**Notes about fractals, L-systems and Processing syntax**

**Links**

- https://paulbourke.net/fractals/juliaset/
- https://en.wikipedia.org/wiki/L-system
- https://paulbourke.net/fractals/lsys/
- https://www.youtube.com/watch?v=z7LeTu94qEc
- https://processing.org/reference/

**Aim**

I want to do two things: (1) write a program that allows a user to initialise an L-system and see its visualisation, and (2) create a Julia set visualiser. Both of these topics come under the broader topic of fractals which is something I am interested in from a mathematical perspective already.

**Notes on Processing syntax and functions**

The basic structure of a Processing sketch is composed of a setup() function and a draw() function. The setup() function runs once when the program starts, and is used to initialise variables and environment properties. The draw() function continuously executes the lines of code within it, and is used to process events like mouse/key presses.

After reading the documentation and doing some research into fractal drawing, it seems I will need to use the following language features for the actual drawing process:

| |
|---|
| • pushMatrix() and popMatrix()<br>These functions are used to save and use transformation matrices on a typical LIFO stack to isolate transformations. This will be needed so that we can apply the correct transformation on each line we draw in the iteration. |
| • translate(float x, float y, float z)<br>Translates an object in the x and y directions. Definitely needed to connect all the lines to make the fractal look continuous. |
| • line(float x1, float y1, float x2, float y2)<br>Draws a line from (x1, y1) to (x2, y2). |
| • rotate(float angle)<br>Rotates a shape by the specified angle. Useful to graphically represent L-systems with constants like + or - that specify some sort of rotation. |
| • background(float colour)<br>Sets the colour of the background of the processing window. This is important if our drawing/animation is iterative (which in the case of drawing fractals it certainly will be), and we want to blank out previous iterations. |

- JOptionPanes

Going to use these to accept input from the user to initialise the L-system.

**Notes on L-systems**

Lindenmayer Systems, or L-systems, were introduced by the Hungarian theoretical biologist Aristid Lindenmayer in 1968. L-systems are formal grammars used to model the growth of complex biological structures, such as plants and algae. They have since found applications in computer graphics, procedural generation, and the simulation of natural patterns. There are 4 main components:

1. Alphabet (Variables and Constants):

L-systems use an alphabet consisting of variables and constants.
Variables represent symbols that can be replaced by a set of symbols, while constants remain unchanged.

2. Axiom:

The axiom is the initial string of symbols, typically representing the starting structure or seed of the system.

3. Production Rules:

Production rules define how variables are replaced in each iteration or generation.
A rule consists of a variable and its replacement string.

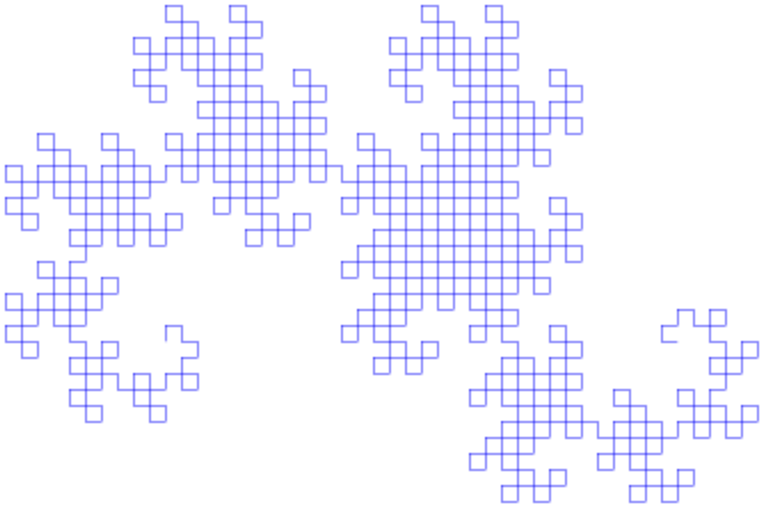
4. Generations:

The L-system evolves over multiple generations, with each iteration applying the production rules to generate a new string.

For example, suppose you have

- Variables F and G and constants + and - (which indicate rotation in a clockwise and anticlockwise direction respectively)
- Axiom F.
- Production rule 1: F -> F + G
- Production rule 2: G -> F - G.

This L-system generates what is known as a dragon curve.



Dragon curve for $n$ = 10

**Notes on the Julia and Mandelbrot Sets**

The mathematics of this is very interesting. The Mandelbrot set is the set of numbers c in the complex plane such that $f_c(z) = z^2 + c$ does not diverge to infinity on repeated iteration for some fixed z, usually z = 0.. The boundary of the Mandelbrot set is perhaps the most famous fractal curve.

The Julia set is very similar, except that we fix the point c and vary z. So that the Julia set for a particular c is the set of numbers z such that $f_z(c) = z^2 + c$ does not diverge to infinity on repeated iterations.

Colouring these sets is very simple. The colours are usually chosen from how fast the sequence diverges, measured by how many iterations it takes until $|z_k|$ has become larger than some certain value that guarantees divergence.

Programming this iterative process shouldn't be too hard. We just need to iterate over each pixel in the window, map it to a complex number, and then run the repeated iterations of the Julia function to determine a colour for it.