Practical 1: Lindenmayer systems and fractal plants

The deadline for this practical is **22/04/2024 23h59** on Moodle. You will return a zip archive containing the practical itself (same structure as you downloaded it).

1 Introduction and motivation

In the early 90s, Przemyslaw Prusinkiewicz and Aristid Lindenmayer published a book called *The Algorithmic Beauty of Plants*, discussing the computer simulation of plants through fractals. This was the first book that treated in depth the generation of virtual plants. One of these ways to perform this generation is through *L-systems*, created by Lindenmayer in 1968.

The goal of this practical is to create such plants. In this practical, you are going to write the code

1. that generates L-systems
2. that performs the geometric transformations that will draw the fractals.

The files where the functions you will have to write are

— `mathsutils.py`

— `lsystem.py`

The symbol ♣ means that there are unit tests for the associated function. To use the unit tests, launch the command `pytest tests/namefile/namefunction.py` from the tp1 directory. For example, if I want to test the function `r2d` which is in the file `mathsutils.py`, I launch `pytest tests/mathsutils/r2d.py`.

Important : Imports must not be changed !

2 L-systems : derivation and generation

2.1 Definitions

Définition 1 *An L-system is a formal grammar defined by a tuple (V, Σ, P, w), where :*

— *V is a set of symbols, known as variables ;*

— *Σ is a set of symbols, known as constants or terminal symbols, representing the basic building blocks ;*

— *P is a set of production rules, each rule consisting of a variable and its corresponding replacement string of variables and/or constants ;*

— *w is an initial axiom, a string of variables and/or constants, serving as the starting point for the generation process.*

A derivation of a string is applying the production rules to this string. It consists in replacing every symbol in V by the content of the associated produciton rule.

Example 2 *Let us consider the following L-system :*

— $V = \{F, X\}$
— $\Sigma = \{[,], +, -\}$
— $P = \{(F, FF), (X, F[+X][-X]FX)\}$
— $w = X$

The axiom contains only one symbol, X. The first derivation simply replaces X by $F[+X][-X]FX$. In a second derivation, we read this string and replace every X by $F[+X][-X]FX$ and every F by FF. Thus, after two derivations, we obtain the string

$$FF[+F[+X][-X]FX][-F[+X][-X]FX]FFF[+X][-X]FX$$

Examples of L-systems are given in the files `lsystems/lsystem2d.py` and `lsystems/lsystem3d.py`. They also contain other parameters that will be necessary for the drawings.

2.2 Implementation

In our implementation, strings will be strings. Production rules are given by a dictionary. A way to know if a symbol we meet is a variable is by checking if it is in the keys of the dictionary.

W1 ♣ In the file `lsystem.py`, write the function `derivation` that applies one derivation on the input `axiom` according to the production rules given by the input `rules`.

W2 ♣ In the same file, write the function `generation` that takes as input an axiom, a set of rules and the number of times the derivation must be done.

```
derivation('X', {'F':'FF', 'X':'F[+X][-X]FX'})
>> 'F[+X][-X]FX'

generation('X', {'F':'FF', 'X':'F[+X][-X]FX'}, 2)
>> 'FF[+F[+X][-X]FX][-F[+X][-X]FX]FFF[+X][-X]FX'
```

3 Drawing in 2D

3.1 Mathematical tools

In order to draw, we will need functions that perform transformations of the plan (and later of the space). First, standard libraries provide us useful trigonometric functions, but their input as to be given in radians and our measures are in degree.

W3 ♣ In the file `mathsutils.py`, write the function `degreetorad` that takes as input an angle in degrees and returns its value in radians.
Then, our transofmations are modeled by matrices and vectors. We need basic tools to handle these.

W4 ♣ In this same file, write the function `multmatvector` that takes as input a matrix and a vector and returns their product.

W5 ♣ In this same file, write the function `multmat` that takes as input two matrices and returns their product.

```
degreetorad(40)
>> 0.6981317007977318

multmatvector([[1, 2, 3], [-1, 0, 2]], [-1, 1, 2])
>> [7, 5]

multmat([[1, 2, 3], [-1, 0, 2]], [[-1, 1], [0, 1]])
>> [[-1, 3], [1, -1]]
```

For the drawing, we will use the turtle system [1]. To do that, we store the orientation of the turtle as a rotation matrix. Given an angle θ, the rotation matrix is given by

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

W6 ♣ In this same file, write the function `r2d` that takes as input an angle in degrees and returns the corresponding matrix. (remember, radians !)

```
r2d(30)
>> [[0.8660254037844387, -0.49999999999999994],
    [0.49999999999999994, 0.8660254037844387]]
```

(Floatings...)
Now we need to be able to move forward by a distance d from an initial position (x, y) with an angle given by the rotation matrix R. This is given by the formula

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + R \begin{pmatrix} d \\ 0 \end{pmatrix}$$

W7 ♣ In this same file, write the function `move2d` that takes as inputs the position, the rotation matrix and the distance, and returns the new coordinates.

```
move2d(0, 5, r2d(30), 3)
>> (2.598076211353316, 6.5)
```

3.2 Parsing and drawing

Now that all our mathematical tools have been defined, we are ready. The result of the generation has to be understood as a sequence of commands to execute. Here is the list of commands.

1. F : move forward and create a line
2. G : move forward and create a line
3. $+$: turn left
4. $-$: turn right
5. [: save position and direction on a stack

1. `https://en.wikipedia.org/wiki/Turtle_graphics`

6.] : retrieve position and direction from stack

Other symbols have to be ignored. Note the management of the stack can be simplified by using a recursive process. To perform this, we use the class `LSystem2d` (as a struct) to store the informations needed. The left and right angles are stored (in absolute value) in the fields `l_angle` and `r_angle`. Similarly, the initial angle is stored in `init_angle` and so is the distance.

W8 In the file `lsystem.py`, write the function `axiomtoline2d` that takes as input a string, a couple of floats representing the initial position, and an lsystem (object of the class `LSystem2d`). This function must return a list of Line (class `Line`) whose constructor takes as input two couples of floats (the coordinates). Everytime the turtle moves forward, a line is created between its starting position and its ending position.
Finally, once this has been done, you can launch the command `python 2dgen.py` and admire (or not ?) the result. Notice that some well known fractals have been included, so you can know if your function is correct.

4 Drawing in 3D

This section is similar to the previous one, but the objective is to draw our fractals in 3D.

4.1 Mathematical tools

Now, rotations are around three axes. The rotation matrices are the following :

$$R_U(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R_L(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \quad R_H(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

W9 ♣ In the file `mathsutils.py`, write the three corresponding functions.
Now, similarly to the 2D version, we need to be able to move forward by a distance d from an initial position (x, y, z) with a direction given by the rotation matrix R. This is given by the formula

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + R \begin{pmatrix} d \\ 0 \\ 0 \end{pmatrix}$$

W10 ♣ In this same file, write the function `move3d` that takes as inputs the position, the rotation matrix and the distance, and returns the new coordinates.

4.2 Parsing and drawing

Here is the new list of commands. A novelty is that we want to add color. This coloring system will be piloted by a parameter depth (relative to the recursive depth in the process).

1. F : move forward and create a line
2. G : move forward and create a line
3. $+$: turn left by an angle θ, using rotation matrix $R_U(\theta)$
4. $-$: turn right by an angle θ, using rotation matrix $R_U(-\theta)$

5. & : pitch down by an angle θ, using rotation matrix $R_L(\theta)$
6. ˆ : pitch up by an angle θ, using rotation matrix $R_L(-\theta)$
7. $>$: roll right by an angle θ, using rotation matrix $R_H(\theta)$
8. $<$: roll left by an angle θ, using rotation matrix $R_H(-\theta)$
9. [: save position and direction on a stack, increase depth by one
10.] : retrieve position and direction from stack, decrease depth by one

Other symbols have to be ignored. To perform this, we use the class `LSystem3d` (as a struct) to store the informations needed. All the angles are stored (again, in absolute value) in the corresponding fields. Similarly, the initial angle is stored in `init_angle` and so is the distance.

W11 In the file `lsystem.py`, write the function `axiomtoline3d` that takes as input a string, a couple of floats representing the initial position, and an lsystem (object of the class `LSystem3d`). Now the coordinates in the line will be triplets (as we are in 3d). Moreover, each time a line is created, the current depth will be stored alongside.
This function must return a list of couples, where the first part of the couple is a line, and the second is its depth.
Finally, once this has been done, you can launch the command `python 3dgen.py` and admire the result.

4.3 To go further

In nature, plants are not always the same. Create the function `axiomtoline3drand` that is similar to `axiomtoline3d` but adds some randomness in the creation of the lines. It may be the angles, the lengths of the lines... Be creative ! (This function will not be evaluated).