Lindenmayer Systems

Lindenmayer systems (also written *L-systems*) are a method of modeling the process of plant growth and describing fractal shapes. In this exercise, we will implement some simple L-systems to create their visual representations. As an example we use the fractal plant shown in Figure 1.

Files and directories in the project:

- `src/lsystem.h` - definitions of the classes you are editing
- `src/lsystem.cpp` - edit this file
- `data_in` contains the specifications of L-systems in JSON format. We provide the code that reads those files
- `data_out` the location of your output sequence and image files
- `data_out_example` example output for the first system
- `reverse_in` - incomplete specifications of L-systems, complete the rules using the pictures in the description
- `reverse_out` - output sequence and image files to be created from the completed rules
- `src/utils` - the code we provide for your convenience
- `build` - build the project here
- `run_deterministic.sh` - the script will run your program for all input files (deterministic systems)
- `run_reverse.sh` - the script will run your program for all input files in `reverse_in`
- `run_stochastic.sh` - the script will run your program for all input files (stochastic systems)

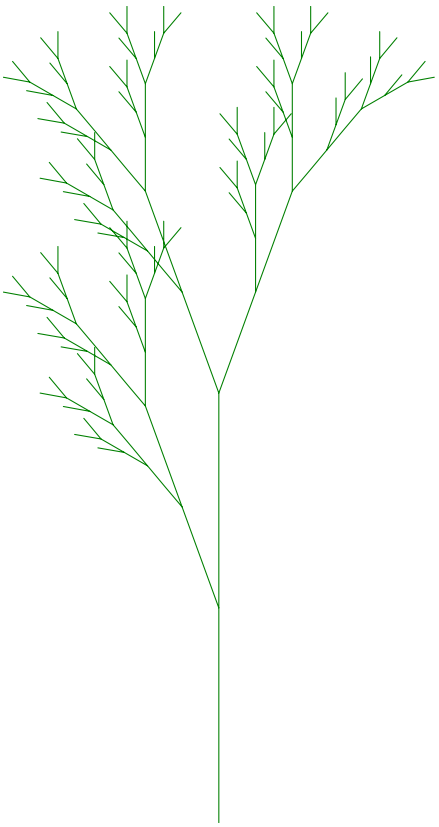


Figure 1: Fractal plant
Iterations $n = 4$
Rotation angle $\delta = 20°$
Initial state: $s_0 = X$
Rules:
$X \rightarrow F[+X]F[-X] + X$
$F \rightarrow FF$

To submit: `src/lsystem.cpp`, `data_out`, `reverse_in`, `reverse_out`

Drawing to images is done with the following command:

```
build/lsys input_data_file output_file_no_extension
```

The images will be saved in SVG (vector graphics), which can be viewed in all web browsers and many image viewer programs. If you wish to convert SVG to PNG, you can use imagemagick:

```
convert a.svg a.png
```

or Inkscape (which is a graphical editor for SVG images):

```
inkscape a.svg --export-png=a.png
```

Interactive view - apart from writing the outputs to files, you can run your system interactively, with the command:

`build/lsys input_data_file`

For example:

`build/lsys data_in/plant_01.json`

Press the right or left arrow to change the rotation angle, up/down arrows (or mouse scroll wheel) to change the number of iterations.

1 Grammar expansion

Our first step is to produce the sequence of symbols that will be used for drawing later. We start with the initial sequence, in our example the symbol X. Then in each iteration, we expand the symbols according to the rules: each X is replaced with $F[+X]F[-X]+X$ and each F is replaced with FF. We denote the sequence after n-th iteration as s_n, here is the first iteration:

$$\begin{aligned} s_0 &= X \\ s_1 &= F[+X]F[-X]+X \end{aligned}$$

We continue to the second iteration:

$$\begin{aligned} s_1 &= F[+X]F[-X]+X \\ s_2 &= FF[+F[+X]F[-X]+X]FF[-F[+X]F[-X]+X]+F[+X]F[-X]+X \end{aligned}$$

We repeat this procedure until we find s_n where n is the number of iterations given by the system specification.

Task 1.1 Implement the function `LindenmayerSystemDeterministic::expandSymbol` which takes a single symbol (eg. F) and returns the corresponding expansion (eg. FF).

Task 1.2 Implement the function `LindenmayerSystem::expandOnce` which takes s_m and returns s_{m+1}.

Task 1.3 Implement the function `LindenmayerSystem::expand` which takes s_0 and n, returns s_n.

Required output: Produce the output `.txt` files with the final symbol sequences. The given code takes care of saving the files, you just need to put the sequence in the `sym_sequence_expanded` variable.

2 Drawing

Now we use the produced symbol sequence to draw the representation of the system. During the drawing process, we need to remember the position p of our pen and the forward direction f. We start with $p = (0, 0)^\intercal$ and the direction f pointing "up" along the y axis. We look at the symbols in order, performing the actions corresponding to each symbol:

- `+` rotates the direction f by the angle δ counter-clockwise
- `-` rotates the direction f by the angle δ clockwise
- `[` saves the current state (p, f) on a stack (starts a branch)
- `]` restores the most-recently-saved state (ends the branch)
- `F` **or any other symbol not listed above** moves the pen forward (along the f direction) and draws a line segment **of length 1**

Task 2.1 Implement the function `LindenmayerSystem::draw`, which takes the sequence of symbols and returns an array of drawn line segments.

Required output: Produce the output image files for the deterministic systems. The given code takes care of drawing and saving the images, you just need to put the segments in the `lines` variable. Execute the `run_deterministic.sh` script to draw all systems with your program.

3 Understanding the expansion rules

(a) $F \to FF$ (b) $X \to F[+X]F[-X] + X$

Figure 2: Visual explanation of the expansion rules.

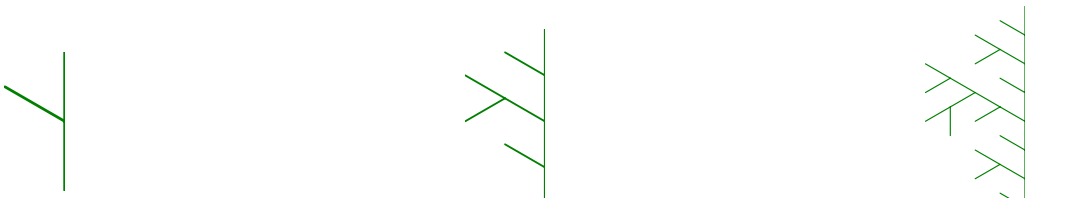
Lindenmayer systems embody the idea that low-level, local behaviours give rise to a complex macro system. Likewise, biological organisms are a combination of cells, each performing its small function. In L-systems, each expansion rule represents such a low-level behaviour; let us analyze the rules of the example system:

- $F \to FF$ means that F is a stem which grows longer on each iteration - see Figure 2(a)
- $X \to F[+X]F[-X] + X$ means X splits into 3 branches as it grows - see Figure 2(b)

In this task we will reverse engineer the system rules from images. In the directory `reverse_in` you will find definitions of systems but without the rules. The interactive mode of the program may be useful for testing your designs.

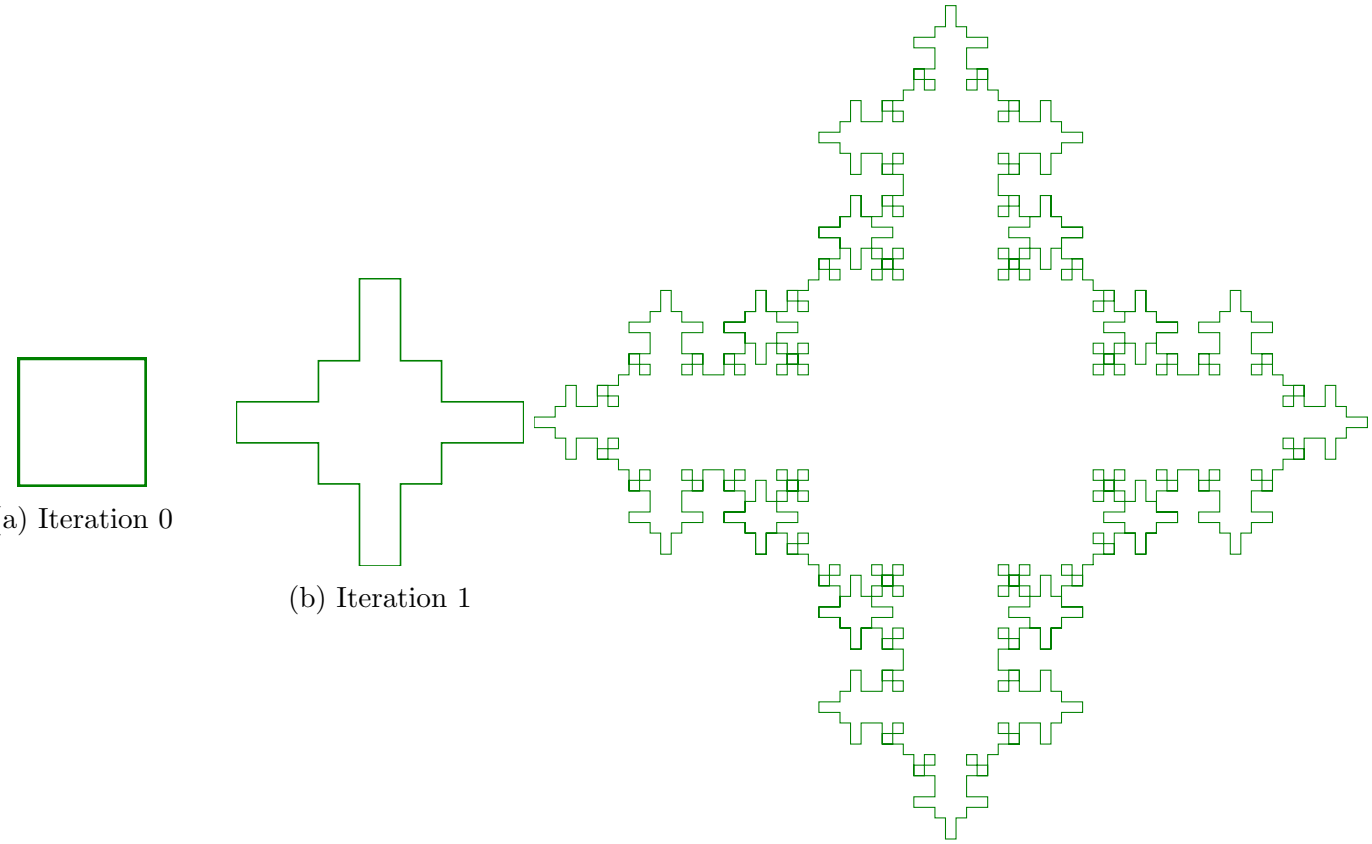
Task 3.1 Determine the rules based on the example images below and write them in the files in `reverse_in`. The systems are shown in Figures 3, 4, 5, 6.

Required output: System definitions in `reverse_in` and their corresponding images and sequences in `reverse_out`. Execute the `run_reverse.sh` script to draw all systems with your program.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

Figure 3: The **left_plant** system to reverse engineer.
Iterations $n = 7$
Rotation angle $\delta = 60°$
Initial state: $s_0 = F$
Rules: ?

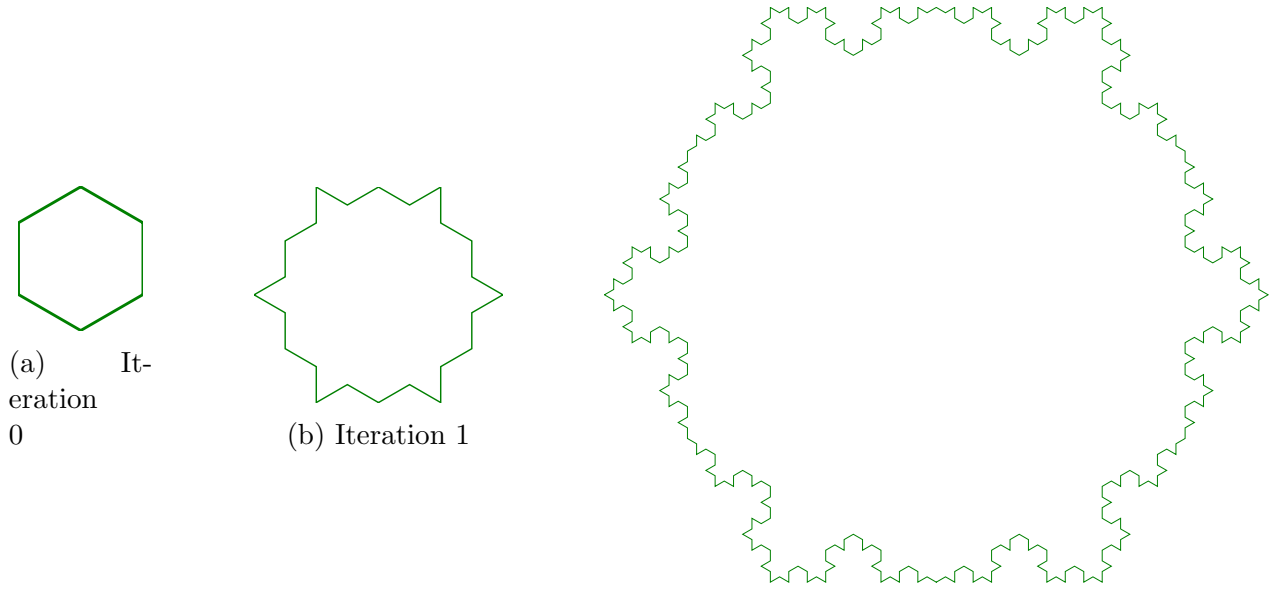


(a) Iteration 0

(b) Iteration 1

(c) Iteration 3

Figure 4: The **crossout** system to reverse engineer.
Iterations $n = 4$
Rotation angle $\delta = 90°$
Initial state: $s_0 = F + F + F + F$
Rules: ?

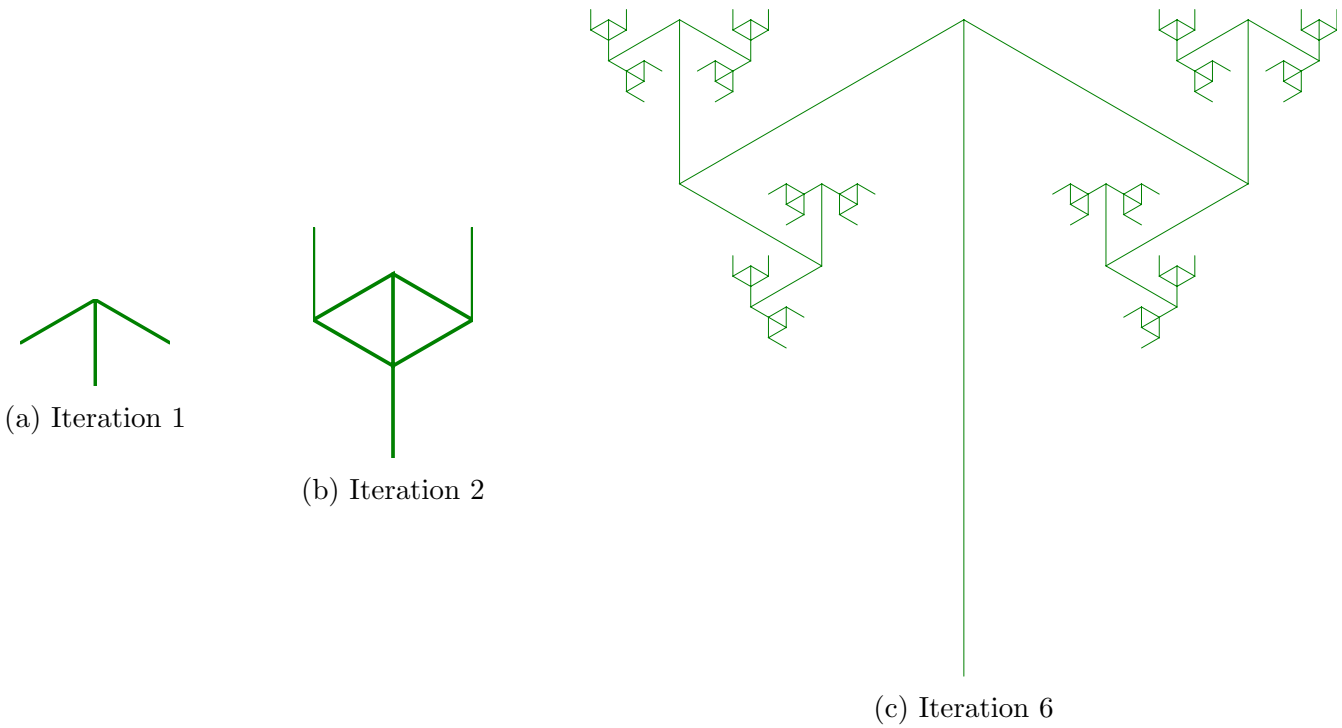


(a) Iteration 0

(b) Iteration 1

(c) Iteration 3

Figure 5: The **hexerode** system to reverse engineer.
Iterations $n = 4$
Rotation angle $\delta = 60°$
Initial state: $s_0 = -F + F + F + F + F + F$
Rules: ?



(a) Iteration 1

(b) Iteration 2

(c) Iteration 6

Figure 6: The **flower_plant** system to reverse engineer.
Iterations $n = 7$
Rotation angle $\delta = 120°$
Initial state: $s_0 = T$
Rules: ?

4 Stochastic systems

Biological processes often behave with a degree of randomness and to account for that, we extend our systems with stochastic rules. The following rule, used in the system in Figure 7:

$$F \xrightarrow{p=0.5} F + F$$
$$F \xrightarrow{p=0.5} F - F$$

means that the segment F is replaced with 50% probability with $F+F$ (left turn) and with 50% probability with $F-F$ (left turn).

Task 4.1 Implement the function `LindenmayerSystemStochastic::expandSymbol` which does symbol expansion in the stochastic case. You can use the `Dice` class to generate random values easily.

Required output: Produce the output images and sequence files for the stochastic systems. The script `run_stochastic.sh` will run your program for all stochastic systems.

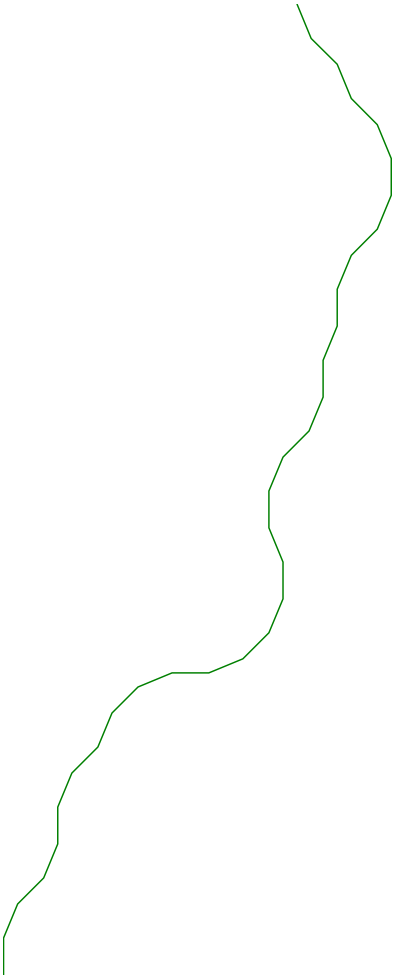


Figure 7: Stochastic path
Iterations $n = 5$
Rotation angle $\delta = 22.5°$
Initial state: $s_0 = F$
Rules:
$F \xrightarrow{p=0.5} F + F$
$F \xrightarrow{p=0.5} F - F$