A level Computer Science Project

2022

My details

Name	Peter Metcalfe
Candidate number	[redacted]
School	The [redacted] School
Centre number	[redacted]

Contents

Analysis

Introduction

Students and teachers today are often expected to draw graphs for exams, textbooks and to practice real life examples too. Graphs are an important and a significant part of many subjects, especially at A level, in maths, physics, chemistry, biology, and many more…

To assist students and teachers in their use of graphs and maths in general, I intend to create an easy-to-use graphing calculator application.

Existing graphing software is limiting and complicated and I have experienced this first-hand. It is much easier for students to understand a question or problems by simply being able to visualise it and see what is happening. I intend to address this with my solution.

My application will enable the user to plot graphs and shapes with ease. I hope to make this a tool for students to further their understanding, as well as a tool for teachers to display solutions and problems. My program can include more tools and functions than existing software, making it more useful for students (like myself) taking subjects that go into more depth (such as further maths), where you may be looking at Argand diagrams as well as ordinary cartesian grids.

I now need to research what features students and teachers may want to see or use in a graphing calculator and what is important to include to make sure that the application is useful and valued.

Stakeholder(s)

As mentioned above, my application will aim to help both students and teachers. Therefore, I am going to make sure that I am in constant contact with both parties: my maths teacher (Mrs Evans) and my peers who specifically take subjects suited to using my application. This way, I will be able to get live feedback on what I am creating, ensuring a good result for my final product.

Different users are going to use the application in different ways. Students rely on calculators to perform calculations correctly and to make solving problems easier.

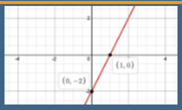
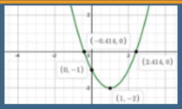
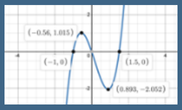
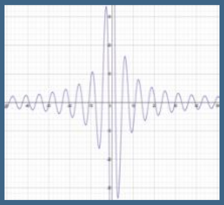
Whereas teachers rely on them to visualise and "solve tricky problems or functions" that they "don't know the shape of" (quoting Mrs Evans). The difference here is that teachers do a lot more of the simple sums in their head and don't need the calculator for 'easier' math. Another difference is that teachers like displaying graphs and functions to the class on their interactive whiteboards, so the whole class can learn at once.

Research

Prior knowledge needed

For this project I am going to need to know and be able to:
1. Find the roots of functions.
2. Find the turning points of functions.
3. Convert from radians to degrees.
4. Calculate the equation of a regression line.
5. Generate the Mandelbrot set.

Here is a table showing points 1 and 2.

Function	Y-intercept	Turning Point(s)	Root(s)	Graphical Representation
Linear $f(x) = ax + b$	$Y = b$	N/A	$X = -b / a$	
Quadratic $f(x) = ax^2 + bx + c$	$Y = c$	$X = (-c/2*b)$ $Y = (b*(-c/2*b)^2 + c*(*(-c/2*b)+d)$	$X = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	
Cubic $f(x) = ax^3 + bx^2 + cx + d$	$Y = d$	$X_1 = ((-2*b)+\text{sqrt}((4*(b^2))-(12*(a*c))))/(6*a)$ $Y_1 = a*((X_1)^3) + b*((X_1)^2) + (c*(X_1)) + d$ $X_2 = ((-2*b)-\text{sqrt}((4*(b^2))-(12*(a*c))))/(6*a)$ $Y_2 = a*((X_2)^3) + b*((_2)^2) + (c*(X_2)) + d$	Explained Below (*)	
Any other function… $f(x) = ax^{34} + yx^{12} + z$ (Generalised method)	$Y = z$ (the constant)	Solve the first derivative OR Approximate, by iteratively substituting X into the function and finding when f(x) changes from **increasing** to **decreasing** or vice versa	Approximate, by iteratively substituting X into f(x) and finding when it changes from **negative** to **positive** or vice versa	

(I derived the formulas for the turning points by solving the first derivative)

(*) Roots of cubic functions: The formula for these gets quite complicated as shown here.

$$x = \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) + \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}}$$
$$+ \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) - \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}} - \frac{b}{3a}.$$

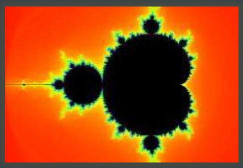
However, this formula only outputs one real root, even if there are three (depending on the function, it can have either 0, 1, 2, or 3 roots). It disguises the other two roots as complex numbers that you must transform into real numbers, which then become your answer.
To do this you use the fact that the other two roots are complex conjugates and that all three roots form an equilateral triangle when plotted. With this information, you can re-write the complex numbers in their modulus-argument form or polar form, which then cancel out leaving you with just the real part of the complex number. These are your other roots (Mathologer, 2019).

3. Converting from radians to degrees:
Angle in degrees = 180 * (Angle in radians / π)
This will be used when performing calculations with trigonometry and drawing the trig' graphs.

4. Calculating the equation of the regression line:
This is the mathematical linear line of best fit for a set of given points. To calculate the equation of a line you need the gradient and a point on that line (TLMaths, 2015).
Therefore, to calculate a point on the regression line, you take an average of all the points provided
= (average(x), average(y))
To find the gradient:
Sxy = $\sum$(x - average(x))*(y – average(y))
Sxx = $\sum$(x – average(x))^2
Gradient = Sxy / Sxx = $\frac{Sxy}{Sxx}$

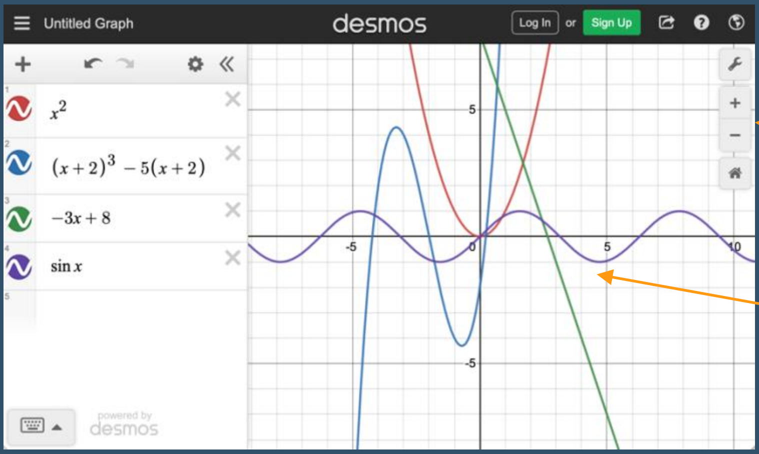
The equation of a line: $y_1-y_2 = m(x_1-x_2)$
Therefore the equation of the regression line: y - average(y) = $\frac{Sxy}{Sxx}$ (x - average(x))
y = $\frac{Sxy}{Sxx}$(x - average(x)) + average(y)

5. Generating the Mandelbrot Set:
To make this, you iteratively sub the complex number c in the function f(c) = $z^2 + c$ where z increments by some step (e.g. 0.1) on each pass. If f(c) is ever > 2 you know it is unstable, but if after a set limit of iterations, it still isn't greater than 2 you say it is stable and you plot that as black on an Angrand diagram (Numberpile, 2015).



Existing programs

There are two existing programs that I have researched:
Desmos = https://www.desmos.com/calculator

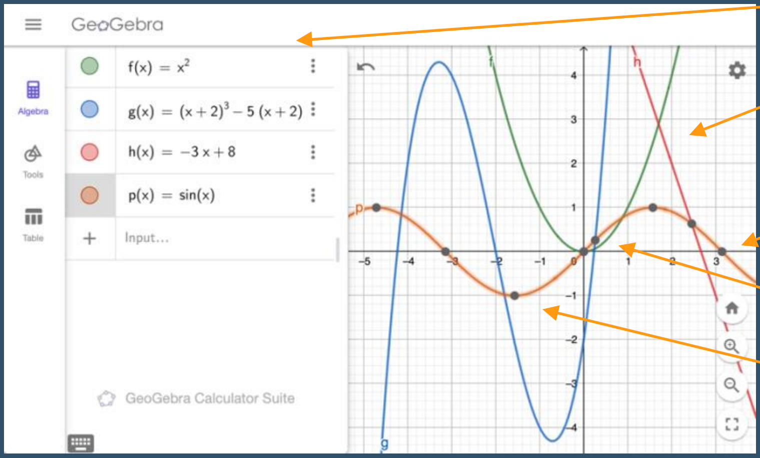


Online / web based

Zoom

Draw multiple functions on same axes

GeoGebra = https://www.geogebra.org/calculator



Label graphs

Colour graphs

Find roots

Find intersections

Find turning points

Issues / limitations:

- Does not calculate complex numbers
- Does not calculate regression line
- Does not draw Mandelbrot set
- Does not draw / sketch
- Does not work offline

I hope to include all the features **and** limitations listed above in my application, with the aim to make my program more useful and helpful (except for making it web-based). These current programs do not consider complex numbers (to date), so this is where I will try to make my program excel. One limitation of my program will be that it will only be in two dimensions, whereas GeoGebra does have the ability to draw in three dimensions. However, this could be something I could consider adding if I have enough time.

Both these programs are written in JavaScript and have HTML and CSS elements to them too. They perform the processing on the client side for faster speeds and better performance. Desmos can store graphs, this involves processing on their servers and allows the user to close the tab and reopen the same graph setup. As my program is an application run on the user's device, my code is all client side processing and I will try to allow the user to save their graph setups too, this may just be by adding the ability to save an image of the graph.

Stakeholder and user needs

Having discussed with my client (Mrs Evans – Math teacher), I discovered some of the things she would like to see in an improved graphical calculator.

She said that a graphical calculator should be able to calculate, at least some basic sums and be able to draw, at least some basic graphs. She also added that some of the things that a graphical calculator should output are the turning points, roots, intercepts and asymptotes.

She added that current calculators are too complicated, especially for beginners and that it would be advantageous if it could be simpler and quicker to use. Furthermore, these calculators are usually difficult to navigate and locate all the functions, so this is also something I must consider when designing mine.

My ideas

In addition to the basic functioning calculator, I would like to involve complex numbers. I feel having the ability to process imaginary numbers and plot on an argand diagram is a unique feature and something that will help students taking further maths at A level. This may mean making a separate section in my application dedicated to dealing with complex numbers. That way, users are not forced into trying to use or understand them if they don't wish to, but still have the option.

Alongside this, I have thought about including the Mandelbrot set as this is also plotted on an argand diagram and links well with complex numbers. It would certainly be a unique feature to my application and would add educational and artistic befits for the user.

Another idea is to add the ability to plot all kinds of graphs, similar to Excel. These may include scatter graphs, bar charts, pie charts, cumulative frequency graphs and box plots. This would allow the user to visualise a variety of statistical distributions helping students taking statistics.

Other possible things to include are 'solvers'. For example, a page that solves SUVAT equations or simultaneous equations or matrices, etc… This would be helpful to both students and teachers as it would allow them to double check their calculations on the dedicated 'solver' page and possibly show them how to solve it if they get stuck.

My approach will involve lots of prototyping and constant feedback from my stakeholders, if I am to achieve all my aims. It will be important to adjust my design to keep everyone's expectations aligned.

Features for the Proposed Solution

The feature	Including / Removing	Justification
Drawing functions	Including	This is essential to any graphical calculator
Drawing functions on same axis	Including	This is useful for comparing different functions against each other
Label functions	Including	Useful to distinguish between functions
Colour functions	Including	Useful to distinguish between functions
Find roots	Including	A key feature to any calculator
Find turning points	Including	A key feature to any calculator
Find intersections with axis	Including	Easy to include, so why not?
Find intersections with other functions	Removing	Not so easy to include – possibly something to add if I have time
Zoom	Including	Useful to get a better view of the functions drawn on axis
Complex numbers	Including	Unique feature, useful for further maths students
Regression line	Including	Useful for scientific experiments with data
Mandelbrot set	Including	Unique feature useful for educational and artistic purposes
Sketching	Removing	I will only include this if I have enough time, as it is not essential
Calculator	Including	Essential for a calculator
Drawing statistical graphs	Removing	I will only include this if I have enough time, as it is not essential
‘Solvers’	Including	This is a very useful tool, but I am not sure how many I will be able to complete.

The above table lists all the features I am considering for my application and my justification for including them. Some topics are not essential, but I will try and include them if time permits.

Limitations

When drawing functions there are few limitations I may encounter; the speed at which I draw the graph and the resolution of the graph. These two things are inversely proportional. The faster I draw the graph, the less the resolution and the higher the resolution the lower the speed. There is always a compromise between these two variables, so I will collect user feedback to help me ascertain what is felt to be acceptable.

Furthermore, the resolution of the graph effects how accurate the approximations are for the turning points and roots. For this reason, I will need to carefully choose a resolution that offers an acceptable waiting time to draw the function but still outputs an accurate enough result for the roots and turning points.

Time is also a limiting factor. Whilst this project is well defined, there are always more features to include.

Computational Methods

My application involves a significant amount of math, whether this is solving calculations, approximating answers or performing statistical tests. This makes a computer perfectly suited for this application, for many reasons:

Firstly, they are reliable and reduce human input and error. This eliminates the possibility of performing a calculation wrong as this is what computers were literally designed to do – perform calculations.

Secondly, it decreases the amount of time needed to solve problems as computers are much faster at processing data than humans are. It would take a human hours to approximate the roots of large, irregular polynomials but a computer only milliseconds.

Thirdly, computers are designed to store data for long periods of time. This makes them perfect for storing graphs or plots that need to be preserved. Storing such data on paper is much less reliable, easier to lose, and requires space.

Finally, computers around the world are connected via the internet, so can share data almost instantaneously. This would allow the user of my application to share their graphical plots or statistical data with others, anywhere in the world seamlessly and reliably. It would also allow for automatic backs-up too.

Hardware and Software Requirements

For this project, the hardware consists of a keyboard, mouse and monitor.

The software will be written in Python. Python is a useful language to use when performing mathematical tasks due to the vast amount to useful libraries that can be used and implemented. I did consider using C or C++ for the added speed when it comes to performing many iterative calculations but decided against the idea due to the simplicity of Python.

The libraries I plan to use involve 'Tkinter' for the main GUI components, 'Matplotlib' for the graphs and visual elements and 'Math' for some of the mathematical calculations.

I am currently running a Windows operating system and will be writing my code using Visual Studio Code and the traditional Python IDLE.

Success Criteria

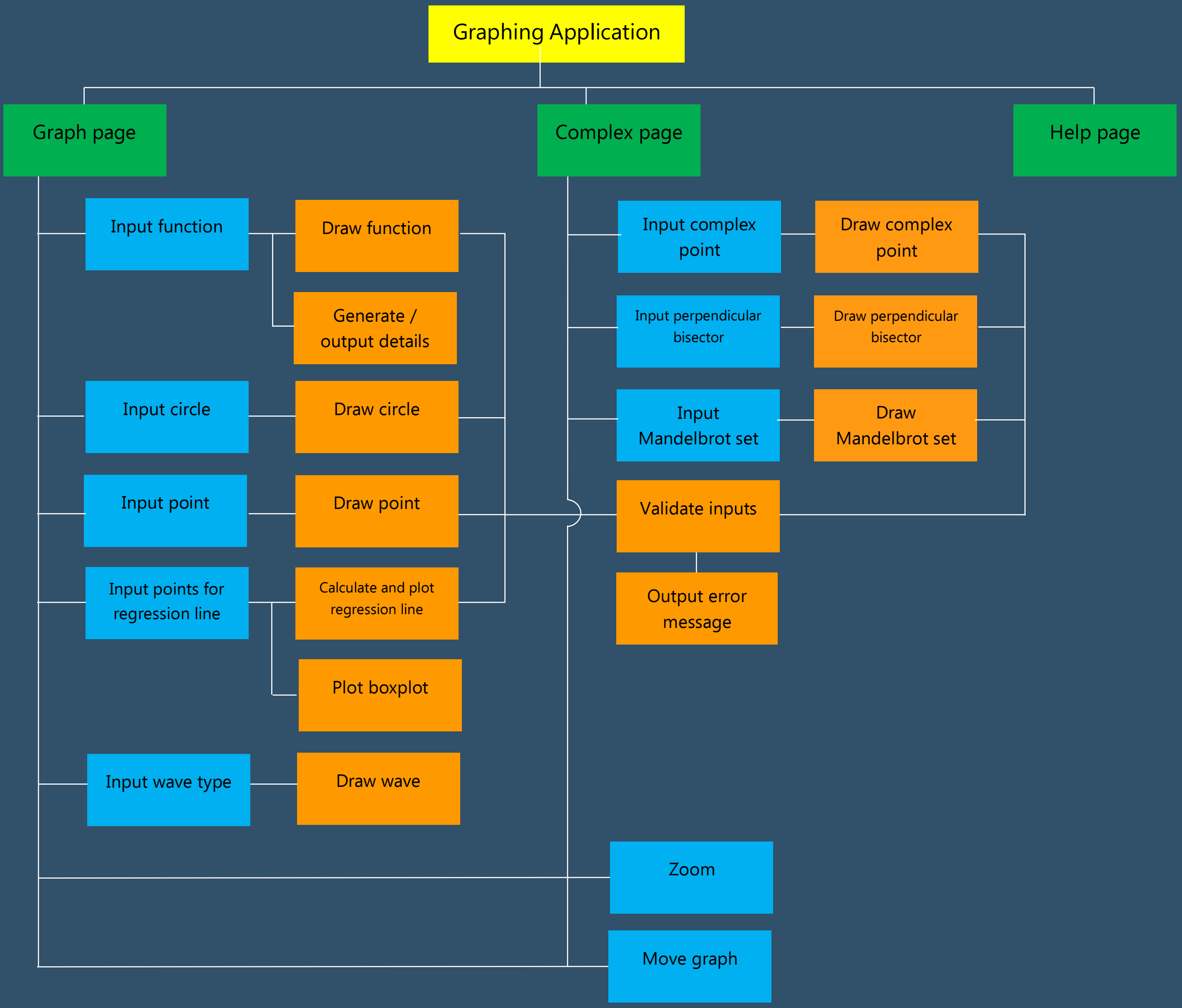
No.	Criteria	Justification
1	Have 5 functions allowing the user to freely select the graphing function they would like to perform. This should include as a minimum: polynomials, circles, regression lines and complex numbers.	This gives the user control and allows them to achieve the result they were looking for. The more graphing functions, the more freedom the user has and the more useful my program will be.
2	The user can input graph functions to be processed.	This is vital for the program to be able to know what to output, including the graph and extra details like roots and intersections.
3	The user can clearly see the graph they inputted on the screen and be able to make necessary observations form it. This can include the ability to manipulate the graph by zooming and moving it around the screen.	The reason for this is to really allow the user to make the most of the application. Being able to visualise the inputted graph functions should aid the user in their understanding.
4	The user should be able to clear the set of axes.	This is important if the user plots several graphs on the same axis as it could get messy. The user should be able to clear this without having to restart the application.
5	Allow the user to plot up to cubic functions.	The purpose of this is that the user may have several types of graphs they would like to input, and I would like my program to be able to handle this.
6	The program should output data about the inputted graph, including roots, turning points and intersections.	Extra information about the graph will be useful for the user as my stakeholders has mentioned. It not only makes my program more useful but also helps the user with mathematical problems too.
7	The program must recognise and alert the user when the user has inputted something that is invalid.	This is important so that the user knows they have done something incorrect. My program must be interactive and malleable in this way to prevent a lack of communication between the user and my program, resulting in a lack of interest and usage.
8	The user must be able to distinguish between different graphs drawn on the same axis, by colour coordination and / or labelling.	This allows the user to use the application easier and with less effort. It stops the set of axes getting out of control and messy when lots of graphs are drawn.
9	There should be a calculator within the application to allow the user to perform quick calculations.	This is a necessary function of any calculator and should be expected.

10	Functions should be plotted with a high enough resolution and that approximated answers are accurate to at least two decimal places.	This ensures the plot is an accurate representation of the graph and that the approximated calculations are accurate and usable.
11	The user to have the option to draw functions on the imaginary plane / argand diagram.	This should allow users explore new areas of maths or help those who require the imaginary plane to plot in it.
12	The user should have the ability to quite the program whenever they wish to.	The user should not feel trapped withing the application and should have the freedom the quit.

Ultimately, my goal is to make a calculator that is intuitive to use and simple to understand.

Design

Structure Diagram



🟩 Green boxes represent the main pages of my application.

🟦 Blue boxes represent sub-pages / functions with more of a concise purpose.

🟧 Orange boxes are an indication of the objects or functions of code that I will write.

Structure diagram explanation
The diagram shows that my application will have 3 main pages, leading form the home page.
The 'graph page' will have all the functions I am offering, on the traditional cartesian coordinates system, whereas the 'complex page' will be plotted on the complex plane. Finally, the 'help page' will guide the user through different functionalities of my program if they need further instructions.

From each of the main pages there will be sub-pages where the user inputs their values to be drawn. The corresponding functions of code will process this input and draw the result.

The 'zoom' and 'move' functions will be on the 'graph' and 'complex' page and will be used, by the user, to manipulate the graphs they draw. This allows the user to make clearer observations and provides more useability.

Modules and Processes

The graph page will consist of all the functions that I am offering to the user, which can be plotted on the usual cartesian coordinates grid.

First we have the drawing function page. Here the user will have the ability to input functions up to at least cubic equations, maybe more depending on time constraints. This should then draw the graph of the inputted function and output the useful data about the graph. To draw the graph, I plan to use the equation of the function to substitute in a value for 'x'. Then, as it is a function, it will output a value, this being the 'y' coordinate. I can then plot this coordinate and will repeat this for every value of 'x' on the axis, until I have drawn a line.

The graph 'details' function will calculate all the roots and turning points of the inputted function and then the user should have the option to view this information if they wish. The calculations for this have already been discussed, however I may need to optimise this, as some calculations may take a significant amount of time to process.

Next, there is drawing circles, this should take input values for the radius and centre coordinate, then plot the circle on the axis. This allows for a wider variety of questions / problems to be drawn using my program, making it more useful for users. To draw the circles, I will repeat the same process as drawing functions but I will have to complete two passes, once for the top half of the circle and once for the bottom half.

The draw point module will simply draw a point (small circle) at the inputted coordinate. This gives the user the ability to plot specific points that may be useful for their problem. This ability also lets the user plot many points, maybe data collected in a science experiment, and see what shape the points make. This power, to observe the shape of graphs at an instant, will be very useful for students and teachers to come to conclusions about their experiments.

The regression line function will work in the same way as the drawing function but first the equation of the line must be calculated, using the maths in the 'Prior knowledge section'. There is also another function connected to this page that allows the user to generate or draw a boxplot. This will involve calculating some statistical variables from the inputted data (like the median and inter quartile range) and then plotting two rectangles accordingly.

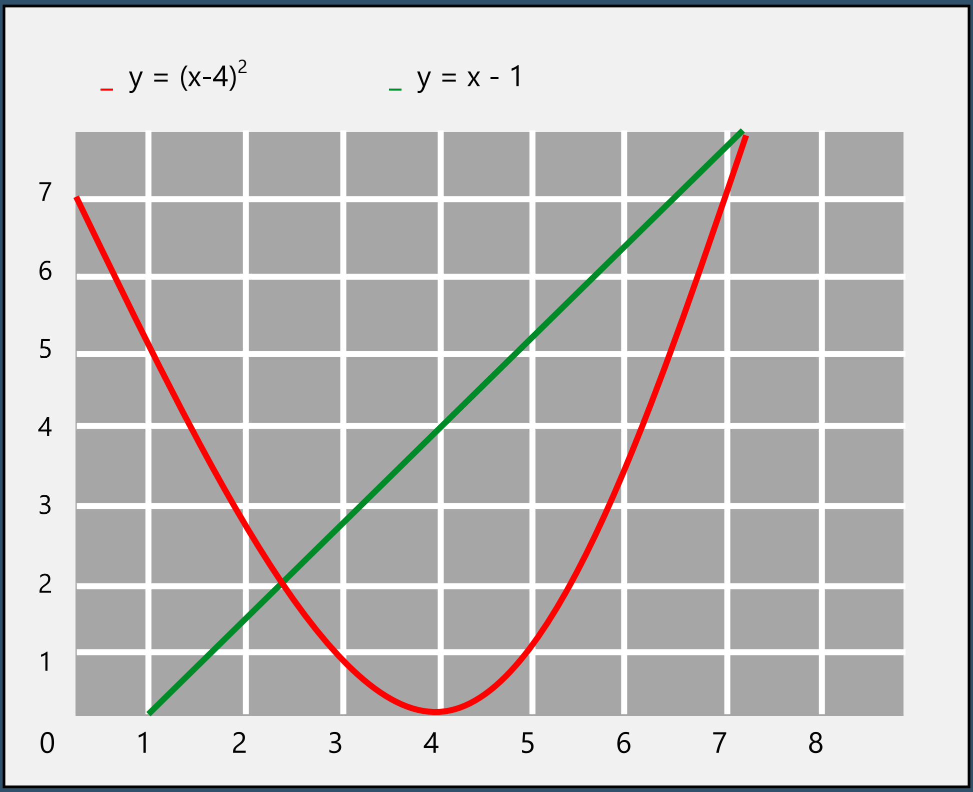
Moving on to the complex plane page, the first function will be to plot complex numbers. This is a useful tool for students taking Further Maths studies as they will be using complex numbers. My program will output the modulus and argument of these complex numbers, which I know from experience could be very useful as these terms are used frequently throughout the course.

Finally, the Mandelbrot set. This will output the same image, as there is only one Mandelbrot set, however it will allow the user to control the colour scheme used and the resolution. The reason for this, is there are multiple points to calculate and the smaller the resolution the quicker it will take to render. This ability to picture the Mandelbrot set may come in useful for visualisation and possibly artistic endeavours too.

I plan on generating the Mandelbrot set by processing every point on the Argand diagram and deciding (in the algorithm) whether it is stable or not, in other words, whether to colour it black or not. Then I should generate the whole image, however, this may need adjusting as I predict this may take a long time to complete.

Other less important modules may involve the 'colour changing' function that changes the colour of the graphs being plotted, the ' input validation' object that will check the input values for my program and the 'clear axes' function that resets axes.

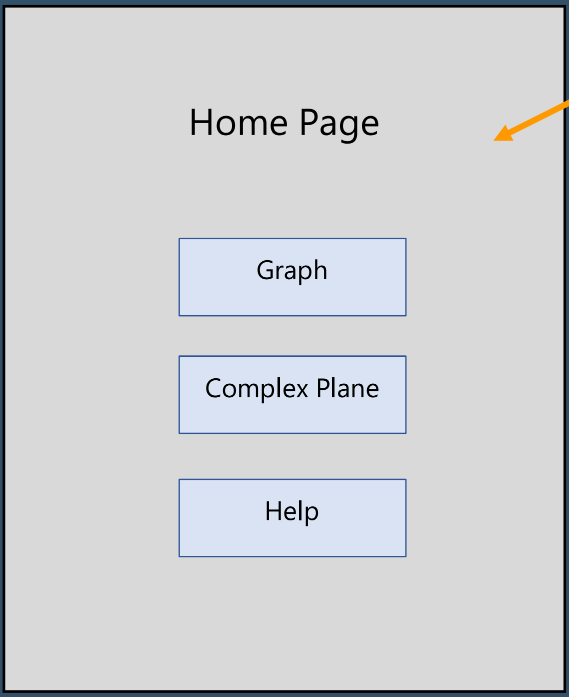
Interface design



This is an example of what the graph might look like.

Notice each line plotted has a colour coordinated label, so the user can keep track.

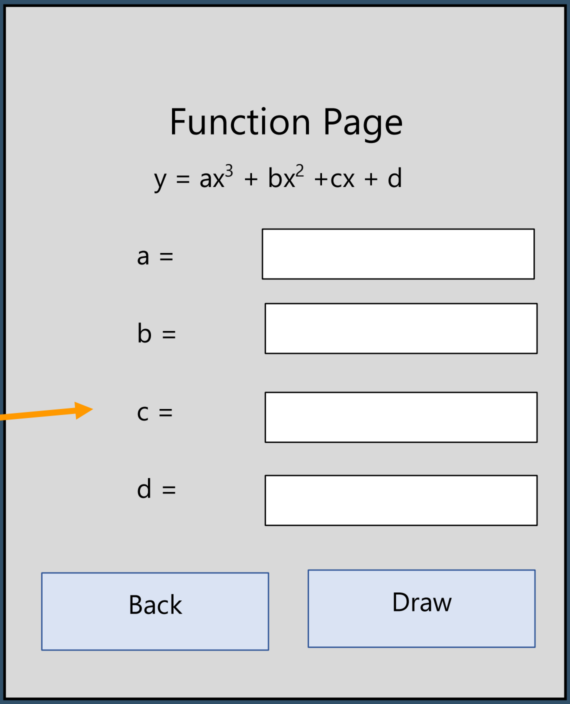
Here, to make things as simple as possible, I have tried to abstract all the unnecessary information. When the user requires information, I will provide it as a pop-up window so that the user has the option to close it again. These leaves the cleanest user interface possible, as suggested by my stakeholders.



This is the initial design for the application, with 3 simple clear buttons on the home page.

This is the polynomial page, where the user will input their values of a, b, c, d.

All the pages where the user must input values should have this same style.



I have tried to make the user interface as simple to navigate as possible. I have sorted all the functions into two main categories that the user can switch between depending on what their goal is. My aim was to make this easy and quick for the user to navigate, while also being informative and useful.

Validation of inputs

As my GUI contains text-based user inputs, I need to validate these inputs before I can use them to do any calculations or drawings. This is what I will check for:

Validation check	How it works	Example
Left blank	Checks whether the text bar was left empty. If it is, it makes the input equal to zero (sanitisation).	Input: ' ' Output: 0
String check	If the input requires an integer / float, this will reject any strings or characters and alert the user.	Input: 'hello' Output: [Pop-up message] ['This input is not valid']
Range check	This checks to see whether the inputted values are within the range of the axes to be drawn on.	Input: '100000' Output: [Pop-up message] ['Input too large']
Sign check	Inputs cannot be negative, like the circle radius.	Input: '-54' Output: [Pop-up message] ['Input must be greater than 0']

Pseudocode

Draw function

```
a , b , c , d = check_vars( "function", a , b , c , d )          #a, b, c, d are parts of inputted function

step = 0.1                                                        #'step' and 'axes' are global variables
axes = 200
x = -axes
Xvalues = [ ]
Yvalues = [ ]

WHILE x <= axes:
    Yvalues.append( (a * x^3 )+ (b * x^2 ) + (c * x) + d )        #For a cubic function
    Xvalues.append( x )
    x += step
END WHILE

change_colour()                                                   #Changes the colour of the line
PLOT( Xvalues ,Yvalues )                                          #Draws the function as a line of points
```

This algorithm needs to start at the left-hand side of the axis (e.g. x = -200) and on each loop increment x by some step. On each loop, the algorithm needs to calculate the y coordinate relating to that x value. To do this, it will sub the value of x into the inputted function. It will then store all the points calculated in a list to be plotted once the loop has ended at the right-hand side of the axis (x = 200).

The variable 'step' controls the size of the increment of x, each time the loop is performed. I have made this a variable as I may want to change this depending on the type of graph. This variable is going to be responsible for how quickly, but also how well, it draws the graph. As bigger 'steps' means more jagged curves.

I have also made the axis size a variable ('axes'), as I may also want to change this, either at the user's discretion or I may set it depending on the type of graph. If the graph inputted is only small, I only need a small set of axes to draw it on, so this will save time.

I have also made sure to include the 'check_vars()' routine that will validate the input variable and return them as floats, to be used in calculations.

This algorithm will be one function of a 'draw' object. This object may also contain the other algorithms that follow, such as the 'draw circle' function and the 'draw point' function.

Draw circle

```
INPUT radius
INPUT centreX
INPUT centreY

radius , centreX , centreY = check_vars( "circle", radius , centreX , centreY )

step = 0.1
axis = 200
Xvalues = [ ]
Yvalues = [ ]

x = -axis
WHILE x <= axis:
        Yvalues.APPEND( $( radius^2 - x^2 – ( 2 * centreX * x ) – centreX^2 )^{1/2}$ – centreY )
        Xvalues.APPEND( x )
        x += step
END WHILE

x = axis
WHILE x <= -axis:
        Yvalues.APPEND( $- ( radius^2 - x^2 – ( 2 * centreX * x ) – centreX^2 )^{1/2}$ - centreY )
        Xvalues.APPEND( x )
        x -= step
END WHILE

change_colour()
PLOT( Xvalues, Yvalues )
```

For the algorithm to draw the circle, it is very similar to the one that draws functions, but I have to do it twice as the circle will give two outputs each time I substitute 'x' in, to to find 'y'. Consequently, I start at the left-hand side of the x-axes, increment 'x' until I reach the right-hand side, then go back again, decreasing 'x' until I am back at the left-hand side. Doing this only once produces a semi-circle.

The vairables 'axes', 'step', 'Xvalues' and 'Yvalues' occur in many of my algorithms so I plan on making these global variables. Therefore, every algorithm will have access to the same variable.

Draw regression line

```
Xvalues = [ ]
Yvalues = [ ]
Xpoints = [ ]
Ypoints = [ ]
X = 0
Y = 0
Sxx = 0
Sxy = 0

FOR _ IN user_inputs:                        #loops for the number of points the user has
        INPUT FLOAT = x
        INPUT FLOAT = y
        Xpoints.APPEND( x )                  #adds these points to list
        Ypoints.APPEND( y )
END FOR

FOR x IN Xpoints:                            #finds sum of the points
        X += x
        Y += x
END FOR

X = X / LENGTH( Xpoints )                    #X is set to the average of the 'x' points
Y = Y /  LENGTH ( Ypoints )                  #Y is set to the average of the 'y' points

FOR p IN RANGE( 0 , LENGTH( Xpoints ) ):     #loops for the number of points
        Sxy += ( Xpoints[ p ] - X ) * ( Ypoints[ p ] - Y )
        Sxx += ( Xpoints[ p ] -X )^2
END FOR
m = Sxy / Sxx                                #finds gradient
c = Y – ( c * X )                            #finds y intercept

WHILE x <= axes:
        Yvalues.APPEND( (m * x) + c )        #calculates all the points of the line
        Xvalues.APPEND( x )
        x += step

END WHILE

change_colour()
PLOT( Xvalues , Yvalues )                    #outputs line
```

To find the equation of the regression line, the user will input at least 2 coordinates and then this algorithm will ouput the result using the math descirbed in the 'Prior knowledge' section. It can then plot this graph using a similar alogrithm to the 'draw function' psueodcode.

Draw wave

```
INPUT type                                          #user inputs which wave: sin, cos or tan

IF type == "sine":
        WHILE X < axes:
                Yvalues.APPEND( sin(X) * π / 180 )  #add points for sine
                Xvalues.APPEND( X )
                X += step

ELSE IF type == "cosine":
        WHILE X < axes:
                Yvalues.APPEND( cos(X) * π / 180 )  #add points for cos
                Xvalues.APPEND( X )
                X += step

ELSE type == "tangent":
        WHILE X < axes:
                Y = ( tan(X) * π / 180 )            #calculate y point, but don't add it yet
                IF Y < axes and Y > -axes:
                        Yvalues.APPEND( Y )         #only add if its on axes, as tan goes to infinity
                        Xvalues.APPEND( X )
                ELSE:
                        PLOT( Xvalues , Yvalues )   #plot this much, then repeat
                        Xvalues = [ ]
                        Yvalues = [ ]
                END IF
                X += step
END IF

IF Xvalues AND Yvalues:
        change_colour()
        PLOT( Xvalues , Yvalues )                   #plot sin or cos if user picked these
END IF
```

In this algorithm, I take a user input to deduce what wave they would like to draw. Depending on this result, I calculate the points for this trig' function. It is slightly different if the 'type' inputted is 'tan' due to the different shape of this function. The tangent function goes off to infinity every 180° and therefore has asymptotes too. That is why I have designed the algorithm to plot this function in stages between each asymptote.

Draw point

```
INPUT x
INPUT y

x, y = check_vars(“point”, x, y)

change_colour()

PLOT( x ,  y )
```

This is a simple algorithm that takes inputs for ‘x’ and ‘y’, validates them and then plots them.

Draw complex point

This is similar to the ‘draw point’ algorithm except I am plotting it on the complex plane.
On the comple plane, ‘x’ and ‘y’ are refered to as ‘a’ and ‘b’.

```
INPUT a
INPUT b

a, b = check_vars(“point”, a, b)

change_colour()

PLOT( a ,  b )
```

Perpendicular bisector

```
INPUT a1
INPUT b1
INPUT a2
INPUT b2

a1, b1, a2, b2 = check_vars( “pBisector”, a1, b1, a2, b2 )

PLOT( [a1, a2] , [b1, b2] )                                   #plots the points, so clearer to see

m = (b1 – b2) / (a1 – a2)                                     #calculate gradient
m = -( (m)^-1 )                                               #find perpendicular gradient
mdptx = (a1 + a1) / 2                                         #find midpoint
mdpty = (b1 + b2) / 2

OUTPUT “y = ” + str( m ) + “x + ” + str( ( mdptx * m) + mdpty )   #ouput bisector equation

change_colour()

PLOT( lst_function( m , ( mdptx * m) + mdpty )  )             #draw function equation
```

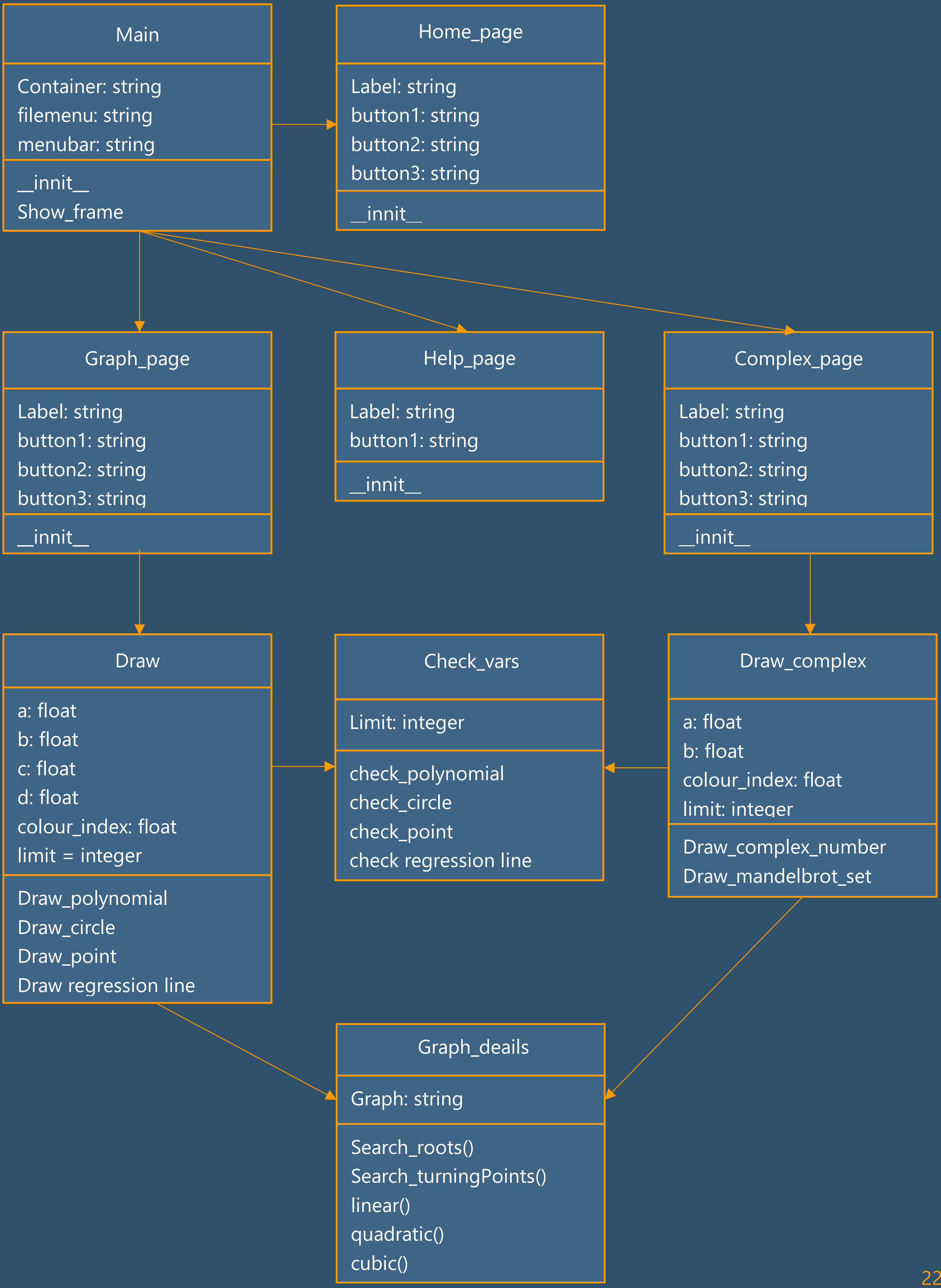
This algorithm is using some simple math to calculate the perpendicular bisector from two inputted points. It then outputs this to the user and draws the function.

Mandelbrot set #'MS' = Mandelbrot set

```
stable = FALSE
colour_before = " "
colour = " "
WHILE x <= ( axis - step ):
        x += step                                           #x and y increments, to get every point
        y = -axes
        WHILE y <= axes:
                y += step
                Xscale = x / ( 0.5 * axes )                 #makes it bigger as the MS is only between -2 → 2
                Yscale = y / ( 0.5 * axes )
                answer = COMPLEX( 0 , 0 )
                FOR i IN RANGE( 0 , 100 ):
                        answer = ( answer$^2$ + COMPLEX( Xscale , Yscale ) )        #MS sum
                        IF ( answer.real$^2$ ) + (answer.imag$^2$) >= 4:         #circle with radius of 2, stable?
                                stable = FALSE
                                IF INPUT == "red":                  #user picks form colour scheme
                                        FOR c IN RANGE( 0 , LENGTH( MBredScale ) ):
                                                IF i > MSredScale[ c ][ 0 ]:        #picks colour gradient
                                                        colour = MBredScale[ c ][ 1 ]
                                                        BREAK
                                                END IF
                                        END FOR
                                END IF
                                BREAK
                        ELSE:
                                stable = TRUE
                        END IF
                END FOR
                IF stable == TRUE:                                  #if stable, colour black
                        colour = 'black'
                END IF
                IF colour_before != colour OR y >= axes:
                        PLOT( x , y , colour_before )               #plots points in that colour
                        colour_before = colour
                END IF
                IF y >= axes:
                        PLOT( ( x + step ) , -axes )
                        colour_before = " "
                END IF
        END WHILE

END WHILE
```

In this algorithm, I am going through the loop, checking every coordinate to see whether it is 'stable' or not. I then plot this point in a specific colour. This is going to be very time consuming so I will probably have to optimise this code later on. Here I am also only using a red colour scheme, 'MSredScalse' but I hope to have more. A scheme is a 2D array wich contains every colour used and the iteration it should be used on.

Class Diagrams



Variables and Justification

Variable	Data Type	Justification
Button	String	This valiable stores the the data that shows the buttons. I will reference this variabale whenever I need to change the appearance or position of a button.
a, b, c, d	Float	These will store the inputs from the user, for eache part of the polynomial. I need a separate variable for each, to be able to perform calculations with each individual part.
Colour_index	Integer	This is where I will store the value for the colour I will use to draw the next line on the axes. Each time I draw a line, I will increment this value so that each line is a different colour.
Limit	Integer	Here I will store the size of the axes, in other words, the greatest value of X and Y (the limit).
Graph	String	Stores the whole string that the user inputs when plotting a function. I will then process this variable to identify the values of the previous variables a, b, c and d.

Test Data

I have a lot of functions and objects to complete. I have made a list of milestones that I need to reach, to focus my programming and to help structure my coded solution in an organised manner:

1. Creating the user interface
2. Drawing functions (Taking and validating an input and drawing the associated graph)
3. Drawing circles
4. Drawing points
5. Drawing regression lines
6. Displaying detailed information along with the ouputted plot
7. Drawing the Mandelbrot set

	Milestone 1: User Interface	
Test No.	**What is being tested and inputs**	**Expected outputs**
1	That buttons do as they say. Test – Press them.	Button performs as expected. e.g. links to the correct page.
2	Text inputs work and reject unusable inputs. Test – Enter numbers, strings, 0, negatives.	Usable input leads to associated response. Unusable inputs leads to popup message.
3	Check-boxes perform as expected. Test – tick them, and untick them.	The check-boxes should have the intended result.
4	Scroll bars work as intended. Test – Scroll them.	The scroll bars should have the intended result.

	Milestone 2: Drawing functions	
Test No.	**What is being tested and inputs**	**Expected outputs**
1	Test linear functions. e.g. 2x – 3	The correct line should be displayed. e.g. gradient of 2 and y-interept of -3.
2	Test quaratic functions. e.g. 2x^2 +3x -5	The correct line should be plotted in the correct position on the axis.
3	Test cubic functions e.g. x^3 +2x^2 +3x +5	The correct line should be plotted in the correct position on the axis.
4	Test the 3 previous functions but with negative coefficients.	The correct line should be plotted in the correct position on the axis.
5	Test the colour of the graph changes when a new function is plotted.	The colours should cycle as different functions are plotted.
6	Test limits of axis. e.g. make the y-intercept greater than or less than the axis.	If the graph doesn't fit on the axis, either display an error message or resize the axis and plot.

	Milestone 3: Drawing circles	
Test No.	**What is being tested and inputs**	**Expected output(s)**
1	Draw cicle at origin. e.g. radius = 10, centre X = 0, centre Y = 0.	This should draw a circle at the origin with a radius of 10.
2	Draw circles with a differrent centre. e.g. radius = 10, centre X = 2, centre Y = -5.	This should draw a circle at this position with that radius.
3	Test inputing negative radii. e.g. Radius = -23	This should ouput a suitable error massage, as this is not possible to draw.
4	Test axis limits. E.g. Input a centre off the axis.	If the graph does not fit on the axis, display an error message.
5	Test the colour of the graph changes when a new circle is plotted.	The colours should cycle as different things are plotted.

	Milestone 4: Drawing points	
Test No.	**What is being tested and input(s)**	**Expected output(s)**
1	Draw a point. e.g x= 10, y =2	This should draw a point at (10,2).
2	Test negatives. e.g. x=-10, y-2	This should draw a point at (-10,-2).
4	Test point off the axes scale. e.g. x=200, y=1000	This should display a suitable error message.
5	Test inputting a string. e.g. "Hello"	This should output a suitable error message.

	Milestone 5: Regression line	
Test No.	**What is being tested and input(s)**	**Expected output(s)**
1	Test what values can be inputted. e.g. x=10, y=5	Programm should accept this input and ask the user for the next point.
2	Test inputtig a string instead of an integer. e.g. "test"	This should output a suitable error message.
3	Test the correct regression line is outputted after at least 2 points have been inputted.	The correct regression line should be outputted.
4	Test point off the axes scale. e.g. x=200, y=1000	This should show a suitable error message.

	Milestone 6: Outputting data about graph	
Test No.	**What is being tested and input(s)**	**Expected output(s)**
1	Test if the outputted roots are correct for linear graphs. E.g. Input = $2x$	E.g. Output = 0
2	Test if the outputted roots are correct for quadratic graphs. E.g. Input = $(x+4)^2$	E.g. Output = -4
3	Test if the outputted roots are correct for cubic graphs. E.g. Input = $5x^3 + 5x^2 + 5x + 5$	E.g. Output = -1
4	Test if the outputted roots are approximately correct for other graphs. E.g. Input = $x^7 - 2x^3 + 2x – 2$	E.g. Output = ~1.158
5	Test If the outputted turning points are corect. E.g. Input = $5x^3 + 5x^2 + 5x + 5$	E.g. Output = (0,5)
6	Test if the outputted graph type is correct. E.g. Input = $x^3 + 3x^2 - 2$	E.g. Output = 'Cubic'
7	Test that the colour of the graph changes.	Every gragh should have a different colour to the one plotted before it.

	Milestone 7: Mandelbrot Set	
Test No.	**What is being tested and input(s)**	**Expected output(s)**
1	Test the plotted Mandelbrot set is acurate and to scale.	Should be plotted on imaginary axis between 2 and -2.
2	Test that the inputted colour scheme is correclty used.	If the user selects the Red scheme, then the MS should be Red.

Post-Development Testing

I intend to perform alpha testing, checking my program works, in line with my success criteria.

Beta testing will allow my stakeholders to offer their feedback and for me to make any small changes. I plan on providing them with a small feedback survey to fill out, once they have had the chance to use the app.

I will also perform integration testing as laid out below.

Post-development test	**Justification**
Test the program can open and close at the user's discretion.	This gives the user the freedom to use the application and gain some trust / control.
Test that all the buttons work and navigate to the correct page or perform their associated task.	This ensures the user can explore the application freely and that all the button functions can be used extensively and reliably.
Test that the axes are to scale. Test plotting y = x. This should be at exactly 45°. Circles plotted should be perfectly circular.	It is essential that a graphing calculator is accurate and to scale, allowing the user to trust and use the application for important tasks.
Test that infinitely long numbers cannot be inputted.	This would be a problem as when labels are added to the graph, the number would obstruct the graph from being viewed.
Test that the application window can be resized, but not too little that content is lost.	This would be annoying for the user and therefore needs to be catered for.

As well as testing, I intend to observe, where possible how people interact with my user interface. This will allow me to adjust subtleties that I am unable to pick up from my survey, improving my GUI further.

Developing The Coded Solution

Prototyping

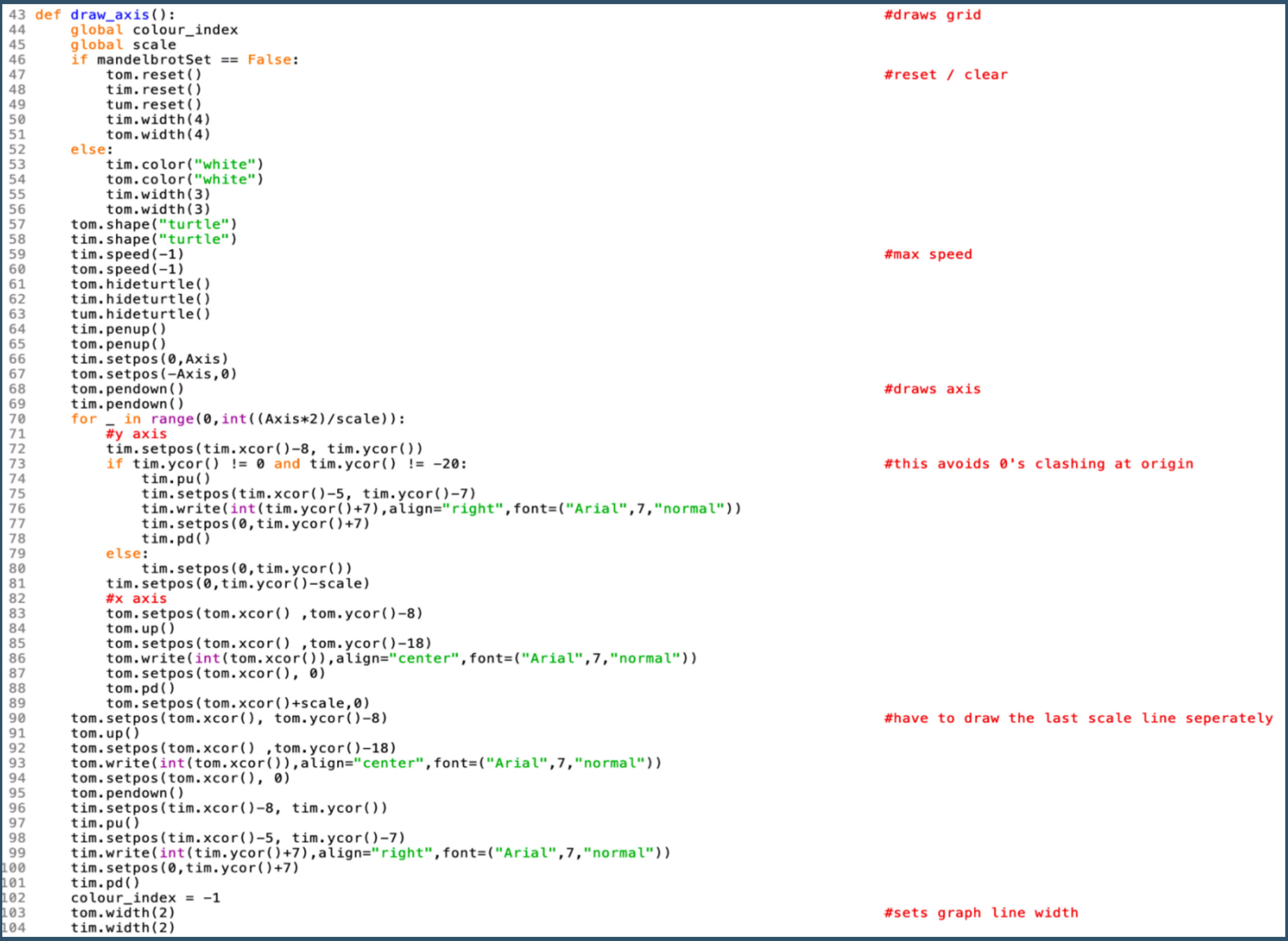
The first thing I am going to do, is create a prototype of my application. This will have simpler graphics and slower speeds but I am going to do this to make sure I can get all the math working first. I am going to write some 'simple' Python code using the 'Turtle' library that will be able to do all the functions I have described above. I will then use parts of this code to create my final product which will be of a higher quality and run faster.

This effectively means creating each milestone twice. Once for the prototype (a), and once for the final product (b). I will do all the testing, verification and user feedback in version 2.

For the prototype I am NOT using an object orientated approach and will be using the Python IDLE. That way it is easier to distinguish between code written for the prototype (code with white background) and code for the real application (written in Visual Studio Code with black background).

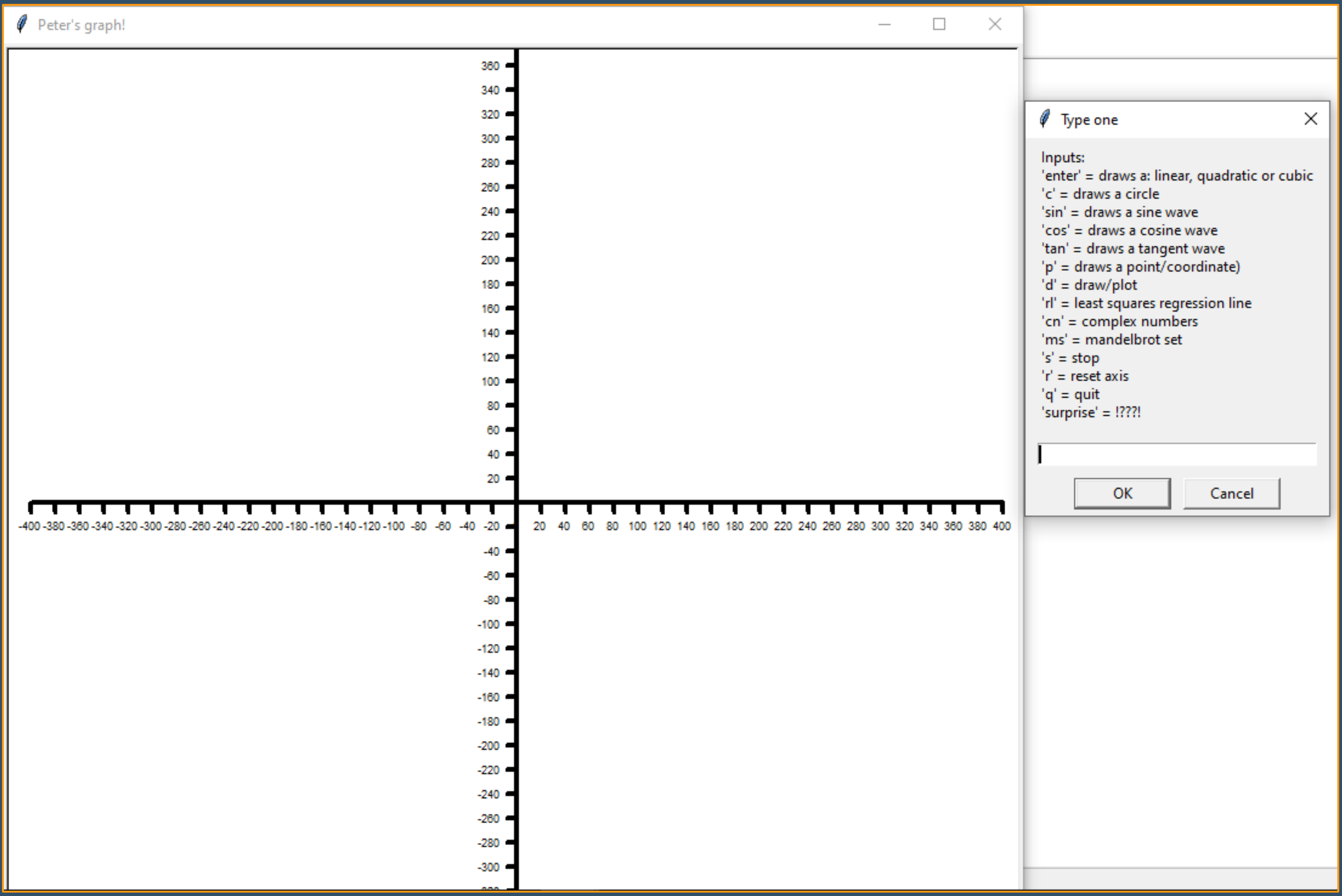
Milestone 1a – User Interface

For this prototype I am going to keep the interface simple as this is not what I need to test. I will create a simple 'Turtle Canvas' to draw graphs on and a simple input bar to perform tasks.



```
def draw_axis():                                                                                 #draws grid
    global colour_index
    global scale
    if mandelbrotSet == False:
        tom.reset()                                                                              #reset / clear
        tim.reset()
        tum.reset()
        tim.width(4)
        tom.width(4)
    else:
        tim.color("white")
        tom.color("white")
        tim.width(3)
        tom.width(3)
    tom.shape("turtle")
    tim.shape("turtle")
    tim.speed(-1)                                                                                #max speed
    tom.speed(-1)
    tom.hideturtle()
    tim.hideturtle()
    tum.hideturtle()
    tim.penup()
    tom.penup()
    tim.setpos(0,Axis)
    tom.setpos(-Axis,0)
    tom.pendown()                                                                                #draws axis
    tim.pendown()
    for _ in range(0,int((Axis*2)/scale)):
        #y axis
        tim.setpos(tim.xcor()-8, tim.ycor())
        if tim.ycor() != 0 and tim.ycor() != -20:                                                #this avoids 0's clashing at origin
            tim.pu()
            tim.setpos(tim.xcor()-5, tim.ycor()-7)
            tim.write(int(tim.ycor()+7),align="right",font=("Arial",7,"normal"))
            tim.setpos(0,tim.ycor()+7)
            tim.pd()
        else:
            tim.setpos(0,tim.ycor())
        tim.setpos(0,tim.ycor()-scale)
        #x axis
        tom.setpos(tom.xcor() ,tom.ycor()-8)
        tom.up()
        tom.setpos(tom.xcor() ,tom.ycor()-18)
        tom.write(int(tom.xcor()),align="center",font=("Arial",7,"normal"))
        tom.setpos(tom.xcor(), 0)
        tom.pd()
        tom.setpos(tom.xcor()+scale,0)
    tom.setpos(tom.xcor(), tom.ycor()-8)                                                         #have to draw the last scale line seperately
    tom.up()
    tom.setpos(tom.xcor() ,tom.ycor()-18)
    tom.write(int(tom.xcor()),align="center",font=("Arial",7,"normal"))
    tom.setpos(tom.xcor(), 0)
    tom.pendown()
    tim.setpos(tim.xcor()-8, tim.ycor())
    tim.pu()
    tim.setpos(tim.xcor()-5, tim.ycor()-7)
    tim.write(int(tim.ycor()+7),align="right",font=("Arial",7,"normal"))
    tim.setpos(0,tim.ycor()+7)
    tim.pd()
    colour_index = -1
    tom.width(2)                                                                                 #sets graph line width
    tim.width(2)
```

This code makes produces a simple set of axes that I can now draw graphs on.

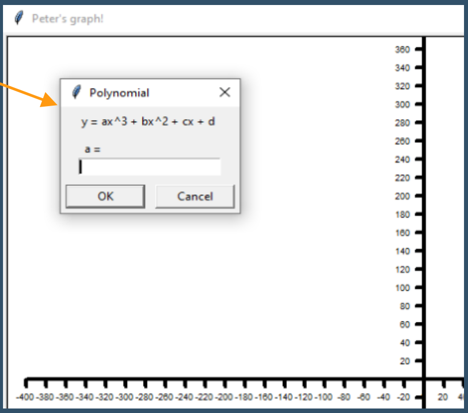


```
Inputs:
'enter' = draws a: linear, quadratic or cubic
'c' = draws a circle
'sin' = draws a sine wave
'cos' = draws a cosine wave
'tan' = draws a tangent wave
'p' = draws a point/coordinate)
'd' = draw/plot
'rl' = least squares regression line
'cn' = complex numbers
'ms' = mandelbrot set
's' = stop
'r' = reset axis
'q' = quit
'surprise' = !???!
```

Milestone 2a – Drawing functions

I now plan to get the 'turtles' to draw curves and graphs on the set of axes shown on the previous page. For the purpose of this prototype, I will only go up to cubic equations. So, the user can input function(s) with exponents up to and including cubic. This will be in the format $ax^3 + bx^2 + cx + d$, where a, b, c and d are the constants that the user will input.

Here is the simple interface that I implemented to get these inputs.



```
y = ax^3 + bx^2 + cx + d

a =
```

Now that I have the function to draw, I need to draw it.
I thought of a couple of different ways of doing this:
- I could scan the x axis, substituting in every x value into the function, to find the y value, then plot this.
- I could do the same thing as the point above, but to increase resolution, differentiate the function, find the gradient at this x value, and draw a small line, at this gradient, between the previous x value and the next.
- I could do the same thing but scan the y axis instead of the x axis. However, this will often output two results, for example if there is a 'U' shaped curve, so this will involve two plots.

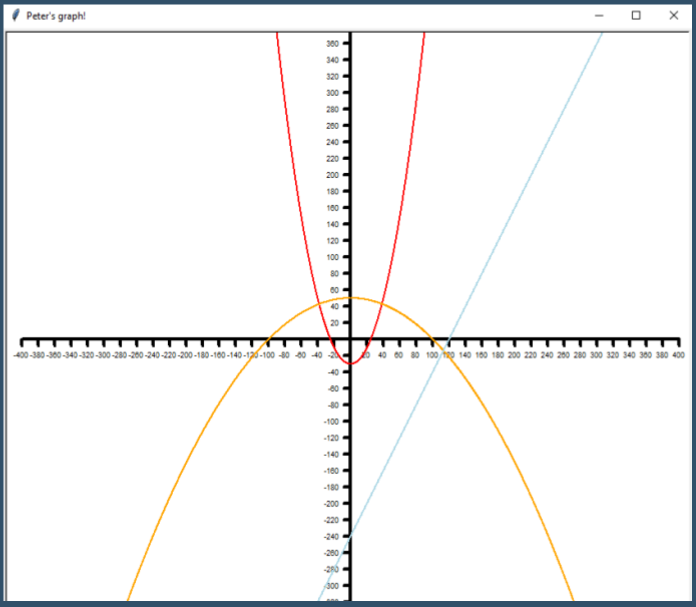
I went with my first option as it was the easiest to implement and the resolution could be increased by decreasing the step at which I scan the x axis (however, this increases the time taken to draw it).

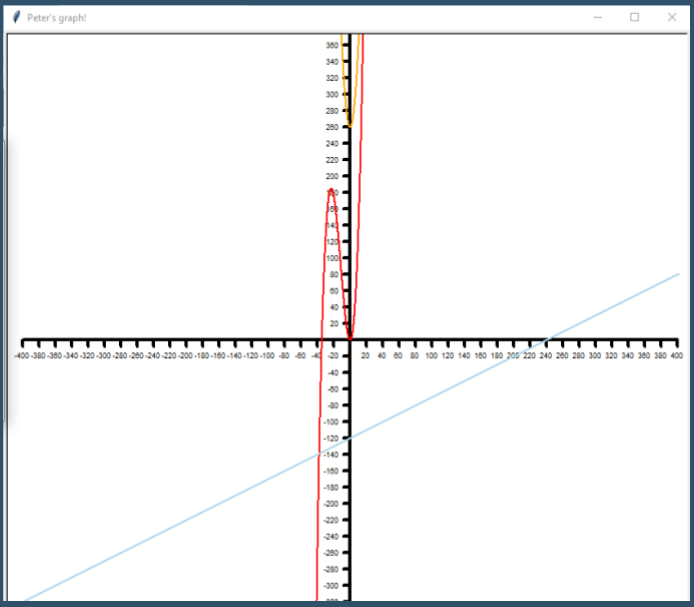
I implemented each method, but you were unable tell them apart, so I chose based on complexity.

```
while tom_drawing or tim_drawing:
    tim_step = step
    tom_step = step
    if tim.xcor() < Axis and tim.ycor() < Axis and tim.xcor() >-Axis and tim.ycor() > -Axis:
        tim.setpos(tim.xcor()+tim_step, (a*((tim.xcor()+step)**3))+(b*((tim.xcor()+tim_step)**2))+(c*(tim.xcor()+tim_step))+d)
    else:
        tim_drawing = False
    if tom.xcor() < Axis and tom.ycor() < Axis and tom.xcor() >-Axis and tom.ycor() > -Axis:
        tom.setpos(tom.xcor()-tom_step, (a*((tom.xcor()-tom_step)**3))+(b*((tom.xcor()-tom_step)**2))+(c*(tom.xcor()-tom_step))+d)
    else:
        tom_drawing = False
```

This code follows my pseudo code algorithm, with these adaptations which make it work in reality:
- 'tim' and 'tom' are the 'turtles' that draw the graph.
To make it quicker I used two 'turtles' that draw half the graph each.
- 'step' is the rate at which they scan the x axis. E.g. (step =1) 1 → 2 → 3 (step = 0.2) 1 → 1.2 → 1.4
- 'Axis' is the size of the axes; in this case it is = 200.
- '.xcor()' / '.ycor()' is a built-in function that returns the 'turtles' position.

The results:





Milestone 3a – Drawing circles

This is the code that I wrote to calculate all the points on the inputted circle.

```
while nx < limit:
    try:
        ny = sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx += step
nx = limit
while nx > -limit:
    try:
        ny = -(sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx -= step
```

This works just as the pseudo code does.

'cx' and 'cy' refer to the centre coordinates of the circle (centre X and centre Y) and 'r' refers to the radius; these are all inputs of the user.

These is one difference to the pseudo code though. I ended up using the ‘try’ and ‘except’ instructions. This is because I wanted to avoid errors being thrown up when trying to square root negatives.

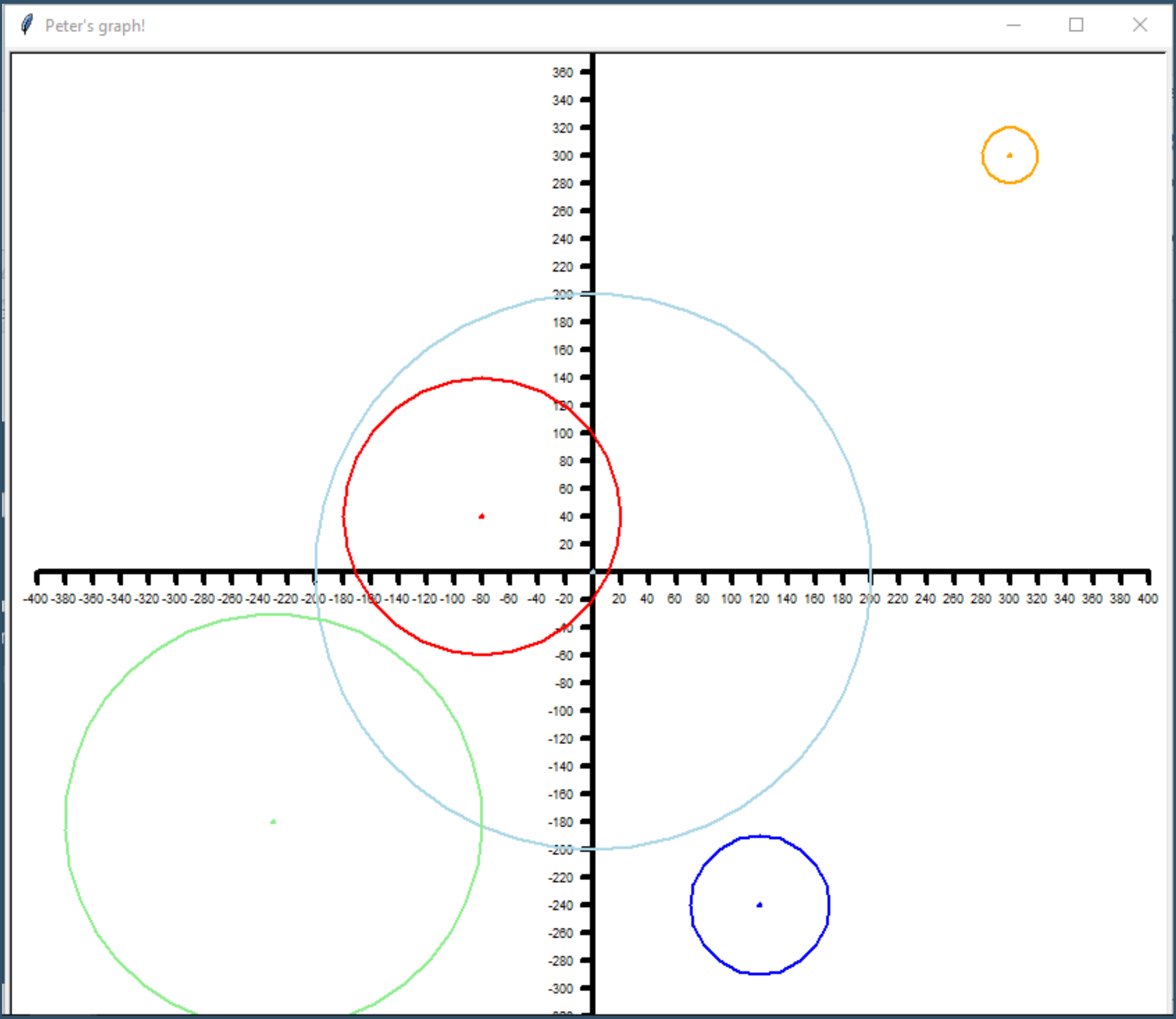
I also found that this code was very slow as it iterates over the whole axes twice. To improve this, I scanned the axis once, to draw the top half of the circle and recorded the first and last values of x used. I then used this to draw the rest of the circle, just between these two recorded points. Here is the updated code:

```
while nx < limit:                              #search whole axes
    try:
        ny = sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx += step

x_range.append(x[0])                           #here i record the new limits
x_range.append(x[-1])

nx = x_range[1]
while nx > x_range[0]:                         #only look between these new limits
    try:
        ny = -(sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx -= step
```

But I then thought of a more efficient way. I realised I could use the inputs from the user to set the limits. Here is the final code:

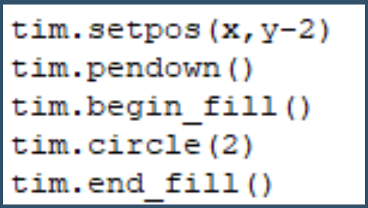
```
nx = cx - r
while nx < cx + r:                         #just checks the circle size
    try:
        ny = sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx += step

nx = cx + r
while nx > cx - r:                         #litteraly only checks the circle
    try:
        ny = -(sqrt( ((r)**2) - ((nx)**2) - (2*cx*nx) - ((cx)**2) ) ) - cy
        real = True
    except:
        real = False
    if real == True and ny < limit and ny > -limit :
        x.append(nx)
        y.append(ny)
    nx -= step
```

The output:


Peter's graph!
360
340
320
300
280
260
240
220
200
180
160
140
120
100
80
60
40
20
-400 -380 -360 -340 -320 -300 -280 -260 -240 -220 -200 -180 -160 -140 -120 -100 -80 -60 -40 -20
20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
-40
-60
-80
-100
-120
-140
-160
-180
-200
-220
-240
-260
-280
-300

Milestone 4a - Drawing points

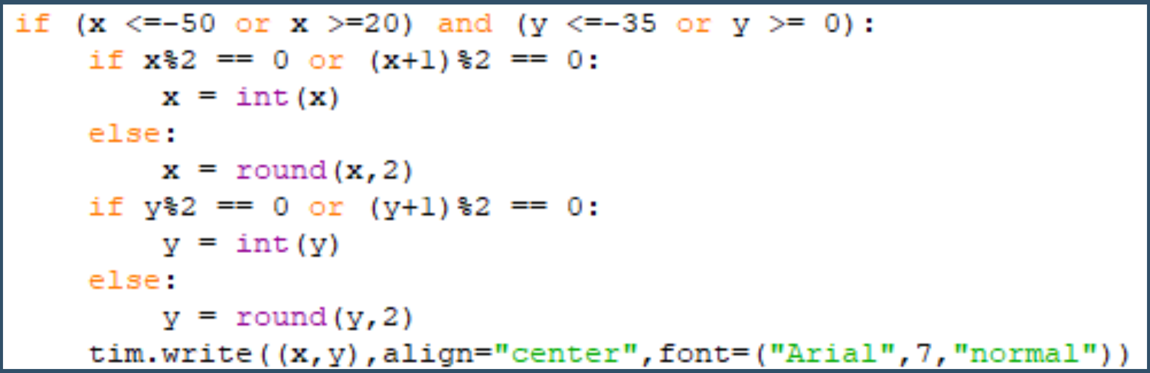
This was simple to implement.



```
tim.setpos(x,y-2)
tim.pendown()
tim.begin_fill()
tim.circle(2)
tim.end_fill()
```

I set "tim's" position to 'y-2' because I am using the built-in function to draw these small circles to represent points. When I do this, with the circle having a radius of 2, it starts drawing the circle 2 units below the circle's centre, which is the position of the point I am trying to represent. I then counter this by drawing the circle 2 units below the actual inputted position.

I can then also give these points labels, by doing this:

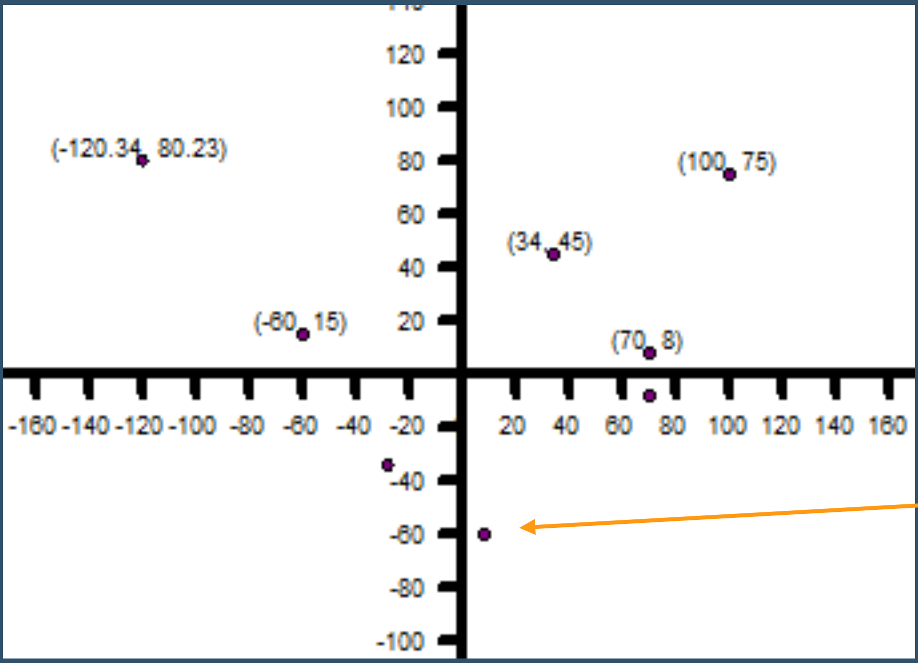


```
if (x <=-50 or x >=20) and (y <=-35 or y >= 0):
    if x%2 == 0 or (x+1)%2 == 0:
        x = int(x)
    else:
        x = round(x,2)
    if y%2 == 0 or (y+1)%2 == 0:
        y = int(y)
    else:
        y = round(y,2)
    tim.write((x,y),align="center",font=("Arial",7,"normal"))
```

Here I am writing the points position above the point so that the user can accurately see the point's position. However, I found that when the points were too close to the axes and the labels overlapped and were hard to read. I solved this by using the main 'If' statement above to decide where to write the label, depending on how close the point was to the axes.

The other 'if' statements after this, round up the inputted number (if it is a large decimal). This prevents the user from covering the whole axes with one long label. These subtle changes made the GUI display cleaner.

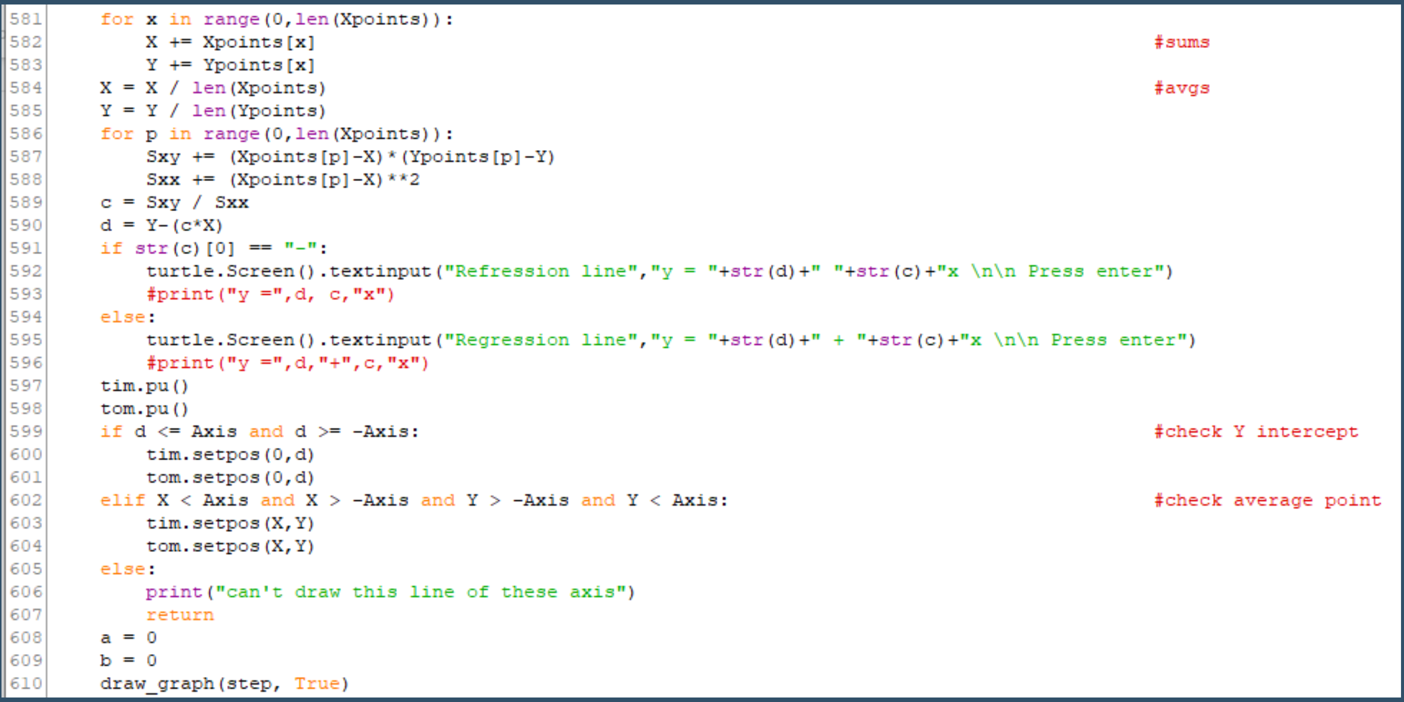
The results:



Points near axis don't show labels

Milestone 5a – Drawing the regression line

I drew this using the ‘draw polynomial’ function I created for milestone 2a. The more challenging part is that I must mathematically create the function of the line to be plotted, rather than just use the inputted one.



```
581        for x in range(0,len(Xpoints)):
582            X += Xpoints[x]                                                          #sums
583            Y += Ypoints[x]
584        X = X / len(Xpoints)                                                         #avgs
585        Y = Y / len(Ypoints)
586        for p in range(0,len(Xpoints)):
587            Sxy += (Xpoints[p]-X)*(Ypoints[p]-Y)
588            Sxx += (Xpoints[p]-X)**2
589        c = Sxy / Sxx
590        d = Y-(c*X)
591        if str(c)[0] == "-":
592            turtle.Screen().textinput("Refression line","y = "+str(d)+" "+str(c)+"x \n\n Press enter")
593            #print("y =",d, c,"x")
594        else:
595            turtle.Screen().textinput("Regression line","y = "+str(d)+" + "+str(c)+"x \n\n Press enter")
596            #print("y =",d,"+",c,"x")
597        tim.pu()
598        tom.pu()
599        if d <= Axis and d >= -Axis:                                                 #check Y intercept
600            tim.setpos(0,d)
601            tom.setpos(0,d)
602        elif X < Axis and X > -Axis and Y > -Axis and Y < Axis:                      #check average point
603            tim.setpos(X,Y)
604            tom.setpos(X,Y)
605        else:
606            print("can't draw this line of these axis")
607            return
608        a = 0
609        b = 0
610        draw_graph(step, True)
```

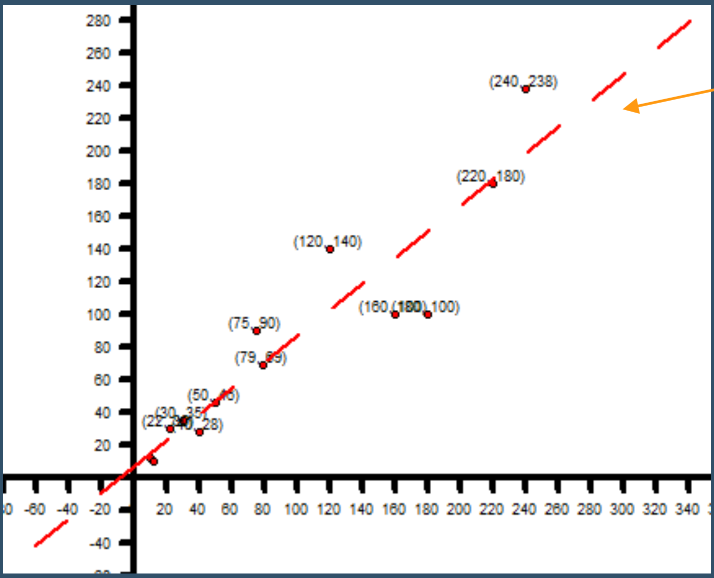
The first half of this (lines: 581 - 596) calculates the equation of the ‘regression line ‘and the other half validates and plots it. To test if the output was correct I would compare it with the results that my real physical calculator outputs and check they were the same.

The second half checks that this line I have calculated, fits on the axes and sets the “turtles’” to their designated positions to start drawing the line.

Finally, on the last line (610), I use the polynomial function I created before. This takes two parameters, the step which is set to 1 in this case, and a Boolean variable on whether the line to be drawn should be dashed.

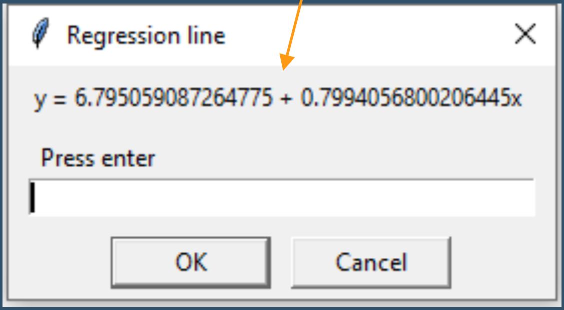
I am also using the ‘Draw points’ function to show all the inputted points.

Results:



I inputted the points, and it drew this line, the line of best fit.

It also outputted the equation of this line.



Regression line

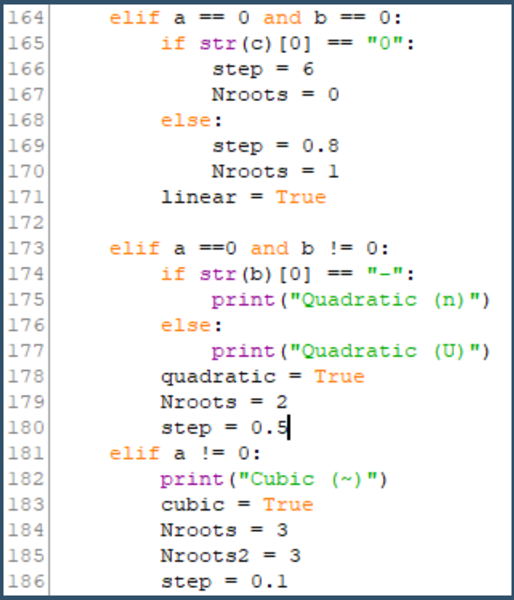
y = 6.795059087264775 + 0.7994056800206445x

Press enter

Milestone 6a – Graph details:

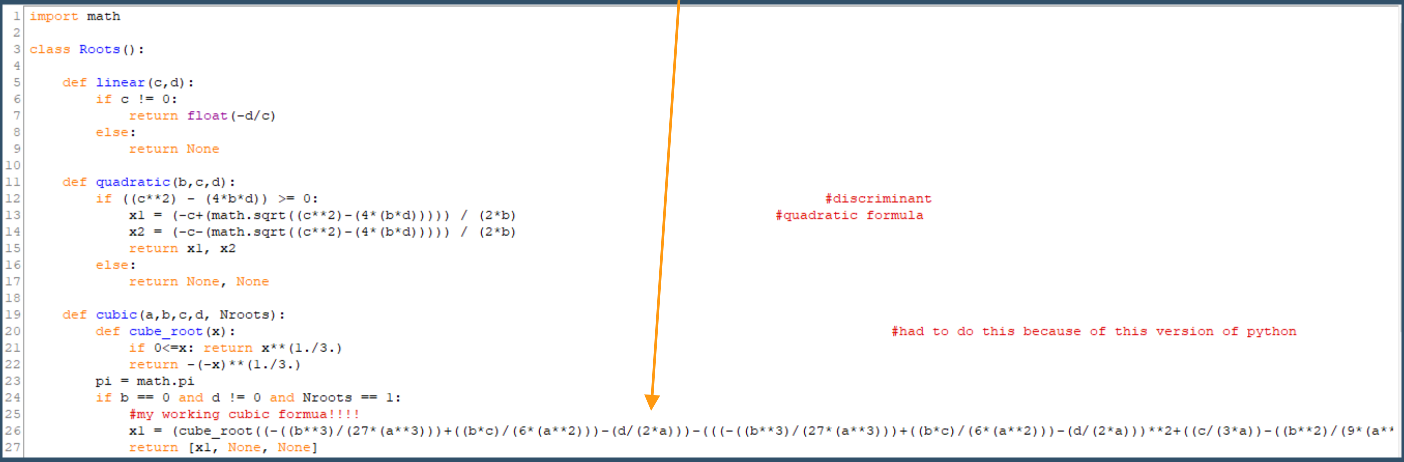
When the polynomial function is called, I also want to output the 'graph details'.

First I need to find what type of graph it is. The inputs being a, b, c, and d, in the form $ax^3 + bx^2 + cx + d$. By looking at which if these variables are 0, I can decide what type of graph it is.
For example,
if a = 0, I know it cannot be cubic.



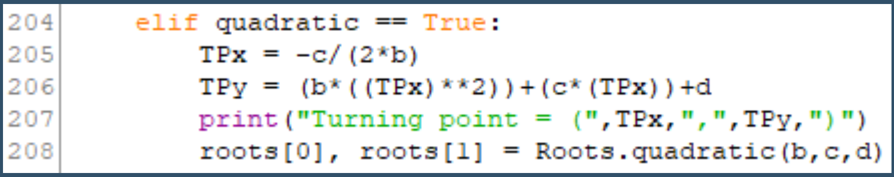
```python
    elif a == 0 and b == 0:
        if str(c)[0] == "0":
            step = 6
            Nroots = 0
        else:
            step = 0.8
            Nroots = 1
        linear = True

    elif a ==0 and b != 0:
        if str(b)[0] == "-":
            print("Quadratic (n)")
        else:
            print("Quadratic (U)")
        quadratic = True
        Nroots = 2
        step = 0.5
    elif a != 0:
        print("Cubic (~)")
        cubic = True
        Nroots = 3
        Nroots2 = 3
        step = 0.1
```

Next, I need to find the roots. So, to simplify things, I created a class called 'Roots()' where I will use this code to calculate the roots, depending on what type of graph it is. I have created three functions for the three types of equation linear, quadratic and cubic. In each function I calculate the roots and return them. I underestimated how long it would take to get this complex math working.

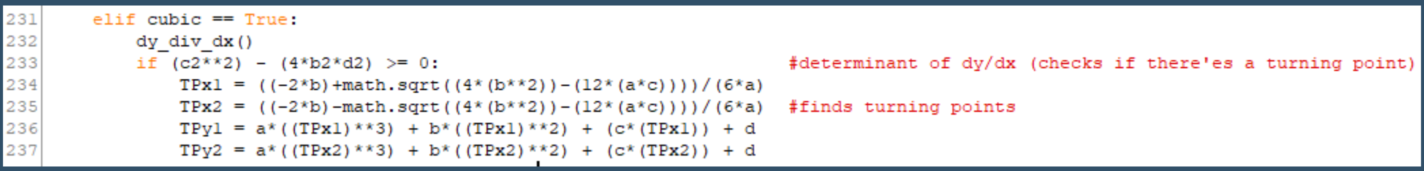


```python
import math

class Roots():

    def linear(c,d):
        if c != 0:
            return float(-d/c)
        else:
            return None

    def quadratic(b,c,d):
        if ((c**2) - (4*b*d)) >= 0:                                              #discriminant
            x1 = (-c+(math.sqrt((c**2)-(4*(b*d))))) / (2*b)                  #quadratic formula
            x2 = (-c-(math.sqrt((c**2)-(4*(b*d))))) / (2*b)
            return x1, x2
        else:
            return None, None

    def cubic(a,b,c,d, Nroots):
        def cube_root(x):                                                                   #had to do this because of this version of python
            if 0<=x: return x**(1./3.)
            return -(-x)**(1./3.)
        pi = math.pi
        if b == 0 and d != 0 and Nroots == 1:
            #my working cubic formua!!!!
            x1 = (cube_root((-((b**3)/(27*(a**3)))+((b*c)/(6*(a**2)))-(d/(2*a)))-(((-((b**3)/(27*(a**3)))+((b*c)/(6*(a**2)))-(d/(2*a)))**2+((c/(3*a))-((b**2)/(9*(a**
            return [x1, None, None]
```

Next, I needed to find the turning points (if any) of the curves that were drawn.
To do this I use the math explained in the 'Prior knowledge section'.

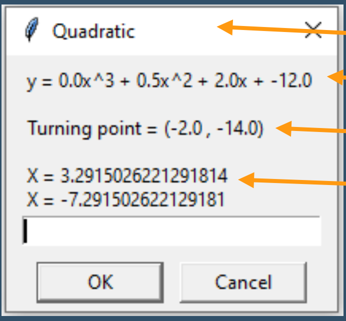


```python
    elif quadratic == True:
        TPx = -c/(2*b)
        TPy = (b*((TPx)**2))+(c*(TPx))+d
        print("Turning point = (",TPx,",",TPy,")")
        roots[0], roots[1] = Roots.quadratic(b,c,d)
```



```python
    elif cubic == True:
        dy_div_dx()
        if (c2**2) - (4*b2*d2) >= 0:                                  #determinant of dy/dx (checks if there'es a turning point)
            TPx1 = ((-2*b)+math.sqrt((4*(b**2))-(12*(a*c))))/(6*a)
            TPx2 = ((-2*b)-math.sqrt((4*(b**2))-(12*(a*c))))/(6*a)      #finds turning points
            TPy1 = a*((TPx1)**3) + b*((TPx1)**2) + (c*(TPx1)) + d
            TPy2 = a*((TPx2)**3) + b*((TPx2)**2) + (c*(TPx2)) + d
```

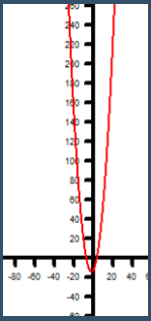
The results:



```
Quadratic

y = 0.0x^3 + 0.5x^2 + 2.0x + -12.0

Turning point = (-2.0 , -14.0)

X = 3.2915026221291814
X = -7.291502622129181
```

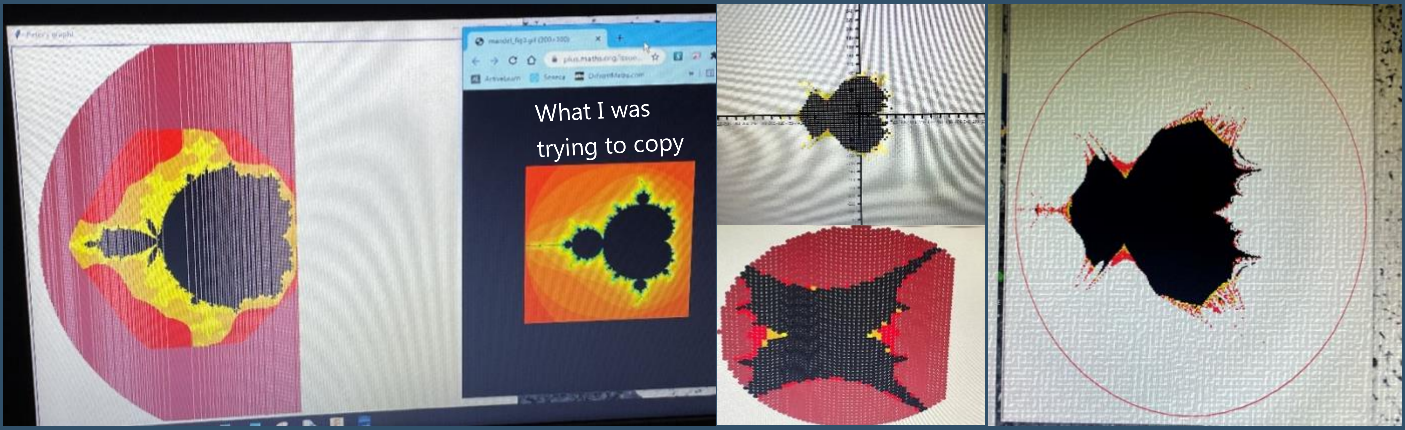
graph type
input
turning point
roots

graph



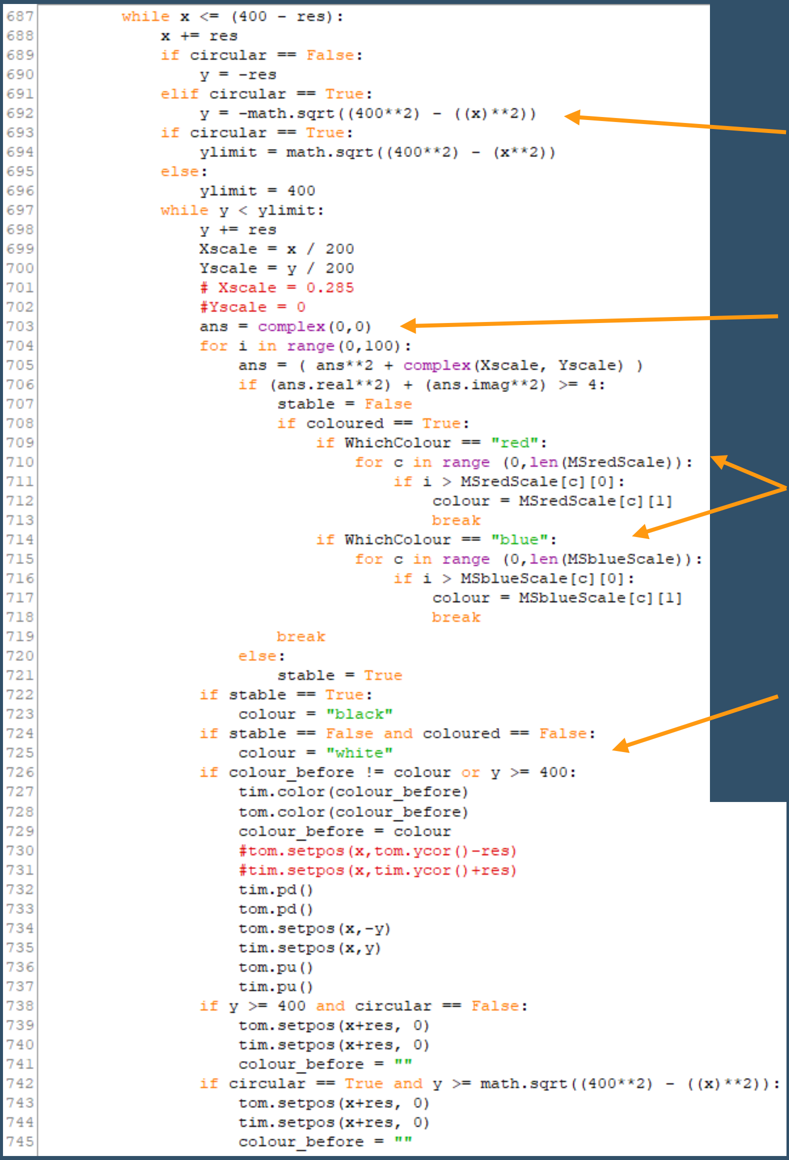
Milestone 7a – The Mandelbrot Set:

Following the algorithm I found online, I attempted to create the image of the Mandelbrot set.
These were my first attempts:



Whilst these were failures, I learnt a lot, especially about 'optimisation' as I started by drawing this one pixel at a time and they were taking hours to render (using the same 'turtle' library). I reduced this time by using:
- Run Length Encoding (RLE).
- The fact that the shape I was making was symmetrical, meant I could halve the number of calculations.
- A circle around the shape, not a full square, also reduced the number of calculations required.

After a lot of trial and error, I discovered that Python was able to deal with complex numbers.
So, when I changed my variables to imaginary ones I was able to get it working.



```
        while x <= (400 - res):
            x += res
            if circular == False:
                y = -res
            elif circular == True:
                y = -math.sqrt((400**2) - ((x)**2))
            if circular == True:
                ylimit = math.sqrt((400**2) - (x**2))
            else:
                ylimit = 400
            while y < ylimit:
                y += res
                Xscale = x / 200
                Yscale = y / 200
                # Xscale = 0.285
                #Yscale = 0
                ans = complex(0,0)
                for i in range(0,100):
                    ans = ( ans**2 + complex(Xscale, Yscale) )
                    if (ans.real**2) + (ans.imag**2) >= 4:
                        stable = False
                        if coloured == True:
                            if WhichColour == "red":
                                for c in range (0,len(MSredScale)):
                                    if i > MSredScale[c][0]:
                                        colour = MSredScale[c][1]
                                        break
                            if WhichColour == "blue":
                                for c in range (0,len(MSblueScale)):
                                    if i > MSblueScale[c][0]:
                                        colour = MSblueScale[c][1]
                                        break
                        break
                    else:
                        stable = True
                if stable == True:
                    colour = "black"
                if stable == False and coloured == False:
                    colour = "white"
                if colour_before != colour or y >= 400:
                    tim.color(colour_before)
                    tom.color(colour_before)
                    colour_before = colour
                    #tom.setpos(x,tom.ycor()-res)
                    #tim.setpos(x,tim.ycor()+res)
                    tim.pd()
                    tom.pd()
                    tom.setpos(x,-y)
                    tim.setpos(x,y)
                    tom.pu()
                    tim.pu()
                if y >= 400 and circular == False:
                    tom.setpos(x+res, 0)
                    tim.setpos(x+res, 0)
                    colour_before = ""
                if circular == True and y >= math.sqrt((400**2) - ((x)**2)):
                    tom.setpos(x+res, 0)
                    tim.setpos(x+res, 0)
                    colour_before = ""
```

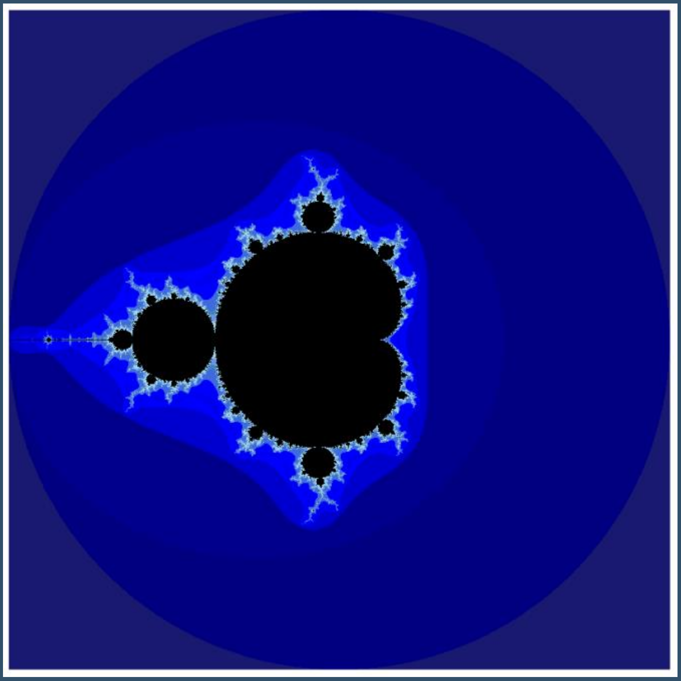
Circle equation for circular shape

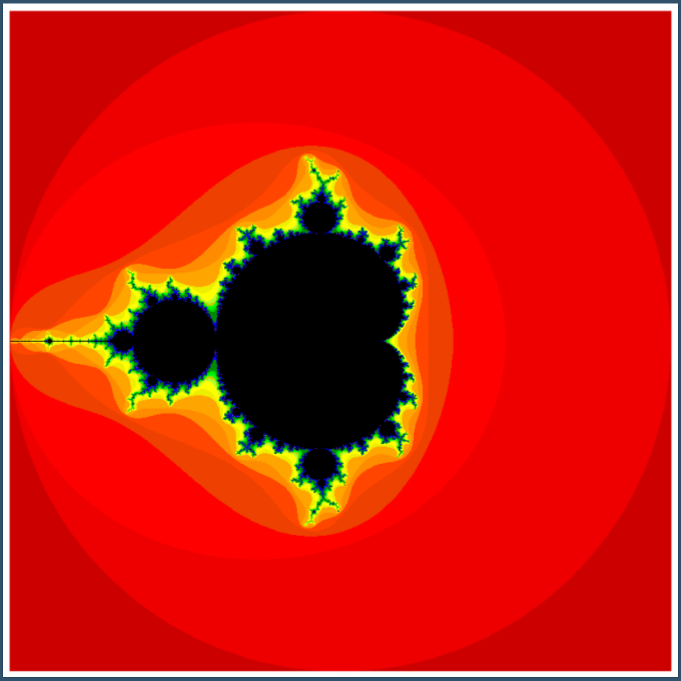
Use of complex variable types

2 colour schemes

Only moves 'turtle' if colour has changed (RLE)

The results:





As the images above show, I was finally able to produce clear, colourful, beautifully-detailed MS plots.

Finished prototype:

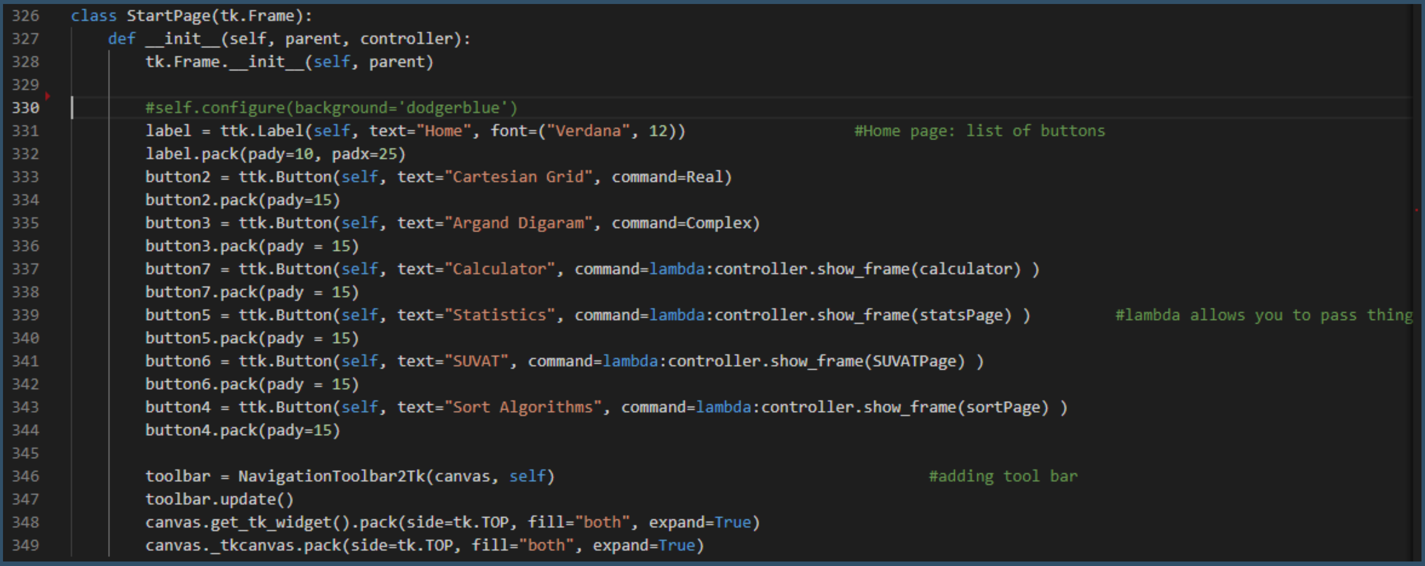
After weeks of work, I have now completed all the milestones that I set, in the prototype. I have manged to get all the math working using Python, with mostly just functions (not object orientated yet).

I have received some positive feedback from other students and my client. They like the functions and capability of my program so far but they think it is perhaps a bit too slow. This is something to consider and try to rectify in the next iteration.

I will now re-do the seven milestones with the following changes:
- Include validation and testing
- Consider the feedback and try and make my program more efficient and quicker
- Start using VS code for editing my code rather than the Python IDLE (which I have used up till now)
- Stop using 'turtles' and start using 'matplotlib' and 'Tkinter'
- Write an object orientated program as I designed and planned to implement.

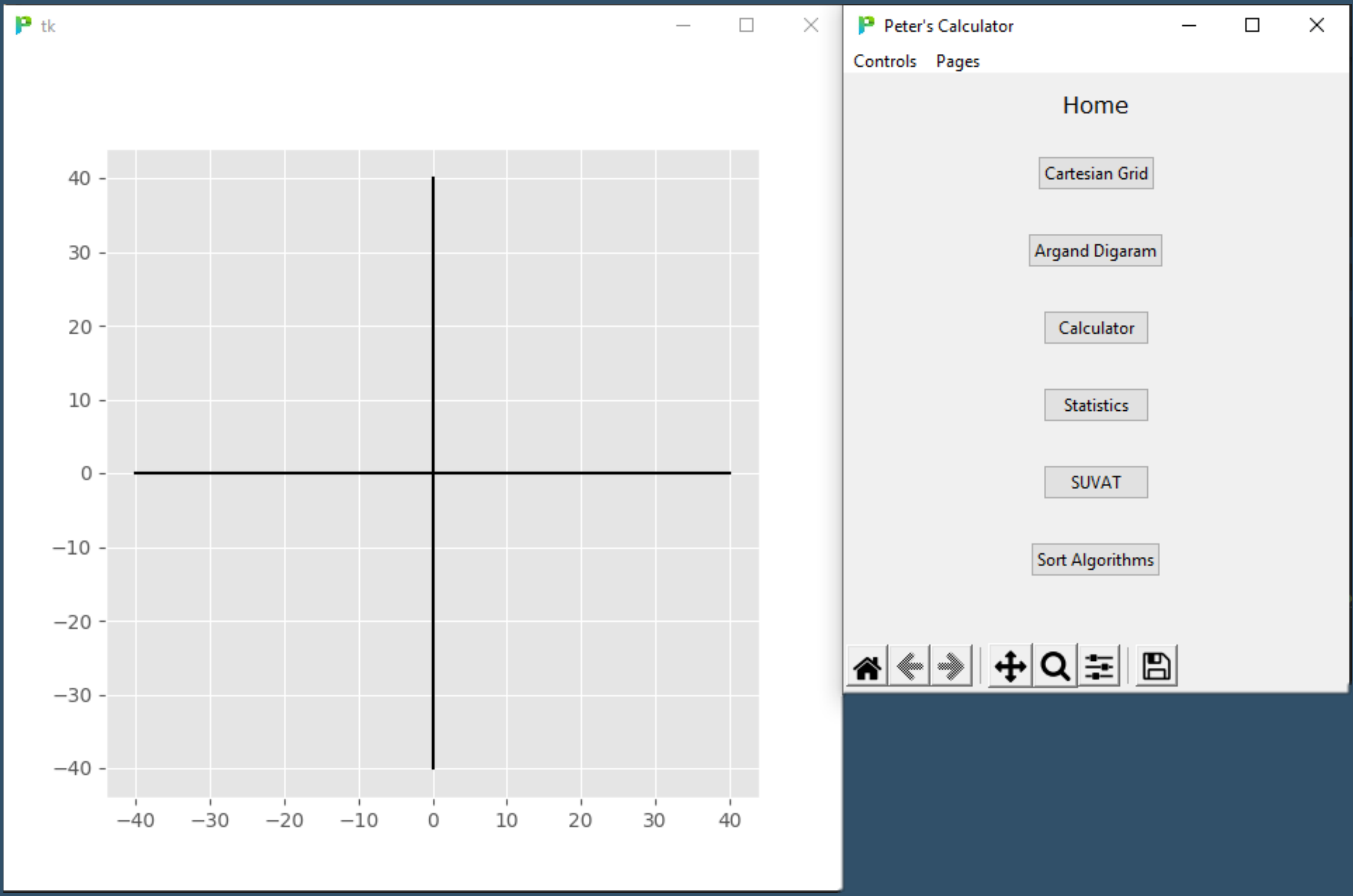
Milestone 1b – User Interface:

I will now design the real user interface as I planned. This should be easier to follow and understand.



```
class StartPage(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        #self.configure(background='dodgerblue')
        label = ttk.Label(self, text="Home", font=("Verdana", 12))               #Home page: list of buttons
        label.pack(pady=10, padx=25)
        button2 = ttk.Button(self, text="Cartesian Grid", command=Real)
        button2.pack(pady=15)
        button3 = ttk.Button(self, text="Argand Digaram", command=Complex)
        button3.pack(pady = 15)
        button7 = ttk.Button(self, text="Calculator", command=lambda:controller.show_frame(calculator) )
        button7.pack(pady = 15)
        button5 = ttk.Button(self, text="Statistics", command=lambda:controller.show_frame(statsPage) )        #lambda allows you to pass thing
        button5.pack(pady = 15)
        button6 = ttk.Button(self, text="SUVAT", command=lambda:controller.show_frame(SUVATPage) )
        button6.pack(pady = 15)
        button4 = ttk.Button(self, text="Sort Algorithms", command=lambda:controller.show_frame(sortPage) )
        button4.pack(pady=15)

        toolbar = NavigationToolbar2Tk(canvas, self)                            #adding tool bar
        toolbar.update()
        canvas.get_tk_widget().pack(side=tk.TOP, fill="both", expand=True)
        canvas._tkcanvas.pack(side=tk.TOP, fill="both", expand=True)
```

Here is the code I wrote to show the start page. As you can see, I have created a class called 'StartPage' so that I can reuse this code whenever the user selects this page. You can also see the list of labels and buttons that I have made that make up this page, as well as the tool bar at the bottom that will allow the user to manipulate the graph.

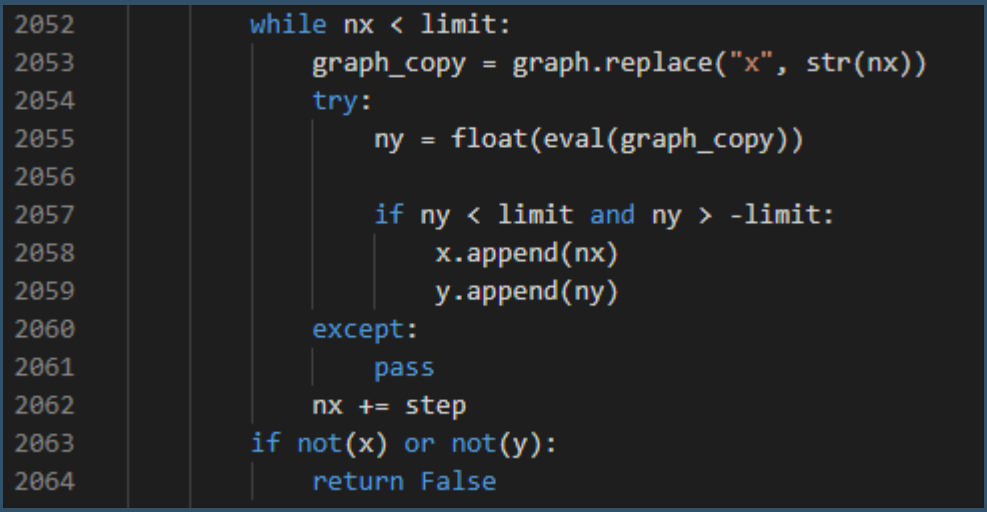


Here you can see the new layout – I have changed the design slightly to have split screens. This separates the axes and the input page. This feature allows the user to present their graphs independently of the controls, for example teachers could show the axes on the board while having the controls on their computer.

When I spoke to my Math teacher, she liked this feature and thought it was a good idea. She also stated the interface "looked friendly" and "more simple to use" than the traditional graphical calculator.
All the feedback was very positive and the testing I did went smoothly, I have not had to modify anything with the GU interface yet, which reflects all the thoughts I had successfully implemented.

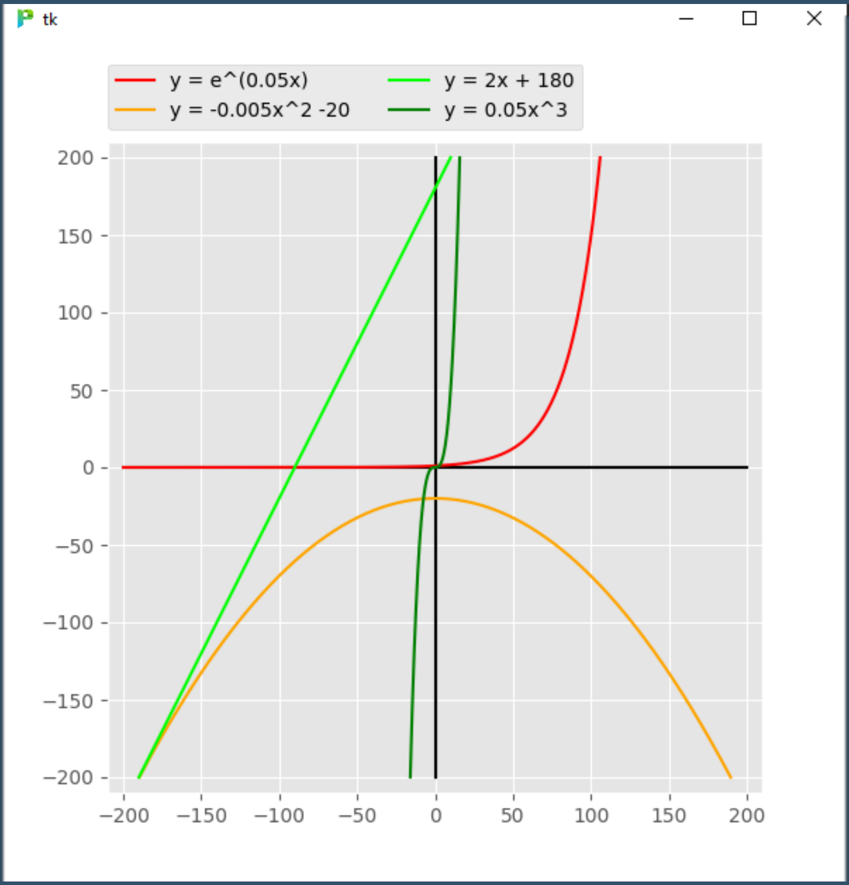
Milestone 2b – Drawing functions:

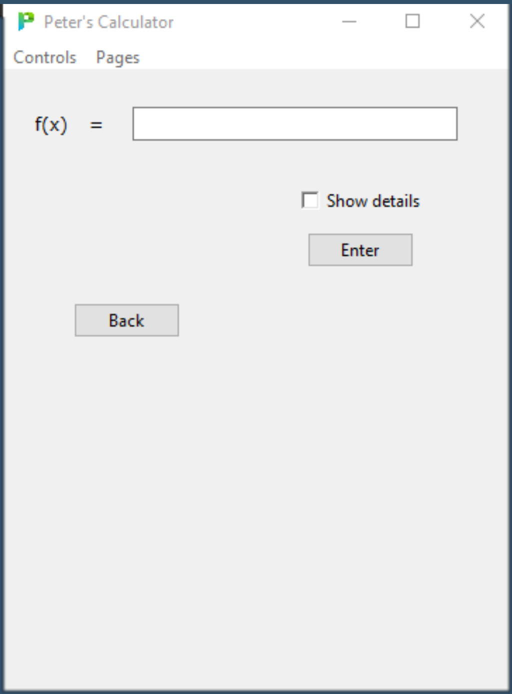
Using the math I got working in my prototype, I now need to plot graphs on this new interface. This works differently to 'Turtles'. I now need to generate a list of X and Y points to plot.



```python
        while nx < limit:
            graph_copy = graph.replace("x", str(nx))
            try:
                ny = float(eval(graph_copy))

                if ny < limit and ny > -limit:
                    x.append(nx)
                    y.append(ny)
            except:
                pass
            nx += step
        if not(x) or not(y):
            return False
```

Here I found an alternative solution.
'graph_copy' refers to the function that the user inputted e.g. '$3x^2$+2x'.
I replace 'x' with the variable 'nx' e.g. '$3nx^2$+2nx.
Then, on each iteration I 'evaluate' the equation (the inputted string) with an incremented number of 'nx' e.g. nx = 0 → $3*0^2+2*0$. Then nx = 1 → $3*1^2+2*1$.
On each iteration, I check the calculated value fits on the axis and then add it to the x / y list of points to plot. The reason I have changed this code from the original is because I found this to be more efficient and more importantly, it allows the user to input **any** type of function, not just linear, quadratic or cubic.





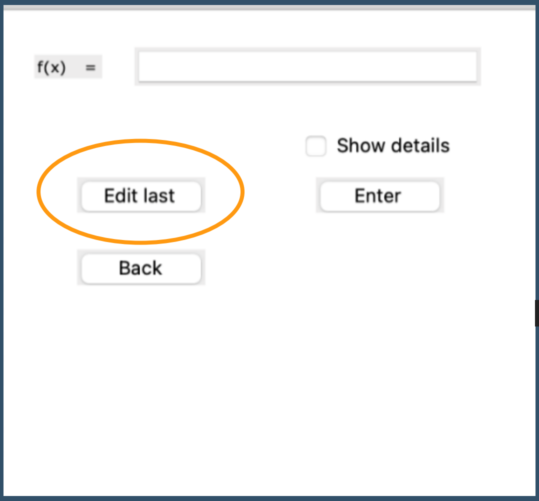
Here you can see how I have set up the user interface to allow the user to input functions. And you can also see the labels and colour coordinated graphs. This is much faster than 'turtles' and makes the inputting far more user-friendly.

I fully tested this feature against the test table I created earlier, verifying that all these sections work. This includes negatives, ranges, colour changing etc… As shown above.

After talking to my peers, they were impressed with the increased speed compared to my prototype. "It was able to plot lines much faster" they said, "this makes it much easier to use".

There was one improvement, suggested by my teacher. That was to add a 'Edit last plot' button that allows you edit the last thing you plotted; in case you made a mistake. To achieve this, I made a global list that stores all the points of everything plotted on the graph. Then, if the user would like to undo or edit something, I am able to remove the last item from this list and add the edited function.

So now we have this:

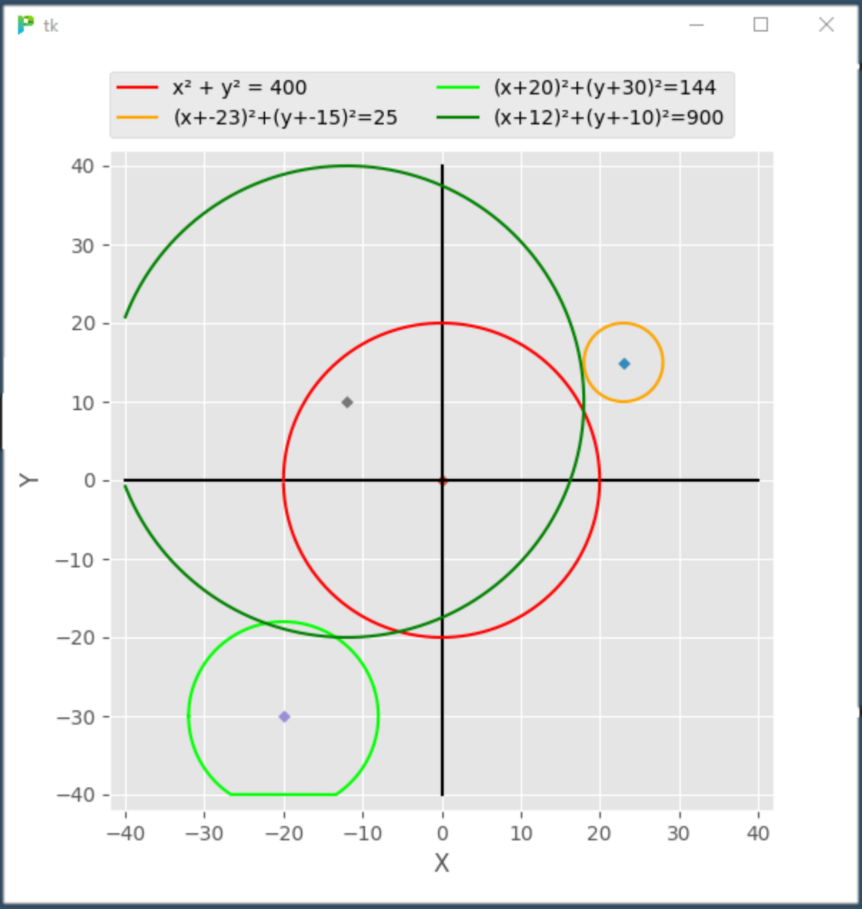


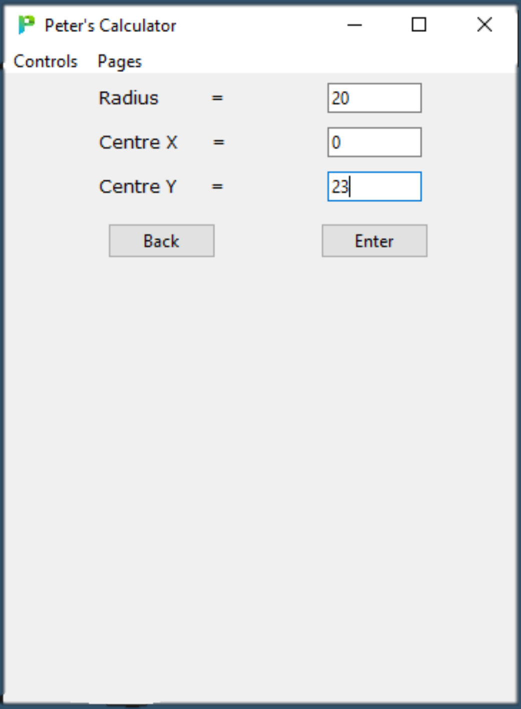
By adding this line of code to the end of the 'draw' function, I am adding the plot to my global 'storage' list, which I can refer to if the user wants to edit something.

```
graph_storage.append( s.plot(x,y, colours[colour_index], label=org_graph) )
```

Milestone 3b – Drawing Circles

Similarly, the math is all be the same as the prototype, it is just the new interface that I had to get working, as demonstrated here.

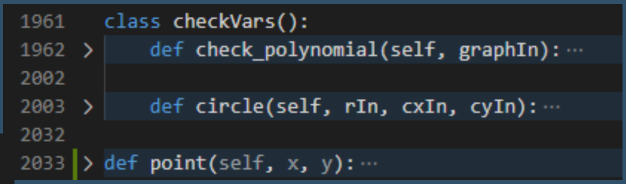




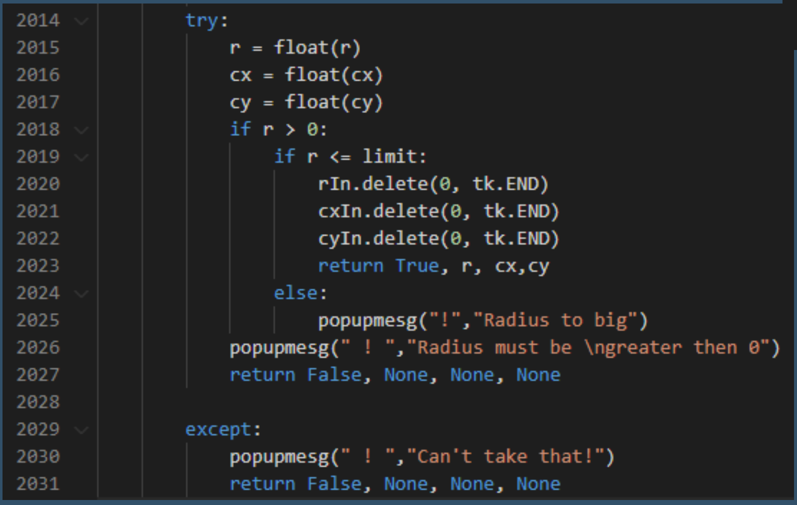
Radius = 20
Centre X = 0
Centre Y = 23

As you can see, I have issues that I have encountered when the circle crosses the boundary of the axis. This is down to the way I plot the circles by storing points. Lines are drawn between the points I calculate, to create one flowing shape. When I start drawing points from the left-hand side of the circle, if the circle goes off the bottom, the points are just joined up to create a flat edge but if the circle goes off the left edge, it forms a gap as this is the last point calculated (as shown above).

While testing, I realised I needed to add validation checking to the user inputs, so that my program would not crash if the user put a character in. For this I created another class for checking and I created functions for all the drawing tools that required inputs validating.



```
class checkVars():
    def check_polynomial(self, graphIn):…

    def circle(self, rIn, cxIn, cyIn):…

def point(self, x, y):…
```

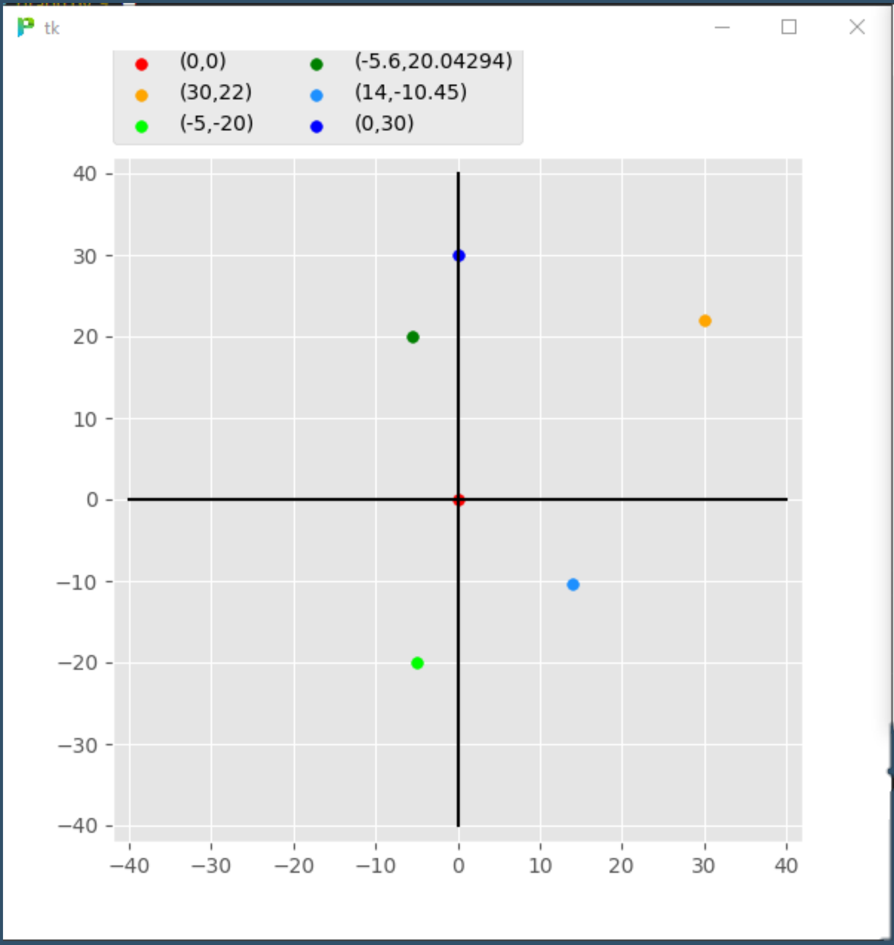


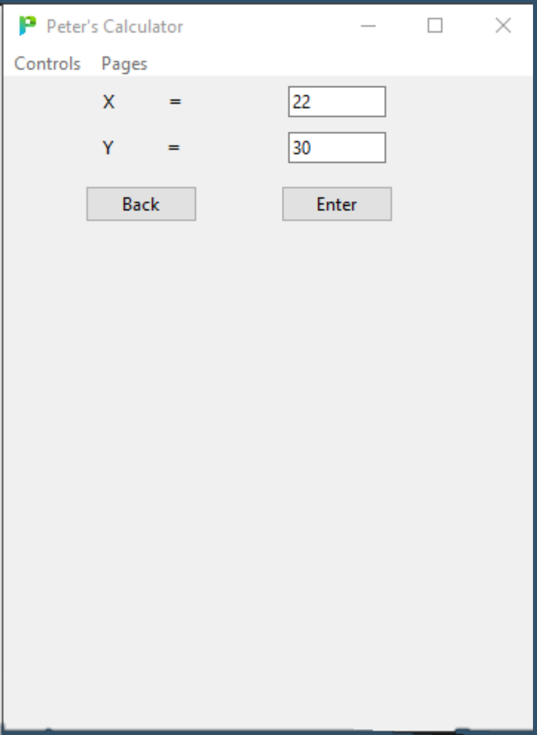
```
        try:
            r = float(r)
            cx = float(cx)
            cy = float(cy)
            if r > 0:
                if r <= limit:
                    rIn.delete(0, tk.END)
                    cxIn.delete(0, tk.END)
                    cyIn.delete(0, tk.END)
                    return True, r, cx,cy
                else:
                    popupmesg("!","Radius to big")
            popupmesg(" ! ","Radius must be \ngreater then 0")
            return False, None, None, None

        except:
            popupmesg(" ! ","Can't take that!")
            return False, None, None, None
```

The first thing I check for, is to see if the three inputs are floats or strings (radius, centre X and Centre Y).
I then check to see if the radius is greater than 0.
Finally, I check if the radius is less than the size of the graph.
If any of these fail, I display an error message.

Milestone 4b – Drawing Points:

The only real change I have made that is different to the prototype, for drawing points, is the fact that the labels for the corresponding points are no longer on the graph but are displayed above instead.
This means that the labels are no longer obstructing the user's view of other things on the graph.

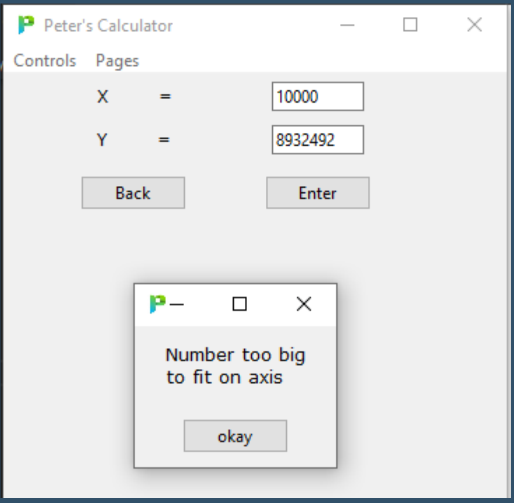




X = 22

Y = 30

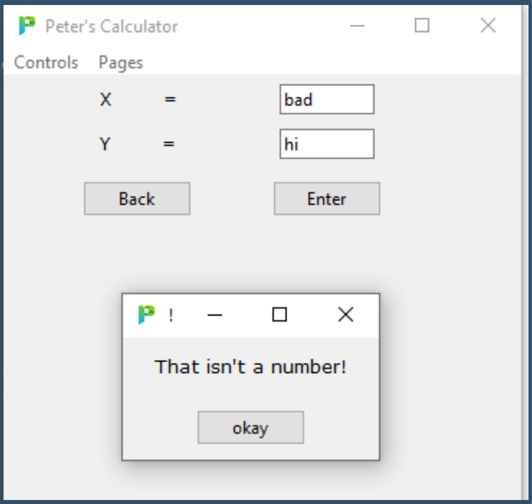
Fully tested now, this part of my program and has had no issues. Similarly, with all the other parts of my program, when the user types in something wrong, my program outputs an appropriate error message.



X = 10000

Y = 8932492

Number too big to fit on axis



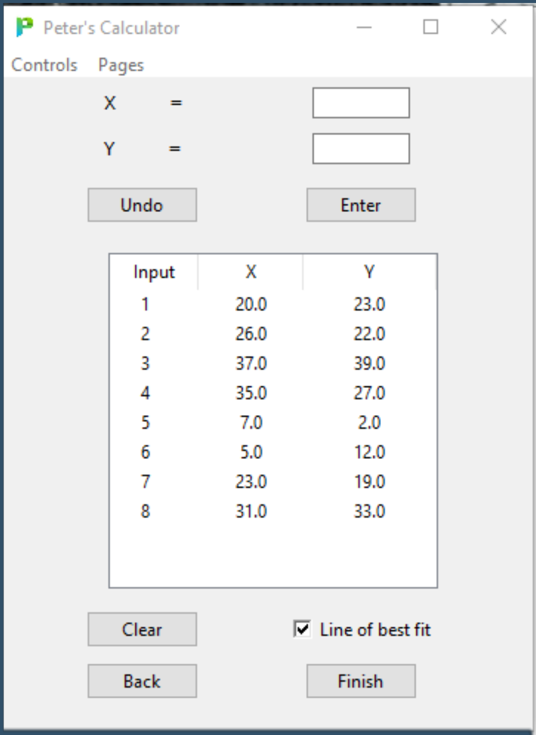
X = bad

Y = hi

That isn't a number!

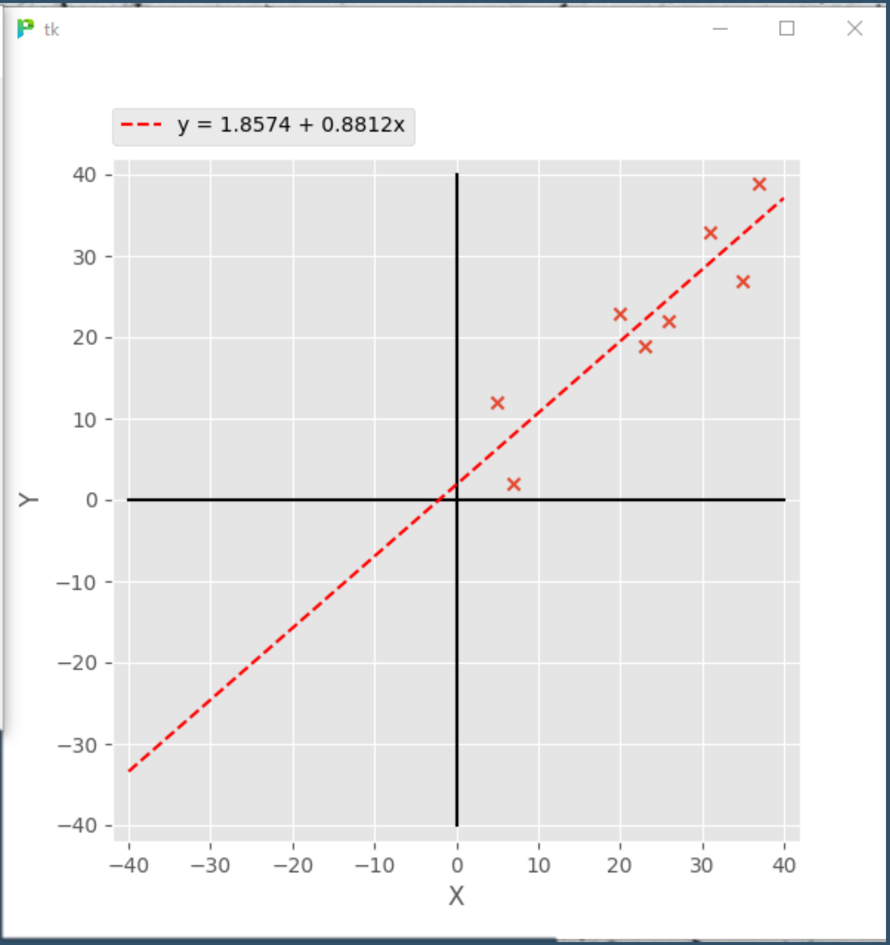
Milestone 5b – Regression line:

The most significant modification for the regression line, from my prototype version, is the user interface. I added a table with an 'undo' button to make everything look and feel both updated yet consistent. Plus, I added a check box for the regression line. If the user just wants a scatter graph, they can untick this box.

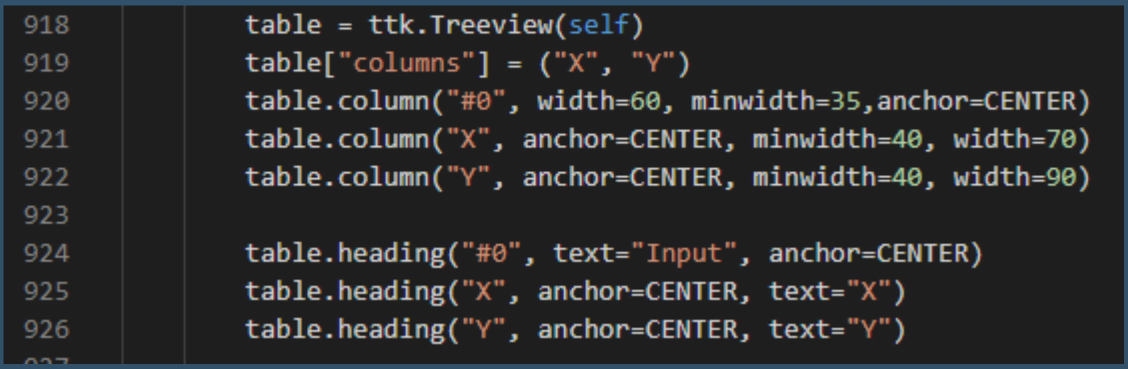


Input	X	Y
1	20.0	23.0
2	26.0	22.0
3	37.0	39.0
4	35.0	27.0
5	7.0	2.0
6	5.0	12.0
7	23.0	19.0
8	31.0	33.0

[✓ Line of best fit]

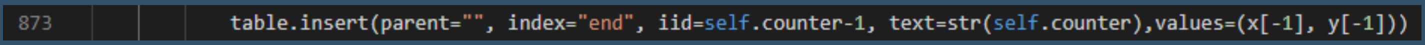


This is how I added the table within the UI.
Using the built-in 'treeview' function of 'Tkinter' I was able to create headings and columns.

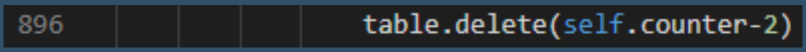


```
        table = ttk.Treeview(self)
        table["columns"] = ("X", "Y")
        table.column("#0", width=60, minwidth=35,anchor=CENTER)
        table.column("X", anchor=CENTER, minwidth=40, width=70)
        table.column("Y", anchor=CENTER, minwidth=40, width=90)

        table.heading("#0", text="Input", anchor=CENTER)
        table.heading("X", anchor=CENTER, text="X")
        table.heading("Y", anchor=CENTER, text="Y")
```

I was then able to get the buttons working by using these commands that add and remove items, retrospectively, from that table.



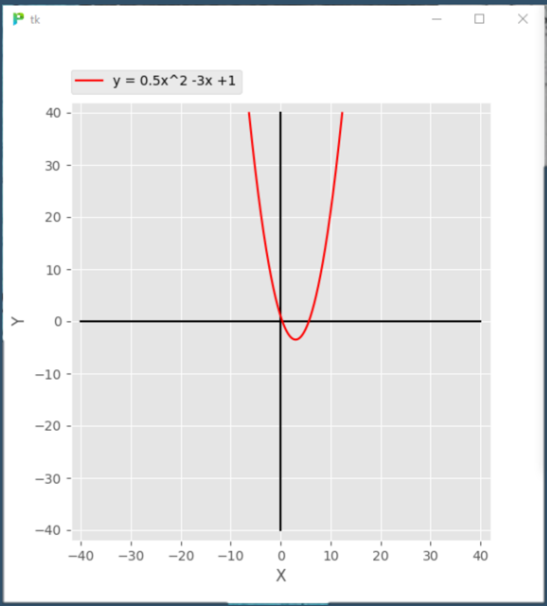
```
            table.insert(parent="", index="end", iid=self.counter-1, text=str(self.counter),values=(x[-1], y[-1]))
```

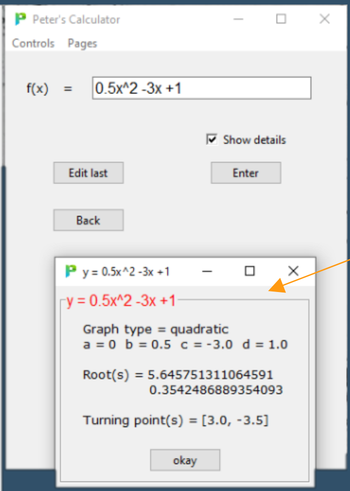


```
                table.delete(self.counter-2)
```

My stakeholder was very impressed and certainly preferred this over the previous version.
She found it a lot easier to use and more aesthetically pleasing.

Milestone 6b – Graph details:





y = 0.5x^2 -3x +1

```
Graph type = quadratic
a = 0  b = 0.5  c = -3.0  d = 1.0

Root(s) = 5.645751311064591
          0.3542486889354093

Turning point(s) = [3.0, -3.5]
```

Here, using the same math, I have improved the UI with this new, colour coordinated, pop-up. This shows all the necessary details as planned.

This was very challenging to complete. Before, the user inputted each variable separately (a=2 then b=6 then c=0 for example). Now, the user inputs the whole equation in one go, in any format (for example ax^2 + 3 – 3x). This makes the GUI very simple and easy to use but behind the scenes it is now my program that must understand and organise the variables before performing any mathematics.

To do this I first split this into its individual variables (no matter the spacing) and store this in a list.
I also put in place brackets and make sure that squares are in the right format (^ should be **).
As demonstrated below:

Input = 0.5x^2 -3x+1 → Output = `['0.5*(x)**2', '-3*(x)', '+1']` - Part 1

I then continue, working on each element of the list separately, I extract all the numerical characters left of the left bracket of 'x'. For example: <u>0.5</u>*(x)**2 → 0.5.
Now I would know that b = 0.5. - Part 2

There is not enough space here to show all the code (lines 2249 – 2344 or graph_details.details()), so here is a snippet from part 2:

This is just the 'IF' statement for the 'b' value or quadratic value, altogether that are four of these, for the four separate values I must find: a, b, c and d.

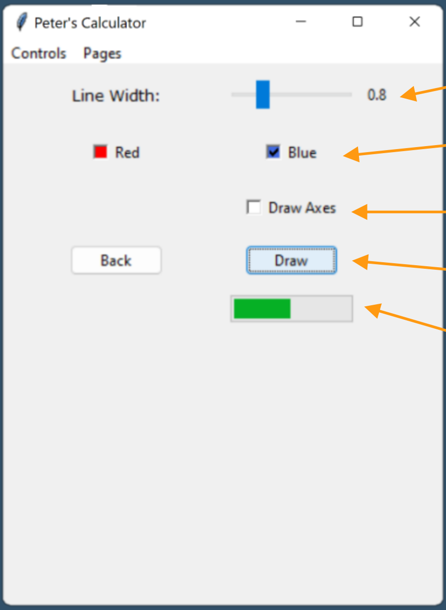
```
2297            elif abcd[i][temp1+2] == "2" and abcd[i][temp1-2]== "x":   #if not linear
2298                for j in range(0, len(abcd[i][:temp1-3])):
2299                    if abcd[i][j].isnumeric() or str(abcd[i][j]) == '.':
2300                        b += abcd[i][j]
2301                if not(b):
2302                    b = "1"
2303                if abcd[i][0] == "-":
2304                    b = b[:0] + "-" + b[0:]
```

'abcd' is an array that holds the output shown in part 1. 'temp1' is a temporary variable that holds the position of any '**' within the string: `temp1 = abcd[i].find("**")       #if linear or not?`

I then set the value of 'b' equal to any values before that left bracket position (line 2300), if they are numerical or a decimal point (line 2299). Finally, if 'b' has no visible value, I set it to equal 1 (line 2301) and if it was originally negative I add that too (line 2304). Now that I have this value (and 3 others) I can calculate the roots, intercepts etc … to display.

Milestone 7b – Mandelbrot Set:

The only modification here, is that I have given the user the option to change the resolution or line width of the Mandelbrot set that they want to draw. Smaller resolutions can take a while to render (~10mins).



Slider: step = 0.1, min = 0.1, max =3.

Tick boxes: 2 options (only 1 can be selected at a time)

Draw axes tick box

Draw / Back buttons

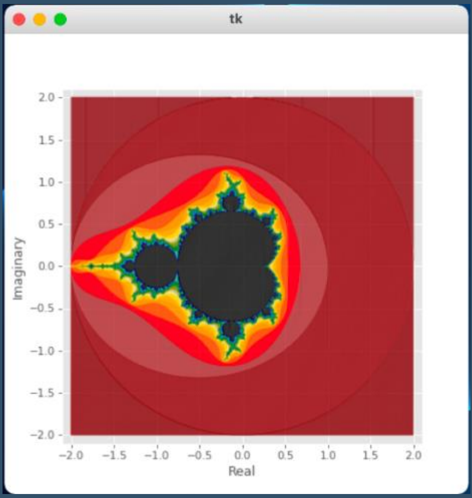
Progress bar

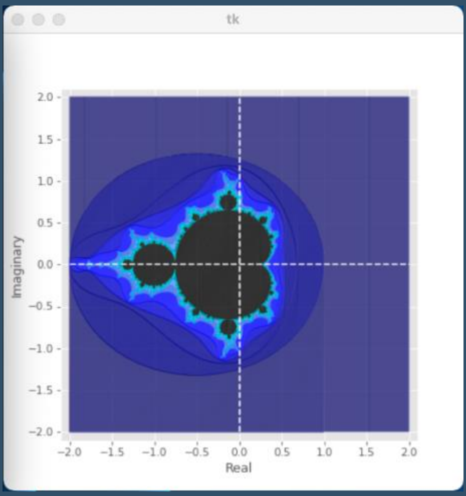
To add the progress bar, I added this line of code to the Mandelbrot set function:

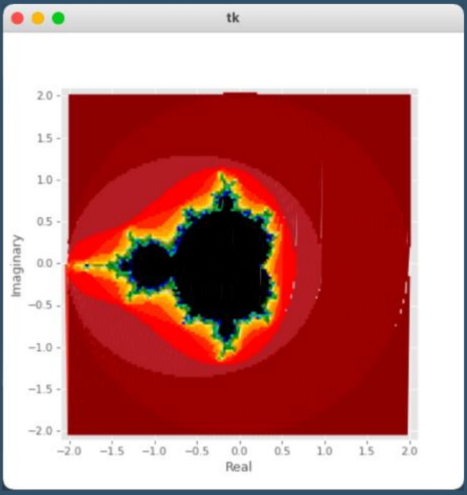
```
progress['value'] = (((X+limit)/(2*limit))*100)
```

This is just a ratio of how far the algorithm has been completed versus how much is left to process.

Testing:







Line width 0.2

Line width 0.4, with axes

Line width 3

As shown, I have also switched operating systems, from Windows to Mac, as I got a new laptop.

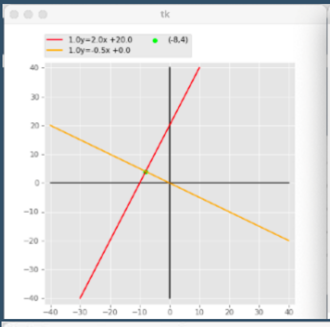
User feedback was great on this; so much so, Mrs Evans printed it off to display in her classroom.

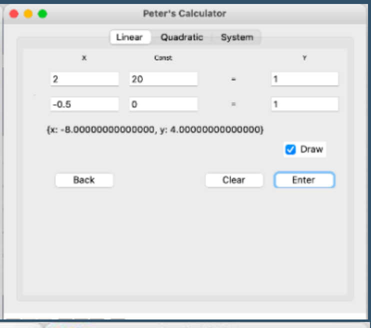
There are additions I can think to add to this page but I have chosen to prioritise other mathematical functions as drawing the Mandelbrot set is not a graphical calculator's most important function. Improvements could include zooming, adding more colours, personalised colours and zooming in on a specific point.

Extra Features:

With extra effort, I decided to add these features, some of which I mentioned at the start:

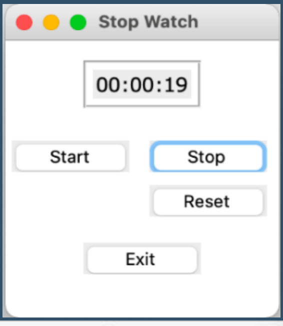
- Simultaneous equation plotter
- Simultaneous equation solver linear and quadratic
- Wave drawer
- Bar graphs
- Complex numbers, points and vectors
- Complex circles
- Complex half line
- Complex perpendicular bisector
- Calculator
- Equations page
- Statistics page
- Cumulative frequency graphs
- Box plots
- Maths/Physics constants page
- SUVAT equations solver
- Sorting algorithms
- Stop-watch
- Sketch page



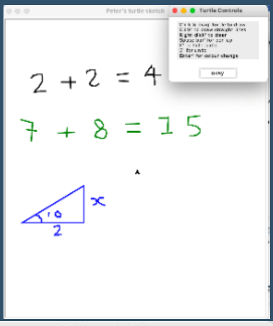


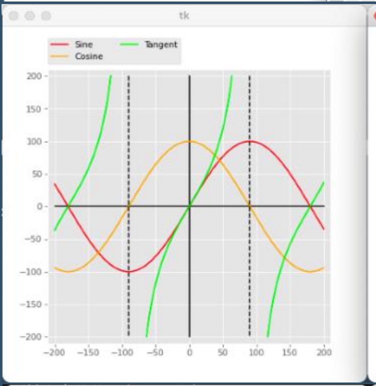
X	Const		Y
2	20	=	1
-0.5	0	=	1

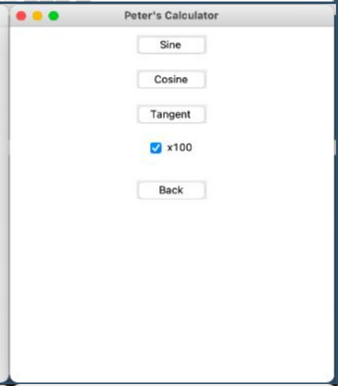
{x: -8.00000000000000, y: 4.00000000000000}

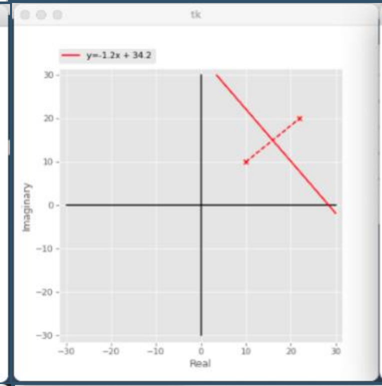


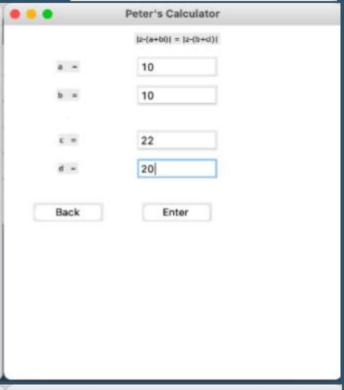
00:00:19





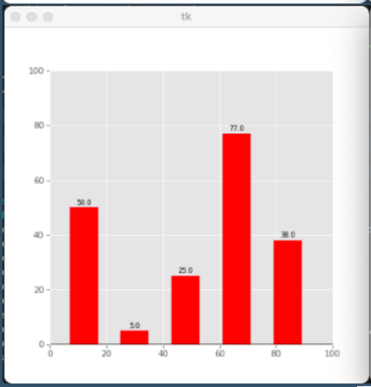


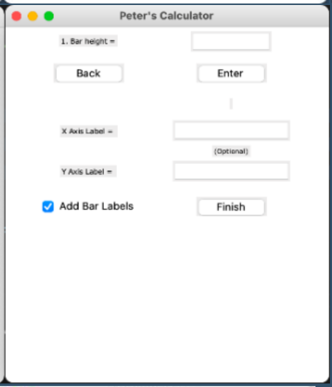


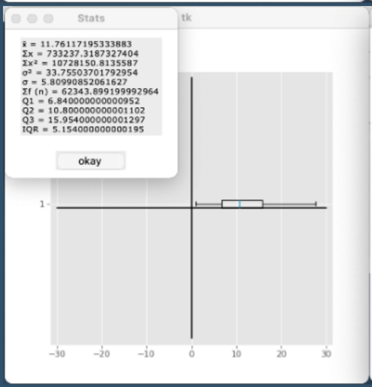


|z-(a+bi)| = |z-(c+di)|

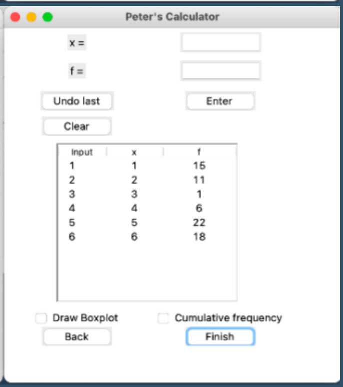
a = 10
b = 10
c = 22
d = 20



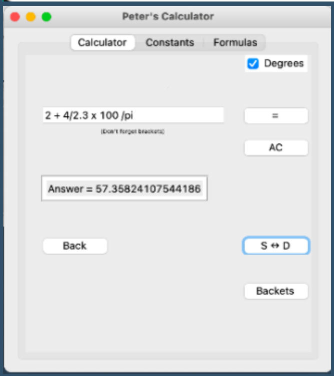




```
x̄ = 11.76117195333883
Σx = 733237.3187327404
Σx² = 10728150.8135587
σ² = 33.75503701792954
σ = 5.80990852061627
Σf (n) = 62343.899199992964
Q1 = 6.840000000000952
Q2 = 10.800000000001102
Q3 = 15.954000000001297
IQR = 5.154000000000195
```



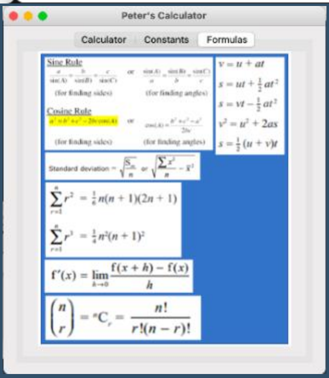
Input	x	f
1	1	15
2	2	11
3	3	1
4	4	6
5	5	22
6	6	18



2 + 4/2.3 x 100 /pi

(Don't forget brackets)

Answer = 57.35824107544186



Sine Rule

$\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)}$ or $\frac{\sin(A)}{a} = \frac{\sin(B)}{b} = \frac{\sin(C)}{c}$

(for finding sides) (for finding angles)

Cosine Rule

$a^2 = b^2 + c^2 - 2bc\cos(A)$ or $\cos(A) = \frac{b^2 + c^2 - a^2}{2bc}$

(for finding sides) (for finding angles)

$v = u + at$

$s = ut + \frac{1}{2}at^2$

$s = vt - \frac{1}{2}at^2$

$v^2 = u^2 + 2as$

$s = \frac{1}{2}(u + v)t$

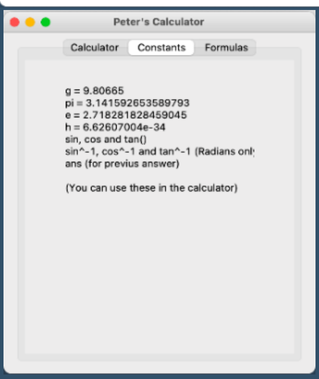
Standard deviation $= \sqrt{\frac{S_{xx}}{n}}$ or $\sqrt{\frac{\sum x^2}{n} - \bar{x}^2}$

$$\sum_{r=1}^{n} r^2 = \frac{1}{6}n(n+1)(2n+1)$$

$$\sum_{r=1}^{n} r^3 = \frac{1}{4}n^2(n+1)^2$$

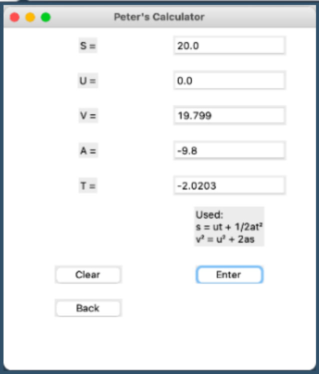
$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\binom{n}{r} = {}^nC_r = \frac{n!}{r!(n-r)!}$$



g = 9.80665
pi = 3.141592653589793
e = 2.718281828459045
h = 6.62607004e-34
sin, cos and tan()
sin^-1, cos^-1 and tan^-1 (Radians onl
ans (for previus answer)

(You can use these in the calculator)



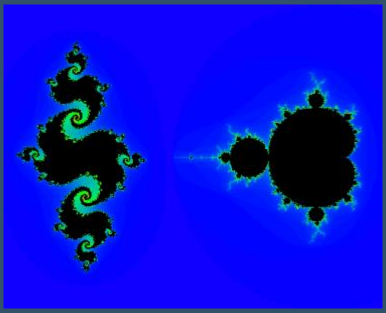
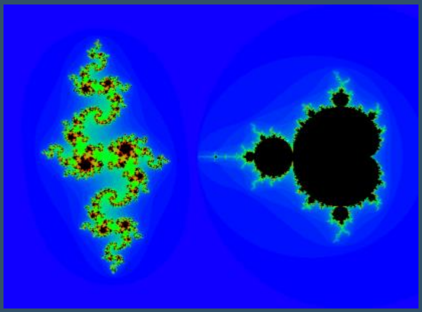
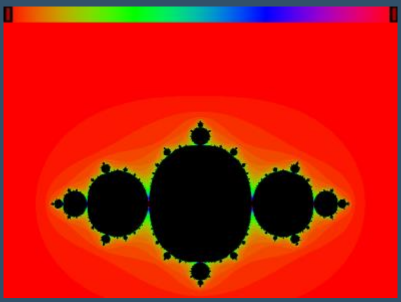
S = 20.0
U = 0.0
V = 19.799
A = -9.8
T = -2.0203

Used:
s = ut + 1/2at²
v² = u² + 2as

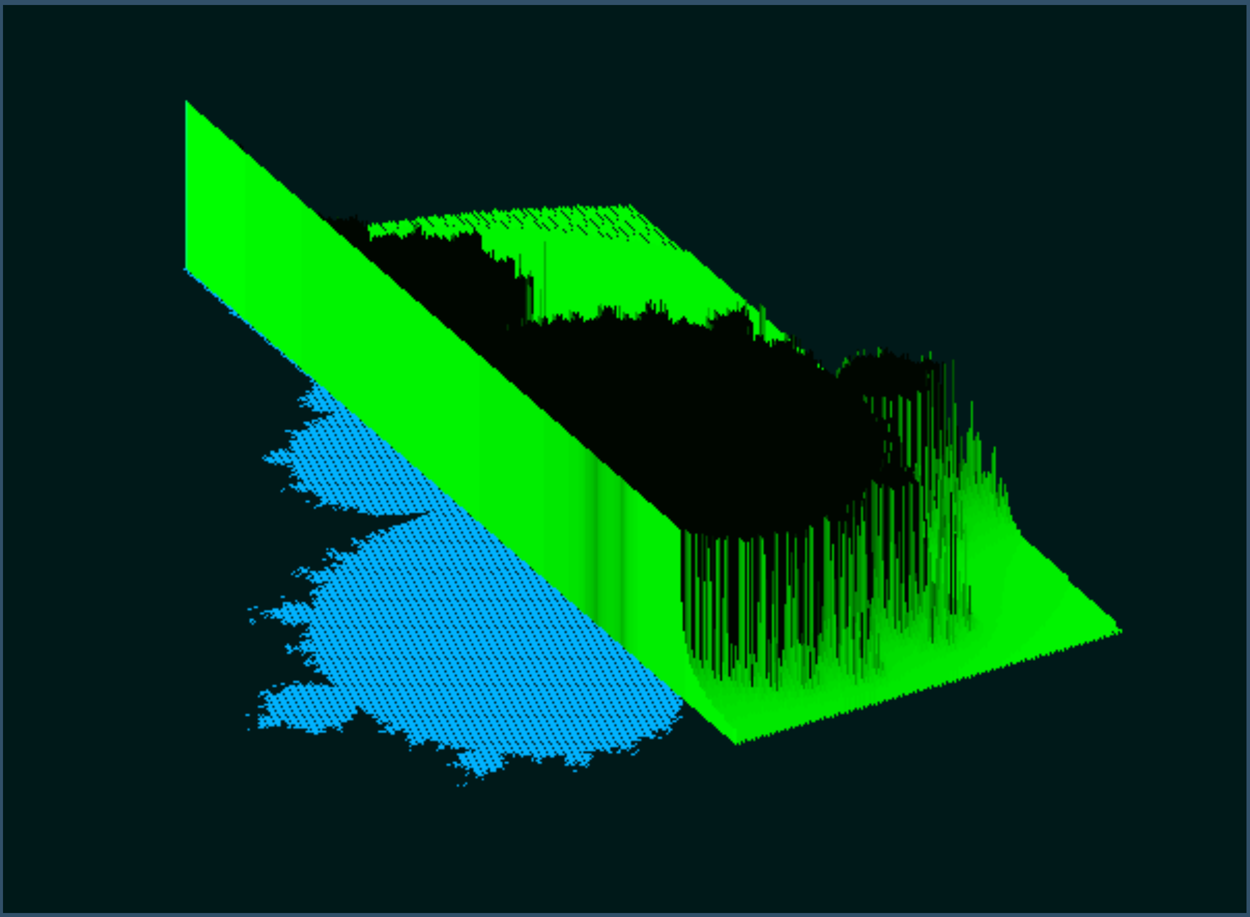
I learned a lot developing these features. For example, I learned how to introduce threading into my program (which enabled me to get the stopwatch working accurately), how to import images into 'Tkinter' interfaces, how to convert unknown decimals to fractions and much, much more…

I also experimented with improving the Mandelbrot set. I tried using C++ and 'shaders' to improve the speed required to process the images. This did work but I wasn't able to get this working within the existing application and they were incompatible. I did however, get it working in a separate test application and these are some of the results:

https://youtu.be/8L85te7QoQs

I also experimented with three dimensions but had the same problems with compatibility.



Evaluation

Alpha Testing

Test video link: https://drive.google.com/file/d/1sXi_xzA1Yn_lLsNWnUh5s4xHzxBSLU68/view?usp=sharing

	Milestone 1: User interface			
Test No.	**What is being tested and inputs**	**Expected output(s)**	**Time stamp**	**Pass / Fail**
1	That buttons do as they say. Test – Press them.	The button performed as expected e.g. links to the correct page.	0:55	Pass
2	Text inputs work and reject unusable inputs. Test – Enter text, try numbers, letters, strings, 0, negative numbers.	When correct data type entered, the assosiated reaction should occur. If unwanted data inputted, an error message is displayed.	1:55	Pass
3	Check-boxes work. Test – tick them, and untick them.	The check-boxes should have the intended result.	2:42	Pass
4	Scroll bars work as intended. Test – Scroll them.	The scroll bars should have the intended result.	N/A	N/A

	Milestone 2: Drawing polynomials			
Test No.	**What is being tested and inputs**	**Expected output(s)**	**Time stamp**	**Pass / Fail**
1	Test linear functions. e.g. 2x - 3	The correct line should be plotted. e.g. gradient of 2 and y-interept of -3.	3:20	Pass
2	Test quaratic functions. e.g. 2x^2 +3x -5	The correct line should be plotted on the correct posistion on the axis.	3:55	Pass
3	Test cubic functions e.g. x^3 +2x^2 +3x +5	The correct line should be plotted on the correct posistion on the axis.	4:48	Pass
4	Test the 3 previus functions but with negative coefficents.	The correct line should be plotted on the correct posistion on the axis.	5:12	Pass
5	Test the colour of the graph changes when a new function is plotted.	The colours should cycle as different functions are plotted, so the user can distinguish between them.	6:10	Pass
6	Test limits of axis. e.g. make the y-intercept greater than or less than the axis.	If the graph doesn't fit on the axis, either display an error message or resize the axis and plot.	6:22	Pass

Milestone 3: Drawing circles

Test No.	What is being tested and inputs	Expected output(s)	Time Stamp	Pass / Fail
1	Draw circle at origin. e.g. radius = 10, centreX = 0, centreY = 0.	Draw a circle at the origin with a radius of 10.	7:20	Pass
2	Draw circles with a different centre. e.g. radius = 10, centreX = 2, centreY = -5.	Draw a circle at this position with that radius.	7:31	Pass
3	Test inputing negative radii. e.g. Radius = -23	Ouput a suitable error message as this is not possible to draw.	7:46	Pass
4	Test axis limits. E.g. Input a centre off the axis.	If the graph does not fit on the axis, either display an error message or resize the axis and then plot.	8:00	Fail
5	Test the colour of the graph changes when a new circle is plotted.	The colours should cycle as different things are plotted, so the user can distinguish between them.	8:35	Pass

Milestone 4: Drawing points

Test No.	What is being tested and inputs	Expected output(s)	Time Stamp	Pass / Fail
1	Draw a point. e.g x= 10, y =2	Draw a point at (10,2).	8:45	Pass
2	Test negatives. e.g. x=-10, y-2	Draw a point at (-10,-2).	9:12	Pass
3	Test mixture of both negatives and positives. e.g. x=10, y=-2	Draw a point at (10, -2).	9:00	Pass
4	Test point off the axes scale. e.g. x=200, y=1000	Display suitable error message.	9:20	Pass
5	Test inputting a string. e.g. “Hello”	Display a suitable error message.	9:29	Pass

Milestone 5: Regression line

Test No.	What is being tested and inputs	Expected output(s)	Time Stamp	Pass / Fail
1	Test what values can be inputted. e.g. x=10, y=5	Program should accept this input and ask the user for the next point.	9:55	Pass
2	Test inputtig a string instead of an integer. e.g. "test"	Display a suitable error message.	10:04	Pass
3	Test the correct regression line is outputted after at least 2 points have been inputted.	The correct regression line should be outputted.	10:45	Pass
4	Test point off the axes scale. e.g. x=200, y=1000	Display a suitable error message.	10:30	Pass

Milestone 6: Outputting data about graph

Test No.	What is being tested and inputs	Expected output(s)	Time Stamp	Pass / Fail
1	Test if the outputted roots are correct for linear graphs. E.g. Input = 2x	E.g. Output = 0	11:10	Pass
2	Test if the outputted roots are correct for quadratic graphs. E.g. Input = $(x+4)^2$	E.g. Output = -4	11:25	Pass
3	Test if the outputted roots are correct for cubic graphs. E.g. Input = $5x^3 +5x^2 +5x + 5$	E.g. Output = -1	11:42	Pass
4	Test if the outputted roots are approximately correct for other graphs. E.g. Input = $x^7 - 2x^3 + 2x – 2$	E.g. Output = ~1.158	12:30	Pass
5	Test If the outputted turning points are corect. E.g. Input = $5x^3 +5x^2 +5x + 5$	E.g. Output = (0,5)	13:37	Fail
6	Test if the outputted graph type is correct. E.g. Input = $x^3 + 3x^2- 2$	E.g. Output = 'Cubic'	14:43	Pass
7	Test that the colour of the graph changes.	Every gragh should have a different colour to the one plotted before it.	15:15	Pass

	Milestone 7: Mandelbrot Set			
Test No.	**What is being tested and inputs**	**Expected output(s)**	**Time Stamp**	**Pass / Fail**
1	Test the plotted Mandelbrot Set is acurate, to scale.	Should be plotted on imaginary axis between 2 and -2.	15:36	Pass
2	Test that the inputted colour scheme is correclty used.	If the user selects the red scheme. The MS should be red.	15:55	Pass

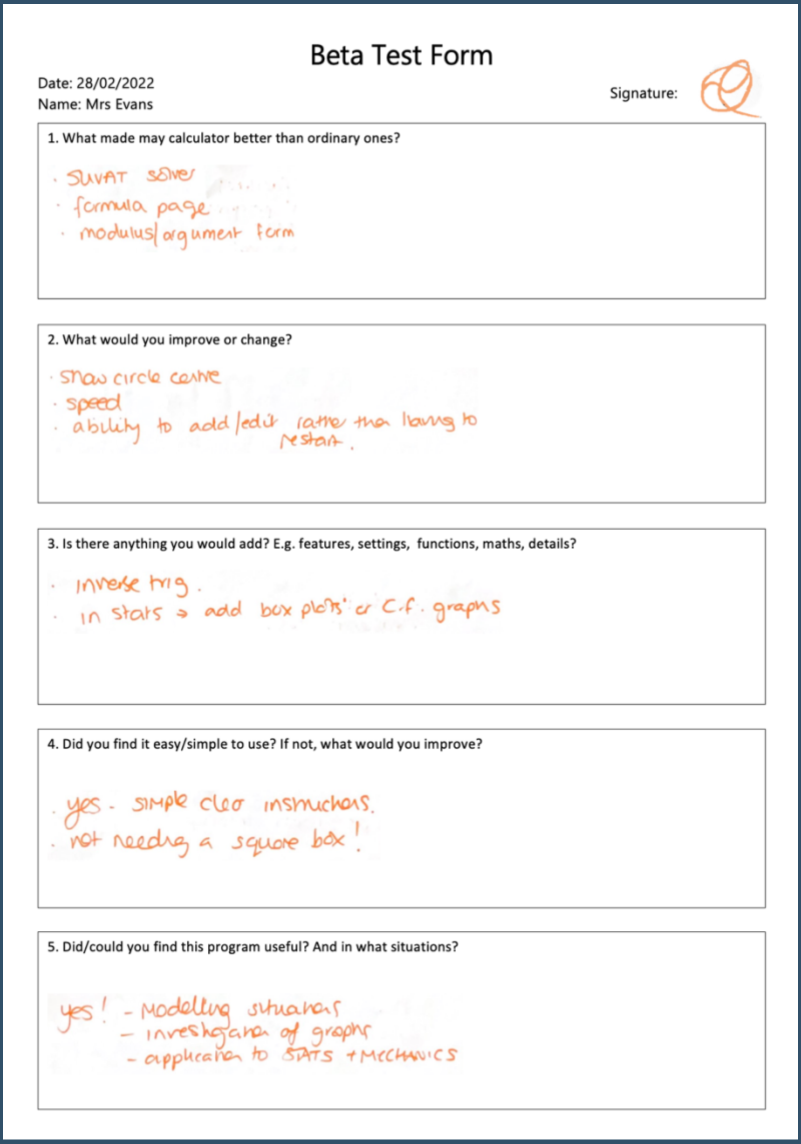
Out of all these tests, I only encountered 2 small errors.

As a result, I have had the chance to go back and fix them.

Beta Testing

I presented the final application to my stakeholder to get her feedback: My math teacher was impressed. She said, "This would be a great tool to use in lessons" and "Even my calculator doesn't solve SUVAT equations for me!".

As a result, I would say that this application is at a stage where it would be ready to ship and make public. After following an agile development path of constantly developing, validating and adjusting I feel the process was successful.



Beta Test Form

Date: 28/02/2022
Name: Mrs Evans

Signature: [illegible]

1. What made may calculator better than ordinary ones?

· SUVAT solver
· formula page
· modulus/argument form

2. What would you improve or change?

· show circle centre
· speed
· ability to add/edit rather than having to restart.

3. Is there anything you would add? E.g. features, settings, functions, maths, details?

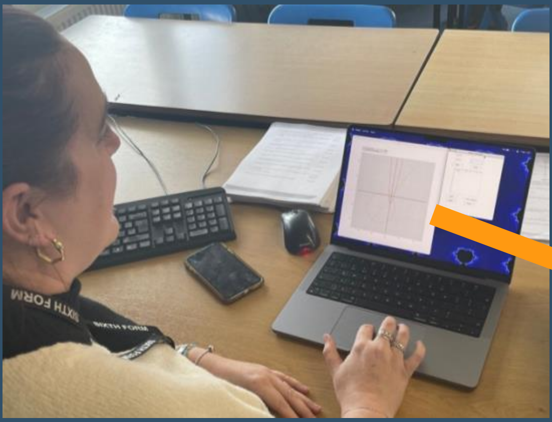
· inverse trig.
· in stats → add box plots' or C.f. graphs

4. Did you find it easy/simple to use? If not, what would you improve?

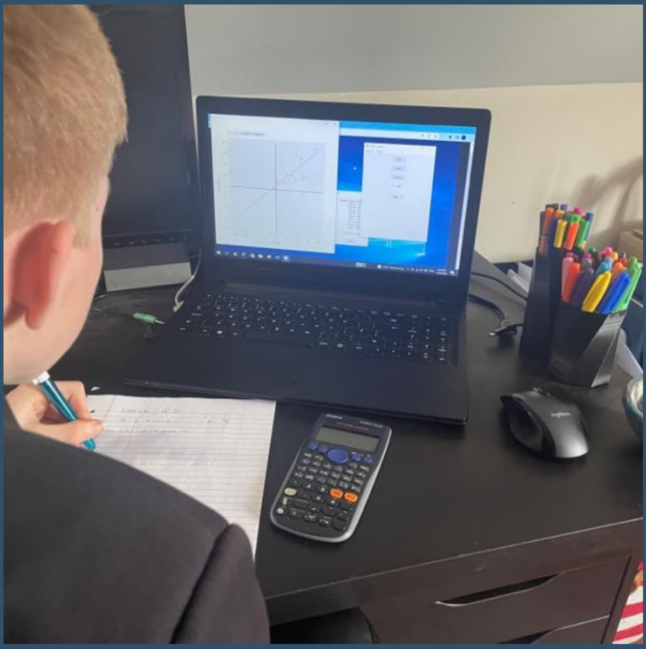
· yes - simple clear instructions.
· not needing a square box!

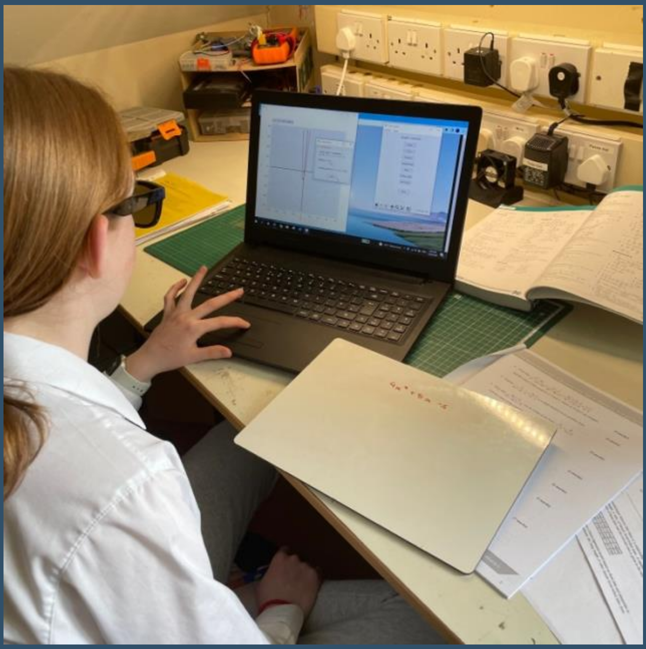
5. Did/could you find this program useful? And in what situations?

yes! - Modelling situations
- investigation of graphs
- application to STATS + MECHANICS

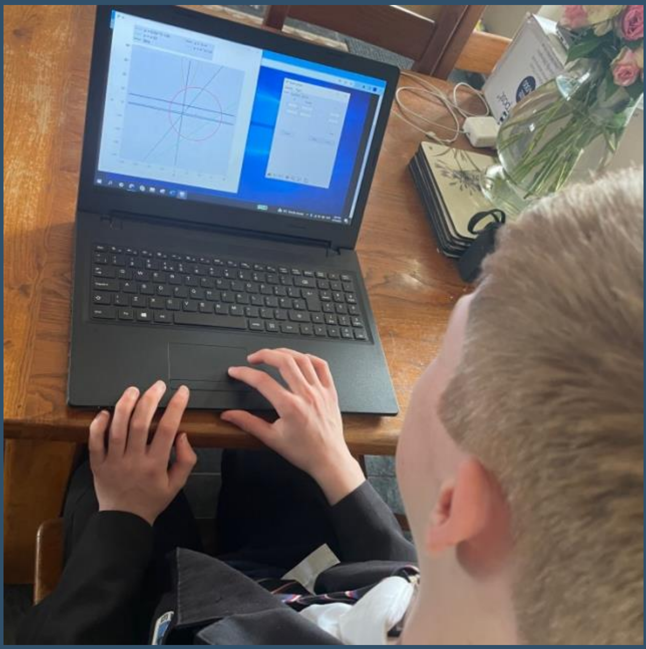


As well as my teacher helping to test my application, I enlisted the help of other students.





They suggested small modification for things like the button sizes and the colours I had used.



Whilst the application was working well at this point, these little suggestions helped the application feel and look better, increasing the aesthetics of my design.

Having other students test my application was also great as I got feedback from a different perspective. These students found my application best for helping with homework, rather than for use in lessons. They loved the range of functions it had, but used the ‘functions’ button most, to solve quadratics.

Using other students as my test bed, allowed me to iron out lots of irritations and small errors before testing commenced with my stakeholder – my math teacher.

Her time was more precious and so I could ensure that my app was working well before she sat down to test it out.

Product Review

The product is now finished, all the code is written and is working as specified. All the original features are fully tested and many extra features have been added along the way.

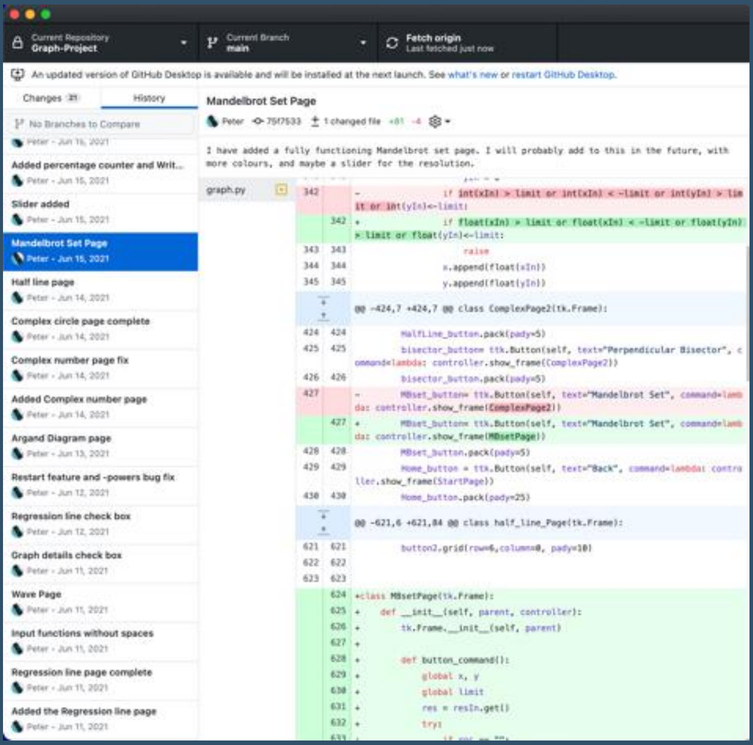
Listed below are the success criteria as stated at the beginning and whether these criteria were met.

No.	Criteria	Achieved?	Evaluation
1	Have 5 functions allowing the user to freely select the graphing function they would like to perform. This should include as a minimum: polynomials, circles, regression lines and complex numbers.	Fully met	This was achieved and exceeded. Users can draw functions, circles, regression lines, scatter graphs, bar charts, cumulative frequency graphs and all the above on the complex plane.
2	The user can input graph functions to be processed.	Fully met	This was completed. Users are easily able to type in whatever graph they would like to see.
3	The user can clearly see the graph they inputted on the screen and be able to make necessary observations form it. This can include the ability to manipulate the graph by zooming and moving it around the screen.	Fully met	See test video. User has a clear view of everything that they choose to draw. There was also evidence of being able to move the graph around and zoom in to specific areas.
4	The user should be able to clear the set of axes.	Fully met	This was also demonstrated in the test video many times before moving on to testing the next milestone.
5	Allow the user to plot up to cubic functions.	Fully met	This was certainly exceeded. The user can plot any type of function they wish, not just polynomials (infinite).
6	The program should output data about the inputted graph, including roots, turning points and intersections.	Fully met	This was met. Although in testing I came across some small issues, these have since been fixed.
7	The program must recognise and alert the user when the user has inputted something that is invalid.	Fully met	Again, this has also been met. I came across some small things to clear up in the test video but everything is working now.
8	The user must be able to distinguish between different graphs drawn on the same axis, by colour coordination and/or labelling.	Fully met	This was fully achieved. There are clear colours differentiating the different lines and correspond labels.

9	There should be a calculator within the application to allow the user to perform quick calculations.	Fully met	This exists are works perfectly.
10	Functions should be plotted with a high enough resolution and that approximated answers are accurate to at least two decimal places.	Fully met	This is done. Lines are smooth and the user can zoom in infinitely to make accurate observations about the graph.
11	The user should have the option to draw functions on the imaginary plane or Argand diagram.	Not Met	While the user can draw circles, points and bisectors on the imaginary plane, they cannot draw functions. My curriculum did not include drawing these.
12	The user should have the ability to quit the program whenever they wish to.	Fully met	This was achieved, using the red cross in the corner & button in drop down menu.

Throughout the project, I made sure to upload all my code to a personal GitHub repository so that I had a backup of my code.

This helped when I made a mistake, as I had a previous version to fall back on. It was also a tidy way of keeping track and keeping a log of everything I did.



An updated version of GitHub Desktop is available and will be installed at the next launch. See what's new or restart GitHub Desktop.

Peter - Jun 15, 2021

Added percentage counter and Writ...
Peter - Jun 15, 2021

Slider added
Peter - Jun 15, 2021

Mandelbrot Set Page
Peter - Jun 15, 2021

Half line page
Peter - Jun 14, 2021

Complex circle page complete
Peter - Jun 14, 2021

Complex number page fix
Peter - Jun 14, 2021

Added Complex number page
Peter - Jun 14, 2021

Argand Diagram page
Peter - Jun 13, 2021

Restart feature and -powers bug fix
Peter - Jun 12, 2021

Regression line check box
Peter - Jun 12, 2021

Graph details check box
Peter - Jun 11, 2021

Wave Page
Peter - Jun 11, 2021

Input functions without spaces
Peter - Jun 11, 2021

Regression line page complete
Peter - Jun 11, 2021

Added the Regression line page
Peter - Jun 11, 2021

Mandelbrot Set Page

Peter 75f7533 1 changed file +81 -4

```
I have added a fully functioning Mandelbrot set page. I will probably add to this in the future, with
more colours, and maybe a slider for the resolution.
```

graph.py

```
-                   if int(xIn) > limit or int(xIn) < -limit or int(yIn) > lim
it or int(yIn)<-limit:
+                   if float(xIn) > limit or float(xIn) < -limit or float(yIn)
> limit or float(yIn)<-limit:
                        raise
                    x.append(float(xIn))
                    y.append(float(yIn))
@@ -424,7 +424,7 @@ class ComplexPage2(tk.Frame):
          HalfLine_button.pack(pady=5)
          bisector_button= ttk.Button(self, text="Perpendicular Bisector", c
ommand=lambda: controller.show_frame(ComplexPage2))
          bisector_button.pack(pady=5)
-         MBset_button= ttk.Button(self, text="Mandelbrot Set", command=lamb
da: controller.show_frame(ComplexPage2))
+         MBset_button= ttk.Button(self, text="Mandelbrot Set", command=lamb
da: controller.show_frame(MBsetPage))
          MBset_button.pack(pady=5)
          Home_button = ttk.Button(self, text="Back", command=lambda: contro
ller.show_frame(StartPage))
          Home_button.pack(pady=25)
@@ -621,6 +621,84 @@ class half_line_Page(tk.Frame):
          button2.grid(row=6,column=0, pady=10)


+class MBsetPage(tk.Frame):
+    def __init__(self, parent, controller):
+        tk.Frame.__init__(self, parent)
+
+        def button_command():
+            global x, y
+            global limit
+            res = resIn.get()
+            try:
```

Usability

As a PC application, this calculator works well and is easy to navigate with a mouse. However, I do not believe this would work as well as a mobile application. If I was to consider releasing this as a mobile app I would certainly think about adjusting the window size and making the buttons more accessible for touch pad usage, as that is not what it was optimised for.

In conclusion, this application was designed to be a PC app that teachers can use in the classroom and that students could use at home and I believe that it has delivered on these aspects.

Limitations

Mathematics is a complicated subject covering many complex topics. For a calculator to cover everything, would be almost impossible, so I tried to include functions that I knew my audience & stakeholder would find useful for specifically resolving 'A level' problems.

I could have added more complex functions, for example, my app could integrate functions or it could allow the user to write Python directly into it, allowing you to perform statistical analysis, but I had to be realistic with my given time frame. It was important that I remained focused on ensuring that my app met the needs of the average student or teacher as my success criteria dictated, without becoming over-complicated.

Physical limitations: The application only requires a computer with a mouse, keyboard and monitor.

Software limitations: Currently, I can only export it as a windows executable but in the future I hope to get this working for Mac too. The application is not very intensive to run, so doesn't require specialist components but higher performance computers would help when rendering the Mandelbrot Set as this is a memory intensive task.

The Future of My Application

If I had more time, or in future revisions or updates, there are a few things I would add and improve. Extending my program into three dimensions is something that I wanted to try. I did try experimenting with 3D using 'OpenGL' however, I was unable to get this functioning as it was not compatible with my existing code.

For practicality, I would like to consider making this into a mobile application or a web application. I feel this would make my program more accessible and obviously portable. It would allow students to use my program on any device, not just limited to Windows computers. This may involve creating a web server to host my program and making a web page that is compatible with my existing code, that would allow clients to run and use the application online.

Obviously, there are endless mathematical functions to add. As I learn more at school, I find myself wanting to add more features that will help with the work I am currently completing. However, this would be a never-ending task, so I need to keep in mind what the average student would find useful and keep focused on these features.

Conclusion

After months of work, weeks of programming and hours of research, I believe that I have met my specified criteria and succeeded in producing an application that is both capable and user-friendly (Over 3000 lines of code!).

I went well beyond my initial specification, adding loads of extra, unique and useful features, to create a fully functioning graphing calculator that is more than capable of helping both students and teachers perform calculations and plot graphs.

I have personally been using this application in math lessons to help solve SUVAT equations and to plot graphs in my physics classes. It is particularly useful in physics as I can use the built-in stopwatch to time an experiment and record data. I can then plot this data and calculate the regression line to find the gradient which may be the result needed for the experiment.

My stakeholder was very impressed that I had met and exceeded her expectations in the allotted time frame. She complemented me on the vast functionality and yet simplicity of the application.
She has subsequently used it to generate print offs of the Mandelbrot Set to display in her classroom.

Many students have tried it out and I can conclude that the application is functional and fun, yet also both simple and practical to use. This leads me to conclude it has been very successful.

The app in use, in the classroom 8/4/22.

References

Mathologer (2019) ‘500 years of NOT teaching THE CUBIC FORMULA. What is it they think you can't handle?’, August 24th. Available at: https://youtu.be/N-KXStupwsc.

TLMaths (2015) ‘AQA Statistics 1 7.02 Introducing Linear Regression Part 2’, May 24th.
Available at: https://youtu.be/MVJv01xr9lM.

Numberpile (2015) ‘The Mandelbrot Set – Numberpile’, July 25th.
Available at: https://youtu.be/NGMRB4O922I.

Sentdex (2014) ‘How to add a Matplotlib Graph to Tkinter Window in Python 3 - Tkinter tutorial Python 3.4 p. 6’, November 13th. Available at: https://youtu.be/Zw6M-BnAPP0.

Palopha (2020) ‘disable matplotlib toolbar’, April 20th.
Available at: https://stackoverflow.com/questions/13942956/disable-matplotlib-toolbar.