Comp 361/ Math 361 Connections and Beyond

L-system in Theory and Action

Tony Bach, Austin Kim, Ayoub Belemlih, Qinxi Wang

Introduction

Aristid Lindenmayer, a Hungarian biologist, developed Lindenmayer systems (also known as L-systems) to model plant cell and fungi growth. As a rewriting system, L-systems share a lot of similarities with formal grammars and are commonly used to model trees and generate fractals. In this essay, we present the theory behind L-systems and their applications, and we explore a Python implementation of L-systems.

I. What is an L-system?

L-system is a parallel system that relies on rewriting grammars, and is often used to generate biological fractals. The formal construction of L-systems includes the following components, as defined by Prusinkiewicz and Lindenmayer in their book "The Algorithmic Beauty of Plants"::

1. A set of variables: symbols that can be replaced by production rules
2. A set of constants: symbols that do not get replaced.
3. A single axiom which is a string composed of some number of variables and/or constants. The axiom is the initial state of the system and is notated by "ω".
4. A set of production rules defining the way variables can be replaced with combinations of constants and other variables. A production consists of two strings - the predecessor (consisting of a single symbol) and the successor.

The rewriting pattern is the most significant feature of the L-systems. When it is rewritten at a higher order, every instance of the letter p will be replaced by the string s. Variations are allowed in the rules, although p must be a single character, and cannot be one of the special symbols =, +, -, !, |, [,], <, >, @, /, \, _, c, or the space character. Rewriting is case-sensitive, so, x = xXx is different from x = xxx.

For example, here is Lindenmayer's original L-system for modeling the growth of algae:

axiom: A
rules: (A->AB), (B->A)
On iteration n, the following strings are produced:
n = 0: A
n = 1: AB
n = 2: ABA

```
n = 3: ABAAB
n = 4: ABAABABA
...
```

And so on. The difference between L-system and a formal grammar is that in every iteration, we apply all the rules possible, while in the latter, only one rule is applied in every iteration.

L-systems take on many forms, a few of the examples we have investigated including L-systems that have symbols(null symbols) that have no drawing operations associated with them, but add complexity when applying production rules; Stochastic L-systems(L-systems combined with chance operations) that allows multiple production rules for a single expression, each with a probability with the sum of all the probabilities must total to 1:

w: F
p1: F (0.33)→ F[+F]F[-F]F
p2: F (0.33)→ F[+F]F
p3: F (0.33)→ F[-F]F

After examining different variations in the rules and formats of L-systems, we now move to the graphical interpretations of the system. Several interpretations have been proposed, including one by Prusinkiewicz, which use a LOGO-style turtle. Given the state of the turtle (*x, y* coordinates, and an angle α to indicate which direction the turtle is facing), a step size *d*, and an angle increment δ, the turtle can respond to commands represented by the following symbols:

- F: move forward a step of length *d*, drawing a line of the path
- f: move forward a step of length *d,* without drawing the line
- +: turn left by angle δ
- -: turn right by angle δ

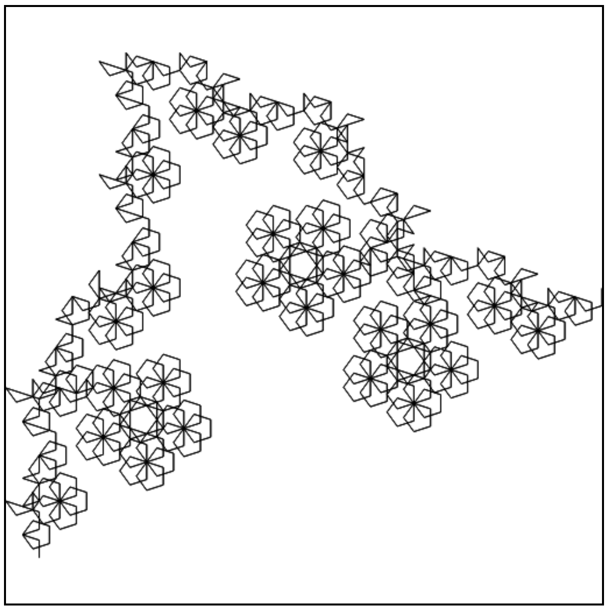
Using L-systems, scientists can model the branching structures and plant growth over time. To do this, two new symbols, "[" and "]: are introduced to simulate a branch. They are interpreted by the turtle as follows:

- [: Push the current state of the turtle onto a pushdown. The information saved on the stack contains the turtle's position and orientation, and possibly other attributes such as the color and width of lines being drawn.
-]: Pop a state from the stack and make it the current state of the turtle. No line is drawn, although in general the position of the turtle changes.

All of these symbols and their interpretations are described by Prusinkiewicz and Lindenmayer in their book.

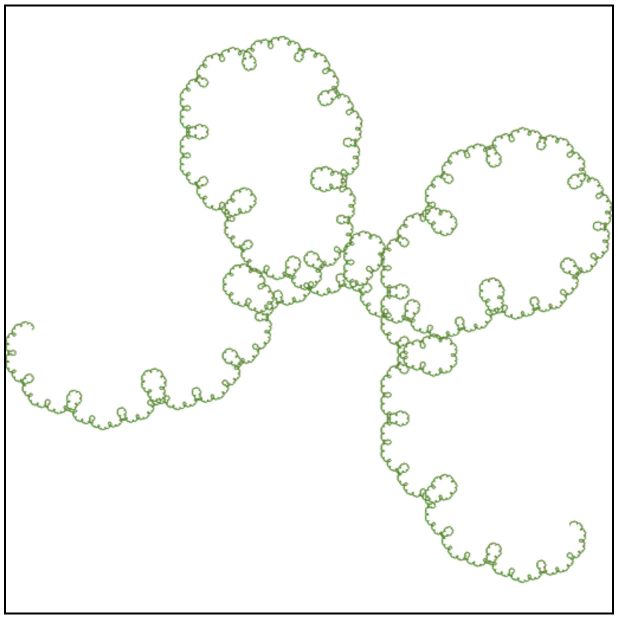
Here we generate and describe some fractals and plants created using L-systems:

Example 1:



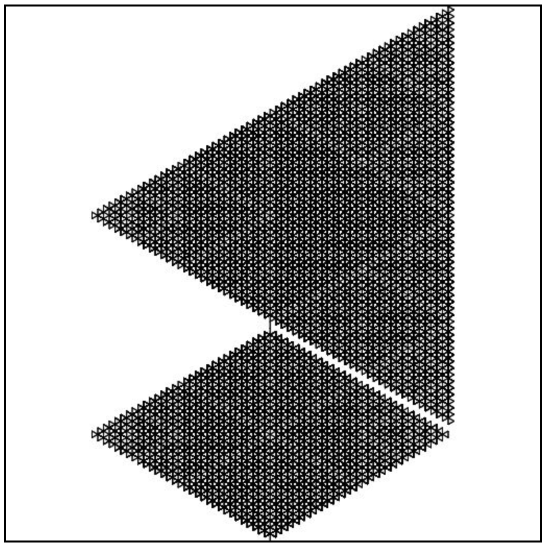
In this example, the language takes on one rule: F=F+F-CF--F+F+F, and ran through 4 iterations to generate this pattern, which I found quite interesting because it looks like three flowers resting on the tree edges, but all the fundamental structures are actually generated by replacing the previous symbol. I was also experimenting with different angles while creating this image, and combinatorics gives me the inspiration and apply 72. So between each ‘flower’, the twisting is more natural comparing to 90 degrees, which also gives the nice shape that resemble the ‘flower’ itself.

Example 2:



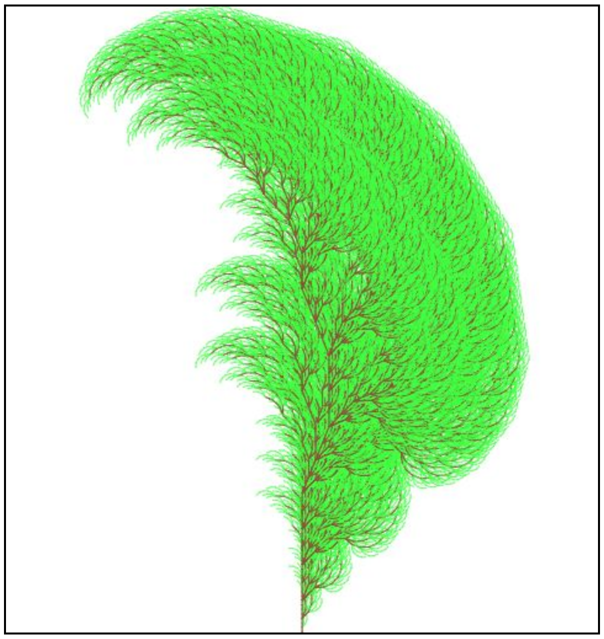
In this example, the language takes on the axoim: F++F++F, and the rule F=+C0F--C1F+ and replace the process for 10 iterations to generate this symmetric pattern. I found this output especially interesting because instead of rigid geometry looks that I was experimenting, this pattern demonstrate a smooth contour, which looks like a butterfly. Same as the previous example, the fundamental structures are generated by replacing the previous symbol, the pattern has this mirroring characteristics base on the way the rule is set up. I am interested in investigating more about how the 35 degree angle is leading to a "overlapping" visual effect on the edges.

Example 3:



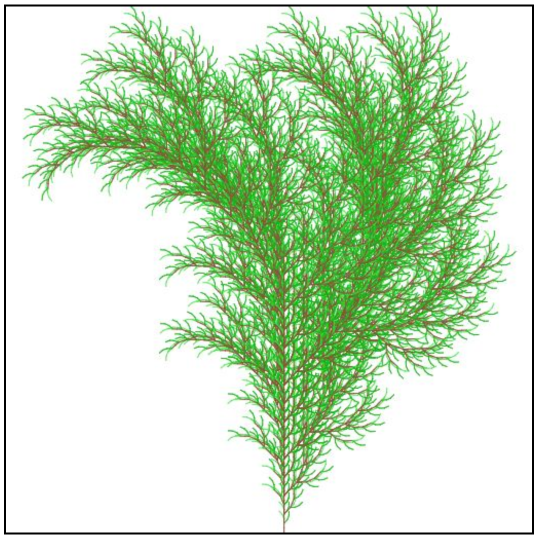
In this L-system, the image represented has one axiom (F=F-F-F) and one following rule. The first 'iteration' of this system can be seen as the equatorial triangles constructed by connecting each midpoint of each side in the largest triangles. From each side of these triangles, we get the midpoints and extrude triangles from that point outward. By repeating this methodology 5 times, we get the image below. I thought it was interesting to make the angle 120, as the 'Forward' movement followed by 120 degree turns meant the system's lines constantly intersected each other.

Example 4:



In this example, the language takes the axiom F and has one rule F=C0FF-[C3-F+F+F+F+F]+[C7-F+F+F]. The angle in this example is 22 and the number of iterations is 5. The output above is a tree and we found L-systems particularly interesting for their use in predicting the growth of plants and trees. The rule in this example contains C0 and C3 to help add color to the diagram and brackets to help generate the branches. Similar to all examples above, the structures of the tree are generated by replacing the previous F symbols and the more we increase iterations the more branches are added to the tree. The majority of the branches shown in the picture above are on the right side of tree, which is due to a greater number of Fs in the first brackets compared to the second ones. By increasing iterations from 2 to 5 in this case, we were particularly interested in the increase of the branches in the tree and the changes in its structure.

Example 5:



This was another plant structure that we came up with. Similar to the above, we also used different colors and branching to generate a beautiful, natural-looking plant. However in this one the left and the right are more balanced, which is reflected in the roughly equal number of symbols in the two brackets.

As for the general application of L-systems in generating fractals and modeling plant growth, we captured the recursive nature of the system itself leads to similarity within one self-developing output, and thereby helps stimulate the uniformity and natural lookings for fractal-like forms. Thus L-systems productions are intended to capture cell divisions in multicellular organisms, where many divisions may occur at the same time. Another advantage of applying L-system in bio-growth modeling is that plant models and natural-looking organic forms develops and becomes more complex by increasing the iteration level of the form, and the recursive system helps complicating the system through each iteration, which helps resolving the problem.

II. Applications and Research Papers

L-systems have a wide applications in variety of modeling tasks. We have worked out a few examples in tree-structure modeling, and we are going to expand our field of investigation into applying L-systems in virtual creature creation and music creation.

The first article we read was *Evolving L-systems to generate virtual creatures*, by Gregory S. Hornby and Jordan B. Pollack. In which they discussed virtual creatures generated using direct encoding lack symmetries and natural structures. Generative encoding is a better alternative that relies on re-writing grammatical rules and helps generating creatures with more-natural looking structures. In this paper, L-systems are the generative encoding tool used by Encoding Algorithms to build creatures with larger number of parts (hundreds for instance) and a natural look.

Encoding Algorithm relies on the grammar construction of L-system to devise rules that iteratively replace unnatural ‘branches’ by variation on the good ‘branches’ in the graph structure. Once the L-system generate a construction command, the string is passed into the constructor for evaluation. The constructor would then build the pieces of the creature using the turtle, which is similar to how the L-system languages create plants, to create bars. The construction module also relies on the L-system commands to indicate if a bar is attached to a fixed or actuated joint, which decides the direction/angles the turtle is moving.

After the L-system specification is executed, which initially created by making random production rules, the constructed creatures will be evaluated on how well they moves using a quasi-dynamic simulator, so that the natural it moves, the higher a score would it get. The Encoding Algorithm would then collect the scores of the performance of each creature constructions, and later replace the lower scores constructed by the command with a high performance score.

We find the approach very clever and adaptive. Since the Encoding Algorithms iteratively use the collection of pieces of creatures that belongs to parents with higher performance scores, and apply variation such as mutation and recombination, which makes the pieces reusable.

Also, Encoding Algorithm starts with a blank template, which is good because the graph structure would not require any prerequisites, and evolves based on the L-systems training/devising on the previous structures. So the construction will get better as the number of iteration increases, which means the creature would gain more natural lookings as the evolution goes along.

Lastly, the paper mentions the possibility of using other types of L-systems to generate virtual creatures: stochastic and context-sensitive L-systems. On one hand, Stochastic L-systems can be used to generate similar structures using the same set of rules. On the other hand, context-sensitive L-systems help implement conditions that are examined to choose the successor of a character.

Besides this one, we also read *Growing Music: musical interpretations of L-Systems* by Peter Worth, Susan Stepney. In this article, the authors seek to answer the questions: Can we create music by working within the rules of L-system grammars? That is, can music be written "generatively"? In addition, is it possible to generate L-systems that are both aesthetically and aurally pleasing, when rendered as plants and music respectively?

As music is normally portrayed in a linear pattern, the idea of using 'sequential' generative grammars to represent music is not a new concept. One way to do this is through *sequential rendering*, such as interpreting [and] as 'push and pop current state except the time', F as 'play a note of duration 1', and a sequence of *n* Fs as 'play a single note of duration *n*'. We can also modify the rendering depending on the resulting sound that we want. For example, in *Schenkerian rendering*, an F is instead interpreted as 'increase note duration by a quarter note', while +/- means to 'move up/down one note in the chosen scale'. To generate multiple pieces in the same musical style, we can use stochastic L-systems, which are also used if we want to generate plants from the same "family" but with slightly different details.

The interesting concept with using L-systems to create music is that it mimics the actual process of writing music, similarly to starting with a 'key'. By using context-sensitive L-systems, as well as the idea of 'push' and 'pop' as placeholders, we can start with one musical note spanning the entire defined musical passage, and then apply "rules" to replace this note by shorter and shorter notes, until all variables become terminals. In addition, L-systems can have a probability characteristic to their rules, which promote musical variation and makes the generated music pleasing to listen to. By relying on a set of "rules" to generate a passage, the generated music automatically follows certain motifs and themes albeit with key changes and passage transposes, which makes context-sensitive L-systems a very powerful way to generate both classical and jazz music. Given Bach's infamous musical tendencies of counterpoint melody, we weren't really surprised that it was possible to create classical music by using a generative system, and that Mozart played a dice game to write some of his pieces.

However, the idea of using L-systems which spatially react relative to other L-systems (L-systems using environmental inputs) to simulate instruments "talking" to each other is a really interesting concept that can be explored to create symphonies.

III. L-systems in Code

In this section, we explore a Python implementation of L-systems. The source code can be found here: http://berniepope.id.au/html/pycol/lsystem.py.html.

How the code works:

In this program, there's an LSystem class that takes in a list of string as its arguments, where each string is a single rule. For example:

```
rules = ['X -> F-[[X]+X]+F[+FX]-X', 'F -> FF']
my_system = LSystem(rules)
```

Once created, an L-System object has a ***run***() method that takes in two arguments: an integer that specifies the number of iterations to run, and an interpreter object. To display the result in turtle graphics, the interpreter should be of the class **Visualize**, where we can describe the mapping of characters to movements. We can use the methods ***basic_actions***(left_angle, right_angle, fwd_distance) and **initPosition()** to outline the initial parameters of our systems.

One minor drawback of this implementation is that there's no way to specify the axiom; the first rule is also considered the starting point. So if we wanted a different axiom, we had to do a little hack, which is to create an unrelated variable that points to the axiom. This can be seen with the variable 'A' in example 1.

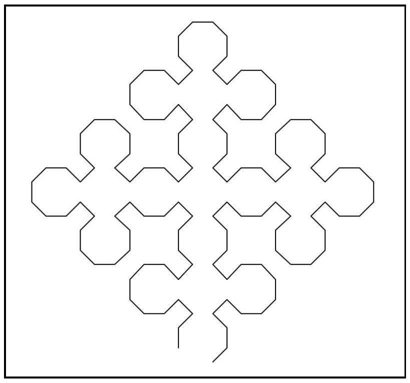
Example 1**:**

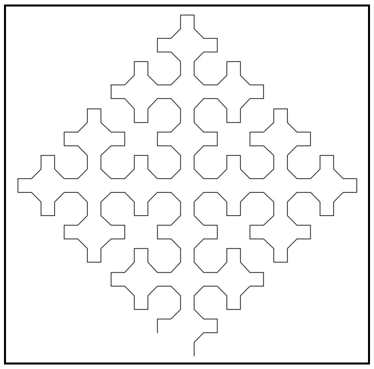
Angle: 45
Axiom: A->L--F--L--F
Rule 1: L -> +R-F-R+
Rule 2: R -> -L+F+L-
L, R, F: move forward

Code (for 6 iterations):

```
def demo():
  def init():
     initPosition()
     left(90)
  actions = basic_actions(45,45,20)
  actions['L'] = lambda _ : forward(20)
  actions['R'] = lambda _ : forward(20)
  vis = Visualise(actions, init)
  sierpinski_median_curve().run(6,vis)

def sierpinski_median_curve():
   return LSystem(['A-> L--F--L--F', 'L -> +R-F-R+', 'R -> -L+F+L-', 'F -> F'])
```


6 iterations


7 iterations

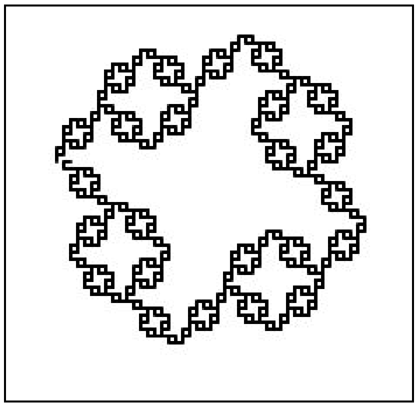
Example 2**:**

Angle: 90
Axiom: F+F+F+F
Rule 1: F -> FF+F+F+F+F+F-F

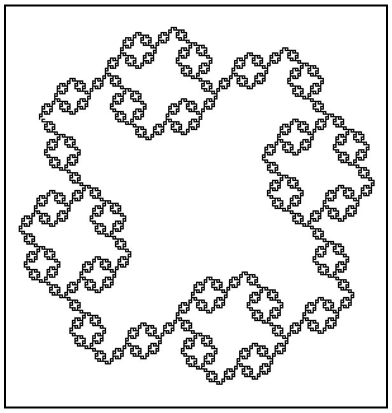
Code (for 6 iterations):

```
def demo():
  def init():
     initPosition(lambda width, height : (-width/4, 0))
     left(90)
  actions = basic_actions(90,90,2)
  vis = Visualise(actions, init)
  rings().run(5,vis)

def rings():
   return LSystem(['A-> F+F+F+F', 'F -> FF+F+F+F+F+F-F'])
```



4 iterations



5 iterations

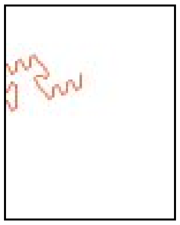
Example 3:

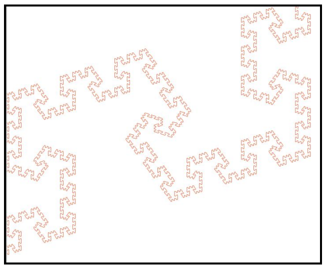
Angle: 72
Axiom: X -> XFYFX+F+YFXFY-F-XFYFX
Rule 1: X -> XFYFX+F+YFXFY-F-XFYFX'
Rule 2: 'Y -> YFXFY-F-XFYFX+F+YFXFY'
Rule 3: 'F -> F'

Code (for 4 iterations):

```
def Qinxi1():
  rules = ['X -> XFYFX+F+YFXFY-F-XFYFX'
        , 'Y -> YFXFY-F-XFYFX+F+YFXFY'
        , 'F -> F' ]
  return LSystem(rules)

def Qinxi1demo():
  def init():
    initPosition(lambda width, height : (-width/2, -height/3))
    left(90)
  actions = basic_actions(72,72,2)
  vis = Visualise(actions, init)
  Qinxi1().run(4,vis)
```


2 iterations


4 iterations

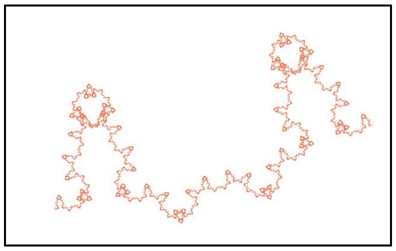
Example 4:

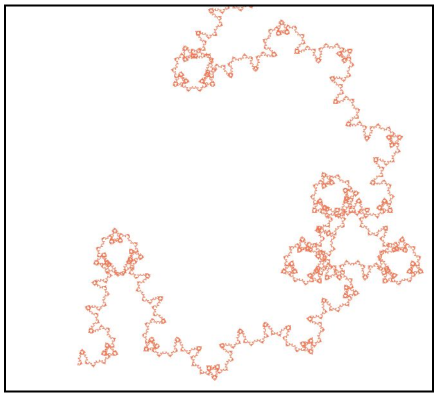
Angle: 72
Axiom: F -> F+F-F-F+F'
Rule 1: F -> F+F-F-F+F'

Code (for 5 iterations):

```
def Qinxi2():
  rules = ['F -> F+F-F-F+F']
  return LSystem(rules)
```

```
def Qinxi2demo():
  def init():
    initPosition(lambda width, height : (-width/3, -height/3))
    left(75)
  actions = basic_actions(75,135,3)
  vis = Visualise(actions, init)
  Qinxi2().run(5,vis)
```


5 iterations


6 iterations

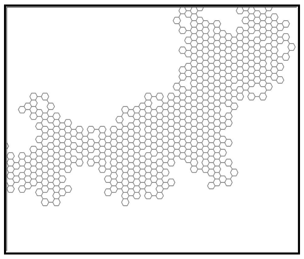
Example 5:

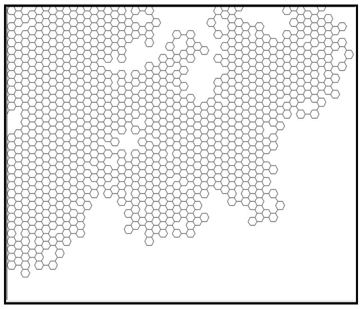
Angle: 60
Axiom: A
Rule 1: A=B-A-A
Rule 2: B=A+B+B

Code for 10 iterations

```
def AyoubDemo():
  def init():
    initPosition(lambda width, height : (-3*width/8, -height/4))
  actions = basic_actions(60,60,10)
  actions['A'] = lambda _ : forward(10)
  actions['B'] = lambda _ : forward(10)
  actions['-'] = lambda _ : right(60)
  actions['+'] = lambda _ : left(60)
  vis = Visualise(actions, init)
  LSystem_X().run(10,vis)

def LSystem_X():
  return LSystem(['A -> B-A-A', 'B -> A+B+B'])
```


8 iterations


10 iterations

Conclusion

Throughout our research, we explored the use of L-systems by computer scientists, biologists, and musicians. We were fascinated with the structures built by L-systems implementations, especially those modelling implementations that allow humans to get a closer look at the future of a plant. In the “L-systems in Code” section of this paper, experimenting with a variety of iterations has also allowed us to develop a better understanding of the functioning of formal grammars in general.

Bibliography

[0]Pope,Bernie. “Bernie's L System demo in Python”. www.cs.mu.oz.au/~bjpop/ http://berniepope.id.au/html/pycol/lsystem.py.html.

[1]Hornby, Gregory S., and Jordan B. Pollack. "Evolving L-systems to generate virtual creatures." *Computers & Graphics* 25.6 (2001): 1041-1048.

[2]Prusinkiewicz, Przemyslaw. "A look at the visual modeling of plants using L-systems." *Agronomie* 19.3/4 (1999): 211-224.

[3]Prusinkiewicz, Przemyslaw. "Modeling of spatial structure and development of plants: a review." *Scientia Horticulturae* 74.1 (1998): 113-149.

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

[5]Worth, Peter, and Susan Stepney. "Growing music: musical interpretations of L-systems." *Applications of Evolutionary Computing*. Springer Berlin Heidelberg, 2005. 545-550.