[26, 29]

3. Initialize and link the thread to the host OS

This takes a lot of processing time, in this article[27] one user manages to spawn about 10000 threads a second, which means that it takes approximately 0.1 seconds to spawn every thread. This is to be taken with a pinch of salt however since the article is quite old (8 years in fact) and processors are better and the JVM should be more efficient nowadays than before. To test this for myself I ran their benchmark program. [6] I did change some of the parameters to make each test do the same amount of work (the same number of instructions executed) and made the number of threads spawned different as well. I also ran it 3 times so I could calculate a mean to able to identify anomalous data. This also alleviates some of the randomness of the scheduler. I did this by compiling and executing the program 3 times, to reduce the effect of cached memory affecting the results. While the test does accommodate for this issue and allows for multiple tests, I think it is best to give the JVM no chance for optimization by just destroying and recreating the JVM. The results table and the corresponding graphs are below.

Table 7.1: Thread Creation Test

| Number of Threads | Time Taken/ms | | | | | | | | | | | | Mean Total |
| | To Create | | | | To Complete Work | | | | To Join | | | | |
| | 1 | 2 | 3 | Mean | 1 | 2 | 3 | Mean | 1 | 2 | 3 | Mean | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2.45 | 2.36 | 2.27 | 2.36±0.09 | 0.14 | 0.24 | 0.10 | 0.16±0.07 | 0.44 | 0.20 | 0.20 | 0.28±0.12 | 3.00±0.28 |
| 2 | 2.34 | 2.36 | 2.78 | 2.49±0.22 | 0.17 | 0.15 | 0.14 | 0.15±0.02 | 0.27 | 0.39 | 0.37 | 0.34±0.05 | 2.98±0.29 |
| 5 | 2.49 | 3.04 | 2.29 | 2.61±0.38 | 0.35 | 0.36 | 0.33 | 0.35±0.02 | 0.66 | 0.69 | 0.84 | 0.73±0.09 | 3.69±0.49 |
| 20 | 2.81 | 2.28 | 2.38 | 2.49±0.27 | 1.02 | 1.71 | 0.98 | 1.24±0.37 | 3.11 | 1.03 | 1.93 | 2.02±1.04 | 5.75±1.68 |
| 100 | 2.03 | 2.14 | 2.25 | 2.14±0.11 | 3.26 | 2.77 | 2.97 | 3.00±0.25 | 5.69 | 5.34 | 5.15 | 5.39±0.27 | 10.53±0.63 |
| 1000 | 5.87 | 4.11 | 5.66 | 5.21±0.88 | 47.30 | 42.29 | 39.90 | 43.16±3.70 | 15.22 | 18.61 | 32.20 | 22.01±8.49 | 70.38±13.07 |

From the analysis above it seems that the effect of using a few threads seems minimal and the overhead may be outweighed by the performance gain from using multithreading in other parts of the function. However this may be specific to this scenario, so to test this properly I implemented the multi-threaded version which is on the next page.

---

[6]The code for this is on the StackOverflow thread, made by a user called, at the time of writing this, "Jaan"[27]

```java
1  // create the abstract syntax tree for the expression - multi-threaded
2  private static BinaryTree createTreeThread(String expression)
3              throws StringIndexOutOfBoundsException, UnequalBracketsException,
                ↪  InterruptedException, ExecutionException {
4      // remove enclosing matching brackets
5      expression = checkBracket(expression);
6      // find the least significant operator, if an operator remains
7      int leastSigOperatorPos = leastSigOperatorPos(expression);
8      if (leastSigOperatorPos == -1) { // base case - no operators remain
9              return new BinaryTree(expression);
10     } else { // recursive case - recurse on the sub-expressions
11             // locate and hold the operator
12             String operator = String.valueOf(expression.charAt(leastSigOperatorPos));
13             // split the expression into sub-expressions by the operator and recurse on
                ↪  them
14             String a = expression.substring(0, leastSigOperatorPos);
15             String b = expression.substring(leastSigOperatorPos + 1);
16             BinaryTree tree0;
17             BinaryTree tree1;
18             // create the threads and execute them
19             ExecutorService executor0 = Executors.newSingleThreadExecutor();
20             Future<BinaryTree> future0 = executor0.submit(() -> {
21                     return createTree(a);
22             });
23             ExecutorService executor1 = Executors.newSingleThreadExecutor();
24             Future<BinaryTree> future1 = executor1.submit(() -> {
25                     return createTree(b);
26             });
27             // return the tree values from the threads
28             tree0 = future0.get();
29             tree1 = future1.get();
30             // shutdown the threads
31             executor0.shutdown();
32             executor1.shutdown();
33             // return the new tree containing the trees of the sub-expressions and the
34             // operator as the root value
35             return new BinaryTree(operator, tree0, tree1);
36     }
37 }
```

I then tried to test my 2 `createTree()` functions to identify if a purely single-threaded approach is more efficient than a multi-threaded approach. However the two approaches may change in effectiveness depending on the complexity of the expression we are parsing. In order to control this for a fair test we need to effectively measure the complexity hence we need to analyze the algorithm in detail.

Assuming that a single call of the `createTree()` algorithm takes the same time, $t$, to execute (exclude the recursive calls and the overhead that is need for them), then $\mathcal{O}(n) = n + 1$ where $n$ is the number of operators. Assuming that the expression is valid, this value comes from the fact that we can split a standardized expression into by each operator. From the nature of our approach, there will be no operators on the start or the end of a valid expression, and each variable/constant is split apart by an operator[7]. Now from this property we notice that each operator has a uniquely positioned variable/constant on the right of it, apart from the first operator which has an extra variable/constant to the left of it. Now our function recurses $k$ times where k is the number of base sub-expressions, i.e. variables/constants. However it can be hard to count these variables/constants since they can be of variable length (1.2 is a constant and has a length in terms of strings of 3, but $x$ is also a variable but has a length of 1). On the other hand operators are incredibly easy to spot and count since there is a finite number of types of operators, and they are all of length 1. Hence it makes sense to define the complexity of our function in terms of the number of operators. This means that one of our independent variables, the one we will change, will be the number of operators. We will also change whether we use threads or not since this is the point of our experiment. The code for the testing is below:

```
// create the standardized expressions
String ex = standardize(""); //insert the expression to test in the speech marks
// create the trees
BinaryTree treeA, treeThreadA;
Instant start, end;
// run and time the threaded version
start = Instant.now();
treeThreadA = createTreeThread(ex);
end = Instant.now();
System.out.println("time taken for threaded approach - " + Duration.between(start,
↪   end).toMillis() + "ms");
// run and time the sequential version
start = Instant.now();
treeA = createTree(ex);
end = Instant.now();
System.out.println("time taken for sequential approach - " + Duration.between(start,
↪   end).toMillis() + "ms");
```

I will be testing expressions upto 1000 operators so I used the following code,
```
new String(new char[n+1]).replace("\0", "x");[28]
```
where $n$ is the number of operators needed, to create the expressions automatically. For $n = 5$ this creates the string `"xxxxxx"` and when it is standardized using the previous function it becomes `"x*x*x*x*x*x"`, which produces the expression needed to test. This relies on the fact that all the elements in arrays in `Java` are initialized as **null** and can all be replaced which is what is done.

---

[7]This is an inherent property within any valid expression since otherwise without this we cannot separate each variable/constant hence we cannot define explicitly where each variable/constant starts and ends. We have only applied this to operators and the idea becomes blurred when we include functions within our expression, such as trigonometric or logarithmic functions, but this does not matter since our algorithm does not currently does not deal with this.

Table 7.2: Sequential vs Multi-Threaded Approach Test

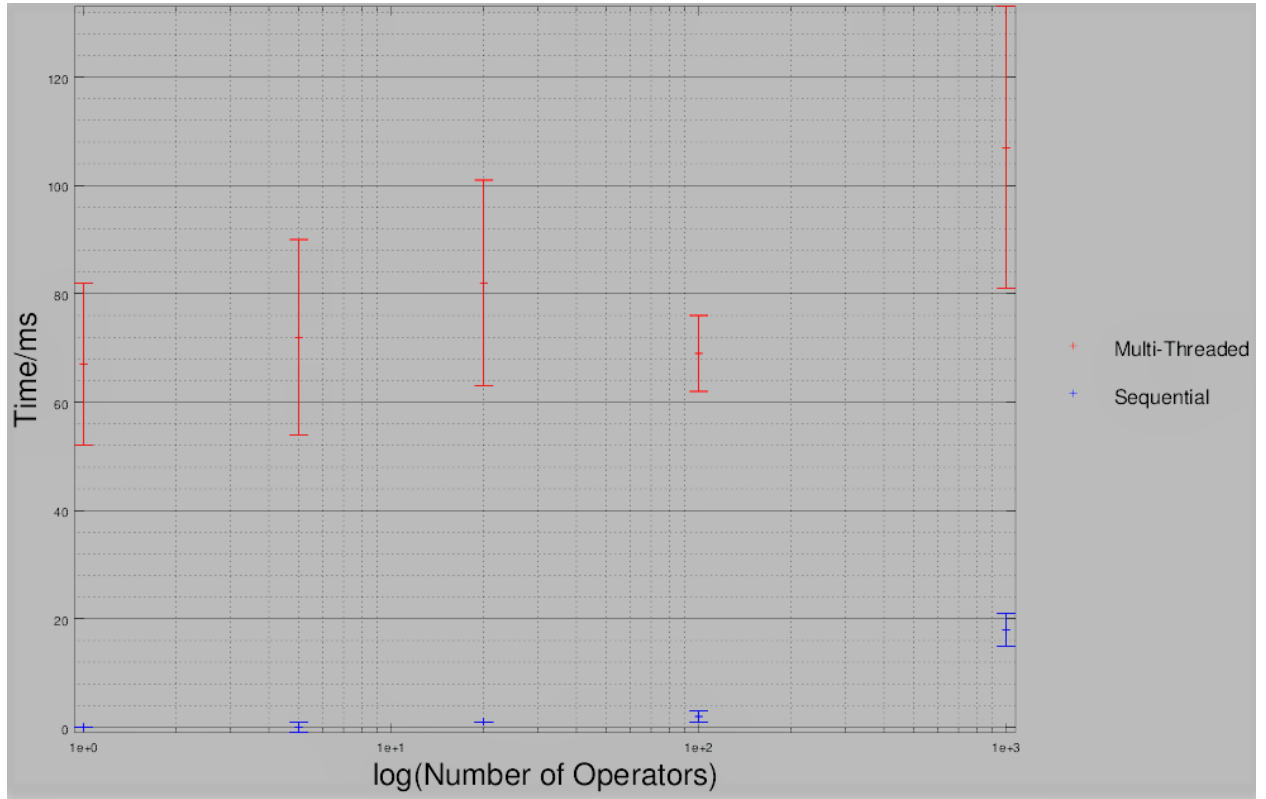| Number of Operators | Time Taken/ms | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | With Threads | | | | Without Threads | | | |
| | 1 | 2 | 3 | Mean | 1 | 2 | 3 | Mean |
| 1 | 67 | 81 | 52 | 67±15 | 0 | 0 | 0 | 0±0 |
| 5 | 96 | 61 | 60 | 72±18 | 1 | 0 | 0 | 0±1 |
| 20 | 90 | 97 | 59 | 82±19 | 1 | 1 | 1 | 1±0 |
| 100 | 77 | 66 | 64 | 69±7 | 3 | 2 | 2 | 2±1 |
| 1000 | 100 | 124 | 97 | 107±26 | 16 | 22 | 16 | 18±3 |



Figure 7.1: Plot of the Table Above

When representing the data I collected in a scatter graph, I chose to use a logarithmic scale for the number of threads since the number of threads I used had a very large variation in value, ranging from 1 to 1000. From analyzing the graph, it is clear that using threads took much more time than the normal sequential approach, in some cases over a 6000% increase in time taken. However does this sort of relationship between a sequential and threaded approach continue for a larger number of operators?

Looking at the graph more closely it appears as though there is some kind of linear relationship between time and the logarithm of the number of operators (obviously ignoring the anomalous data at 100 operators). From using regression tools this relationship comes out as $T = 42.118 \cdot \ln n + 67$[8].

Similarly the relationship between the sequential version seems to be exponential on the graph so it is probably a linear relationship between time and the number of operators (not the logarithm this time). From using regression tools this relationship comes out as $T = 0.0178 \cdot n + 0.1858$[9].

Since a linear function increases quicker than a logarithmic one, if we find the point of intersection, we will find the theoretic point[10] at which the threaded version outperforms, i.e. takes less time than, the sequential one. By using numerical method tools this comes out to be at $n \approx 28000$. Realistically we would never have this many operators in a simple function to sketch, however from a programming point of view, this is also ridiculous. From a call stack and memory point of view, this would require about 28000 recursive calls which not only is memory intensive and quite likely cause a stack overflow error[30], it will consume even more memory if using threads. Just to test if it would cause a stack overflow I tried to an example with 28000 operators and I got the following output:

```
Exception in thread "main" java.util.concurrent.ExecutionException:
↪   java.lang.StackOverflowError
        at java.util.concurrent.FutureTask.report(FutureTask.java:122)
        at java.util.concurrent.FutureTask.get(FutureTask.java:192)
        at structures.Expression.createTreeThread(Expression.java:149)
        at structures.Expression.main(Expression.java:284)
Caused by: java.lang.StackOverflowError
        at java.util.regex.Pattern.compile(Pattern.java:1054)
        at java.lang.String.replace(String.java:2239)
        at structures.Expression.checkBracket(Expression.java:197)
        at structures.Expression.createTree(Expression.java:105)
        at structures.Expression.createTree(Expression.java:118)
```

It is therefore clear that using threads in this part of the program and quite possibly in other parts of the program where the number of threads required to spawn is high, is incredibly foolish and not worth the increased effort to implement.

---

[8]This has a PMCC of $r = 0.9937$ so this is very good correlation

[9]This has a PMCC of $r = 0.9993$ so this is very good correlation

[10]It is important to realize that the sample size used here is small, but still legitimate so we can make careful extrapolations.

### 7.3.5 Constructor

Now that the `createTree()` function is implemented, the constructor can finally be implemented. Along with the constructor, I will override the `(toString()` method so that I can view the contents of an object of this type by calling a single method. It keeps the attributes private, and removes the need to have getters and setters for those attributes just to view them. However the most important role it fulfills is avoiding the repetition of code whenever we need to view the inside of this object, which is a good programming practice that also makes our code more organized and clean.

```
1   // constructor
2   public Expression(String expression, char parameter) throws StackOverflowException,
    ↪   StackUnderflowException, UnequalBracketsException {
3           this.parameter = parameter; // sets the parameter value
4           this.expression = standardize(expression); // removes whitespace + standardizes
5           BinaryTree tree = createTree(this.expression); // creates the tree
6           this.postFixStack = tree.traverse(); // creates the post-fix stack
7   }
8
9   @Override
10  // output the stack and the expression this object represents
11  public String toString() {
12          return this.expression + " -> " + this.postFixStack.toString();
13  }
```

To test these methods I used the following code with comments for what I expected:

```
Expression ex = new Expression("2x", 'x');
        // valid data
        System.out.println(ex);        // expected result: 2*x -> [*], x, 2
```

Here are the results:

```
2*x -> [*], x, 2
```

Which is exactly what I expected.

### 7.3.6 Substitute

This function is linked to the class so it will not be `static`, however it will only be used within the class, specifically within the `evaluate()` function, so it will be `private`.

```java
// substitute a value of the parameter into the expression
private Stack<String> substitute(double x) throws StackUnderflowException,
    StackOverflowException {
        Stack<String> copy = new Stack<String>(this.postFixStack);          //copy the post fix
            stack so we don't edit it
        Stack<String> subStack = new
            Stack<String>(this.postFixStack.getHeight());          //create a stack to store the
            substituted value of the same size as the post-fix stack
        for (int i = copy.getHeight(); i > 0; i--) {          //loop through the copied stack
                String pop = copy.pop();          //pop an item off and store it in a variable
                if (pop.equals(String.valueOf(this.parameter))) {          //check if it is a
                    variable if so replace it with the value
                    pop = Double.toString(x);
                }
                subStack.push(pop);          //push the value onto the new stack
        }
        return subStack;          //return the new stack
}
```

To test these methods I used the following code with comments for what I expected:

```java
Expression ex = new Expression("2x", 'x');
        // valid data
        System.out.println(ex.substitute(2));          // expected result: [2], 2.0, *
ex = new Expression("2eˆx", 'x');
        // valid extreme data
        System.out.println(ex.substitute(10));          // expected result: [2], (value for e
            approximately 2.718), 10.0, ˆ, *
```

Here are the results:

```
[2], 2.0, *
[2], 2.718281828459045, 10.0, ˆ, *
```

Which is exactly what I expected.

### 7.3.7 Evaluate

This function is linked to the class so it will not be `static`, however it will only be used outside the class since other modules in our program will need to know the value of the expression at a specific value, so it will be `public`. There are some language specific things that I added to this method. If the current value I have popped of the stack is not an operator, the algorithm casts it to an integer. However this can cause issues, since if we have an expression $xy$, this contains two parameters but we take only one parameter. So say our parameter is $x$, then the $y$ will be treated as a number since it is not an operator, and it will not have been substituted for a value. So when we cast this value we will get an error. To remedy this, my method will be flagged as throwing the related exception `NumberFormatException`. The usual problems in arithmetic such as dividing by 0, are handled by `Java double` primitive and the IEEE standard[31, 32].

```java
// evaluate the expression for a value of the parameter
public double evaluate(double x) throws StackUnderflowException, StackOverflowException,
    NumberFormatException {
        // create a stack to store the numerical values
        // and the substitute stack
        Stack<String> subStack = substitute(x);
        Stack<Double> numStack = new Stack<Double>(this.postFixStack.getHeight());
        for (int i = subStack.getHeight(); i > 0; i--) {
                String pop = subStack.pop(); // pop a value of the substitute stack and store
                    it
                double a, b; // initialize two double variables for potential calculation
                switch (pop) { // check if the popped value is a number or operator
                // if it is an operator
                // do the operation on the two numbers at the top of the number stack
                // and replace them both with the result
                case "+":
                        b = numStack.pop();
                        a = numStack.pop();
                        numStack.push(a + b);
                        break;
                case "-":
                        b = numStack.pop();
                        a = numStack.pop();
                        numStack.push(a - b);
                        break;
                case "*":
                        b = numStack.pop();
                        a = numStack.pop();
                        numStack.push(a * b);
                        break;
                case "/":
                        b = numStack.pop();
                        a = numStack.pop();
                        numStack.push(a / b);
                        break;
                case "^":
                        b = numStack.pop();
                        a = numStack.pop();
                        numStack.push(Math.pow(a, b));
                        break;
```

```
39                    // if it is a number push it onto the number stack
40                default:
41                        // try to cast the popped value, it is of type String, to double
42                        // this may fail so the function throws NumberFormatException
43                        // this will fail if there are multiple parameters e.g. "xy"
44                        // since it will treat the y as a number which it is not
45                        numStack.push(Double.valueOf(pop));
46                        break;
47                }
48            }
49            // pop and return the remaining value in the number stack, this is the evaluated value
50            return numStack.pop();
51    }
```

To test these methods I used the following code with comments for what I expected:

```
Expression ex1 = new Expression("2x + 4", 'x');
        // valid data
        System.out.println(ex1.evaluate(3));        // expected value: 10
Expression ex2 = new Expression("e^a", 'a');
        // valid data - using constants
        System.out.println(ex2.evaluate(0));        // expected value: 1
Expression ex3 = new Expression("1/b", 'b');
        // valid extreme data
        System.out.println(ex3.evaluate(0));        // expected value: infinity
Expression ex4 = new Expression("c/c", 'c');
        // valid extreme data
        System.out.println(ex4.evaluate(0));        // expected value: NaN
Expression ex5 = new Expression("xy", 'x');
        // invalid data
        System.out.println(ex5.evaluate(2));        // expected value: an exception will be
        ↪   thrown since there are multiple variables
```

Here are the results:

```
10.0
1.0
Infinity
NaN
Exception in thread "main" java.lang.NumberFormatException: For input string: "y"
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at java.lang.Double.valueOf(Double.java:502)
        at structures.Expression.evaluate(Expression.java:76)
```

Which is exactly what I expected.

## 7.4 Functions

At the moment we only need a class for explicit functions, however to illustrate the reason behind creating a separate class I will also create a quick Normal Distribution function that extends our class for explicit functions. This will be more useful later on when we implement layers.

### 7.4.1 Explicit Function

This class will essentially be a wrapper, a layer to manage the expression objects underneath.

```java
package structures;

import exceptions.StackOverflowException;
import exceptions.StackUnderflowException;
import exceptions.UnequalBracketsException;

public class ExplicitFunction {

        // attributes
        private Expression f; // the actual mathematical function

        // methods
        // empty constructor to allow for inheritance
        public ExplicitFunction() {
        }

        // constructor
        public ExplicitFunction(String expression, char parameter) throws
        ↪   StackOverflowException, StackUnderflowException, UnequalBracketsException {
                this.setF(new Expression(expression, parameter));
        }

        // evaluate method calls the expression evaluate method
        public double evaluate(double x) {
                // try to evaluate the expression
                //if it fails simply return NaN and the associated error
                try {
                        return f.evaluate(x);
                } catch (StackUnderflowException e) {
                        System.out.println("underflow");
                } catch (StackOverflowException e) {
                        System.out.println("overflow");
                } catch (NumberFormatException e) {
                        System.out.println("multiple parameters");
                }
                return Double.NaN;
        }




```

```
41          // set the expression attribute - used for child classes
42          protected void setF(Expression f) {
43                  this.f = f;
44          }
45
46          @Override
47          // output the representation of the expression object
48          public String toString() {
49                  return this.f.toString();
50          }
51
52  }
```

To test this class I used the following code with comments for what I expected:

```
ExplicitFunction f = new ExplicitFunction("x^2",'x');
        // valid data
        System.out.println(f.evaluate(2.5));        // expected result: 6.25
```

Here are the results:

```
6.25
```

Which is exactly what I expected.

### 7.4.2   Normal Distribution Function

This class will inherit the Explicit Function class and will basically have its own expression that we will substitute the parameters, mean, $\mu$, and variance, $\sigma^2$, into.

```
1   package structures;
2
3   import exceptions.StackOverflowException;
4   import exceptions.StackUnderflowException;
5   import exceptions.UnequalBracketsException;
6
7   public class NormalDistribution extends ExplicitFunction {
8
9           // attributes
10          private double ; // mean
11          private double 2; // variation
12
13          // methods
14          // constructor
15          public NormalDistribution(double , double 2)
16                          throws StackOverflowException, StackUnderflowException,
                          ↪ UnequalBracketsException {
17                  // super constructor must be called first in Java
18                  super();
19
20                  // set the attributes
21                  this. = ;
```

```
22                    this.2 = 2;
23
24                    // create the expression for the normal distribution function
25                    String expression = "(1/(2*pi*2)^(1/2))*e^(-(x-)^2/(2*2^(1/2)))";
26                    // substitute the mean and variance values
27                    expression = expression.replace("2", Double.toString(2));
28                    expression = expression.replace("", Double.toString());
29
30                    // set the new expression
31                    this.setF(new Expression(expression, 'x'));
32            }
33
34        // return the mean
35        public double get() {
36                    return ;
37            }
38
39        // return the variance
40        public double get2() {
41                    return 2;
42            }
43
44        // return the standard deviation
45        public double get() {
46                    return Math.sqrt(2);
47            }
48
49  }
```

To test this class I used the following code with comments for what I expected:

```
ExplicitFunction f = new NormalDistribution(0,1);
        // valid data
        System.out.println(f.evaluate(1));          // expected value: 0.24197 to 5 s.f.
```

Here are the results:

0.24197072451914337

Which is exactly what I expected.

67

## 7.5 Layers

As described in the design section, I will create classes that inherit from one a top class that I will call `Layer`. This class will be `abstract` since we will never instantiate this class and only inherit from it. The `CartesianLayer` will be a child class to the abstract `Layer` class. Again this will be `abstract` since we will only inherit from it. While it seems redundant for the super class `Layer` to exist, it fulfills the purpose of being able to create an array of type `Layer` which can then be filled with objects that are of types which are child classes to `Layer`. Our first instantiatable class will be a child class to `CartesianLayer` and will be called `ExplicitXFunctionCartesianLayer`. The X signifies that the parameter will be $x$. Likewise we will have a class called `ExplicitYFunctionCartesianLayer`, where this will be a function in terms of $y$. These two function classes are very similar, they both use algorithm 16 on page 36 but one uses the primary variable $x$ and the other uses the primary variable $y$. Our `AxesCartesianLayer` will also inherit from the `CartesianLayer` parent class. This makes actually drawing the axes easier, since we can simply draw 2 lines using the methods in the parent class, `CartesianLayer`.

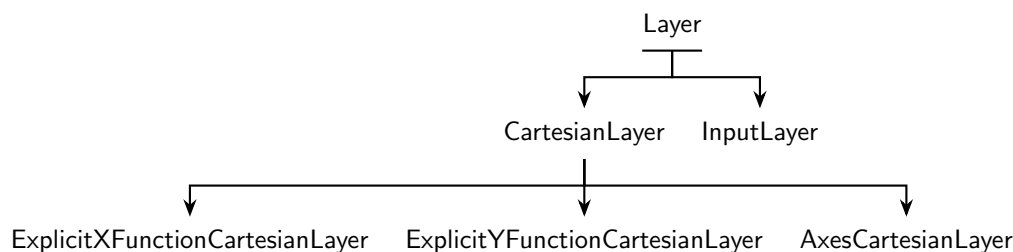A diagram of this inheritance is shown below:



Figure 7.2: `Layer` Inheritance Diagram

The problem with this part of the program is that everything is interdependent on everything else. Specifically here the `PlotPane` class is used within the `Layer` classes and vice versa. Since I will implement the `Layer` classes first I will create a bare bones `PlotPane` class that just contains its attributes for use within the `Layer` classes and implement the `PlotPane` class later. This is simply to avoid compiler errors. This also means that we cannot properly test this portion until we complete both parts. As such I will combine the mini tests that I would have carried out, as a part of the overall final testing for this prototype.

### 7.5.1 Layer

This is an abstract super class, as described in the design. Since we cannot instantiate this class, we cannot directly test this class, but by testing its child classes we can test it indirectly. I added an the attribute `Color` and setter for it, which controls what the color of anything drawn onto the canvas is.

```java
package layer;

import application.PlotPane;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public abstract class Layer {

        // attributes
        // drawing tools
        protected Canvas canvas;
        protected GraphicsContext gc;
        protected Color color = Color.BLACK;
        // drawing parameters
        protected IntegerProperty steps = new SimpleIntegerProperty(0);
        protected DoubleProperty minX = new SimpleDoubleProperty(0);
        protected DoubleProperty maxX = new SimpleDoubleProperty(0);
        protected DoubleProperty minY = new SimpleDoubleProperty(0);
        protected DoubleProperty maxY = new SimpleDoubleProperty(0);
        protected DoubleProperty pixelWorthX = new SimpleDoubleProperty(0);
        protected DoubleProperty pixelWorthY = new SimpleDoubleProperty(0);

        // methods
        // constructor
        protected Layer() {
                // instantiate the canvas and create the OpenGL context
                this.canvas = new Canvas();
                this.gc = canvas.getGraphicsContext2D();
        }

        // will draw onto the canvas, will be overridden
        public abstract void draw();

        // will connect this layer to a PlotPane instance, will be overridden
        public abstract void bindProperties(PlotPane plotPane);

        // clear the canvas to be redrawn
        protected void clearCanvas() {
                gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
        }

        // getter for the canvas
        public Canvas getCanvas() {
                return canvas;
        }

        // change the color of the function drawn
        public void setColor(Color color) {
                this.color = color;
        }

}
```

### 7.5.2 Input Layer

This class, unlike the previous, is not an abstract class. It was quite simple to implement with the algorithms 13, 17, 18 and 19 on pages 35 and 37. In the bindProperties() method I initialized the values for the minimum and maximum values of $x$ and $y$ to go from $-10$ to 10.

```java
package layer;

import java.text.DecimalFormat;

import application.PlotPane;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class InputLayer extends Layer {

        // attributes
        private BooleanProperty changeViewport = new SimpleBooleanProperty(true);

        // methods
        // constructor
        public InputLayer() {
                // call the super constructor
                super();
                //update pixel worth when the canvas changes size
                this.canvas.heightProperty().addListener(event -> updatePixelWorth());
                this.canvas.widthProperty().addListener(event -> updatePixelWorth());
        }


        @Override
        // this canvas will not draw anything so this method is empty
        // it must still be implemented since the method was abstract in the parent class
        public void draw() {}

        // update the pixelworth
        private void updatePixelWorth() {
                this.pixelWorthX.set((this.maxX.doubleValue() - this.minX.doubleValue()) /
                ↪   this.canvas.getWidth());
                this.pixelWorthY.set((this.maxY.doubleValue() - this.minY.doubleValue()) /
                ↪   this.canvas.getHeight());
        }

        @Override
        // initialize the properties required so they can be bound later
        public void bindProperties(PlotPane plotPane) {
                // initialize the properties
                this.steps = new SimpleIntegerProperty(400);
                this.minX = new SimpleDoubleProperty(-10);
                this.maxX = new SimpleDoubleProperty(10);
                this.minY = new SimpleDoubleProperty(-10);
                this.maxY = new SimpleDoubleProperty(10);
                this.pixelWorthX = new SimpleDoubleProperty(1);
                this.pixelWorthY = new SimpleDoubleProperty(1);
                // make the canvas resize as its parent resizes by binding the associated properties
                this.canvas.heightProperty().bind(plotPane.heightProperty());
                this.canvas.widthProperty().bind(plotPane.widthProperty());
        }

        //return minX
        public DoubleProperty getMinX() {
                return minX;
        }
```

```
60
61        //return minX
62        public DoubleProperty getMaxX() {
63               return maxX;
64        }
65
66        // return minX
67        public DoubleProperty getMinY() {
68               return minY;
69        }
70
71        // return minX
72        public DoubleProperty getMaxY() {
73               return maxY;
74        }
75
76        // return the worth of each pixel in the x-direction
77        public DoubleProperty getPixelWorthX() {
78               return pixelWorthX;
79        }
80
81        // return the worth of each pixel in the y-direction
82        public DoubleProperty getPixelWorthY() {
83               return pixelWorthY;
84        }
85
86        // return the number of draw steps
87        public IntegerProperty getSteps() {
88               return steps;
89        }
90
91        // return the property that notifies when it is time to redraw layers
92        public BooleanProperty getChangeViewport() {
93               return changeViewport;
94        }
95
96  }
```

### 7.5.3 Cartesian Layer

This class is also abstract and hence cannot be instantiated. It will contain methods to convert Cartesian coordinates to the Canvas coordinate system and bind its properties to a `PlotPane`. The algorithms for these are on page 35. Since all the classes that inherit this class will use the same properties I will finally override the `bindProperties` method and bind the necessary properties to reduce the amount of redundant code and ensure consistency in how the properties are bound.

```java
package layer;

import application.PlotPane;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleIntegerProperty;

public abstract class CartesianLayer extends Layer {

        @Override
        // must be overriden since the parent class made it abstract
        // but keep it abstract so child class must override
        public abstract void draw();

        @Override
        // bind the needed properties from the input layer to itself
        public void bindProperties(PlotPane plotPane) {
                // initialize the properties with the same values as their partners
                this.steps = new SimpleIntegerProperty(plotPane.getInputLayer().getSteps().intValue());
                this.minX = new SimpleDoubleProperty(plotPane.getInputLayer().getMinX().doubleValue());
                this.maxX = new SimpleDoubleProperty(plotPane.getInputLayer().getMaxX().doubleValue());
                this.minY = new SimpleDoubleProperty(plotPane.getInputLayer().getMinY().doubleValue());
                this.maxY = new SimpleDoubleProperty(plotPane.getInputLayer().getMaxY().doubleValue());
                this.pixelWorthX = new
                ↪  SimpleDoubleProperty(plotPane.getInputLayer().getPixelWorthX().doubleValue());
                this.pixelWorthY = new
                ↪  SimpleDoubleProperty(plotPane.getInputLayer().getPixelWorthY().doubleValue());
                // bind the two properties together one-directionally
                // so these attributes change when plot pane changes, not vice versa
                this.steps.bind(plotPane.getInputLayer().getSteps());
                this.minX.bind(plotPane.getInputLayer().getMinX());
                this.maxX.bind(plotPane.getInputLayer().getMaxX());
                this.minY.bind(plotPane.getInputLayer().getMinY());
                this.maxY.bind(plotPane.getInputLayer().getMaxY());
                this.pixelWorthX.bind(plotPane.getInputLayer().getPixelWorthX());
                this.pixelWorthY.bind(plotPane.getInputLayer().getPixelWorthY());
                // make the canvas resize as its parent resizes by binding the associated properties
                canvas.heightProperty().bind(plotPane.heightProperty());
                canvas.widthProperty().bind(plotPane.widthProperty());
        }

        // convert x-Cartesian coordinate to x-Canvas coordinate
        protected double convertX(double x) {
                x = x - this.minX.doubleValue();
                x = x / this.pixelWorthX.doubleValue();
                return x;
        }

        // convert y-Cartesian coordinate to y-Canvas coordinate
        protected double convertY(double y) {
                y = this.maxY.doubleValue() - y;
                y = y / this.pixelWorthY.doubleValue();
                return y;
        }

}
```

### 7.5.4 Axes Cartesian Layer

Drawing the axes is quite simple so we will begin our implementation of non-abstract drawing classes with this one. This class will draw whenever the $x$ or $y$ coordinate is 0. This results in two lines. The $x$ axis will be drawn from the leftmost point where $y$ is 0 to the rightmost point where $y$ is (rightmost and leftmost on the canvas). This means that we draw a line from the point $(x_{min,0})$ to the point $(x_{max,0})$. These $x_min$ and $x_max$ values are values that we have stored as properties within our class. Similarly we can draw the $y$ axis by drawing a line from the point $(0, y_{min})$ to the point $(0, y_{max})$. These Cartesian coordinates must be first converted to the canvas coordinates, and this will be done in the implementation.

```java
package layer;

public class AxesCartesianLayer extends CartesianLayer {

        // constructor
        public AxesCartesianLayer() {
                // call the super to setup the canvas
                super();
        }

        @Override
        // draw the pair of axes
        public void draw() {
                // clear the canvas before drawing
                this.clearCanvas();
                // make the line width thinner then usual to remove emphasis from axes
                gc.setLineWidth(1.5);
                gc.setStroke(color);
                // draw the y-axis
                gc.strokeLine(this.convertX(0), this.convertY(this.minY.doubleValue()),
                ↪    this.convertX(0),
                                this.convertY(this.maxY.doubleValue()));
                // draw the x-axis
                gc.strokeLine(this.convertX(this.minX.doubleValue()), this.convertY(0),
                                this.convertX(this.maxX.doubleValue()), this.convertY(0));
        }

}
```

### 7.5.5 Explicit $x$-Function Cartesian Layer

While designing the algorithm to draw functions I talked about how the function must be continuous otherwise we would come across arithmetic errors such dividing by 0. This is no longer an issue since the `double` will handle all these errors for us. This was mentioned during the implementation of the evaluation of a function. Apart from that resolved issue there were no other issues during the implementation of this class.

```java
package layer;

import exceptions.StackOverflowException;
import exceptions.StackUnderflowException;
import exceptions.UnequalBracketsException;
import structures.ExplicitFunction;

public class ExplicitXFunctionCartesianLayer extends CartesianLayer {

        // attributes
        private ExplicitFunction f;

        // methods
        // constructor - create a layer with an expression given
        public ExplicitXFunctionCartesianLayer(String expression) {
                // super constructor must be invoked first
                super();
                // try to create the function with the given expression
                try {
                        this.f = new ExplicitFunction(expression, 'x');
                } catch (StackOverflowException | StackUnderflowException |
                ↪   UnequalBracketsException e) {
                        // output the error in parsing the function
                        e.printStackTrace();
                }
        }

        // constructor - create a layer with a set function
        public ExplicitXFunctionCartesianLayer(ExplicitFunction f) {
                // super constructor must be invoked first
                super();
                // set the function
                this.f = f;
        }

        // draw the function
        public void draw() {
                // setup the canvas by clearing it and set up the line to be drawn
                this.clearCanvas();
                gc.setLineWidth(2);
                gc.setStroke(color);
                double x1, x2, y1, y2;
                // initialize the first value of x
                x1 = this.minX.doubleValue();
```

```
44              // evaluate the first value of y
45              y1 = f.evaluate(x1);
46              // calculate the dx value i.e. the step value
47              double step = (this.maxX.doubleValue() - this.minX.doubleValue()) /
   ↪  this.steps.doubleValue();
48              // loop until we draw everything in the range, minX to minX, required
49              for (x2 = this.minX.doubleValue() + step; x2 < this.maxX.doubleValue(); x2 =
   ↪  x2 + step) {
50                      // evaluate the next value of y
51                      y2 = f.evaluate(x2);
52                      // draw the line
53                      gc.strokeLine(this.convertX(x1), this.convertY(y1), this.convertX(x2),
   ↪  this.convertY(y2));
54                      // update the old values
55                      x1 = x2;
56                      y1 = y2;
57              }
58      }
59
60      @Override
61      // output the function that this layer is drawing
62      public String toString() {
63              return this.f.toString();
64      }
65
66 }
```

### 7.5.6   Explicit $y$-Function Cartesian Layer

This is essentially a copy of the previous layer but we are looping over the $y$ variable instead of the $x$ variable. As such I simply replaced every instance of $X$ with a $Y$ and vice versa.

```
1  package layer;
2
3  import exceptions.StackOverflowException;
4  import exceptions.StackUnderflowException;
5  import exceptions.UnequalBracketsException;
6  import structures.ExplicitFunction;
7
8  public class ExplicitYFunctionCartesianLayer extends CartesianLayer {
9
10      // attributes
11      private ExplicitFunction f;
12
13      // methods
14      // constructor - create a layer with an expression given
15      public ExplicitYFunctionCartesianLayer(String expression) {
16              // super constructor must be invoked first
17              super();
18              // try to create the function with the given expression
```

```
19              try {
20                      this.f = new ExplicitFunction(expression, 'y');
21              } catch (StackOverflowException | StackUnderflowException |
       ↪  UnequalBracketsException e) {
22                      // output the error in parsing the function
23                      e.printStackTrace();
24              }
25          }
26
27      // constructor - create a layer with a set function
28      public ExplicitYFunctionCartesianLayer(ExplicitFunction f) {
29              // super constructor must be invoked first
30              super();
31              // set the function
32              this.f = f;
33          }
34
35      // draw the function
36      public void draw() {
37              // setup the canvas by clearing it and set up the line to be drawn
38              this.clearCanvas();
39              gc.setLineWidth(2);
40              gc.setStroke(color);
41              double x1, x2, y1, y2;
42              // initialize the first value of y
43              y1 = this.minY.doubleValue();
44              // evaluate the first value of x
45              x1 = f.evaluate(y1);
46              // calculate the dy value i.e. the step value
47              double step = (this.maxY.doubleValue() - this.minY.doubleValue()) /
       ↪  this.steps.doubleValue();
48              // loop until we draw everything in the range, minY to maxY, required
49              for (y2 = this.minY.doubleValue() + step; y2 < this.maxY.doubleValue(); y2 =
       ↪  y2 + step) {
50                      // evaluate the next value of y
51                      x2 = f.evaluate(y2);
52                      // draw the line
53                      gc.strokeLine(this.convertX(x1), this.convertY(y1), this.convertX(x2),
       ↪  this.convertY(y2));
54                      x1 = x2;
55                      y1 = y2;
56              }
57          }
58
59      @Override
60      // output the function that this layer is drawing
61      public String toString() {
62              return this.f.toString();
63          }
64
65  }
```

## 7.6   Plot Pane

The `PlotPane` was a bit difficult to implement. It required me to learn about lists in Java[33] and about what listeners[34] were and how they are used. Lists are not primitives in Java but an implementation of them are in standard libraries. Listeners are bits of code which are executed when an event occurs. An example of an event would be when a property changes value or if a button is presses. Listeners can be bound to any property so that when a property changes value the listener is run. In our implementation listeners are used to redraw the layers whenever the pane changes size.

When I decided whether to use a layer system or not, I mentioned how using threads to draw multiple canvases at once is an advantage. However from our tests while implementing our `Expression` class, we discovered that creating threads in Java is very expensive and as such it would do more harm than good to implementing a multithreaded system to draw multiple canvases at once.

Again we cannot test this component individually until the very end of this prototype where we will utilize our test plan which we designed in the design section. The code for this class is shown below and is commented to explain some of the less obvious parts:

```
1    package application;
2
3    import java.util.ArrayList;
4
5    import exceptions.StackOverflowException;
6    import exceptions.StackUnderflowException;
7    import exceptions.UnequalBracketsException;
8    import javafx.beans.property.BooleanProperty;
9    import javafx.beans.property.SimpleBooleanProperty;
10   import javafx.beans.value.ChangeListener;
11   import javafx.scene.layout.Pane;
12   import layer.AxesCartesianLayer;
13   import layer.ExplicitXFunctionCartesianLayer;
14   import layer.InputLayer;
15   import layer.Layer;
16
17   public class PlotPane extends Pane {
18
19           // attributes
20           // layers
21           private ArrayList<Layer> layers = new ArrayList<Layer>();
22           private InputLayer inputLayer;
23           private AxesCartesianLayer axesLayer;
24           // properties
25           private BooleanProperty changeViewport = new SimpleBooleanProperty(true);
26
27           // methods
28           // constructor
29           public PlotPane() {
30                   // initialize the input layer and bind its properties
31                   this.inputLayer = new InputLayer();
32                   this.inputLayer.bindProperties(this);
33                   // bind the plotpane's properties to it
34                   this.changeViewport.bind(this.inputLayer.getChangeViewport());
35                   // make the background of the plotpane white
36                   this.setStyle("-fx-background-color: rgb(255,255,255)");
37                   // this is portion of code which can be bound to a property
38                   // and will run when that property changes
39                   // this listener will redraw the layers
40                   ChangeListener<Object> redrawListener = (observable, oldValue, newValue) -> {
41                           try {
42                                   drawAll();
```

```
43              } catch (StackOverflowException | StackUnderflowException |
   ↪    UnequalBracketsException
44                      | InterruptedException e) {
45                  // error when drawing functions
46                  e.printStackTrace();
47              }
48          };
49          // add the redraw listener to the size properties
50          // so when the plotpane changes size the layers are redrawn
51          this.heightProperty().addListener(redrawListener);
52          this.widthProperty().addListener(redrawListener);
53          // add the redraw listener to the changeViewport property
54          // so when the input layer notifies to draw, the layers are redrawn
55          this.changeViewport.addListener(redrawListener);
56          // initialize the axes layer and bind its properties
57          this.axesLayer = new AxesCartesianLayer();
58          axesLayer.bindProperties(this);
59      }
60
61      private void drawAll()
62              throws StackOverflowException, StackUnderflowException,
   ↪    UnequalBracketsException, InterruptedException {
63          // loop through every layer and draw it
64          for (Layer l : layers) {
65              l.draw();
66          }
67          // draw the axes
68          axesLayer.draw();
69          // remove all the canvases in the plotpane
70          this.getChildren().clear();
71          // readd all the canvases in the plotpane
72          for (Layer l : layers) {
73              this.getChildren().add(l.getCanvas());
74          }
75          // add the input and axes canvases again
76          // with the input layer on top so that it can receive input
77          this.getChildren().add(axesLayer.getCanvas());
78          this.getChildren().add(inputLayer.getCanvas());
79      }
80
81      // add a layer
82      private void addLayer(Layer l) {
83          // bind the properties to this plotpane
84          l.bindProperties(this);
85          // add the layer to the list
86          this.layers.add(l);
87      }
88
89      // return the inputlayer to access its properties
90      public InputLayer getInputLayer() {
91          return inputLayer;
92      }
93
94  }
```

## 7.7 Main Border Pane

When I created a `JavaFX` project in the Eclipse IDE, it automatically created a main pane and a template for initializing the scene and loading the pane. I added a few lines of code to control how big the window will originally be.

```java
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.fxml.FXMLLoader;

public class Main extends Application {

        // methods
        @Override
        // a method to start the gui
        public void start(Stage primaryStage) {
                try {
                        // load the main FXML file
                        BorderPane root = (BorderPane)
                        ↪   FXMLLoader.load(getClass().getResource("Main.fxml"));
                        // create the scene with the borderpane as the root node
                        Scene scene = new Scene(root);
                        scene.getStylesheets().add(getClass()
                                .getResource("application.css").toExternalForm());
                        primaryStage.setScene(scene);
                        // make the window 1020 x 720 big
                        primaryStage.setHeight(720);
                        primaryStage.setWidth(1020);
                        // show the window
                        primaryStage.show();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        // run the program
        public static void main(String[] args) {
                // launch the gui
                launch(args);
        }
}
```

After I edited this main class, I opened the main `BorderPane FXML` file in scene builder. I removed all the automatically generated regions and gave the main pane a `fxid` which basically allows, me to access it in the controller class.
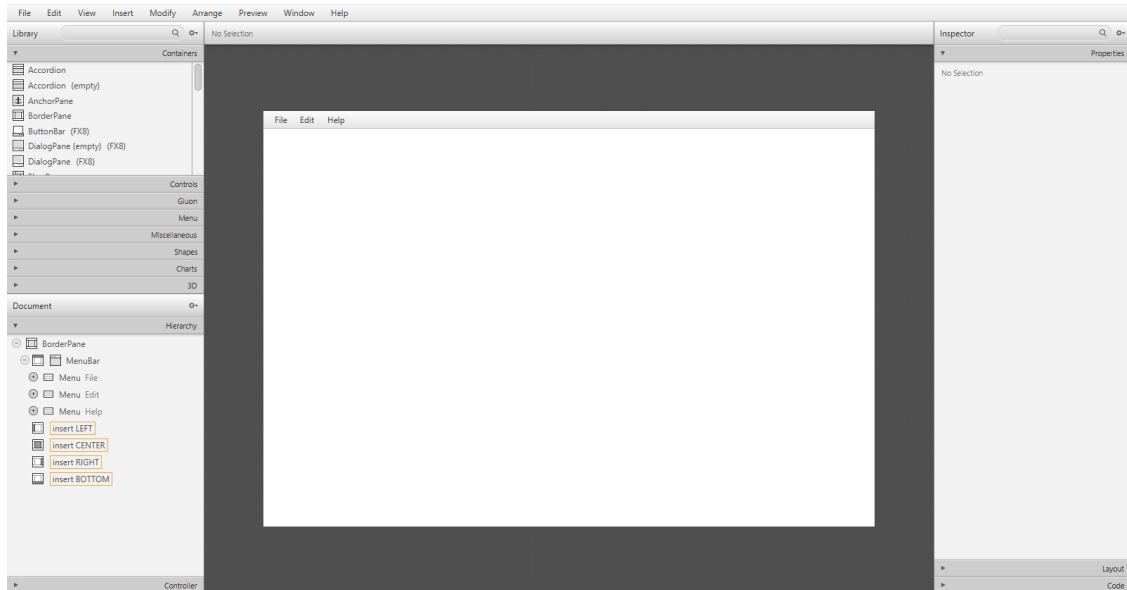
Figure 7.3: Creating the Main Pane in Scene Builder

The `FXML` for this pane is shown below:

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <?import javafx.scene.control.Menu?>
4   <?import javafx.scene.control.MenuBar?>
5   <?import javafx.scene.control.MenuItem?>
6   <?import javafx.scene.layout.BorderPane?>
7
8   <BorderPane fx:id="rootPane" prefHeight="590.0"
9           prefWidth="870.0" xmlns="http://javafx.com/javafx/8.0.141"
10          xmlns:fx="http://javafx.com/fxml/1"
11          fx:controller="application.MainController">
12          <top>
13                  <MenuBar BorderPane.alignment="CENTER">
14                          <menus>
15                                  <Menu mnemonicParsing="false" text="File">
16                                          <items>
17                                                  <MenuItem mnemonicParsing="false" text="Close" />
18                                          </items>
19                                  </Menu>
20                                  <Menu mnemonicParsing="false" text="Edit">
21                                          <items>
22                                                  <MenuItem mnemonicParsing="false" text="Delete" />
23                                          </items>
24                                  </Menu>
25                                  <Menu mnemonicParsing="false" text="Help">
26                                          <items>
27                                                  <MenuItem mnemonicParsing="false" text="About" />
28                                          </items>
29                                  </Menu>
30                          </menus>
31                  </MenuBar>
32          </top>
33  </BorderPane>
```

I then created a corresponding controller file in which I initialized a `PlotPane` and set it as the center node in the border pane:

```java
package application;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.layout.BorderPane;

public class MainController implements Initializable {

        @FXML
        private BorderPane rootPane;

        @Override
        public void initialize(URL arg0, ResourceBundle arg1) {
                // initialize the plotpane
                PlotPane plotPane = new PlotPane();
                // add the panes to the root pane
                rootPane.setCenter(plotPane);
        }

}
```

I then tested if the program would run without crashing, expecting to see a pair of axes:



Figure 7.4: Creating the Main Pane in Scene Builder

And I saw exactly that.

# Chapter 8

# Testing

I have not yet created a system where the user can input function so to add layers (this is the only test that I can do at the moment) to the `PlotPane` I will simply inject the code at the end of the `PlotPane` constructor to "add" layers artificially. Using the table created in the design section I built upon it to create a more suitable tests on the code that I have. I have added tests to test the function in terms of the variable $y$ and to test the Normal Distribution function that I made. The new and updated table is below:

| Test Number | Function being Tested | Sub-Test Number | Input |
|:---:|:---:|:---|:---|
| 1 | Horizontal Lines | a | 4 |
| | | b | $-2$ |
| 2 | Linear | a | $2x - 5$ |
| | | b | $-0.5x$ |
| 3 | Positive Integer Factorised Polynomials | a | $x(x-1)$ |
| | | b | $x \wedge 2(x \wedge 2 + 1)$ |
| | | c | $-x(3x + 1)$ |
| 4 | Positive Integer Unfactorised Polynomials | a | $x \wedge 2 + 2x + 1$ |
| | | b | $x \wedge 2 - 1$ |
| | | c | $x \wedge 3 + 4x \wedge 2 + x - 6$ |
| 5 | Non-Integer Polynomials | a | $x \wedge (1/2)$ |
| | | b | $x \wedge (1/3)$ |
| 6 | Exponentials | a | $e \wedge x$ |
| | | b | $2 \wedge x$ |
| | | c | $x \wedge x$ |
| 7 | Asymptotal | a | $1/(x-1)$ |
| | | b | $1/(x-1) + 1/(x+1)$ |
| 8 | Explicit Functions in terms of $y$ | a | $y \wedge 2$ |
| | | b | $e \wedge y$ |
| 9 | Normal Distribution Function | a | $\mu = 0$ and $\sigma^2 = 1$ |
| | | b | $\mu = 4$ and $\sigma^2 = 0.01$ |
| 10 | Multiple and Coloured Functions | | N/A |

Table 8.1: Prototype 1: Updated Testing Table

I have laid all of the test evidence into separate sections, with their expected outputs and relevant code. For consistency, I will make every "a" sub test black in line colour, every "b" sub test red in line colour and every "c" sub test green in line colour. This also allows me to test another feature of the program. A summary and analysis of the results are recorded in a table at the end of this section.

## 8.1 Test 1

```
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("4");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("-2");
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```
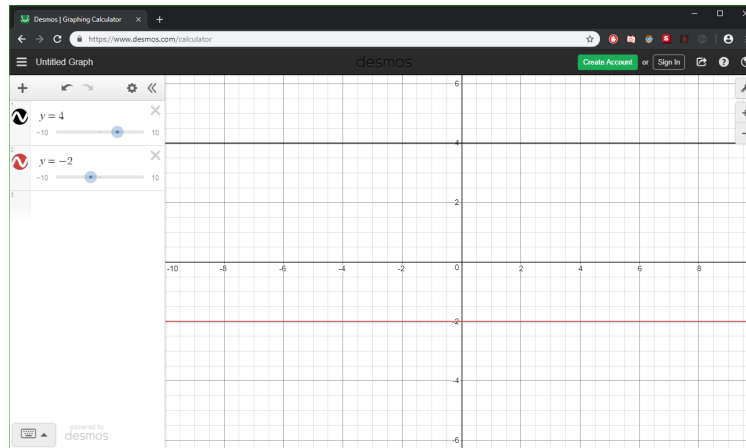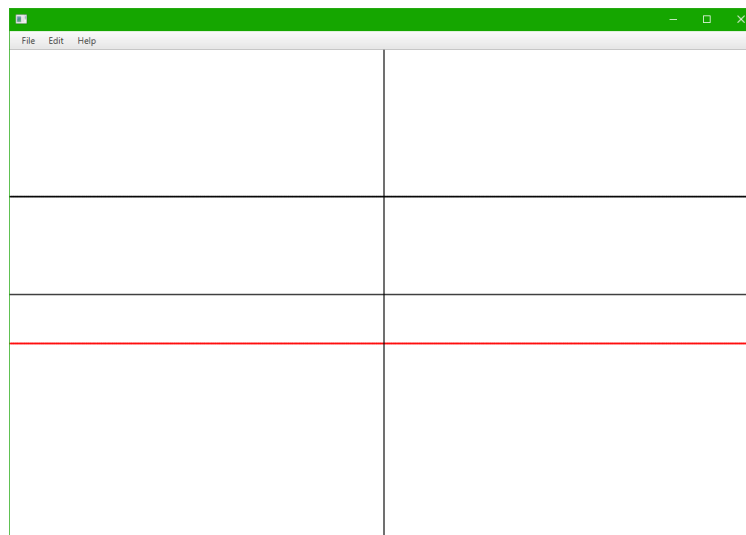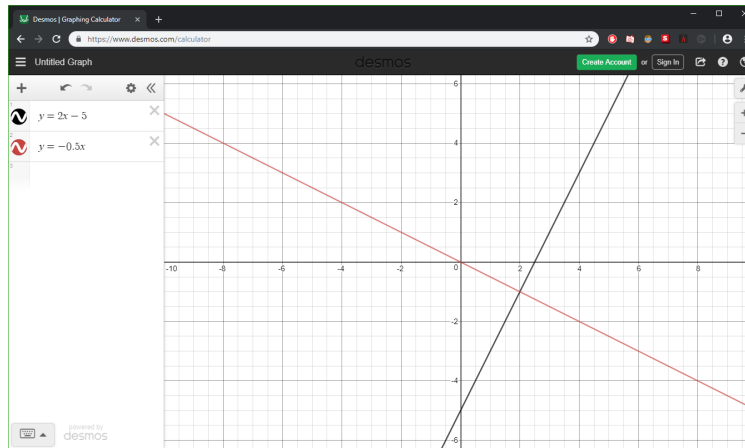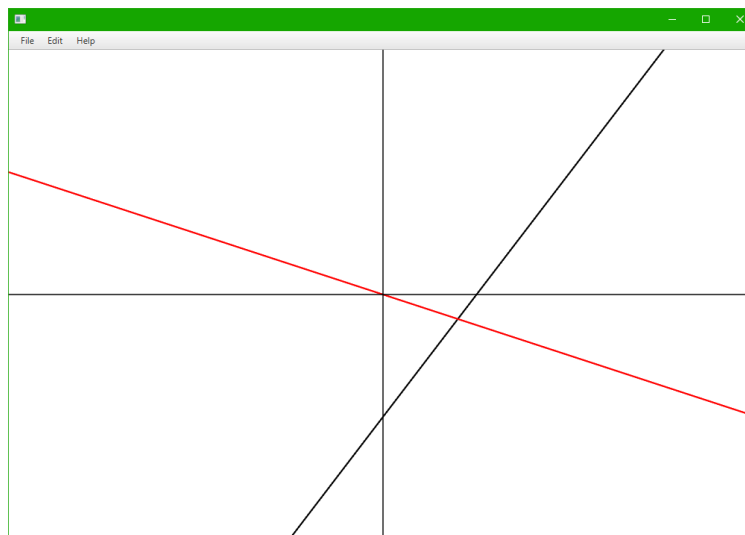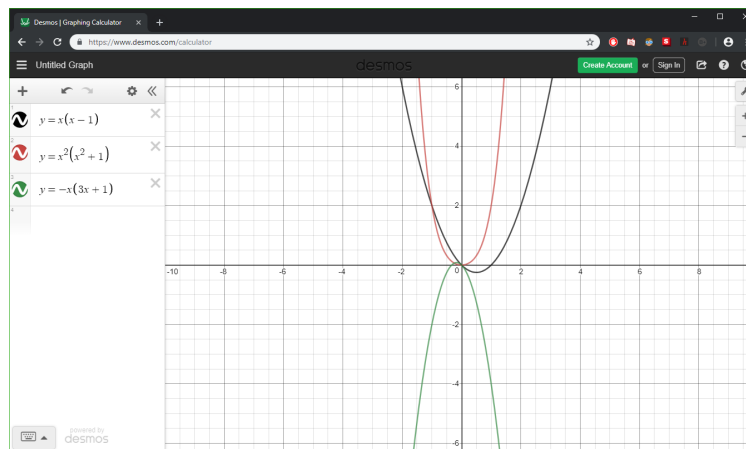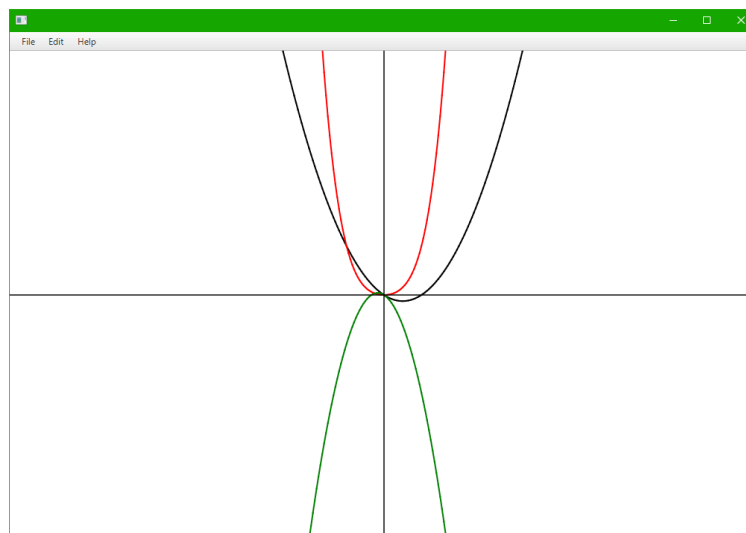


Figure 8.1: Expected Output



Figure 8.2: Actual Output

## 8.2   Test 2

```java
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("2x-5");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("-0.5x");
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```



Figure 8.3: Expected Output



Figure 8.4: Actual Output

## 8.3 Test 3

```
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("x(x-1)");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("x^2(x^2+1)");
ExplicitXFunctionCartesianLayer h = new ExplicitXFunctionCartesianLayer("-x(3x+1)");
g.setColor(Color.RED);
h.setColor(Color.GREEN);

this.addLayer(f);
this.addLayer(g);
this.addLayer(h);
```
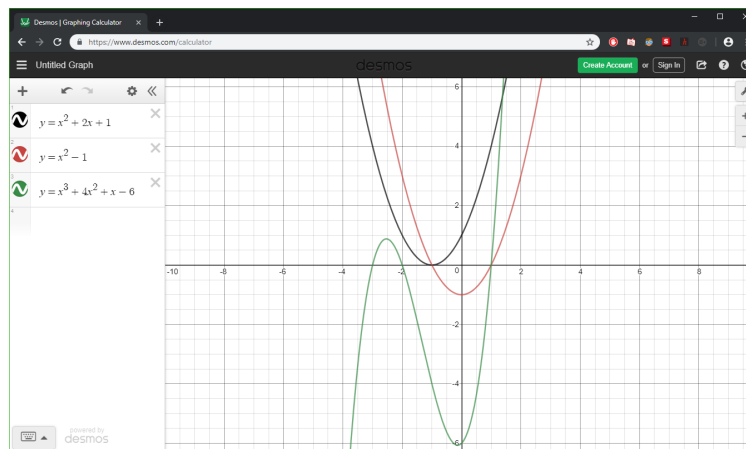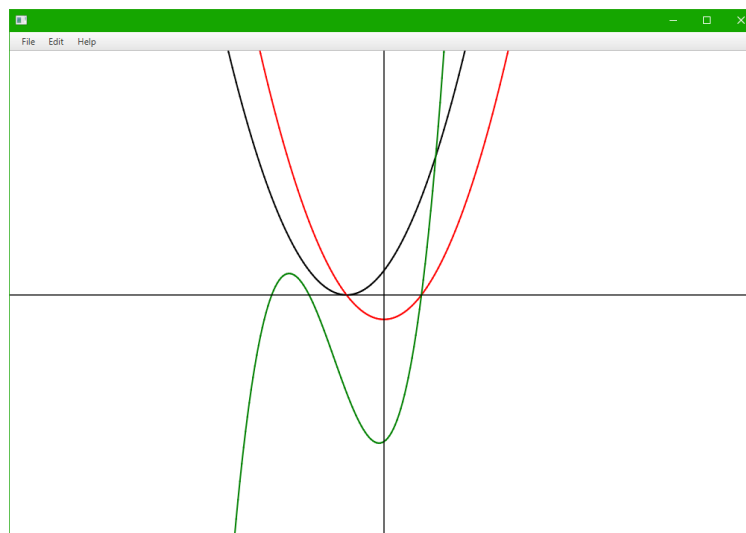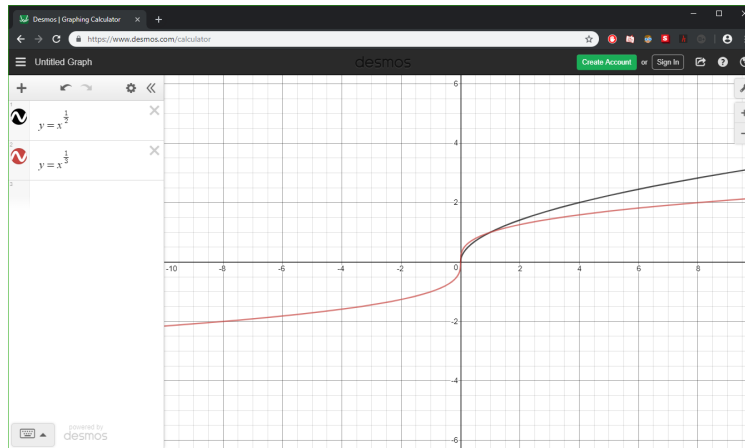


Figure 8.5: Expected Output



Figure 8.6: Actual Output

## 8.4 Test 4

```
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("x^2+2x+1");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("x^2-1");
ExplicitXFunctionCartesianLayer h = new ExplicitXFunctionCartesianLayer("x^3+4x^2+x-6");
g.setColor(Color.RED);
h.setColor(Color.GREEN);

this.addLayer(f);
this.addLayer(g);
this.addLayer(h);
```



Figure 8.7: Expected Output



Figure 8.8: Actual Output

## 8.5   Test 5

```java
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("x^(1/2)");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("x^(1/3)");
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```



Figure 8.9: Expected Output



Figure 8.10: Actual Output

## 8.6    Test 6

```
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("e^x");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("2^x");
ExplicitXFunctionCartesianLayer h = new ExplicitXFunctionCartesianLayer("x^x");
g.setColor(Color.RED);
h.setColor(Color.GREEN);

this.addLayer(f);
this.addLayer(g);
this.addLayer(h);
```
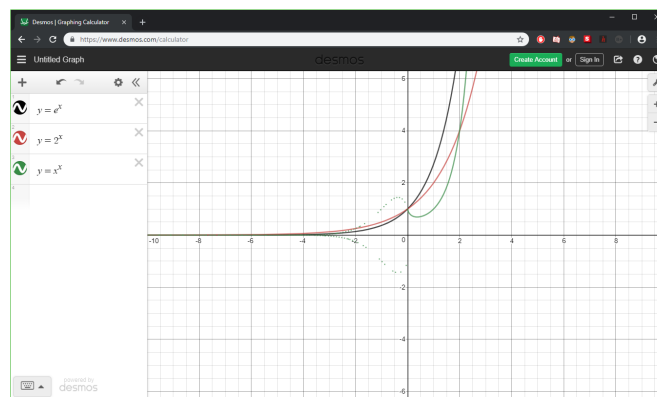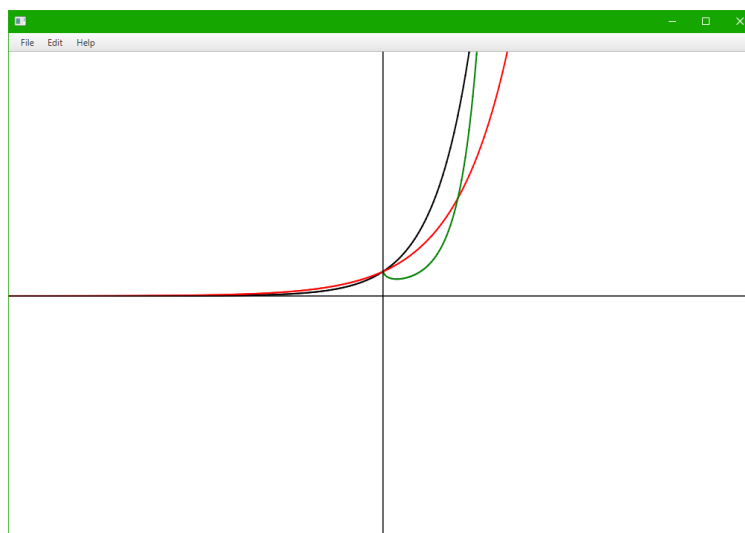


Figure 8.11: Expected Output



Figure 8.12: Actual Output

## 8.7   Test 7

```
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer("1/(x-1)");
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer("1/(x-1) + 1/(x+1)");
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```
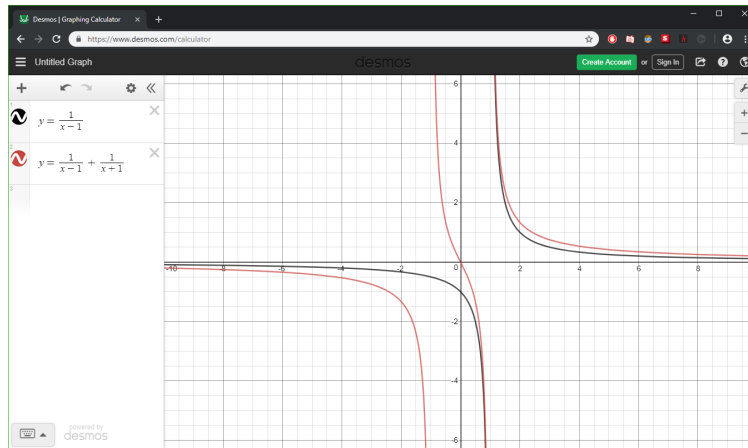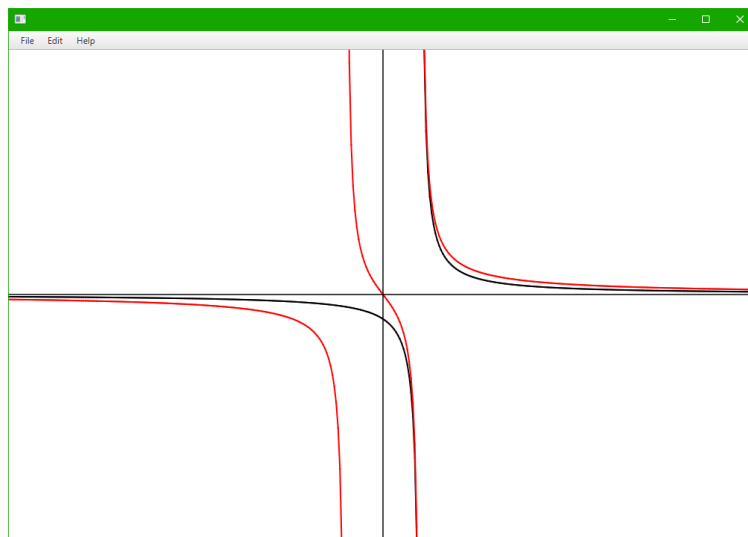


Figure 8.13: Expected Output



Figure 8.14: Actual Output

## 8.8   Test 8

```
ExplicitYFunctionCartesianLayer f = new ExplicitYFunctionCartesianLayer("y^2");
ExplicitYFunctionCartesianLayer g = new ExplicitYFunctionCartesianLayer("e^y");
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```
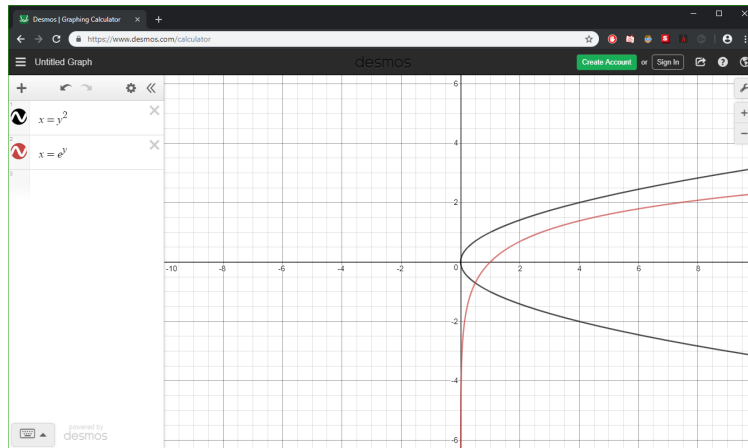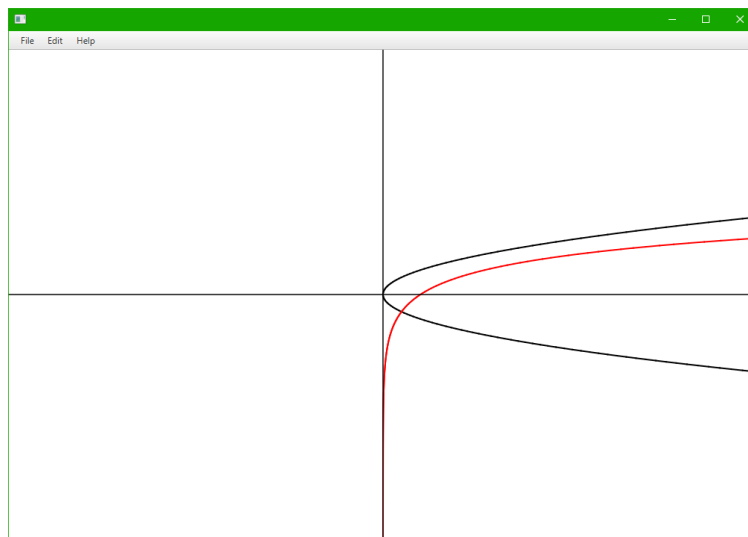


Figure 8.15: Expected Output



Figure 8.16: Actual Output

## 8.9 Test 9

```java
ExplicitXFunctionCartesianLayer f = new ExplicitXFunctionCartesianLayer(new
↪   NormalDistribution(0, 1));
ExplicitXFunctionCartesianLayer g = new ExplicitXFunctionCartesianLayer(new
↪   NormalDistribution(4, 0.01));
g.setColor(Color.RED);

this.addLayer(f);
this.addLayer(g);
```
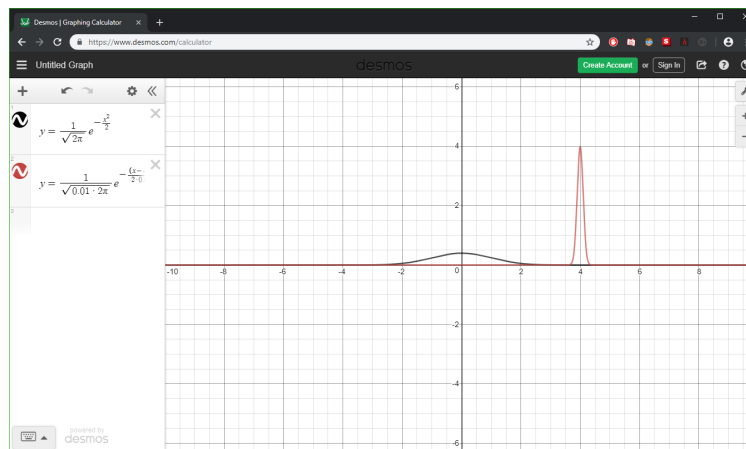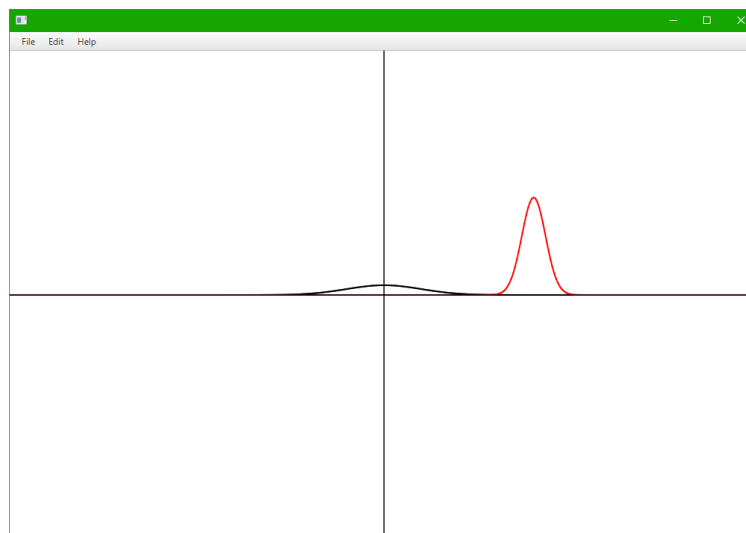


Figure 8.17: Expected Output



Figure 8.18: Actual Output

## 8.10 Conclusion

| Test Number | Function being Tested | Pass/Fail | Analysis |
|---|---|---|---|
| 1 | Horizontal Lines | ✓ | The output was exactly what was expected. |
| 2 | Linear | ✓ | The output was exactly what was expected. |
| 3 | Positive Integer Factorised Polynomials | ✓ | The output was exactly what was expected. |
| 4 | Positive Integer Unfactorised Polynomials | ✓ | The output was exactly what was expected. |
| 5 | Non-Integer Polynomials | ✗ | Test part b partially failed since for $x < 0$ there was nothing drawn. This is due to the function `Math.pow()` returning `NaN`, Not a Number, when given a negative base and a non-integer index[35]. This is a limitation of the language I am using and I cannot do anything about it. |
| 6 | Exponentials | ✗ | Test part c partially failed. Looking closely at the two images you can see that there were individual points for $x < 0$ in the expected result where the function existed. However in the actual result, there was nothing drawn. This seems to be due to there only being one value in the local range for which the function is valid. Since our program works by connecting many points together with a straight line it makes sense that it doesn't work for single points. While this is a loss in accuracy, one of the targets for this project was for a responsive application and if we used a pixel by pixel drawing algorithm it would be too slow. So this is not worth attempting to fix since it would go against one of the targets for this projects. |
| 7 | Asymptotal | ✓ | The output was exactly what was expected. |
| 8 | Explicit Functions in terms of $y$ | ✓ | The output was exactly what was expected. |
| 9 | Normal Distribution Function | ✓ | The output was exactly what was expected. |
| 10 | Multiple and Colored Functions | ✓ | The multiple functions at once and different colors for all the separate functions worked perfectly for all the tests. |

Table 8.2: Conclusion and Analysis of the Prototype 1 Tests

# Chapter 9

# Evaluation

Prototype 1 has been quite successful. Out of the original success criteria:

- Plot a explicit function in $x$

- Plot multiple functions on the same plot

,both have been fulfilled and been extended. I have been able to plot an explicit function in terms of $y$ and made a Normal Distribution Function. I can plot all of the functions that I originally wanted quite accurately, with the only inaccuracies due to the programming language and not my algorithms.

I showed this prototype to my stakeholders and they said that they *"It looks impressive so far"* and *"I like the simplistic and uncluttered look of it and I hope that you keep it that way"*. Jeevon asked for a new feature where you can *"hide and show any function"* and Palvinder asked for a feature where you can *"see the coordinates of the point your cursor is at"*. So I will add these to the list of requirements.

Sadly this project has presented some problems in terms of time. It is clear that a third prototype will not happen in the time available and some of prototype 2 will not happen either. This is the new requirements for prototype 2:

- Let the user input functions

- Zoom in/out of the graph

- Pan around the graph

- Save plots as pictures

- Show/Hide functions (suggestion from a stakeholder)

- Show coordinates in top left of pane (suggestion from a stakeholder)

From talking to my stakeholders, these are the key things that they want in a graphing program and as such these are the things that I will definitely implement. If I have time I will implement the other features.

# Part III

# Prototype II

# Chapter 10

# Design

The requirements defined at the end of prototype 1 can be split them up into 2 distinct sections:

1. User Input

   - Let the user input functions
   - Show/Hide functions (suggestion from a stakeholder)

2. User Interaction

   - Zoom in/out of the graph
   - Pan around the graph
   - Save plots as pictures
   - Show coordinates in top left of pane (suggestion from a stakeholder)

The first section appears deceptively small however it is probably the most complicated part of the program. It needs to be simple to use and it will probably be the most difficult part to debug.

## 10.1 User Interaction

Almost all of the user interaction will done through the input layer except the saving the plot as a picture. Most of these interactions are essentially coordinate transformations, translations for panning, scaling for zooming etc. The pan and zoom functionality actually change the functions drawn upon the screen so the plot pane needs to be notified to draw the functions again. This will be done by changing the viewport property, by negating the value of the viewport, i.e. changing from true to false or vice versa.

### 10.1.1 Save Picture

JavaFX has a feature where you can take a snapshot[36] any node. Since we want to take to take a picture of of all the functions we will take a snapshot of the plot pane. We will configure the save a picture action to be accessible through a right click menu when you click on the pane. This right click menu is called a context menu[37] in JavaFX. The user should also be able to name the image and save it where they want. We will create a method in the plot pane class to create a snapshot and save it to wherever the user wants.

### 10.1.2 Coordinates

The current coordinates will be shown in the top left of the input pane. Whenever the mouse moves over the pane, the current coordinates will change to their new value. Since the coordinates will probably be a very long and horrible decimal I will round the coordinates to 2 decimal places. I will also have to convert from coordinates from the canvas to the Cartesian coordinates. This is essentially the reverse of the conversion algorithm in prototype 1. Here is the algorithm for showing the coordinates:

---

**Algorithm 20:** Show the Current Coordinates

**1 function** *drawCoords()*:
2     clearCanvas()
3     **Double** x = (mouseX * this.pixelWorthX) + this.minX
4     **Double** y = this.maxY - (mouseY * this.pixelWorthY)
5     Round x to 2 decimal places
6     Round y to 2 decimal places
7     **String** out = "(" + x + "," + y + ")"
8     drawText(out,0,0)
**9 end**

---

### 10.1.3 Pan

Panning is essentially a translation of the viewport. It will edit the minimum and maximum values of $x$ and $y$ (the boundaries of our viewport) to crate the illusion of panning around. It will be a method in the input layer class. We will essentially make this occur when holding down left click and dragging across the pane. It will work by storing the current and previous values of where the cursor has been relative to the coordinate axes. These will be attributes in the input layer class. We will then work out the vector from the previous position to the new position. Then add this vector to the maximum and minimum values. Finally we need to notify the plot pane to draw again by changing the viewport attribute and updating the pixel worth. Here is the algorithm for panning:

---

**Algorithm 21:** Pan around the Plot

**1 function** *pan()*:
2     **if** *mouse left click is down* **then**
3        currentX = event.getX()
4        currentY = event.getY()
5        double xTrans = (previousX - currentX)
6        double yTrans = (previousY - currentY)
7        previousX = currentX
8        previousY = currentY
9        maxX = (this.pixelWorthX * xTrans) + this.maxX
10       minX = (this.pixelWorthX * xTrans) + this.minX
11       maxY = -(this.pixelWorthY * yTrans) + this.maxY
12       minY = -(this.pixelWorthY * yTrans) + this.minY
13       this.changeViewport = ! this.changeViewport          `// notify the plot pane`
**14 end**

---

## 10.1.4 Zoom

There are two ways to implement a zooming functionality. One is where we zoom about the center of the viewport. This is unintuitive since no matter where the cursor is, it always zooms around the center. Another one is where we zoom around the cursor. This feels incredibly natural and most graphing programs like Desmos and Geogebra have this. It may be harder to implement than the first idea but it fits with the target that the program should be responsive and natural.

The zooming will occur when the scroll wheel on the mouse is scrolled. If scroll wheel is scrolled up the viewport will zoom in and if it is scrolled in the viewport will zoom out. To achieve effect of zooming into the cursor we can use some basic transformations to achieve what we need. So first we need to translate the viewport from the cursor to the origin. Then scale it up or down. Then translate it back to where the cursor was.

Again we need to notify the plot pane that the plot needs to be drawn again. We also need to update the pixel worth since the how much each pixel is worth will have changed since we are effectively scaling each pixel. So we need to call the `updatePixelWorth()` method.

| **Algorithm 22:** Zoom into or out of the Plot |
|---|

```
 1 function pan():
 2     Double zoomFactor = 1.05
 3     Double x = (mouseX * this.pixelWorthX) + this.minX        // get the cursor coordinates
 4     Double y = this.maxY - (mouseY * this.pixelWorthY)
 5     if zoomFactor = 1 / zoomFactor
 6      then Mouse is Scrolling Up

 7     Double newMinX = this.minX- x                              // translate viewport to origin
 8     Double newMaxX = this.maxX - x
 9     Double newMinY = this.minY - y
10     Double newMaxY = this.maxY - y
11     newMinX = newMinX * zoomFactor                            // scale the viewport
12     newMaxX = newMaxX * zoomFactor
13     newMinY = newMinY * zoomFactor
14     newMaxY = newMaxY * zoomFactor
15     newMinX = newMinX + x                                     // translate back to original place
16     newMaxX = newMaxX + x
17     newMinY = newMinY + y
18     newMaxY = newMaxY + y
19     this.minX = newMinX
20     this.maxX = newMaxX
21     this.minY = newMinY
22     this.maxY = newMaxY
23     this.updatePixelWorth()
24     this.changeViewport = ! this.changeViewport              // notify the plot pane
25 end
```

## 10.2   User Input

Back when designing prototype 1 I talked about the idea of a class that essentially stores data that different parts of the program can access. The reason for this is to remove the reason for global variables which are bad practice.

### 10.2.1   Shared Layer Access

The shared layer access will be a way for the input method to give layers to the plot pane to then add to its layers. It will contain an attribute that is the dictionary type data structure to hold all of the layers. This is so that a specific layer can be identified and removed (if the user wishes to remove a function). It will also contain public methods to add and remove layers from the list, and a getter for the dictionary values (the plotpane doesn't need the identifiers) in the form of a list so that the plotpane can access the list to use. It will also contain a property that is connected to the plotpane that will notify when the list has changed so that the plotpane can draw again. Here is the class diagram for the Shared Layer Access class:

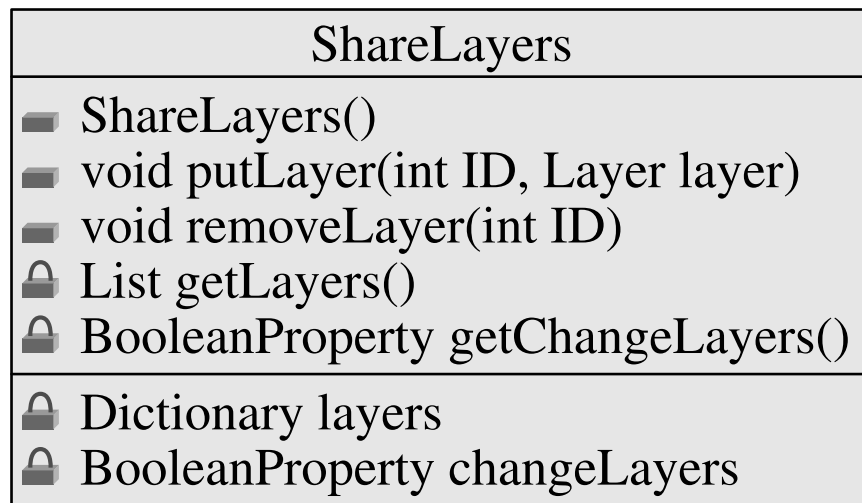| ShareLayers |
|---|
| ⬤ ShareLayers() <br> ⬤ void putLayer(int ID, Layer layer) <br> ⬤ void removeLayer(int ID) <br> 🔒 List getLayers() <br> 🔒 BooleanProperty getChangeLayers() |
| 🔒 Dictionary layers <br> 🔒 BooleanProperty changeLayers |

Figure 10.1: Shared Layer Access Class

The plotpane will have have to bind the Boolean property so the plot pane will get an extra method and attribute to set up this shared layer access class.

## 10.2.2 Input Method

There are two parts to this input. I will have a miniature pane to input and control individual functions, which I will call an expression box, and then a scroll pane to store all these expression boxes. The reason I am using a scroll pane for this is because if you keep adding expression boxes I want the user to be able to scroll down to see all off them.

### 10.2.2.1 Input Pane

The input pane will store all the expression boxes and it will have a button to add a new box. This is how it will look like the sketch below.

From an implementation point of view I will just create a class that will inherit scroll pane. This allows it to mesh with the rest of the program, and is feasible to do without a GUI builder since it is quite a simple design. I will need some way to identify each expression box to be able to remove and update the function/layer associated with it. I will do this through an attribute of type integer called ID. I will simply use an attribute within the input pane class to keep track of what the next ID is to assign the next expression box. I will automatically make each function a new colour and this will be done with an array that will contain the standard colours that I will use and each new expression box will cycle through each of them. The expression boxes will be stored in a VBox which is a JavaFX component. It is a simple layout since it lays nodes out vertically, which is exactly how the expression will be laid out.
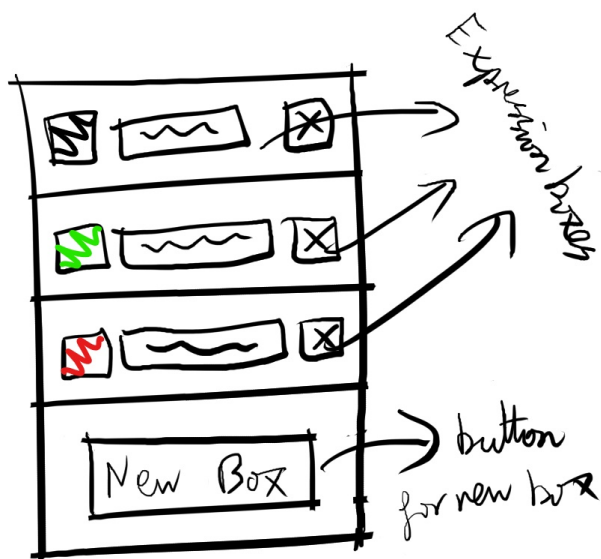


Figure 10.2: Input Pane

```
                    InputPane
━  InputPane()
🔒  void addExpressionBox()
━  void removeExpressionBox(int ID)
━  void putLayer(int ID, Layer layer)
━  void removeLayer(int ID)
🔒  void setupNewButton()
━  setShareLayerStore(ShareLayers shareLayers)

🔒  VBox expressionBoxStore
🔒  ShareLayers shareLayerStore
🔒  int nextID
🔒  Color[] colors
```

Figure 10.3: Input Pane Class

The input pane class will have to communicate with the plot pane to update the layers. It will do this through the shared layers access class so it will have an attribute for it. It will also have methods that the expression box can access to add, remove or edit layers.

#### 10.2.2.2 Expression Box

I will make the expression box look similar to the one in Desmos shown below.



Figure 10.4: Input in Desmos

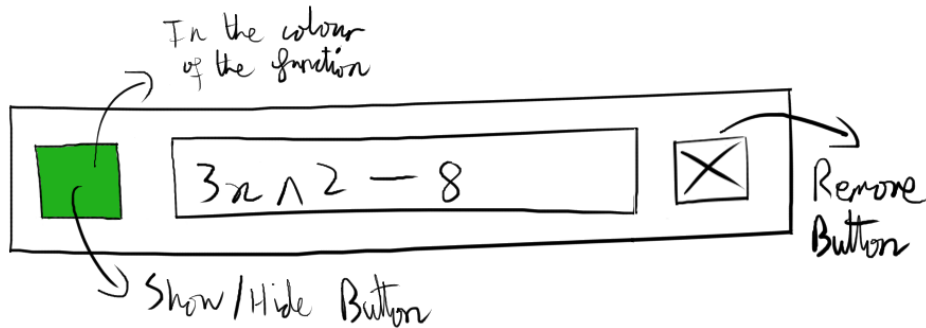I made my design more simplistic and fit the theme of my program. It is shown below:



Figure 10.5: Expression Box

   I showed Lewis, a stakeholder, this design and he loved it. He said that *"It looks really easy to use and I like the look."*.

From an implementation point of view, I will create the expression box in FXML and then use a controller class to implement the methods to remove itself and add functions to the plot through the shared layer class. It will have an attribute called ID, to identify itself, this will be explained in the next section. It will also have an attribute for the colour of the function that it will draw. To access the plot pane it will have a attribute to access its parent the input pane, through which it can access the shared access class though which it can access the plot pane. The class diagram for this is below:

| ExpressionBox |
| --- |
| ⚊ ExpressionBox() |
| 🔒 void changeLayer() |
| 🔒 void remove() |
| ⚊ void setID(int ID) |
| ⚊ int getID() |
| ⚊ void setInputPane(InputPane inputPane) |
| ⚊ void set(Color color) |
| 🔒 int ID |
| 🔒 Color color |
| 🔒 InputPane inputPane |
| 🔒 boolean functionVisibility |

Figure 10.6: Input Pane Class

100

## 10.3 Testing Strategy

# Chapter 11

# Implementation

When beginning to implement prototype 2 I realized that I need to change some things while implementing that I said in design:

1. The user interaction section is more complicated than I made it out to be since it requires knowledge of specific types of events in JavaFX like Mouse and Scroll events, but I will look into and explain it more at that stage.

2. The classes that I designed will probably have more methods and attributes so that I can break down the problems into manageable chunks.

3. The Expression Box class will actually be a controller class so it will be implemented slightly differently than what I designed. It will have many more attributes and methods related to the FXML side and will be explained more when I get to it.

I will implement the user interaction elements first and then user input. This is because the user interaction is easier to unit test and the user input will need to be tested all at once since it is too interconnected, as explained in the test strategy. It also allows me to show my stakeholders something significant sooner.

## 11.1 User Interaction

All of these features require me to know about how mouse events are handled in JavaFX since all of these actions require some kind of mouse input. I used these websites[40] to get a deeper understanding of how they worked. Events are essentially objects with attributes that describe the event that just occurred. For example an attribute could tell you the position of the event. They are quite simple to use and if required I will give specifics when needed.

### 11.1.1 Coordinates

I used the algorithm I designed in the design section and implemented it as a method in the input layer class. I then bound this method to the `OnMouseMoved` mouse event in the input layer constructor, so the coordinates would update whenever the moused moved in the plot pane. So I added this to the input layer constructor:

```
this.canvas.setOnMouseMoved(event -> drawCoords(event));
```

Here is the `drawCoords()` method:

```
// draw the coordinates
private void drawCoords(MouseEvent event) {
        // clear the previous coordinates
        clearCanvas();
        // get the coordinates
        double x = (event.getX() * this.pixelWorthX.doubleValue()) + this.minX.doubleValue();
        double y = this.maxY.doubleValue() - (event.getY() * this.pixelWorthY.doubleValue());
        // round the coordinates to 2 decimal places
        String sX = new DecimalFormat("#.##").format(x);
        String sY = new DecimalFormat("#.##").format(y);
        String out = "(" + sX + "," + sY + ")";
        // draw the coordinates to the canvas
        gc.setLineWidth(1);
        gc.strokeText(out, 0, 10);
}
```

It works perfectly and an image of what it looks like is shown below. I asked Palvinder, the stakeholder who suggested this feature, about what he thought of the implementation and he said that it was *"exactly how I envisioned it to be"*.
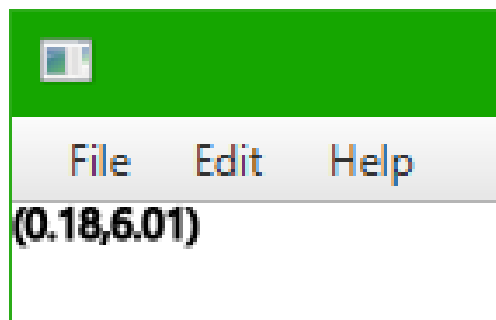


Figure 11.1: Coordinates in Action

### 11.1.2 Pan

The panning feature was quite hard to implement due to the lack of clear documentation on DragEvent. From lots of searching I found out that I had to call startFullDrag() method when the drag is initiated. Using this information I added these lines to the input layer constructor:

```
this.canvas.setOnMousePressed(event -> dragEntered(event));
this.canvas.setOnMouseDragged(event -> pan(event));
```

I also added 4 new attributes of type double to the input layer class, previousX, previousY, currentX and currentY. Here are the two methods used in the previous snippet of code dragEntered() and pan():

```
1   // when drag starts initiate the full-drag sequence
2   private void dragEntered(MouseEvent event) {
3           // only initiate the drag if its the left mouse button
4           if (event.isPrimaryButtonDown()) {
5                   // start full drag
6                   event.setDragDetect(true);
7                   this.canvas.startFullDrag();
8           }
9   }
10
11  // pan around the plotpane
12  private void pan(MouseEvent event) {
13          if (event.isPrimaryButtonDown()) {
14                  // get the current mouse coordinates
15                  currentX = event.getX();
16                  currentY = event.getY();
17                  // get the vector of the mouse movement
18                  double xTrans = (previousX - currentX);
19                  double yTrans = (previousY - currentY);
20                  // update the previous values to be the current ones
21                  previousX = currentX;
22                  previousY = currentY;
23                  // translate the viewport
24                  this.maxX.set((this.pixelWorthX.doubleValue() * xTrans) +
                    ↪   this.maxX.doubleValue());
25                  this.minX.set((this.pixelWorthX.doubleValue() * xTrans) +
                    ↪   this.minX.doubleValue());
26                  this.maxY.set(-(this.pixelWorthY.doubleValue() * yTrans) +
                    ↪   this.maxY.doubleValue());
27                  this.minY.set(-(this.pixelWorthY.doubleValue() * yTrans) +
                    ↪   this.minY.doubleValue());
28                  this.changeViewport.set(!this.changeViewport.get());
29          }
30  }
```

Then I tested the pan functionality. When I first held down the left mouse click button, the viewport jumped to another location and then the pan functionality worked fine, as in I could move it about. This told me that there was a problem while initiating the drag sequence. So I used breakpoints, debugging tools and the variable stack viewer in Eclipse to debug the problem:
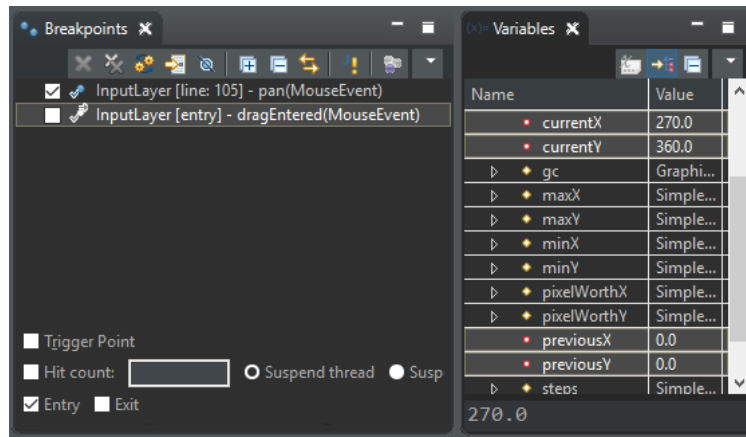


Figure 11.2: Coordinates in Action

From this I realized that the values for `previousX` and `previousY` were the last values in the previous drag which caused the viewport to jump when initiating the drag. To fix this when entering the drag I gave them the value of the current location of the cursor:

```
1   // when drag starts initiate the full-drag sequence
2   private void dragEntered(MouseEvent event) {
3           // only initiate the drag if its the left mouse button
4           if (event.isPrimaryButtonDown()) {
5                   // need to initialize the values of previousX and previousY
6                   previousX = event.getX();
7                   previousY = event.getY();
8                   // start full drag
9                   event.setDragDetect(true);
10                  this.canvas.startFullDrag();
11          }
12  }
```

This fixed the issue, although an exception seems to be thrown every time the drag is initiated:

```
Exception in thread "JavaFX Application Thread" java.lang.IllegalStateException:
Cannot start full drag outside of DRAG_DETECTED event handler
```

However the panning feature works so this exception can be ignored. Evidence of it working will be shown in the testing section in video form.

### 11.1.3 Zoom

I used the algorithm I designed in the design section and implemented it as a method in the input layer class, with some slight changes talked about in the next paragraph. I then bound this method to the `OnScroll` scroll event in the input layer constructor, so that the it would zoom in or out when scrolling. This is what the input layer constructor is and this is the final change:

```java
public InputLayer() {
        // call the super constructor
        super();
        // update pixel worth when the canvas changes size
        this.canvas.heightProperty().addListener(event -> updatePixelWorth());
        this.canvas.widthProperty().addListener(event -> updatePixelWorth());
        // bind the mouse events to their respective actions
        this.canvas.setOnMousePressed(event -> dragEntered(event));
        this.canvas.setOnMouseDragged(event -> pan(event));
        this.canvas.setOnScroll(event -> zoom(event));
        this.canvas.setOnMouseMoved(event -> drawCoords(event));
}
```

When the scroll event is triggered it is an object with various attributes. One of them is the *deltaY* attribute. It is an integer and it tells us if the scroll wheel is being scrolled down or up. If it is positive it is being scrolled down and if it is negative it is being scrolled up. So using this knowledge I took my algorithm I designed and adapted it to to implemented `zoom()` method:

```java
// zoom in and out
private void zoom(ScrollEvent event) {
        double zoomFactor = 1.05;
        // this is whether it is scrolling up or down
        double deltaY = event.getDeltaY();
        // get the coordinates
        double x = (event.getX() * this.pixelWorthX.doubleValue()) + this.minX.doubleValue();
        double y = this.maxY.doubleValue() - (event.getY() * this.pixelWorthY.doubleValue());
        // if scrolling down shrink the viewport instead
        if (deltaY > 0) {
                zoomFactor = 1 / zoomFactor;
        }
        // translate the viewport to the origin, scale the viewport,
        // translate the viewport back and set the new values for the viewport
        this.minX.set((this.minX.doubleValue() - x) * zoomFactor + x);
        this.maxX.set((this.maxX.doubleValue() - x) * zoomFactor + x);
        this.minY.set((this.minY.doubleValue() - y) * zoomFactor + y);
        this.maxY.set((this.maxY.doubleValue() - y) * zoomFactor + y);
        // update and notify the plotpane
        this.updatePixelWorth();
        this.changeViewport.set(!this.changeViewport.get());
}
```

It works perfectly and evidence of it in action will be shown in video form in the testing section. Using alone images will not be able to capture the zooming in and out.

### 11.1.4 Save Picture

I found these two posts[38, 36], which I combined to be able to save the image where the user wants and with any name. To create the context menu, the right click menu, I used this example[39]. I edited and inserted this code to add a right click menu at the end of the plot pane constructor:

```
// create a right click menu
ContextMenu contextMenu = new ContextMenu();
MenuItem save = new MenuItem("Save as Picture");
contextMenu.getItems().addAll(save);
// set an action to call the savePlot() method when clicking the save button
save.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
                savePlot();
        }
});
// when right clicking open the menu
this.setOnContextMenuRequested(
                event -> contextMenu.show(this.getScene().getWindow(), event.getScreenX(),
                ↪   event.getScreenY()));
```

I then created the `savePlot()` method that is called when the *"Save as Picture"* menu item is clicked:

```
// save picture of plot
private void savePlot() {
        // create the file navigator object
        FileChooser fileChooser = new FileChooser();
        // set extension filter (make the image of format png)
        fileChooser.getExtensionFilters().add(new
        ↪   FileChooser.ExtensionFilter("png files (*.png)", "*.png"));
        // prompt user to select a file
        File file = fileChooser.showSaveDialog(null);
        if (file != null) {          //if the user chose to save then snapshot the pane
                try {
                        // create the snapshot of the pane
                        WritableImage snapshot = this.snapshot(new SnapshotParameters(),
                        ↪   null);
                        RenderedImage renderedImage = SwingFXUtils.fromFXImage(snapshot,
                        ↪   null);
                        // write the snapshot to the chosen file
                        ImageIO.write(renderedImage, "png", file);
                } catch (IOException ex) {
                        ex.printStackTrace();
                }
        }
}
```

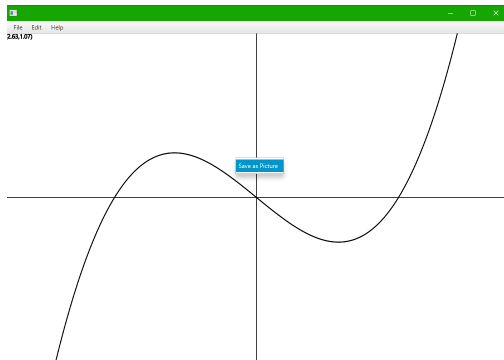I then tried to save an image of the plot and it worked perfectly:



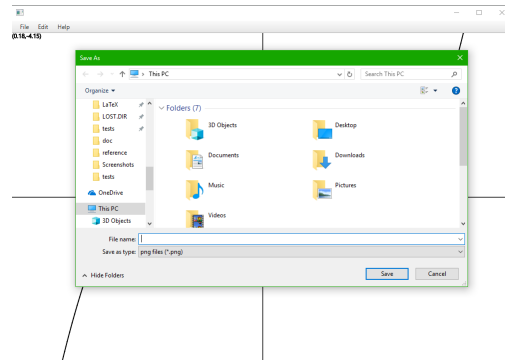Figure 11.3: The Right Click Menu



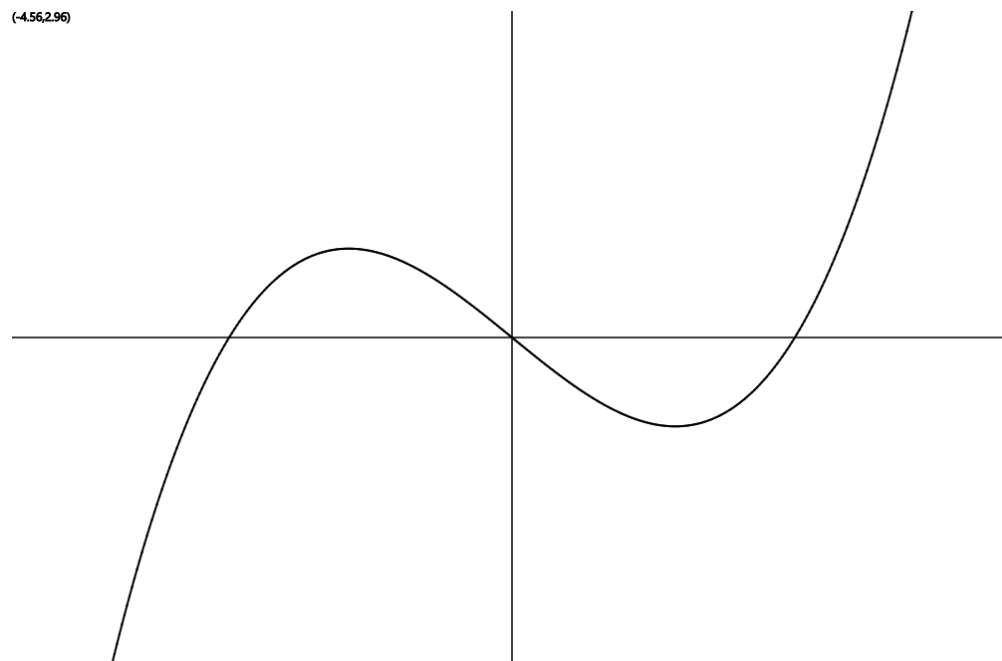Figure 11.4: The File Navigator to Save the File



Figure 11.5: The Image Produced

A recording of the process to save will be done during the overall testing.

I let the stakeholder who originally asked for this feature, Lewis, try it and he said that he said it was all good except for the *"coordinates that were shown in the top left corner"* and he said that he didn't want it in the saved picture. I took his suggestion and tried to fix it. Since the coordinates are part of the input layer, I fixed this by removing the input layer canvas, taking the snapshot and then readding the input layer canvas. It worked, so I showed Lewis again and he said it was great. Here is the code and evidence for the updated feature:

```java
// save picture of plot
private void savePlot() {
        // create the file navigator object
        FileChooser fileChooser = new FileChooser();
        // set extension filter
        fileChooser.getExtensionFilters().add(new
        ↪   FileChooser.ExtensionFilter("png files (*.png)", "*.png"));
        // set extension filter (make the image of format png)
        File file = fileChooser.showSaveDialog(null);
        if (file != null) {          //if the user chose to save then snapshot the pane
                try {
                        // remove input layer canvas, the top node, to not show the coords
                        this.getChildren().remove(this.getChildren().size() - 1);
                        // create the snapshot of the pane
                        WritableImage snapshot = this.snapshot(new SnapshotParameters(),
                        ↪   null);
                        RenderedImage renderedImage = SwingFXUtils.fromFXImage(snapshot,
                        ↪   null);
                        // write the snapshot to the chosen file
                        ImageIO.write(renderedImage, "png", file);
                } catch (IOException ex) {
                        ex.printStackTrace();
                } finally {
                        // readd the input layer
                        this.getChildren().add(inputLayer.getCanvas());
                }
        }
}
```
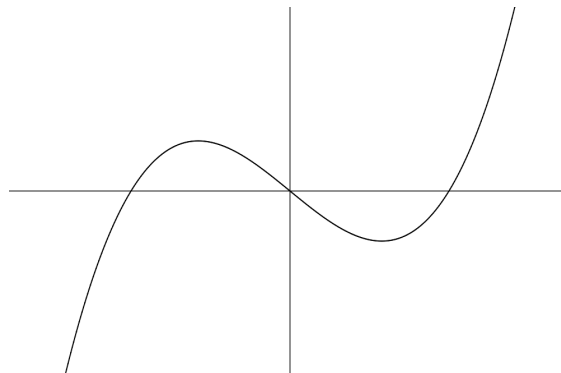


Figure 11.6: The Image Produced

## 11.2 User Input

### 11.2.1 Shared Layer Access

Within Java there is a standard dictionary like structure called a Map which we can use to store the layers. More specifically a Map[41] is actually a interface and the specific class we will be using to store the layers is called a HashMap[42] (It can be said that HashMap inherits Map). I am using a HashMap since it is lightweight and simple to useWith this information I implemented the ShareLayers class:

```java
package application;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;
import layer.Layer;

public class ShareLayers {

        // attributes
        private Map<Integer, Layer> layers;
        private BooleanProperty changeLayers;

        // methods
        // constructor
        public ShareLayers() {
                // initialize attributes
                layers = new HashMap<Integer, Layer>();
                changeLayers = new SimpleBooleanProperty(true);
        }

        // add a layer
        public void putLayer(int ID, Layer layer) {
                // add the layer
                this.layers.put(ID, layer);
                // notify the plot pane to draw again
                this.changeLayers.set(!this.changeLayers.get());
        }

        // remove a layer
        public void removeLayer(int ID) {
                // remove the layer
                this.layers.remove(ID);
                // notify the plot pane to draw again
                this.changeLayers.set(!this.changeLayers.get());
        }

        // return a list of layers
        public ArrayList<Layer> getLayers() {
                return new ArrayList<Layer>(this.layers.values());
        }

        // return the property which notifies the plotpane
        public BooleanProperty getChangeLayers() {
                return changeLayers;
        }

}
```

Since the plot pane will need to be connected to the shared layer access, I created a method called setShareLayerStore() which sets up the connection between the plot pane and the shared layer access. I also gave it two new attributes, a reference to the shared layer access to get the list of layers shareLayerStore and the boolean property changeLayers that the shared layer access will bind with. Here is the implementation of the new method:

```java
// set up the shared layer access
public void setShareLayerStore(ShareLayers shareLayerStore) {
        // set the shared layer access attribute
        this.shareLayerStore = shareLayerStore;
        // bind and set the value for the property
        this.changeLayers.set(this.shareLayerStore.getChangeLayers().get());
        this.changeLayers.bind(this.shareLayerStore.getChangeLayers());
        // create a listener to execute when the plot pane needs to draw again
        ChangeListener<Object> redrawListener = (observable, oldValue, newValue) -> {
                // remove all the old layers
                this.layers.clear();
                // add all the new layers
                for (Layer l : shareLayerStore.getLayers()) {
                        this.addLayer(l);
                }
                // draw the functions again
                try {
                        drawAll();
                } catch (StackOverflowException | StackUnderflowException |
                →  UnequalBracketsException
                                | InterruptedException e) {
                        // error in drawing a function
                        e.printStackTrace();
                }
        };
        // bind the listener to the property
        this.changeLayers.addListener(redrawListener);
}
```

### 11.2.2 Expression Box

Before creating the expression box and its controller class I created a dummy input pane class that contained all its methods and attributes, none of which were properly implemented, so that I wouldn't get any compilation errors. I first created the FXML file using scene builder. This is what I produced:
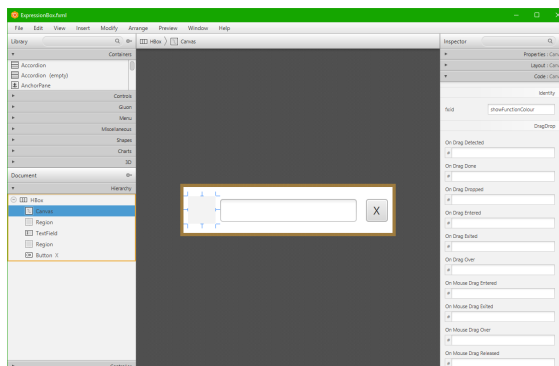


Figure 11.7: Creating the Expression Box in Scene Builder

For the button to show/hide the function I decided to use a canvas to achieve the effect. It gives me finer control about the colour of the button and is easier to manipulate. I showed my stakeholders this and Matthew said that he *"liked the simple design"*. The FXML for the expression box is shown below:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import javafx.geometry.Insets?>
4  <?import javafx.scene.canvas.Canvas?>
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.control.TextField?>
7  <?import javafx.scene.layout.HBox?>
8  <?import javafx.scene.layout.Region?>
9
10 <HBox alignment="CENTER" xmlns="http://javafx.com/javafx/8.0.171"
11        xmlns:fx="http://javafx.com/fxml/1"
12        fx:controller="application.ExpressionBoxController">
13     <children>
14         <Canvas fx:id="showFunctionColour" height="30.0" width="30.0">
15             <HBox.margin>
16                 <Insets left="5.0" />
17             </HBox.margin>
18         </Canvas>
19         <Region HBox.hgrow="ALWAYS" />
20         <TextField fx:id="inputTxt">
21             <HBox.margin>
22                 <Insets left="5.0" right="5.0" />
23             </HBox.margin>
24         </TextField>
25         <Region />
26         <Button fx:id="removeBtn" mnemonicParsing="false" text="X">
27             <HBox.margin>
28                 <Insets left="5.0" right="5.0" />
29             </HBox.margin>
30         </Button>
31     </children>
32     <padding>
33         <Insets bottom="10.0" top="10.0" />
34     </padding>
35 </HBox>
```

I then created the controller class. I introduced a few new methods such as the `changeColor()` method. This one is triggered when the canvas is clicked and shows/hides the function and updates the visual aid. The `changeLayer()` method is called whenever the input text changes or the user clicks on the show/hide canvas. It parses the input text determines whether or not it is a function in $x$, $y$, the Normal Distribution, or none of those. It then creates a layer for that specific function and then adds it to the shared access layer through the input pane. The code for the rest of this class was quite simple to implement and it is below:

```java
package application;

import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;

import exceptions.StackOverflowException;
import exceptions.StackUnderflowException;
import exceptions.UnequalBracketsException;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.paint.Color;
import layer.ExplicitXFunctionCartesianLayer;
import layer.ExplicitYFunctionCartesianLayer;
import layer.Layer;
import structures.Expression;
import structures.NormalDistribution;

public class ExpressionBoxController implements Initializable {

        // attributes
        private int ID;
        private GraphicsContext gc;
        private boolean functionVisiblity = true;
        private Color color;
        private StringProperty functionText = new SimpleStringProperty();
        private InputPane inputPane;
        // FXML components
        @FXML
        private TextField inputTxt;
        @FXML
        private Button removeBtn;
        @FXML
        private Canvas showFunctionColour;

        // methods
        // empty constructor
        public ExpressionBoxController() {
        }

        @Override
        // called when initialized a pseudo constructor of sorts
        public void initialize(URL arg0, ResourceBundle arg1) {
                // bind the input to a string property in the class to manipulate
                this.functionText.set(inputTxt.textProperty().getValueSafe());
                this.functionText.bind(inputTxt.textProperty());
                // add a listener so that every time it changes the layer is updated
                this.functionText.addListener(event -> {
                        try {
                                changeLayer();
                        } catch (StackOverflowException | StackUnderflowException |
                        ↪ UnequalBracketsException e) {
                                e.printStackTrace();
```

```
59                              }
60                      });
61                      // if the remove button is clicked call the remove method
62                      this.removeBtn.setOnAction(event -> {
63                              try {
64                                      remove();
65                              } catch (IOException e) {
66                                      e.printStackTrace();
67                              }
68                      });
69                      // create the canvas to show the color of the function
70                      this.gc = this.showFunctionColour.getGraphicsContext2D();
71                      this.gc.setLineWidth(2);
72                      // when the canvas is clicked change its color by calling the change Color
73                      // method
74                      this.showFunctionColour.setOnMouseClicked(event -> {
75                              try {
76                                      changeColor();
77                              } catch (StackOverflowException | StackUnderflowException |
                           ↪   UnequalBracketsException e) {
78                                      e.printStackTrace();
79                              }
80                      });
81              }
82
83              // update the layer
84              private void changeLayer() throws StackOverflowException, StackUnderflowException,
                 ↪   UnequalBracketsException {
85                      // only change the layer if is not hidden by the user
86                      if (functionVisiblity) {
87                              Layer l;
88                              String f = this.functionText.getValueSafe();
89                              // if the function is empty remove the layer
90                              if (f.length() == 0) {
91                                      inputPane.removeLayer(ID);
92                                      return;
93                              }
94                              // if it begins with normal create a normal distribution layer
95                              if (f.startsWith("Normal(")) {
96                                      f = f.replaceAll("Normal\\(", "");
97                                      f = f.replaceAll("\\)", "");
98                                      String[] variables = f.split(",");
99                                      // ;
100                                     double  = 0;
101                                     // 2;
102                                     double 2 = 1;
103                                     // get the values for the mean and variance
104                                      = new Expression(variables[0], 'x').evaluate(0);
105                                     2 = new Expression(variables[1], 'x').evaluate(0);
106                                     // create the function
107                                     NormalDistribution normalDist = null;
108                                     normalDist = new NormalDistribution(, 2);
109                                     // create the layer with the specific color
110                                     l = new ExplicitXFunctionCartesianLayer(normalDist);
111                                     l.setColor(color);
112                                     // add the layer to the shared layer access
113                                     inputPane.putLayer(this.ID, l);
114                                     // terminate this function
115                                     return;
116                             }
117                             // determine the whether it is an explicit x or y function by counting their
118                             // respective instances
119                             // count the number of x instances
120                             int xInstances = f.length() - f.replaceAll("x", "").length();
121                             // count the number of y instances
122                             int yInstances = f.length() - f.replaceAll("y", "").length();
123                             if (xInstances > 0 && yInstances > 0) { // if it contains both x and y remove it
                           ↪   since it is implicit
```

114

```java
                                inputPane.removeLayer(ID);
                                return;
                        } else if (yInstances > 0) { // if it contains only y create a explicit function
                        ↪  in y
                                l = new ExplicitYFunctionCartesianLayer(f);
                                l.setColor(color);
                                inputPane.putLayer(this.ID, l);
                        } else if (xInstances > 0) { // if it contains only x create a explicit function
                        ↪  in x
                                l = new ExplicitXFunctionCartesianLayer(f);
                                l.setColor(color);
                                inputPane.putLayer(this.ID, l);
                        }
                } else { // if the function is hidden then remove the layer
                        inputPane.removeLayer(ID);
                }
        }

        // show/hide the function and update the visual identifier
        private void changeColor() throws StackOverflowException, StackUnderflowException,
        ↪  UnequalBracketsException {
                // clear the canvas
                gc.clearRect(0, 0, showFunctionColour.getWidth(), showFunctionColour.getHeight());
                if (!functionVisiblity) {
                        // fill it in color if showing the function
                        gc.fillRect(0, 0, showFunctionColour.getWidth(),
                        ↪  showFunctionColour.getHeight());
                }
                // draw a black border so it is clickable
                gc.strokeRect(0, 0, showFunctionColour.getWidth(), showFunctionColour.getHeight());
                // negate the visibility since it has changed
                functionVisiblity = !functionVisiblity;
                // remove or add the layer to show/hide it
                changeLayer();
        }

        // remove itself from the input pane
        private void remove() throws IOException {
                this.inputPane.removeExpressionBox(this.ID);
        }

        // set the id
        public void setID(int iD) {
                ID = iD;
        }

        // get the id
        public int getID() {
                return ID;
        }

        // set the parent input pane
        public void setInputPane(InputPane inputPane) {
                this.inputPane = inputPane;
        }

        // set the color of the function
        public void setColor(Color color) {
                this.color = color;
                gc.setStroke(color);
                gc.setFill(color);
                gc.fillRect(0, 0, showFunctionColour.getWidth(), showFunctionColour.getHeight());
        }

}
```

### 11.2.3 Input Pane

I then implemented the input pane class.

# Chapter 12

# Testing

# Chapter 13

# Evaluation

# Chapter 14

# Bibliography

[1] Desmos
    https://www.desmos.com/

[2] GeoGebra
    https://www.geogebra.org/

[3] GNU Octave
    https://octave.org/doc/interpreter/

[4] MATLAB - MATrix LABatory
    https://uk.mathworks.com/help/matlab/

[5] OpenGL
    https://www.opengl.org/documentation/

[6] RegEx - Regular Expressions
    https://en.wikipedia.org/wiki/Regular_expression

[7] Lua Example
    http://www.luaj.org/luaj/3.0/README.html

[8] Stack Based Programming
    https://en.wikipedia.org/wiki/Stack-oriented_programming

[9] JavaFX
    https://en.wikipedia.org/wiki/JavaFX#JavaFX_11

[10] JavaFX example without FXML
    https://en.wikipedia.org/wiki/JavaFX#JavaFX_application_example

[11] Creating a GUI using FXML
    https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm

[12] Shared Access between two or more Controllers
    https://stackoverflow.com/questions/29639881/javafx-how-to-use-a-method-in-a-controller-
    from-another-controller

[13] Property Class from JavaFX explained
    https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm

[14] JavaFX interface Hierarchy
    http://www.ntu.edu.sg/home/ehchua/programming/java/Javafx1_intro.html

[15] Border Pane Layout Explained
http://dammsebastian.blogspot.com/2012/03/javafx-20-layot-panes-borderpane.html

[16] Canvas Example
https://docs.oracle.com/javafx/2/canvas/jfxpub-canvas.htm

[17] Multithreading in OpenGL
https://stackoverflow.com/questions/11097170/multithreaded-rendering-on-opengl

[18] Generic Programming
https://en.wikipedia.org/wiki/Generic_programming

[19] Generics within Java
https://docs.oracle.com/javase/tutorial/extra/generics/index.html

[20] Duplicating objects in Java
https://stackoverflow.com/questions/12072727/duplicating-objects-in-java

[21] Serialization in Java
http://javatechniques.com/blog/faster-deep-copies-of-java-objects/

[22] Passing by Reference in Java
https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

[23] Checking String Equivalency in Java
https://stackoverflow.com/questions/513832/how-do-i-compare-strings-in-java

[24] Counting instances of a character within a String in Java
https://stackoverflow.com/questions/275944/how-do-i-count-the-number-of-occurrences-of-a-char-in-a-string

[25] Costs of instantiation of threads in Java
https://stackoverflow.com/questions/5483047/why-is-creating-a-thread-said-to-be-expensive

[26] The Thread Stack in Java
https://stackoverflow.com/questions/36898701/how-does-java-jvm-allocate-stack-for-each-thread

[27] Thread Creation Rate in Java
The article - https://stackoverflow.com/questions/2117072/java-thread-creation-overhead
The benchmark I used - https://stackoverflow.com/a/4371508

[28] Repeating a String in Java
https://stackoverflow.com/questions/1235179/simple-way-to-repeat-a-string-in-java

[29] The Call Stack
https://en.wikipedia.org/wiki/Call_stack
https://www.youtube.com/watch?v=Q2sFmqvpBe0

[30] Maximum Number of Recursive Calls in Java
https://stackoverflow.com/questions/4734108/what-is-the-maximum-depth-of-the-java-call-stack

[31] The Java implementation of the Double-precision floating-point format
https://docs.oracle.com/javase/7/docs/api/java/lang/Double.html

[32] The IEEE standard for undefined arithmetic behavior
https://www.doc.ic.ac.uk/~eedwards/compsys/float/nan.html

[33] Java Implementation of a list
https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

[34] Listeners in JavaFX
https://code.makery.ch/blog/javafx-2-event-handlers-and-change-listeners/

[35] Limitation of the `Math.pow()` function in Java
https://docs.oracle.com/javase/6/docs/api/java/lang/Math.html#pow(double,%20double)

[36] Snapshot feature in JavaFX
https://stackoverflow.com/questions/32708250/how-snapshot-the-entire-scene-in-javafx

[37] Context Menu in JavaFX
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ContextMenu.html

[38] Example of saving an image of a pane
https://stackoverflow.com/questions/38028825/javafx-save-view-of-pane-to-image

[39] Creating a right click menu in JavaFX
https://stackoverflow.com/questions/20635192/how-to-create-popup-menu

[40] How events work and are handled in JavaFX
https://www.tutorialspoint.com/javafx/javafx_event_handling.htm
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/MouseEvent.html
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/ScrollEvent.html

[41] Map, a dictionary-like structure in Java
https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

[42] HashMap, a dictionary like structure in Java
https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html