2D-landscapes generation: an introduction

The goal of this practical is to generate a landscape seen from the side. In a first part, you will create your own source of pseudo-random numbers. In a second part, you will use this generator in order to create *horizon lines*. Finally, you will explore non-random generation for vegetation (especially trees).

The following files are given:

- rng.py: contains material for a pseudo random numbers generator, you will have to modify it
- utils.py: contains some functions useful for the backend, do not modify it
- tp1.py: where you will do most of the work
- main.py: contains the procedure for drawing with pygame. Contains some calls to the functions you are defining (commented for now). You can modify it, but proceed with caution !

Assignment: while this practical is not graded, you will submit it by april 05, 23h42. A git is available with the following command, replacing `$USER` by your login.

```
git clone $USER@git.cri.epita.fr:p/2026-s2-nts/tp1-$USER
```

1 Pseudo Random Number Generator

The first part consists in defining a Linear Congruential Generator (LCG), as seen in the courses. We remind what an LCG is:

Definition

Let $(m, a, c) \in \mathbb{N}^3$ such that $0 < a < m$ and $0 \leq c < m$. The sequence $(X_n)_{n\geq 0}$ is defined by

$$\begin{cases} 0 \leq X_0 < m \\ X_{n+1} = (a \times X_n + c) \mod m \end{cases}$$

X_0 is called the *seed*.

1.1 Creation

You are given a file rng.py which contains a class PseudoRNG where the constructor is given. However, this class is not complete. It contains two methods you have to complete.

1. nextInt computes the next integer in the sequence and stores it in *val*.
2. randInt(inf,sup) returns an integer between inf (included) and sup (excluded). It should call nextInt in order to generate the next number of the sequence, and then "transform" it with arithmetic operations to obtain a number between inf and sup.

1.2 Initialization

Now that you have a generator, we should choose *good* values to initialize it. In this practical, you will use $m = 2^{31} - 1$, $a = 48271$ and $c = 0$. These are the values used for `minstd_rand` in C++11.

Do not use 0 as a seed. Any other value is ok.

Now, in a shell, try the command line *python3 main.py*. A window appears with a blue zone (the sky) and a green one (the ground). You will add mountains and trees to it. If you look at this file, you will see the commands for mountains and tree generations have been commented. You will uncomment them when ready.

2 Horizon lines

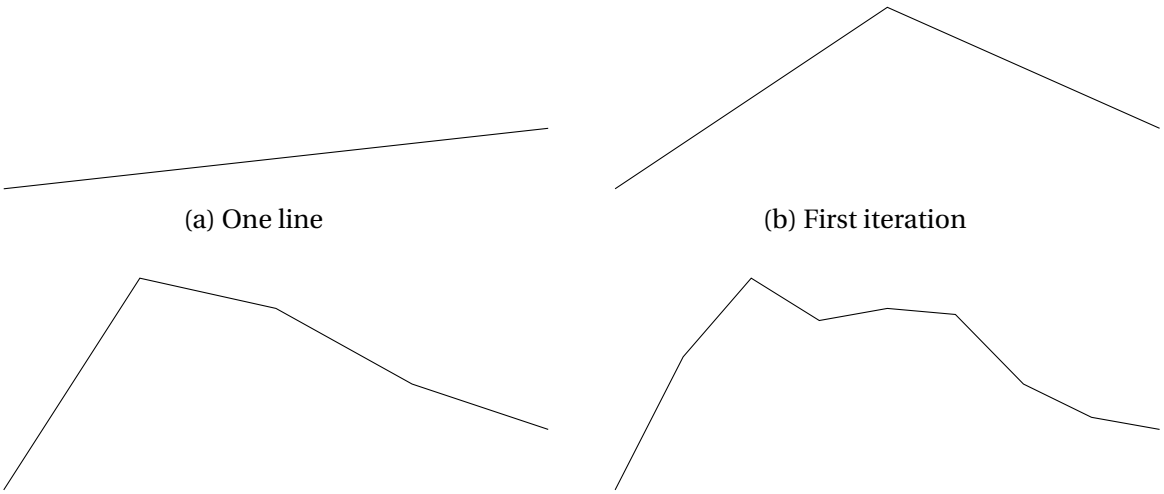
The goal of this section is to create a landscape seen from the side. For that, you will generate broken lines following this principle.

Midpoint displacement algorithm

Begin with a straight line segment, compute its midpoint and displace its y coordinate by a bounded random value. The first iteration will result in two straight lines, on which the process can be repeated, resulting in four, eight... segments, until the difference between the horizontal coordinates of two consecutive points is smaller than some threshold δ.

For now, let us do it step by step.

1. Create a PRNG and initialize it.
2. Implement the function `midpoint` such that for two points (*ie* two couples of integers) p and q, midpoint(p,q) returns the middle of the segment, rounded to have integer coordinates.



(a) One line (b) First iteration

(c) Second iteration (d) Third iteration

Figure 1: Three successive iterations

3. Implement the midpoint displacement algorithm. Here are the explanations for two parameters: delta and dh. delta is the threshold for the precision. We iterate the algorithm as depicted in Figure 1 until the **horizontal** difference between two successive points is lower than delta. dh is how much we change the height. For example, you can displace by a random number between -dh and dh.

   Feel free to explore how to change the size dh at each iteration of the displacement algorithm ! However, no point should be lower than the parameter base_height. The result is a list of points that are ordered `from left to right`.

4. To generate a mountain, first create a "triangle": a list containing three points: the left basis, the summit, and the right basis. The two basis one should be at height base_height. Then, `main.py` will call the midpoint displacement algorithm on the result (do not do it yourself !).

   The parameters are schematized in Figure 2. The mountain width should be (randomly) between *min_width* and *max_width*, and similarly for the mountain height. And of course, you should place the mountain randomly, but somewhere in the window !

5. Then, you can generate several mountains !

In the main file, the generation of mountains is commented. You can uncomment them when you feel you are ready.

3 L-systems and plants

In this last section, the goal is to generate "fractal plants" with L-systems.

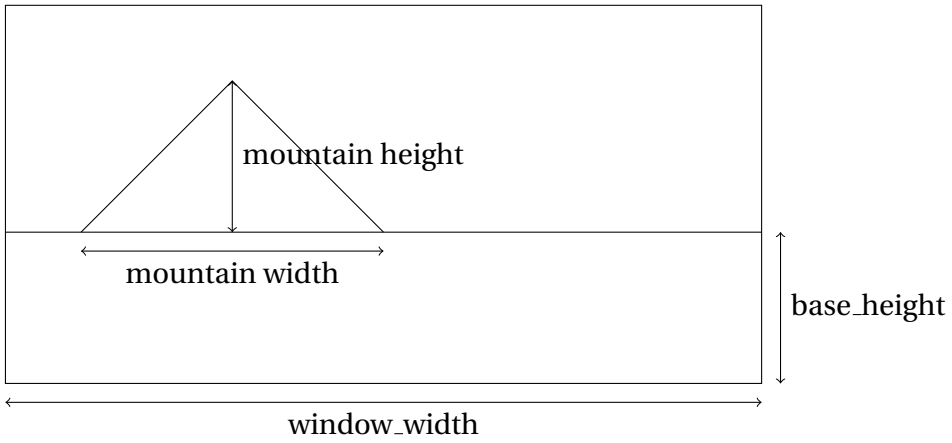


Figure 2: Schema - mountain in the drawing window

L-systems

An L-system (also called *Lindenmayer system*) is a rewriting system allowing one to generate strings from a set of rules. It has two main components.

- A starting string, called *axiom*
- At least one rewriting rule

Rules describe how to use rewriting and obtain a new string (generally more complex than the initial one). These strings can then (among other things) be interpreted as instructions for drawing.

As an example, let us take the following starting string $s_0 = F$ and the single rule $F : F[+F]F[-F]F$. This rule states that each F has to be rewritten as $F[+F]F[-F]F$.

After one step, we obtain the string $s_1 = F[+F]F[-F]F$. After two steps, we obtain

$$s_2 = F[+F]F[-F]F[+F[+F]F[-F]F]F[+F]F[-F]F[-F[+F]F[-F]F]F[+F]F[-F]F$$

which is s_1 where each F has been replaced according to the rule. *Note that the size of the string grows exponentially, it may not be wise too apply the rule too many times.*

In this work, given an initial distance d, an left turn angle l, a right turn angle r, and a memory stack we will give the following meanings to the symbols:

- F: go forward on d units
- +: turn right r degrees
- -: turn left l degrees
- [: put current position and angles on stack

-]: unstack position and angles
- X: do nothing

For example, with the rule $F : F[+F]F[-F]F$, $l = 20$ degrees and $r = 20$ degrees, we obtain:

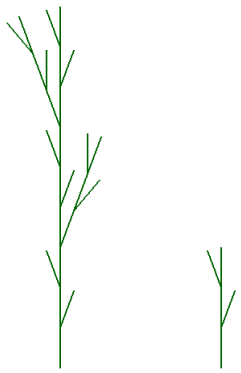


Figure 3: Some trees corresponding to s_2 and s_1

In order to do that, you have three functions to do:

1. generation(axiom,rules,n) which will apply the rules n times on the axiom. If a character is not in the rules, then you copy it without changing it.

   Some tools for a dictionary *dico*:

   - Get the value associated to key c: *dico[c]*
   - Test if a key is in the dictionary: c in *dico.keys()*

2. new_position(posX,posY,angle,distance) that given an initial position, an angle and a distance will return the new position as a couple of **integers**. Reminder: cos and sin functions take radians as parameters !

3. string_to_lines(string, initial_x, initial_y, initial_angle, angle_l, angle_r, distance) that given initial parameters, the rules and the distance, will return a list of couples of couples $((x_1, y_1), (x_2, y_2))$, that are the segments to draw.

   The initial angle given in parameter is the up direction.

In the *main* file, several calls to the draw_tree function are commented. You can uncomment them when you feel you are ready.