Chapter 1

Computational Methods

1.1 Abstraction, Heuristics and Computability

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

1. The input of expressions, and their conversion into a form that the computer can then manipulate.
2. The drawing of the function onto the screen.

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

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

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

1. Let x be our starting value, $f(x)$ our function and dx be a small value. Now draw a line between the points $(x, f(x))$ and $(x + dx, f(x + dx))$; $(x + dx, f(x + dx))$ and $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$ and $(x + (n + 1) \cdot dx, f(x + (n + 1) \cdot dx))$. Here we are effectively creating line segments and joining them together to approximate the curve. As $dx \to 0$ our approximation tends towards the real curve. However as $dx \to 0$ the time taken tends to infinity and therefore we only need to make dx small enough so that the human eye can not see individual line segments. Also as a practical note, as $dx \to 0$, the pixel density required to view the curve tends to infinity, therefore dx needs to be big enough so that the dx is not less than the individual pixel width.
2. In the method above we approximated a curve by creating line segments, here we will approximate a curve by using individual points. Let x be our starting value, $f(x)$ our function and dx be a small value. Now draw a point at $(x, f(x))$; $(x + dx, f(x + dx))$; $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$. This can take a lot of time since calculate where to draw a pixel w number of times, where w is the pixel width. In some cases, for complicated functions, it may be intractable.

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

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

1.2 Logical Approach

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

1.3 Decomposition

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

1. A function class which consists of:
   (a) Converting an expression into a form the computer can manipulate.
   (b) Evaluating the expression for a value x, where $x \in \mathbb{R}$.
   (c) Calculating the roots and turning points of the function.
2. A plot class which consists of:
   (a) Drawing functions onto the screen
   (b) Heuristically adjusts the scale so the graph looks good
   (c) Drawing the axes

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

1.4 Divide and Conquer and Concurrent Processing

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