HAW HAMBURG

BACHELORTHESIS
Anatolie Sirbu

Responsive Natural Patterns: A Computational Approach with Fractals and L-Systems

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Anatolie Sirbu

Responsive Natural Patterns: A Computational Approach with Fractals and L-Systems

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Björn Gottfried
Second examiner: Prof. Dr. Martin Lapke

Day of delivery: 08. November 2024

Anatolie Sirbu

Title of Thesis

Responsive Natural Patterns: A Computational Approach with Fractals and L-Systems

Keywords

Fractal, L-System, Axiom, Formal Grammar, Alphabet, Rewrite Rules

Abstract

The following thesis explores the synthesis of computational models with natural patterns, using fractals and L-systems to create graphic representations of these models and structures like tree and plants. The core of this work is the development of such a system that would, in future, integrate real-time sensor data leading to the adjustment of the parameters of these model's growth angles and branching patterns.

The system is able to create an ever-changing set of dynamic visualizations in response to changes within the environment through modification of the fractal and L-system rules based on sensor input. It would allow possible applications on sensor data to be made in both data visualization and interactive art for more engagement and interpretation.

Anatolie Sirbu

Thema der Arbeit

Reaktive Naturmuster: Ein rechnergestützter Ansatz mit Fraktalen und L-Systemen.

Stichworte

Fraktal, L-System, Axiom, Formale Grammatik, Alphabet, Umschreibregeln

Kurzzusammenfassung

Die vorliegende Arbeit untersucht die Synthese von Rechenmodellen mit natürlichen Mustern und verwendet Fraktale und L-Systeme, um grafische Darstellungen dieser Modelle und Strukturen wie Bäume und Pflanzen zu erstellen. Der Kern dieser Arbeit ist die Entwicklung eines solchen Systems, das in Zukunft Echtzeit-Sensordaten integrieren könnte, was zur Anpassung der Parameter dieser Modelle Wachstumswinkel und Verzweigungsmuster – führen würde.
Das System ist in der Lage, eine ständig wechselnde Menge von dynamischen Visualisierungen als Reaktion auf Veränderungen in der Umgebung durch Modifikation der Fraktal und L-Systemregeln basierend auf Sensoreingaben zu erstellen. Dies würde mögliche Anwendungen von Sensordaten sowohl in der Datenvisualisierung als auch in der interaktiven Kunst für mehr Engagement und Interpretation ermöglichen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the rapidly evolving field of computational modeling, Lindenmayer Systems (L-Systems) have established their value through extensive applications in biology, graphics, and beyond. Originally designed to model plant growth, L-systems operate on fixed rules that form the backbone of their structure[3]. However, these systems have not yet been fully adapted to respond to the continuous influx of real-time data, a capability increasingly vital in today's data-driven environments.

This thesis proposes to advance L-systems by making them dynamic—capable of adjusting their rules in response to new information. This innovation would significantly enhance the flexibility and applicability of L-systems, enabling them to model complex, changing phenomena more accurately. The goal is to transform L-systems from static to adaptive systems, which can update themselves automatically as new data becomes available.

Implementing responsiveness in L-systems means embedding mechanisms that allow the system to detect relevant environmental or contextual changes and to alter its generative rules. This could involve integrating sensors or data feeds that provide continuous updates on external conditions, which the system could then interpret and use to adjust its parameters.

The need for such dynamic systems is clear, as they would provide a more realistic simulation of natural processes and other phenomena that are subject to change. For example, dynamically adaptive L-systems could better simulate environmental changes or growth patterns in plants that respond to varying conditions, offering more precise tools for researchers and professionals in many fields.

2 State of Art

2.1 Overview

With the recent technological advancements, we have been offered the opportunity to accurately translate mathematical frameworks into programming language and thus devise a symbiotic way of developing and analyzing various complex models.
Two of those mathematical frameworks that are of interest with regards to the goal of this thesis are: L-Systems, introduced in 1968 by Aristid Lindenmayer, a Hungarian theoretical biologist and botanist at the University of Utrecht, and fractal geometry, introduced in 1982 by Benoit Mandelbrot in his book "The Fractal Geometry of Nature".

With this in mind, the ability to combine L-systems and fractals with the power of a modern CPU or GPU has lead to fascinating discoveries. For instance, the frameworks describing the "growth" of mountains and trees has made it possible to visually recreate natural patterns that can accurately resemble the real-world. An illustrative example would be the fractal terrain generation, used in the movie Star Trek II the Wrath of Khan (1982).

Figure 2.1: The "Genesis Effect" for Star Trek II.[5]

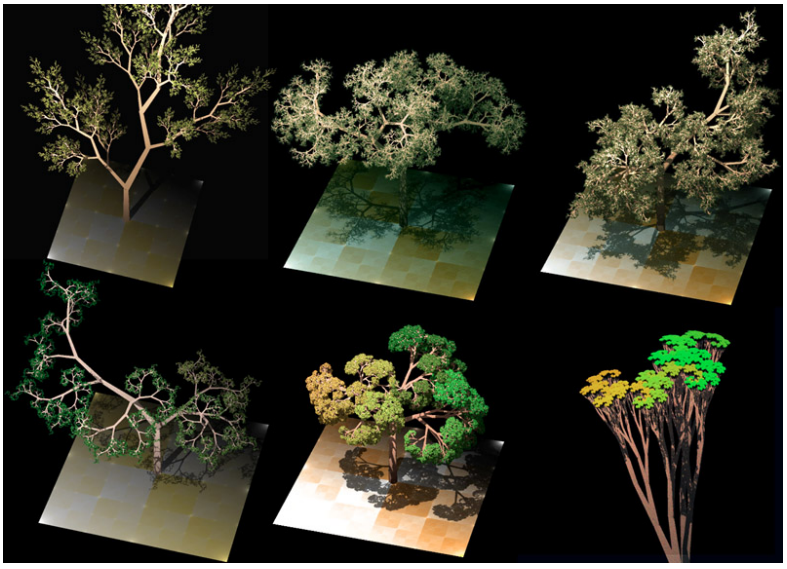


Figure 2.2: Images of trees generated from using an L-system.[31]

This thesis will primarily focus on L-Systems, hence analysing the possibility expanding and modifying its formal grammar in order to achieve a system that would help dynamically visualize external inputs.

2.2 L-Systems

2.2.1 Origin and Application

In 1968, biologist Aristid Lindenmayer developed Lindenmayer systems, commonly known as L-systems, to formalize the patterns observed in bacterial growth. These systems utilize a recursive string-rewriting mechanism and are widely employed in computer graphics today for visualizing and simulating organic growth. L-systems have applications across various fields, including plant development, procedural content generation, and the creation of fractal-like art.[13]
Over time, L-systems have been adapted for use in a wide range of diverse fields. For instance, they have been employed to create rivers in fractal mountains, lay out streets in virtual cities, and describe the subdivision of curves. Beyond computer graphics, L-systems have also found applications in music generation. They continue to be a popular tool in plant modeling, with models generated using L-systems featuring in modern video games and films.[6]

2.2.2 Formal definition

An L-system, denoted as L, is defined as a triplet $L = (\Sigma, \omega, R)$, where:

- Σ is an **alphabet**, a non-empty set of symbols. The set Σ^* represents all possible strings (words) that can be formulated from Σ, and Σ^+ denotes the set of all non-empty strings.[3]
- $\omega \in \Sigma^+$ serves as the **axiom** or the initial state of the L-system. This axiom defines the starting point of the system.
- $R \subseteq \Sigma^* \times \Sigma^*$ is a collection of **rewrite rules** or production rules. A rule $(s \to v) \in R$ specifies how a substring s in any string from Σ^* is replaced by v.[11]

If a symbol $s \in \Sigma$ does not appear on the left-hand side of any rewrite rule in R, the system assumes the identity rewrite rule $s \to s$. Symbols that only appear in identity rules are referred to as **constants** or **terminals**.

The formulation of an L-system resembles that of a deterministic context-free grammar, with several key distinctions:

- Unlike typical grammars, L-systems do not explicitly define terminal and non-terminal symbols, except through the identity rewrite rules applied to terminal symbols.
- The initial string in L-systems can be a non-empty word, unlike some grammars that might allow a non-terminal initial state.
- The primary distinction lies in the rewriting mechanism, which operates under different principles outlined in subsequent sections.

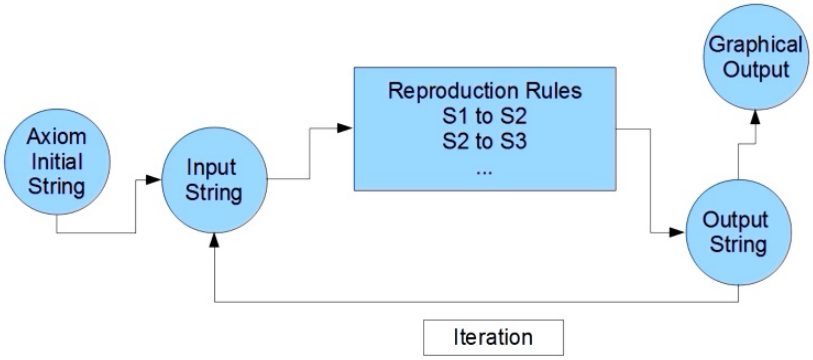


Figure 2.3: Simplified Version of the System where S=String.[31]

2.3 Principles of Rewriting

The theoretical principles of L-systems are rooted in their ability to model biological growth processes through deterministic and parallel rewriting rules.[3]

Deterministic Rewriting: Each symbol in an L-system string is rewritten according to specific, predefined rules. This deterministic approach ensures that the rewriting process is entirely predictable, based entirely on the initial axiom. There is no ambiguity in the transformation of symbols, making the evolution of the string fully reproducible and dependent only on its initial configuration.

Parallel Rewriting: Unlike sequential rewriting systems where transformations occur one after another, L-systems employ parallel rewriting. This means that all symbols in the string are simultaneously rewritten in each iteration. Such a method is vital for simulating natural growth, where various parts of a biological entity grow concurrently.[10]

In implementation, this simultaneous rewriting ensures that the new characters resulting from one symbol's transformation do not influence the rewriting of another symbol within the same iteration.[22]

This parallel processing distinguishes L-systems from formal grammars used in computational linguistics, where not all symbols necessarily need to be rewritten, and multiple derivations can result from the same initial state.[6] L-systems, by contrast, follow a strict set of rules with no variation in the production process, making them particularly suited for modeling predictable and structured patterns found in nature.

The implementation of an L-system in a programming language like Python demonstrates these principles effectively. Through a simple script, one can observe how complex structures evolve from straightforward beginnings, governed by clear and consistent rules. This simulation not only highlights the power of L-systems in generating intricate patterns but also underscores their utility in educational and research settings, particularly in the fields of biology and computer graphics.

```
class LSystem:
    def __init__(self, axiom, rules):
        self.axiom = axiom
        self.rules = rules
        self.current_state = axiom

    def apply_rules(self):
        new_state = ""
        for character in self.current_state:
            new_state += self.rules.get(character, character)
        self.current_state = new_state

    def generate(self, num_iterations):
        for i in range(num_iterations):
            self.apply_rules()
            print(f"Iteration {i + 1}: {self.current_state}")
rules = {'A': 'AB', 'B': 'A'}
l_system = LSystem(axiom='A', rules=rules)
l_system.generate(5)
```

Code Snippet 2.1: Python L-system Parallel Rewriting Example

This Python example illustrates how parallel rewriting in L-systems enables simultaneous transformations of all elements in the string. This process is crucial for modeling complex, natural phenomena that involve concurrent developments in different parts of a structure. Below is the table showing the output for each iteration of the L-System starting with the axiom 'A' and rules where 'A' -> 'AB' and 'B' -> 'A':

Iteration	**Output**
1	A
2	AB
3	ABA
4	ABAAB
5	ABAABABA
6	ABAABABAABAAB
7	ABAABABAABAABABAABABA
8	ABAABABAABAABABAABABAABAABABAABAAB

Table 2.1: L-System Outputs Across Iterations

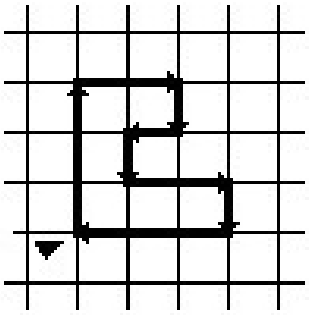
2.4 L-system Symbols

The symbols of the sytem can be graphically represented using the concept of **turtle graphics**.[28] In this graphical representation, commands are given to a virtual turtle that moves within a 2D space, similar to how a pen plotter operates. For instance, to draw a square, one might instruct a plotter to draw forward one centimeter, turn right, and repeat the process three more times. Unlike a plotter, however, a turtle in turtle graphics maintains an orientation, defined by Cartesian coordinates x and y, and a direction angle, which is defined by the symbol α.

To facilitate the turtle's movement in two dimensions, specific symbols are designated to represent actions such as movement and rotation. Common symbols used in L-system interpreters for these purposes are F, $+$, and $-$. After an L-system's production rules have been applied to generate a string, this string is parsed from left to right, with each symbol affecting the state of the turtle as follows:

- F: The turtle moves forward by d units, drawing a line.
- $+$: The turtle rotates left by an angle δ.
- $-$: The turtle rotates right by an angle δ.

Here, δ and d represent the global values dictating the magnitude of each symbol's effect on the turtle's rotation and movement, respectively. In systems where these parameters are non-parametric, the values remain constant throughout the system's execution. All the symbols of the system can be found in Apend[1] A.



$$FFF - FF - F - F + F + FF - F - FFF$$

Figure 2.4: Parsing a string.[3].

Now, a Python script has been generated to depict a more complex structure, namely the **Koch snowflake**[32]. Initially defined by the axiom 'F+F+F', representing an equilateral triangle, the L-system employs a production rule that replaces each 'F' with 'F-F+F-F'. This rule intricately adds segments and increases pattern complexity with each iteration.

```
koch = 'F+F+F' #axiom for Koch snowflake

#make the final L-System based on the number of iterations
for i in range(iterations):
    koch = koch.replace('F', 'F-F+F-F')
```

Code Snippet 2.2: Defining the Koch axiom as a string. B

Each iteration of the rule expands the string, which directs the turtle in a 2D space to draw lines ('F') and make turns ('+' for 120 degrees right and '-' for 60 degrees left). The movement commands are executed to depict the fractal nature of the Koch

snowflake, which exhibits increasingly detailed perimeters within a finite area. The length of each segment decreases geometrically with each iteration to fit the growing number of segments within the defined drawing space, ensuring that the fractal remains within viewable limits.

```
for move in koch:
    if move == 'F':
        turtle.forward(startLength / (3**(iterations-1)))
    elif move == '+':
        turtle.right(120)
    elif move == '-':
        turtle.left(60)
```

Code Snippet 2.3: Defining the Koch axiom rules. B.1

This iterative process demonstrates how a simple rule can generate complex natural patterns, showcasing the application of mathematical models in graphical simulations. A depiction of the output can be seen in the figures below:

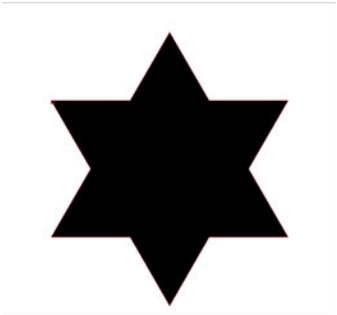


Figure 2.5: Koch Snowflake, 1 iteration

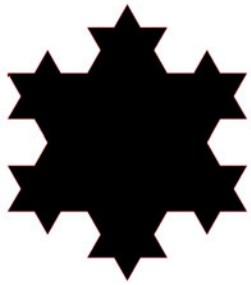


Figure 2.6: Koch Snowflake, 2 iterations

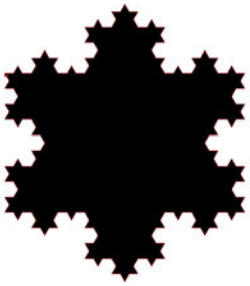


Figure 2.7: Koch Snowflake, 3 iterations

2.5 Types of L-Systems

This section of the thesis is dedicated to the classification of Lindenmayer Systems, which are pivotal in modeling the iterative and recursive processes observed in natural growth and fractal patterns. This exposition categorizes L-systems into five distinct types: *Deterministic, Stochastic, Context-sensitive, Parametric, and Bracketed L-systems*. Each classification encapsulates a unique approach to integrating environmental, probabilistic, or contextual influences into the modeling process, thus facilitating sophisticated simulations and analyses of complex systems.[20] The ensuing discourse aims to analyse the mechanics and applications of each L-system type, enhancing the comprehension of their transformative impact on theoretical and applied sciences.

2.5.1 Deterministic L-Systems

Deterministic L-systems, adhere to a strict set of substitution rules that lack any form of randomness. In these systems, each symbol in the axiom or initial string is consistently replaced by a predefined string according to unambiguous production rules[3]. The deterministic nature ensures that the same initial conditions and production rules will invariably produce the same output each time the system is iterated.

Key Features of Deterministic L-systems:

1. **Fixed Production Rules:** Each symbol in the axiom or previous string has a corresponding rule that applies deterministically, resulting in predictable and consistent outcomes. This feature allows deterministic L-systems to model structures that follow precise and repeatable patterns.

2. **Modeling Regular Structures:** Deterministic L-systems are particularly useful for simulating natural phenomena with highly regular structures. They are ideal for modeling organisms and phenomena where the same patterns repeat without variation, such as certain types of algae, ferns, and other fractal-like structures.

3. **Ease of Analysis:** Because the production rules are fixed and the outcomes are predictable, deterministic L-systems are easier to analyze and understand compared to stochastic systems.[7] This predictability allows researchers to precisely predict the growth and form of the modeled structures over time.

4. **Efficiency in Computation:** The absence of randomness in rule application makes deterministic L-systems computationally efficient. They can generate complex structures from simple rules without the need for probabilistic calculations, making them faster and more straightforward to implement in simulations.

One example of such a system is the **Dragon Curve**, this fractal is particularly famous for its appearance in Michael Crichton's novel "Jurassic Park", where it is used metaphorically to describe the unpredictable nature of dinosaur behavior, despite the curve itself being a deterministic and predictable pattern. In reality, the Dragon Curve is a visually appealing example of how simple rules can create complex and beautiful patterns, a characteristic feature of deterministic L-systems.[15]

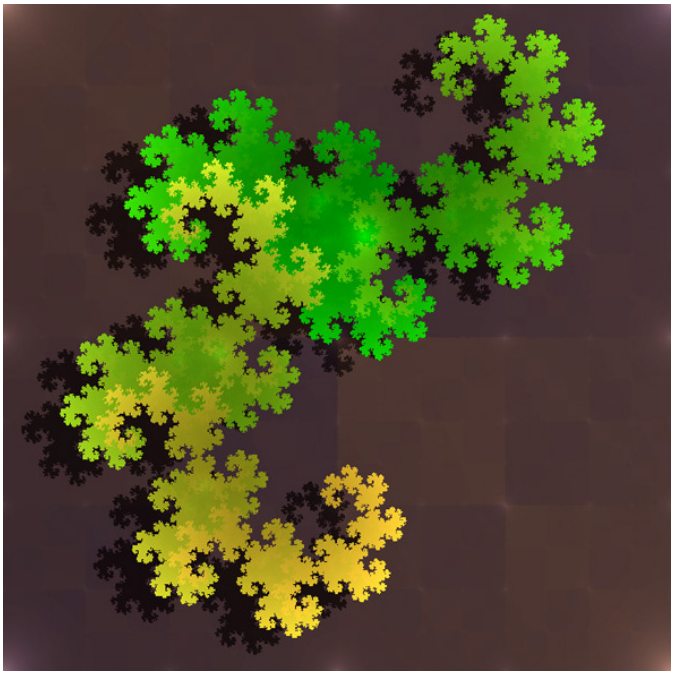


Figure 2.8: Graphical Representation of the Dragon Curve.[30]

```
axiom = "FX"
rules = {
    "X": "X+YF+",
    "Y": "-FX-Y"
}
```

Code Snippet 2.4: Defining a Dragon Curve in Python. B

2.5.2 Stochastic L-Systems

Stochastic L-systems introduce an element of randomness into the generative rules of Lindenmayer Systems, allowing for the simulation of more natural and diverse patterns that cannot be fully predicted by deterministic approaches. Unlike deterministic L-systems, where each symbol in the string has a fixed replacement, stochastic L-systems assign probabilities to different possible replacements for each symbol, thus incorporating variability in the generation process.[3]

Key Features of Stochastic L-systems:

1. **Probabilistic Rules:** Each production rule in a stochastic L-system is associated with a probability. When a rule is applied, one of several possible outcomes is selected based on their assigned probabilities. This allows the system to simulate natural variations seen in biological and ecological phenomena.
2. **Modeling Realistic Scenarios:** The inherent randomness in stochastic L-systems makes them particularly useful for modeling environments where biological variations and irregularities are the norm, such as the distribution of leaves on a tree or the branching patterns of coral.
3. **Simulation of Uncertainty:** Stochastic L-systems can simulate the uncertainty and environmental factors affecting growth patterns in nature, making them ideal for studies in theoretical biology and ecology where exact prediction is impossible.[27]

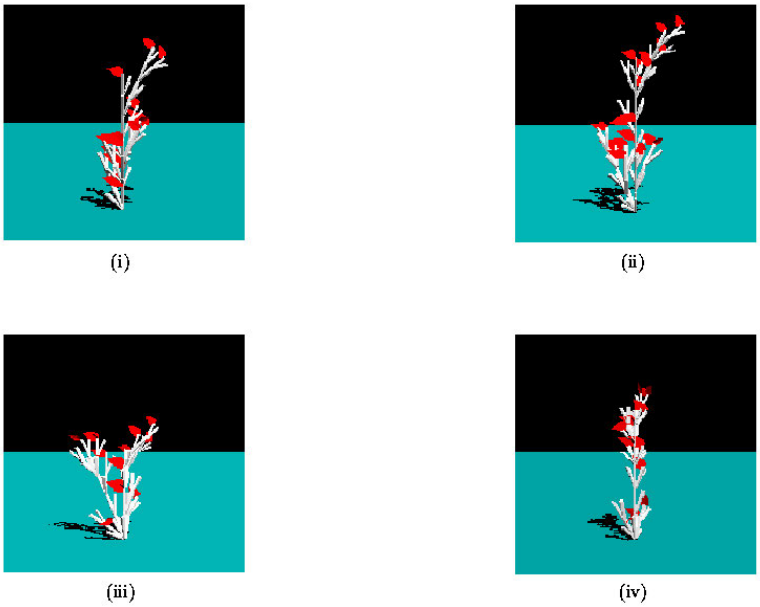


Figure 2.9: Graphical Representation of a stochastic plant with various probabilities of the generation pattern.[17]

```
# Stochastic L-system
axiom = "A"
rules = {
    "A": [("AB", 0.5), ("A", 0.5)],
    "B": [("A", 0.7), ("B", 0.3)]
# Each tuple contains a production and its probability
}
```

Code Snippet 2.5: Defining a Stochastic L-System in Python. B

2.5.3 Bracketed L-Systems

Bracketed L-systems provide a method for modeling branching structures such as those found in plants, trees, and other organic forms.

Key Features of Bracketed L-systems:

1. **Incorporation of Branching Structures:** Bracketed L-systems use square brackets ‘[‘ and ‘]‘ to denote the start and end of a branch, respectively. This

allows for the simulation of complex branching structures. The brackets enable the system to push and pop states onto a stack, facilitating the return to previous points in the structure for further development.

2. **Simulation of 3D Structures:** The use of brackets makes these systems particularly suitable for generating three-dimensional models. By using branching rules encoded within brackets, bracketed L-systems can effectively represent the spatial orientation of branches and other offshoots, which is crucial for realistic 3D rendering and analysis.[9]

3. **Flexible and Complex Modelling:** Bracketed L-systems provide a flexible framework that can model a wide range of natural and artificial structures. The ability to incorporate multiple branching levels within a single system allows for the creation of highly detailed and complex models that more accurately reflect the intricacies of natural growth patterns.

Figure 2.10: 2D plant-like structures from bracketed L-Systems.[21]

```
axiom = "X"
rules = {
    "X": "F-[[X]+X]+F[+FX]-X",
    "F": "FF"
}
```

Code Snippet 2.6: Defining a Bracketed L-System in Python. B

2.5.4 Parametric L-Systems

Parametric L-systems represent an advanced type of Lindenmayer system that incorporates parameters into the production rules to control and modify the attributes of the generated structures dynamically

Key Features of Parametric L-systems:

1. **Parameterization of Production Rules:** Parametric L-systems extend the classic L-system by incorporating parameters within the production rules. These parameters can vary and are used to control aspects of growth and development, such as angle, length, and width of structures. This allows for a more dynamic and detailed description of the modeled organisms.

2. **Enhanced Control Over Growth:** By using parameters, these systems offer enhanced control over the modeling process. Parameters can be adjusted based on environmental factors or developmental stages, allowing the system to simulate how changes in conditions affect the growth patterns of plants and other structures.[19]

3. **Increased Realism and Complexity:** The inclusion of parameters enables the simulation of more complex and realistic forms. This is particularly beneficial for studies in which biological realism is crucial, such as in the simulation of adaptive growth behaviors in response to environmental stimuli.

4. **Support for Conditional Logic:** Parametric L-systems can incorporate conditional logic into the production rules, making it possible to execute different rules based on the current value of parameters. This supports more complex decision-making processes within the model, closely mimicking natural decision processes observed in biological organisms.

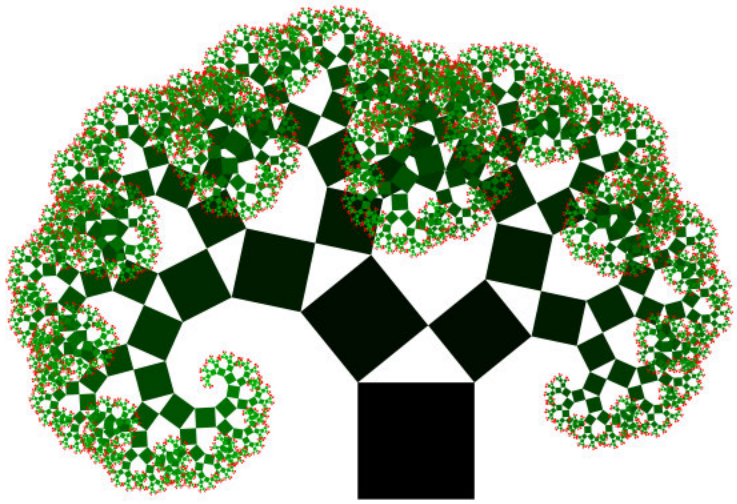


Figure 2.11: Pythagoras tree created with parametric L-system.[6]

```
def rule_A(params):
    x, y = params
    return [('B', [x + 1, y]), ('A', [x * 2, y / 2])]

def rule_B(params):
    x = params[0]
    return [('A', [x - 1]),]

# Example usage
axiom = [('A', [1, 2])]
rules = {
    'A': rule_A,
    'B': rule_B
}
```

Code Snippet 2.7: Defining a Parametric L-System in Python. B

2.5.5 Context-sensitive L-Systems

Context-sensitive L-systems are an extension of the classic Lindenmayer systems, designed to model environments and organisms where growth patterns depend on the local context of individual components.

Key Features of Context-Sensitive L-systems:

1. **Dependence on Neighboring Information:** Context-sensitive L-systems, unlike traditional L-systems, consider the neighboring characters (both predecessors and successors) around a focal character when applying production rules. This context dependency allows for the simulation of more complex biological phenomena where growth patterns depend on local interactions.

2. **Enhanced Modeling of Biological Processes:** These systems can more accurately model biological processes such as the phyllotactic arrangement of leaves, branching patterns in plants, and even cellular development, where interactions between adjacent elements play a critical role in determining growth outcomes.[23]

3. **Increased Computational Complexity:** While context-sensitive L-systems offer more detailed modeling capabilities, they also require more complex computations. The need to constantly evaluate the context of each character increases the computational overhead compared to context-free systems.[12]

4. **Ability to Simulate Environmental Interactions:** The context-sensitivity of these systems makes them particularly suited for simulating how environmental factors or local densities affect growth and development. This can be crucial for ecological and environmental modeling, where the behavior of one part of a system can be significantly influenced by its immediate surroundings.

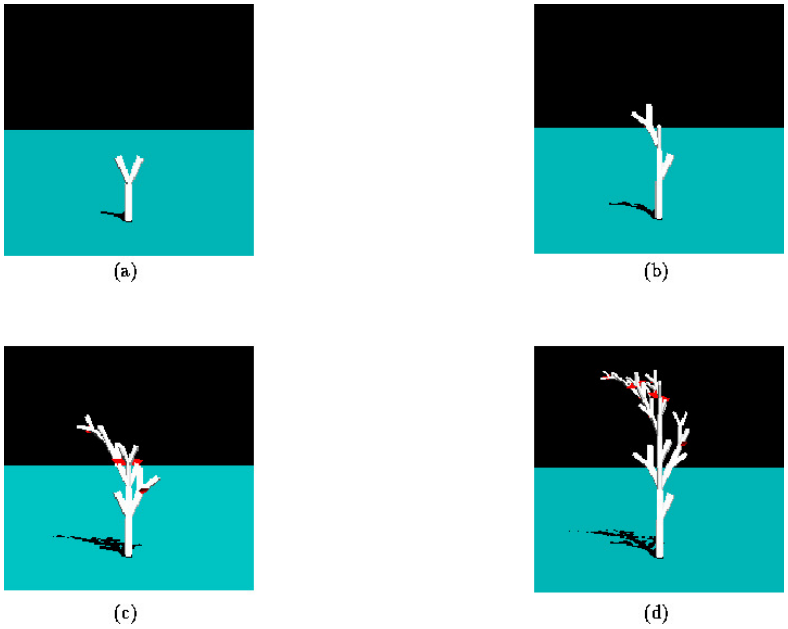


Figure 2.12: Context-sensitive L-System.[16]

```
axiom = ['A', 'A', 'A']
rules = {
    (None, 'A', 'A'): lambda: 'A',  # Context: Nothing on the left
        , A on the right
    ('A', 'A', 'A'): rule,  # Context: A on both sides
    ('A', 'A', None): lambda: 'A'  # Context: A on the left,
        Nothing on the right
}
```

Code Snippet 2.8: Defining a Context-Sensitive L-System in Python. B

2.6 Software & Development Environment

This section focuses on thoroughly explaining the necessary choices regarding the software that is going to be utilised throughout the entire experiment.

2.6.1 Python

For the following thesis **Python** has been chosen as the main programming language, since it is highly regarded for its applicability in simulating and visualizing L-systems due to its extensive library support, and robust community resources. The language's readability makes it ideal for educational and research purposes. Its rich ecosystem includes libraries such as *matplotlib* for plotting, *numpy* for numerical operations, and *turtle* for graphical outputs, which are essential for visualizing the intricate patterns generated by L-systems.

Furthermore, Python's cross-platform nature ensures that L-system simulations are widely accessible across different operating systems. Although not the fastest language, Python's performance is adequate for most L-system simulations and can be enhanced by integrating with performance-optimized languages like C when necessary.

Miniconda

For the purposes of this thesis, which focuses on the exploration and visualization of L-systems, **Miniconda** offers several advantages that are critical to ensuring efficient and reproducible research.

Miniconda is a minimalistic version of Anaconda, a popular open-source distribution of the Python and R programming languages for scientific computing. By providing only the package manager and Python, Miniconda offers a lightweight, flexible alternative to Anaconda, which includes a large number of scientific packages by default. This minimal setup allows users to install only the specific packages they need, thus avoiding unnecessary bloat and reducing potential conflicts between package dependencies.[2]

Jupyter Notebook

The Jupyter Notebook is the original web application for creating and sharing computational documents. It offers a simple, streamlined, document-centric experience.[14]

In the context of this thesis, which involves the exploration and visualization of Lindenmayer Systems (L-Systems), Jupyter Notebooks offer several compelling advantages. Firstly, the interactive environment provided by Jupyter Notebooks facilitates exploratory research and iterative testing, which are essential for developing and fine-tuning the algorithms associated with L-systems. Users can write code and observe the output directly in the notebook, making adjustments as needed and visually tracking the impact of these changes on the generated fractals and graphical outputs.

Tkinter

Tkinter, the standard Python interface to the Tcl/Tk GUI toolkit, has been selected for the development of graphical user interfaces (GUIs) in the visualization component of this thesis, which focuses on the implementation and analysis of L-systems. Tkinter provides a robust and platform-independent framework, which is particularly advantageous for developing educational and research applications that require a straightforward and effective means for users to interact with the system.[24]

The choice of Tkinter was motivated by several factors that align well with the needs of the project. Firstly, Tkinter's simplicity and integration with Python allow for rapid GUI development, which is crucial in academic projects where time and resources may be limited. This simplicity also facilitates quick modifications and iterations, which are often necessary in a research setting to refine user interactions based on experimental feedback.

Matplotlib

Matplotlib, a widely recognized plotting library in Python, is employed in this thesis to facilitate the visualization of Lindenmayer Systems (L-systems). Matplotlib offers a comprehensive suite of plotting functions that are adept at creating static, animated, and interactive visualizations, which is essential for illustrating the complex geometrical structures generated by L-systems.

The decision to use Matplotlib was grounded in its robust capabilities and flexibility, which are crucial for the detailed visualization tasks required in this research. The library's ability to produce a wide range of plots, from simple line diagrams to complex color maps and 3D diagrams enables the detailed representation of fractal patterns and growth processes modeled by L-systems. Furthermore, Matplotlib's extensive customization options allow these visualizations to be precisely tailored to specific research needs, enhancing the clarity and effectiveness of data presentation.[25]

Numpy

NumPy, a fundamental package for numerical computing with Python, plays a pivotal role in the computational analysis and modeling of Lindenmayer Systems (L-systems) in this thesis. Renowned for its powerful N-dimensional array object and broad suite of mathematical functions, NumPy provides the computational efficiency and functionality necessary for handling the complex calculations that L-systems require.[18]

The choice to utilize NumPy in this research is justified by its array-centric architecture, which is highly optimized for performance. NumPy arrays facilitate efficient storage and manipulation of large datasets, which is essential for generating and exploring the intricate fractal patterns associated with L-systems. These capabilities allow for rapid processing and transformation of data, crucial for real-time visualization and analysis.

Turtle

The Turtle module, a popular Python library for creating visual graphics and drawings, is employed in this thesis to demonstrate the development and structure of Lindenmayer Systems (L-systems). Originating from the Logo programming language, the Turtle module provides an intuitive and accessible means for users to visualize and interact with geometric computations and fractal patterns, which are central to L-systems.[26]

The selection of Turtle for this project is primarily due to its straightforward graphical capabilities that allow users to directly translate algorithmic logic into visual form. This feature is particularly beneficial for L-systems, where the growth patterns and rules can be complex and abstract. By using Turtle, these patterns can be rendered visually as they develop, offering immediate graphical feedback that is crucial for understanding and fine-tuning the L-system parameters.

2.6.2 VS Code

Visual Studio Code (VS Code) is a powerful and versatile Integrated Development Environment (IDE) that has been selected for the software development tasks in this thesis on L-systems. Developed by Microsoft, VS Code offers comprehensive coding support and a wide range of extensions that enhance its functionality, making it an ideal choice for modern software development, especially in the context of academic research.[4]

3 Requirements

3.1 Stakeholders

In this section, we identify and describe the primary and secondary stakeholders involved in the development and use of the system. Understanding the stakeholders is crucial for ensuring that the system meets their needs and expectations.

3.1.1 Primary Stakeholders

The primary stakeholders are those directly involved in the development and primary usage of the proposed L-system[29]. They include:

Stakeholder	**Description**
Developers and Programmers	Responsible for designing, implementing, and maintaining the system. They ensure that the system is robust, efficient, and user-friendly, incorporating feedback from other stakeholders. They aim to create a system that is both powerful for research purposes and accessible for educational use.

Table 3.1: Details for Developers and Programmers

Stakeholder	Description
Researchers and Students	Primary users of the system. Their main interest lies in exploring and understanding fractals and L-systems. **Exploring** involves a hands-on interaction with the system, where users experiment with various L-system rules, parameters, and initial conditions to generate diverse patterns. This includes testing deterministic or stochastic rules, modifying growth angles, and adjusting branching patterns. The goal of exploration is not merely academic curiosity, but to discover new behaviors and properties of fractals, identify emergent patterns, and test hypotheses regarding fractal generation and real-world phenomena. **Understanding** is a deeper analytical process where users focus on the mathematical and geometric properties of generated fractals. This involves analyzing how different parameters influence fractal structures, interpreting their practical implications, and developing theories that explain these behaviors. The goal of understanding is to generalize these findings, allowing researchers to contribute to scientific knowledge, publish their results, and apply fractal models to real-world problems.

Table 3.2: Details for Researchers and Students

3.1.2 Secondary Stakeholders

The secondary stakeholders are those who indirectly interact with the system or benefit from its results[29]. They include:

Stakeholder	Description
Environmental Scientists	Use the system to model natural growth patterns and ecological phenomena, focusing on its capability to simulate realistic environmental interactions and changes.
Educational Institutions	Incorporate the system into their curriculum for teaching computational modeling, natural patterns, and related subjects.

Table 3.3: Secondary Stakeholders

3.2 Functional Requirements

The functional requirements outline the specific behaviors and functions that the system must support to meet the needs of its stakeholders[8]. These requirements ensure that the system performs the necessary tasks effectively.

Requirement	**Description**
Interactive Visualization	The system must provide an interface for interactive visualization of fractals and L-systems, allowing users to manipulate parameters and observe real-time changes.
Rule Modification Interface	Users should be able to define and modify L-system rules through a user-friendly interface, supporting both deterministic and stochastic variations.
User Interface Elements	The system must include sliders, buttons, text boxes, and color pickers to allow users to adjust parameters such as growth angles, branching patterns, and visualization colors. Given the educational background of the primary stakeholders, the system should prioritize efficiency and clarity, allowing users to focus on research and experimentation. Advanced functionality should be easily accessible but not overwhelming.

Table 3.4: User Interaction and Visualization Requirements

Requirement	Description
Real-Time Data Integration	The system must be capable of integrating real-time sensor data to modify the parameters of fractal and L-system models dynamically.
Saving and Loading Configurations	The system should allow users to save their configurations and load them later, enabling reproducibility and sharing of specific setups.
Exporting Visualizations	The system must support exporting visualizations as images or data files for further analysis, presentations, or academic papers. Researchers and students need this functionality to document findings and share them in research publications or classroom settings.
Scalability	The system should be scalable to accommodate various levels of complexity in fractal and L-systems models, ensuring that it can handle both simple and highly detailed patterns. This scalability is particularly crucial as research may require testing L-systems of increasing complexity.

Table 3.5: Data Management and System Capabilities Requirements

3.3 Non-Functional Requirements

The non-functional requirements describe the overall qualities and constraints of the system. These requirements ensure that the system is usable, reliable, and efficient, meeting the broader expectations of the stakeholders[8].

Requirement	Description
Performance	The system must perform efficiently, handling large datasets and complex computations without significant delays. This is crucial for researchers and students who may work with highly detailed fractal models.
Usability	While the primary stakeholders are highly educated, the user interface should still be intuitive and efficient, offering advanced options for experienced users without creating unnecessary complexity. Ease of use is important for speeding up the exploration and understanding processes.
Reliability	The system must be reliable enough to ensure that research tasks can be completed without critical errors. However, given the experimental nature of the research environment, occasional system failures or maintenance are acceptable as long as they are well communicated and do not hinder the reproducibility of results.
Extensibility	The system should be designed with extensibility in mind, allowing for easy integration of new features, models, and data sources in the future. This is important for accommodating future research needs.
Compatibility	The system must be compatible with various operating systems and devices, ensuring broad accessibility for all users. This allows researchers to work in diverse computing environments.

Table 3.6: Non-Functional Requirements

3.4 User Requirements

The user requirements detail what the end-users expect from the system. These requirements are derived from the needs and goals of the stakeholders.

Requirement	Description
Ease of Use	The system should be easy to use, with an intuitive interface that allows users to quickly learn and operate the system without extensive training. Given their educational background, primary stakeholders should be able to navigate the system efficiently while having access to advanced features as needed.
Customization	Users should be able to customize the visualization parameters and L-system rules to suit their specific needs and preferences. This is crucial for exploratory research.
Feedback	The system should provide immediate visual feedback to users as they adjust parameters, allowing them to see the impact of their changes in real-time. This supports the exploratory and experimental nature of their work.
Documentation	Comprehensive documentation should be available to guide users on how to use the system effectively, including examples and tutorials. This supports both new and experienced users in utilizing the system's full capabilities.

Table 3.7: User Requirements

3.5 Use Cases

The following use cases describe how users will interact with the system to achieve specific goals. Each use case outlines the steps involved in performing a particular task.

3.5.1 Use Case 1: Parameter Exploration and Adjustment

Description: Users interact with sliders and buttons to adjust parameters like temperature, angle, branch length, and thickness, observing real-time changes in the L-system visualization.

Actors: Researchers, Students

Steps:

1. User selects a parameter to adjust (e.g., angle, branch length).
2. User modifies the parameter using sliders or input boxes.
3. System dynamically updates the visualization to reflect the changes.
4. User observes the changes in real-time and makes notes or adjustments.

Goal: To understand the impact of individual parameters on the fractal patterns, allowing for hypothesis testing and analysis of sensitivity to initial conditions.

3.5.2 Use Case 2: Comparative Analysis of Models

Description: Users can switch between deterministic and stochastic variations to see how these changes affect the growth patterns of the L-system.

Actors: Researchers, Students

Steps:

1. User selects the model type (deterministic or stochastic).
2. User runs simulations for both models with the same initial conditions.
3. System generates and displays the visualizations for both models.
4. User compares the visualizations side-by-side to identify differences and similarities.

Goal: To explore and compare different growth models, enhancing the understanding of fractal generation under varying conditions.

3.5.3 Use Case 3: Real-Time Data Integration

Description: The system adjusts fractal parameters based on real-time environmental data.

Actors: Environmental Scientists, Researchers

Steps:

1. System fetches real-time environmental data (e.g., temperature, humidity).
2. System updates the relevant parameters based on the fetched data.
3. Visualization dynamically reflects the changes due to real-time data input.
4. User observes the impact of real-time data on fractal growth.

Goal: To study the effects of real-world conditions on fractal patterns and to simulate dynamic systems.

3.5.4 Use Case 4: Hypothesis Testing

Description: Researchers formulate and test hypotheses about fractal behavior.

Actors: Researchers, Students

Steps:

1. User formulates a hypothesis regarding fractal behavior.
2. User sets initial conditions and parameters to test the hypothesis.
3. User runs simulations and observes the outcomes.
4. User records results and compares them against the hypothesis.
5. User refines the hypothesis based on observed data.

Goal: To validate or refute hypotheses about fractal behavior and to advance scientific understanding.

3.5.5 Use Case 5: Save and Load Configurations

Description: Users save their current system settings to a file and load them later for further examination or continuation of work.

Actors: Researchers, Students

Steps:

1. To save, the user clicks the 'Save' button, and the system writes the current configuration to a file.
2. To load, the user clicks the 'Load' button, and the system retrieves configurations from a file and applies them to the visualization.

Goal: To ensure reproducibility of experiments and facilitate the sharing of specific setups with colleagues or for publication purposes.

3.5.6 Use Case 6: Export Visualization as Image

Description: Users export the current visualization of the L-system as an image file for use in reports or presentations.

Actors: Researchers, Students, Educational Institutions

Steps:

1. User clicks the 'Export' button.
2. System prompts the user to choose a file location and name.
3. System saves the current visualization to the specified location as an image file.

Goal: To document and present findings in academic papers, presentations, or educational materials, ensuring that visual evidence of the research is easily accessible and shareable.

3.5.7 Use Case 7: Randomize Parameters

Description: At the click of a button, users can randomize the visualization parameters to explore various random configurations of the L-system.

Actors: Researchers, Students

Steps:

1. User clicks the 'Randomize' button.
2. System randomly adjusts all adjustable parameters.
3. Visualization updates to reflect the new, randomized settings.

Goal: To stimulate creativity and discover unexpected patterns or properties by exploring a wide range of random configurations.

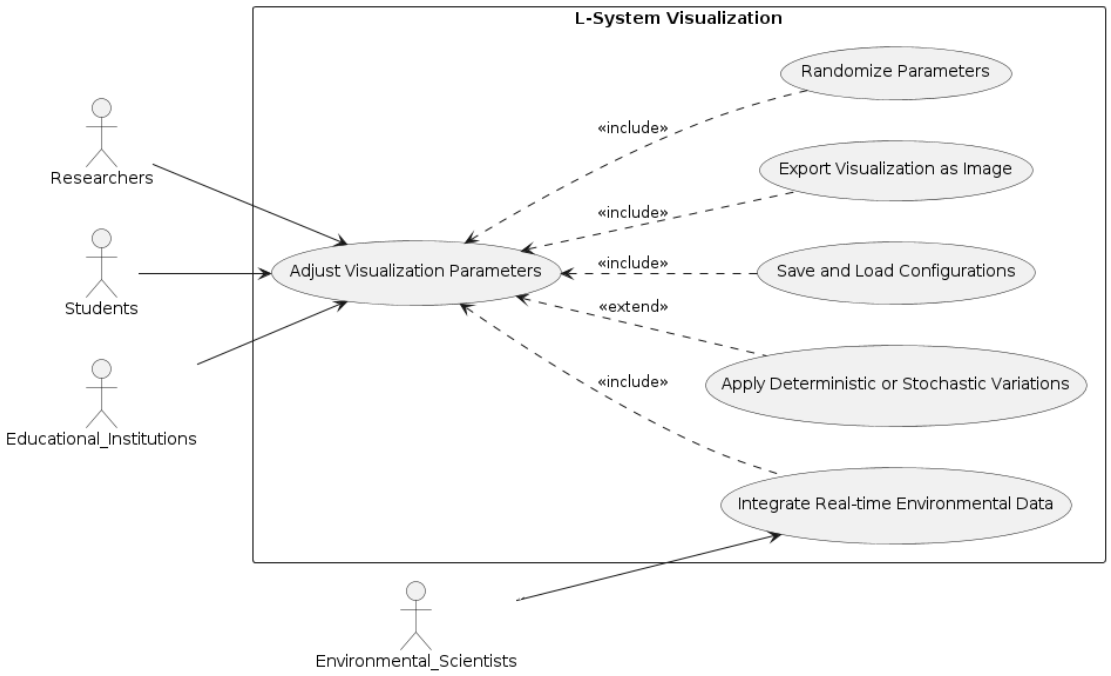


Figure 3.1: UML Use Case Diagram

4 Concept

This section outlines the conceptual framework for developing a dynamic, responsive L-system that can adjust its parameters based on real-time data. The aim is to create a system that not only simulates natural growth patterns but also adapts to varying environmental conditions, making it highly flexible and applicable for modeling complex, changing phenomena. By planning the structure and approach meticulously, we ensure a robust implementation that meets the objectives effectively.

4.1 Objectives

The primary goal of this project is to develop an interactive L-system capable of real-time adaptation. Specifically, we aim to:

- Simulate natural growth patterns that respond dynamically to environmental changes.
- Incorporate user interface elements that allow real-time manipulation of parameters.
- Integrate real-time data inputs to drive the L-system's adaptability.
- Utilize both deterministic and stochastic rule variations to enhance realism.
- Ensure the system is modular and extensible, allowing future enhancements and modifications.

4.2 Planned Structure

The implementation will be structured into several key components:

1. **Imports and Logging Setup**

2. **Branching Parameter Calculation**
3. **L-System Class**
4. **User Interface Elements**
5. **Helper Functions**
6. **Main Execution Flow**

4.2.1 Imports and Logging Setup

The foundation of the implementation involves importing necessary libraries and setting up logging. The libraries to be used will support mathematical operations, visualization, event recording, time-based functions, random number generation, configuration management, and file dialogs. Logging will be configured to capture and log messages with timestamps, helping track the flow of the program and debugging issues.

4.2.2 Branching Parameter Calculation

To achieve adaptability, the system will dynamically adjust the branching parameters based on external factors such as temperature. The temperature variable will be utilized as a dummy in this case. Further implementation shall include real-time data. The planned approach includes:

- Defining realistic bounds for parameters to ensure natural growth patterns.
- Calculating branch attributes (length, angle, thickness) based on environmental data.
- Introducing random variations to simulate natural randomness.
- Generating L-system rules that incorporate these dynamic parameters.
- Applying context-sensitive rules for more sophisticated transformations.

4.2.3 L-System Class

The core logic of the L-system will be encapsulated in a dedicated class. This class will manage the generation and rendering of the L-system, including:

- Initializing with the starting axiom and iteration settings.
- Generating the L-system string by applying rules iteratively.
- Supporting context-sensitive rules for complex transformations.
- Visualizing the L-system on a graphical interface, including branch and leaf drawing.

4.2.4 User Interface Elements

To facilitate user interaction, various UI elements will be developed:

- **Sliders** for adjusting parameters like temperature, angle, and branch length.
- **Buttons** for actions such as saving and loading configurations, and resetting parameters.
- **Text input boxes** for defining and modifying L-system rules.
- **Color pickers** for customizing the colors of branches and leaves.

4.2.5 Helper Functions

Helper functions will support the core functionality, including:

- Drawing a reference grid to help users understand the scale and orientation of the L-system.
- Updating the L-system drawing in real-time based on user inputs.
- Simulating real-time environmental data to dynamically adjust parameters.
- Managing configuration files for saving and loading setups.
- Exporting images of the current L-system state for documentation and analysis.

- Randomizing parameter values to facilitate quick testing of different configurations.

4.2.6 Main Execution Flow

The main function will initialize the graphical interface and UI elements, and manage the event loop. This will involve:

- Setting up the Pygame environment and initializing UI components.
- Handling user interactions with sliders, buttons, and text inputs.
- Continuously updating and redrawing the L-system based on current parameters.
- Implementing functionalities for saving, loading, exporting, and applying rule variations.

4.3 UML Diagram

For a better visual understanding of the system, the following UML-Diagram has been created to sum up all the previously mentioned aspects and modules:

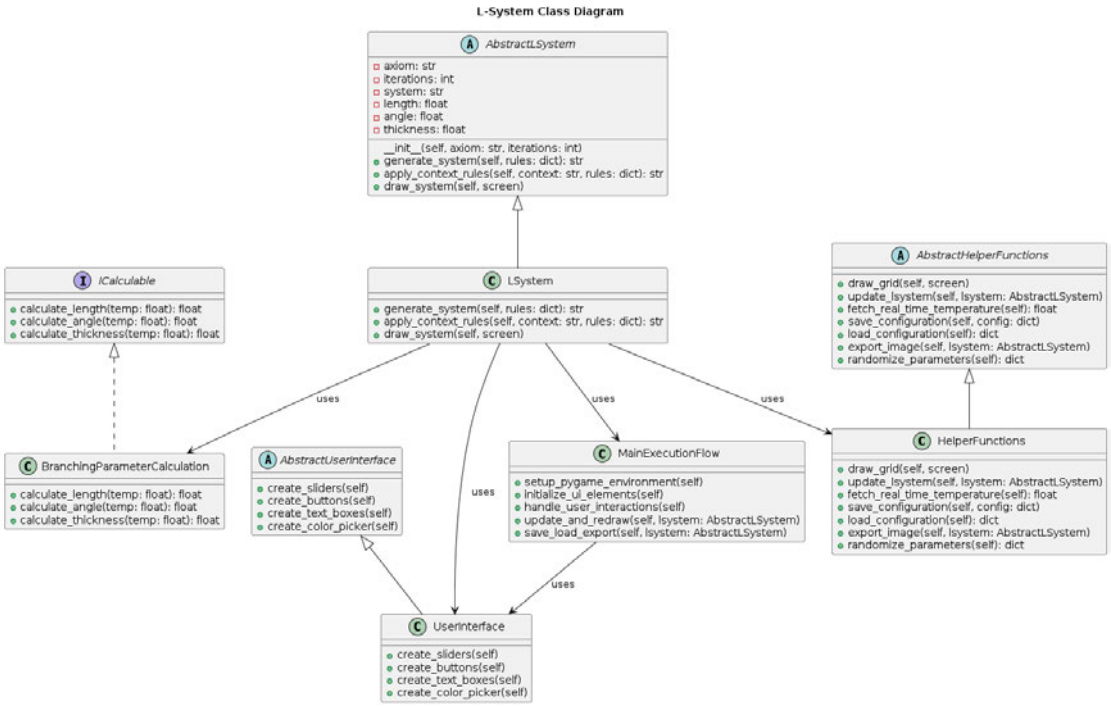


Figure 4.1: UML-Diagram of the system.

4.4 Summary

This conceptual plan lays the groundwork for a dynamic and interactive L-system. By integrating real-time data, user interface elements, and both deterministic and stochastic rule variations, the system will provide a robust platform for exploring natural growth patterns and their responses to environmental changes. The modular and extensible design ensures that the system can be adapted and expanded for various applications, making it a valuable tool for modeling complex phenomena.

5 Implementation

In this section, the development of a *dynamic, responsive L-system* capable of adjusting its parameters based on *real-time data* is detailed. This implementation enhances the *flexibility* and *applicability* of L-systems. The primary goal is to create an interactive system that can simulate natural growth patterns influenced by varying environmental conditions. The system's adaptability is achieved through the integration of user interface elements and real-time data, providing a robust platform for exploring various growth scenarios. The most crucial part of this experiment is analyzing the **branching behavior** of the system.

The implementation is structured into several key components:

1. **Imports and Logging Setup**
2. **Branching Parameter Calculation**
3. **L-System Class Implementation**
4. **User Interface Elements**
5. **Helper Functions**
6. **Main Execution Flow**

5.1 Imports and Logging Setup

To begin with, the necessary libraries are imported. This includes **numpy** for mathematical operations, **Pygame** for visualization, **logging** for recording events, **datetime** for time-based functions, **random** for random number generation, **json** for configuration management, and **tkinter** for file dialogs. The `lru_cache` from **functools** is employed to cache function results, optimizing performance. These imports form the backbone

of the implementation, ensuring that all required tools to handle computations, visualizations, and configurations efficiently are available. The choice of libraries reflects a balance between performance and ease of use, making the development process both straightforward and effective.

```
import numpy as np
import pygame
import logging
import datetime
import random
import json
from functools import lru_cache
from pygame.locals import *
from tkinter import filedialog
from tkinter import Tk
import cProfile
import pstats
import io
```

Code Snippet 5.1: Imports

Logging is configured to capture and log messages into a file named `lsystem_growth.log`. This setup helps in tracking the flow of the program and debugging issues. The log file format includes **timestamps**, which are crucial for understanding the sequence of events and the state of the system at any given time. The logging configuration is set to log information level messages, ensuring that both regular operational messages and error messages are recorded.

```
logging.basicConfig(filename='lsystem_growth.log',
                    level=logging.INFO,
                    format='%(asctime)s - %(message)s')
```

Code Snippet 5.2: Logging Setup

Timestamp	Temperature	Angle	Formula
2024-06-04 11:35:23,153	17.625	10	F[+++F][-F]F[+F][–F]F[+++F]

Table 5.1: L-system parameters at timestamp 1 (taken from the log file).

Timestamp	Temperature	Angle	Formula
2024-06-04 11:35:23,186	13.0	10	F[+F][—F]F[+F][-F]F

Table 5.2: L-system parameters at timestamp 2 (taken from the log file).

5.2 Branching Parameter Calculation

To introduce adaptability in L-systems, it is essential to dynamically adjust the branching parameters based on external factors such as temperature. The function `calculate_branching_parameters` is defined to compute these parameters. Initially, the bounds for temperature, branch length, and angles are set to ensure that the system operates within realistic limits. This setup is vital for simulating natural growth patterns that respond to environmental changes. By defining these bounds, the system can prevent unrealistic growth scenarios and maintain a level of biological plausibility.

For testing purposes the temperature in this case will be an easily adjustable dummy value. Further implementations might use real-time data from sensors.

5.2.1 Formula

Given the temperature T, base angle θ, branch length L, and branch thickness σ:

$$L(T) = L + \frac{(T - T_{\min})}{(T_{\max} - T_{\min})} \cdot (L_{\max} - L_{\min}) + \text{length\_variation},$$
$$\theta(T) = \theta + \frac{(T - T_{\min})}{(T_{\max} - T_{\min})} \cdot (\theta_{\max} - \theta_{\min}) + \text{angle\_variation},$$
$$\sigma(T) = \sigma + \frac{(T - T_{\min})}{(T_{\max} - T_{\min})} \cdot 2.$$

Here, length_variation and angle_variation are random variations if use_random is True, otherwise they are zero.

5.2.2 Implemented Code

```
@lru_cache(maxsize=None)
def calculate_branching_parameters(temperature, base_angle,
    branch_length, branch_thickness, use_random,
    deterministic_func, stochastic_rule=None, context_rules=()):
    min_length, max_length = 8, 18
    min_angle, max_angle = 5, 18
    min_temp, max_temp = 0, 30
    temperature = min(max(temperature, min_temp), max_temp)
    length = branch_length + (temperature - min_temp) / (max_temp
        - min_temp)* (max_length -min_length)
    angle = base_angle + (temperature - min_temp) / (max_temp -
        min_temp)* (max_angle - min_angle)
    thickness = branch_thickness + (temperature - min_temp) / (
        max_temp - min_temp) * 2
    length_variation = random.uniform(-0.5, 0.5) if use_random
        else 0
    angle_variation = random.uniform(-2, 2) if use_random else 0
    length += length_variation
    angle += angle_variation
```

Code Snippet 5.3: Branching Parameter Calculation

The attributes of a branch in a fractal structure are calculated, allowing the following functionality:

1. Compute the length, angle, and thickness of the branch based on temperature.
2. Optionally add random variations to length and angle.

Structure:

- **Length Calculation**: Adjusts the branch length based on temperature, interpolating between a minimum and maximum length.
- **Angle Calculation**: Adjusts the branch angle similarly, interpolating between a base and maximum angle.
- **Thickness Calculation**: Adjusts the branch thickness with a linear scale factor.
- **Random Variations**:
  - If randomness is enabled, adds a random variation to the length (between -0.5 and 0.5) and to the angle (between -2 and 2 degrees).
  - If deterministic, no variation is added.
- **Final Adjustment**: Updates the length and angle with the calculated variations.

5.3 Generation of the L-System

5.3.1 Formula

Given the axiom A, set of rules R, and number of iterations I. The generation process can be described recursively:

$$S_{k+1} = \begin{cases} A & \text{if } k = 0, \\ R(S_k) & \text{if } k > 0, \end{cases}$$

where S_k is the string at iteration k.

5.3.2 Implemented Code

```
    steps = int(temperature // 5)
    rule = "F"
    for i in range(steps):
        if use_random:
            plus_variation = '+' * (random.randint(1, 3))
            minus_variation = '-' * (random.randint(1, 3))
```

```
7         else:
8             plus_variation, minus_variation =
                  deterministic_rule_variation(i, steps,
                  deterministic_func)
9         rule = f"F[{plus_variation}F][{minus_variation}F]{rule}"
```

Code Snippet 5.4: Rule Generation

Structure

- **Steps Calculation**: The number of iterations is determined by dividing the temperature by 5.
- **Initialization**: The initial rule is set to a forward movement, denoted by "F".
- **Iteration and Variation**:
  - If randomness is enabled, generate random sequences of '+' and '-' with lengths between 1 and 3.
  - If deterministic, use a predefined function to generate variations.
  - Append and prepend these variations to the rule in each iteration, building complexity.

5.3.3 Overwriting Default Behaviour

Finally, if a stochastic rule is provided, it overrides the deterministic rule. Context rules, if any, are applied to the final rule string. The structure is as follows:

```
1     if stochastic_rule:
2         rule = stochastic_rule
3
4     context_rules_dict = dict(context_rules)
5     if context_rules_dict:
6         for context, replacement in context_rules_dict.items():
7             rule = rule.replace(context, replacement)
8
9     return {
10        "F": rule,
```

```
        "length": length,
        "angle": angle,
        "thickness": thickness
    }
```

Code Snippet 5.5: Returning Rules

1. Optionally replace the rule with a stochastic rule.
2. Apply context-specific replacements to the rule.
3. Return the final rule and branch attributes.

Structure

- **Stochastic Rule Replacement**:
  - If a stochastic rule is provided, it replaces the current rule.
- **Context-Specific Rule Replacement**:
  - Convert the list of context-specific rules into a dictionary.
  - Iterate over the dictionary, replacing each context in the rule with its corresponding replacement.
- **Return Statement**:
  - Return a dictionary containing the final rule, branch length, angle, and thickness.

5.4 Deterministic Approach

In order to further incorporate rule variations: two functions, `calculate_variation` and `deterministic_rule_variation`, are defined to achieve this. These functions generate variations for deterministic rule creation based on mathematical functions such as sine, cosine, and exponential (etc). By employing different mathematical functions, the system can create a wide range of branching patterns, each with unique characteristics.

5.4.1 Formulae

Let i be the current step, and n be the total number of steps. The variation calculation depends on the chosen deterministic function $f(i, n)$:

$$\text{variation} = \begin{cases} +* & \text{if } f(i,n) > 0, \\ -* & \text{if } f(i,n) < 0, \end{cases}$$

where $f(i, n)$ can be:

$$\begin{aligned}
\text{step} : f(i,n) &= i\%3, \\
\text{sine} : f(i,n) &= \sin\left(\frac{i\pi}{n}\right), \\
\text{cosine} : f(i,n) &= \cos\left(\frac{i\pi}{n}\right), \\
\text{exponential} : f(i,n) &= \exp\left(\frac{i}{n}\right), \\
\text{logarithmic} : f(i,n) &= \log(i+1), \\
\text{polynomial} : f(i,n) &= \left(\frac{i}{n}\right)^2, \\
\text{tangent} : f(i,n) &= \tan\left(\frac{i\pi}{n}\right), \\
\text{sqrt} : f(i,n) &= \sqrt{i}, \\
\text{sinh} : f(i,n) &= \sinh(i), \\
\text{sigmoid} : f(i,n) &= \frac{1}{1+\exp(-i)}, \\
\text{tanh} : f(i,n) &= \tanh(i), \\
\text{inverse} : f(i,n) &= \frac{1}{i+1}.
\end{aligned}$$

5.4.2 Implemented Code

```
1 def calculate_variation(value, multiplier):
2     return '+' * (int(abs(value * multiplier) + 1)),
3            '-' * (int(abs(value * multiplier) + 1))
```

Code Snippet 5.6: Calculate Variation

```
def deterministic_rule_variation(i, steps, deterministic_func):
    multiplier = 3  # Common multiplier for variation calculations

    if deterministic_func == "sine":
        plus_variation, minus_variation = calculate_variation(np.
            sin(i), multiplier)
    elif deterministic_func == "cosine":
        plus_variation, minus_variation = calculate_variation(np.
            cos(i), multiplier)
    elif deterministic_func == "exponential":
        value = np.exp(i / steps)
        plus_variation, minus_variation = calculate_variation(
            value, 2)
```

Code Snippet 5.7: Deterministic Rule Variation

The function `deterministic_rule_variation` generates deterministic variations based on different mathematical functions. The structure is as follows:

1. Introduce a common multiplier to scale the variations.
2. Depending on the chosen deterministic function, calculate the variation.
    - For the `sine` function, compute the sine of the current step i.
    - For the `cosine` function, compute the cosine of the current step i.
    - For the `exponential` function, compute the exponential of the step normalized by the total number of steps.

This approach allows the `deterministic_rule_variation` function to apply predictable and mathematically varied modifications to L-system rules. By selecting different deterministic functions, a wide range of fractal patterns with unique characteristics can be generated.

Function	Resulting Formula
Initial	F
Step	$F[+F][-F]F[+F][-F]F[+F][-F]F$
Exponential	$F[++++F][----F]F[+++F][---F]F[+++F][---F]F$
Logarithmic	$F[+++F][---F]F[++F][--F]F[+F][-F]F$
Polynomial	$F[++F][--F]F[+F][-F]F[+F][-F]F$
Tangent	$F[++++++F][------F]F[++++++F][------F]F[+F][-F]F$
Square Root	$F[+++++F][-----F]F[++++F][----F]F[+F][-F]F$
Sinh	$F[++++++++++++F][------------F]F[++++F][----F]F[+F][-F]F$
Sigmoid	$F[+++F][---F]F[+++F][---F]F[++F][--F]F$
Tanh	$F[+++F][---F]F[+++F][---F]F[+F][-F]F$
Inverse	$F[++F][--F]F[++F][--F]F[++++F][----F]F$

Table 5.3: Modification of the formula F based on different mathematical functions.

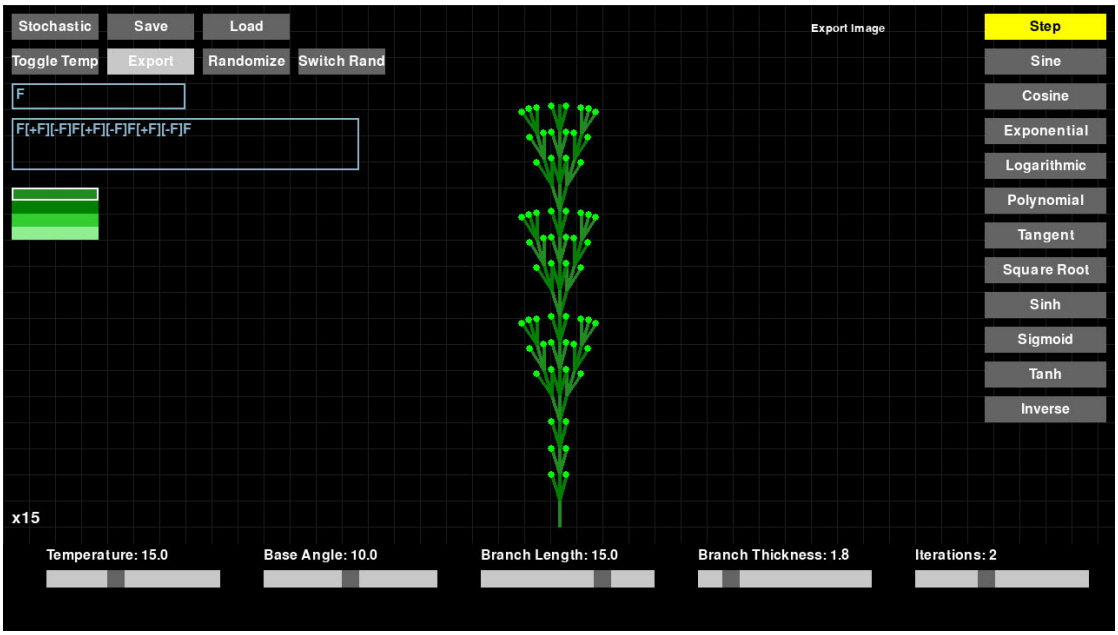


Figure 5.1: Generated L-System using step function branching variations.

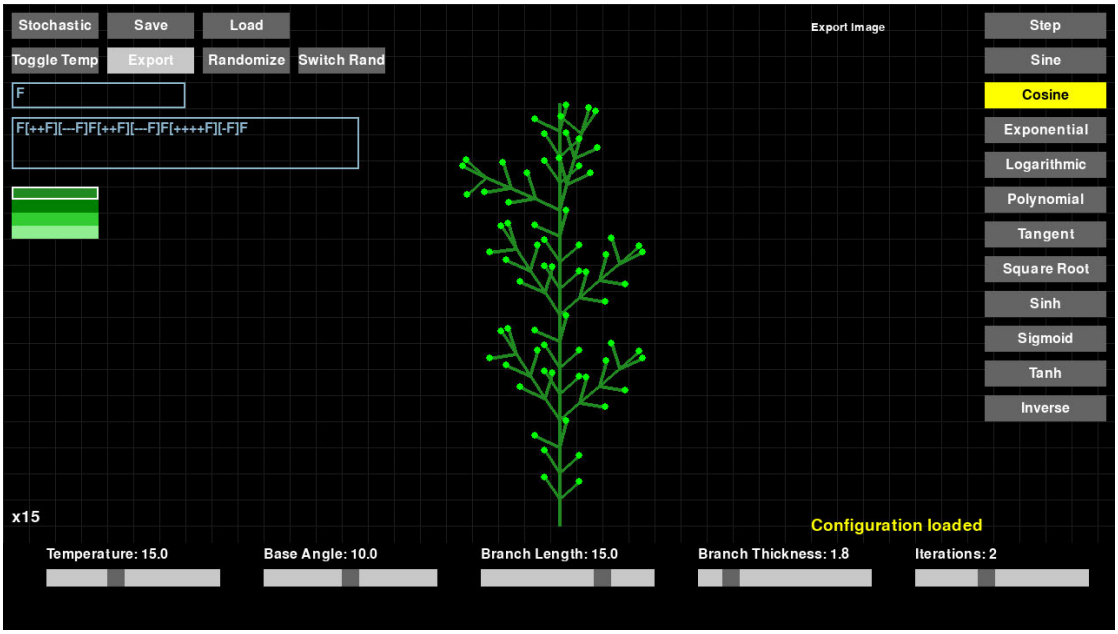


Figure 5.2: Generated L-System using the cosine function branching variations.

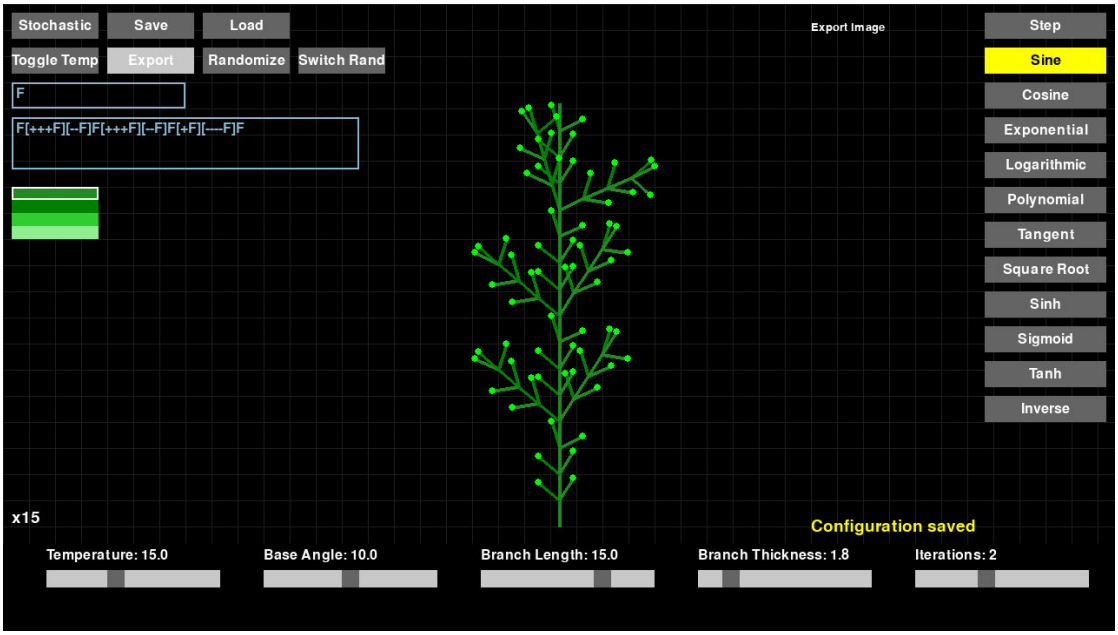


Figure 5.3: Generated L-System using the sine function branching variations.

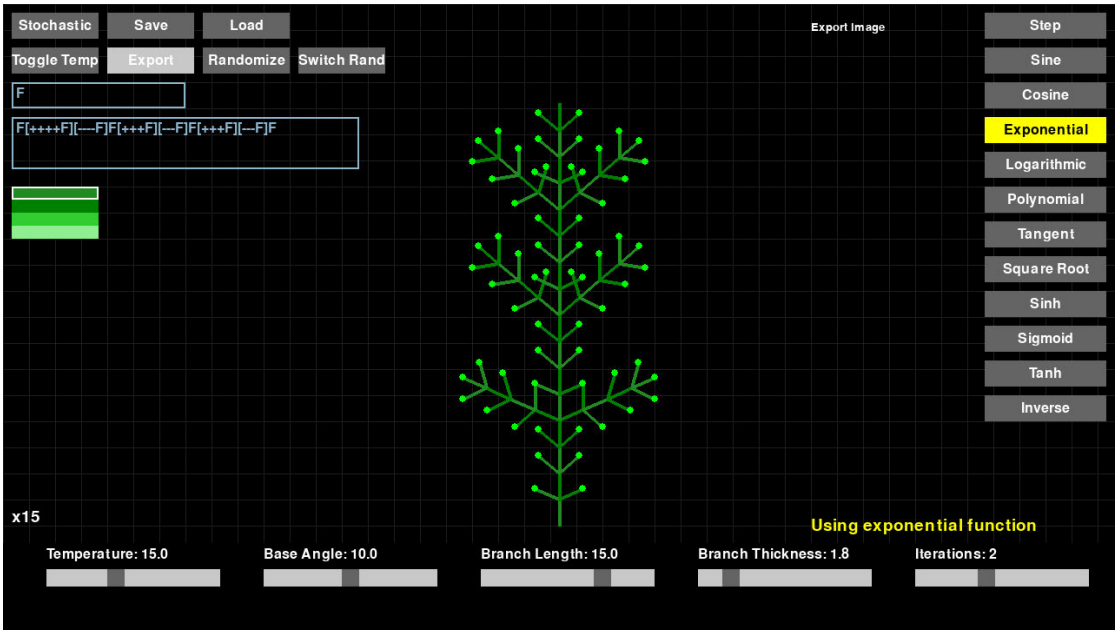


Figure 5.4: Generated L-System using the exponential function branching variations.

It is worth highlighting that the following functions have allowed for a natural and dynamic growth of the plant: step, sine, cosine, exponential, logarithmic, polynomial, sigmoid, hyperbolic tangent and inverse. However, the tangent and hyperbolic sine have exhibited an opposite behaviour, illustrating a non-natural growth pattern. This might be due to the fact that the tangent grows too quickly due to its extreme and rapid variations near its asymptotes. These characteristics lead to highly unpredictable and unstable branching patterns with sharp turns and erratic angles, whereas the hyperbolic sine simply grows too fast and thus leads to the same extreme variations.

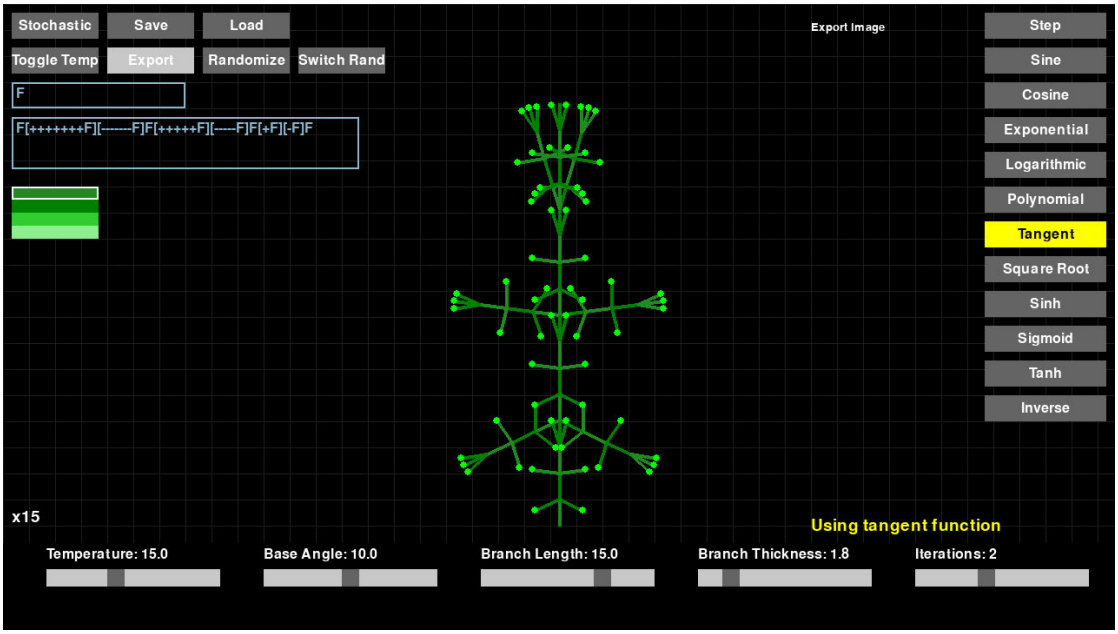


Figure 5.5: Generated L-System using hyperbolic sine function branching variations.

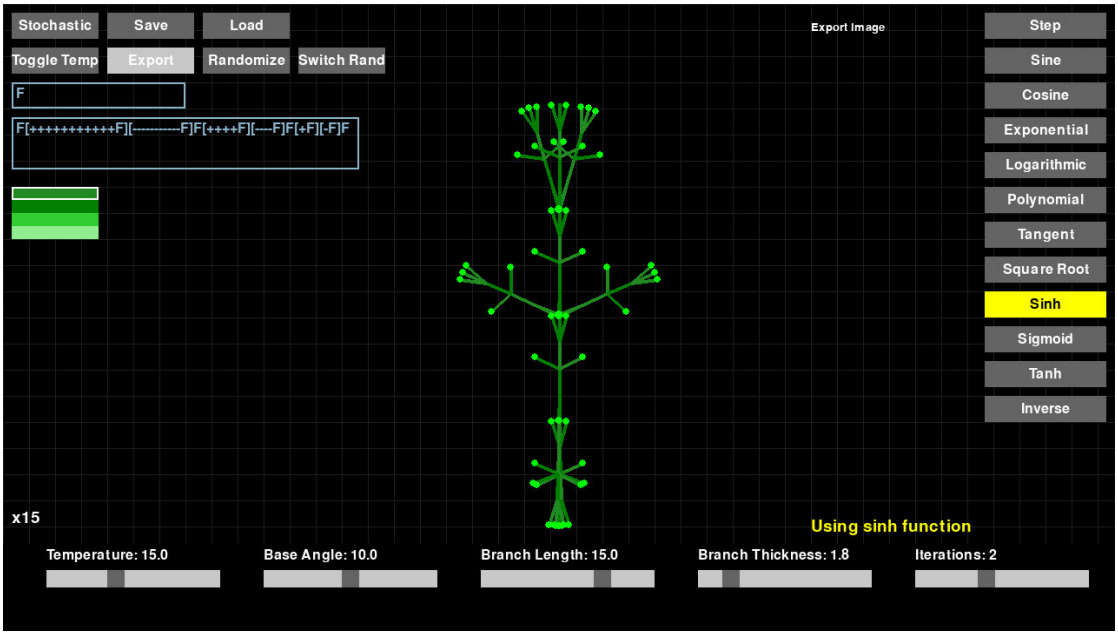


Figure 5.6: Generated L-System using square root function branching variations.

5.5 Stochastic Approach

Furthermore, a function to generate stochastic rules, `generate_stochastic_rule` is implemented. This function introduces randomness into the L-system rules, further enhancing the system's ability to simulate natural and varied growth patterns. This function is essential for adding a layer of unpredictability. By incorporating random choices, the system can produce a wide range of possible growth forms, each unique yet following the underlying rule set. Every variation has a 0.2 chance of being implemented.

5.5.1 Formula

Let p be the probability of replacing F, and V be the set of variations. The rule generation process is:

$$\text{rule} = \begin{cases} \text{random choice from } V & \text{with probability } p, \\ F & \text{with probability } 1 - p. \end{cases}$$

5.5.2 Implemented Code

```
def generate_stochastic_rule(base_rule):
    stochastic_rule = ""
    variations = ["F[+F][-F]", "F[++F][--F]", "F[+F][-F][++F][--F]
        "]
    for char in base_rule:
        if char == "F" and random.random() < 0.2:
            stochastic_rule += random.choice(variations)
        else:
            stochastic_rule += char
    return stochastic_rule
```

Code Snippet 5.8: Generate Stochastic Rule

Initial Formula	Formula after Pressing Stochastic Button
F	F[+F][-F]
F[+F][-F]	F[+F][–F]F
F[++F][–F]	F[++F][-F]F[+F][–F]
F[++F][-F]F[++F][–F]	F[+F][–F]F[++F][-F]F[+F][++F]

Table 5.4: Effect of Pressing the Stochastic Button on the L-System Formula

Initial Formula `F`:

After pressing the stochastic button, `F` is replaced with `F[+F][-F]`. This change introduces branching at the end of the segment represented by `F`.

Initial Formula `F[+F][-F]`:

One occurrence of `F` is replaced, resulting in `F[+F][-F]F`. The replacement introduces a new branch with a double turn (–) to the left.

Initial Formula `F[++F][-F]`:

One occurrence of `F` is replaced, resulting in `F[++F][-F]F[+F][-F]`. This introduces additional branching and variations in angles.

Initial Formula `F[++F][-F]F[++F][-F]`:

Multiple occurrences of `F` are replaced, resulting in `F[+F][-F]F[++F][-F]F[+F][++F]`. This creates a more complex and varied branching structure, mimicking natural growth patterns.

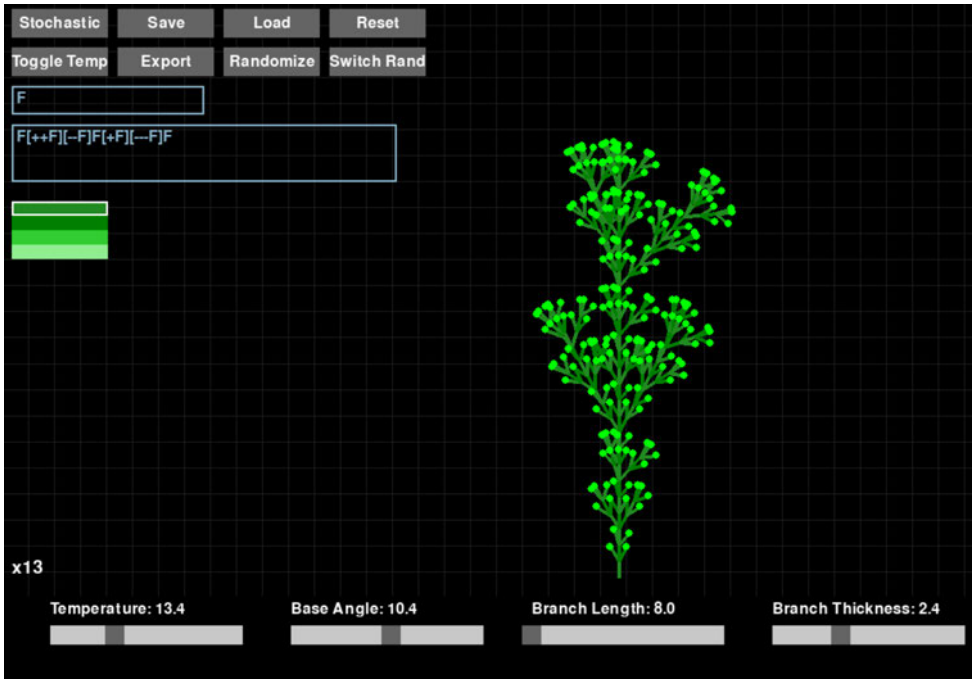


Figure 5.7: Plant generated with random branching attributes without stochastic variations.

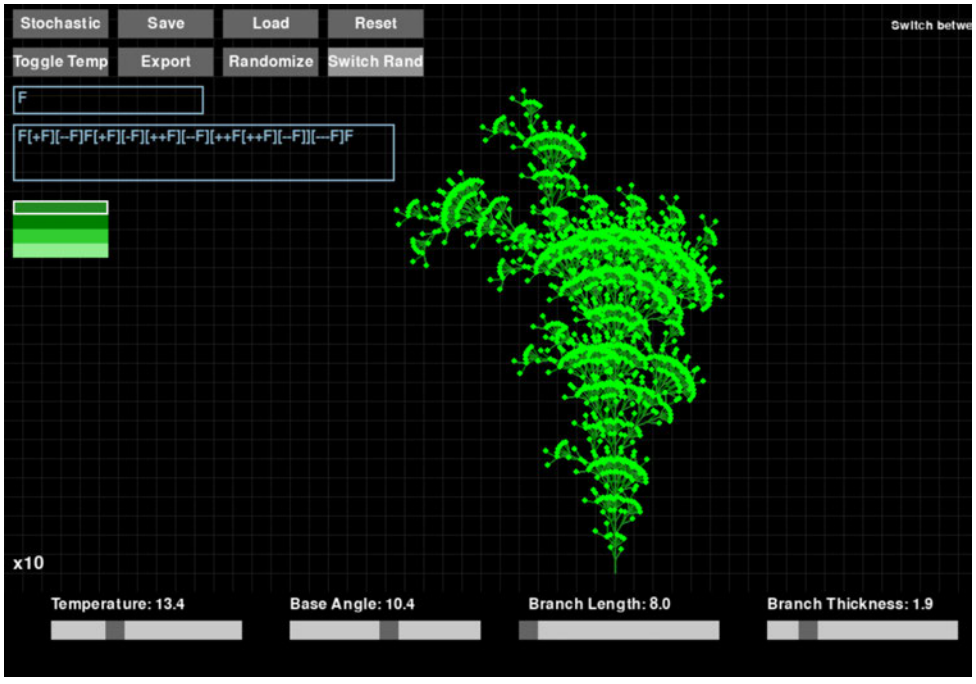


Figure 5.8: Plant generated with random branching attributes and stochastic variation.

In the figures provided above one can witness the nature of the modification that are being applied to the system whenever the stochastic function is being toggled.

5.6 L-System Class Implementation

The `LSystem` class includes the axiom, iterations, and the resulting system string. The class provides methods to generate the L-system string (`generate_system`) and draw it on the Pygame screen (`draw_system`). This class forms the heart of the implementation, managing the generation and rendering of the L-system.

```
class LSystem:
    def __init__(self, axiom, iterations):
        self.axiom = axiom
        self.iterations = iterations
        self.system = axiom
        self.length = 0
        self.angle = 0
        self.thickness = 0
```

Code Snippet 5.9: L-System Class Initialization

5.6.1 Generate the System

Next, the `generate_system` method is implemented. This method creates the L-system string by iterating through the axiom and applying the rules. It also applies context-sensitive rules if specified. This method is fundamental for evolving the L-system from its initial state, ensuring that each iteration reflects the rules and parameters defined. The iterative process is at the core of L-systems, allowing complex patterns to emerge from simple rules through repeated application. This method must efficiently handle large strings and numerous iterations, maintaining performance while generating detailed patterns.

```
    def generate_system(self, rules):
        current_string = self.axiom
        for _ in range(self.iterations):
            next_string = []
            for i, char in enumerate(current_string):
```

```
                context = current_string[i-1:i+2]
                replacement = self.apply_context_rules(context,
                    rules)
                next_string.append(replacement if replacement else
                    rules.get(char, char))
            current_string = ''.join(next_string)
        self.system = current_string
        self.length = rules["length"]
        self.angle = rules["angle"]
        self.thickness = rules["thickness"]
```

Code Snippet 5.10: Generate System

To support context-sensitive rules, the `apply_context_rules` method is implemented. This method checks if the current character matches any context-sensitive rules and applies the replacement if found. This feature allows for more sophisticated and context-aware transformations within the L-system. Context-sensitive rules add a layer of complexity, enabling the system to produce patterns that depend on the local context of each character, thus enhancing the diversity and realism of the generated forms.

```
    def apply_context_rules(self, context, rules):
        context_rules = rules.get("context_rules", [])
        for rule in context_rules:
            if context == rule["context"]:
                return rule["replacement"]
        return None
```

Code Snippet 5.11: Apply Context Rules

5.6.2 Draw the System

Draw Branches

The `draw_system` method is responsible for visualizing the L-system on the Pygame screen. It interprets the L-system string and draws lines representing branches, with variations in thickness and angle. Additionally, it handles the drawing of leaves and logs

the branching data. This visualization is crucial for users to see the results of the L-system generation process in real-time. The method must efficiently render potentially complex and dense structures while maintaining visual clarity.

The function uses a stack to manage the state of the drawing context when dealing with branching. The stack is used to save the current position and angle before a branch is drawn. This allows the system to return to this state after completing a branch, supporting the recursive nature of L-systems and enabling complex branching structures.

```
    def draw_system(self, screen, width, height, length, angle,
        thickness, scale, branch_color, leaf_color, offset_x,
        offset_y):
         stack = []
         x, y = width // 2 + offset_x, height - 120 + offset_y
         current_angle = 90
         leaves = []
         min_thickness = 1
                 for i, char in enumerate(self.system):
             color = branch_color[i % len(branch_color)]
             current_thickness = max(min_thickness, int(thickness *
                 scale))
             if char == "F":
                 x_new = x + np.cos(np.radians(current_angle)) *
                     length * scale
                 y_new = y - np.sin(np.radians(current_angle)) *
                     length * scale
                 pygame.draw.line(screen, color, (x, y), (x_new,
                     y_new), current_thickness)
                 x, y = x_new, y_new
             elif char == "+":
                 current_angle += angle
             elif char == "-":
                 current_angle -= angle
             elif char == "[":
                 stack.append((x, y, current_angle))
             elif char == "]":
                 leaves.append((x, y))
                 x, y, current_angle = stack.pop()
```

Code Snippet 5.12: Draw System

Draw Leaves and Calculate Average Branch Length

The method continues by drawing circles for leaves and calculating the average branch length. Logging the branching data for analysis is also included, which can be useful for understanding the growth patterns and optimizing the L-system parameters. By analyzing the logged data, developers can fine-tune the rules and parameters to achieve desired outcomes, making the system more versatile and effective for various applications.

```
        for (lx, ly) in leaves:
            pygame.draw.circle(screen, leaf_color, (int(lx), int(
                ly)), max(1, int(3 * scale)))

        avg_branch_length = (sum(np.linalg.norm([lx - width // 2,
            ly - (height - 120)]) for (lx, ly) in leaves) / len(
            leaves)) if leaves else 0

        self.log_branching_data(len(leaves), avg_branch_length,
            leaves)

        return len(leaves), avg_branch_length

    def log_branching_data(self, num_branches, avg_branch_length,
        leaves):
        branch_lengths = [np.linalg.norm([lx - width // 2, ly - (
            height - 120)]) for (lx, ly) in leaves]
        logging.info(f"Number of branches: {num_branches}, Average
             branch length: {avg_branch_length:.2f}")
        logging.info(f"Branch lengths: {branch_lengths}")
```

Code Snippet 5.13: Draw System Final

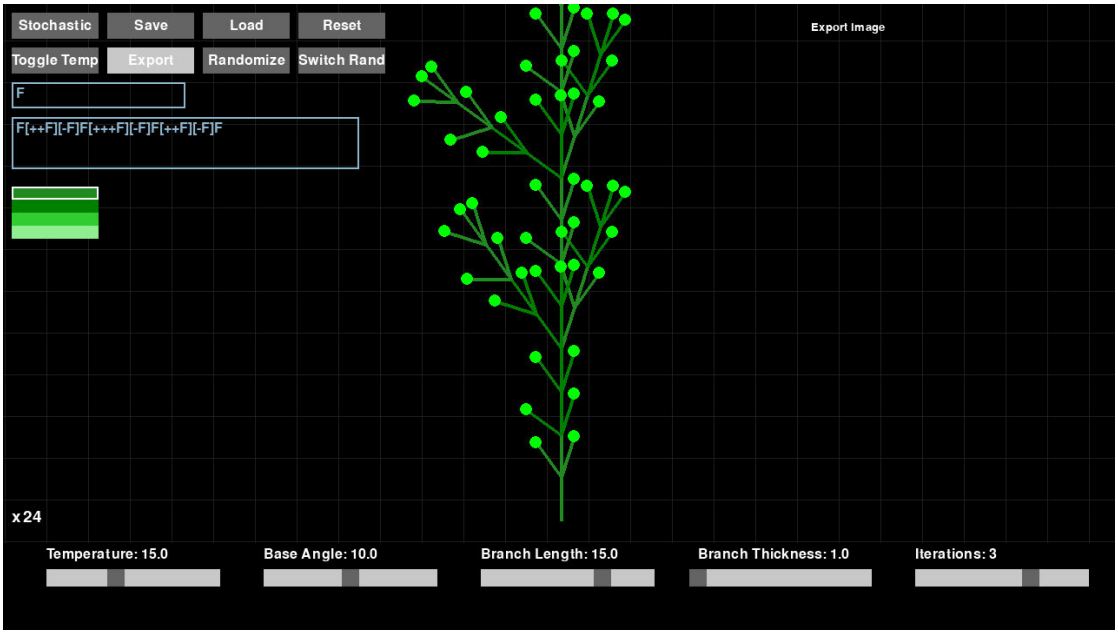


Figure 5.9: Generated leaves on the plant branches(green dots).

5.7 User Interface Elements

The implementation includes several UI elements to facilitate user interaction: `Slider`, `Button`, `TextBox`, and `ColorPicker` classes. These classes manage the interactive components, allowing users to manipulate the parameters dynamically. User interface elements are critical for creating an intuitive and engaging experience, enabling users to explore the effects of different parameters on the L-system's growth patterns. Users can experiment with various configurations and observe the resulting changes in real-time.

5.7.1 Sliders

A slider control for adjusting parameters is created first. The `Slider` class handles mouse events and updates its value accordingly. The slider is a key element in the user interface, enabling real-time adjustments of parameters such as temperature, angle, and length. Sliders provide a smooth and intuitive way for users to fine-tune parameters, ensuring precise control over the simulation settings.

```
class Slider:
    def __init__(self, x, y, w, h, min_val, max_val, init_val,
        label, integer=False):
        self.rect = pygame.Rect(x, y, w, h)
        self.min_val = min_val
        self.max_val = max_val
        self.value = init_val
        self.handle_rect = pygame.Rect(x, y, w // 10, h)
        self.handle_rect.centerx = x + (w * ((init_val - min_val)
            / (max_val - min_val)))
        self.dragging = False
        self.label = label
        self.font = pygame.font.Font(None, 24)
        self.integer = integer
```

Code Snippet 5.14: Slider Initialization

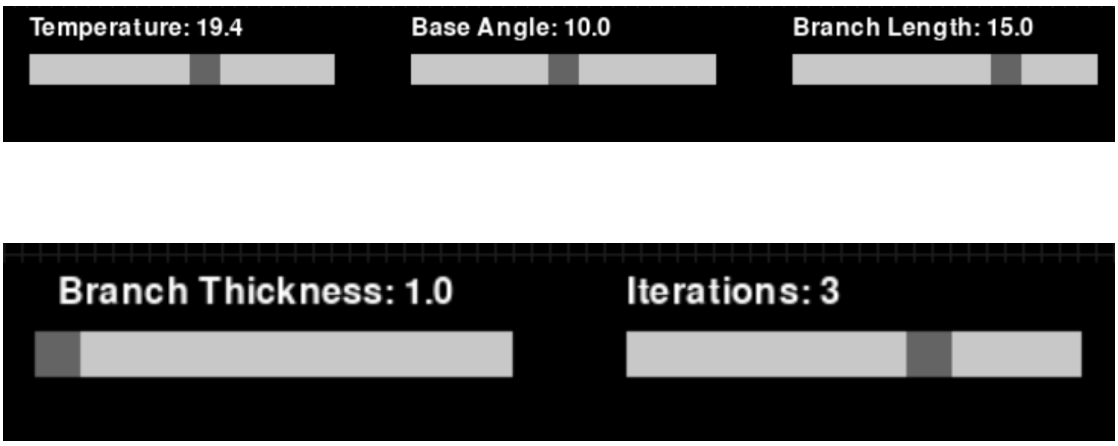


Figure 5.10: Available sliders for the simulation.

The **Angle Slider** controls the base angle of the branches in the L-System. This slider ranges from 0 to 20 degrees. The base angle determines the initial divergence of the branches from the main trunk. Adjusting this slider changes the angular spread of the branches.

The **Length Slider** controls the length of each branch in the L-System. The length value ranges from 8 to 18 units. Adjusting this slider changes the length of the branches, making them longer or shorter.

The **Thickness Slider** controls the thickness of the branches in the L-System. This slider ranges from 1 to 5 units. The thickness affects the visual weight of the branches, making them appear thicker or thinner.

The **Iterations Slider** controls the number of iterations the L-System goes through to generate the final structure. It ranges from 1 to 4 iterations. Each iteration applies the production rules to the current state of the system, increasing the complexity and detail of the L-System.

5.7.2 Buttons

A clickable button that changes its color on hover and click, providing visual feedback to the user, is also implemented. Buttons are used for various actions such as saving configurations, loading configurations, and resetting parameters. Visual feedback, such as color changes, helps users understand the state of the button, confirming that their input has been registered.

```
class Button:
    def __init__(self, x, y, w, h, text, color, hover_color,
        click_color, tooltip=None, selectable=True):
        self.rect = pygame.Rect(x, y, w, h)
        self.text = text
        self.color = color
        self.hover_color = hover_color
        self.click_color = click_color
        self.selected_color = (255, 255, 0)
        self.current_color = color
        self.font = pygame.font.Font(None, 24)
        self.text_surface = self.font.render(text, True, (255,
            255, 255))
        self.text_rect = self.text_surface.get_rect(center=self.
            rect.center)
        self.clicked = False
        self.selected = False
        self.tooltip = tooltip
        self.tooltip_font = pygame.font.Font(None, 20)
        self.selectable = selectable
```

Code Snippet 5.15: Button Initialization

Figure 5.11: Available buttons for the simulation.

The **Stochastic Button** toggles the application of stochastic rules to the L-System. When activated, it introduces random variations to the production rules, resulting in more varied and less predictable patterns.

The **Save Button** saves the current configuration of the L-System parameters to a file. This includes the axiom, iterations, temperature, base angle, branch length, branch thickness, and the current rules.

The **Load Button** loads a previously saved configuration from a file. This restores the L-System parameters and rules to the saved state, allowing continuation from the previous point.

The **Reset Button** resets the stochastic variations, returning the L-System to its deterministic state based on the current parameters and rules.

The **Toggle Temperature Button** switches between using a fixed temperature and a real-time temperature that changes dynamically over time. This simulates environmental changes affecting the L-System growth.

The **Export Button** exports the current visualization of the L-System as an image file. This allows the generated patterns to be saved and shared.

The **Randomize Button** randomizes the L-System parameters within their respective ranges. This provides a quick way to explore different configurations and observe their effects on the L-System.

The **Switch Random Button** toggles between random and deterministic behavior for the L-System. In random mode, parameters and rules are varied randomly, while in deterministic mode, they follow a fixed pattern based on the selected deterministic function.

5.7.3 Text Boxes

A text input box for defining L-system axioms and rules is also necessary, supporting basic text editing operations. This component is crucial for users to input and modify the initial conditions and rules of the L-system dynamically. Text input boxes allow users to enter and edit text-based data, providing a way to directly influence the behavior and structure of the L-system.

```
class TextBox:
    def __init__(self, x, y, w, h, text=''):
        self.rect = pygame.Rect(x, y, w, h)
        self.color_inactive = pygame.Color('lightskyblue3')
        self.color_active = pygame.Color('dodgerblue2')
        self.color = self.color_inactive
        self.text = text
        self.font = pygame.font.Font(None, 24)
        self.txt_surface = self.font.render(text, True, self.color
            )
        self.active = False
        self.caret_visible = True
        self.caret_position = len(text)
        self.caret_timer = pygame.time.get_ticks()
```

Code Snippet 5.16: TextBox Initialization

Below, there are two figures illustrating the ability to dynamically modifying the formula of the system via the textbox as such:

$$F[+F][–F]F[++F][–F]F[++F][-F]F \rightarrow F[+F][–F]F[++F][–F]F[++F]$$

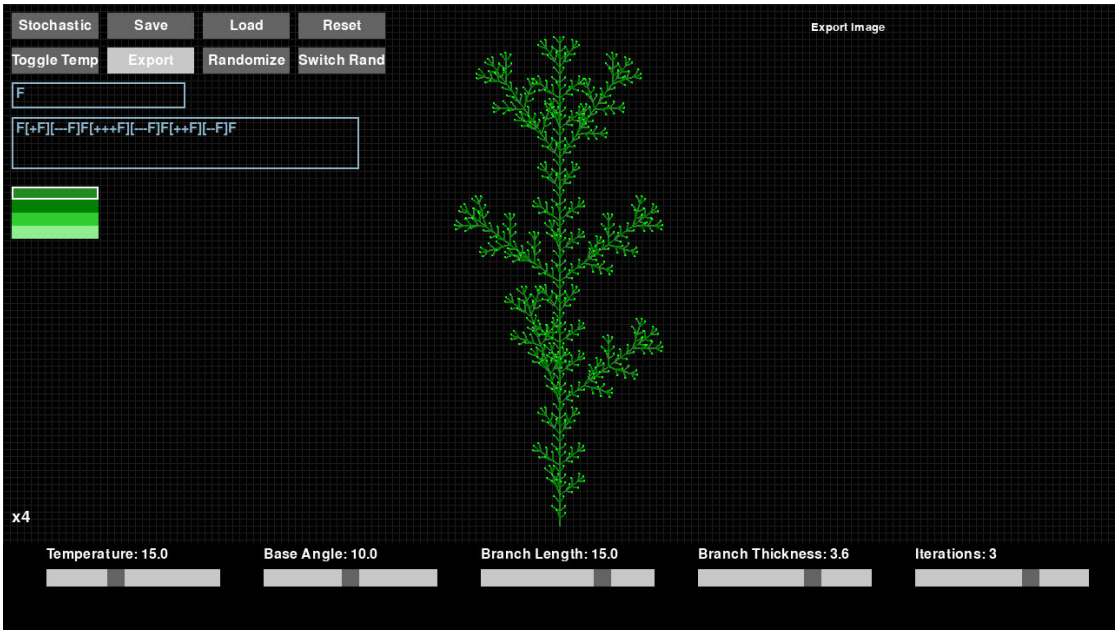


Figure 5.12: Textbox: F[+F][–F]F[++F][–F]F[++F][-F]F

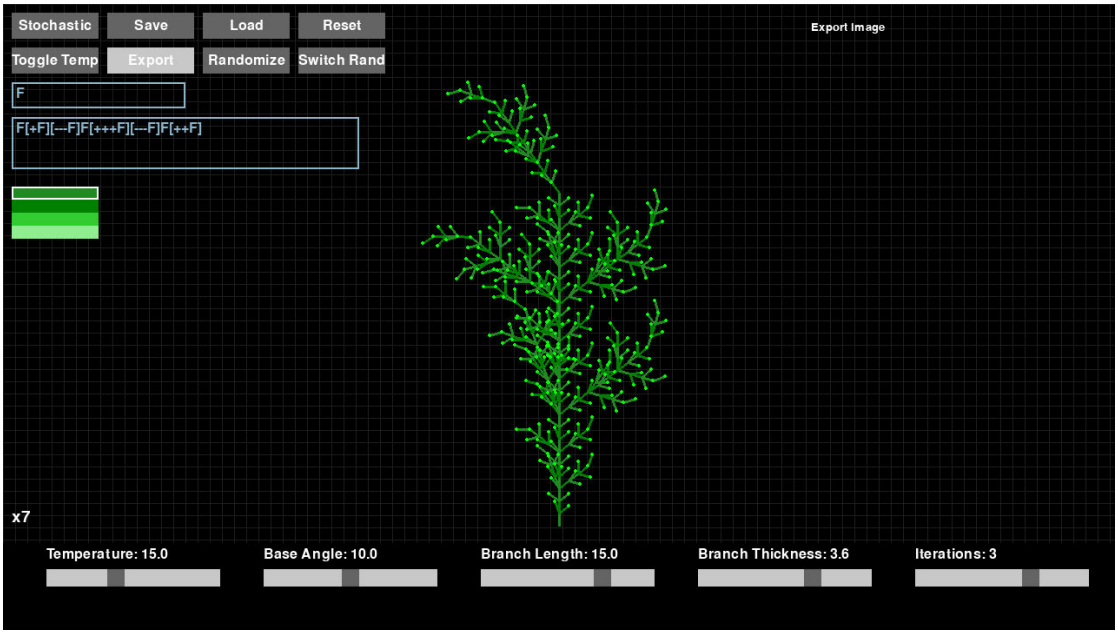


Figure 5.13: Textbox: F[+F][–F]F[++F][–F]F[++F]

The `handle_event` method processes mouse and keyboard events for text input. This functionality enables users to define and adjust the L-system's parameters directly, pro-

viding a hands-on way to experiment with different configurations. Handling keyboard and mouse events involves detecting clicks to activate the text box, managing caret position, and processing text input.

The `blink_caret` method toggles the visibility of the caret. This functionality provides a visual cue that the text box is active and ready for input. This visual feedback is essential for maintaining a responsive and user-friendly text input interface.

5.7.4 Color Picker

A color picker is implemented to display a palette of colors for the user to select the color for branches and leaves. This component enhances the visual customization of the L-system, allowing users to choose colors that best represent their desired growth patterns.

```
class ColorPicker:
    def __init__(self, x, y, w, h, initial_colors, tooltip=None):
        self.rect = pygame.Rect(x, y, w, h)
        self.colors = initial_colors
        self.selected_color_index = 0
        self.font = pygame.font.Font(None, 24)
        self.tooltip = tooltip
        self.tooltip_font = pygame.font.Font(None, 20)
```

Code Snippet 5.17: ColorPicker Initialization

5.8 Helper Functions

Several helper functions support the core functionality, including drawing the grid, updating the L-system, fetching real-time temperature, and managing configurations. These functions enhance the overall usability and functionality of the system, making it more robust and user-friendly. The implementation becomes more organized and maintainable.

5.8.1 Draw the Grid

The `draw_grid` function is first implemented to draw a reference grid on the Pygame screen. The grid provides a visual reference for users, helping them understand the scale and orientation of the L-system. Drawing a grid helps users gauge distances and angles more accurately, providing a useful context for the L-system's growth patterns. The grid must be rendered efficiently to avoid performance issues, especially when dealing with large or complex grids.

```
def draw_grid(screen, width, height, grid_size, offset_x, offset_y
    , color=(30, 30, 30)):
    if grid_size <= 0:
        return
    for x in range(offset_x % grid_size, width, grid_size):
        pygame.draw.line(screen, color, (x, 0), (x, height))
    for y in range(offset_y % grid_size, height, grid_size):
        pygame.draw.line(screen, color, (0, y), (width, y))
```

Code Snippet 5.18: Draw Grid

5.8.2 Update L-System

The `update_lsystem` function is implemented next to update the L-system drawing based on the current rules and parameters. This function is crucial for rendering the L-system in real-time, reflecting any changes made by the user through the UI elements. Updating the L-system dynamically allows users to see the effects of their adjustments immediately, providing a more interactive and engaging experience. The function must efficiently handle updates to the L-system, ensuring that the rendering process is smooth and responsive.

```
def update_lsystem(rules, screen, width, height, scale,
    branch_color, leaf_color, background_color, offset_x, offset_y)
    :
    screen.fill(background_color, (0, 0, width, height - 100))
    draw_grid(screen, width, height - 100, int(20 * scale),
        offset_x, offset_y, (30, 30, 30))
    try:
        lsystem.generate_system(rules)
        log_growth(rules)
```

```
7           num_branches, avg_branch_length = lsystem.draw_system(
                screen, width, height, rules["length"], rules["angle"],
                 rules["thickness"], scale, branch_color, leaf_color,
                offset_x, offset_y)
8           log_branching_complexity(num_branches, avg_branch_length)
9       except Exception as e:
10          logging.error(f"Error updating L-system: {e}")
```

Code Snippet 5.19: Update L-System

5.8.3 Fetch Real-Time Temperature

To simulate fetching a real-time temperature, a helper function is implemented. This function is used to dynamically adjust the L-system parameters based on changing environmental conditions. Simulating real-time temperature data adds an element of realism to the L-system, making it responsive to external factors. The function must generate temperature values that vary over time, providing a dynamic input for the L-system's growth simulation.

```
1   def fetch_real_time_temperature():
2       base_temp = 15
3       noise = np.sin(datetime.datetime.now().timestamp() / 3600) * 5
4       temperature = base_temp + noise + 15 * np.sin(datetime.
            datetime.now().timestamp() / 3600)
5       return np.clip(temperature, 0, 30)
```

Code Snippet 5.20: Fetch Real-Time Temperature

5.8.4 Save/Load

Functions to save and load L-system configurations to/from a JSON file are also necessary. These functions enable users to save their current setup and reload it later, facilitating experimentation and iterative development. Saving and loading configurations allow users to preserve their work and continue from where they left off. The functions must handle the serialization and deserialization of the L-system's state, ensuring that all relevant parameters are accurately saved and restored.

```
def save_configuration(filename):
    configuration = {
        'axiom': lsystem.axiom,
        'iterations': lsystem.iterations,
        'temperature': temperature_slider.get_value(),
        'base_angle': angle_slider.get_value(),
        'branch_length': length_slider.get_value(),
        'branch_thickness': thickness_slider.get_value(),
        'rules': {'F': rule_box_f.text}
    }
    with open(filename, 'w') as file:
        json.dump(configuration, file)
    logging.info(f"Configuration saved to {filename}")
```

Code Snippet 5.21: Save Configuration

5.8.5 Export

An export function is also required to save the current screen image to a file using Tkinter's file dialog. This functionality allows users to save a visual representation of the L-system, which can be useful for documentation or further analysis. The function must handle the file dialog interface and ensure that the image is saved correctly .

```
def export_image(screen):
    Tk().withdraw()
    export_path = filedialog.asksaveasfilename(defaultextension=".
       png", filetypes=[("PNG files", "*.png")])
    if export_path:
        pygame.image.save(screen, export_path)
        logging.info(f"Image exported to {export_path}")
```

Code Snippet 5.22: Export Image

5.8.6 Randomize

Lastly, a function to randomize the values of the sliders is implemented to introduce variability. This function is useful for testing different configurations quickly and explor-

ing the effects of various parameter combinations. Randomizing parameters allows users to explore a wide range of growth patterns without manually adjusting each slider.

```
def randomize_parameters():
    global temperature_slider, angle_slider, length_slider,
        thickness_slider, iterations_slider
    temperature_slider.value = random.uniform(temperature_slider.
        min_val, temperature_slider.max_val)
    angle_slider.value = random.uniform(angle_slider.min_val,
        angle_slider.max_val)
    length_slider.value = random.uniform(length_slider.min_val,
        length_slider.max_val)
    thickness_slider.value = random.uniform(thickness_slider.
        min_val, thickness_slider.max_val)
    iterations_slider.value = random.randint(iterations_slider.
        min_val, iterations_slider.max_val)
```

Code Snippet 5.23: Randomize Parameters

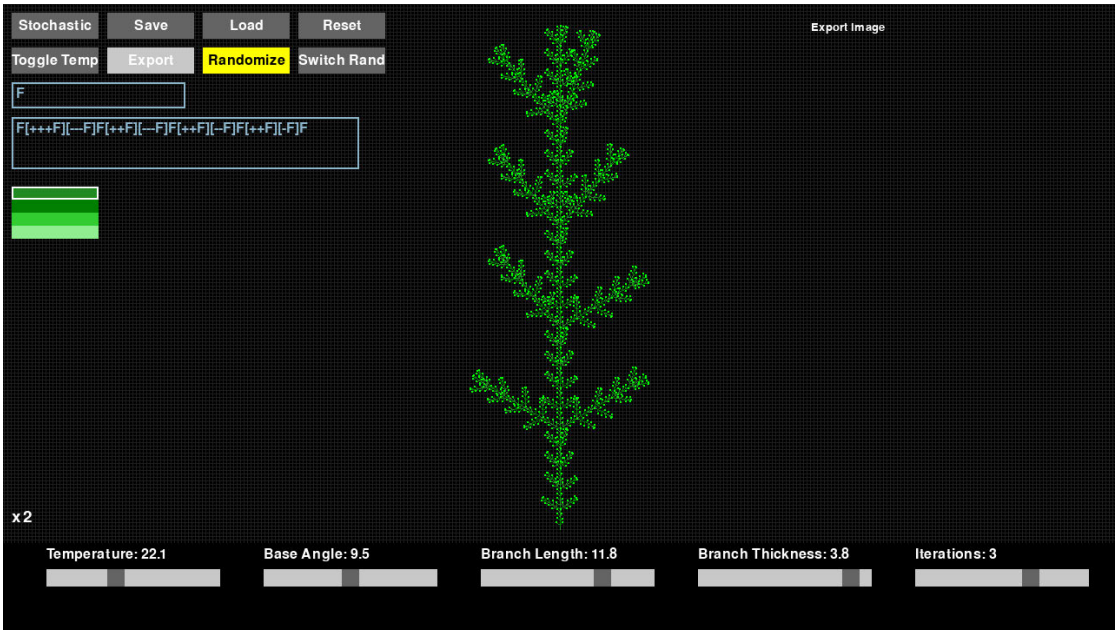


Figure 5.14: First generation of an L-System with randomized values.

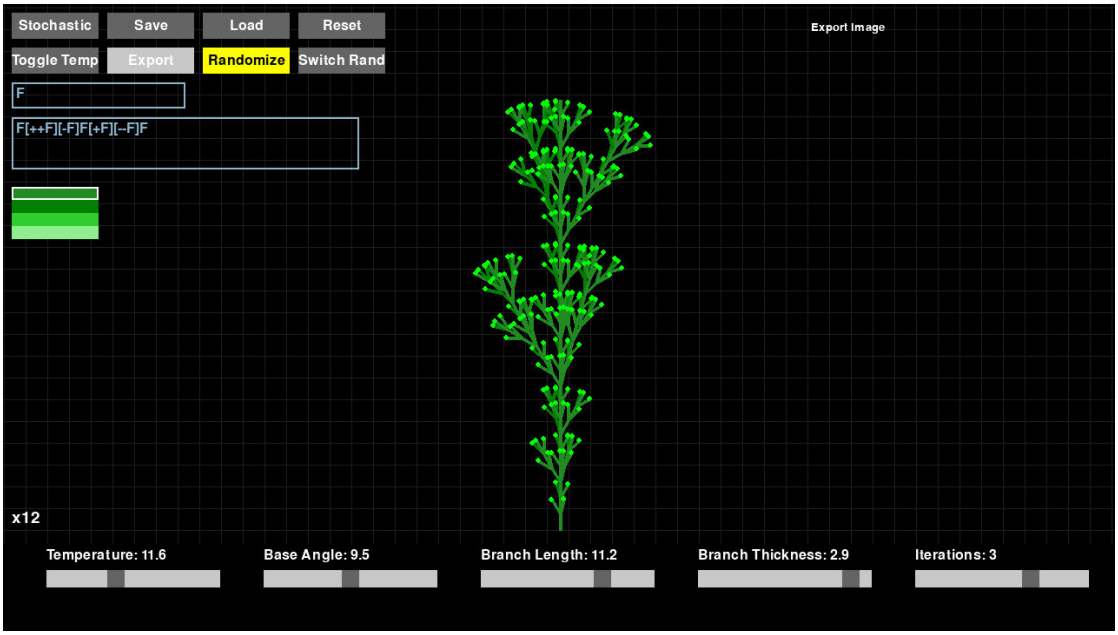


Figure 5.15: Second generation of an L-System with randomized values.

6 Results and Evaluation

This chapter conducts a thorough assessment of the system's performance after its implementation. It focuses on evaluating how accurately the implemented functionalities correspond to the objectives established before implementation. This analysis will highlight significant accomplishments and identify critical areas where improvements can be made.

6.1 Overview

The system successfully met its primary objectives, demonstrating the integration of fractals and Lindenmayer Systems (L-systems) with data inputs to create responsive natural patterns. Several key results and observations were noted and divided into pros and cons:

6.1.1 Pros

Dynamic Adaptation: The system effectively adjusted growth angles and branching patterns in real-time based on simulated environmental data. This capability significantly enhanced the realism and flexibility of the L-system models. The dynamic nature of the system allowed it to respond to varying conditions.

User Interaction: The inclusion of interactive user interface elements, such as sliders and buttons, allowed for real-time manipulation of parameters. This feature provided immediate visual feedback, facilitating user engagement and experimentation.

Diverse Pattern Generation: Incorporating both stochastic and deterministic rule variations enabled the system to generate a wide range of natural patterns. Stochastic rules introduced randomness, mimicking the inherent unpredictability of natural growth,

while deterministic rules ensured repeatable and predictable outcomes. This diversity added depth to the simulations and demonstrated the robustness of the approach.

Visualization and Analysis: The use of graphical libraries like Pygame and Turtle for visualization proved effective. These libraries provided a robust platform for rendering complex L-system structures, ensuring smooth and detailed graphics. The logging mechanisms, implemented to track system parameters and outputs, provided valuable insights into the branching complexity and overall system behavior. This data was crucial for analyzing the effectiveness of different rule sets and for optimizing the L-system parameters.

6.1.2 Cons

Predefined Mathematical Functions: The reliance on predefined mathematical functions for deterministic variations constrained the diversity of patterns that could be generated. While these functions (e.g., sine, cosine, exponential) were effective for certain patterns, they limited the system's ability to explore more complex or less predictable structures. This limitation highlighted the need for more flexible and sophisticated rule generation methods that can dynamically adapt based on broader criteria.

Basic Randomization: While the randomization approach introduced variability, it could benefit from more advanced algorithms to enhance the realism and complexity of the generated patterns. The current implementation of randomization was relatively simple, primarily using uniform distributions to alter parameters. Advanced stochastic models, such as Gaussian processes or Markov chains, could provide more nuanced variations that better mimic natural phenomena.

2D Model: The thesis primarily focuses on the use of fractals and L-systems for generating 2D visualizations of natural patterns. While 2D representations are useful for certain types of analysis and visualization, they inherently limit the depth and realism that can be conveyed, especially when representing complex, three-dimensional structures like trees and plants. This dimensional limitation might restrict the application's effectiveness in fields that require more comprehensive spatial analysis and interpretation, such as ecological research, urban planning, and realistic 3D simulations in gaming or virtual reality environments.

Limited Error Handling: While the code includes basic logging and error handling, it may not be robust enough to manage all potential errors that could occur, especially those related to user input or external data integration. This could lead to unhandled exceptions that crash the application or cause unintended behavior, affecting reliability.

Python Limitations: The use of Python for complex simulations that involve graphical output and real-time data handling can introduce performance bottlenecks. Python, while versatile and easy to use, may not handle large-scale simulations as efficiently as more performance-optimized languages like C++ or Java, especially when high frame rates or real-time interactions are required.

Scalability Concerns: As the complexity or scale of the L-system increases, the recursive nature of the system generation and the real-time update of parameters might lead to scalability issues. The system's ability to handle larger or more intricate L-systems without a degradation in performance could be limited, particularly on less powerful hardware.

Limited Scope of Generated Patterns: The current implementation of the L-system primarily focuses on generating patterns that mimic natural phenomena, specifically trees and plants. This focus, while beneficial for targeted studies, significantly limits the potential of L-systems to explore a broader range of patterns and structures. L-systems are capable of generating an extensive variety of both realistic and abstract forms, including intricate architectural designs, fractal landscapes, and complex organic structures.

6.2 Variation Testing

6.2.1 Iterations

Random Pattern

Figure 6.1: One iteration with random generation pattern at a temperature of 16°C.

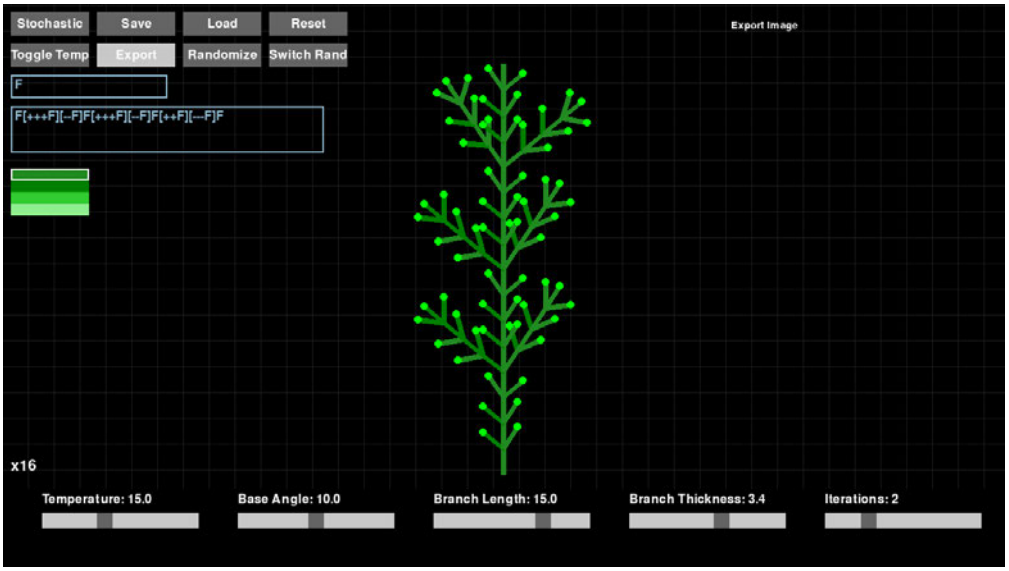


Figure 6.2: Two iterations with random generation pattern at a temperature of 16°C.

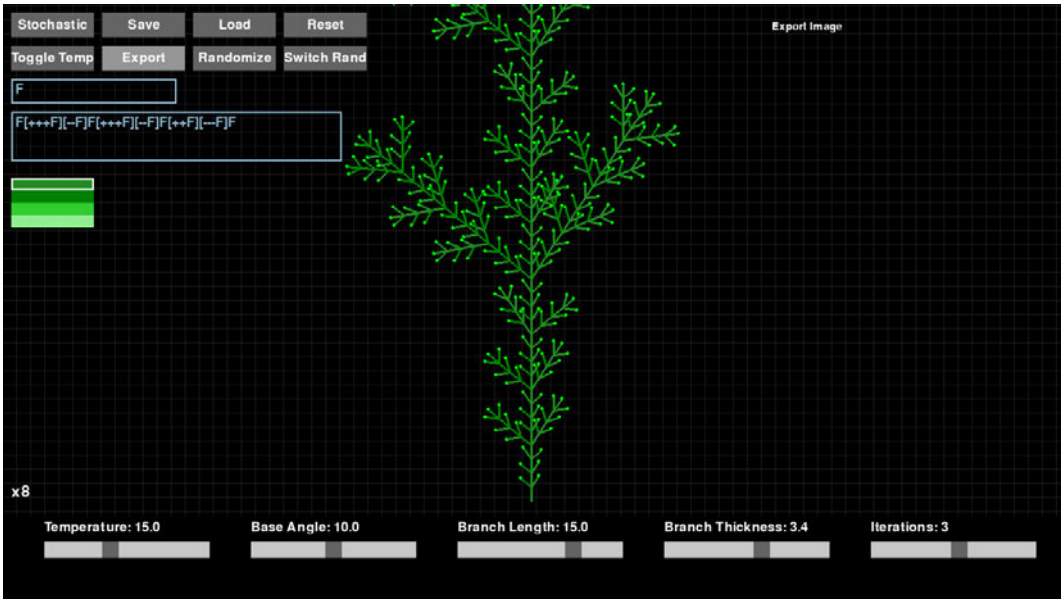


Figure 6.3: Three iterations with random generation pattern at a temperature of 16°C.

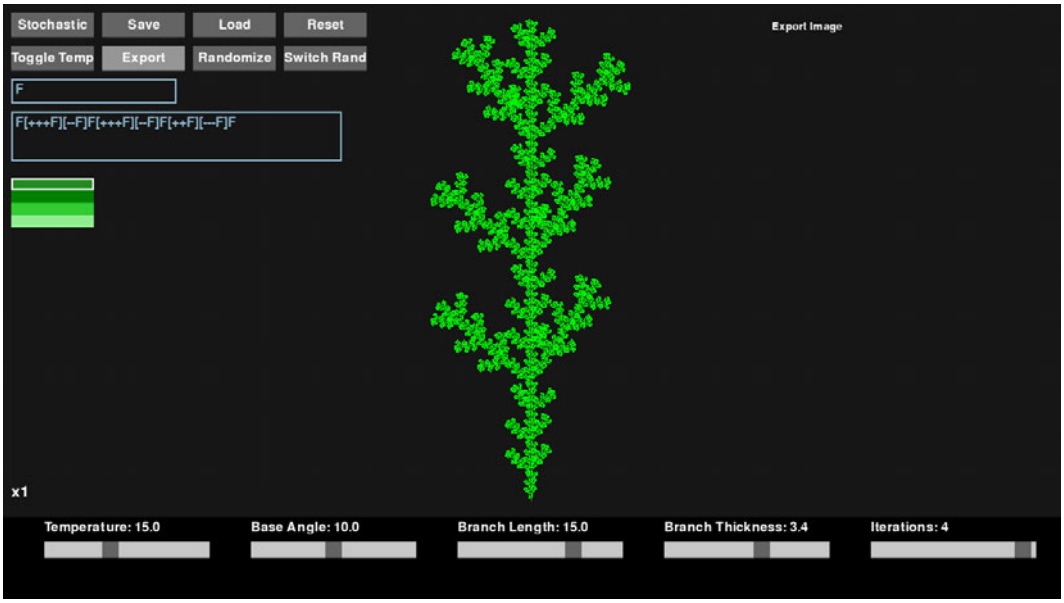


Figure 6.4: Four iterations with random generation pattern at a temperature of 16°C.

Sigmoid Function Pattern

Figure 6.5: One iteration with sigmoid function generation pattern at a temperature of 16°C.

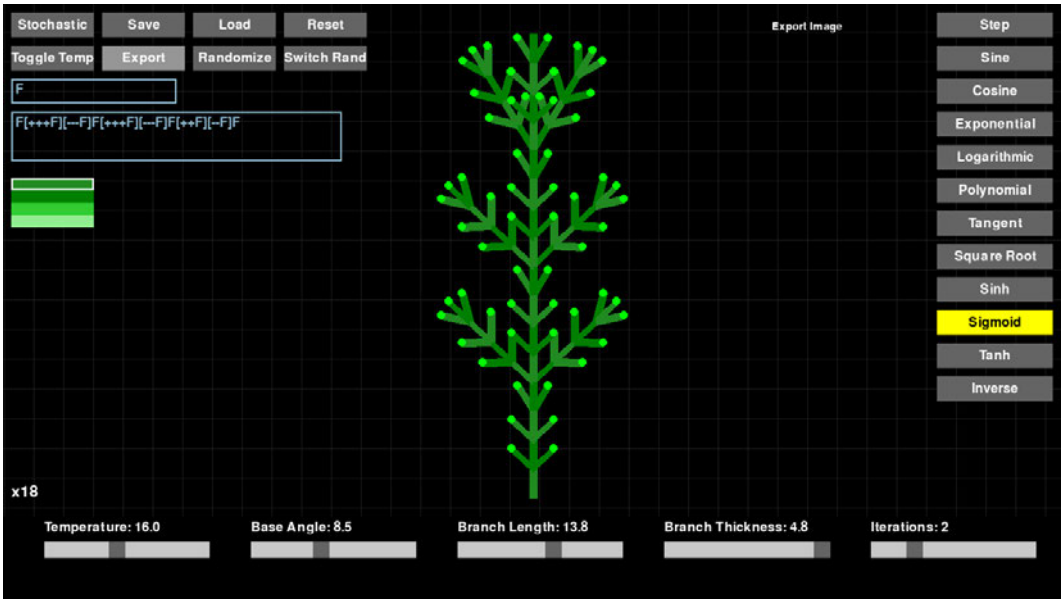


Figure 6.6: Two iterations with sigmoid function generation pattern at a temperature of 16°C.

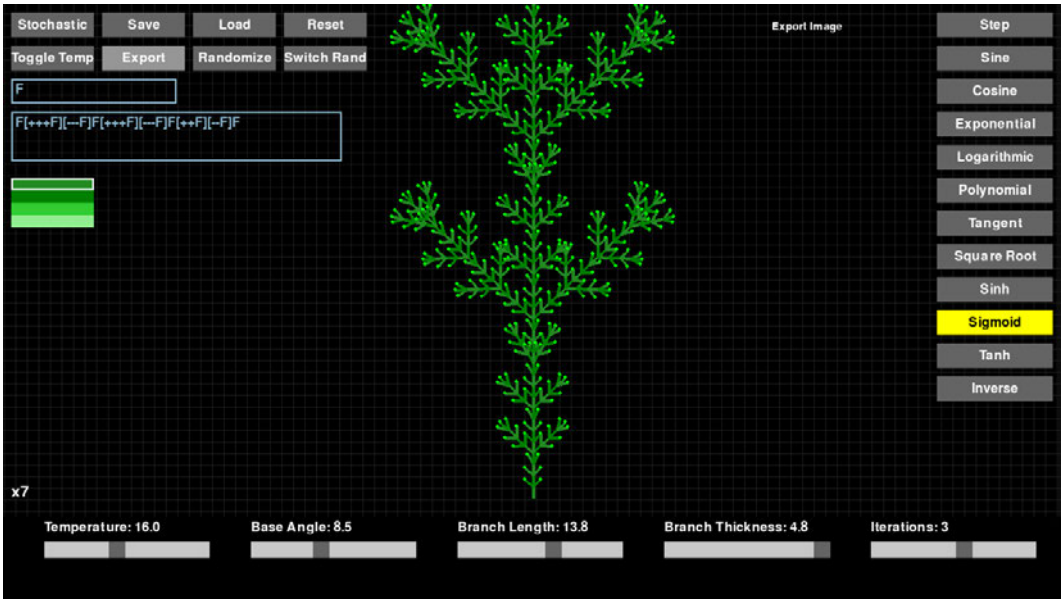


Figure 6.7: Three iterations with sigmoid function generation pattern at a temperature of 16°C.

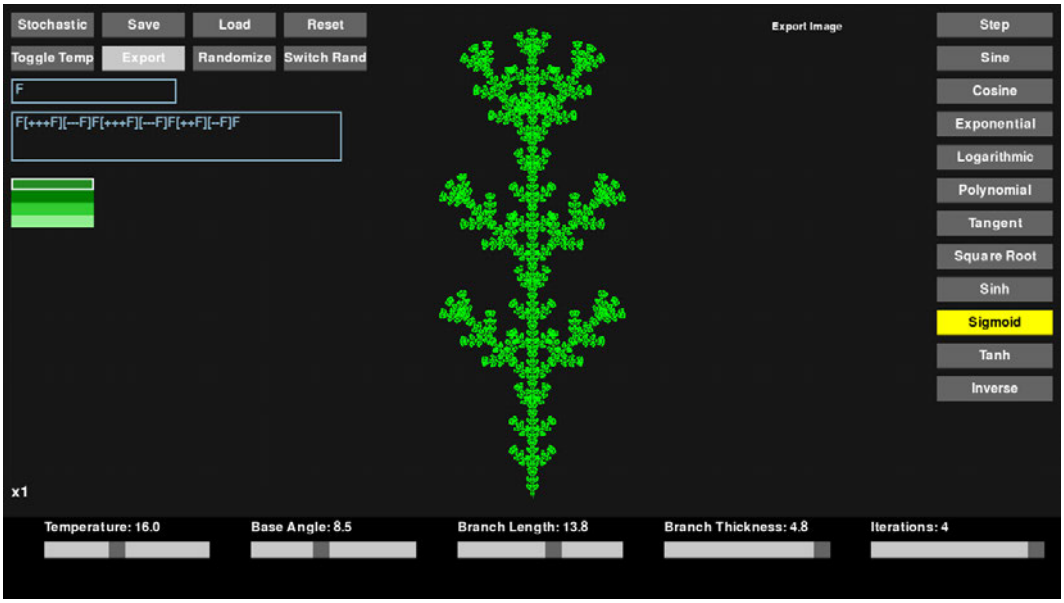


Figure 6.8: Four iterations with sigmoid function generation pattern at a temperature of 16°C.

6.2.2 Temperature

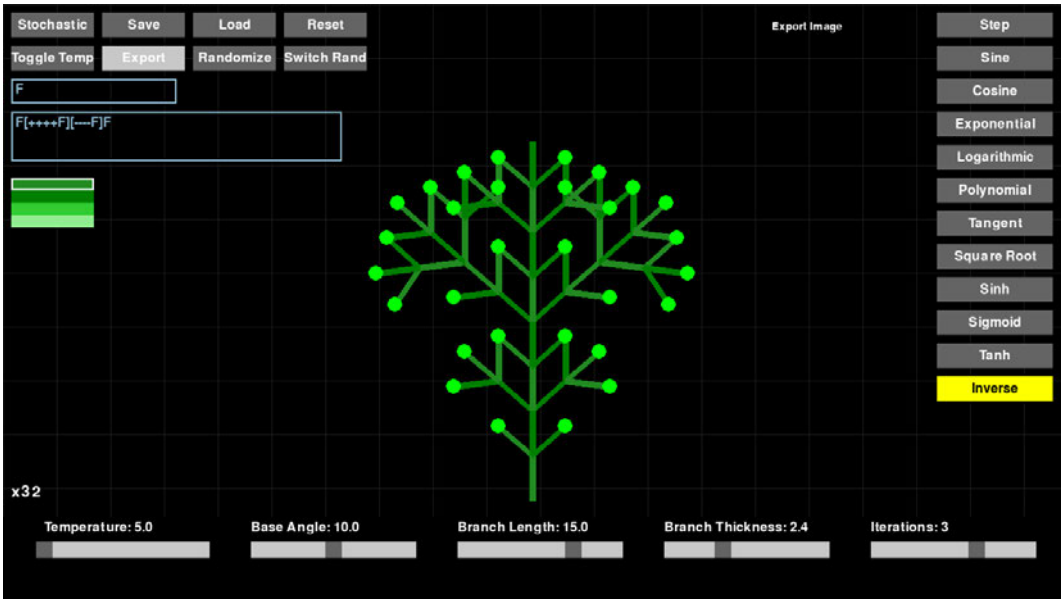


Figure 6.9: Inverse function with 3 iterations at temperature 5.0°C.

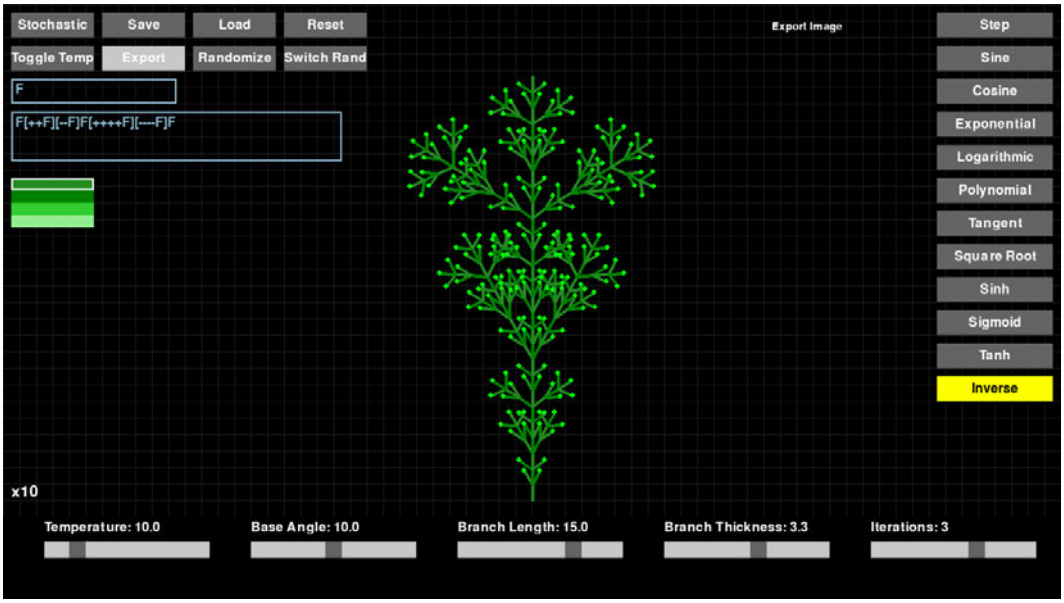


Figure 6.10: Inverse function with 3 iterations at temperature 10.0°C.

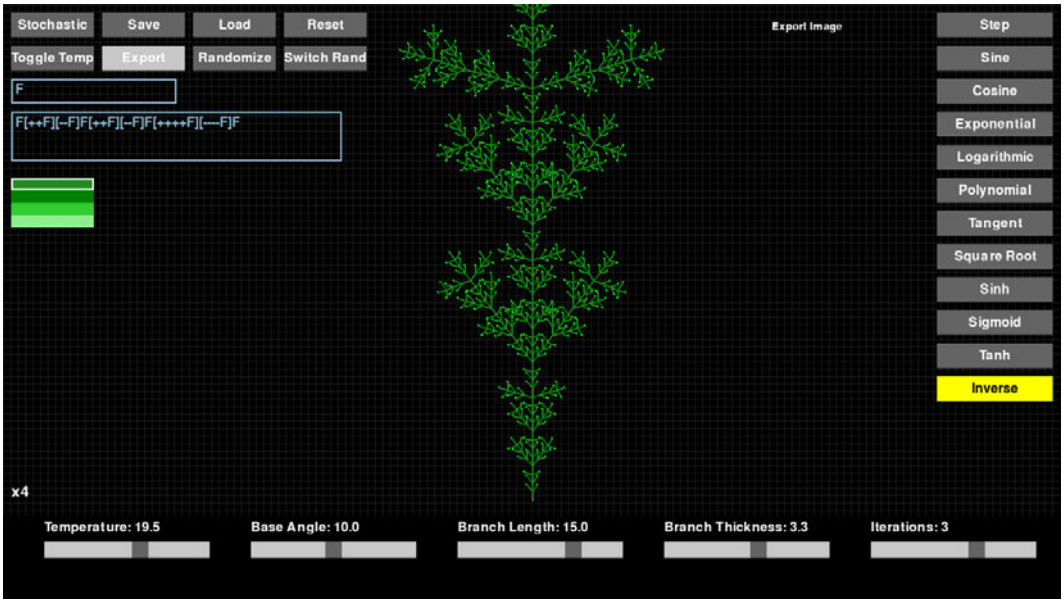


Figure 6.11: Inverse function with 3 iterations at temperature 19.5°C.

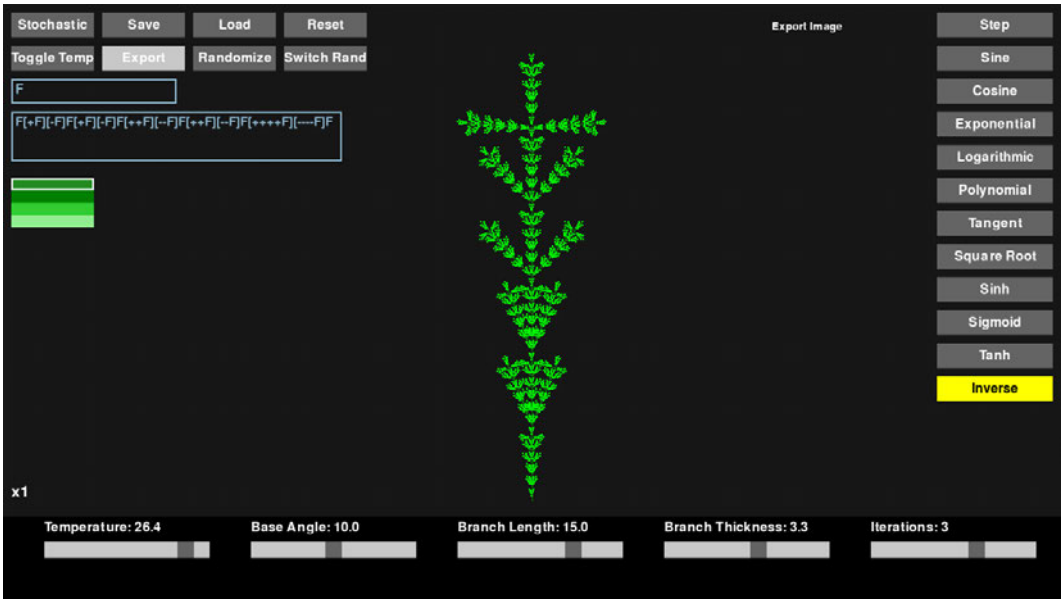


Figure 6.12: Inverse function with 3 iterations at temperature 26.4°C.

7 Conclusion

This thesis explored the integration of fractals and Lindenmayer Systems (L-systems) to model and visualize natural patterns, focusing on a dynamic, responsive L-system that adapted its parameters based on environmental data, specifically temperature. The project enhanced L-systems' flexibility and applicability, allowing more accurate simulations of complex, changing phenomena. The study included a review of the state of the art in fractal and L-system modeling, followed by the implementation of a robust system using Python. This system incorporated user interface elements for real-time interaction and integrated stochastic and deterministic rule variations. The results demonstrated that the system could effectively generate dynamic visualizations of L-systems, responding to changes in temperature and user input. The integration of data inputs to adjust growth parameters showcased the potential for applying such systems in various fields, from data visualization to interactive art.

7.1 Future Outlooks

As we advance with the development of this dynamic, responsive L-system, several future directions and enhancements can be envisioned. These outlooks aim to extend the capabilities of the current project, making it more versatile, powerful, and applicable across various domains.

7.1.1 Integration with Advanced Environmental Data Sources

A significant future enhancement involves integrating the L-system with advanced environmental data sources. By leveraging APIs and IoT devices, the system could utilize real-time data from weather stations, satellite feeds, or sensors deployed in natural environments.

7.1.2 Advanced Rule Systems and Machine Learning

Incorporating advanced rule systems and machine learning algorithms could significantly enhance the L-system's capabilities. Future developments might include:

- Using machine learning to enable the L-system to learn from past simulations and adapt its rules and parameters for more accurate and efficient modeling.
- Developing more complex rule networks that can simulate interactions between multiple species or environmental factors, creating more comprehensive ecological models.
- Implementing optimization algorithms to automatically fine-tune parameters for desired outcomes, improving the efficiency and effectiveness of the simulations.

7.1.3 Collaborative and Educational Platforms

Expanding the system's use in collaborative and educational settings can open new avenues for research and learning:

- Establishing public repositories where users can share their L-system configurations and results, fostering a community of practice and knowledge exchange.

These future outlooks highlight the potential for continued innovation and expansion of the L-system project. By integrating advanced technologies, enhancing user interaction, and exploring practical applications, we can further develop this system into a powerful tool for both research and real-world problem-solving.

Bibliography

[1] Abelson Harold, diSessa Andrea A.: *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Cambridge, MA : MIT Press, 1982

[2] Anaconda: *Miniconda*. https://docs.anaconda.com/free/miniconda/index.html/. 2024. – Accessed: 09.05.2024

[3] Aristid Lindenmayer, Przemysław P.: *The Algorithmic Beauty of Plants*. New York. 1990

[4] CODE, VS: *Visual Studio Code in Action*. https://code.visualstudio.com/docs. n.d.. – Accessed: 05.05.2024

[5] Daniel Terdiman: *35 years of 'impossible' ILM visual effects (photos)*. 2017. – URL https://www.cnet.com/pictures/35-years-of-impossible-ilm-visual-effects-photos/2/. – Accessed on: May. 10, 2024.

[6] Fišer, Marek: *L-systems online*. Charles University in Prague, Faculty of Mathematics and Physics. – URL http://www.malsys.cz/Download/MarekFiser-LsystemsOnline.150208.v101.pdf#page=91&zoom=100,153,428. – Department of Software and Computer Science Education. Accessed: 10.05.2024

[7] Galarreta-Valverde, Macedo: Three-dimensional synthetic blood vessel generation using stochastic L-systems. In: *Medical Imaging: Image Processing*, 2013, S. 86691I

[8] GeeksforGeeks: *Functional vs Non-Functional Requirements*. https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/. 2024. – Accessed: 07.08.2024

[9] Guo, Jiang H.: Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. In: *ACM Transactions on Graphics* 39 (2020), June, Nr. 5, S. 1–13. – URL https://www.cs.purdue.edu/cgvlab/www/publications/Guo20ToG/

[10] Ian McQuillan, Przemyslaw P.: *Algorithms for Inferring Context-Sensitive L-systems*. Department of Computer Science, University of Saskatchewan Saskatoon, SK, Canada. – URL https://algorithmicbotany.org/papers/algorithms-for-inferring-context-sensitive-l-systems.pdf. – Accessed: 30.05.2024

[11] Jason Bernard, Ian M.: *A Fast and Reliable Hybrid Approach for Inferring L-systems*. https://direct.mit.edu/isal/proceedings/alife2018/30/444/99631. 2018. – Accessed: 07.06.2024

[12] JIANWEI GUO, HAIYONG J.: *Inverse Procedural Modeling of Branching Structures by Inferring L-Systems*. Uni Konstanz. – URL https://graphics.uni-konstanz.de/publikationen/Guo2020InverseProceduralModeling/paper.pdf. – ACM Transactions on Graphics, Vol. 39, No. 5, Article 155. P

[13] Jordan Santell: *L-systems*. 2024. – URL https://jsantell.com/l-systems/. – Accessed on: May. 08, 2024

[14] jupyter: *Jupyter Notebook*. https://jupyter.org/. n.d.. – Accessed: 05.05.2024

[15] Jurassic-Pedia: *Dragon Curve (C/N)*. https://www.jurassic-pedia.com/dragon-curve-cn/. 2012. – Accessed: 05.05.2024

[16] Lin, Tong: *Context-sensitive L-systems*. https://redirect.cs.umbc.edu/~ebert/693/TLin/node17.html. n.d.. – Accessed: 07.05.2024

[17] Lin, Tong: *Stochastic L-systems*. https://redirect.cs.umbc.edu/~ebert/693/TLin/node17.html. n.d.. – Accessed: 07.05.2024

[18] Numpy: *NumPy documentation*. https://numpy.org/doc/1.26/. n.d.. – Accessed: 05.05.2024

[19] O. Šťava, R. Mech D. G. A. ; Krištof, P.: *Inverse Procedural Modeling by Automatic Generation of L-systems*. EUROGRAPHICS 2010. – URL https://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Stava10.pdf. – Purdue University, USA

[20] P. Prusinkiewicz, R. Karwowski B. L.: The use of positional information in the modeling of plants. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* ACM (Veranst.), 2001 (SIGGRAPH '01), S. 289–300

[21] Pike, Allen: *Modeling Plants with Lindenmayer Systems*. https://allenpike.com/modeling-plants-with-l-systems/. 2007. – Accessed: 07.05.2024

[22] Prusinkiewicz, P.: Designing and growing virtual plants with L-systems. In: *Proceedings of the XXVI International Horticultural Congress* Bd. 630, 2004, S. 15–28

[23] Prusinkiewicz, Przemysław: *Modeling Plant Development with L-systems*. https://algorithmicbotany.org/papers/modeling-plant-development-with-l-systems.pdf. n.d.. – Accessed: 07.11.2024

[24] Python: *Graphical User Interfaces with Tk*. https://docs.python.org/3/library/tk.html. n.d.. – Accessed: 05.05.2024

[25] Python: *Matplotlib 3.8.4 documentation*. https://matplotlib.org/stable/index.html. n.d.. – Accessed: 05.05.2024

[26] Python: *turtle — Turtle graphics*. https://docs.python.org/3/library/turtle.html. n.d.. – Accessed: 05.05.2024

[27] Radoslaw Karwowski, Przemyslaw P.: *Design and implementation of the L+C modeling language*. University of Calgary, 2008. – URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=019159e152c7af54bc3263efb2e7715c0fd345b7. – Department of Computer Science

[28] Ron Goldman, Tao J.: *Turtle Geometry in Computer Graphics and Computer Aided Design*. CSE.WUSTL.edu.. – URL https://www.cs.wustl.edu/~taoju/research/TurtlesforCADRevised.pdf. – Department of Computer Science, Rice University. Accessed: 30.05.2024

[29] Tasmanian Council of Social Service: *How to Identify and Manage Primary and Secondary Stakeholders*. https://www.tascosslibrary.org.au/how-to/identify-and-manage-primary-and-secondary-stakeholders. 2024. – Accessed: 08.08.2024

[30] Wikipedia: *Heighway dragon curve*. https://en.wikipedia.org/wiki/Dragon_curve#/media/File:Fractal_dragon_curve.jpg. n.d.. – Accessed: 12.05.2024

[31] Wikipedia: *L-system*. https://en.wikipedia.org/wiki/File:Dragon_trees.jpg. n.d.. – Accessed: 05.05.2024

[32] Wolfram MathWorld: *Koch Snowflake*. 2018. – URL https://mathworld.wolfram.com/KochSnowflake.html. – Accessed on: May. 09, 2024.

A Appendix - Turtle interpretation of symbols

Symbol	Interpretation
F	Move forward and draw a line.
f	Move forward without drawing a line.
+	Turn left.
-	Turn right.
∧	Pitch up.
&	Pitch down.
\	Roll left.
/	Roll right.
|	Turn around.
$	Rotate the turtle to vertical.
[	Start a branch.
]	Complete a branch.
{	Start a polygon.
G	Move forward and draw a line. Do not record a vertex.
.	Record a vertex in the current polygon.
}	Complete a polygon.
#	Incorporate a predefined surface.
!	Decrement the diameter of segments.
,	Increment the current color index.
%	Cut off the remainder of the branch.

Table A.1: Turtle commands for L-systems [3]

B Appendix - Source Code

```
import turtle

iterations = input("Enter the number of generations: ") #type in a
    string
iterations = int(iterations) #convert from string to int
startLength = 200 #length of generation 0 line

#pick up the pen and move the turtle over to the left
turtle.up()
turtle.setpos(-startLength*3/2, startLength *3/2/2)
turtle.speed(0)

#generation 0
#axiom type
# koch='F'
koch = 'F+F+F' #axiom for Koch snowflake

#make the final L-System based on the number of iterations
for i in range(iterations):
    koch = koch.replace('F', 'F-F+F-F')

turtle.down()
turtle.color('red', 'black') #draw line in red, enclosed spaces in
    black
turtle.begin_fill()

for move in koch:
    if move == 'F':
        turtle.forward(startLength / (3**(iterations-1)))
    elif move == '+':
        turtle.right(120)
    elif move == '-':
```

```
        turtle.left(60)

turtle.end_fill()
turtle.done()
l_system.generate(5)
```

Code Snippet B.1: Basic Generation of a Koch Snowflake in Python

```
import turtle

def serpinski(side,level):
    angle = 60

    if level == 0:
        for i in range(3): #draw a triangle
            t.fd(side)
            t.left(180-angle)
    else:
        #Triangle, F, Triangle, B, LFR, Triangle, LBR
        serpinski(side/2,level-1)
        t.fd(side/2)
        serpinski(side/2,level-1)
        t.bk(side/2)
        t.left(angle)
        t.fd(side/2)
        t.right(angle)
        serpinski(side/2,level-1)
        t.left(angle)
        t.bk(side/2)
        t.right(angle)



if __name__ == '__main__':
    iterations = int(input("Enter the number of generations: "))
    myLen = int(input("Enter the side length: "))

    t = turtle.Turtle()
    t.shape('turtle')
    t.speed(0)

```

```
34      #position t
35      t.up()
36      t.setpos(-myLen/2,-myLen/2)
37      t.down()
38
39      t.color('black','black')
40      t.begin_fill()
41      serpinski(myLen,iterations)
42      t.end_fill()
```

Code Snippet B.2: Serpinski Triangle Generation in Python

```
1   import turtle as t
2
3   def setTurtle(myTuple):
4       t.up()
5       t.setx(myTuple[0])
6       t.sety(myTuple[1])
7       t.setheading(myTuple[2])
8       t.down()
9
10  def make_fractal(length,langle,rangle,iterations,axiom,target,
        replace,target2,replace2):
11      state = axiom
12      turtleState=[]
13
14      #make the L-System we want to process
15      for i in range(iterations):
16          nextState=''
17          for character in state:
18              if character == target:
19                  nextState += replace
20              elif character == target2:
21                  nextState += replace2
22              else:
23                  nextState += character
24          state = nextState
25
26
27      t.down() #pen down
28      t.color('green','black') #draw line in red, fill black
```

```
29
30      for move in state: #another way to loop through all the
            characters in a string
31          if move == '[':
32              turtleState.append((t.xcor(), t.ycor(), t.heading()))
33          elif move == ']':
34              setTurtle(turtleState.pop())
35          elif move == "F":
36              t.forward(length)
37          elif move == "L":
38              t.left(langle)
39          elif move == "R":
40              t.right(rangle)
41
42      t.done()
43
44  if __name__ == '__main__':
45      iterations = int(input("Enter the number of generations: "))
46      myLen = int(input("Enter the forward movement length: "))
47      t.speed(0)
48      t.bgcolor('black')
49      setTurtle((0, -250, 90))
50      make_fractal(myLen, 25, 25, iterations, 'B', 'F', 'FF', 'B', '
            F[RB]F[LB]RB')
```

Code Snippet B.3: Fractal Weed Generation in Python

```
1   import random
2
3   class StochasticLSystem:
4       def __init__(self, axiom, rules):
5           self.axiom = axiom
6           self.rules = rules
7           self.state = axiom
8
9       def expand(self, iterations):
10          for _ in range(iterations):
11              next_state = []
12              for character in self.state:
13                  if character in self.rules:
```

```
                    # Select a production randomly according to
                        defined probabilities
                    productions = self.rules[character]
                    weights = [prod[1] for prod in productions]
                    choices = [prod[0] for prod in productions]
                    chosen_production = random.choices(choices,
                        weights=weights, k=1)[0]
                    next_state.append(chosen_production)
                else:
                    next_state.append(character)
            self.state = ''.join(next_state)
        return self.state

    def __str__(self):
        return self.state

# Example usage for a stochastic L-system
axiom = "A"
rules = {
    "A": [("AB", 0.5), ("A", 0.5)],
    "B": [("A", 0.7), ("B", 0.3)]
}

l_system = StochasticLSystem(axiom, rules)
result = l_system.expand(10)  # Expand the L-system 10 times
print(result)
```

Code Snippet B.4: Generation of a Stochastic system in Python

```
import random

class ParametricLSystem:
    def __init__(self, axiom):
        self.state = axiom

    def expand(self, rules, iterations):
        for _ in range(iterations):
            next_state = []
            for symbol in self.state:
                name = symbol[0]
                params = symbol[1]
```

```
                rule = rules.get(name)

                if rule:
                    new_symbols = rule(params)
                    if isinstance(new_symbols, tuple):
                        next_state.extend(new_symbols)
                    else:
                        next_state.append(new_symbols)
                else:
                    next_state.append(symbol)
            self.state = next_state
        return self.state

    def __str__(self):
        return ''.join(f"{name}({','.join(map(str, params))})" for
            name, params in self.state)

def rule_A(params):
    x, y = params
    return [('B', [x + 1, y]), ('A', [x * 2, y / 2])]

def rule_B(params):
    x = params[0]
    return [('A', [x - 1]),]

# Example usage
axiom = [('A', [1, 2])]
rules = {
    'A': rule_A,
    'B': rule_B
}

l_system = ParametricLSystem(axiom)
result = l_system.expand(rules, 3)  # Expand the L-system 3 times
print(l_system)
```

Code Snippet B.5: Generation of a Parametric system in Python

```
class ContextSensitiveLSystem:
    def __init__(self, axiom):
        self.state = axiom
```

```
4
5       def expand(self, rules, iterations):
6           for _ in range(iterations):
7               next_state = []
8               for i, symbol in enumerate(self.state):
9                   left = self.state[i-1] if i > 0 else None  # Get
                        the left context if exists
10                  right = self.state[i+1] if i < len(self.state) - 1
                        else None  # Get the right context if exists
11
12                  # Determine the key for the rules dictionary
13                  context = (left, symbol, right)
14                  rule = rules.get(context)
15
16                  if rule:
17                      next_state.append(rule())  # Apply rule if
                            exists
18                  else:
19                      next_state.append(symbol)  # No change if no
                            rule applies
20
21              self.state = next_state
22          return self.state
23
24      def __str__(self):
25          return ''.join(self.state)
26
27  def rule():
28      return 'B'
29
30  # Example usage
31  axiom = ['A', 'A', 'A']
32  rules = {
33      (None, 'A', 'A'): lambda: 'A',  # Context: Nothing on the left
            , A on the right
34      ('A', 'A', 'A'): rule,  # Context: A on both sides
35      ('A', 'A', None): lambda: 'A'  # Context: A on the left,
            Nothing on the right
36  }
37
```

```
l_system = ContextSensitiveLSystem(axiom)
result = l_system.expand(rules, 3)  # Expand the L-system 3 times
print(l_system)
```

Code Snippet B.6: Generation of a Context-sensitive system in Python

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

City Date Signature