Citation: Ling, MHT. 2018. **COPADS V: Lindenmayer System with Stochastic and Function-Based Rules.** In Current STEM, Volume 1, pp. 143-172. Nova Science Publishers, Inc. ISBN 978-1-53613-416-2.

COPADS V: LINDENMAYER SYSTEM WITH STOCHASTIC AND FUNCTION-BASED RULES

Maurice HT Ling [*]

Colossus Technologies LLP, Singapore

ABSTRACT

Lindenmayer system, commonly known as L-system, is a string rewriting system based on a set of rules. In each iteration, the string is repeatedly rewritten based on the rules given. This has been used to model branching processes; such as, plant and animal body patterning, and sedimentation processes. In addition to deterministic rewriting rules, stochastic rules have been used, leading to the development of stochastic L-system (S-L-system). For more complex modeling, parametric rules have been used, leading to the development of parametric L-system (P-L-system). Combining S-L-system and P-L-system leads to the development of L-system capable of handling both stochastic and parametric rules or parametric-stochastic-L-system (PS-L-system). Currently, there is no pure Python PS-L-system library. In this study, a light-weight, pure Python PS-L-system has been implemented. This has been incorporated into COPADS repository (https://github.com/copads/copads) and and licensed under GNU General Public License 3.

Keywords: Rewriting rules, Lindenmayer system, L-system, Parametric-Stochastic system, Algorithmic beauty, Python module

Date Submitted: September 13, 2017. **Date Accepted:** December 31, 2017.

Corresponding Author: mauriceling@acm.org, mauriceling@colossus-tech.com

INTRODUCTION

Lindenmayer system, or commonly known as L-system, is invented by Aristid Lindenmayer (Lindenmayer, 1968a; Lindenmayer, 1968b) as a string rewriting system, originally meant to describe the development of multicellular organisms (Lindenmayer, 1971; Lindenmayer, 1975). The main advantage of L-system lies in its simplicity. For example, if every occurrence of "F" is to be rewritten into "FFRR" and given "F" as a starting string, the next 3 rewriting iterations will generate "FFRR", "FFRRFFRRRR", and "FFRRFFRRRRFFRRFF RRRRRR" respectively. Hence, complex strings can be generated from simple rewriting rules over a number of iteration, commonly known as generations. By coding "F" and "R" as "forward" and "turn 60^o angle right" respectively using a LOGO-like turtle, complex structures may be formed. However, it has also been shown that L-systems are Turing complete (van Dalen, 1971); thus, all algorithmic structures are possible.

Shortly after the initial description of L-system (Lindenmayer, 1968a), Lindenmayer (1968b) explains how such a rewriting system can be model branching processes. Using this approach, L-systems have been used for the modeling of various plant structures (Cici et al., 2009; Room et al., 1996; Perttunen and Sievänen, 2005; Prince et al., 2014; Prusinkiewicz and Lindenmayer, 1990; Taralova et al., 2011), body plans of animals (Hoyal Cuthill and Conway Morris, 2014; Davoodi and Boozarjomehry, 2016), generation of 3-D computer graphics world (Fridenfalk, 2016) and soil sedimentation (Rongier et al., 2017a; Rongier et al., 2017b), in recent years.

The most basic form of L-system consists of only replacement rules where one instruction is replaced with one or more instructions. This can be known as replacement L-system (R-L-system). Given the rules and starting string (also known as axiom), the outcomes of an rL-system is deterministic. Non-determinism can be achieved by adding a stochastic parameter to each rule (Samal et al., 1994; Taralova et al., 2011), which gives a probability of executing a rewriting rulem, resulting in a stochastic L-system (S-L-system). S-L-systems allow for multiple rules identifying the same symbol to be executed at different probabilities. R-L-system is also a context-free system, as the execution of replacement rules is solely determined by the current instruction and not the context (what is before and after) of the instruction. Context sensitivity can be achieved by parameterizing the instruction to take account of the contextual landscape of the current instruction (Shi et al., 2011; Xin et al., 2014), resulting in a parametric L-system (P-L-system). Combining S-L-system with P-L-system results in a L-system, which can be both non-deterministic and context sensitive (Parish and Müller, 2011), that can be known as parametric-stochastic L-system (PS-L-system). Recently, a PS-L-system has been used to model blood vessels in organs (Galarreta-Valverde et al., 2013).

In terms of Python-related implementations, there are L-Py (Boudon et al., 2012) and lsystem by Erik Max Francis (http://www.alcyone.com/pyos/lsystem/). However, lsystem is a R-L-system, which does not allow for probabilistic or parametric rules. L-Py (Boudon et al., 2012) is implemented as a C++ library embedded in Python and focused on the flexibility to use any Python objects as instructions and parameters.

In this code manuscript, a pure Python, single file implementation of a PS-L-system, which focus of string rewriting in contrast to L-Py (Boudon et al., 2012), is presented. This implementation allows rules to be executed based on priorities, a feature not found in most (if not all) L-systems. This implementation has been incorporated into COPADS codebase (https://github.com/copads/copads) and licensed under GNU General Public License 3.

CODE DESCRIPTION

The entire Lindenmayer system is implemented the system as a class (lindenmayer class in lindenmayer.py), which comprises of a constructor, three public methods (*add_rules*, *generate*, and *turtle_generate*), and three private methods. The constructor takes a parameter to indicate the length of each instruction or command to process, which is default at one. For example, given a string of AABBACCC and the instruction length is one, the re-writing rules will be executed using parameters in the following sequence – ‘A’, ‘A’, ‘B’, ‘B’, ‘A’, ‘C’, ‘C’, ‘C’. However, if the instruction length is two, the re-writing rules will be executed using parameters in the following sequence – ‘AA’, ‘BB’, ‘AC’, ‘CC’. This means that varying instruction lengths is not allowed.

Once the system is instantiated, the second step is to add rules, which are implemented as Python list of lists, into the PS-L-system using *add_rules* method. Rewriting rules are the crux of a L-system. There are three types of rules: (1) replacement rule, which replaces the current instruction for one or more instructions; (2) probability rule, which is essentially a replacement rule with less than or equal to 100% chance of execution; and (3) function rule, which calls a function and takes in the entire instruction string and the position of the current instruction. Function rules are used to implement parametric rules; and parametric, stochastic rules.

The simplest rule is a replacement rule without priority. The rule “substitute or rewrite X into XYY” can be written as a 2-element list of `['X', 'XYY']`. If this is the only rule, it has to be written as a list of lists, `[['X', 'XYY']]`, in order to be added into the system. If there are more than one rule in a set; each rule can have a different priority, which is added as the 3rd element in the rule list. For example, the rule set `[['X', 'XYY', 1], ['Y', 'XXY', 2]]`, will result all `'X'`s be rewritten as `'XYY'` before all `'Y'`s be rewritten into `'XXY'` as the execution order of the rules will be in ascending order of priorities. Hence, rule set `[['X', 'XYY']]` is computationally equivalent to `[['X', 'XYY', 1]]` in terms of this PS-L-system.

However, there is no restriction that the same instruction cannot be rewritten into more than one forms. Consider the following replacement rule set – `[['X', 'XYY'], [['X', 'YYX']]`. Given that both rules are of the same priority, the first rule (`['X', 'XYY']`) automatically takes precedence over the second rule (`['X', 'YYX']`), implying that the order of such rules is significant (non-communicative). In reality, the second rule (`[‘X’, ‘YYX’]`) will never be used.

In terms of implementation, all replacement rules are converted into a 4-element list with the 4th element being ‘replacement’. This is to cater for stochastic or probability rules, where the 4th element is ‘probability’; and function rules, where the 4th element is ‘function’. Hence, there three ways to write the same replacement rule, `[‘X’, ‘YYX’]`:

- [<instruction>, <replaced instruction(s)>]; for example, [‘X’, ‘YYX’]
- [<instruction>, <replaced instruction(s)>, <priority>]; for example, [‘X’, ‘YYX’, 1]
- [<instruction>, <replaced instruction(s)>, <priority>, 'replacement']; for example, [‘X’, ‘YYX’, 1, ‘replacement’]

The second type of rule is stochastic or probability rule. Each probability rule is defined as a 5-element list in the format of `[<instruction>, <replaced instruction(s)>, <priority>, 'probability', <probability>]`, where `<probability>` refers to the probability of execution. For example, the rule “for 50% of the time, substitute or rewrite X into XYY” will be implemented as `[‘X’, ‘XYY’, 1, 'probability', 0.5]`. Hence, a replacement rule, such as `[‘X’, ‘YYX’, 1, ‘replacement’]`, is computationally equivalent to a probability rule of 100%, such as `[‘X’, ‘YYX’, 1, ‘probability’, 1.0]`, and vise versa.

The third type of rule is function rule, which caters for other more complex operations not addressed by replacement or probability rules. These cases include parametric rules, and probabilistic parametric rules. Each function rule is defined as a 4-element list in the format of `[<instruction>, <function>, <priority>, 'function']`, where `<function>` is a Python function that takes `instructions` and `position` as parameters. For example, the function

```
def replaceFunction(instructions, position):
    if instructions[position+3] == 'O':
        return 'BAAB'
    elif instructions[position-1] == 'O':
        return 'AABB'
    else: return 'OOAB'
```

will perform the following actions, assuming the instruction length to be one and the current instruction is “A”:

- replace the current instruction “A” with “BAAB” if the 3rd instruction from the current instruction is “O”; if not,
- replace the current instruction “A” with “AABB” if the preceding instruction from the current instruction is “O”; if not,
- replace the current instruction “A” with “OOAB”.

This rule is then implemented as `[‘A’, replaceFunction, 1, 'function']`.

The priority of the rules is crucial to the result outcome as shown in Table 1. In event when there is more than one rule with the same priority, the execution order is based on the type of rule – replacement rule to execute first, followed by probability rule, followed by function rule. However, if there is more than one rule type of the same priority; such as, 2 replacement rules of the same priority; the execution order can be indeterministic, resulting in indeterministic results.

Rules	`[[‘X’, ‘XYY’, 1],`
	` [‘Y’, ‘XXY’, 2]]`
		`[[‘Y’, ‘XXY’, 1],`
		` [‘X’, ‘XYY’, 2]]`

Axiom (Generation 0)	`X`	`X`
Generation 1	`X XXY XXY`	`XYY`
Generation 2	`XXX YXX YXX XYX XYX XXY XXY XXY XXX YXX YXX XYX XYX XY`	`XYY XYY XYY YXY YXY YY`
Generation 3	`XXX YXX YXX XYX XYX XXY XXY XXY XXX YXX YXX XYX XYX XYX XXY XXY XXX YXX YXX XYX XYX XYX XXY XXY XXX YXX YXX YXX XYX XYX XXY XXY XXX YXX YXX YXX XYX XYX XXY XXY XXY XXX YXX YXX XYX XYX XYX XXY XXY XXX YXX YXX YXX XYX XYX XYX XXY XXY XXX YXX YXX XYX XYX XXY XXY XXX YXX YXX YXX XYX XYX XXY XXY XXY XXX YXX YXX XYX XYX XY`	`XYY XYY XYY YXY YXY YYX YYX YYX YYY XYY XYY YXY YXY YXY YYX YYX YYY XYY XYY YXY YXY YXY YYX YYX YYY XYY XYY XYY YXY YXY YYX YYX YYY`

Table 1: Differences in Results from the Same Set of Rules but Different Priorities.

The third step is to perform one or more iterations of rewriting, given one or more starting instructions (also known as axiom).

The fourth and last step, which is optional, is to take the iteratively rewritten instruction list and generate a LOGO-like Turtle graphics using Tkinter library within Python Standard Library. The eventual graphics can be saved as a scalable vector graphics (SVG) and the commands used to generate the Turtle graphics can be saved as a Python script file. However, the resulting Python script file can be large as it is a naïve code generation without using loops for code optimization. In order to generate the graphics from the instruction string, a mapping dictionary is required to map each instruction into a Turtle action. The following Turtle commands are defined: forward, backward, right (turn), left (turn), pen up, pen down and home. The default mapping is given as

```
mapping = {'set_angle': 90,
           'random_angle': 0,
           'set_distance': 1,
           'random_distance': 0,
           'set_heading': 0,
           'set_colour': 'black',
           'background_colour': 'ivory',
           'F': 'forward',
           'B': 'backward',
           'R': 'right',
           'L': 'left',
           'H': 'home',
           'U': 'penup',
           'D': 'pendown',
           '[': 'push',
           ']': 'pop'}
```

which can be read as

- a left or right turn is set at 90 degrees (set_angle). This setting is mandatory.
- random angles of turn can be set using 'random_angle', where the actual angle will be from the set_angle to set_angle + random_angle (by uniform distribution). For example, if random_angle is 10 degrees, it means that the actual angle at each turn will be uniformly distributed from 90 to 100 degrees. This will be set to 0 if not given.
- each forward or backward move is set at 1 (set_distance). This setting is mandatory.
- random distance of each move can be set using 'random_distance', following the same logic as 'random_angle'. This will be set to 0 if not given.
- Turtle is set to head towards 0 degrees (east or to the right of the screen). North (toward top), west (towards left), and south (towards bottom) are 90, 180, 270 degrees respectively. This will be set to 0 if not given.
- default pen colour can be set using TK colour names as 'set_colour'. This will be set to black if not given. Other colours can be set and any un-used symbols (other than 'F', 'B', 'R', 'L', 'H', 'U', and 'D') can be used to set pen colours (please see http://wiki.tcl.tk/37701 for available colours).
- canvas background colour can be set using background_colour. This will be set to ivory if not given.

- 'F', 'B', 'R', 'L', 'H', 'U', and 'D' represents the Turtle commands of forward, backward, right turn, left turn, home, pen up, and pen down respectively. Home is defined as the start coordinate.
- '[' pushes the current state (position and heading) of the Turtle into the stack.
- ']' pops (in a last in first out manner) and sets the Turtle to the last pushed state without drawing the move.

CODE FILE (LINDENMAYER.PY)

```
'''
Framework for Lindenmayer System (L-System)
Copyright (c) Maurice H.T. Ling <mauriceling@acm.org>
Date created: 4th January 2015
'''
import random

import constants

class lindenmayer(object):
    '''Lindenmayer system, also commonly known as L-System,
    is developed by Aristid Lindenmayer in 1968 (reference:
    Mathematical models for cellular interaction in
    development. Journal of Theoretical Biology 18:280-315).
    It is a set of formal grammar of production rules for
    rewiting an initial axiom or seed text over generations.

    This implementation defines 3 types of rules, also known
    as production rules or predicates; replacement,
    probability, and function rules. Each rule can be given
    a priority.

    The simplest form of production rule takes the form of
    'A -> BAC', which is read as "whenever 'A' is found, it
    is replaced/rewritten as 'BAC'". For example, if the
    starting axiom is "A", then the following will happen

        - Generation 0: A
        - Generation 1: BAC
        - Generation 2: BBACC
        - Generation 3: BBBACCC
        - Generation 4: BBBBACCCC
        - and so on.

    In this case, the predicate 'A -> BAC' can be written as
    in 4 different ways - C{['A', 'BAC']}, C{['A', 'BAC',
    1]}, C{['A', 'BAC', 1, 'replacement']}, or C{['A',
```

'BAC', 1, 'replacement', 1]}.

When a list of 2-elements is given (e.g., C{['A', 'BAC']}), it is taken to be replacement rule with the highest priority; that is, priority of 1. Hence, C{['A', 'BAC']}, C{['A', 'BAC', 1]}, C{['A', 'BAC', 1, 'replacement']}, and C{['A', 'BAC', 1, 'replacement', 1]} are the same.

This also means that production rules can have different priorities. For example, given C{[['A', 'BAC', 1], ['B', 'BC', 2]]}, rule C{['A', 'BAC', 1]} will be executed before C{['B', 'BC', 2]} in the following manner

```
    - Generation 0: A
    - Generation 1: BCAC       # A -> BAC, BAC -> BCAC
    - Generation 2: BCCBCACC
    - Generation 3: BCCCBCCBCACCC
    - and so on as all production rules in ascending
    order of priorities (with '1' being the highest
    priority) will be executed in sequence on the
    resulting axiom at that current point in time. As a
    result, the order of rules within the same priority
    is significant if they recognize the same
    instruction or command.
```

However, if given C{[['A', 'BAC', 1], ['B', 'BC', 1]]}, then

```
    - Generation 0: A
    - Generation 1: BAC
    - Generation 2: BCBACC
    - Generation 3: BCCBCBACCC
    - Generation 4: BCCCBCCBCBACCCC
    - and so on.
```

The second form of production rule is probabilistic, also known as stochastic grammars. Probabilistic rule will take the format of C{[<domain>, <range>, <priority>, 'probability', <probability>]}. For example, C{['A', 'BAC', 1, 'probability', 0.5]} means that 'A' will only be rewritten into 'BAC' 50% of the time. 'A' will be left unchanged 50% of the time. The same priority principle applies. Hence, C{['A', 'BAC', 1, 'probability', 1]} is in effect the same as C{['A', 'BAC', 1, 'replacement']}.

The third form of production rule is function rule,

```
    which takes the form of C{[<domain>, <function>,
    <priority>, 'function']}. For example, C{['A',
    axiom_func, 1, 'function']} means that when 'A' is
    encounted in the axiom, the command string up to that
    point in time will be used as parameter for axion_func
    function, such as

        - Generation 0: A
        - Generation 1: dependent on the return value of
        axiom_func('A') and so on.

    For example, given an axiom of 'ACCCABABDD', and
    C{['AB', replaceFunction, 1, 'function']} as production
    rule where replaceFunction is defined as

    >>> def replaceFunction(instructions, position):
    >>>     if instructions[position+3] == 'O': return 'BAAB'
    >>>     elif instructions[position-1] == 'O': return
                'AABB'
    >>>     else: return 'OOAB'

        - Generation 0: ACCCABABDD
        - Generation 1: ACCCOOABOOABDD
        - Generation 2: ACCCOOBAABOOAABBDD
        - Generation 3: ACCCOOBABAABOOAABBDD
        - Generation 4: ACCCOOBABABAABOOAABBDD
        - Generation 5: ACCCOOBABABABAABOOAABBDD

    In summary, the following rule formats are allowed:
        - C{[<domain>, <range>]}
        - C{[<domain>, <range>, <priority>]}
        - C{[<domain>, <range>, <priority>, 'replacement']}
        - C{[<domain>, <range>, <priority>, 'probability',
        <probability>]}
        - C{[<domain>, <function>, <priority>, 'function']}
        '''
    def __init__(self, command_length=1):
        '''Constructor method.

        @param command_length: length of each instruction or
        command. Default = 1
        @type command_length: integer'''
        self.command_length = command_length
        self.rules = []

    def add_rules(self, rules):
        '''Method to add a list of production rules /
        predicates into the system.
        @param rules: a list of list describing the
        production rules. Please see above for rule syntax.
        @type rules: list'''
        for x in rules:
            if len(x) == 2:
                # [predicate, replacement]
                self.rules.append([x[0], x[1], 1,
                                   'replacement', None])
            elif len(x) == 3:
                # [predicate, replacement, priority]
                self.rules.append([x[0], x[1], int(x[2]),
                                   'replacement', None])
            elif len(x) == 4 and (x[3] not in
                ['probability', 'replacement', 'function']):
                # [predicate, replacement, priority,
                # '<something_else>']
                print('''Warning: Rule type can only be
'probabilistic', 'replacement' or 'function'. Rule, %s, is not
added into system.''' % str(x))
            elif len(x) == 4 and x[3] == 'replacement':
                # [predicate, replacement, priority,
                # 'replacement']
                self.rules.append([x[0], x[1], int(x[2]),
                                   x[3], None])
            elif len(x) == 4 and x[3] == 'probability':
                # [predicate, replacement, priority,
                # 'probability']
                print('''Warning: Function rule will require a
probability. Rule, %s, is added into system as a replacement rule
(100% activation probability).''' % str(x))
                self.rules.append([x[0], x[1], int(x[2]),
                                   'replacement', None])
            elif len(x) == 5 and x[3] == 'probability':
                # [predicate, replacement, priority,
                # 'probability', probability]
                self.rules.append([x[0], x[1], int(x[2]),
                                   x[3], float(x[4])])
            elif len(x) == 4 and x[3] == 'function':
                # [predicate, function, priority,
                # 'function']
                self.rules.append([x[0], x[1], int(x[2]),
                                   x[3], None])
        self.priority_levels = [x[2]
                                for x in self.rules][-1]

    def _apply_priority_rules(self, priority, data_string):
        '''Private method - to be used by apply_rules method
        to apply production rules of a particular priority.

        @param priority: order of priority
        @type priority: integer
        @param data_string: data or symbol string to be
        processed
        @type data_string: string
        @return: rewritten data_string
        '''
        rules = [x for x in self.rules
                 if x[2] == int(priority)]
        ndata = ''
        pointer = 0
        while pointer < len(data_string):
            cmd = data_string[pointer:
                              pointer+self.command_length]
            for rule in rules:
                if cmd == rule[0] and \
                rule[3] == 'replacement':
                    cmd = rule[1]
                    break
                if cmd == rule[0] and \
                rule[3] == 'probability' \
                and random.random() < x[4]:
                    cmd = rule[1]
                    break
                if cmd == rule[0] and rule[3] == 'function':
                    cmd = rule[1](data_string, pointer)
                    break
            if cmd == None: cmd = ''
            ndata = ndata + cmd
            pointer = pointer + self.command_length
        return ndata

    def _apply_rules(self, data_string):
        '''Private method - to apply all production rules on
        axiom string (in the first generation) or
        data/symbol string (in the subsequent generations).
        This method is implemented as a generator.

        @param data_string: data or symbol string to be
        processed
        @type data_string: string
        @return: rewritten data_string'''
        for priority in
        list(range(1, self.priority_levels+1)):
            data_string = \
                self._apply_priority_rules(priority,
                                            data_string)
        return data_string

    def generate(self, axiom, iterations):
        '''Method to apply all production rules on an
        initial axiom string over a number of iterations.

        @param axiom: data or symbol string to be processed
        @type axiom: string
        @param iterations: number of repetitions /
        iterations
        @type iterations: integer
        @return: rewritten axiom'''
        self.axiom = axiom
        iterations = int(iterations)
        count = 1
        while count < iterations + 1:
            self.axiom = self._apply_rules(self.axiom)
            print('Generation %s: Axiom length = %s' % \
                  (str(count), str(len(self.axiom))))
            count = count + 1
        return self.axiom

    def turtle_generate(self, scriptfile=None,
                        imagefile=None, start=(0, 0),
                        mapping={}, data_string=None):
        '''Method for naive code generation to visualize the
        data or symbol string using Turtle graphics. This
        method generates the Python codes for Turtle
        graphics using the TK Turtle graphics module, and
        prints out the resulting Python code as a file.

        This method does not use any loops to reduce
        repetitive Turtle commands; hence, the resulting
        code file can be huge.

        A mapping dictionary is used to convert the symbol
        string into Turtle commands. The following Turtle
        commands are defined: forward, backward, right
        (turn), left (turn), pen up, pen down and home. The
        default mapping is given as

        >>> mapping = {'set_angle': 90,
        >>>             'random_angle': 0,
        >>>             'set_distance': 1,
        >>>             'random_distance': 0,
        >>>             'set_heading': 0,
        >>>             'set_colour': 'black',
        >>>             'background_colour': 'ivory',
        >>>             'F': 'forward',
        >>>             'B': 'backward',
        >>>             'R': 'right',
        >>>             'L': 'left',
        >>>             'H': 'home',
        >>>             'U': 'penup',
        >>>             'D': 'pendown',
        >>>             '[': 'push',
        >>>             ']': 'pop'}

        which can be read as
            - a left or right turn is set at 90 degrees
            (set_angle). B{This setting is mandatory.}
            - random angles of turn can be set using
            'random_angle', where the actual angle will be
            from the set_angle to set_angle + random_angle
            (by uniform distribution). For example, if
            random_angle is 10 degrees, it means that the
            actual angle at each turn will be uniformly
            distributed from 90 to 100 degrees. B{This will
            be set to 0 if not given.}
            - each forward or backward move is set at 1
            (set_distance). B{This setting is mandatory.}
            - random distance of each move can be set using
            'random_distance', following the same logic as
            'random_angle'. B{This will be set to 0 if not
            given.}
            - turtle is set to head towards 0 degrees (east
            or to the right of the screen). North (toward
            top), west (towards left), and south (towards
            bottom) are 90, 180, 270 degrees respectively.
            B{This will be set to 0 if not given.}
            - default pen colour can be set using TK colour
            names as 'set_colour'. B{This will be set to
            black if not given.} Other colours can be set
            and any un-used symbols (other than 'F', 'B',
            'R', 'L', 'H', 'U', and 'D') can be used to set
            pen colours (please see http://wiki.tcl.tk/37701
            for available colours).
            - canvas background colour can be set using
            background_colour. B{This will be set to ivory
            if not given.}
            - 'F', 'B', 'R', 'L', 'H', 'U', and 'D'
            represents the Turtle commands of forward,
            backward, right turn, left turn, home, pen up,
            and pen down respectively. Home is defined as
            the start coordinate.
            - '[' pushes the current state (position and
            heading) of the Turtle into the stack.
            - ']' pops (in a last in first out manner) and
            sets the Turtle to the last pushed state without
            drawing the move.

        @param scriptfile: file name to write out the Turtle
        commands. Default = None, no file will be written
        @type scriptfile: string
        @param imagefile: SVG file name to write out Turtle
        graphics. Default = None, no file will be written
        @type imagefile: string
        @param start: starting or home coordinate. Default =
        (0, 0) which is the centre of the TK window
        @type start: tuple
        @param mapping: map to convert the symbol string
        into Turtle commands. Please see explanation above.
        @type mapping: dictionary
        @param data_string: data or symbol string to be
        processed. Default = None, internally stored axiom
        string (by lindenmayer.generate() method) will be
        used instead.
        @type data_string: string
        @return: Python script file of Turtle commands'''
        if len(mapping) == 0:
            mapping = {'set_angle': 90,
                       'random_angle': 0,
                       'set_distance': 1,
                       'random_distance': 0,
                       'set_heading': 0,
                       'set_colour': 'black',
                       'background_colour': 'ivory',
                       'F': 'forward',
                       'B': 'backward',
                       'R': 'right',
                       'L': 'left',
                       'H': 'home',
                       '[': 'push',
                       ']': 'pop'}
        if data_string == None:
            data_string = self.axiom
        stack = []
        if 'random_angle' not in mapping:
            mapping['random_angle'] = 0
        if 'random_distance' not in mapping:
            mapping['random_distance'] = 0
        if 'set_heading' not in mapping:
            mapping['set_heading'] = 0
```

```
if 'set_colour' not in mapping:
    mapping['set_colour'] = 'black'
if 'background_colour' not in mapping:
    mapping['background_colour'] = 'ivory'
data_string = [cmd for cmd in data_string
               if cmd in mapping]
if scriptfile != None:
    f = open(scriptfile, 'w')
    f.write("''' \n")
    f.write('Turtle Graphics Generation from \
            Lindenmayer System \n')
    f.write('in COPADS (http://github.com/ \
            copads/copads) \n\n')
    f.write('Code length = %s \n' % \
            str(len(data_string)))
    f.write('Code string = %s \n' % \
            ''.join(data_string))
    f.write('Code mapping = %s \n' % (mapping))
    f.write("''' \n\n")
    f.write('import turtle \n\n')
    f.write('t = turtle.Turtle() \n')
    f.write('t.setundobuffer(1) \n')
    f.write("turtle.bgcolor('%s') \n" % \
            mapping['background_colour'])
    f.write('t.speed(0) \n\n')
    f.write('t.setheading(%s) \n' % \
            float(mapping['set_heading']))
    f.write('t.penup() \n')
    f.write('t.setposition(%s, %s) \n' % start)
    f.write("t.pencolor('%s') \n" % \
            mapping['set_colour'])
    f.write('t.pendown() \n\n')
import turtle
t = turtle.Turtle()
t.setundobuffer(1)
exec("turtle.bgcolor('%s')" % \
     mapping['background_colour'])
exec('t.speed(0)')
exec('t.setheading(%s)' % \
     float(mapping['set_heading']))
exec('t.penup()')
exec('t.setposition(%s, %s)' % start)
exec("t.pencolor('%s')" % mapping['set_colour'])
exec('t.pendown()')
count = 0
for cmd in data_string:
    count = count + 1
    if count % 1000 == 0:
        print('%s instructions processed ...' % \
              str(count))
    if mapping[cmd] == 'push':
        status = [t.position(), t.heading()]
        stack.append(status)
    if mapping[cmd] == 'pop':
        try:
            status = stack.pop()
            if scriptfile != None:
                f.write('t.penup() \n')
                f.write('t.setposition(%s, %s) \n' \
                        % status[0])
                f.write('t.setheading(%s) \n' % \
                        status[1])
                f.write('t.pendown() \n')
            exec('t.penup()')
            exec('t.setposition(%s, %s)' % \
                 status[0])
            exec('t.setheading(%s)' % status[1])
            exec('t.pendown()')
        except IndexError: pass
    if mapping[cmd] == 'forward':
        distance = mapping['set_distance'] + \
          random.random()*mapping['random_distance']
        if scriptfile != None:
            f.write('t.forward(%s) \n' % (distance))
        exec('t.forward(%s)' % (distance))
    if mapping[cmd] == 'backward':
        distance = mapping['set_distance'] + \
          random.random()*mapping['random_distance']
        if scriptfile != None:
            f.write('t.backward(%s) \n' % \
                    (distance))
        exec('t.backward(%s)' % (distance))
    if mapping[cmd] == 'home':
        if scriptfile != None:
            f.write('t.penup() \n')
            f.write('t.setposition(%s, %s) \n' % \
                    start)
            f.write('t.pendown()')
        exec('t.penup()')
        exec('t.setposition(%s, %s)' % start)
        exec('t.pendown() \n')
    if mapping[cmd] == 'right':
        angle = mapping['set_angle'] + \
          random.random()*mapping['random_angle']
        if scriptfile != None:
            f.write('t.right(%s) \n' % str(angle))
```

```
                exec('t.right(%s)' % str(angle))
            if mapping[cmd] == 'left':
                angle = mapping['set_angle'] + \
                  random.random()*mapping['random_angle']
                if scriptfile != None:
                    f.write('t.left(%s) \n' % str(angle))
                exec('t.left(%s)' % str(angle))
            if mapping[cmd] in constants.TKColours:
                f.write("t.pencolor('%s') \n" % \
                        mapping[cmd])
                exec("t.pencolor('%s')" % mapping[cmd])
        exec('t.penup()')
        exec('t.hideturtle()')
        if imagefile != None:
            exec('import canvasvg')
            exec("canvasvg.saveall('%s',
                 t.getscreen()._canvas)" % imagefile)
            f.write('\n')
            f.write('try: \n')
            f.write('    import canvasvg \n')
            f.write("    canvasvg.saveall('%s',
                    t.getscreen()._canvas) \n" \
                    % imagefile)
            f.write('except ImportError: pass \n')
        exec('turtle.done()')
        print('%s instructions processed. Drawing
              completed.' % str(count))
        if scriptfile != None:
            f.write('\n')
            f.write('t.penup() \n')
            f.write('t.hideturtle() \n')
            f.write('turtle.done() \n')
            f.close()
```

TEST CASES (T-LINDENMAYER.PY)

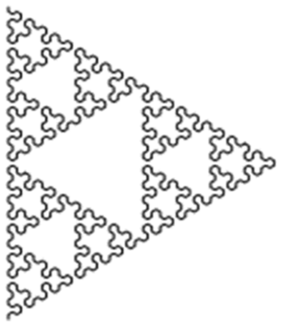
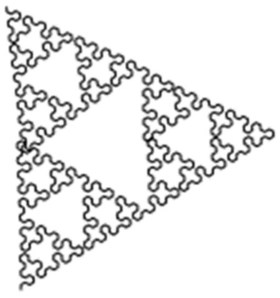
```
import sys
import os
import unittest
import re

sys.path.append(os.path.join(os.path.dirname(os.getcwd()),
'copads'))
import lindenmayer as N

class testReplacement(unittest.TestCase):
    '''
    Test for single replacement rule without priority.
    Command length = 1
    '''
    def setUp(self):
        s = N.lindenmayer(1)
        r = [['A', 'BAC']]
        s.add_rules(r)
        axiom = 'A'
        self.result = []
        for i in range(10):
            axiom = s._apply_rules(axiom)
            self.result.append(axiom)
        self.answer = ['BAC',
                       'BBACC',
                       'BBBACCC',
                       'BBBBACCCC',
                       'BBBBBACCCCC',
                       'BBBBBBACCCCCC',
                       'BBBBBBBACCCCCCC',
                       'BBBBBBBBACCCCCCCC',
                       'BBBBBBBBBACCCCCCCCC',
                       'BBBBBBBBBBACCCCCCCCCC']
    def testGeneration(self):
        self.assertEqual(self.result, self.answer)

class testReplacementPriority(unittest.TestCase):
    '''
    Test for replacement rules with priority.
    Command length = 1
    '''
    def setUp(self):
        s = N.lindenmayer(1)
        r = [['A', 'BAC', 1],
             ['B', 'BC', 2]]
        s.add_rules(r)
        axiom = 'A'
        self.result = []
        for i in range(8):
            axiom = s._apply_rules(axiom)
            self.result.append(axiom)
        self.answer = ['BCAC',
                       'BCCBCACC',
                       'BCCCBCCBCACCC',
                       'BCCCCBCCCBCCBCACCCC',
                       'BCCCCCBCCCCBCCCBCCBCACCCCC',
                       'BCCCCCCBCCCCCBCCCCBCCCBCCBCACCCCCC',
              'BCCCCCCCBCCCCCCBCCCCCBCCCCBCCCBCCBCACCCCCCC',
    'BCCCCCCCCBCCCCCCCBCCCCCCBCCCCCBCCCCBCCCBCCBCACCCCCCCC']
    def testGeneration(self):
        self.assertEqual(self.result, self.answer)

class testReplacementNoPriority(unittest.TestCase):
    '''
    Test for replacement rules without priority.
    Command length = 1
    '''
    def setUp(self):
        s = N.lindenmayer(1)
        r = [['A', 'BAC'],
             ['B', 'BC']]
        s.add_rules(r)
        axiom = 'A'
        self.result = []
        for i in range(8):
            axiom = s._apply_rules(axiom)
            self.result.append(axiom)
        self.answer = ['BAC',
                       'BCBACC',
                       'BCCBCBACCC',
                       'BCCCBCCBCBACCCC',
                       'BCCCCBCCCBCCBCBACCCCC',
                       'BCCCCCBCCCCBCCCBCCBCBACCCCCC',
               'BCCCCCCBCCCCCBCCCCBCCCBCCBCBACCCCCCC',
      'BCCCCCCCBCCCCCCBCCCCCBCCCCBCCCBCCBCBACCCCCCCC']
    def testGeneration(self):
        self.assertEqual(self.result, self.answer)

class testReplacement2(unittest.TestCase):
    '''
    Test for replacement rules without priority.
    Command length = 2
    '''
    def setUp(self):
        s = N.lindenmayer(2)
        r = [['AA', 'BBAAC'],
             ['AB', 'ABC'],
             ['AC', 'BCC']]
        s.add_rules(r)
        axiom = 'AA'
        self.result = []
        for i in range(8):
            axiom = s._apply_rules(axiom)
            self.result.append(axiom)
        self.answer = ['BBAAC',
                       'BBBBAACC',
                       'BBBBBBAACCC',
                          'BBBBBBBBAACCCC',
                          'BBBBBBBBBBAACCCCC',
                          'BBBBBBBBBBBBAACCCCCC',
                          'BBBBBBBBBBBBBBAACCCCCCC',
                          'BBBBBBBBBBBBBBBBAACCCCCCCC']
    def testGeneration(self):
        self.assertEqual(self.result, self.answer)

class testFunction(unittest.TestCase):
    '''
    Test for function rules.
    Command length = 1
    '''
    def replaceFunction(self, dstring, position):
        if dstring[position+3] == 'O': return 'BAAB'
        elif dstring[position-1] == 'O': return 'AABB'
        else: return 'OOAB'
    def setUp(self):
        s = N.lindenmayer(2)
        r = [['AB', self.replaceFunction, 1, 'function']]
        s.add_rules(r)
        axiom = 'ACCCABABDD'
        self.result = []
        for i in range(5):
            axiom = s._apply_rules(axiom)
            self.result.append(axiom)
        self.answer = ['ACCCOOABOOABDD',
                          'ACCCOOBAABOOAABBDD',
                          'ACCCOOBABAABOOAABBDD',
                          'ACCCOOBABABAABOOAABBDD',
                          'ACCCOOBABABABAABOOAABBDD']
    def testGeneration(self):
        self.assertEqual(self.result, self.answer)


if __name__ == '__main__':
    unittest.main()
```

DEMONSTRATION

Two demonstrations are performed using Sierpinski Triangle and Tree. Sierpinski Triangle, also known as Sierpinski Sieve, is a symmetrical fractal design. Structural symmetry is maintained by minimizing the error in each angle. In this demonstration, error is gradually increased from no error, where all angles are 60 degrees; to 12.5% random error, which translates to 60 degrees ± 7.5 degrees. The code for Sierpinski Triangle without randomness in angles is shown as

```
import random
from lindenmayer import lindenmayer

axiom = 'A'
rules = [['A', 'BLALB'],
         ['B', 'ARBRA']]
start_position = (-300, 250)
iterations = 8
turtle_file = '19_lindenmayer_sierpinski_turtle.py'
image_file = '19_lindenmayer_sierpinski_turtle.svg'
mapping = {'set_angle': 60,
           'random_angle': 0,
           'set_distance': 2.5,
           'random_distance': 0,
           'background_colour': 'RoyalBlue1',
           'A': 'forward',
           'B': 'forward',
           'R': 'right',
           'L': 'left'}

l = lindenmayer(1)
l.add_rules(rules)
l.generate(axiom, iterations)
l.turtle_generate(turtle_file, image_file, start_position,
mapping)
```

The generated Turtle graphics (Figure 1) demonstrated that the effects of random angles on the appearance of Sierpinski Triangle. Minor overlaps can be seen even at a 2.5% randomness. By 7.5% randomness, the geometry appears to be collapsing on itself. By 12.5%, it is almost unrecognizable as a Sierpinski Triangle.

(A) No randomness	(B) 2.5% randomness
	
(C) 5% randomness	(D) 7.5% randomness

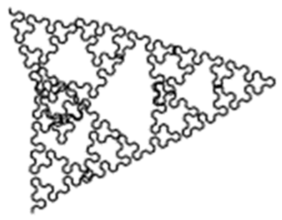
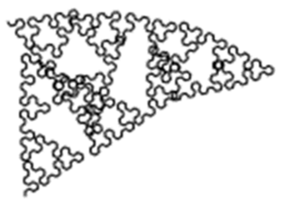
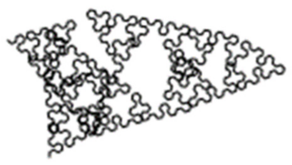
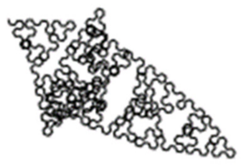
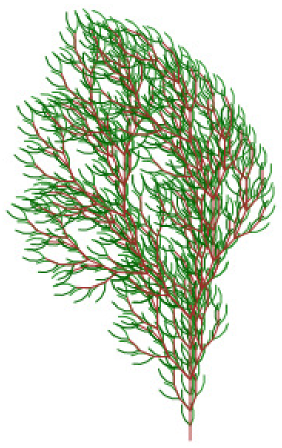
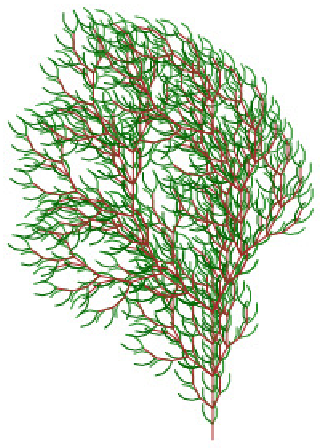
	
(E) 10% randomness 	(F) 12.5% randomness 

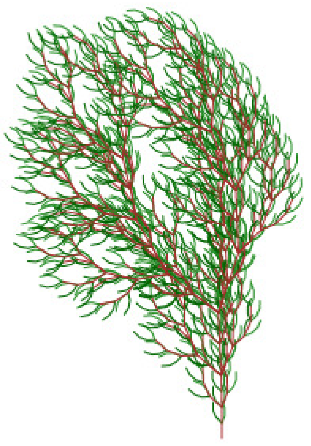
Figure 1: Sierpinski triangle with varying degrees of angle randomness. Each of these images are generated with 6 generations at 60° angle, with or without randomness in angles. Panel A shows no randomness and all angles are 60° angle. Panel B shows 2.5% randomness in angles (60° ± 1.5°). Panels C, D, E, and F show 5% (60° ± 3.0°), 7.5% (60° ± 4.5°), 10% (60° ± 6.0°), and 12.5% (60° ± 7.5°) randomness in angles respectively.

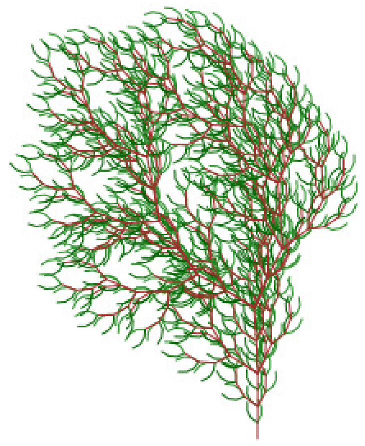
On the other hand, randomness in angles does not appear to affect the "algorithmic beauty" of the tree. The code for the tree without random angle is

```python
import random
from lindenmayer import lindenmayer

axiom = 'F'
rules = [['F', '0FFL[1LFRFRF]R[2RFLFLF]']]
start_position = (0, -200)
iterations = 5
turtle_file = '19_lindenmayer_tree_turtle.py'
image_file = '19_lindenmayer_tree_turtle.svg'
mapping = {'set_angle': 22,
           'random_angle': 0,
           'set_distance': 5,
           'random_distance': 0,
           'set_heading': 90,
           'background_colour': 'ivory',
           'F': 'forward',
           'R': 'right',
           'L': 'left',
           '[': 'push',
           ']': 'pop',
           '0': 'brown',
           '1': 'dark green',
           '2': 'forest green'}

l = lindenmayer(1)
l.add_rules(rules)
l.generate(axiom, iterations)
l.turtle_generate(turtle_file, image_file, start_position,
mapping)
```

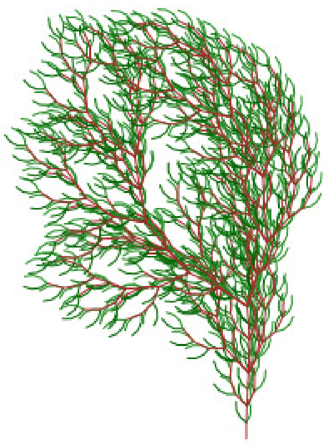
The generated Turtle graphics for the tree (Figure 2) shows that randomness in angles does not affect the "feel" and beauty of the tree even at 20% random angles, which is remarkably different from Sierpinski Triangle (unrecognizable at 12.5% randomness in angles). This suggests that algorithmic beauty can exist in two different ways – symmetric beauty as in the case of Sierpinski Triangle and overall structural beauty as in the case of the tree.

(A) 22^{o}, no randomness	(B) 26.4^{o}, no randomness
	
(C) 22^{o} + 10% randomness	(D) 26.4^{o} + 10% randomness





(E) 22° + 20% randomness



(F) 26.4° + 20% randomness

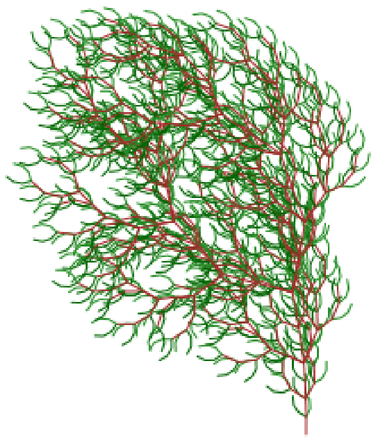


Figure 2: Tree with varying degrees of angles and randomness. Each image is generated 4 generations. Panels A and B show 22°, and 26.4° (20% increase of 20% increase in angle from 22°) respectively. Panels C and E show 22° angle with 10% ($22^\circ \pm 2.2^\circ$), and 20% ($22^\circ \pm 4.4^\circ$) randomness respectively. Panels D and F show 26.4° angle with 10% ($26.4^\circ \pm 2.64^\circ$), and 20% ($26.4^\circ \pm 5.28^\circ$) randomness respectively.

References

Boudon, F., Pradal, C., Cokelaer, T., Prusinkiewicz, P., and Godin, C. (2012) L-Py: An L-System Simulation Framework for Modeling Plant Architecture Development Based on a Dynamic Language, Frontiers in Plant Science 3, 76.

Cici, S. Z.-H., Adkins, S., and Hanan, J. (2009) Modelling the morphogenesis of annual sowthistle, a common weed in crops, Computers and Electronics in Agriculture 69, 40-45.

Davoodi, A., and Boozarjomehry, R.B. (2016). Developmental model of an automatic production of the human bronchial tree based on L-system. Computer Methods and Programs in Biomedicine 132, 1–10.

Eichhorst, P., and Ruskey, F. (1981) On unary stochastic Lindenmayer systems, Information and Control 48, 1-10.

Fridenfalk, M. (2016). Application for Real-Time Generation of Mathematically Defined 3D Worlds. In Transactions on Computational Science XXVIII: Special Issue on Cyberworlds and Cybersecurity, M.L. Gavrilova, C.J.K. Tan, and A. Sourin, eds. (Berlin, Heidelberg: Springer Berlin Heidelberg), pp. 45–68.

Galarreta-Valverde, M. A., Macedo, M. M., Mekkaoui, C., and Jackowski, M. P. (2013) Three-dimensional synthetic blood vessel generation using stochastic L-systems, In SPIE Medical Imaging, pp 86691I-86691I-86696, International Society for Optics and Photonics.

Hoyal Cuthill, J. F., and Conway Morris, S. (2014) Fractal branching organizations of Ediacaran rangeomorph fronds reveal a lost Proterozoic body plan, Proceedings of the National Academy of Sciences of the United States of America 111, 13122-13126.

Lindenmayer, A. (1968a) Mathematical models for cellular interactions in development. I. Filaments with one-sided inputs, Journal of Theoretical Biology 18, 280-299.

Lindenmayer, A. (1968b) Mathematical models for cellular interactions in development. II. Simple and branching filaments with two-sided inputs, Journal of Theoretical Biology 18, 300-315.

Lindenmayer, A. (1971) Developmental systems without cellular interactions, their languages and grammars, Journal of Theoretical Biology 30, 455-484.

Lindenmayer, A. (1975) Developmental algorithms for multicellular organisms: a survey of L-systems, Journal of Theoretical Biology 54, 3-22.

Room, P., Hanan, J., and Prusinkiewicz, P. (1996) Virtual plants: new perspectives for ecologists, pathologists and agricultural scientists, Trends in Plant Science 1, 33-38.

Parish, Y. I., and Müller, P. (2001) Procedural modeling of cities, In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, pp 301-308, ACM.

Perttunen, J., and Sievänen, R. (2005) Incorporating Lindenmayer systems for architectural development in a functional-structural tree model, Ecological Modelling 181, 479-491.

Prince, D. R., Fletcher, M. E., Shen, C., and Fletcher, T. H. (2014) Application of L-systems to geometrical construction of chamise and juniper shrubs, Ecological Modelling 273, 86-95.

Prusinkiewicz, P., and Lindenmayer, A. (1990) The algorithmic beauty of plants, Springer-Verlag.

Rongier, G., Collon, P., and Renard, P. (2017a). Stochastic simulation of channelized sedimentary bodies using a constrained L-system. Computers & Geosciences 105, 158–168.

Rongier, G., Collon, P., and Renard, P. (2017b). A geostatistical approach to the simulation of stacked channels. Marine and Petroleum Geology 82, 318–335.

Samal, A., Peterson, B., and Holliday, D. J. (1994) Recognizing plants using stochastic L-systems, In. Proceedings of IEEE International Conference on Image Processing, 1994 (ICIP-94), pp 183-187, IEEE.

Shi, Y., Cheng, X., and Zhang, H. (2011) Three dimensional trees emulation based on Parametric L-system, Journal of Tongji University. Natural Science 39, 1871-1876.

Taralova, E. H., Schlecht, J., Barnard, K., and Pryor, B. M. (2011) Modelling and visualizing morphology in the fungus Alternaria, Fungal Biology 115, 1163-1173.

van Dalen, D. (1971) A note on some systems of Lindenmayer, Theory of Computing Systems 5, 128-140.

Xin, L., Xu, L., Li, D., and Fu, D. (2014) The 3D reconstruction of greenhouse tomato plant based on real organ samples and parametric L-system, In Sixth International Conference on Digital Image Processing, International Society for Optics and Photonics.