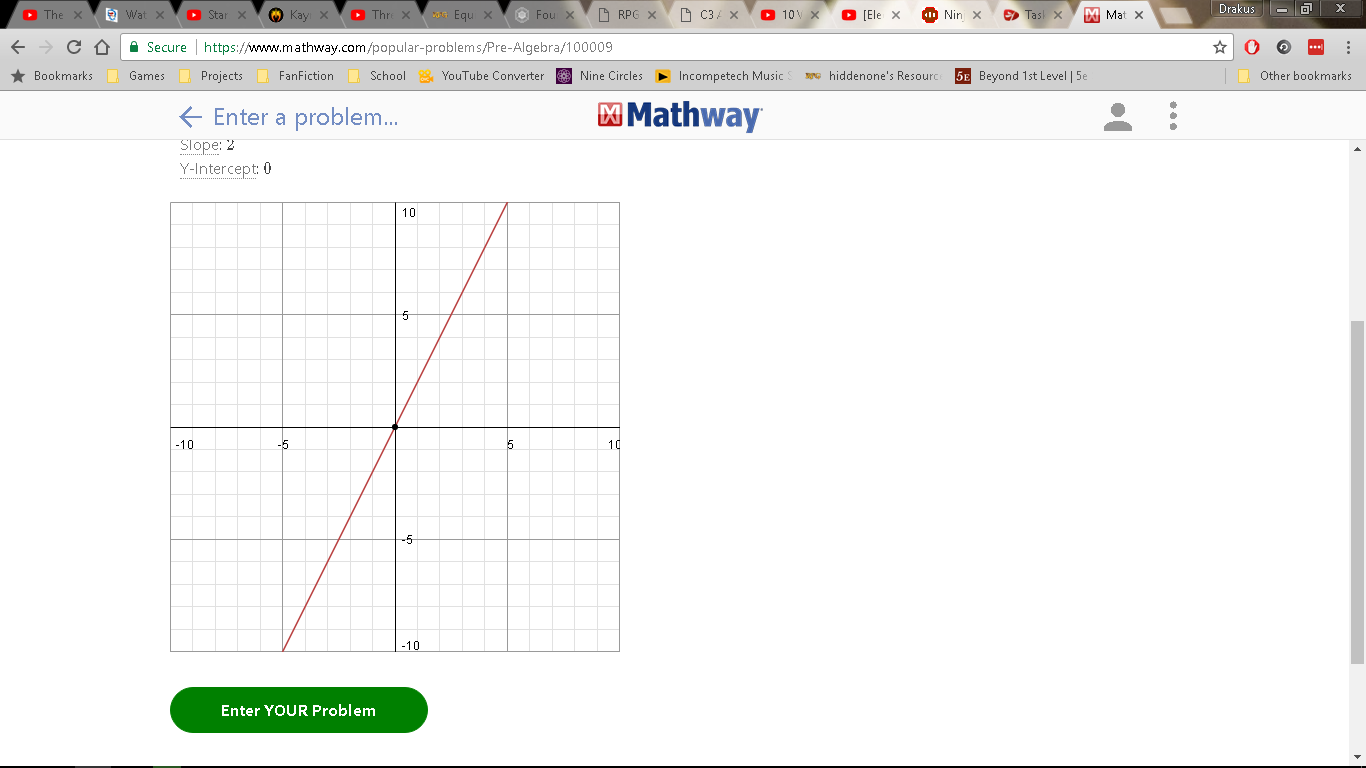
**Problem Statement**

For this project, I wish to make a graphing calculator in Python. A graphing calculator acts, in essence, in the same way as a normal calculator, except for the fact that it has the additional functionality to create curves from algebraic equations it is given.

This application will be required to be able to read the equations is it given and calculate answers for them with given values for the variables that need to be substituted.

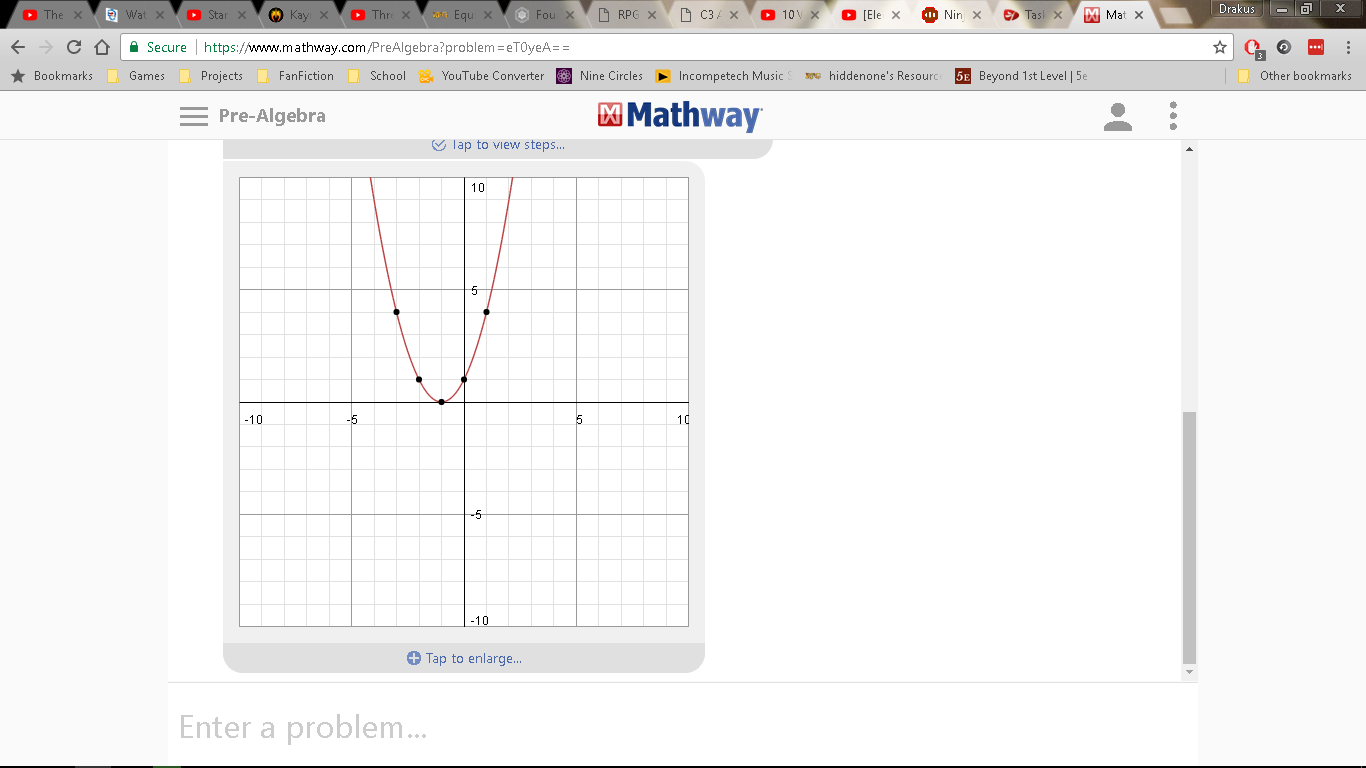
For example, an equation such as would produce a straight line heading to the top right with a steep slope, as shown here:

As one can see, the y-value of any point on this curve is always double the x-value of that same point. This can be proven via substitution into the equation:

Let :

This is known as a linear equation or a straight-line graph, and is shown in the form .

Another type of equation called a quadratic equation produces a parabola when graphed, and is shown in the form :

 For a quadratic equation, a root is defined as a place where the curve touches the x-axis, the horizontal axis. Roots can be found by factorising the equation into the form .

This graph represents the equation . If one substitutes 0 as , we get .

This can be factorised into the form , or . This creates one root at , as if , then .

In addition to this, the gradient of a graph can be calculated via a process called differentiation. This process is one of two processes that form calculus, called differential calculus. Calculus is used to find the properties of curves that cannot be found from simple observation.

In this respect, differential calculus is used to find the exact gradient of a curve at any point of its line. This is done via a process of multiplying the term by the power, and then subtracting one from the power. In a more succinct and understandable way:

Where is the equation to be differentiated, is the coefficient of the term, is the term, and is the power. is the gradient of the equation. After this is done, one can calculate the gradient via substituting into the differential equation.

For example, the gradient of at is calculated via:

Therefore, the differential of is:

Therefore, the gradient is:

My program will likely use this process to calculate gradients for the curves it creates.

As stated before, the program will also function as a calculator, much like the calculator programs online or installed on one’s computer. It will likely use a selection of buttons that can be clicked, and that will enable one to create an expression algebraically or arithmetically. This will be done via use of a GUI program, and will require knowledge of the BIDMAS order of operations.

The BIDMAS order of operations is a set of instructions that is carried out when calculating an answer that states which operations are carried out in which order, in accordance with their appearance in the expression. B denotes Brackets, I denotes Indices, D denotes Division, M denotes Multiplication, A denotes Addition and S denotes Subtraction. For example, if one had an expression like :

You would eliminate the brackets first, performing the addition in the brackets: .

You would then eliminate the indices: .

You would then eliminate the fraction (division line): .

You would then eliminate the multiplication symbol, making the answer: .

This is the way I will most likely be calculating my expressions in this program.

**Stakeholders**

My general stakeholders will be mathematics students who would be in need of a tool that would allow them to quickly calculate answers to complex problems and whom would also need to quickly sketch shapes of graphs for both revision and question solving purposes, and teachers, who would use the tool generally as a graph sketcher and a tool to quickly solve graphical problems.

**Areas of Concern**

This application will require a Graphical User Interface, or GUI. This would require the addition of a module that would be able to allow the manipulation of visual widgets in a defined window.

Due to the nature of the application, I will require a process to enable calculation of the input given by the user. Rather than using the *eval* function which would allow me to calculate the expression immediately, I shall create my own process to simulate the BIDMAS operations to allow me to completely control and alter the process to account for any logical errors in the calculation of the answer.

This custom calculation process will be much more difficult to create than simply using *eval*, however it would allow me to make my program more complex than would otherwise be possible. It would allow me to make my program more modular, and would allow for the utilisation of operators and functions that wouldn’t ordinarily be possible with *eval*.

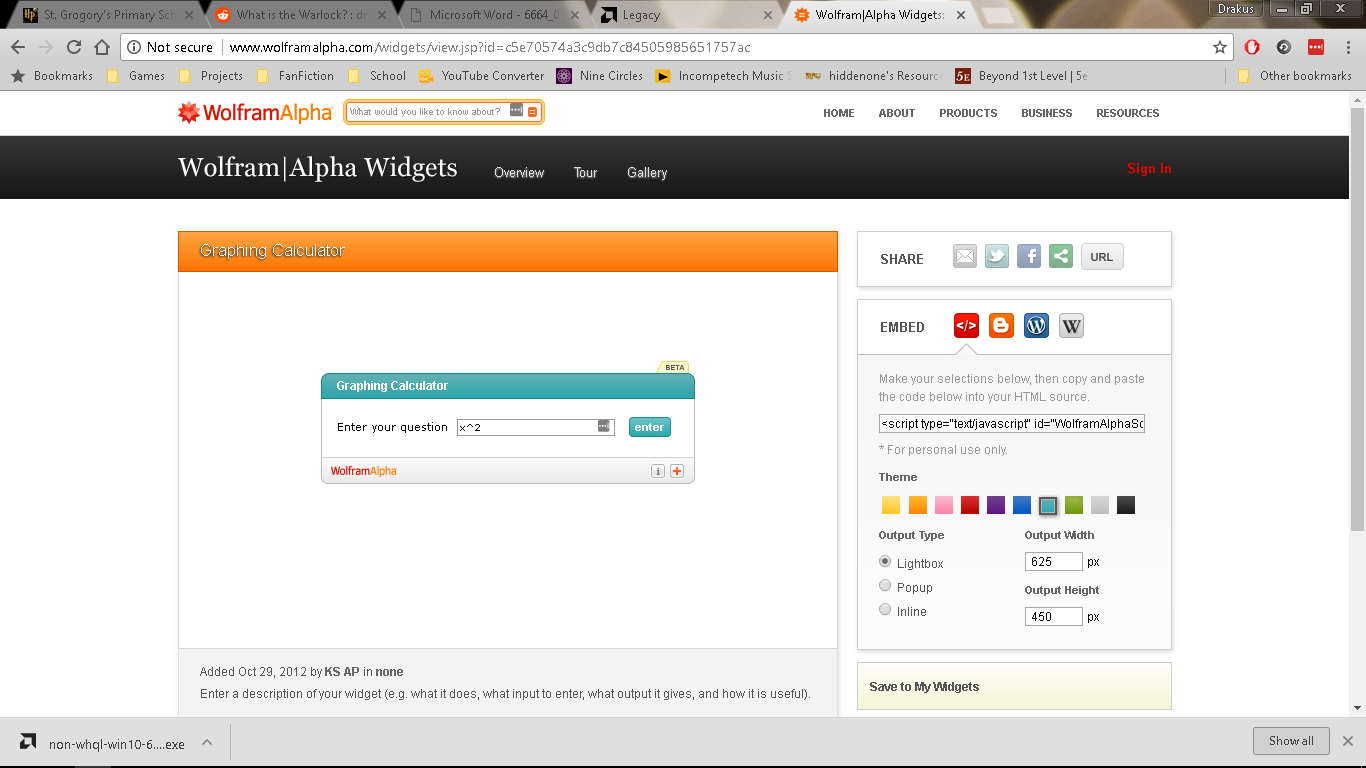
Another problem that would arise would be the creation of curves to match any equation the user inputted. I would be able to solve this problem by utilising the calculation process to calculate a y-value for an x-value that would be subject to the bounds used. This process would be repeated through the width of the window, creating a set of points, one for each column of pixels in the window of the application. A line would be drawn between these points, creating a curve.

Another problem would be the design of the graphical user interface. I have a choice for which GUI module I use, between tKinter and PyQt. I will most likely choose PyQt, as that has a much more professional appearance than tKinter, and also allows one to change the cosmetic settings of widgets in a much more varied way.

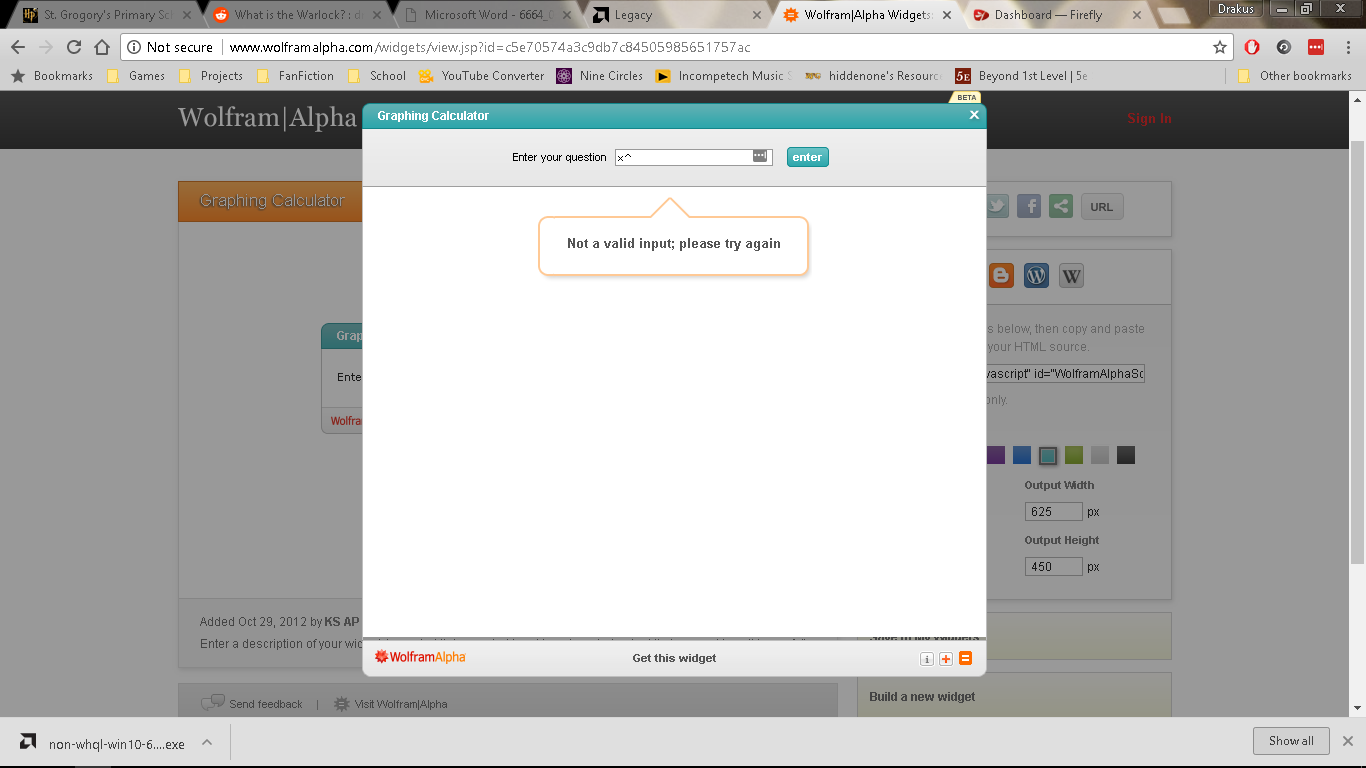
In addition to this, the mathematical and algebraic knowledge of the stakeholders means that I will need a way to format the expressions I use to make them more mathematically correct. For example, x^2 would become x2, and 2\*x would become 2x. This formatting would make it easier to read for the stakeholders who would already be well versed in algebraic notation, and for whom expressions and terms would become a lot more readable and familiar if it was in algebraic notation.

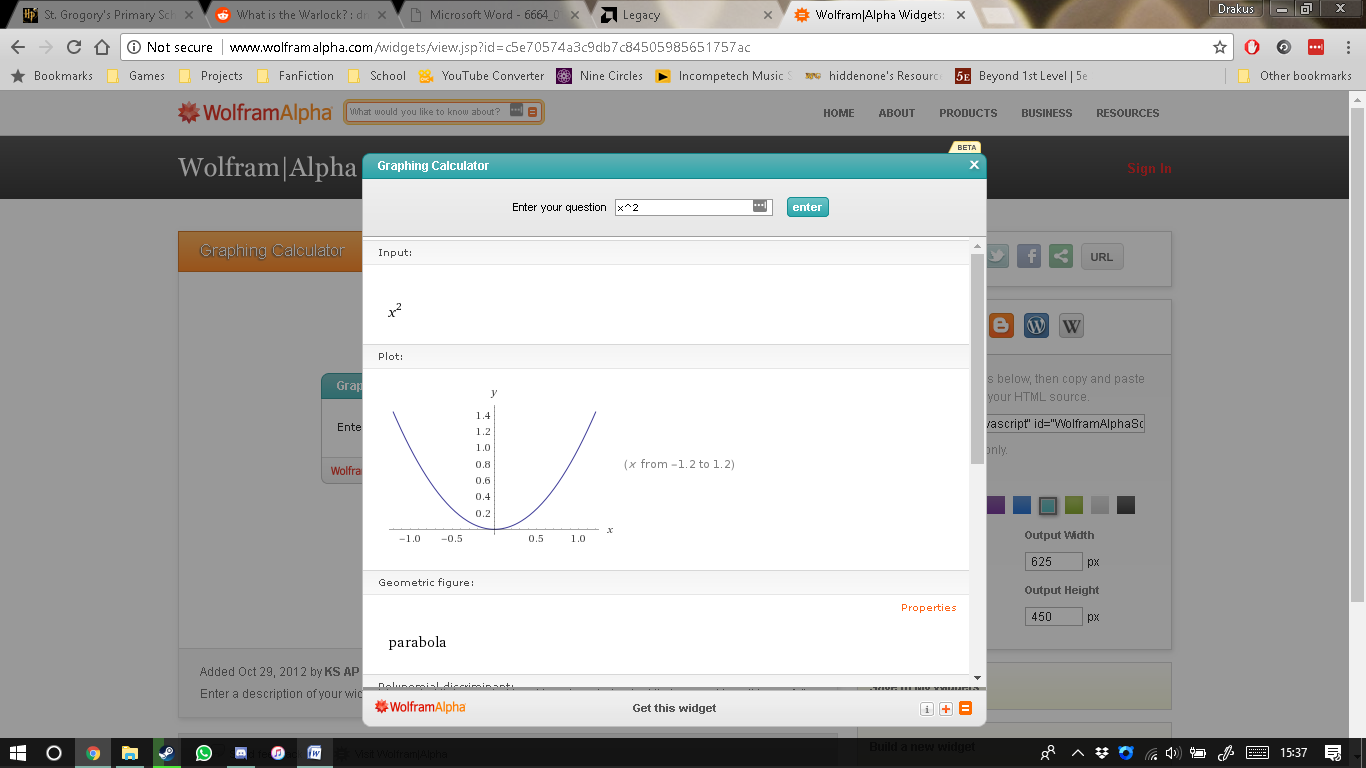
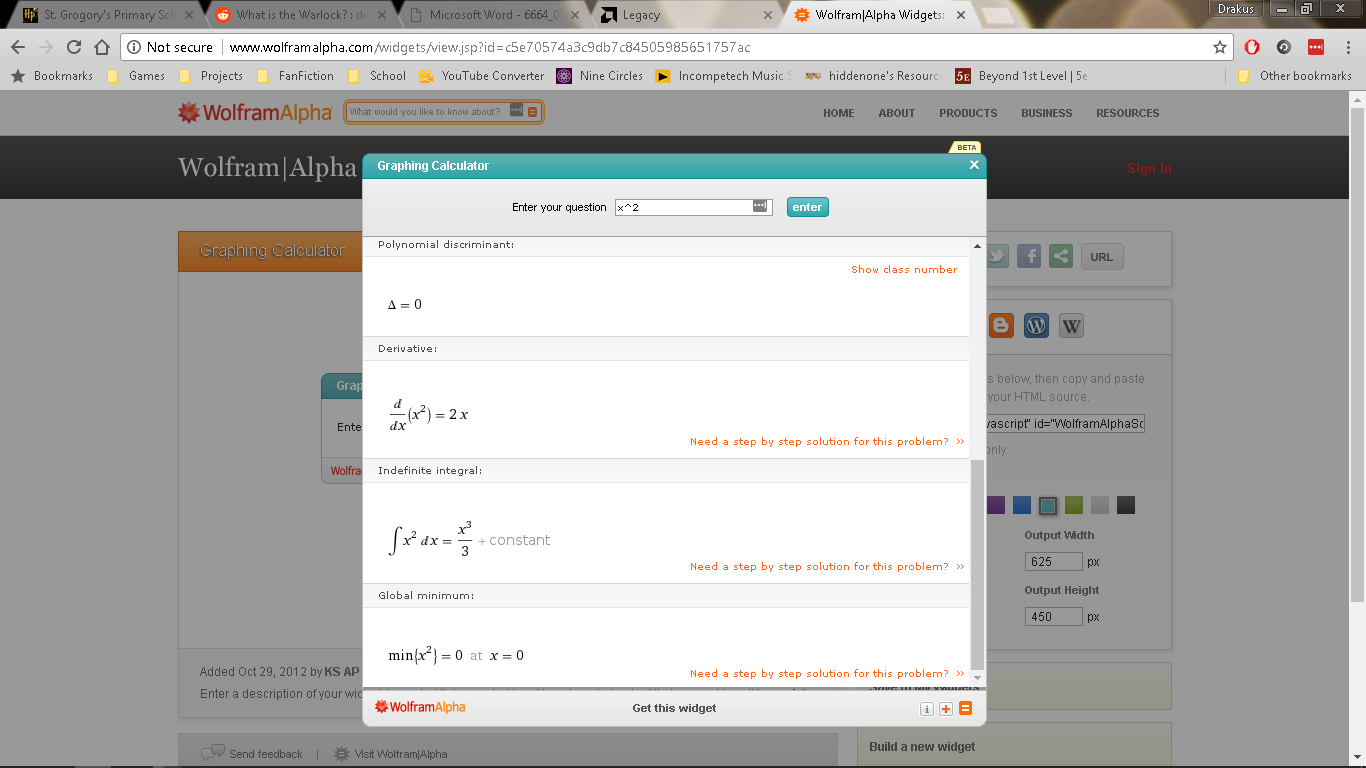
**Research**

**Wolfram Alpha: Graphing Calculator1**

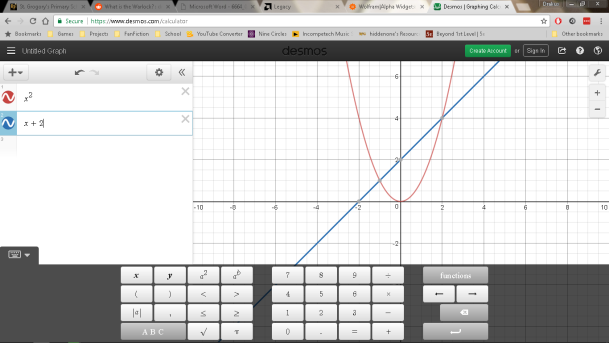
Wolfram Alpha’s Graphing Calculator Widget handles the equations given using particular syntax. For example, a^b would be interpreted as a power of b, and a\*b would be interpreted as a multiplication of a by b. It allows equations to be entered via a text field, as shown:

Ordinarily, this would have a high risk of causing errors in syntax, especially since there does not seem to be any documentation on the correct usage and definition of operators. However, testing shows that an invalid input of an expression gives the display to the left.

On the other hand, the fact that there is no documentation to aid users in the construction of correctly formatted equations acts as a detriment to the user-friendliness of the program, as it would become tedious and difficult to use it if one did not know how it worked.

When this particular calculator computes a curve, it gives a series of properties of properties of the curve, including its differential equation, its integral, and its roots. This helps in allowing the end user to understand how curves differ from one another in a mathematical sense, and allows easy understanding of how to calculate the various properties of the curve. This property of the program is shown to the left. As can be seen, the program formats all of the resulting expressions correctly according to algebraic convention, and gives a section of the graph bounded between x = -1.2 and x = 1.2. I will most likely utilize this bounding; however will likely make the bounds variable rather than only using one particular bounding. This is because for certain curves, it may be unreasonable to use certain bounds for the inadequacy of the information obtained from the observation of these particularly bounded graphs.

For example, if one wanted to know the structure of a sin(x) curve, one would not use a bound of -1.2o to 1.2o, as that would not enable one to see the full wave shape of the curve, instead seeing what would look much like a line graph. This is because a sine graph’s turning points occur when x = ((n \* 180) – 90)o, meaning that using bounds of -1.2o to 1.2o would not include any of the turning points. As such, I believe the use of bounds capable of being adjusted would be useful for my program.

**Desmos Calculator2**

Another program that acts much more like what I would wish to achieve is Desmos calculator. Its main interface utilises a structure much like a calculator, with various clickable buttons that are used to enter terms and functions into the graph equation. Interestingly, it also utilises list functionality to allow for the graphing of multiple curves. This would be a useful addition to the program, as it would allow the end user to see the intersection points of multiple curves, as is a common problem in mathematics when dealing with Cartesian equations on the (x, y) plane.

An interesting property of the formatting used for equations in Desmos calculator is that rather than completely eliminating the multiplication symbol as is a general algebraic convention, it instead replaces it with the less commonly used ∙ symbol. This means that rather than ‘ab’, ‘a\*b’ would instead be formatted as ‘a∙b’. I will most likely not be using this formatting in my program.

**Success Criteria**

* The program will require expressions to be inputted via a series of functional buttons.
* The program will require that all expressions and equations be formatted to be consistent with algebraic notation.
  + The program should correctly format:
    - Indices (“x^2” -> “x2”)
    - Variable Multiplication (“2\*x” -> “2x”)
    - Bracket Multiplication (“2\*(” -> “2(”)
* The program will require that expressions entered into the calculator are calculated correctly.
  + The algorithm used for calculation should be custom and should use BIDMAS notation.
  + The algorithm should be able to withstand calculating answers from expressions with nested brackets, as well as expressions with multiple surface-level brackets interacting with each other.
  + The algorithm should be able to withstand operations on negative and fractional numbers (abstract and decimal).
* The program should create correctly a multitude of basic curves.
  + The program should create:
    - Linear curves.
    - Quadratic curves.
    - Cubic curves.
    - Exponential curves.
    - Reciprocal curves.
* The program allows for the input of variable bounds.
  + The program should allow:
    - -1.2 ≤ x < 1.2
    - -10 ≤ x < 10
    - 0 ≤ x < 10
    - -90 ≤ x < 90
    - 0 ≤ x < 360
* All functions should work correctly.
  + Sine.
  + Cosine.
  + Tangent.
  + Logarithm.
  + Natural Logarithm.
* The program has a round function that should at least be able to round answers to 5 d.p. worth of precision.

**Variables**

I have decided to name the main variables before I start the creation of the program.

|  |  |
| --- | --- |
| Name | Function |
| Term | Stores the current term that is to be added to the calculator’s expression. |
| Equation | Stores the equation to be answered and calculated. |
| CalculatorExpression | A formatted version of the Equation that will be displayed to the user on the calculator screen. |
| ExpList | The Equation variable, split into a list of characters that can be searched and altered by the Calculate function |
| e | Shorthand for the ExpList variable that will be used in the Calculate function. |
| x | Iterative Variable. Used in the calculator grid search and the graph drawing. |
| y | Iterative Variable. Used in the calculator grid search, and the graph drawing loop as a shorthand for the calculated value for each point. |
| i | Iterative Variable. Used in non-Cartesian lists. |

**Processes**

I have decided to name the main functions before I start the creation of the program, as it would make it easier to refer back to them when needed.

|  |  |
| --- | --- |
| Name | Function |
| Calculate | Calculates the answer to an expression given to it using the BIDMAS process. The input expression has to be in string form. |
| Differentiate | Differentiates an expression given to it using calculus. The input expression has to be in string form. |
| Menu | Creates the main menu for the program. Creates the three menu buttons at the start of the program’s runtime. |
| MOVECalculator | Moves the user to the calculator display screen. This is the screen with the various calculator buttons, including “3”, “+” and “=”, for example. |
| MOVEGraphCreation | Moves the user to the graph creation screen, which is where you can create a graph to be generated and displayed to you within specific bounds. |
| MOVEGraph | Moves the user to the graph display screen, where the graph is displayed to the user within specific bounds, and including axes if applicable to the given bounds. |

As can be seen here, the first thing that would occur when the program is started is the creation of the buttons with the labels ‘Calculator’, ‘Graph’ and ‘Exit’. These buttons are created in a vertical line as menu options. Every frame, the program would check for three things.

* The program would check for whether the ‘Calculator’ button has been clicked, and when it is, it sends a signal to run the MOVECalculator function which changes the screen to that of the calculator display.
* The program would check for whether the ‘Graph’ button has been clicked, and when it is, it sends a signal to run the MOVEGraphCreation function which changes the screen to that of the graph creation screen.
* The program would check for whether the ‘Exit’ button has been clicked, and when it is, it sends a signal to close the program.

This flowchart describes the MOVECalculator function. As can be seen, the program would first read the grid which contains the calculator layout, line by line. Each index it reads the string of, it would decide what to do based on a series of selection statements.

* If the grid(x, y) = “ ”, it will not create a button and will instead leave that area blank.
* If the grid(x, y) is only one character long, it will create a button of a small size.
* An index in grid(x, y) of a larger length will make the program create a button of a respective length, and will make the label the contents of that string when all spaces are eliminated. For example, an index of “ GRAPH ” would make the label become “GRAPH”.
* If an index contains “ROUND”, it will create a spin box instead of a button. This spin box will contain the number of decimal places the resulting answer is to be rounded to.

This flowchart describes the MOVEGraph function. As can be seen, the program will take the bounds given to it in a string format, and calculates an answer to them using the Calculate function. This is because some bounds may be in standard form because of their immense or tiny size. Next, it calculates the differential of the equation in preparation to create the gradient of the equation at every x-value.

Then, it will initiate a loop of a finite amount of iterations equal to the width of the screen in pixels. At each iteration, it calculates both the y-value of the curve at that x-value (the current number of iterations) using the Calculate function, and calculates the gradient at that x-value using the differential equation obtained before. It will store these values in lists.

After this is done, it will find where O(0, 0) (the origin of the axes of this curve) is in the window, and draw a horizontal line at its y-position and a vertical line at it’s x-position. These become the axes of the curve, which are relative in their placement on the screen based on the position of the curve in relation to them.

After the axes are drawn, it will again iterate like before, drawing a line from the point on the screen given by (i, Points[i]) to the point given by (i+1, Points[i+1]). This means that the curve will be drawn assuming a perfect area obtained by the trapezium rule, with the number of trapeziums used being . This means that if zoomed into far enough, the curve would be very angular, not actually being a curve at all but a collection of joined up straight lines with varying angles. However, this lack of resolution shouldn’t matter due to the curve creating enough trapeziums to make this effect almost impossible to distinguish to the human user.

After the graph is created and drawn, it creates a ‘Close’ button in the top left corner this ‘Close’ button would take the user back to the title screen.

**Testing**

These are the series of testes I have done to ensure the main parts of the program are working correctly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Number | Test Description | User Input | Expected Result | Actual Result |
| 1 | Testing the menu’s ‘Calculator’ button. | Clicking the ‘Calculator’ button. | The user is moved to the calculator screen. | As expected. |
| Testing the menu’s ‘Graph’ button. | Clicking the ‘Graph’ button. | The user is moved to the graph creation screen. | As expected. |
| Testing the menu’s ‘Exit’ button. | Clicking the ‘Exit’ button. | The program closes. | As expected. |
| 2.1.1 | Testing the calculator’s numerical buttons (0, 1, 2, 3, etc.). | Clicking the number buttons. | The numbers are appended to the expression to be calculated. | As expected. |
| Testing the calculator’s operator buttons. (+, -, xn, etc.). | Clicking the operator buttons. | The operators are appended when appropriate to the expression to be calculated. | As expected. |
| 2.1.2 | Testing the calculator’s sine function. | Inputting ‘sin(90)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘sin(0)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘sin(180)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘sin(54.235)’ into the calculator. | The calculator outputs ’0.8114209979’ or some expanded or contracted variation of it. | As expected. |
| Inputting ‘sin(45)’ into the calculator. | The calculator outputs ‘0.7071067812’ or some expanded or contracted variation of it. | As expected. |
| 2.2.1 | Testing the calculator’s cosine function. | Inputting ‘cos(90)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘cos(0)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘cos(180)’ into the calculator. | The calculator outputs ‘-1’. | As expected. |
| Inputting ‘cos(54.235)’ into the calculator. | The calculator outputs ’0.5844621152’ or some expanded or contracted variation of it. | As expected. |
| Inputting ‘cos(45)’ into the calculator. | The calculator outputs ‘0.7071067812’ or some expanded or contracted variation of it. | As expected. |
| 2.2.2 | Testing the calculator’s tangent function. | Inputting ‘tan(90)’ into the calculator. | The calculator expresses a ‘Complex Error’ in red text. | As expected. |
| Inputting ‘tan(0)’ into the calculator. | The calculator outputs ‘0’. | The calculator expresses a ‘Complex Error’ in red text. |
| Inputting ‘tan(180)’ into the calculator. | The calculator outputs ‘0’. | The calculator expresses a ‘Complex Error’ in red text. |
| Inputting ‘tan(54.235)’ into the calculator. | The calculator outputs ’1.388320948’ or some expanded or contracted variation of it. | As expected. |
| Inputting ‘tan(45)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| 2.2.3 | Testing the calculator’s logarithm function. | Inputting ‘log(1)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘log(10)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘log(100)’ into the calculator. | The calculator outputs ‘2’. | As expected. |
| Inputting ‘log(107)’ into the calculator. | The calculator outputs ‘7’. | As expected. |
| Inputting ‘log(65)’ into the calculator. | The calculator outputs ’1.812913357’ or some expanded or contracted variation of it. | As expected. Gave a more precise value for the output than expected (‘…566’ instead of ‘…57’). |
| 2.2.4 | Testing the calculator’s natural logarithm function. | Inputting ‘ln(1)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘ln(e)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘ln(√e)’ into the calculator. | The calculator outputs ‘0.5’. | As expected. |
| Inputting ‘ln(107)’ into the calculator. | The calculator outputs ’16.11809565’ or some expanded or contracted variation of it. | As expected. Gave a more precise value for the output than expected (‘…651’ instead of ‘…65’). |
| Inputting ‘ln(65)’ into the calculator. | The calculator outputs ‘4.17438727’ or some expanded or contracted variation of it. | As expected. Gave a more precise value for the output than expected (‘…2699’ instead of ‘…27’). |
| 2.2.5 | Testing the calculator’s modulus (absolute value) function. | Inputting ‘abs(1)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘abs(-1)’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘abs(-√e)’ into the calculator. | The calculator outputs ‘1.648721271’ or some expanded or contracted variation of it. | As expected. |
| Inputting ‘abs(-107)’ into the calculator. | The calculator outputs ’10000000’. | As expected. |
| Inputting ‘abs(-π)’ into the calculator. | The calculator outputs ‘3.141592654’ or some expanded or contracted variation of it. | As expected. Gave a more precise value for the output than expected (‘…536’ instead of ‘…54’). |
| 2.3.1 | Testing for a correct formatting of multiplied variable terms. | Inputting ‘2\*x’ into the calculator. | The calculator formats to ‘2x’ as expression is being inputted. | As expected. |
| 2.3.2 | Testing for a correct formatting of multiplied numerical terms. | Inputting ‘2\*2’ into the calculator. | The calculator formats to ‘2×2’ as expression is being inputted. | As expected. |
| 2.3.3 | Testing for a correct formatting of multiplied open brackets | Inputting ‘2\*(’ into the calculator. | The calculator formats to ‘2(’ as expression is being inputted. | As expected. |
| 2.3.4 | Testing for a correct formatting of multiplied open brackets | Inputting ‘…)\*2’ into the calculator. | The calculator formats to ‘…)×2’ as expression is being inputted. | As expected. |
| 2.4.1 | Testing for a correct formatting of single character indices. | Inputting ‘2^2’ into the calculator. | The calculator formats to ‘22’ as expression is being inputted. | As expected. |
| 2.4.2 | Testing for a correct formatting of bracketed (multi-character) indices. | Inputting ‘2^(2\*x)’ into the calculator. | The calculator formats to ‘2(2x)’ as expression is being inputted. | As expected. |
| 2.4.3 | Testing for a correct formatting of single character indices when there is another character after the power. | Inputting ‘2^2+’ into the calculator. | The calculator formats to ‘22+’ as expression is being inputted. | As expected. |
| 2.4.4 | Testing for a correct formatting of single character indices when there is another character after the power. | Inputting ‘2^(2\*x)+’ into the calculator. | The calculator formats to ‘2(2x)+’ as expression is being inputted. | As expected. |
| 2.5.1 | Testing the addition operator. | Inputting ‘2+2’ into the calculator. | The calculator outputs ‘4’. | As expected. |
| Inputting ‘2+(-2)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘2+-2’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘-2+2’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘2+2.5’ into the calculator. | The calculator outputs ‘4.5’. | As expected. |
| Inputting ‘2+(-2.5)’ into the calculator. | The calculator outputs ‘-0.5’. | As expected. |
| 2.5.2 | Testing the subtraction operator. | Inputting ‘2-2’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘2-(-2)’ into the calculator. | The calculator outputs ‘0’. | As expected. |
| Inputting ‘2--2’ into the calculator. | The calculator outputs ‘4’. | As expected. |
| Inputting ‘-2-2’ into the calculator. | The calculator outputs ‘-4’. | As expected. |
| Inputting ‘2-2.5’ into the calculator. | The calculator outputs ‘-0.5’. | As expected. |
| Inputting ‘2-(-2.5)’ into the calculator. | The calculator outputs ‘4.5’. | As expected. |
| 2.5.3 | Testing the multiplication operator. | Inputting ‘2\*2’ into the calculator. | The calculator outputs ‘4’. | As expected. |
| Inputting ‘2\*(-2)’ into the calculator. | The calculator outputs ‘-4’. | As expected. |
| Inputting ‘-2\*2’ into the calculator. | The calculator outputs ‘-4’. | As expected. |
| Inputting ‘2\*2.5’ into the calculator. | The calculator outputs ‘5’. | As expected. |
| Inputting ‘2\*(-2.5)’ into the calculator. | The calculator outputs ‘-5’. | As expected. |
| 2.5.4 | Testing the division operator. | Inputting ‘2/2’ into the calculator. | The calculator outputs ‘1’. | As expected. |
| Inputting ‘2/(-2)’ into the calculator. | The calculator outputs ‘-1’. | As expected. |
| Inputting ‘-2/2’ into the calculator. | The calculator outputs ‘-1’. | As expected. |
| Inputting ‘2/2.5’ into the calculator. | The calculator outputs ‘0.8’. | As expected. |
| Inputting ‘2/(-2.5)’ into the calculator. | The calculator outputs ‘-0.8’. | As expected. |
| 2.5.4 | Testing the index operator. | Inputting ‘2^2’ into the calculator. | The calculator outputs ‘4’. | As expected. |
| Inputting ‘2/(-2)’ into the calculator. | The calculator outputs ‘0.25’. | As expected. |
| Inputting ‘-2^(0.5)’ into the calculator. | The calculator expresses a ‘Complex Error’ in red text. | As expected. |
| Inputting ‘-2^2.5’ into the calculator. | The calculator expresses a ‘Complex Error’ in red text. | As expected. |
| Inputting ‘2^(-2.5)’ into the calculator. | The calculator outputs ‘0.1767766953’ or some expanded or contracted variation of it. | As expected. Gave a more precise value for the output than expected (‘…52966369’ instead of ‘…53’). |