CS 211
Project 9 – L-Systems

1. Introduction

This project explores the fascinating connection between mathematics and the natural world, specifically focusing on the production of graphical representations of branching structures of plants. We will explore the concepts proposed by the Hungarian botanist Aristid Lindenmayer. This project, based on his book The Algorithmic Beauty of Plants [1], highlights two key aspects:

1. **Developmental Algorithms:** The elegance and relative simplicity of the mathematical rules that govern plant growth and form.
2. **Self-Similarity:** The phenomenon where a part of a plant structure resembles the whole plant at an earlier stage. This property creates a sense of repetition and fractal-like patterns in plant growth.

L-systems (Lindenmayer systems) are a mathematical formalism for describing plant development. They use rewriting rules to iteratively transform strings of symbols representing plant structures into more complex forms. This approach allows for the simulation of various plant growth patterns on computers.
The outcome of the successive (iterative) application of formal grammar rules to the axiom (initial string) produces a sequence of commands amenable to be executed using Python's turtle module. That execution displays fractal structures, some of which correspond to plants.
Adding a state stack increases L-Systems' generation capability to produce branching structures. An L-System may include multiple grammar rules that apply to the same symbol, each with an associated probability of being selected randomly. The new model produces stochastic structures that are similar in nature but different in form (using the same grammar to produce them.)
In this project, you will create three classes, `Stack`, `State`, and `LSystem`, in the file `l_system.py`.

2. L-Systems

The central concept of L-systems is that of rewriting. In general, rewriting is a technique for defining complex objects by successively replacing parts of a simple initial object using a set of rewriting rules or productions. The classic example of a graphical object defined in terms of rewriting rules is the snowflake curve, studied last term in CS 210. See Fig. 1.

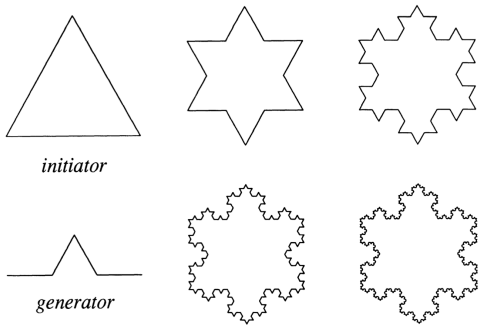

Fig. 1. The Snowflake Fractal

In 1968, a biologist, Aristid Lindenmayer, introduced a new type of string-rewriting mechanism, subsequently termed L-systems [82]. The essential difference between Chomsky's grammars and L-systems lies in how they apply production rules. Chomsky's grammars apply productions sequentially, whereas L-systems do it in parallel and simultaneously replace all letters in a given word. This difference reflects the biological motivation of L-systems. Production rules capture the dynamics of cell divisions in multicellular organisms, where many divisions may occur simultaneously.

Consider strings (words) built of two letters, a and b, which may occur many times in a string. Each letter is associated with a rewriting rule. The rule a → ab means that the letter a is to be replaced by the string ab, and the rule b → a means that the letter b is to be replaced by a. The rewriting process starts from a distinguished string called the axiom. Assume that it consists of a single letter b. In the first derivation step (the first step of rewriting), the axiom b is replaced by a, using production b → a. In the second step, a is replaced by ab using production a → ab. The word ab consists of two letters, both of which are simultaneously replaced in the next derivation step. Thus, a is replaced by ab, b by a, and the string aba results. Similarly, the string aba yields abaab, which in turn yields abaababa, then abaababaabaab, and so on (Figure 2).

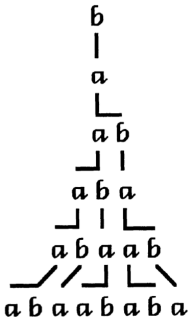

Fig. 2. Derivation Example from an L-System Grammar

L-Systems consist of a grammar (w, r), which contains a starting symbol or expression called the axiom and a set of term rewriting rules (like those included in the previous example). The axiom is a string of arbitrary length, and each rule has the form a → b, where a is a single character and b is a string of arbitrary length. You will find a formal definition of L-Systems in [1].

3. The Turtle

The basic idea of turtle interpretation is the following. The turtle's state is a triplet (x, y, α), where the Cartesian coordinates (x, y) represent the turtle's position, and the angle α, called the heading, is the direction the turtle faces. Given the step size d and the angle increment δ, the turtle can respond to commands represented by the following symbols (Figure 3(a)):

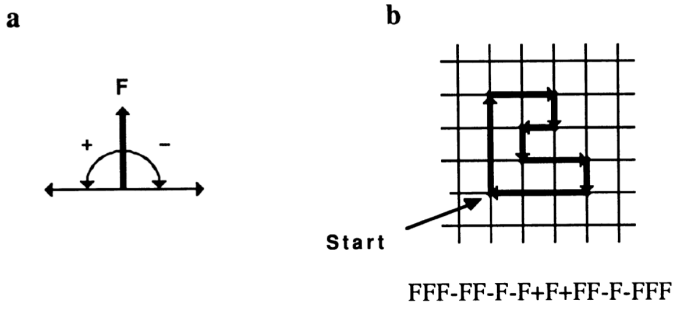

Figure 3. (a) Turtle interpretation of string symbols *F*, **+**, **-**. (b) Interpretation of a string. The angle increment is equal to 90°. Initially, the turtle faces up.

F Move forward a step of length *d*. The state of the turtle changes to *(x', y', a),* where $x' = x + d \cos \alpha$ and $y' = y + d \sin \alpha$. The turtle draws a line segment between points *(x, y)* and *(x', y')*.

f Move forward a step of length d without drawing a line.

+ Turn left by angle δ. The turtle's next state is (x, y, $\alpha + \delta$). The positive orientation of angles is counterclockwise.

- Turn right by angle δ. The next state of the turtle is (x, y, $\alpha - \delta$).

Given a string *v,* the initial state of the turtle (x_0, y_0, α_0) and fixed parameters *d* and δ, the *turtle interpretation* of *v* is the figure (set of lines) drawn by the turtle in response to the string *v* (Figure 3 (b)).

This method can be applied to interpret strings generated by L-systems. For example, Figure 4 presents four approximations of the *quadratic Koch island*. These figures were obtained by interpreting strings generated by the following L-system:

```
w: F-F-F-F
r: F→F-F+F+FF-F-F+F
```

The images correspond to the strings obtained in derivations of length 0 to 3. The angle increment δ is equal to 90°. The step length *d* is decreased four times between subsequent images, making the distance

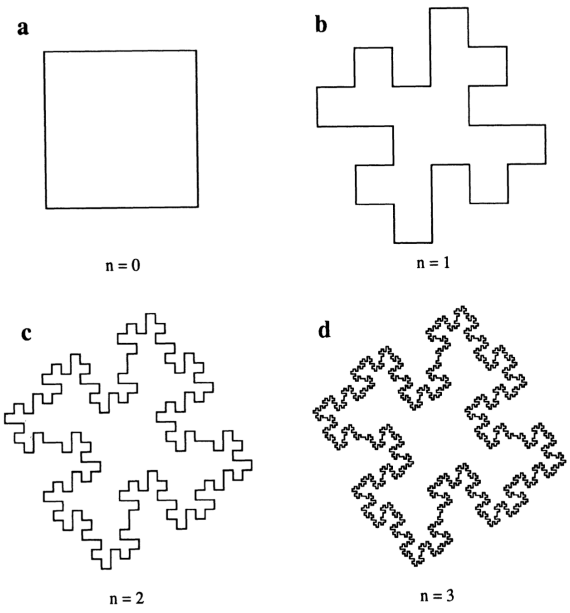

Fig. 4. The Koch Island, generated by an L-System. n is the number of iterations – successive applications of the grammar rules to the axiom.

4. Implementation

The project's main component is a class that defines an L-System. An object of this class contains an axiom, a set of rewriting rules, an angle, the step size, and the order of the substitution. Additionally, it keeps information about the turtle that will draw its graphical expression. That information includes the starting position, starting angle, and color.
You are responsible for providing all the implementation related to the LSystem class.

4.1. The constructor

The constructor accepts that information with default values for some of the object's attributes. You are responsible for providing the **`LSystem`** constructor.

```
class LSystem:
    """
    Represents an L-system with its axiom, rules, rewriting,
    and drawing methods.
    """
def __init__(self, axiom, rules, angle, step, n=3,
             starting_pos=(-200, 0), starting_angle=0, color="blue"):
      pass
```

As an example of use, the following code fragment shows how to build an LSystem object.

```
ls1 = LSystem(axiom="-L",
    rules={"L": "LF+RFR+FL-F-LFLFL-FRFR+", "R": "-LFLF+RFRFR+F+RF-LFL-FR"},
    angle=25.7, step=10,
)
```

Note that a dictionary stores the rewriting rules, where the key of an attribute-value pair is the symbol to be substituted, and its corresponding value is the substitution string.

4.2. The iterator

An LSystem already contains the substitution order (`n`), i.e., how many times we will scan the current string performing substitutions, so this method does not need any parameters.

```
def iterate(self):
    pass
```

Iterate considers the axiom the first version of the `current_string`. The method scans the current string, one character at a time, assembling a `new_string` by applying the substitution rule whose key is that character. If there are no substitution rules for that character, the character is transferred to the `new_string` intact. This method modifies the `LSystem` object, assigning the `new_string` to an attribute called `commands`. The method `iterate` returns `None`.

4.3. Drawing the commands

Once the method `iterate` has produced the `commands` attribute, the `draw` method uses the `turtle` module to render a graphical expression of the fractal object produced by iterating the `LSystem` object.

First, this method sets the `turtle` to its initial position, angle, and color. It then traverses the command string, analyzing each character and determining the action it represents—it instructs the turtle to perform that action. If a character does not represent any turtle command, it is ignored. All the information this method requires is already in the `LSystem` object. Hint: make the turtle wait for a click at the end so the turtle window does not disappear.

```
def draw(self):
    pass
```

4.1. The plot Method

The method plot just iterates and draws-i.e., invokes those methods.

```
def plot(self):
    pass
```

5. Branching Structures

One prominent feature of a fractal is self-similarity (present in the previous examples). Some branching structures in nature exhibit self-similarity, i.e., they are formed by a stem with branches spawned in all directions (in a 2D structure, left and right). Those branches may exhibit the same structure as the whole structure (yielding recursive, self-similar structures) or represent ending segments. Figure 5 illustrates this structure.

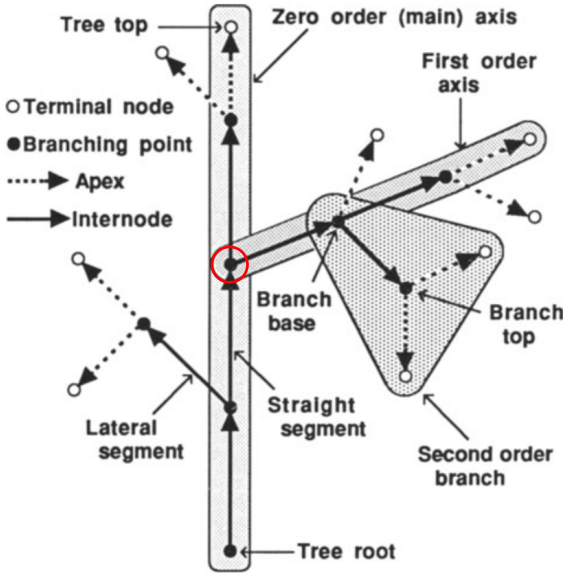

Fig. 5. Self-similar structures

It would be convenient to have a mechanism that allowed us to draw the stem of one such structure, deviate momentarily to produce a left (or right) branch, and then continue where it left off.

5.1. The Turtle State

Let us say the turtle is executing the `commands` derived by the method `iterate` and finds itself at the point indicated by the red circle in Fig. 5. While executing the `commands` string, the `turtle` will stop there to draw the branch that extends to the right. It would be convenient to store the turtle's state to come back to it later. In drawing that branch, the turtle will end up who knows where, and it needs to come back to the state it was before.

The turtle state contains its position (`x`, `y`) and orientation (`heading`). Its constructor has the form

```
class State:
    def __init__(self, x=0, y=0, angle=0):
        pass
    def __str__(self):
        pass
    def __repr__(self):
        pass
    def set_state(self, t):
        pass
```

You are responsible for creating the class state and its methods (indicated above).

5.2. Stack

Fig. 5 shows a red circle where the turtle will start drawing the right branch. When the **turtle** follows that branch, it encounters the point indicated as "branch base", where it starts drawing a sub-branch. The **draw** method must save the **turtle**'s **State** at the red circle to restore it after drawing the branch. The **draw** method must save this second **State** at the branch base to restore it when it finishes drawing the sub-branch. Note that the second **State** must be restored before the first one. This scenario calls for a **Stack** data structure characterized by the property "Last-In, First-Out".

Implement a **Stack class** with the following methods:

```
class Stack:
    def __init__(self):
        pass
    def push(self, item):
        pass
    def pop(self):
        pass
    def is_empty(self):
        pass
```

You will develop the **State** and **Stack** classes in Lab 9, implemented in the file **state.py**.

5.3. Bracketed L-Systems

Iterating over the axiom of an **LSystem** produces a sequence of turtle commands. Those grammars can produce branching structures by extending the set of commands to include *strings with brackets*. We introduce two new symbols to delimit a branch. They are interpreted by the turtle as follows:

[**push** the turtle's current state onto a **Stack**. The information saved on the **Stack** contains the **turtle**'s position and orientation.

] **pop** a state from the **Stack** and make it the current **State** of the **turtle**. The turtle draws no line, although its position changes.

Using a **Stack** of states, as indicated in Section 5.2, you must modify the **LSystem**'s draw method to account for the commands "**[**" and "**]**".

Fig. 6 illustrates an example of a branching structure generated by an L-System.

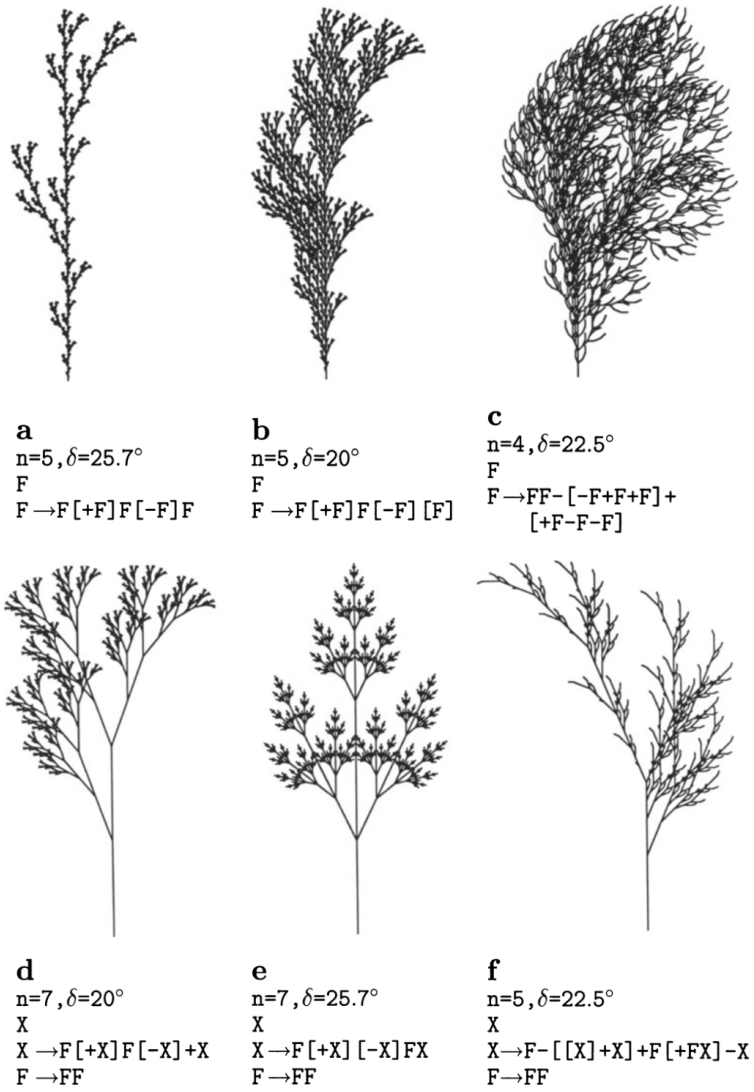

Fig. 6. Examples of plant-like structures generated by bracketed OL- systems.

LSystem's method **draw** executes the commands produced by term rewriting (i.e., iterating substitution from the axiom using the grammar rules). You must modify this method to include the execution of the bracket commands (**[** and **]**). You will use this week's lab's products to accomplish this task.

6. Extra Credit – Non-deterministic Structures

If we were to fill a landscape with structures generated with the system developed so far, it would make a very boring scene. We need to add variety to the landscape. We can achieve this effect using non-deterministic L-Systems.

A non-deterministic or *stochastic L-system* includes several possible substitutions for each symbol. What substitution we use is a stochastic choice. A substitution rule was of the form `{X: st}`, whose semantics were: substitute the symbol `X` by the string `st`. Each substitution is no longer a simple string, but a couple `(p, st)`, where p indicates the probability to use the string `st` to substitute the symbol `X`.

The following code snippet provides an example of use of this mechanism.

```
nd_ls_1 = LSystem(axiom="F",
    rules={"F": [(.33, "F[+F]F[-F]F"), (.33, "F[+F]F"), (.33, "F[-F]F")]},
    angle=25.7, step=10
)
```

Each substitution rule will be used with the same probability (1/3). Figure 7 is an example of the final product of this system applied several times.

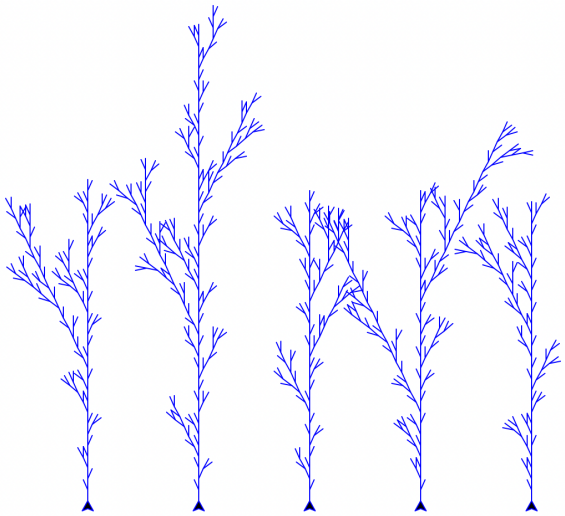

Fig. 7. Stochastic branching structures produced by L-Systems

Design and implement the changes necessary to produce such a rule-rewriting system based on your solution to the deterministic one. Upload your code in the file `nd_l_system.py` and a screenshot of the images produced by your solution using the example in `nd_ls_1`.

Do not forget to seed the random number generator (use 42 as the seed).

7. References

[1] Prusinkiewicz, Przemyslaw, and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

8. Rubric

The following list indicates the required components and their associated mark points.

- [10 pts] The `LSystem` class constructor
- [15 pts] The `iterate` method
- [15 pts] The `iterate` method for branching structures
- [20 pts] The `draw` method
- [20 pts] The `plot` method
- [20 pts] Extra credit (optional) – `nd_l_system.py`, and `ss.png`.

Upload to Coding Rooms the files:

- `state.py`, `l_system.py`, `nd_l_system.py`, and `ss.png`.

9. Do not Risk Cheating

The academic integrity rules for CS 211 are very strict, and we take them very seriously. Suspected violations are reported to the appropriate UO office, and the penalty for plagiarism includes failing the course. If you consulted with anyone other than the instructional staff of CS 211 (the instructor, graduate teaching assistants, and undergraduate learning assistants) to complete the program, you must explicitly credit them in the docstring comment. If you referred to any example code aside from the provided documentation and the Python library documentation, you must explicitly cite it. I hope to see many credits in the projects you turn in because you learn more and better when you collaborate in teaching and learning from each other.

This term, you can work in teams of two people – no more than two. Examples of dishonest behavior are:

- Sharing code between teams
- Forming teams of 3 or more people
- Getting code from online sources, e.g., GitHub, YouTube, chatGPT, StackOverflow, etc., and presenting it as your own

You are encouraged to perform internet searches and take one or two lines of code from the places that appear on the search results. Also, see the syllabus for using AI in the solution of your projects.

I may modify this project as a result of your observations.