

# L-systems

A final project for Foundations of Computer Science (Fall 2023) at Olin College of Engineering

Daniel Sudzilowski and Sparsh Gupta

Implementing the L-system rewrite: <https://en.wikipedia.org/wiki/L-system>.

## What are L-systems?

L-systems (also known as Lindenmayer systems) are a type of formal grammar that is capable of developing various biological/nature patterns by using iterations of simple production rules.

In an L-system, a set of rules describe how each variable should be replaced to generate a grammar over multiple iterations.

L-systems are often used to model and visualize interesting structures such as plants trees, and mathematical graphics.

## Axiom and Rules

The generation of grammar for an L-system starts from a specific start axiom (a symbol or initial string of symbols) and then production rules determine how to generate the grammar over multiple iterations by replacing these symbols.

## Alphabet

The set of symbols that are used in an L-system is known as their alphabet, and these symbols could represent instructions like branching, rotating, popping or pushing a stack, etc.

## L-systems vs Grammars

### Context-free case

At each iteration, an L-system will apply each rule as many times as possible. A traditional grammar, on the other hand, will apply one rule at a time at each iteration. This can make a difference in practice for example:

### Traditional Grammar Example

Start: S

Terminals: {S}

Non-Terminals: {}

Rules:  $\{S \rightarrow SS\}$

Language: {S, SS, SSS, SSS...}

### **L-system Grammar Example**

Start: S

Terminals: {S}

Non-Terminals: {}

Rules: {S  $\rightarrow$  SS}

Language: {S, SS, SSSS, SSSSSSSS}

### **Stochastic Grammar L-systems**

Stochastic Grammar involves introducing randomness into the production rules of L-systems. Deterministic L-systems have specific rules associated to it to have a unique replacement for every symbol, and stochastic L-systems add probabilities to these rules of replacement causing variability and randomness in the generation of grammar.

### **Examples of L-systems**

#### **Fractal Plant**

The Fractal Plant is a mathematical model of plant growth that produces intricate and self-replicating patterns. It simulates the branching structures seen in nature by iteratively applying rules to symbols, resulting in the creation of complex and visually appealing plant-like forms.

#### **Instruction set** [wiki]

variables : X F

constants : + - [ ]

start : X

rules : (X  $\rightarrow$  F+[[X]-X]-F[-FX]+X), (F  $\rightarrow$  FF)

angle : 25°

We initialize an empty stack first. Here, F means “draw forward”, - means “turn right 25°”, and + means “turn left 25°”. X does not correspond to any drawing action and is used to control the evolution of the curve. The square bracket “[” corresponds to saving the current values for position and angle, and we push it to the top of the stack, and when the “]” token is encountered, we pop the stack and reset the position and angle. Every “[” comes before every “]” token.

#### **Stochastic Fractal Plant**

A stochastic fractal plant is a generated using stochastic (random) processes and fractal geometry. It incorporates randomness to simulate the natural variability

found in plants, creating realistic and diverse virtual plant structures.

**Instruction set** [ref - Page 28 (Section 1.7)]

variables : F

constants : + - [ ]

start : F

rules : F -P(0.33)-> F[+F]F[-F]F, F -P(0.33)-> F[+F]F, F -P(0.34)-> F[-F]F

angle : 25°

Here, F means “draw forward”, - means “turn right 25°”, and + means “turn left 25°”. The square bracket “[” corresponds to saving the current values for position and angle, and we push it to the top of the stack, and when the “]” token is encountered, we pop the stack and reset the position and angle. Every “[” comes before every “]” token.

### Koch Curve

The Koch Curve is a mathematical fractal curve that exhibits self-similarity, meaning it retains a similar pattern at different scales. It is constructed by repeatedly replacing each straight line segment with a smaller equilateral triangle, creating a progressively more detailed and complex geometric shape.

**Instruction set** [wiki]

variables : F

constants : + -

start : F

rules : (F → F+F-F-F+F)

Here, F means “draw forward”, + means “turn left 90°”, and - means “turn right 90°”

### Sierpinski Triangle

The Sierpinski Triangle is a geometric fractal that results from recursively removing triangles from an equilateral triangle. Starting with an initial triangle, smaller triangles are successively removed from its center, creating a self-replicating pattern of triangles within triangles.

**Instruction set** [wiki]

variables : F G

constants : + -

start : F-F-F

rules : (F  $\rightarrow$  F-G+F+G-F), (G  $\rightarrow$  GG)

angle : 120°

Here, F means “draw forward”, G means “move forward”, + means “turn left by angle”, and - means “turn right by angle”.

### Dragon Curve

The Dragon Curve is a self-replicating geometric pattern generated by iteratively folding a strip of paper. It is a space-filling curve that exhibits fractal-like properties, forming a complex, dragon-like shape through a sequence of simple folding steps.

**Instruction set** [wiki]

variables : F G

constants : + -

start : F

rules : (F  $\rightarrow$  F+G), (G  $\rightarrow$  F-G)

angle : 90°

Here, F and G both mean “draw forward”, + means “turn left by angle”, and - means “turn right by angle”.

### Computational Setup

We use `python3` (version 3.11) and `tkinter` library for our codebase.

If you’re on a windows/Intel-based Mac machine, you can download the `tkinter` library using the following `pip` command in the command prompt:

```
pip install tk
```

If you’re using an M-series Mac, then you can obtain it using `homebrew` (check this out: <https://github.com/daniel-sudz/focs-lsystems/blob/main/bin/env-macos>)

### Codebase - Computational Description

Let us look deeper into the `lsystem.py` that defines our primary L-system functionality.

We define a class to encapsulate the concept of a production rule in L-systems. Production rules dictate how symbols are rewritten based on their context.

```
class ProductionRule:
    def __init__(self, rewrite_from: str, context_left:
        ↪ Optional[str], context_right: Optional[str]):
```

```

self.rewrite_from = rewrite_from
self.context_left = context_left
self.context_right = context_right

```

Next, we define the class `LSystem` that encapsulates the parameters and generates the L-system visualizations using Turtle graphics.

```

class LSystem:
    def __init__(
        self,
        start: str,
        rules: Dict[str, ProductionRule],
        iterations: int,
        visualizations: Optional[Dict[str,
↳ Callable[[turtle.Turtle], None]]] = None,
        render_start_pos: tuple[int, int] = (0, 0),
        render_heading: int = 0,
        debug: bool = True
    ):

```

Then, we can initialize the Turtle graphics setup by creating the turtle and setting up parameters like turtle speed, delay, start position, start heading, etc.

```

if self.visualizations:
    # ... (continued code)

```

The `visualize` method recursively applies the production rules in each iteration and also visualizes each iteration using the above Turtle graphics setup.

```

def visualize(self, cur_string: str = None, iteration: int = 0):
    # ... (continued code)

```

The code for implementing stochastic `stochastic_lsystem.py` is pretty similar including instantiating the class and the visualization method, however, it introduces one new function to choose a rule randomly and also has another method to apply stochastic rules.

```

def choose_random_rule(rules: List[Tuple[str, float]]) -> str:
    total_prob = sum(prob for _, prob in rules)
    rand_num = random.uniform(0, total_prob)
    cumulative_prob = 0

    for rule, prob in rules:
        cumulative_prob += prob
        if rand_num <= cumulative_prob:
            return rule

    # This should not happen, but in case of rounding errors
    return rules[-1][0]

```

The above function basically selects a rule based on a predefined probability that the rule has. It first calculates the cumulative probabilities, generates a random number within that range, and returns the corresponding rule.

To supplement this function for stochastic grammar generation, we introduce another method in the `StochasticLSystem` class. The below method takes use of the above function to choose a random rule after it extracts a list of rules for each character, and return a new set of characters based on the randomness.

```
def apply_stochastic_rules(self, cur_string: str) -> str:
    new_string = ""
    for char in cur_string:
        rule = self.rules.get(char, [(char, 1.0)])
        new_string += choose_random_rule(rule)

    return new_string
```

Finally, in the `examples` directory, we instantiate the desired L-system and then input the desired parameters to generate and visualize the desired L-system example in the following format:

```
# Example L-system instantiation
lssystem = LSystem(
    start="A",
    rules=
    {
        "A": ProductionRule("AB", None, None),
        "B": ProductionRule("A", None, None)
    },
    iterations=4,
    visualizations=
    {
        "A": lambda t: t.forward(10),
        "B": lambda t: t.left(90),
    },
    render_start_pos=(0, 0),
    render_heading=0,
    debug=True
)

# Visualize L-system evolution
lssystem.visualize()
```