**Modelling**

To create a graphical calculator, the focus will be the software. I will be coding it in Python, relying very little on external libraries (such as NumPy, SciPy etc.). The project needs three general algorithms, as well as some more specific ones for computing the different mathematical functions. The main algorithms needed are explained below.

The aim of the project is to be able to plot graphs of *explicit* (defined in terms of a single variable):

* Polynomials
* Radicals
* Exponentials
* Reciprocals
* Logarithmic graphs
* Trigonometric graphs (sin, cos, tan and reciprocals)
* Parametric graphs
* Differential equations
* Indefinite integrals
* Combinations of the above

Firstly, I need to create a **parser**.

The parser needs to be able to take an input of a human-readable function (with some constraints - all functions will be *explicitly* defined, that is, in terms of a single variable, and correct mathematical syntax is required)

The algorithm I will use will convert the entered function from infix to postfix notation, having pre-processed some operators and implicit multiplication using regular expressions. Infix notation is the regular notation in which calculations are generally written by humans, following the rules of BIDMAS to determine order of operation. [1]

Postfix notation, which is also referred to as Reverse Polish notation, is where operators follow the operands, rather than when they precede the operands as in infix notation. Due to this, it allows the expression to be evaluated left to right, rather than having to consider the order of operations. This is done using a stack, which is useful for processing expressions automatically. It also removes the need for parentheses. [2]

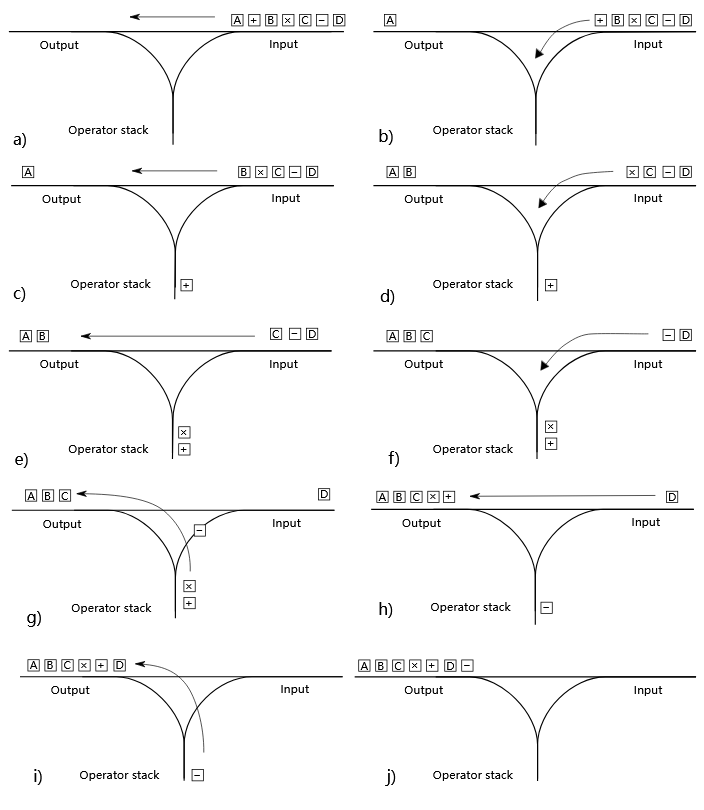
To convert from infix to postfix notation, I will use an algorithm developed by Dijkstra, known as the shunting yard algorithm. The way this algorithm works is that the expression is iterated through from left to right. There is an output queue and an operator stack, where tokens in the expression are added to when encountered, following set rules. [3]

Associativity is also a factor in the algorithm. Left-associative operators, such as the multiply operator, are where operations can be grouped from left to right. This is in contrast with right-associative operators, such as exponentiation, where operations are grouped from right to left.

Whenever a number is encountered, it is pushed to an output queue. If a function, such as the sine function, is encountered, it is pushed to the operator stack. If an operator (*x*) is encountered, the algorithm iterates through each operator currently in the stack (*y*). If *y* is not a left parenthesis and has greater precedence than *x* or has the same precedence with *x* being left-associative, then *y* is popped from the stack to the output. When *y* is not popped, the iteration stops, and *x* is then pushed onto the stack.

There are more detailed rules based on functions that take multiple inputs, and parentheses that comprise functions, but these are not important to the base parsing function that I will implement. Once there are no tokens left to iterate through, all the operators on the stack are popped to the output queue.

Below is a graphical illustration of the shunting yard algorithm for basic operators: [4]



We can see how the tokens are processed from left to right, and the operators are pushed and popped from the stack as needed. In this example, the letters represent numbers, so are pushed to the output when encountered. The other operators are treated according to the BIDMAS order of operations. An explanation of this example is as follows:

* 1. The number A is pushed to the output.
  2. The operator + is pushed to the stack.
  3. The number B is pushed to the output.
  4. The operator x is pushed to the stack. As the + operator has lower precedence, it is not popped from the stack.
  5. The number C is pushed to the output.
  6. The operator - is pushed to the stack. The x operator has a higher precedence so is popped to the output. The + operator has the same precedence and the - operator is left-associative, so it too is popped from the stack.
  7. The number D is pushed to the output.
  8. The operator - is popped from the stack to the output as there are no more tokens to process.

I also need to do some pre-processing as mentioned above to convert the inputted expression into a form suitable for processing with this algorithm. Firstly, I will need to write out operators explicitly. For some operators, such as exponentiation, it will only render a superscript on the user interface, but in the backend, it will use the carat operator, so I will not need to process it further. The main operation to process is implicit multiplication, that is, AB as opposed to A\*B or AxB. I will do this using regular expressions (regex). I can search for valid combinations of two letters next to each other, or a number to the immediate left of a letter and appropriately insert the multiplication operator.

Next, I will need to create a **plotter**.

The plotter algorithm will firstly define a set interval for the function to be evaluated across, and then it will evaluate the function at specific points. It will then use a method of interpolation to join up the curve between the evaluated points, hopefully producing a smooth render, thus converting the vector graphics of the curve into a bitmap format for pixel display in Pygame.

I have done some research on how best to define the interval. I was initially thinking to calculate a number of turning points of the function, and to define the interval so all the turning points lie within it, and also thought to scale the axes appropriately, for example, the sine function in degrees would need to have a larger scale on the x axis. However, after my research into current systems, which I talked about earlier, I realised that this was unnecessary. The majority of graphical calculators, both online and physical devices, have fixed axes, that every graph is plotted to, with features for users to pan and zoom the graph themselves. As I am intending to include these features anyway, it would be a waste of time to try and create an elaborate algorithm to automatically scale the graphs, given the number of edge cases I would have to consider.

As I will discuss later in the hardware section, I have decided on a screen that is 320x480 pixels. I will therefore evaluate points appropriately, at a frequency of x values that will produce a smooth curve through interpolation. I will test out various methods of interpolation with different amounts of values until I find a sweet spot that minimizes computational intensity whilst maximising the smoothness of the function.

I have researched a few different interpolation methods [5] and decided that I will firstly try linear interpolation [6]. This is where each point is joined to the next by a straight-line segment. This method draws a line between the two points, by finding the gradient between the points and then drawing the line. Pygame has a line draw function, which takes a start and end point, so I will use this for my implementation.

There are other types of interpolation [7], such as polynomial [8] and spline [9] interpolation. These will produce a smoother curve, but at the cost of processing power, and will take longer to implement. For this reason, I am intending to compute more points on the curve and then use linear interpolation, rather than computing less points but using a different type of interpolation. Hopefully, this should produce an adequately smooth curve [10].

Next, I will need a **translator**.

This will be a set of functions that can pan and zoom the graph on the screen. I will have a set of 4 arrow buttons on the calculator, as well as either two zoom buttons or a two-way rocker switch with the same function. I will decide which one to use based on the available space on the calculator. There will also be some sort of home button, which will reset the pan and zoom to the defaults. I will describe these buttons in more detail in the hardware section.

Whenever one of the arrow buttons is pressed, the boundaries of the screen will change a specified amount, based on how long it is pressed for. When the button is released, I will recalculate the plotting for the graph with the new bounds. This will help performance as I am only recalculating when the button is released, not while it is being pressed. The zoom button will scale both axes proportionally to the length of the button press, keeping the centre of the screen at the same point.

The maths behind this should be fairly simple - just scalar multiplication or addition. The main purpose of the function is to ensure an ease of use for the user, rather than to add any more information.

For the graphs themselves, simply calculating for a given value of x with the postfix notation should work, as they will all be defined explicitly. If an error occurs, such as an attempted division by zero, then I can mark that point as an asymptote to the curve. A potential hazard could be if two points are calculated that straddle an asymptote, so the linear interpolation joins them up in what appears to be a vertical line - this however should be fine for the purposes of graphical display as it is clear that an asymptote exists there.

Parametric graphs, differential equations, and indefinite integrals will each require a different algorithm to plot however, as they are not given in explicit Cartesian form. I will use numerical methods to plot them rather than attempting to manipulate the given expression, as it would require writing an entirely new method of parsing expressions that also enables manipulation - something that would be both extremely tricky to do and that has already been done (by the Sympy library).

Firstly, I will create an algorithm to deal with **parametric graphs**.

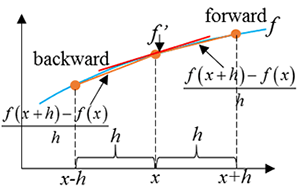
The parametric graphs will be defined explicitly as functions of t, that is, x(t) and y(t). An example is:

As we are working within set boundaries of the graph, the algorithm would firstly need to find the t value(s) that corresponds to each x value and then calculate the resulting y value(s) from it. I had come up with an idea about how to do this by searching for the roots of the equations formed [11] using root bisection [12], however, this is computationally intensive, will not work for every set of equations, and does not account for multiple roots.

Instead, I have come up with another method that I believe should work fairly well. I will firstly do a coarse sweep across a wide, arbitrary range (say -1000 to 1000 with intervals of 1). These values are for the parameter t, and whenever they give a point that lies within the bounds, I will store it in a list. I will then use recursion to search at finer intervals around all the t values within the bounds, and so on, probably with a recursion depth of 4 but depending on performance. I am not sure how well this method will work, but I will not know until I implement and test it, so for now, I will be taking this approach.

Next, I will use numerical methods to plot **differential equations**, for example:

I am intending to use finite difference approximations [13] of quite a low step size. I will calculate the gradient of the secant line between two points very close together on the curve (although the step size cannot be too small due to floating point precision errors). Firstly, I will need a list of x values within the boundaries, as with the other graphs. I will then evaluate the function at that x value, as well as the function at the value of x plus the small h value. The difference of these two y values, divided by h, will give an approximation for the slope at that point, which can be taken as the y value to plot on the screen, thus plotting a derivative. I will experiment with different difference methods - forward, backwards and central difference. These are essentially when h is either above or below x, or x is taken as the midpoint of the secant line. Each has a different error margin and is more suitable in certain situations so I may end up creating an algorithm to apply different difference types in different situations.



The above image [14] shows the different types of method graphically. We can see how a small interval is taken, and the gradient of the secant line is used to approximate the derivative at that point. The central difference method is preferred by most people as it is second order in h, that is, the error margin for the method is O(h²), as opposed to O(h) for the other two methods [15]. Due to this, I will implement the central difference method first and only try the other two if the resulting graph is not accurate enough.

I also need to use numerical methods to plot **indefinite integrals**, for example:

I will be using the trapezium rule [16], which numerically approximates the area under the curve. For each interval, the area will be cumulative and can be plotted as the y value of the corresponding x value. This will shift the graph down by the amount of area that is under the curve to the left of the graph, as there is no way of representing an infinite area. It will thus plot what is known as the antiderivative.

As with the other graphs, I will firstly need a discretized list of points spanning the domain. I will then apply the trapezium rule cumulatively, like so for each interval between the points:

This will give a list of areas to plot as the indefinite integral. Below shows a graphical illustration of how the trapezium rule works, and how the accuracy increases as the number of points in the domain does [17]. It can clearly be seen how the trapezia either overestimate or underestimate the curve, depending on the sign of the derivative at that point, and thus how with smaller trapezia the error margin is minimized. The trade-off is performance again - I will have to choose a number of strips that balances the compute time with accuracy.

A graph of a trap

AI-generated content may be incorrect.

A graph with red lines and numbers

AI-generated content may be incorrect.

A graph with red lines and numbers

AI-generated content may be incorrect.

A graph of a trap

AI-generated content may be incorrect.

A graph of a trap

AI-generated content may be incorrect.

A graph of a trap

AI-generated content may be incorrect.