

---

# **L-Py Documentation**

***Release 0***

**F. Boudon, C. Godin et al.**

**Mar 22, 2023**



---

## Contents

---

<b>1</b>	<b>Documentation</b>	<b>3</b>
1.1	Installing Lpy . . . . .	3
1.2	L-Systems . . . . .	5
1.3	File syntax . . . . .	7
1.4	L-Py Editor . . . . .	8
1.5	L-Py Turtle basic primitives . . . . .	20
1.6	L-Py Turtle advanced primitives . . . . .	41
1.7	Tutorial . . . . .	46
1.8	Subtleties with L-Py . . . . .	49
1.9	L-Py Helpcard . . . . .	51
1.10	L-Py in scripts or in third party applications . . . . .	59
<b>2</b>	<b>References</b>	<b>63</b>



**Version****Version** 3.13.0**Date** Mar 22, 2023

L-systems were conceived as a mathematical framework for modeling growth of plants. L-Py is a simulation software that mixes L-systems construction with the Python high-level modeling language. In addition to this software module, an integrated visual development environment has been developed that facilitates the creation of plant models. In particular, easy to use optimization tools have been integrated. Thanks to Python and its modular approach, this framework makes it possible to integrate a variety of tools defined in different modeling context, in particular tools from the OpenAlea platform. Additionally, it can be integrated as a simple growth simulation module into more complex computational pipelines.



## 1.1 Installing Lpy

L-Py distribution is based on the `conda` software environment management system. To install `conda`, you may refer to its installation page: <https://docs.conda.io/projects/conda/en/latest/user-guide/install/>

### 1.1.1 Installing binaries using conda

To install L-Py, you need to create an environment (named `lpy` in this case) :

```
conda create -n lpy openalea.lpy -c fredboudon -c conda-forge
```

The package is retrieved from the `fredboudon` channel (development) and its dependencies will be taken from `conda-forge` channel.

Then, you need to activate the L-Py environment

```
conda activate lpy
```

And then run L-Py

```
lpy
```

### 1.1.2 Installing from sources

You should first install all dependencies in a `conda` environment. The simplest way to do this is to call

```
conda create -n lpydev
conda activate lpydev
conda install --only-deps openalea.lpy -c fredboudon -c conda-forge
conda install
```

You should clone the lpy project into your computer

```
git clone https://github.com/fredboudon/lpy.git
```

### 1.1.3 Compiling on macOS and Linux

This assumes you installed the usual build tools on Linux, or the Xcode Build Tools on macOS.

You need then to compile lpy with the following command, on macOS and Linux:

```
mkdir build ; cd build
cmake -DCMAKE_INSTALL_PREFIX=${CONDA_PREFIX} \
      -DCMAKE_PREFIX_PATH=${CONDA_PREFIX} \
      -DCMAKE_BUILD_TYPE=Release \
      -LAH ..
make
```

You can use `make -j numproc` if you have several processors. To install L-Py on your environment

```
make install
python setup.py install
```

to install it into you python system.

To run test,

```
cd test/
nosetests
```

To launch the visual editor, you can type in your shell

```
lpy
```

### 1.1.4 Compiling on Windows

On Windows you must install **Visual Studio 2019** with Desktop C++ tools.

For your convenience a build script called *windows\_build\_dev.bat* has been written. If you installed **Visual Studio 2019** with Desktop C++ tools and **miniconda3** at the default location, with your environment called **lpydev**, running the script from the Windows Command Prompt should compile lpy.

If you want to compile manually, open the **Developer Command Prompt for VS 2019** (search for the shortcut in the Start Menu).

Then you should activate **conda** manually in that prompt. If you installed **miniconda3** in the default directory *C:\Users\YourName\miniconda3* and if your environment is named *lpydev*, you can use the command:

```
%USERPROFILE%\miniconda3\Scripts\activate.bat %USERPROFILE%\miniconda3\envs\lpydev
```

Otherwise, adapt the command to the path where you installed miniconda3.

Then you can compile with the following commands:

```
mkdir build
cd build
cmake .. -G "Visual Studio 16 2019" ^
```

(continues on next page)



(continued from previous page)

```
-Wno-dev ^
-DCMAKE_INSTALL_PREFIX=%CONDA_PREFIX%\Library ^
-DCMAKE_PREFIX_PATH=%CONDA_PREFIX%\Library ^
-DCMAKE_INSTALL_RPATH:STRING=%CONDA_PREFIX%\Library\lib ^
-DCMAKE_INSTALL_NAME_DIR=%CONDA_PREFIX%\Library\lib ^
-DPython3_EXECUTABLE=%CONDA_PREFIX%\python.exe

cmake --build . --parallel %NUMBER_OF_PROCESSORS% --config Release --target install
cd ..
```

Note: you can only compile using the config **Release** and the target **install** on Windows.

To install L-Py on your environment

```
python setup.py install
```

to install it into you python system.

To run test,

```
cd test/
nosetests
```

To launch the visual editor, you can type in your shell

```
lpy
```

### 1.1.5 Notes on dependencies

L-Py core is a C++ library but based on the Python language. The communication between both language is made using `Boost.Python`. The `PlantGL` library is used for the 3D modelling and visualization. The `Qt` library and its python wrappers `PyQt` (build with `SIP`) are used to create the visual interface. `PyOpenGL` is used to display and edit the materials.

To compile and install it from sources, the project requires `cmake` and `setuptools`.

To test it, the `nosetests` conventions is used.

All these projects have to be correctly installed before compiling L-Py.

Additionally, the `Cython` module that make it possible to translate python code into C code is automatically integrated to the project if detected. You can install it if you want to test this extension.

## 1.2 L-Systems

### 1.2.1 Introduction

Lindenmayer [Lin1968] proposed a model of development based on *rewriting rules* or *productions*. This model, known as *L-systems*, originally provided a formal description of the development of simple multicellular organisms and was later extended to higher plants [Prusinkiewicz, 89]. For this, the structure of the systems is represented as a *string* of *modules*, i.e. a label and some parameters, representing the different components and their states. The rewriting rules operate on components of a plant; for example an apex, an internode, or a leaf and describe their evolution in time.

A rule consists of two components, a *predecessor* and a *successor*. During a *derivation step*, the predecessor is replaced by the successor. Even very simple L-systems can produce plant-like structures. The difference between L-systems

and other rewriting system such as Chomsky grammars lies in the method of applying productions. In Chomsky grammars productions are applied sequentially, whereas in L-systems they are applied in parallel and simultaneously replace all letters in a given string.

### 1.2.2 String Representation

Lindenmayer developed a string notation that makes it easy to specify productions and carry out simulations. Each component of the system is represented as a module. A module is characterized by a label, for instance **A**, and possibly a set of parameters. For instance, an apex of age 10 can thus be represented by the module **A(10)**, an internode of length 20 and radius 3 by **I(20,3)**. To represent a linear structure, a *string*, called *L-string* can be created as a serie of modules. For branched structures such as trees, special modules [ and ] are used to represent begin and end of branches. After each module in the string, its lateral successors enclosed in square brackets are first given followed by its axial successor. For instance in the structure **A[B]C**, **A** carries a lateral module **B** and is followed by module **C**.

### 1.2.3 Geometric representation of the string

To graphically represent the string, it is possible to use a LOGO like turtle that will parse sequentially the string and interpret some of the modules as geometric commands [Prusinkiewicz, 86]. The commands will be additioned to create the geometry. For instance, module **F** makes the turtle draw a cylinder and move forward, + and - to rotate on the right or left. You can read section [Predefined Symbols] to have the list of interpretable modules.

### 1.2.4 Rules

Production rules are composed of a *predecessor* which represents the pattern of modules to match and replace, and *successor* which is expressed as python code containing production statements.

```
# simple rule
predecessor --> successor
# equivalent to
predecessor : produce successor
# more complex rule
predecessor :
    if condition:
        produce successor
    else:
        produce successor2
```

As an example,

```
I(1) --> I(1+d1)
A(t) :
    if t < APEX_DURATION:
        produce I(INITIAL_LENGTH)
    else:
        produce # empty production
```

The pattern given in the successor is expressed as a set of modules represented by their labels and variables for the parameters. A pattern is matched to a module or a set of modules of a string only if labels and numbers of parameters are the same. If matched, the variables are assigned to the actual values in the string of the parameters and the code of the successor is called.

As one can see in the example the production of the successor can be conditional. The successor can produce an empty string (produce or produce \*) meaning that the successor is replaced by nothing. The production can also

produce nothing. In this case the application will supposed to have failed and other rules can be tested. If no rule matches a module of the string, then identity rule will be applied and the module will be replaced by itself.

The replacement of a successor can depend on the neighbours of the modules to replace. In this case, the rule is called context sensitive and the successor, now called *strict successor* is augmented with pattern for neighbour modules (i.e. the contexts). Lpy supports 4 types of contexts.

- left and right contexts, i.e. neighbour modules on the left and on the right.

```
left_context < strict_predecessor > right_context --> successor
```

For instance,  $A < B > C \rightarrow D$  or  $I < A \rightarrow IA$

- new left and right contexts, i.e. futur neighbour modules in the new string on the left and on the right.

```
new_left_context << strict_predecessor --> successor
                strict_predecessor >> new_right_context --> successor
```

Of course, the `new_left_context` is only available if the rule are applied from left to right on the string and `new_right_context` from right to left.

These contexts can then be combined.

## Different types of Rules

- Production rules are intended to express the development of the modelled structure. They are applied in parallel on the L-string.
- Decomposition rules are intended to decompose recursively a module into an L-string using a set of possibly recursive rules. To avoid infinite recursion a maximum depth of recursion can be specified.
- Interpretation rules allows to specify the geometric interpretation of symbols used in a given simulation. For this, a mapping to interpretable symbols can be made. Recursive rules can be used and similarly to decomposition rules, a maximum depth of recursion can be specified. A Turtle object is managed by the simulation and L-Py translates automatically some predefined modules into the corresponding method call onto the Turtle object. It is also possible to directly access the Turtle object using the option 'Turtle in Interpretation rules'. In such case, the turtle is accessible using the `turtle` variable and any of its methods can be called.

```
Internode(t):
    turtle.F(t)
```

## 1.3 File syntax

The Lpy file format is based on the python language which is extended with L-system particular constructs.

### 1.3.1 Canvas of L-Py file

```
# pure python code
def func():
    # python code
    nproduce lstring # it is possible to use the nproduce statement
                  # in this part of the file

module A,B      # declaration of module name
```

(continues on next page)

(continued from previous page)

```

Axiom: lstring # declaration of axiom

derivation length: int # default = 1
                    # number of derivation step to perform
production: # beginning of production rules

pattern :          # a production rule. Start with successor given as a pattern of
↳module to replace
    python code    # rule core are pure python code with production statement
    produce lstring # production statement giving the new string pattern to produce

# simple rules can be expressed this way
pattern --> new_pattern

homomorphism: # beginning of homomorphism rules.
              # They are called before plotting the string or
              # application of rule with query modules ([PHUR])
maximum depth: int # default = 1
                  # number of homomorphism recursive step to perform.
                  # should be defined only once

decomposition: # beginning of decomposition rules.
               # These rules are applied recursively after each production step
               # usefull to decompose a module into a structure
maximum depth: int # default = 1
                  # number of decomposition recursive step to perform.
                  # should be defined only once

group: int # all following rules will be assign to this group
          # to activate a group of rule see command useGroup
          # by default group 0 is active

production: # again all types of rule can be defined
homomorphism:
decomposition:

endgroup # following rules will be assign to default 0 group

production: # again all types of rule can be defined
homomorphism:
decomposition:

endlsystem # end of rules definition

# pure python code is again possible

```

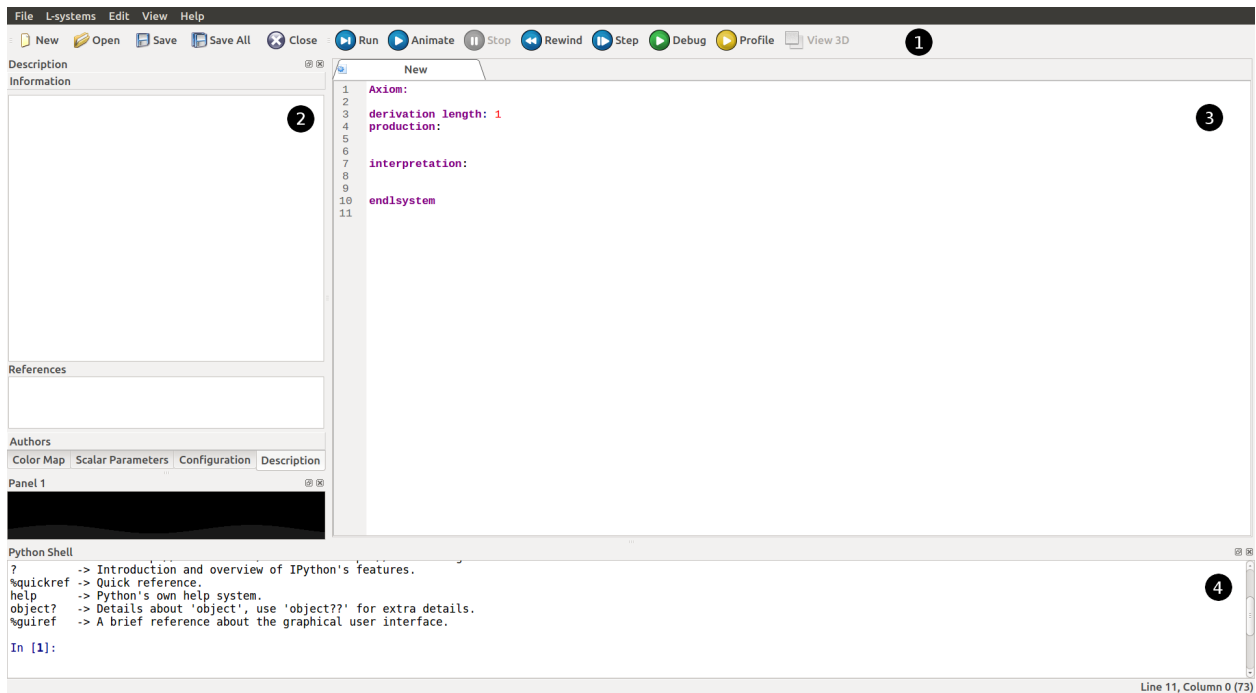
### 1.3.2 L-System specific declaration

## 1.4 L-Py Editor

### 1.4.1 First look on the editor

L-Py has a built-in editor developped with Qt, an UI-designed library.

On start, the editor looks like this:



### 1) Top Toolbar

This toolbar presents the most used features of L-Py. You can create, open or save your current program; you can run and animate your work with the appropriate buttons or even execute it step by step and ultimately you can debug or check the process and rendering time with the Profile button.

### 2) Sidebar tools

On the left sidebar, meta information on the model and its execution can be defined or controlled :

- “Color map” : Creation of custom materials or textures to assign to objects of the simulation.
- “Configuration” : Configure the settings of the execution of the model.
- “Description” : Presentation of the models and its contributors.

### 3) Editor

L-Py gives you the possibility to code inside the application by a built-in editor. All L-Py keywords are recognized and colored for a best readability.

### 4) Python Shell

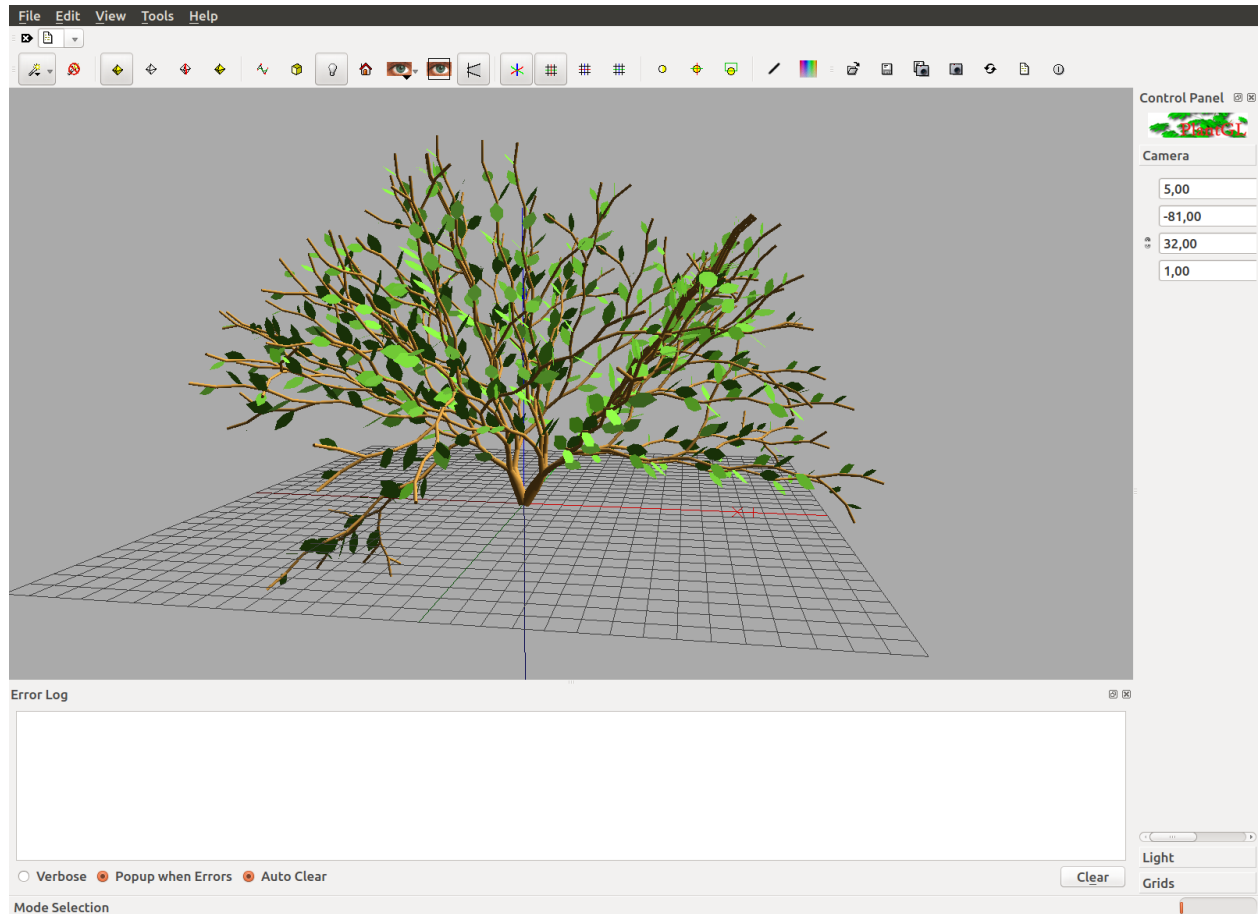
A Python Shell makes it possible to manipulate the lsystems and their variables.

### 5) Custom panels

Some panels for the definition of graphical objects such as function, curves or patches.

## 1.4.2 PlantGL Viewer

By default, the visualization of the model is made within the PlantGL Viewer after clicking on **Run** or **Animate** buttons .

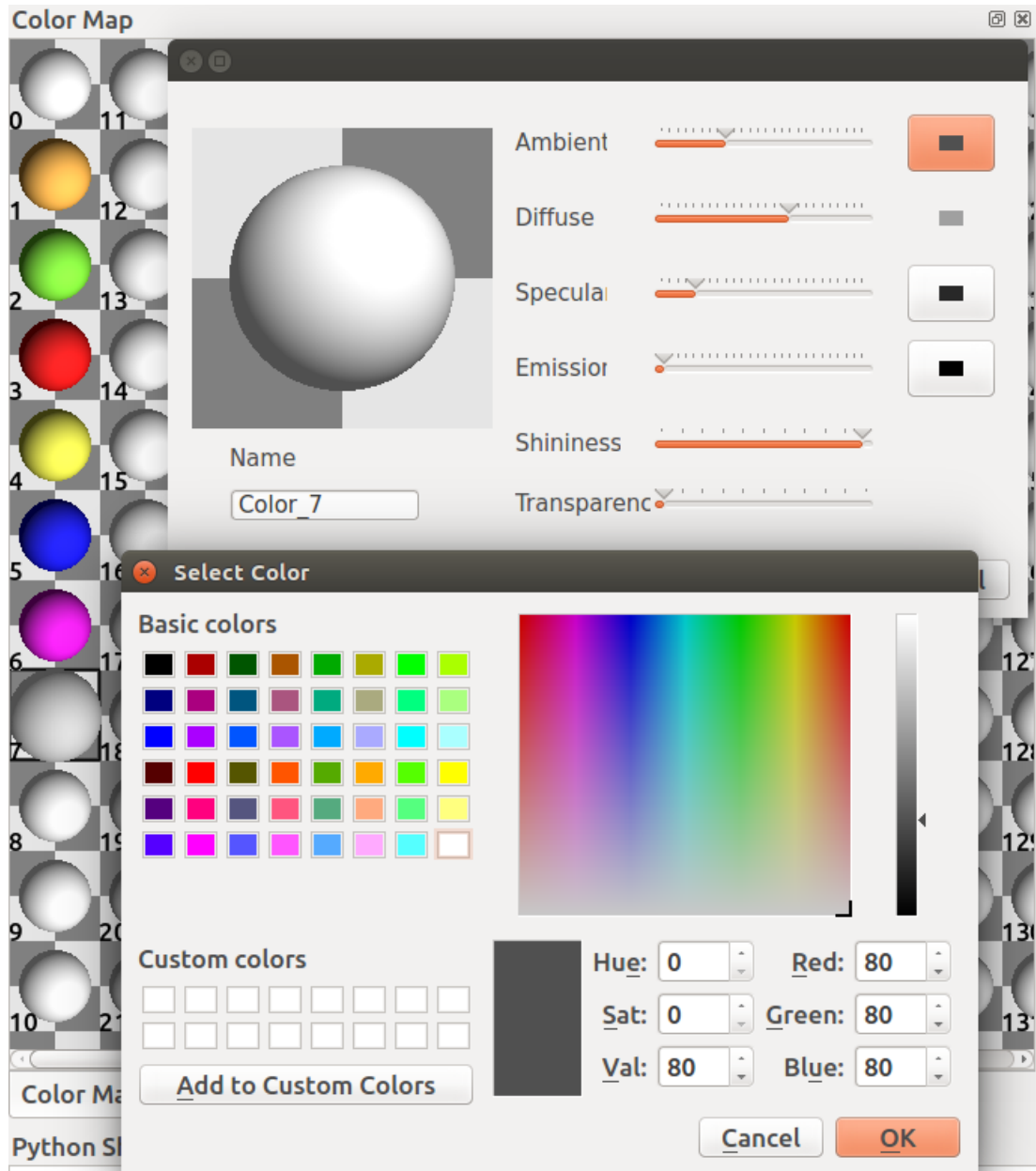


The PlantGL visualizer has a 3D-camera where you can turn around your object. The basic controls you'll mostly use are:

- *Hold Left Click* to turn around X and the Y axis of the camera
- *Wheel Mouse* to zoom / unzoom on the scene
- *Hold Right Click* to shift the scene on the screen

### 1.4.3 Color Map

It can be used to create colors and access it directly in your code by avoiding multiple duplications of `SetColor(r,g,b[,a])` thanks to the `' , '` or `SetColor(index)` instructions.

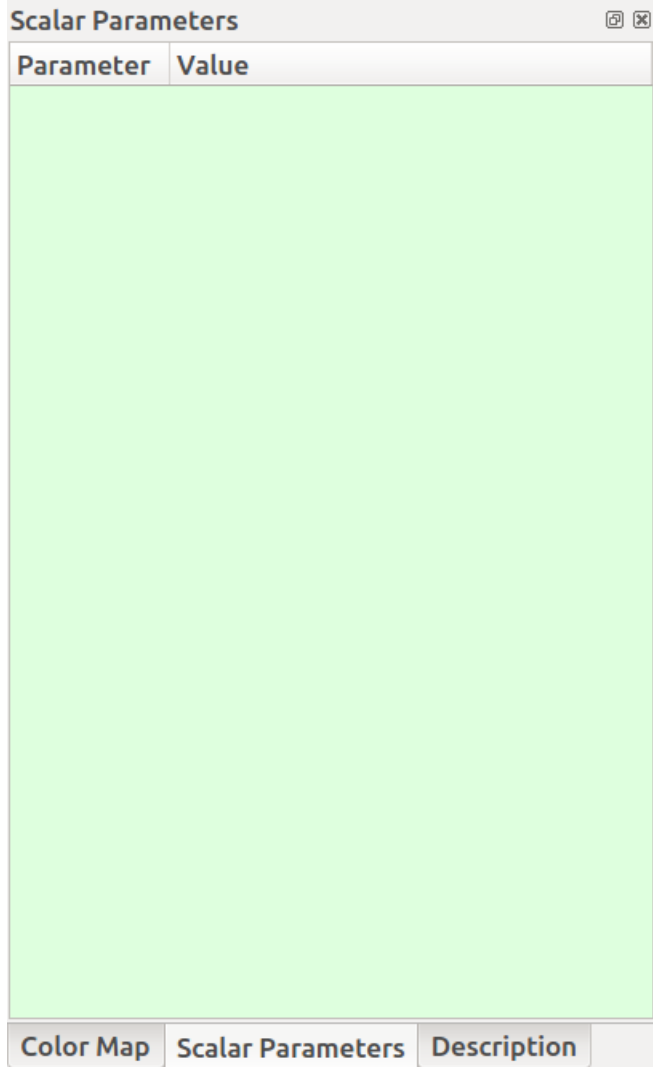


When double-clicking on a material sphere, a dialog appears to configure a custom material.

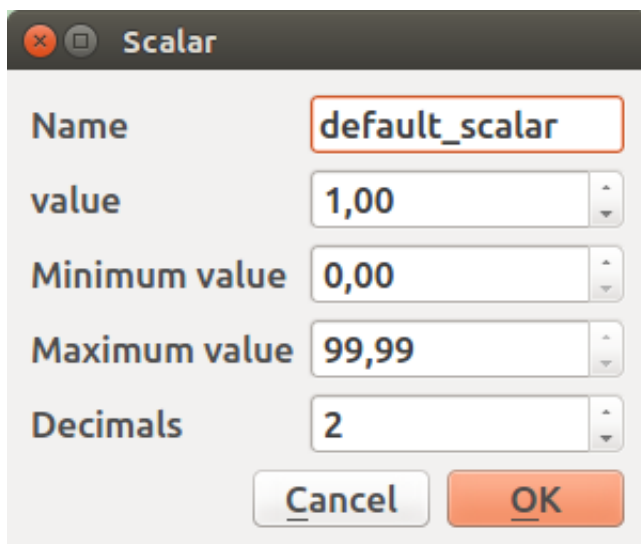
The ambient, diffuse, specular, emission, shininess and transparency components of the material can be controlled.

#### 1.4.4 Scalar Parameters

A model may have critical parameters whose values need to be controlled finely. For this some graphical control are possible using the Scalar Parameters sidebar. In this bar, you can create a scalar parameter by defining a name, a type (bool, integer, float), some bounds. The name of the parameter will correspond to the name of the variable that will be created within the model.

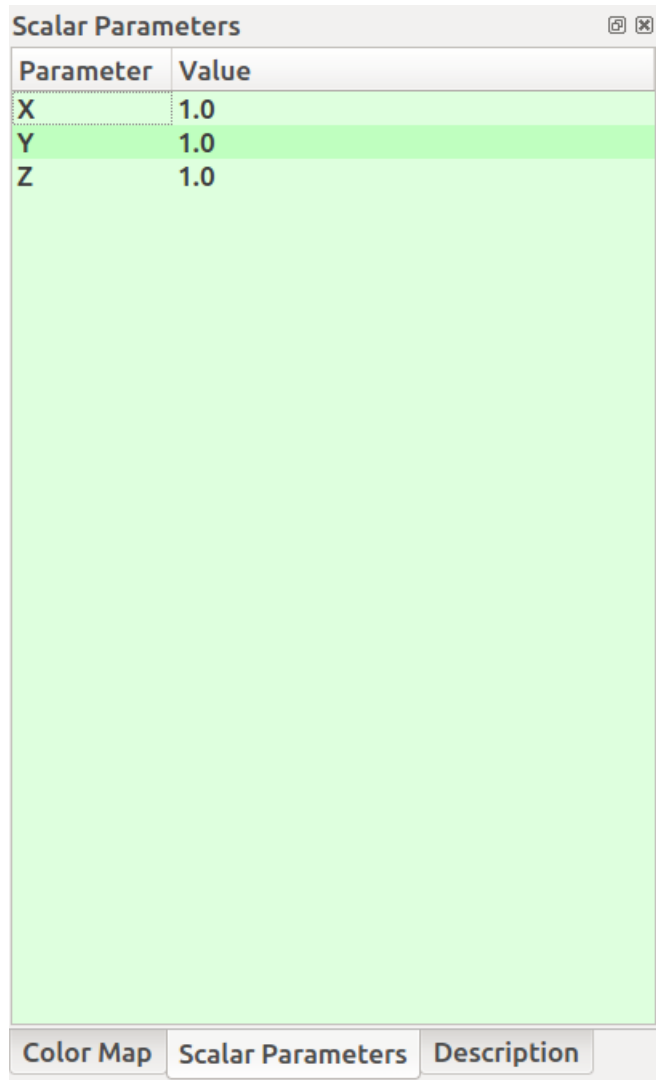


First, a right-click in the green area makes it possible to create a new scalar parameter. A definition dialog pops up.



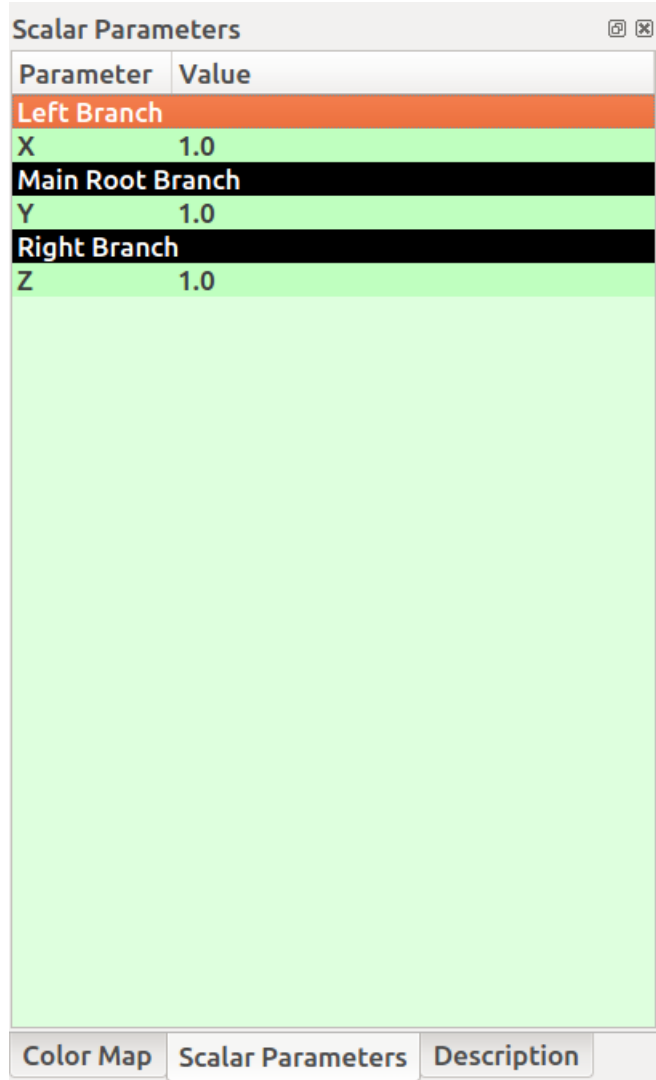


As a result different variables can be added and are accessible in the toolbar.



Parameter	Value
X	1.0
Y	1.0
Z	1.0

To organize these variables, some **categories** can be added, represented in black.



*Code:*

Within the code, the variables can be used as standard variables. In the following example, the previous X,Y, Z parameters are used as value of length of different branch of a simple structure.

```
Axiom: B[+A][-F(Z)]

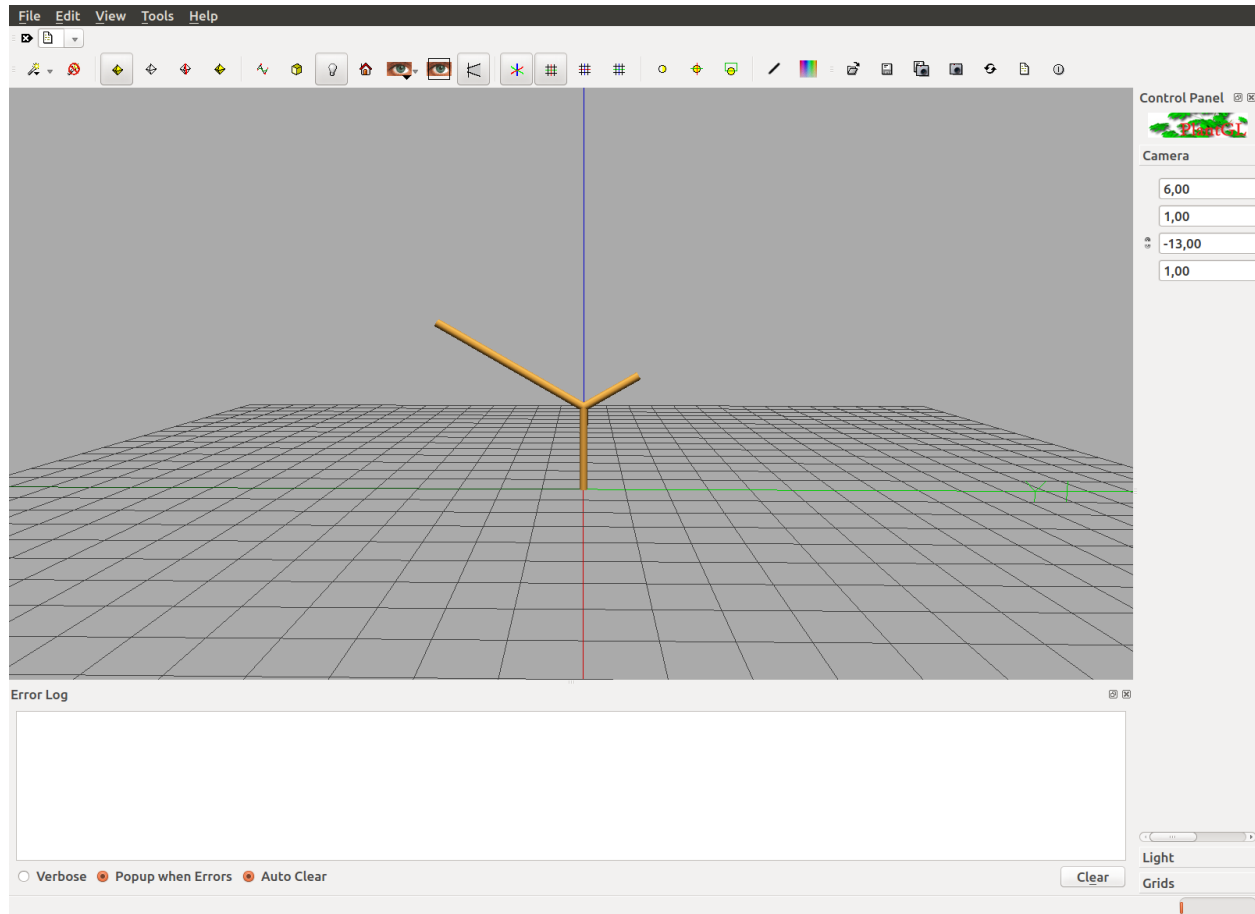
production:

interpretation:
A --> F(X)
B --> F(Y)

endlsystem
```

Then, with the code above, double left-click on the values at the right, play with the slider that appeared and click on **Run** or **Animate**.

The render on PlantGL should display something like this (with X=4, Y=2 and Z=1.5):



The values put on in the *Scalar Parameters* widget are directly modified into the code and then displayed on screen on request.

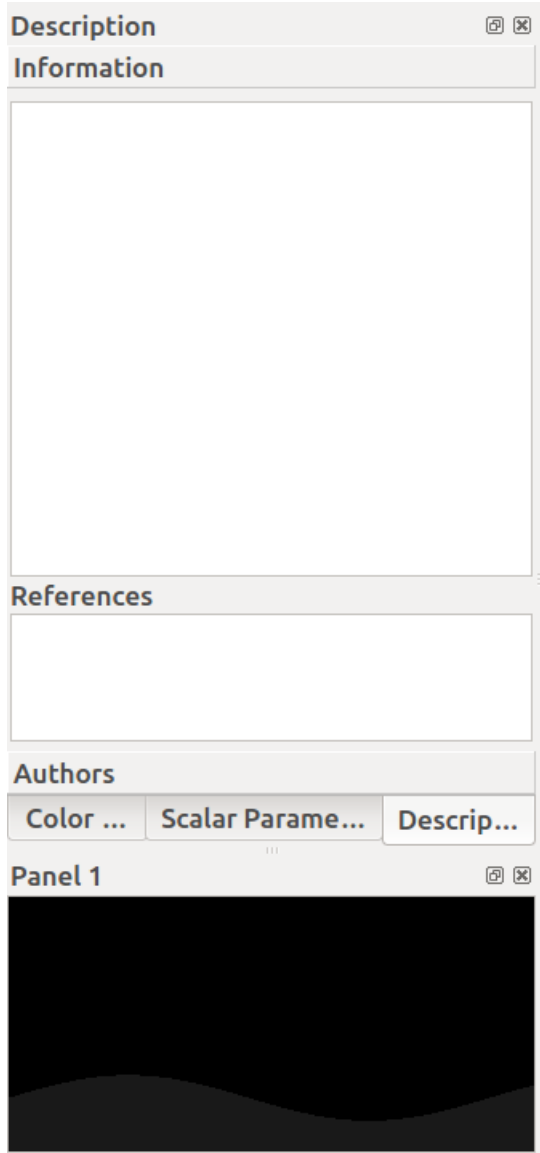
To avoid modifying the values and clicking each time on the **Run** button, the **Auto-Run** feature can be activated in the menu *L-systems > Auto-Run*. Every modification of the value will relaunch automatically the simulation.

## 1.4.5 Custom Curves

### Enable the Curve Panel

First of all, you need to display the widget **Panel 1**. To do this, right click on an empty space in the top toolbar and click on **Panel 1** if it's disabled.

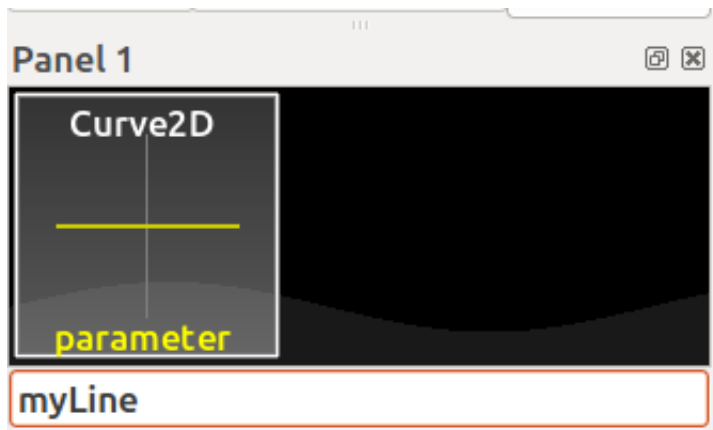
The panel is usually located below the Sidebar Tools:



but you can drag this widget anywhere you want in the window for your needs.

### Create a Bezier curve

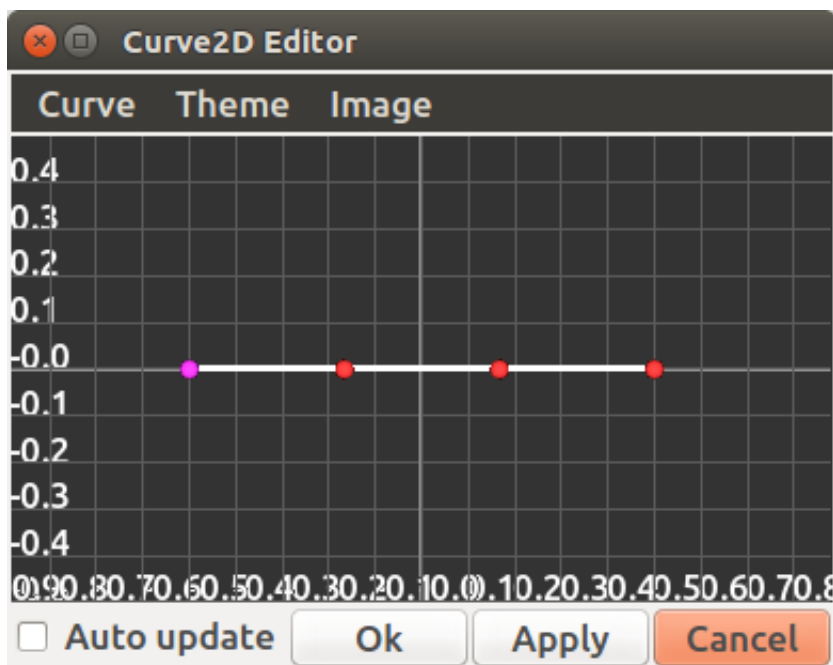
To create a custom curve, just right-click in the black panel and select “*New item > Curve2D > BezierCurve*”



A line edit appears at the bottom of the panel to name your curve and confirm it with *Enter*. You can rename your curve anytime by right-clicking on the curve component and on “*Rename*”.

### Configure a curve component

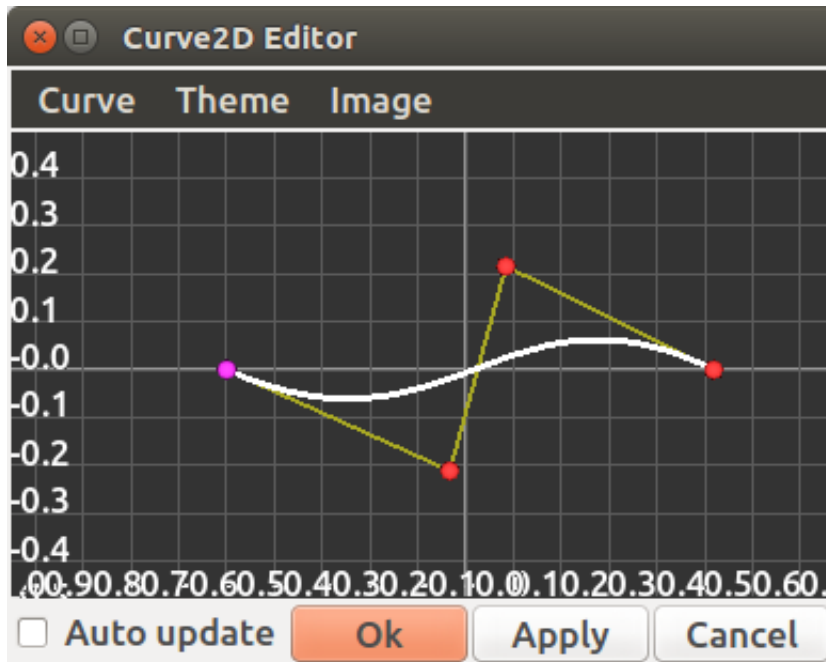
When double left-clicking on your curve component, a new pop-up appears and shows:



In this interface you can:

- *Hold Left Click* on a dot and drag it to change the curvature of the curve
- *Double Left Click* to create a new checkpoint for the curve
- *Double Right Click* on a dot to delete the selected checkpoint
- *Wheel Mouse* to zoom / unzoom in the interface
- *Hold Left Click* in the black area to shift the curve on the screen

Exemple:



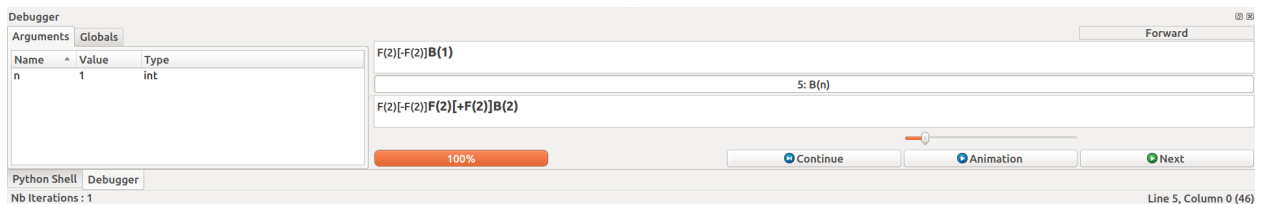
When you're satisfied with your curve configuration, you can click on the **Apply** button and close the pop-up.

### 1.4.6 Debugger

As you may know, the render of your project is done with PlantGL. The fact is that L-Py keep as a *string* your project and, thanks to the string, convert it into instructions to PlantGL.

With the debugger, you can see step by step what is contained in that string and check what's going, to do so, click on the **Debug** button in the top toolbar.

You'll see a new tab "Debugger" opened at the bottom of L-Py:



At the top, you can see the string representing your project at the beginning of the current step and below, the string being transformed into by the rules of your project.

The exemple above can be tested with that code:

```
Axiom: B(0)
derivation length: 4

production:

B(n):
    if (n % 2):
        produce F(2) [+F(2)]B(n + 1)
    else:
        produce F(2) [-F(2)]B(n + 1)
```

(continues on next page)

(continued from previous page)

endlsystem

and at the step 2 of the debug mode.

### 1.4.7 Profiler

The profiler is a widget that can help you to see how much time is being spent in each part of your program. It can be very useful into optimizing your project by fixing some parts of your program.

Profile ⌵ ⌵

Name	% Time	Total Time	Call	Inline Time	Module	Line
▼ run()	107.4304	0.271671	1	0.147651	simulatio...	690
C(x,k):(x,k)	6.0851	0.015388	14619	0.015388	02 - rand...	27
▶ plot(self,scene)	22.6577	0.057297	1	8e-06	lpystudio...	68
k(s):(s)	18.7440	0.0474	48628	0.0474	02 - rand...	36
▶ A(k):(k)	1.3564	0.00343	1142	0.003084	02 - rand...	19
k(s):(s)	0.1993	0.000504	284	0.000504	02 - rand...	52
Start()	0.0004	1e-06	1	1e-06	02 - rand...	10
<built-in method type>	0.0055	1.4e-05	3	1.4e-05		
<function eventFilter at 0x7f10c6617500>	0.0024	6e-06	3	6e-06		
<method 'disable' of '_lsprof.Profiler' obj...>	0.0004	1e-06	1	1e-06		

Profile Color Map Scalar Parameters Description

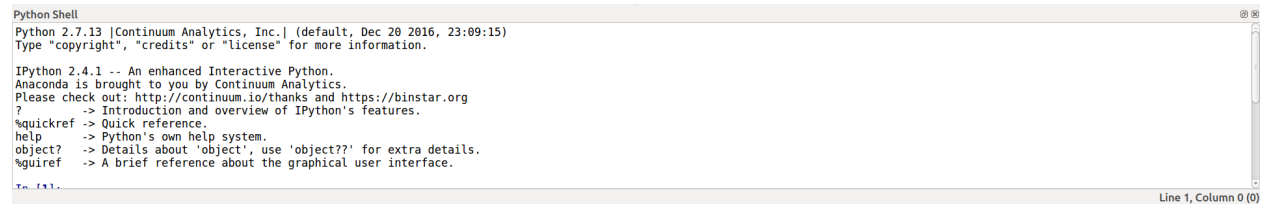
This is sorted as:

- *Name* : The name of the function
- *% Time* : The task time spent divided by the full time spent multiplied by 100
- *Call* : How much time this function has been called
- *Inline time*
- *Module* : In which module the function has been called
- *Line* : Where does the function start in its module

The *run()* function is basically the entire process, but you can find all your *rules* in this *run()* function plus the *plot()* function, which is the scene rendering function by PlantGL.

### 1.4.8 Python Shell

You can find at the bottom of L-Py a Python Shell that can be useful to display at run-time some data from your project. The Python Shell implemented looks familiar to a simple shell if you're used to a Linux or Mac System:



```
Python Shell
Python 2.7.13 [Continuum Analytics, Inc.] (default, Dec 20 2016, 23:09:15)
Type "copyright", "credits" or "license()" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object'; use 'object??' for extra details.
%gui?   -> A brief reference about the graphical user interface.
```

You can find in the [Lpy Helpcard](#) all of the available commands for the Python Shell. Here will be explained all known commands at this date:

### lstring

When `lstring` is called, this command write on the shell the last computed `lsystem` string of the current simulation.

Do you remember the *Scalar Parameters* exemple ? Try to get it again and try to send the *lstring* command in the Python Shell, you should have this being returned:

```
In [1]: lstring
Out[1]: AxialTree(B[+A][-F(1.5)])
```

We can see that, here, the code has been interpreted as an **AxialTree**, which is the system module. This **AxialTree** contains custom turtle instructions (**B** and **A** here) that will be reinterpreted at the end of the computing as **F(Y value)** for **B** and **F(X value)** for **A**.

---

**Note:** Why the X and Y variables has not been replaced by its value is because it is an interpretation of the L-Py program of the element and not a production that replaces the variable !

---

### lsystem

When `lsystem` is called, this command write on the shell the reference to the internal `lsystem` object representing the current simulation.

```
In [1]: lsystem
Out[1]: <openalea.lpy.__lpy_kernel__.Lsystem at 0x7f3b5f0d0890>
```

### window

When `window` is called, this command write on the shell the reference to the `lpy` widget object.

```
In [1]: window
Out[1]: <openalea.lpy.gui.lpystudio.LPyWindow at 0x7f3b866409d0>
```

The *lsystem* and *window* commands can be useful if you need to know some advanced details on the current `lsystem` object represented on-screen.

## 1.5 L-Py Turtle basic primitives

The L-string can be parsed and its modules can be interpreted as commands for a 3D turtle. A number of basic primitives that are understood by the turtle are explained in this page :



Table 1: Table of primitives

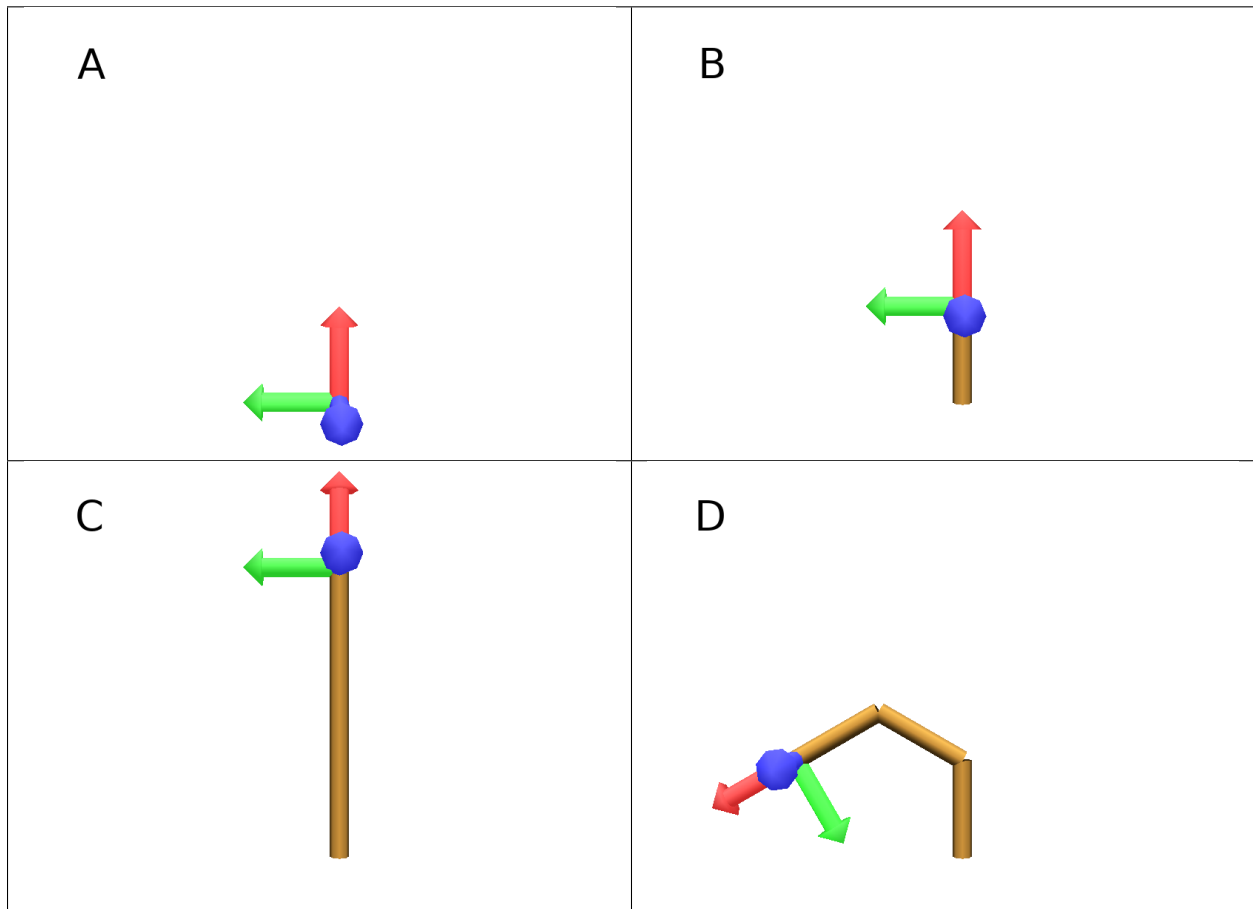
<i>F</i>	<i>width</i> (!)	<i>rotation</i> (/)	<i>MoveTo</i>	<i>nF</i>
<i>f</i>	<i>color</i> (;)	<i>rotation</i> (\)	<i>MoveRel</i>	<i>LineTo</i>
<i>@O</i>	<i>color</i> (;)	<i>rotation</i> (^)	<i>Pinpoint</i>	<i>LineRel</i>
<i>@o</i>	<i>setColor</i>	<i>rotation</i> (&)	<i>PinpointRel</i>	<i>OLineTo</i>
<i>@B</i>	<i>branching</i> (□)	<i>rotation</i> (+)	<i>setHead</i> (@R)	<i>OLineRel</i>
<i>@b</i>	<i>polygons</i> (.)	<i>rotation</i> (-)	<i>EulerAngles</i>	<i>SetGuide</i>
<i>width</i> (—)	<i>Frame</i>	<i>Rescaling</i>		<i>generalisedCylinders</i> (@Gc and @Ge)

## 1.5.1 Constructing basic shapes with the Turtle

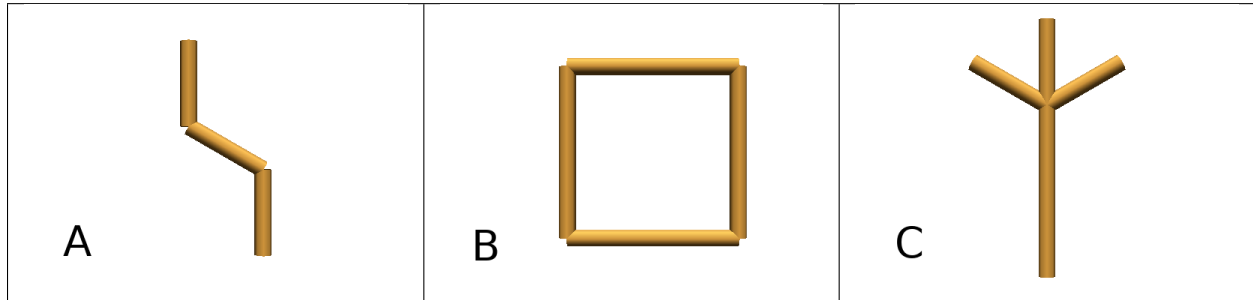
### Simple Turtle instructions

The Turtle is a geometric tool defined by a reference frame that can be moved and oriented in space using a number of instructions. As an illustration the following examples shows how the *F* primitive can be used to draw cylinders and + to rotate counterclockwise.

```
#Turtle initial position (Fig. A)
Axiom: F                      # (Fig. B)
Axiom: FFF                    # (Fig. C)
Axiom: F+F+F                  # (Fig. D)
```



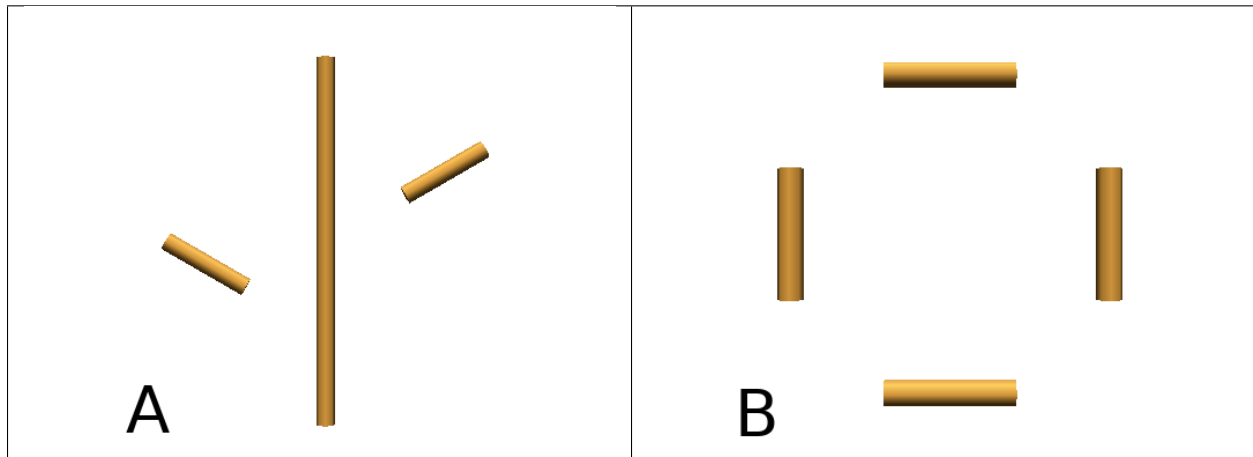
Axiom: <code>F+F-F</code>	<i># (Fig. A)</i>
Axiom: <code>FF-(90)FF-(90)FF-(90)FF</code>	<i># (Fig. B)</i>
Axiom: <code>FF[-F][+F]F</code>	<i># (Fig. C)</i>



The **F** primitive moves the Turtle and draws a cylinder of one unit.

To move the Turtle without drawing something, **f** should be used.

Axiom: <code>F[+fF]F[-fF]FF</code>	<i># (Fig. A)</i>
Axiom: <code>+(90)F-(45)f-(45)F-(45)f-(45)F-(45)f-(45)F</code>	<i># (Fig. B)</i>



**F** can take arguments (of type float). The first argument defines the length of the cylinder (default value = 1). By default, the radius of the cylinder is by set to 0.1.

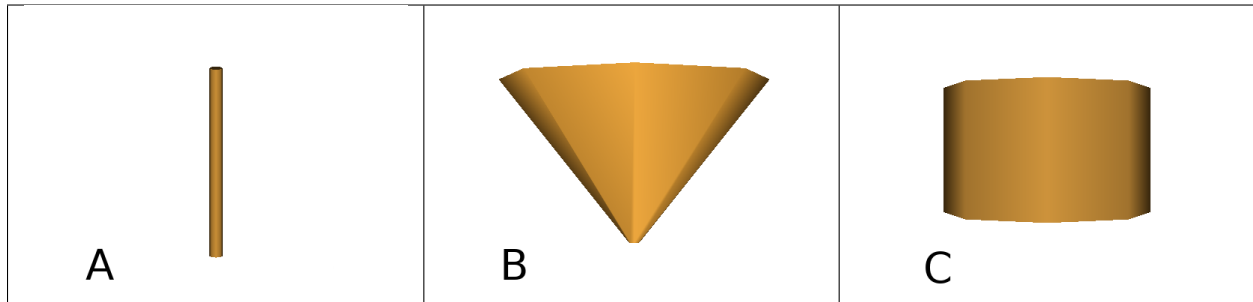
Axiom: <code>F(3)</code>	<i># (Fig. A)</i>
--------------------------	-------------------

Second argument defines the radius at the top of the cylinder (the bottom radius being defined by the current width value of the Turtle state)

Axiom: <code>F(3, 2.5)</code>	<i># (Fig. B)</i>
-------------------------------	-------------------

To change the value of the Turtle's width before applying the **F** command, the `_` primitive can be used:

Axiom: <code>_(2.5)F(3, 2.5)</code>	<i># (Fig. C)</i>
-------------------------------------	-------------------



Similarly, the `+` symbol specifies a rotation whose angle is given by its first parameter.

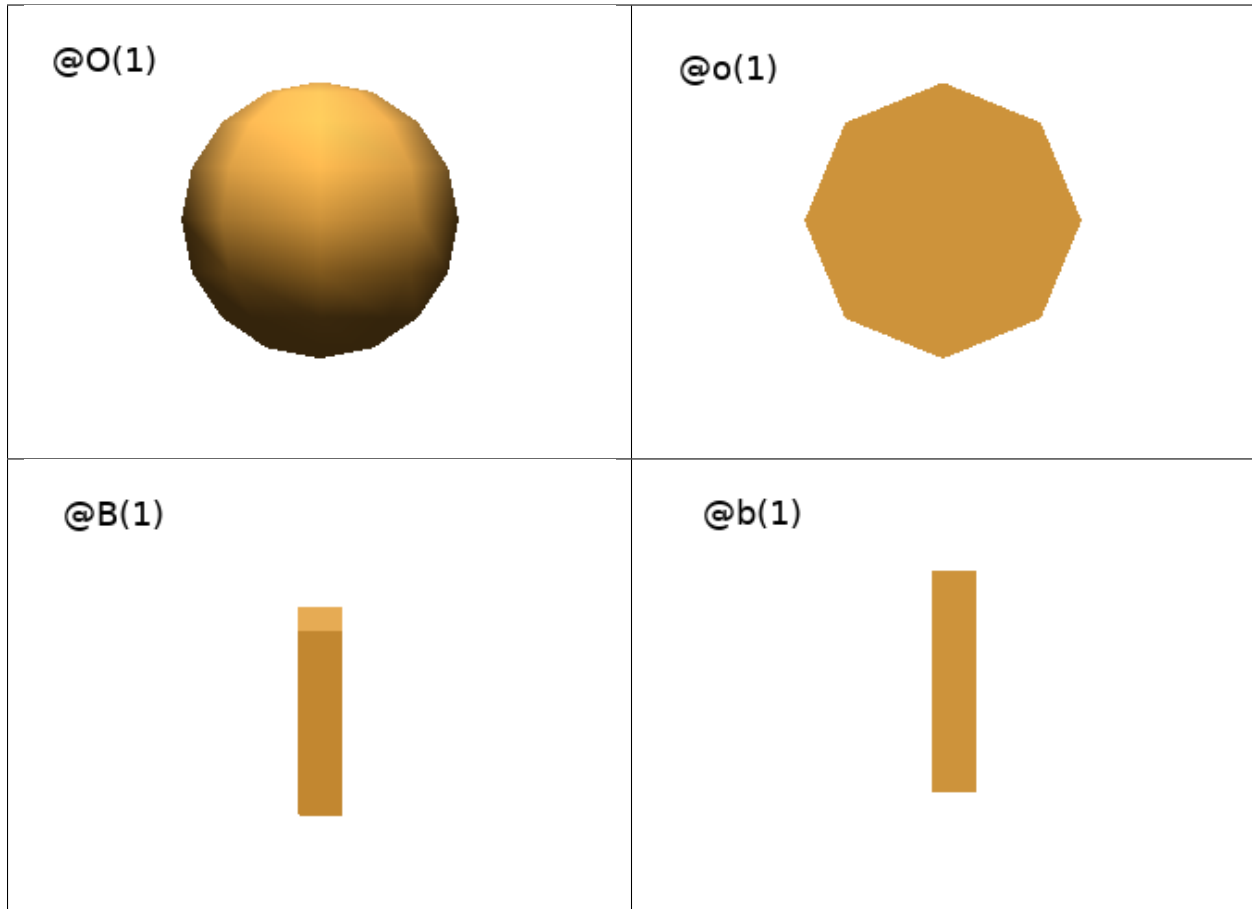
Other basic geometric primitives make it possible to draw other predefined shapes :

```
Axiom: @O(1)      # Draws a sphere at the Turtle's position.
# It can take one argument which is the radius of the sphere.

Axiom: @o(1)      # Draws a circle at the Turtle's position.
# It can take one argument which is the radius of the circle.

Axiom: @B(1)      # Draws a box at the Turtle's position.
# It can take two arguments which are the length of the edges and the topradius.

Axiom: @b(1)      # Draws a quad at the Turtle's position.
# It can take two arguments which are the length of the edges and the topradius.
```



Text can be displayed using the @L primitive.

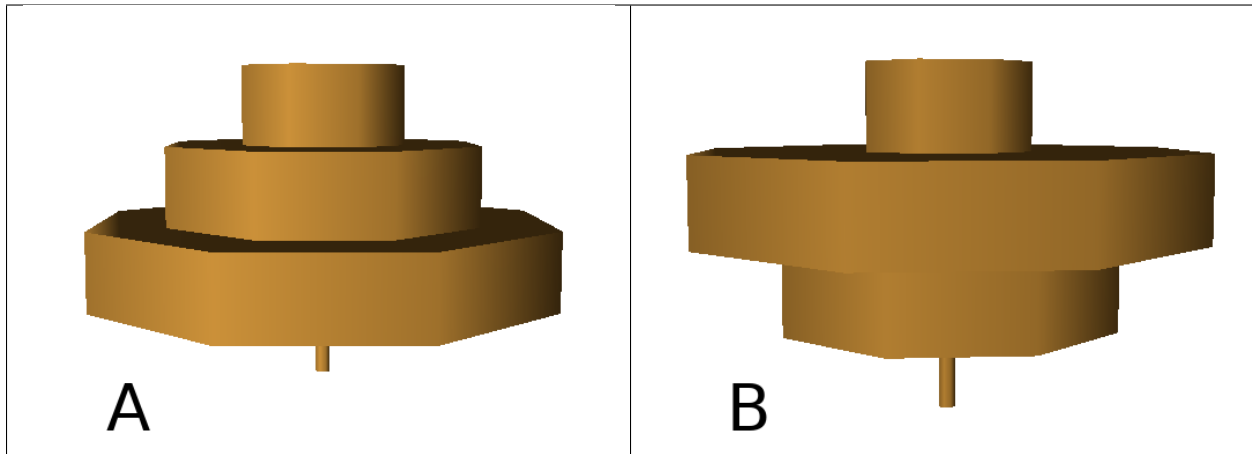
```
Axiom: @L("Some text", 18)      # Draws a text Label at the Turtle's position.
# It can take two arguments which are the text to display and it's size.
```

## Controlling width and color of primitives

### Changing the width

The width of the shapes can be increased (resp. decreased) using \_ (resp. !). These primitives increment or decrement width by 1. The default width is 0.1.

```
Axiom: F_ _ _F!F!F      # At the beginning, the cylinder has a width of 0.1 (default)
↳ then 3.1, then 2.1 and finally 1.1 (Fig. A)
```



Alternatively, the width can be set using **setWidth** or by giving argument to **\_** or **!**

```
Axiom: F_(2)F!(3)F!(1)F          # (Fig. B)
```

Download the example : `width.lpy`

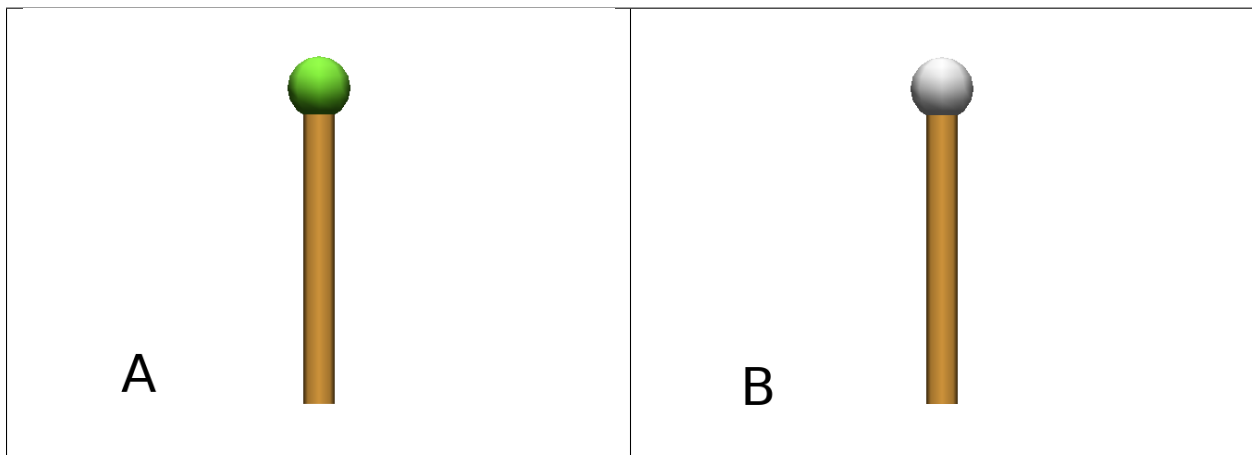
### Color System

To use color system, it is necessary to set materials with the **Color Map** window (*Color Map*).

The semicolon (;) is used to increase the current material index (Fig. A) and the comma (,) to decrease it (Fig. B). A argument can be set to specify the index of the material to use.

```
Axiom: F(2) ; @O(0.2)  # (Fig. A)
# Or equivalently:
Axiom: F(2) ; (2) @O(0.2)

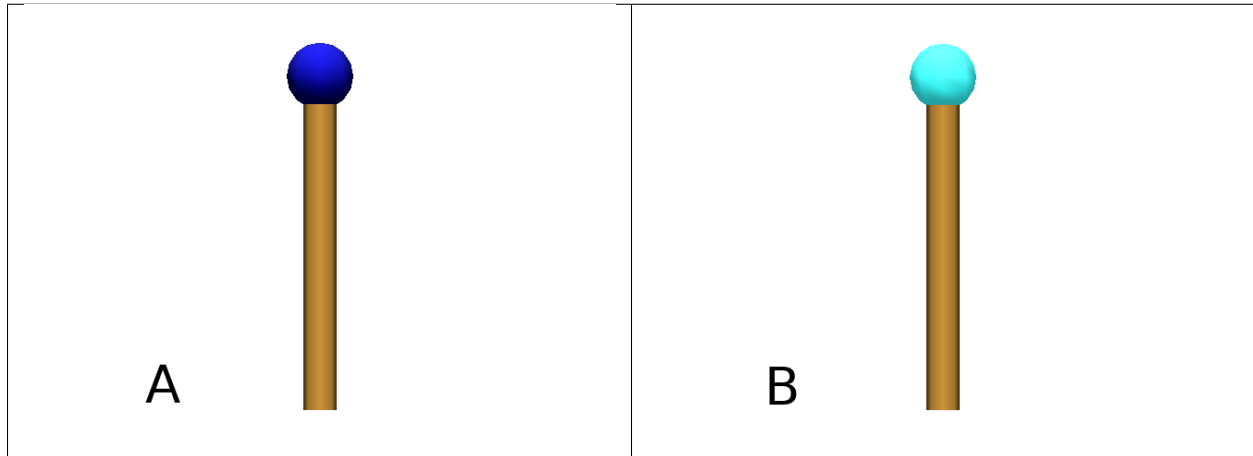
Axiom: F(2) , @O(0.2)  # (Fig. B)
# Or equivalently:
Axiom: F(2) , (0) @O(0.2)
```



The **SetColor** primitive allow you to specify the appearance of the next primitive drawn by the turtle using either an index in the **Color Map** or directly using **red, green, blue** (or **rgba**) values as arguments (Fig. B). It is also possible to pass directly a plantgl **Material** object to specify a more complex appearance.

```
Axiom: F(2) SetColor(5) @O(0.2) # (Fig. A)
```

```
Axiom: F(2) SetColor(45, 200, 200) @O(0.2) # (Fig. B)
```



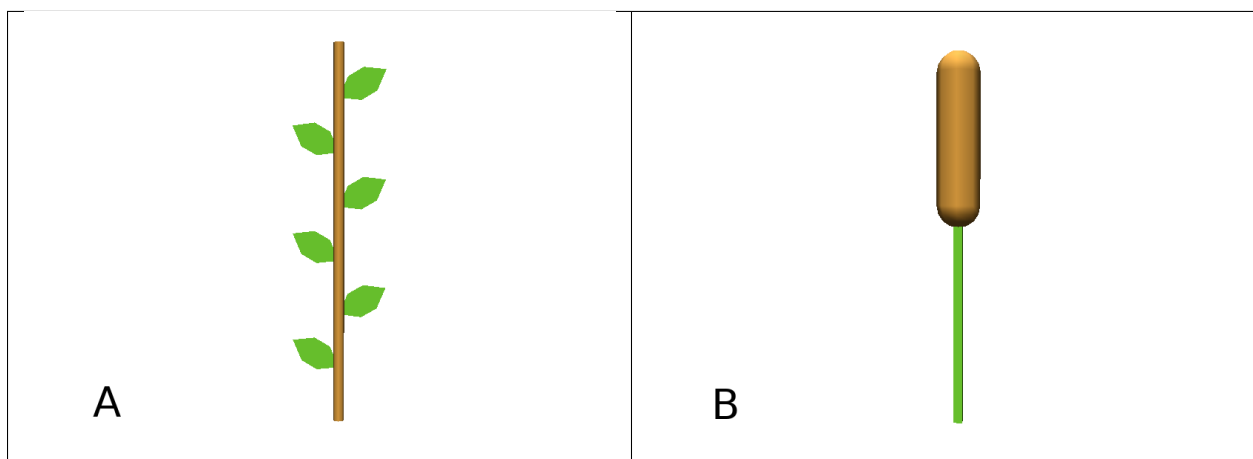
### Drawing more complex shapes

Specific shapes can be drawn using `~`. A predefined leaf shape is available using `~l`.

```
Axiom: F[;+~l]F[;~l]F[;+~l]F[;~l]F[;+~l]F[;~l]F # (Fig. A)
```

```
Axiom: ;@B(5),@O(0.5)_(0.5)F(3,0.5)_(0.2)@O(0.5) # (Fig. B)
```

Download the example : `combined.lpy`



In order to draw complex shapes, some basic knowledge about the Turtle is required.

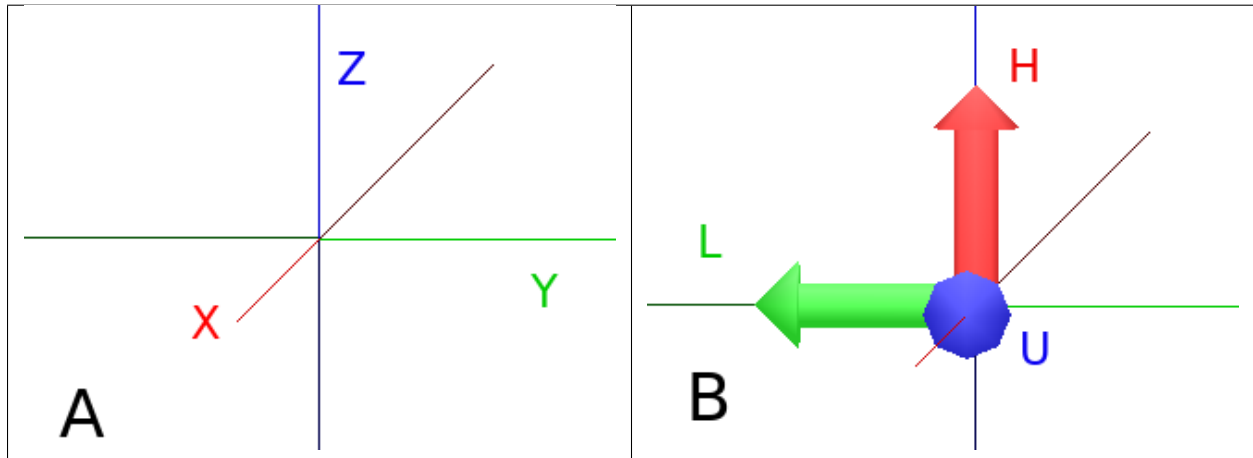
## Definition of the Turtle's reference frame (HLU)

In L-Py, screen coordinates are defined in a global reference frame  $F0 = (X,Y,Z)$  of L-Py (Fig. A).

The Turtle is defined by a reference frame (H,L,U) with respect to  $F0$  (Fig. B) and can be displayed using the primitive **Frame**

- H (Head) pointing in the direction of the Turtle's "head".
- L (Left) pointing in the direction of the Turtle's "left arm".
- U (Up) pointing in the direction of for the Turtle's back ("up").

Axiom: Frame

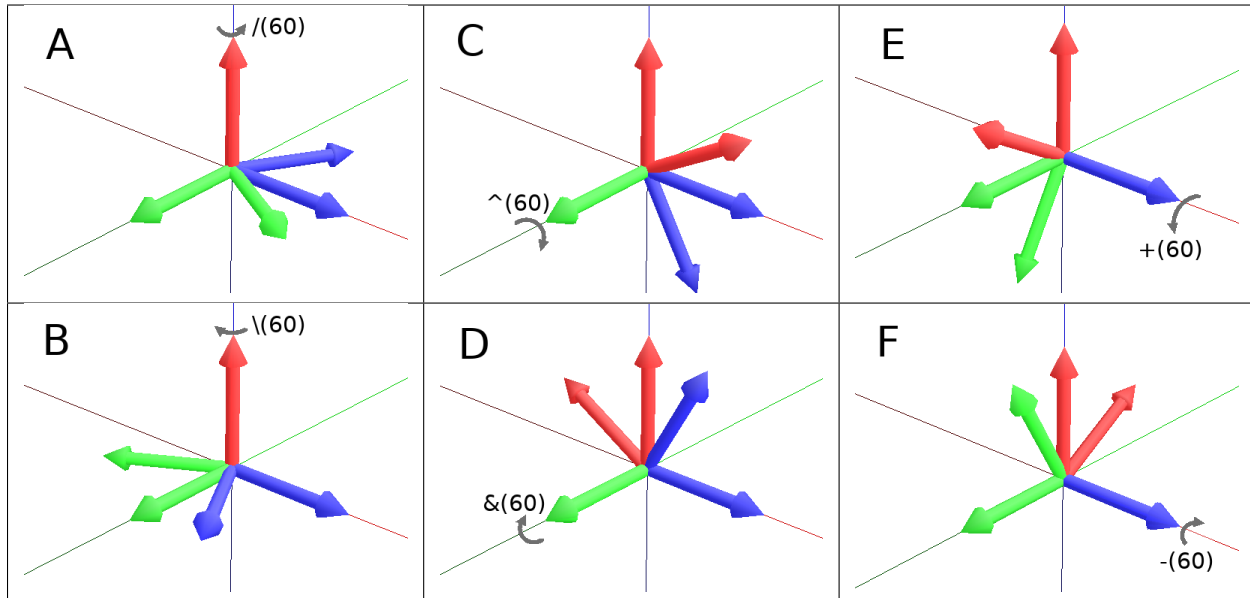


## Rotating with HLU (Main primitives)

Primitives can be used to rotate the Turtle in its current reference frame (H = Head, L = Left, U = Up, angles are expressed by default in degrees). These primitives are paired (one and it's opposite) : / and \, ^ and & and finally + and -.

```
Axiom: Frame(2) /(60) Frame(2)    # Roll left around the H axis. (Fig. A)
Axiom: Frame(2) \ (60) Frame(2)   # Roll right around the H axis. (Fig. B)
Axiom: Frame(2) ^ (60) Frame(2)   # Pitch up around the L axis. (note that the_
↪rotation is indirect) (Fig. C)
Axiom: Frame(2) & (60) Frame(2)   # Pitch down around the L axis. (note that the_
↪rotation is indirect) (Fig. D)
Axiom: Frame(2) + (60) Frame(2)   # Turn left around the U axis. (Fig. E)
Axiom: Frame(2) - (60) Frame(2)   # Turn right around the U axis. (Fig. F)
```

Download the example : `rotation.lpy`

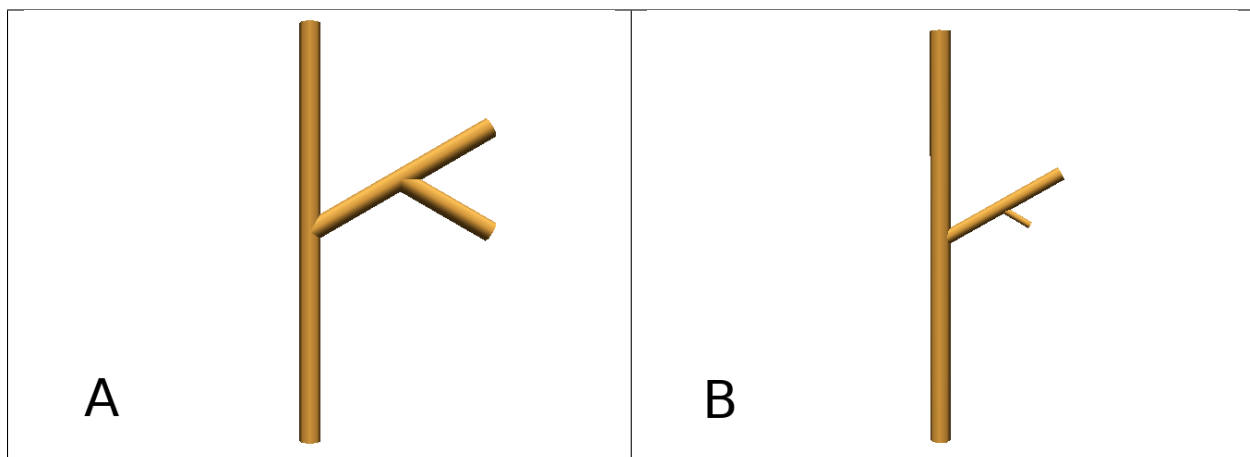


## Rescaling the Turtle

Three primitives can be used to rescale the Turtle : **SetScale**, **DivScale** and **MultScale** (shorter symbols are **@D**, **@Dd** and **@Di** respectively) **SetScale** sets the scale to the value in argument. **DivScale** (resp. **MultScale**) divides (resp. multiplies) the current scale by the value given in argument. The first image is the initial shape (Fig. A) and the second one is the same shape where the branches are rescaled (Fig. B).

```
Axiom: FF[-F[-F]F]FF                                     # (Fig. A)
Axiom: @D(2)FF[@Dd(1.5)-F[@Di(0.5)-F]F]FF               # (Fig. B)
```

Download the example : `scale.lpy`



## How to draw polygonal shapes ?



### Basic method

Turn and move forward : Here, at each +, the Turtle does a rotation of the number of degrees indicated in arguments around the U axis

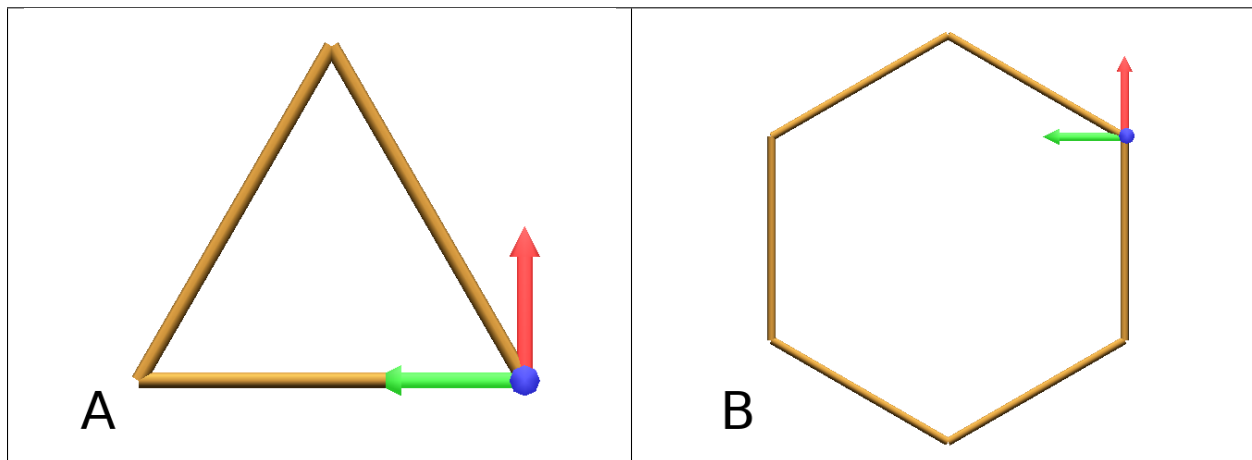
```
Axiom: Frame(2)+(30)F(5)+(120)F(5)+(120)F(5)    # (Fig. A)
```

Download the example : `polygons.lpy`

### Procedural method

A loop construct can be used to produce the L-string specifying the polygon

```
Axiom: Frame(2)+F(5)+F(5)+F(5)+F(5)+F(5)+F(5)    # (Fig. B)
# Or equivalently:
Axiom:
    nproduce Frame(2)
    for i in range(6):
        nproduce +F(5)
```



### Filled polygons

Polygon can be drawn by using {} and positioning a series of dots ('.') in space, corresponding to the consecutive vertices of the polygon (Fig. A)

Here, the instruction starts by positioning the first vertex of the polygon at the origin of the reference frame

```
Axiom: Frame _(0.05), (2){.f(3).-(90)f(3).-(90)f(3).-(90)f(3)}
```

The contour of the polygon can be drawn by using **F** instead of **f**. In this case, dots (.) are no longer required after each **F** (Fig. B)

```
Axiom: Frame _(0.05), (2){.F(3)-(90)F(3)-(90)F(3)-(90)F(3)}
```

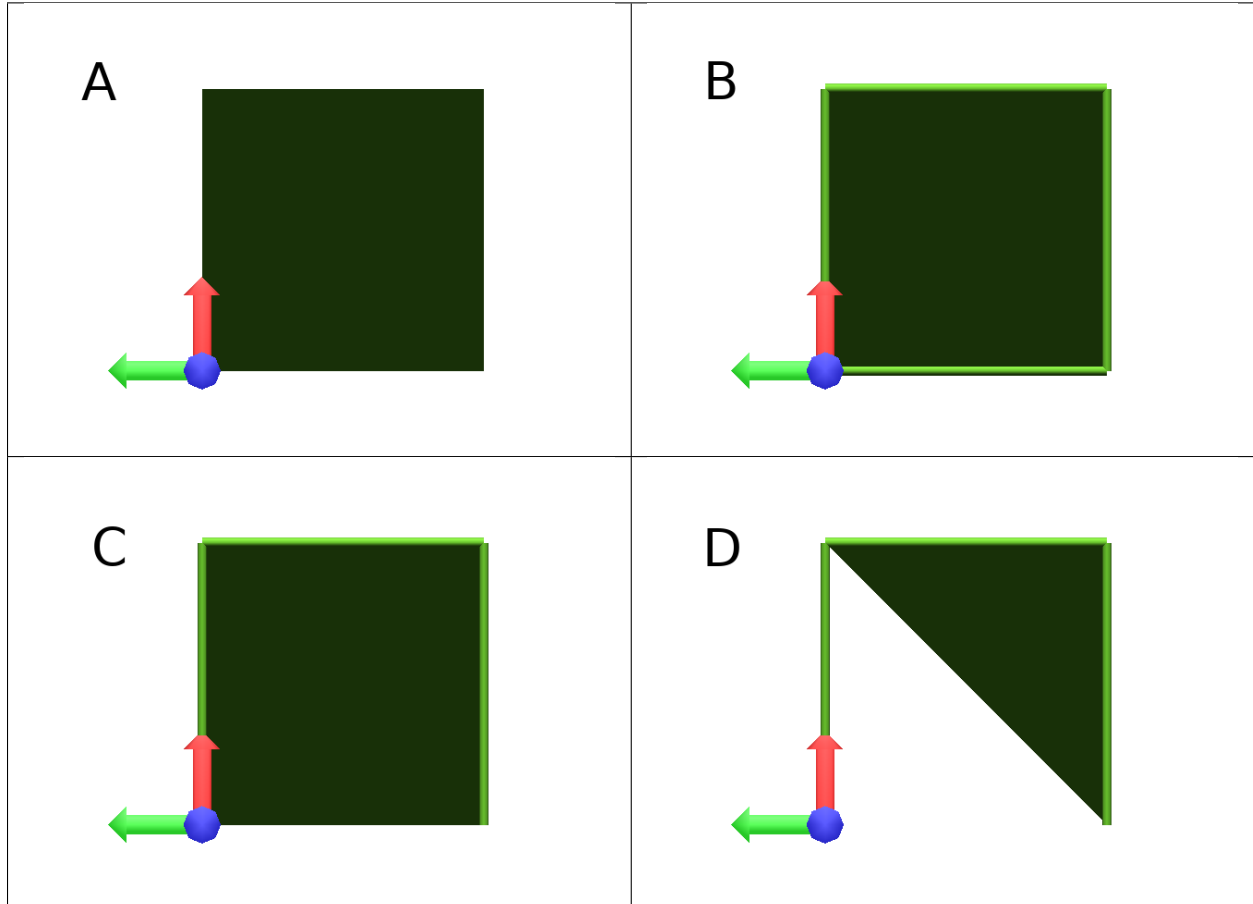
The number of edges drawn can be different from the number of vertices (Fig. C)

```
Axiom: Frame _(0.05), (2){.F(3)-(90)F(3)-(90)F(3)}
```

Note that if the first dot/point is omitted, the polygon is not closed (Fig. D)

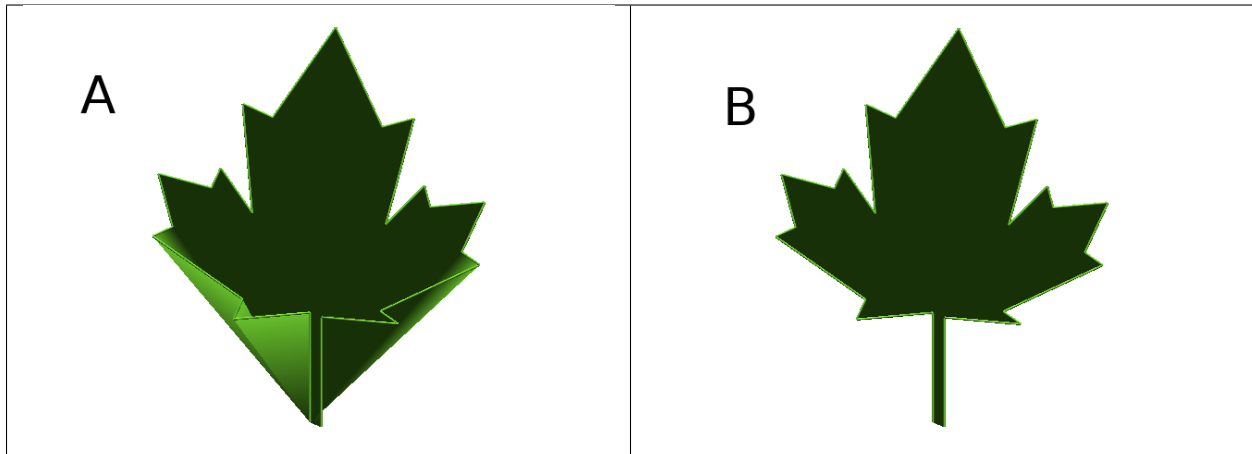
```
Axiom: Frame _(0.05), (2) {F(3)-(90)F(3)-(90)F(3)}
```

Download the example : `filledPolygons.lpy`



Filling concave objects requires to use a smarter filling procedure. This can be achieved by using a **True** argument to the polygon drawing (by default the argument is **False**)

```
# Naive procedure to fill the concave form: (Fig. A)
Axiom: _(0.01), (2) { .F+(95)F(0.7)-(120)F(0.2)+(80)F-(120)F(0.2)+(80)F(0.5)
-(120)F(0.5)+(80)F(0.2)-(120)F(0.5)+(150)F-(120)F(0.3)+(80)F -(120)F+(80)F(0.3)
-(120)F +(150)F(0.5)-(120)F(0.2)+(80)F(0.5)-(120)F(0.5)+(80)F(0.2)-(120)F+(120)F(0.2)
-(150)F(0.7)+(95)F} (False)
# while with a smarter procedure: (Fig. B)
Axiom: _(0.01), (2) { .F+(95)F(0.7)-(120)F(0.2)+(80)F-(120)F(0.2)+(80)F(0.5)
-(120)F(0.5)+(80)F(0.2)-(120)F(0.5)+(150)F-(120)F(0.3)+(80)F -(120)F+(80)F(0.3)
-(120)F +(150)F(0.5)-(120)F(0.2)+(80)F(0.5)-(120)F(0.5)+(80)F(0.2)-(120)F+(120)F(0.2)
-(150)F(0.7)+(95)F} (True)
```



### Branching system

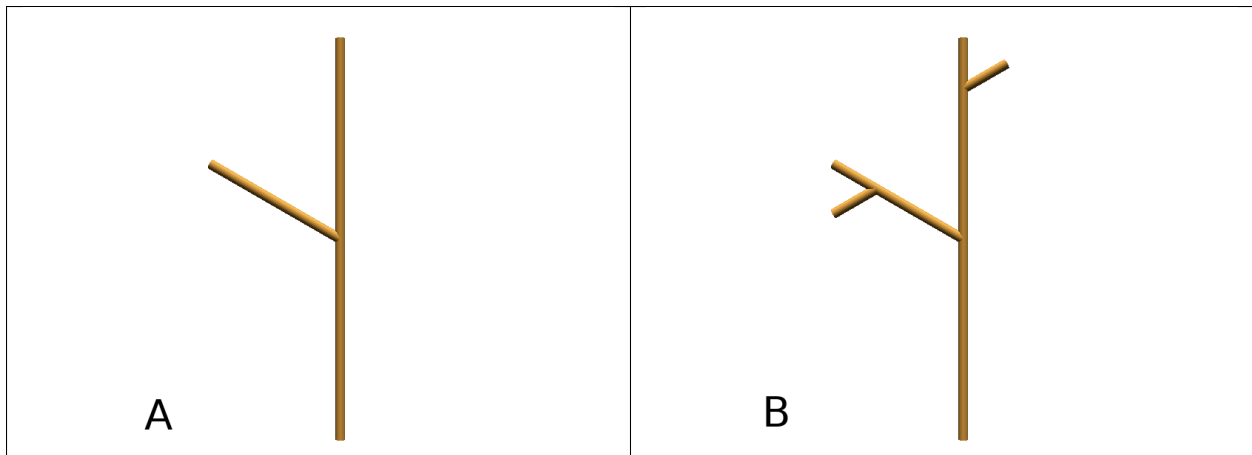
Brackets makes it possible to specify branches. Before each opening bracket, the Turtle's current arguments (position, orientation...) are stored on the Turtle stack. These arguments are then popped back when a closing bracket is found and the drawing restarts from the popped values.

```
Axiom: F(4) [+F(3)] F(4) #(Fig. A)
```

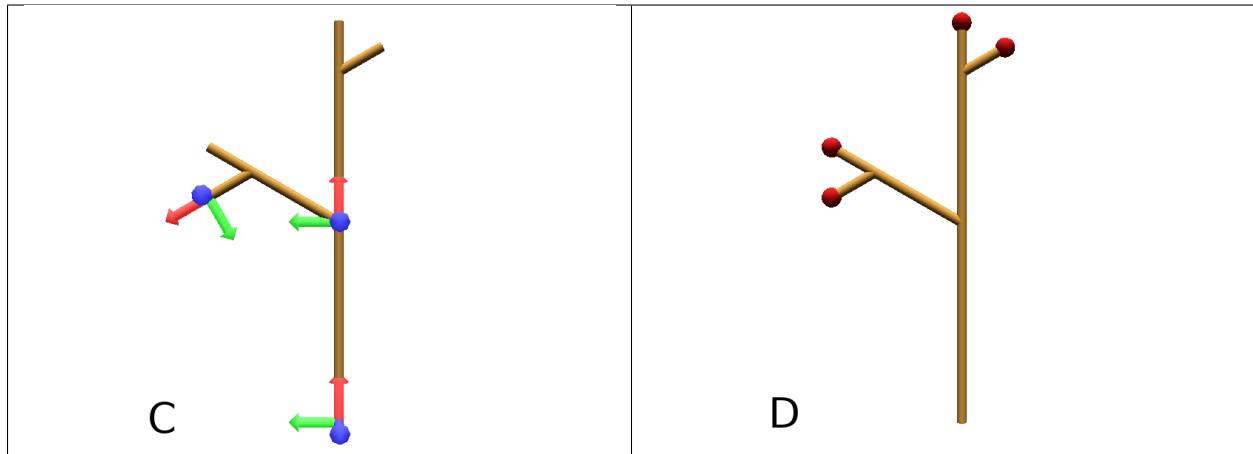
Then it's possible to nest branches inside each others :

```
Axiom: F(4) [+F(2) [+F(1)] F(1)] F(3) [-F(1)] F(1) #(Fig. B)
```

Download the example : `branching.lpy`



```
Axiom: Frame F(4) [+F(2) [+F(1) Frame] F(1)] Frame F(3) [-F(1)] F(1) # New code with ↪  
↪ Frames (Fig. C)
```



The same branching system can be augmented with other modules (**Frame**, **@O**, **@B**,...) (Fig. D)

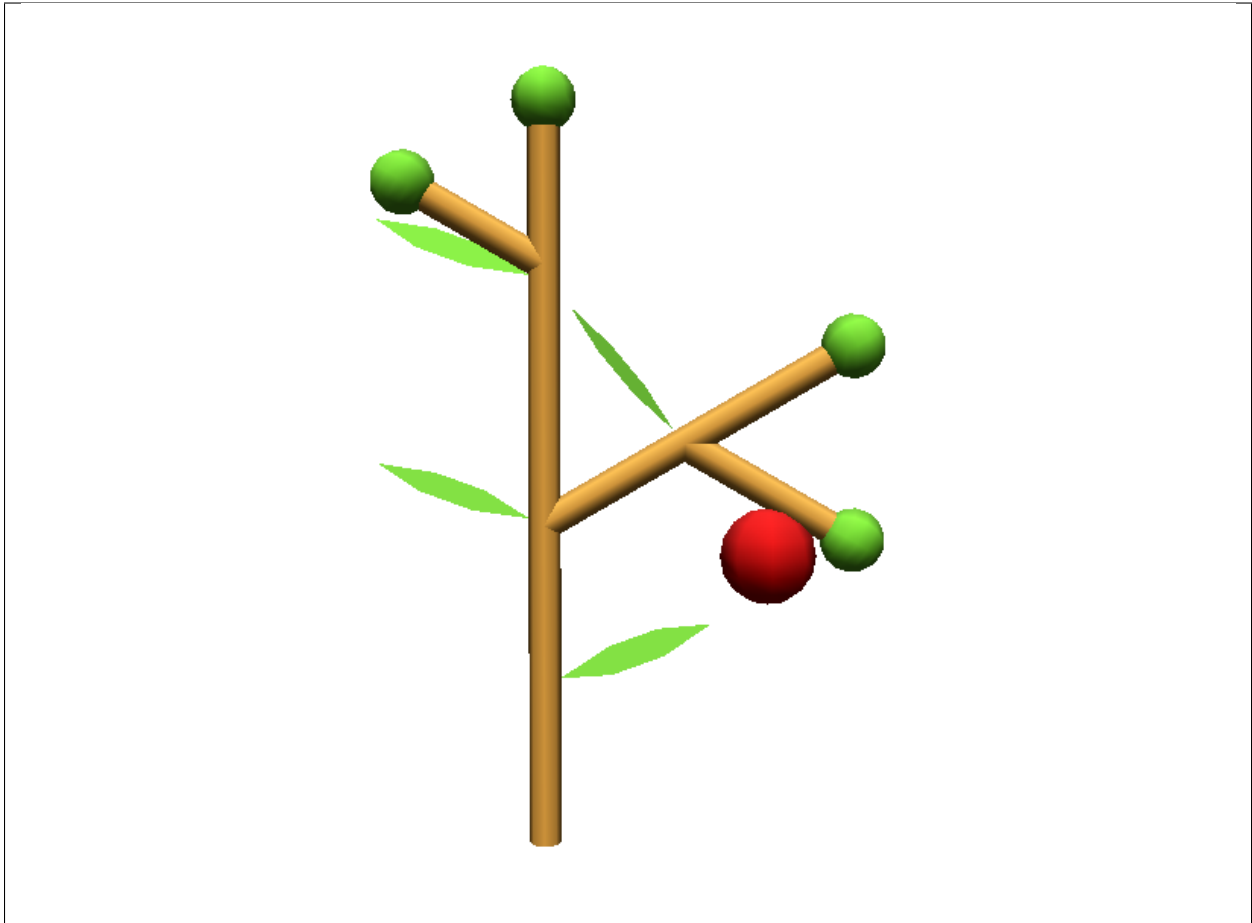
```
Axiom: F(4) [+F(2) [+F(1); (3) @O(0.2)] F(1); (3) @O(0.2)] F(3) [-F(1); (3) @O(0.2)] F(1); (3) @O(0.2)
↪ 2) # (Fig. D)
```

### A more complex combined shape

There is below a more complex shape using the previous primitives. In this example, **~l** is used. This primitive draws a leaf.

```
Axiom: F[;-(70)f(0.1)\(80)~l]F[;+(70)f(0.1)/(80)~l][-F[;+(70)f(0.1)~l][F(1.2);@O(0.
↪ 2)]-F(0.6)[-f(0.4);(3)@O(0.3)]F(0.6);@O(0.2)]
F(1.5)[;+(70)f(0.1)/(70)~l]F(0.1)[+F;@O(0.2)]F;@O(0.2)
```

Download the example : `harderExample.lpy`



## Advanced primitives

### Moving the Turtle

There are some primitives which can be used to change the Turtle's position.

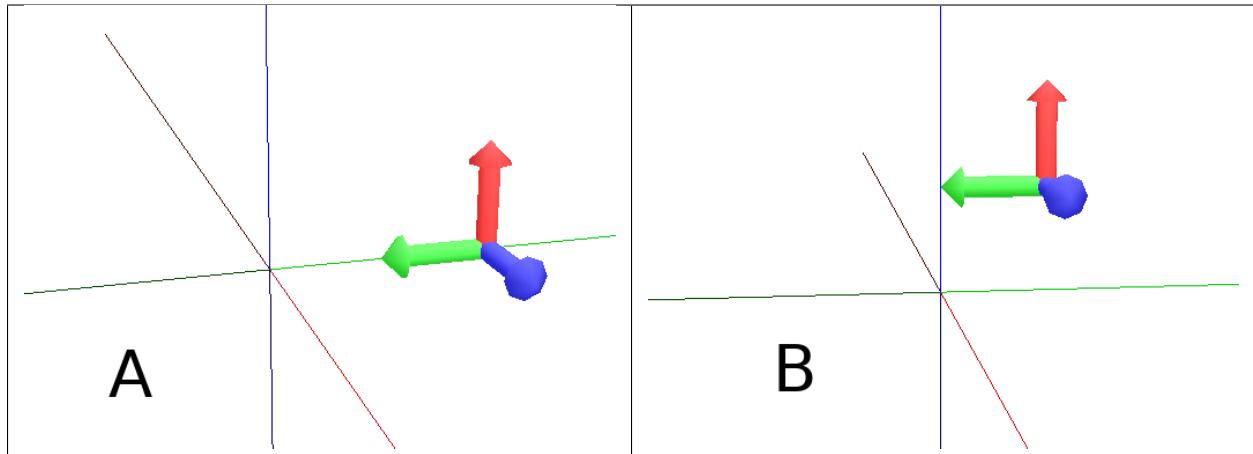
#### *MoveTo and MoveRel*

**@M** (or **MoveTo**) moves the Turtle to the position given in by its arguments. It can be three floats or a vector.

```
Axiom: @M(0,2,0) Frame  #(Fig. A)

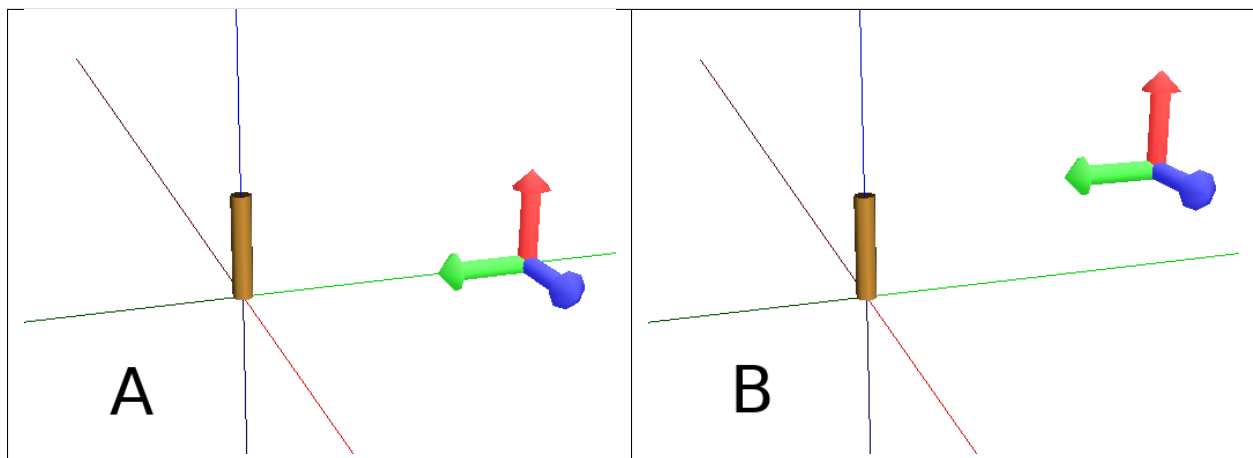
import numpy as np
v = np.array([0,1,1])
Axiom: MoveTo(v)          #(Fig. B)
```

Download the example : `movement.lpy`



**MoveRel** works almost in the same way but it moves the Turtle relatively to the current position :

```
Axiom: F MoveTo(0,3,0) Frame      #The Turtle moves to the position (0,3,0) (Fig. A)
Axiom: F MoveRel(0,3,0) Frame     #The Turtle moves along the Y axis for 3 units (Fig. B)
```



## Orient the Turtle

The Turtle's orientation can be set using some primitives.

### Pinpoint and PinpointRel

**Pinpoint** orients the Turtle towards x,y and z given in arguments. It means that the H axis (the red arrow) will point to the coordinates given. One can use also a vector.

```
Axiom: Pinpoint(1,0,0) Frame      # The H axis point to (1,0,0) (Fig. A)

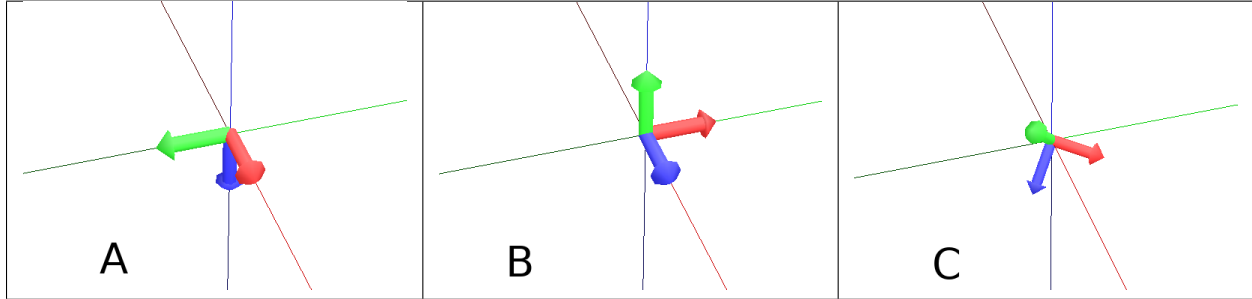
import numpy as np
v = np.array([0,1,0])
```

(continues on next page)

(continued from previous page)

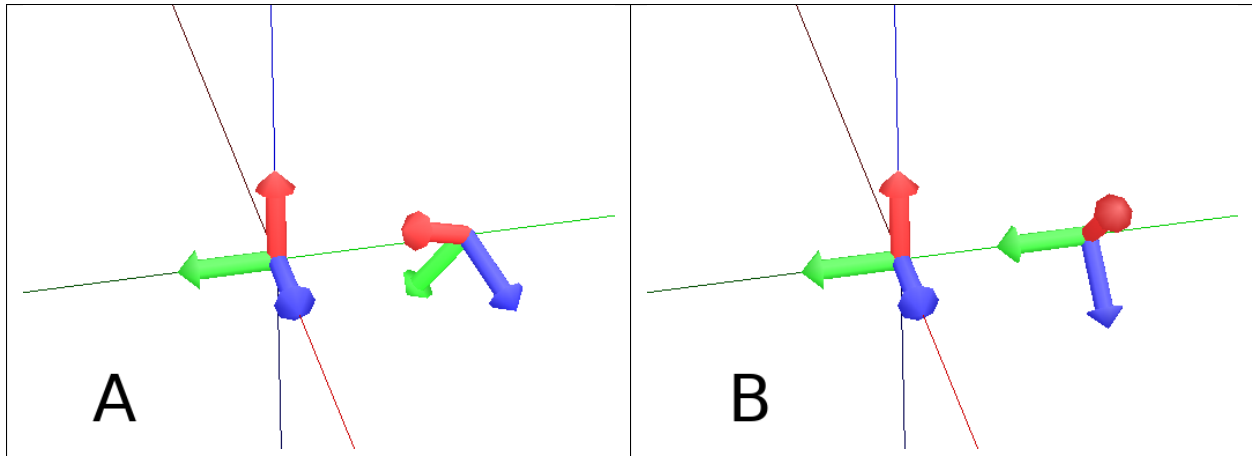
```
Axiom: Pinpoint(v) Frame           # The H axis point to (0,1,0) (Fig. B)
Axiom: Pinpoint(1,1,0) Frame       # The H axis point to (1,1,0) (Fig. C)
```

Download the example : `orientation.lpy`



Such as **MoveRel** for position, **PinpointRel** orients the Turtle relatively to the current position.

```
Axiom: Frame MoveTo(0,2,0) Pinpoint(1,0,1) Frame           # (Fig. A)
Axiom: Frame MoveTo(0,2,0) PinpointRel(1,0,1) Frame        # (Fig. B)
```



### Setting the HLU axis

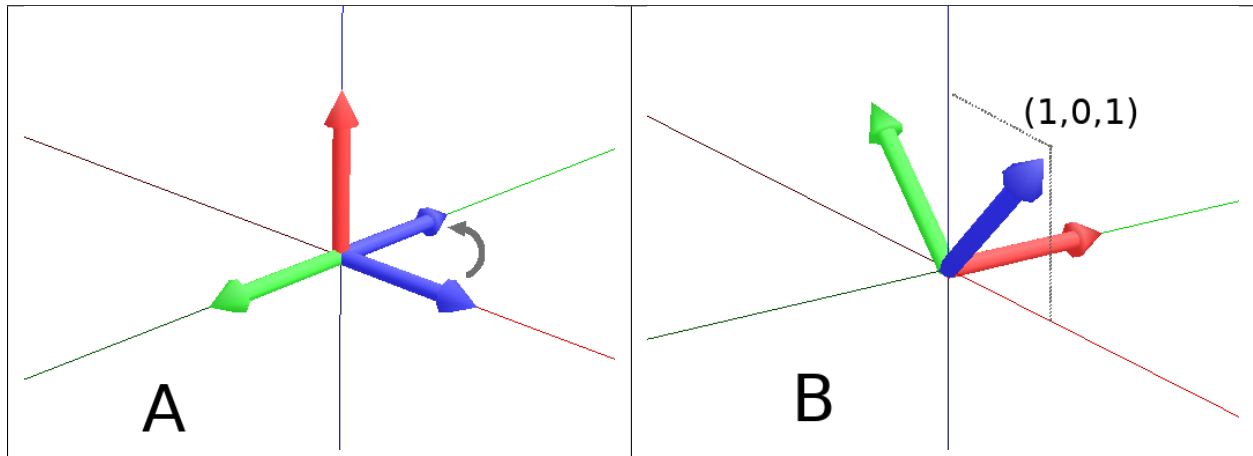
The H and U axis can be set directly using **@R** (or **setHead**). The arguments needed are 6 floats (which represent the coordinates of the two axes) or two vectors.

```
Axiom: Frame(2) @R(0,0,1,0,1,0) Frame(2)           # (Fig. A)

import numpy as np
h = np.array([0,1,0])
u = np.array([1,0,1])
Axiom: Frame(2) @R(h,u) Frame(2)                   # (Fig. B)
```

Download the example : `setHLU.lpy`

In (Fig. A), the H axis point now to  $(0,0,1)$  but it was already the case and the U axis point now to  $(0,1,0)$ . In (Fig. B), the H axis point now to  $(0,1,0)$  and the U axis point now to  $(1,0,1)$ .

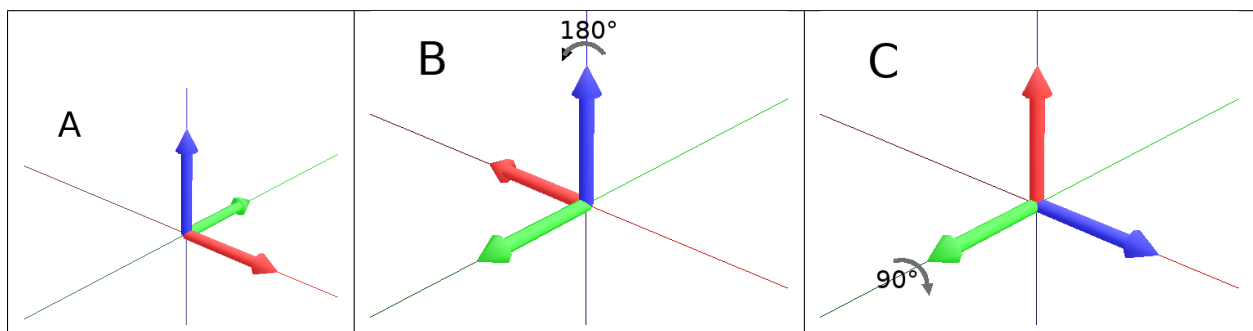


Finally, the Turtle's orientation can at any moment be set using Euler angles with the primitive **EulerAngles**. The Euler angles are defined with respect to the other global reference frame (screen coordinates). By default, the initial Turtle's frame is defined by the Euler angles  $(180,90,0)$  with respect to the original frame.

```
Axiom: Frame(2) EulerAngles(0,0,0) Frame(2) # Turtle's frame corresponds to the
↪global reference frame. (Fig. A)

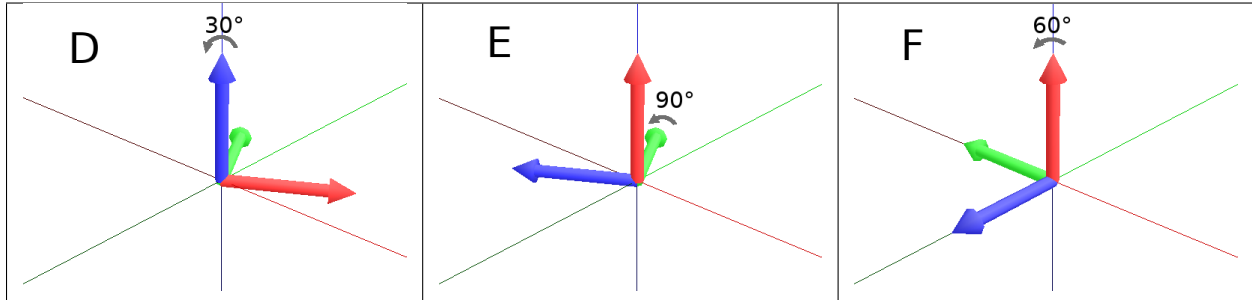
Axiom: Frame(2) EulerAngles(180,0,0) Frame(2) #180° rotation around Z axis. (Fig. B)

Axiom: Frame(2) EulerAngles(180,90,0) Frame(2) #The 90° rotation around the new Y
↪axis. (Fig. C)
#There is the initial Turtle's frame
```



```
#A succession of 3 rotations : First 30° around Z axis (Fig. D), then 90° around
↪the new Y axis (Fig. E)
#and finally 60° around the new X axis. (Fig. F)
Axiom: Frame(2) EulerAngles(30,90,60) Frame(2)
```





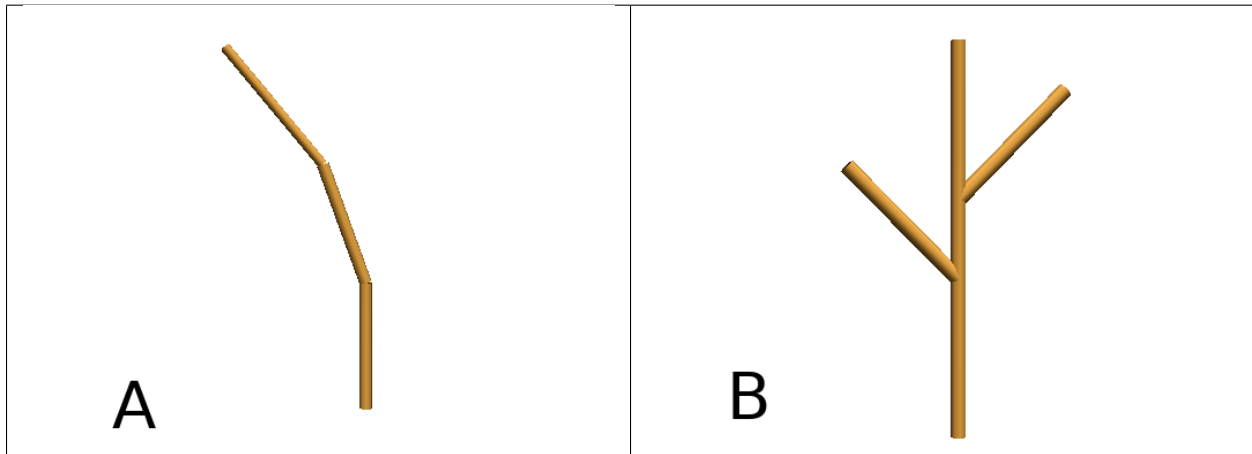
### Long path

The primitive **nF** draws  $n$  steps of cylinders ( $n$  is the first argument). The size can be passed as a second argument.

```
Axiom: nF(2,1)+(20)nF(2,1)+(20)@D(0.8)nF(3,1) # (Fig. A)
#Equivalent to FF+(20)FF+(20)@D(0.8)FFF

#It can be used to create branching shapes too.
Axiom: nF(2,1)[+(45)nF(2,1)]nF(1,1)[- (45)nF(2,1)]nF(2,1) # (Fig. B)
```

Download the example : `longPath.lpy`



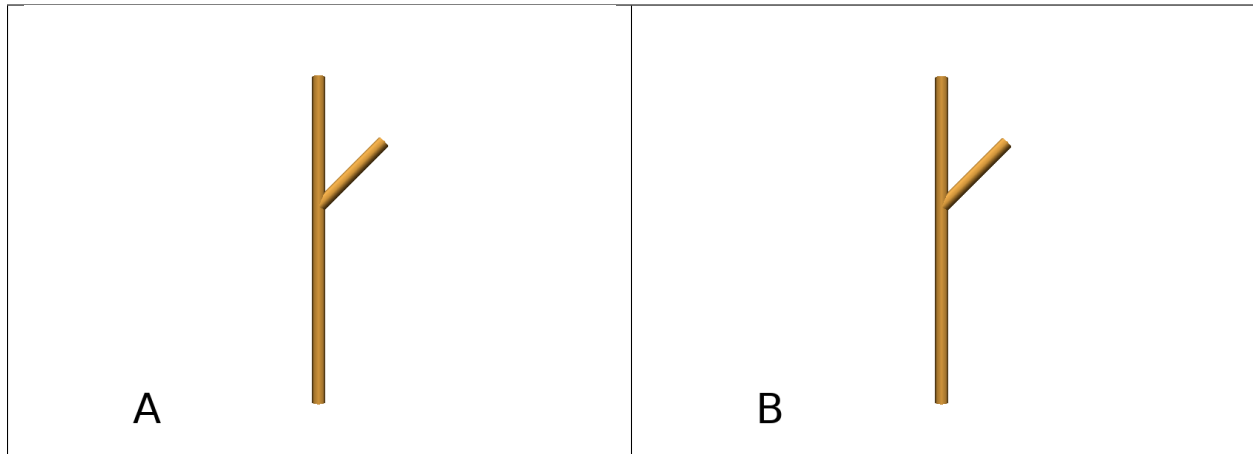
### Drawing lines

The primitive **LineTo** allows to draw a cylinder from the current position of the Turtle to coordinates given in arguments. The topdiameter can also be given as a fourth argument. Such as other primitives using coordinates, a vector can be used.

```
Axiom: LineTo(0,0,3)[LineTo(0,1,4)]LineTo(0,0,5) # (Fig. A)
```

Notice that `+`, `-`, `/` and other rotation primitives don't have any incidence on `LineTo`.

```
Axiom: LineTo(0,0,3)[+(90)LineTo(0,1,4)]-(30)LineTo(0,0,5) # (Fig. B)
```

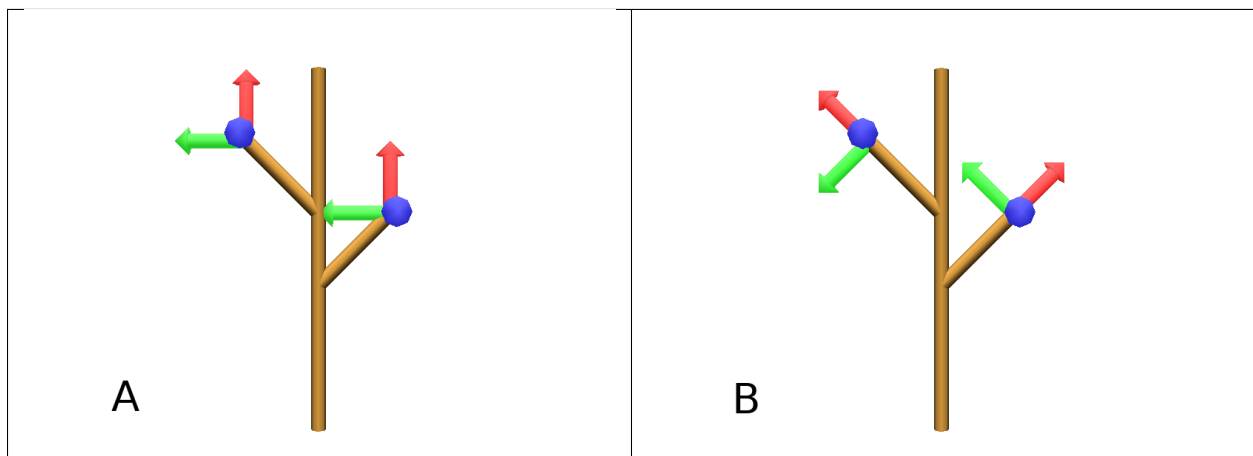


**LineTo** conserve the Turtle's orientation. To change orientation while drawing, **OLineTo** should be used.

```
Axiom: LineTo(0,0,2) [LineTo(0,1,3)Frame]LineTo(0,0,3) [LineTo(0,-1,4)Frame]LineTo(0,0,
↪5) # (Fig. A)
```

```
Axiom: LineTo(0,0,2) [OLineTo(0,1,3)Frame]LineTo(0,0,3) [OLineTo(0,-1,4)Frame]LineTo(0,
↪0,5) # (Fig. B)
```

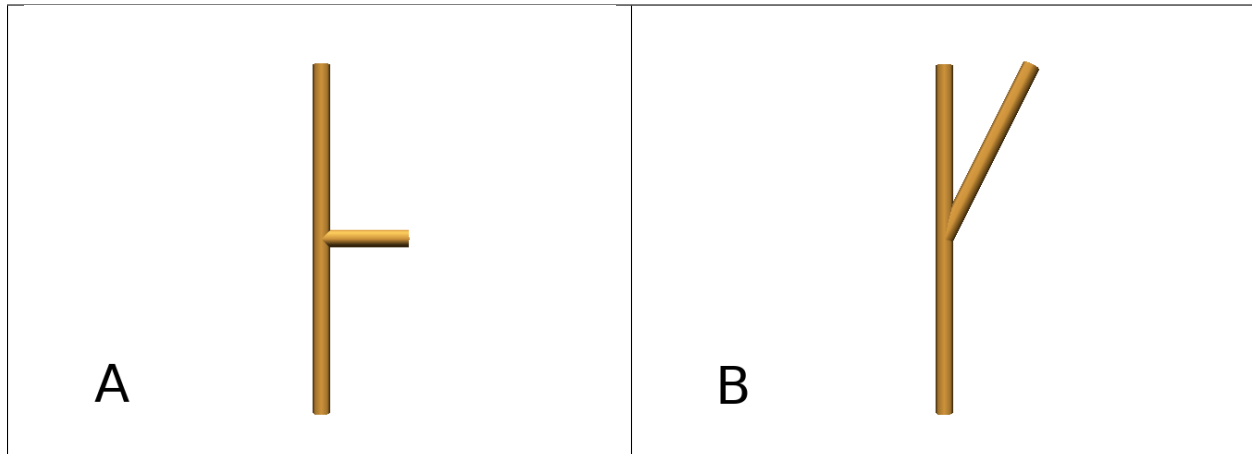
Download the example : `LineTo.lpy`



A relative drawing alternative also exists for **LineTo** and **OLineTo**. These primitives are **LineRel** and **OLineRel**

```
Axiom: LineTo(0,0,2) [LineTo(0,1,2)]LineTo(0,0,4) # (Fig. A)
```

```
Axiom: LineTo(0,0,2) [LineRel(0,1,2)]LineTo(0,0,4) # (Fig. B)
```



### SetGuide

Drawing a straight line made of length  $L=10$  with segments of size  $dl = 1.0$  (and thus contains  $n=10$  segments)

```
Axiom: nF(10, 1.) # (Fig. A)
```

By adding the primitive `SetGuide` before the line drawing, it is possible to specify a curve on which the Turtle is moving (instead of heading straight).

The `SetGuide` primitive must be given two mandatory arguments: a curve (`Polyline2D` or `NurbsCurve2D`) and a length: `SetGuide(C0, L0)`. This means that, following this statement, the Turtle will move on curve `C1` that has been rescaled from `C0` so that its new length is `L0` (whatever its original length).

The guiding curve can be defined in different ways. It can be defined for example by a python function (function `f` defined hereafter), e.g. (Fig. B) :

```
from openalea.plantgl.all import Polyline2D
from numpy import arange

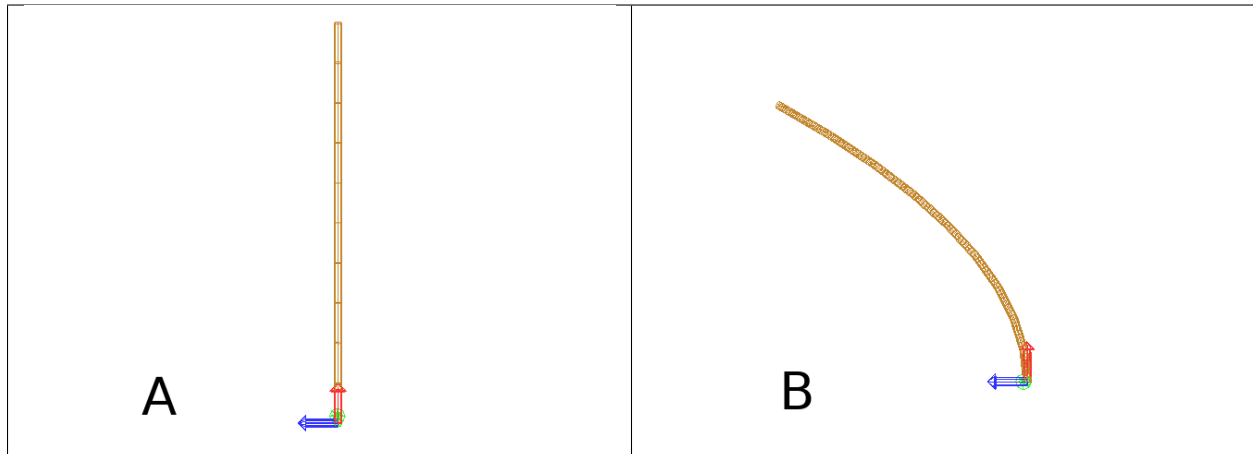
def f(u):
    return (u, u**2)

C0 = Polyline2D([f(u) for u in arange(0,1,0.1)]) # (Fig. B)
```

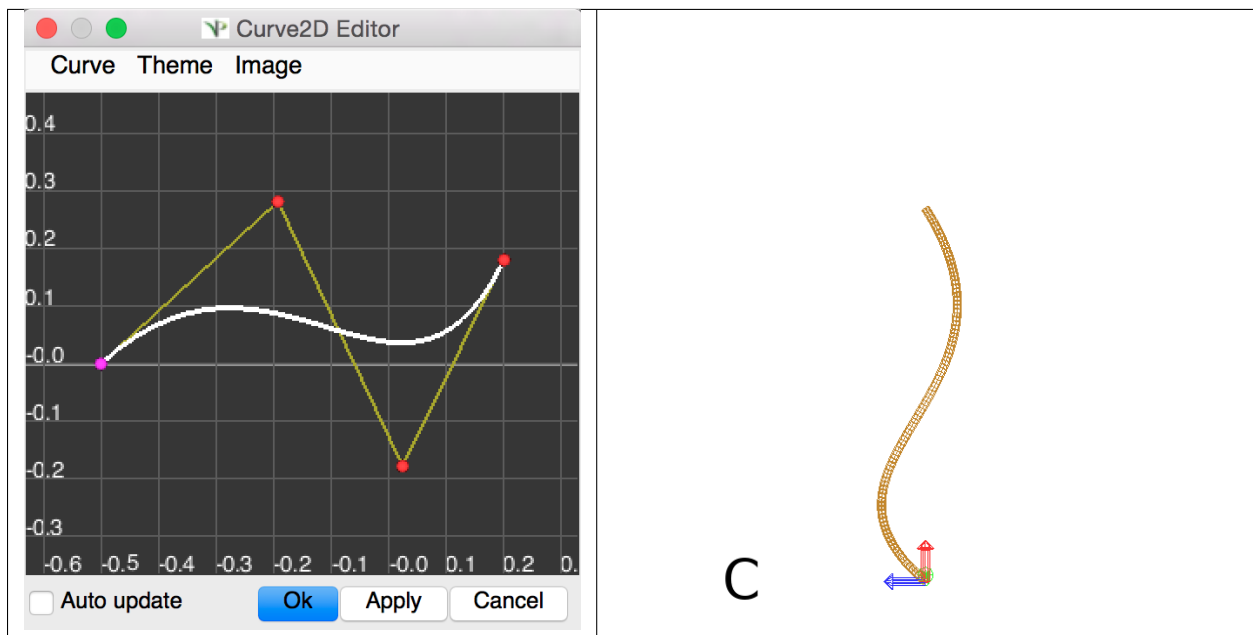
Then using curve `C0` in the `SetGuide` primitive, one can move the Turtle over a cumulated length `L`, thus using the defined curve `C1` (rescaled from `C0`) as a guide for moving up to a total length `L0`:

```
L = 10
L0 = 10
Axiom: SetGuide(C0, L0) nF(L, 0.1)
```

Download the example : `setGuide1.lpy` (With a `Polyline2D` imported from `PlantGL`)



or like the (Fig. C) example, the embedded L-Py graphical interface can be used to specifying 2D curves (the curve is then given the name **C0** for instance in the interface):

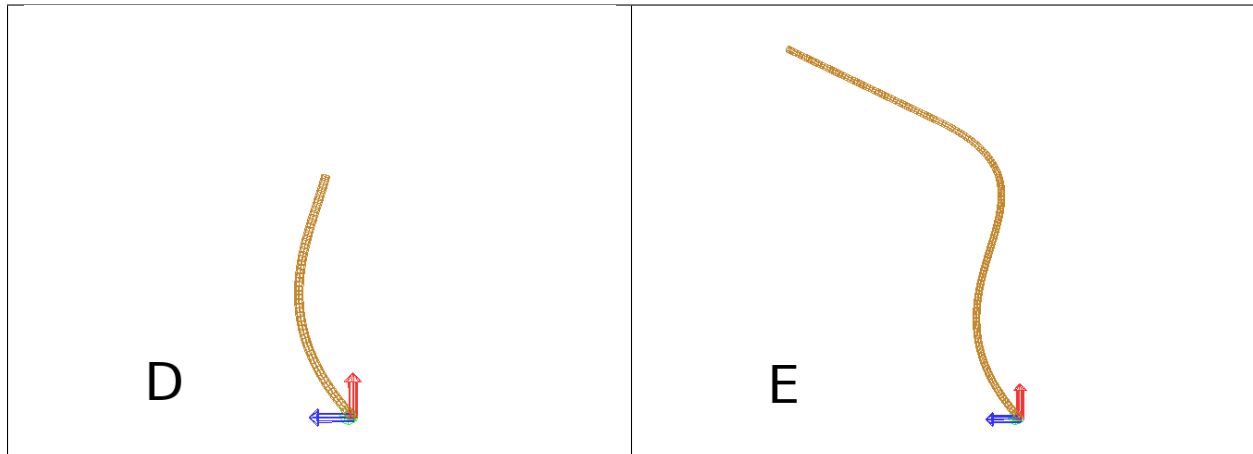


Download the example : `setGuide2.lpy` (With a Polyline2D created in the L-Py graphical interface)

Note that the Turtle can move less than the length of the 2D curve. In this case it will proceed forward over the **L** first units at the beginning of curve **C1** (Fig. D). By contrast, if  $L > L0$ , then the Turtle keeps on moving straight after reaching length **L0** (E).

```
L = 6
L0 = 10
Axiom: SetGuide(C0,L0) nF(L, 0.1) # (Fig. D)

L = 15
L0 = 10
Axiom: SetGuide(C0,L0) nF(L, 0.1) # (Fig. E)
```



To stop using the 2D curve as a guide, **EndGuide** can be used.

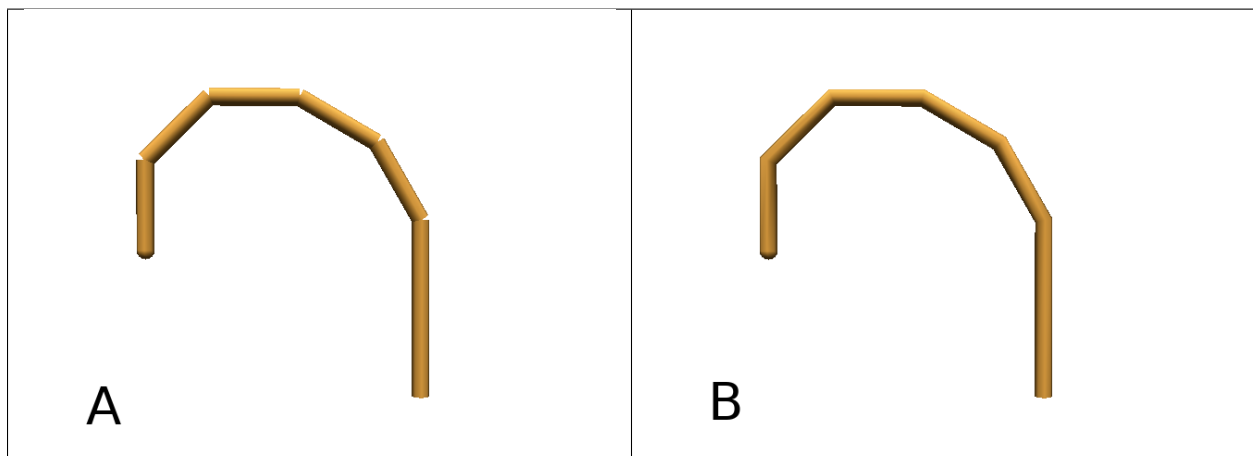
### Generalized cylinders

When several rotations are used while drawing, the rendering at rotation places isn't great. The separation points are really visible. To fix it, **@Gc** (or **StartGC**) can be used. Until a **@Ge** (or **"EndGC"**) all shapes drawn will be merged so that it becomes only one shape.

```
Axiom: F(2)+(30)F+(30)F+(30)F+(45)F+(45)F@O #Cylinders not generalized (Fig. A)
```

```
Axiom: @GcF(2)+(30)F+(30)F+(30)F+(45)F+(45)F@O@Gc #Cylinders generalized (Fig. B)
```

Download the example : `generalizedCylinders.lpy`



## 1.6 L-Py Turtle advanced primitives

### 1.6.1 Using PlantGL primitives

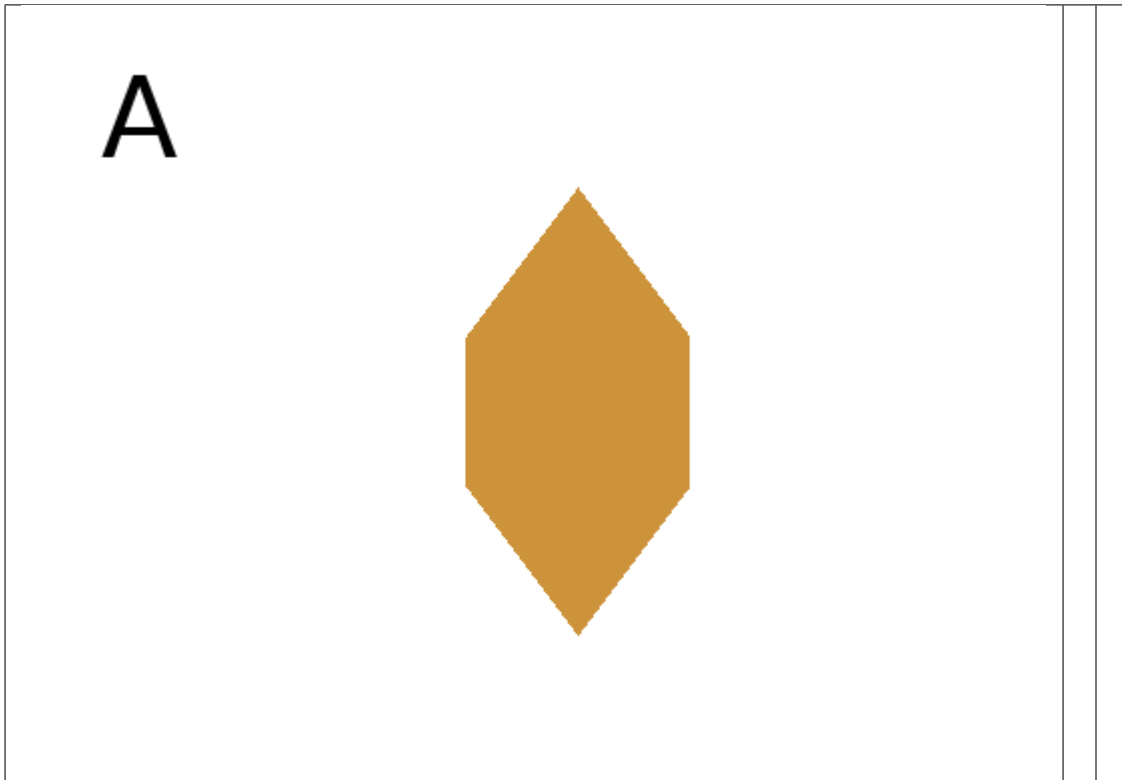
## Drawing PlantGL shapes

In order to draw more complex but predefined shapes, certain methods use PlantGL primitives. These methods are `~` and `@g`. `@g` draws the PlantGL shape in argument. `~` is more complicated, it takes in argument a geometric shape saved in the Turtle. A special primitive is already predefined : `~l`. It draws a leaf.

```
Axiom: ~l                                #(Fig. A)

Axiom: @g(Sphere(radius=3))              #(Fig. B)

execContext().turtle.setSurface('t', Sphere(radius=3))
Axiom: ~t(5)                             #(Fig. C)
```



## 1.6.2 Miscellaneous

In this section, several less common tools can be found.

### Elasticity and Tropism

One can add an elasticity property to a branch using `@Ts` or **Elasticity**. The value in argument should be between 0. and 1.

```
Axiom: FF[Elasticity(0.5)+F+F]F[-F]F      #(Fig. A)
```

A particular tropism can be setted using `@Tp` or **Tropism**. It takes a vector in argument.

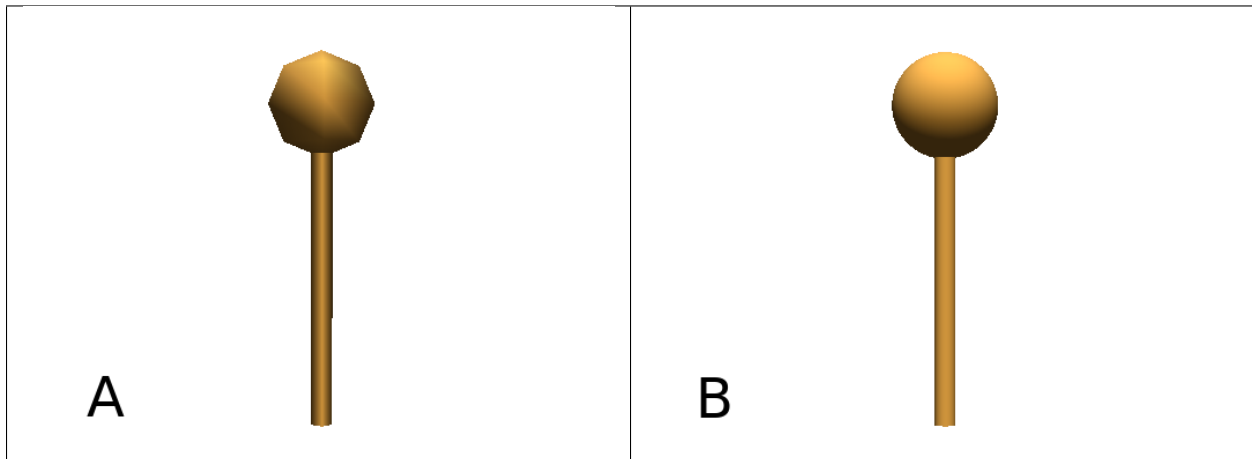
```
import numpy as np
v = np.array([0,1,2])
Axiom: FF[Tropism(v)+F+F]F[-F]F          #(Fig. B)
```

## SectionResolution

**SectionResolution** allows to change the resolution of all following shapes. **Be careful !** If the resolution is too low, the program may not work properly.

```
Axiom: F SectionResolution(4) +F@O(0.5)          #(Fig. A)
Axiom: F SectionResolution(60) +F@O(0.5)         #(Fig. B)
```

Download the example : `resolution.lpy`

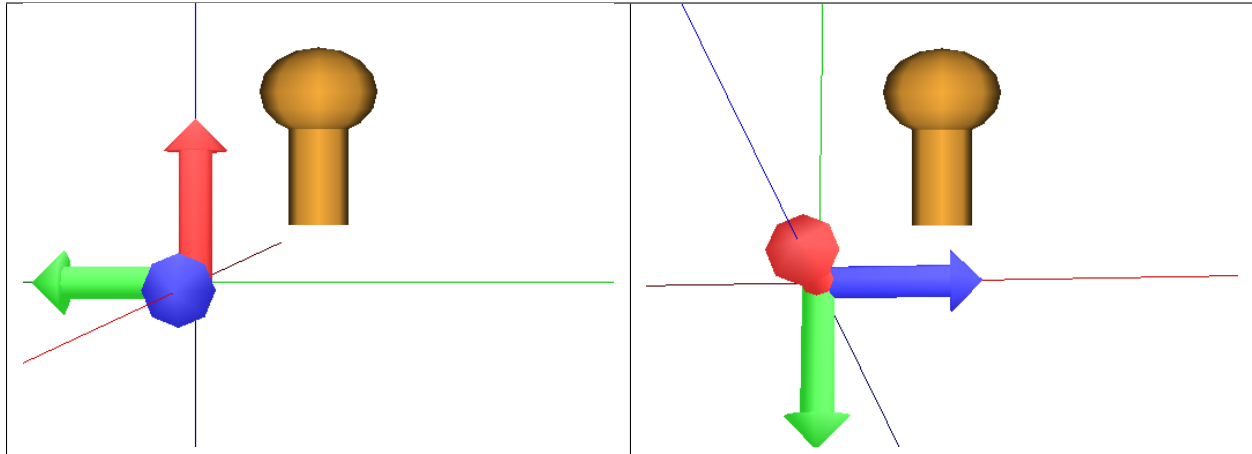


## ScreenProjection

After using **@2D** (or **StartScreenProjection**), the following shapes will be drawn on the screen coordinates system (in two dimensions). The examples below belong to the same axiom, only the camera's orientation is different. It confirms that the shape is in the screen system.

```
Axiom: Frame @2DF(0.6)@O(0.2)
```

Download the example : `screen.lpy`



To switch back to the original coordinates system, `@3D` (or `EndScreenProjection`) can be used.

## InterpolateColors

There is an other way to color shapes using **InterpolateColors**. This method mixes up two colors in one. There are three arguments, the first and the second are the index of materials and the last (optional) sets a priority to the first or the second color in order to make the final color. There are two examples below.

```
Step = 20
DIncr = 1.0 / Step

Axiom:
    d = 0.0
    for i in range(Step):
        nproduce InterpolateColors(3, 5, d) F(2.0/Step)    #(Fig. A)
        d += DIncr
    produce ; (2) @O(0.15)

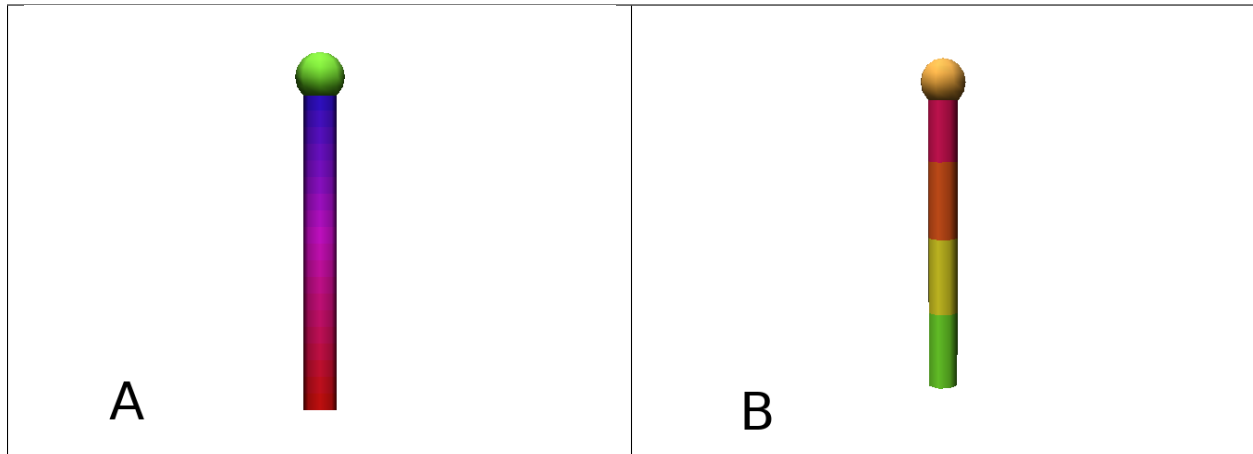
#Other example

Step = 4
DIncr = 1.0 / Step

Axiom:
    d = 0.0
    for i in range(Step):
        nproduce InterpolateColors(2, 5, d) F(2.0/Step)    #(Fig. B)
        d += DIncr
    produce ; (1) @O(0.15)
```

Download the example : `InterpolateColors.lpy`



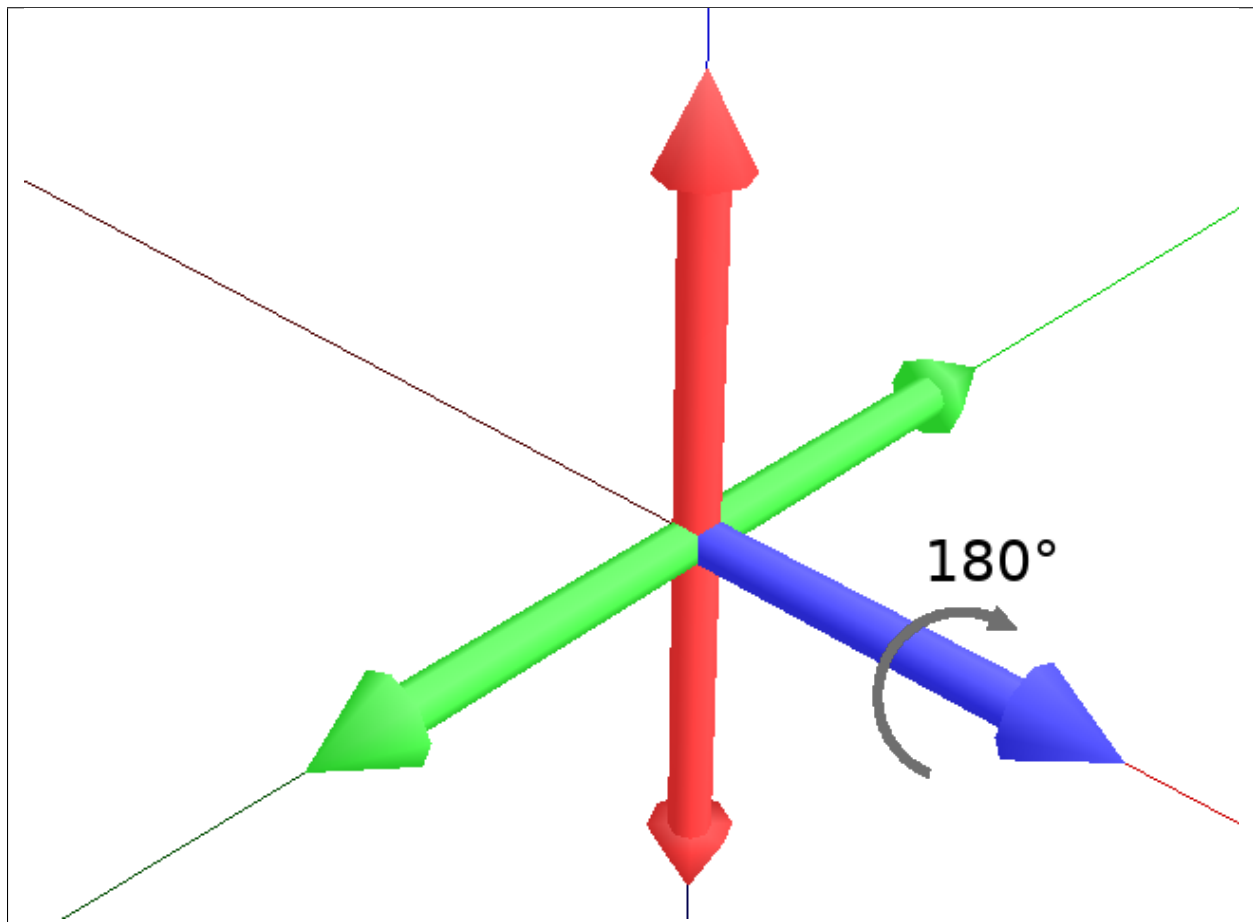


### Advanced rotation primitives

There are other primitives that can be used to rotate the Turtle.

**TurnAround** or **l**, turn the Turtle for  $180^\circ$  around the Up vector. It produce the same result as **+(180)** or **(-180)**

Axiom: `Frame(2) | Frame(2)`



## Requests

These methods allow to get some informations about the Turtle and store it in variables in order to use it after. Except **GetFrame**, it all can take three floats or one vector in arguments. If done, arguments are filled with values requested.

- **GetPos** or **?P**, collect the Turtle's Position vector informations.
- **GetHead** or **?H**, collect the Turtle's Head vector informations.
- **GetUp** or **?U**, collect the Turtle's Head vector informations.
- **GetLeft** or **?L**, collect the Turtle's Left vector informations.
- **GetRight** or **?R**, collect the Turtle's Right vector informations.

**GetFrame** or **?F**, collect the Turtle's Frame vector informations. It can take four vectors in arguments and fill it with the Position vector, the Head vector, the Up vector and the Left vector.

### 1.6.3 Rewriting shapes

To clear the viewer, the primitive **None** can be written in the Axiom part.

`Axiom: None`

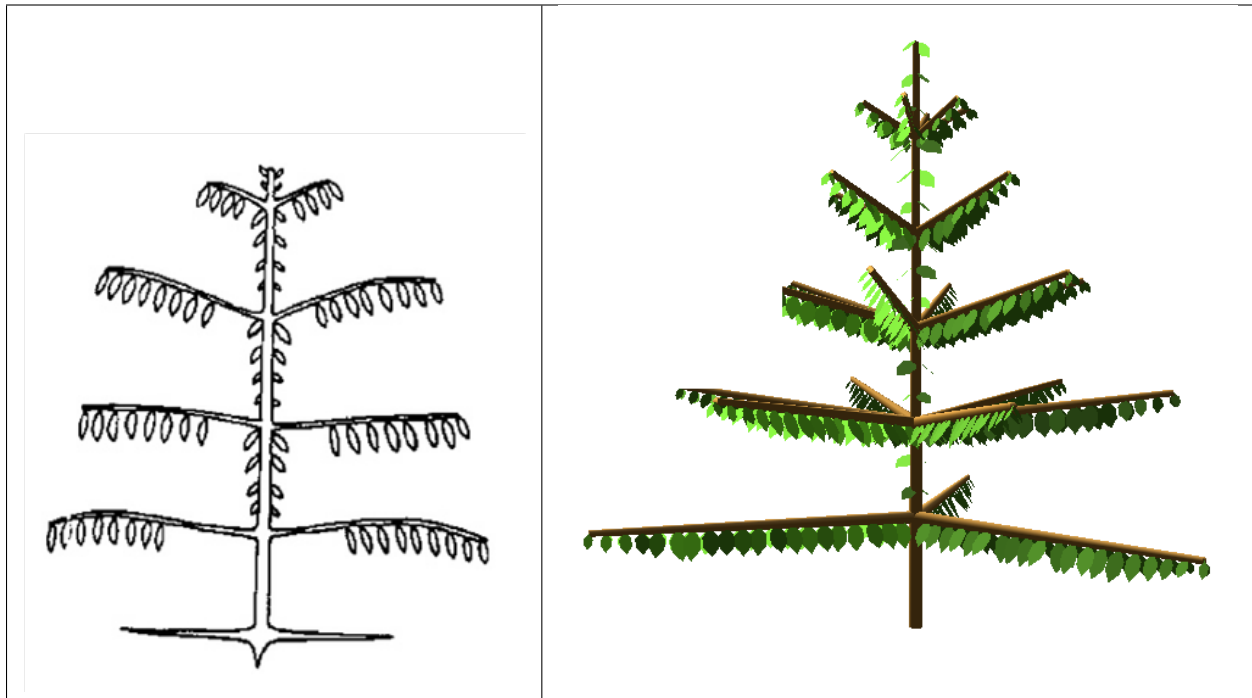
*Work in progress*

## 1.7 Tutorial

### 1.7.1 Architectural models

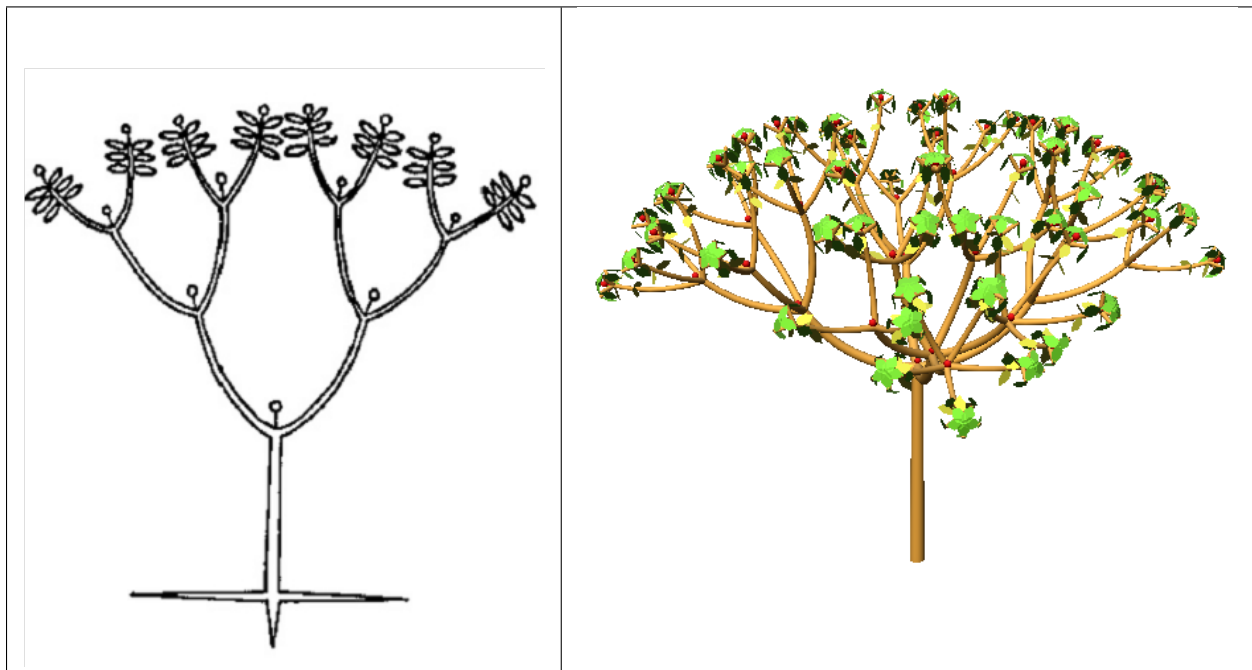
From architectural type defined by [Hallé, 71].

## Massart



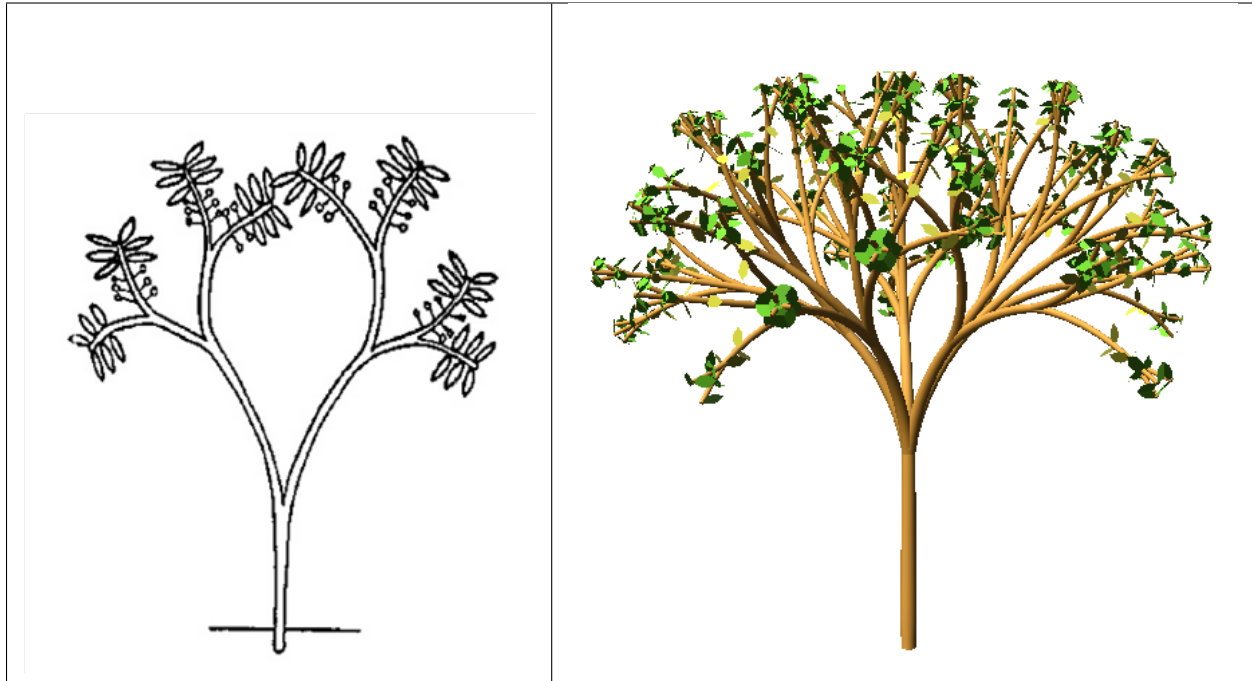
See the [massart.lpy](#)

## Leuwenberg



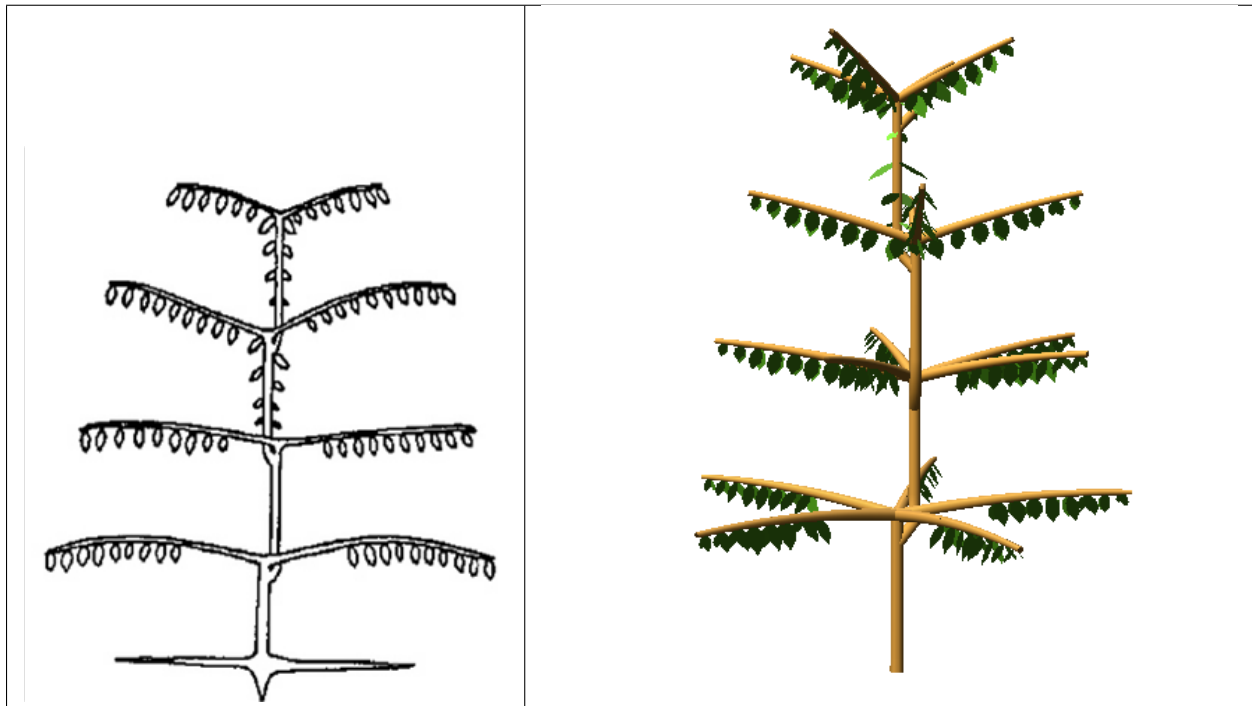
See the [leuwenberg.lpy](#)

## Schoute



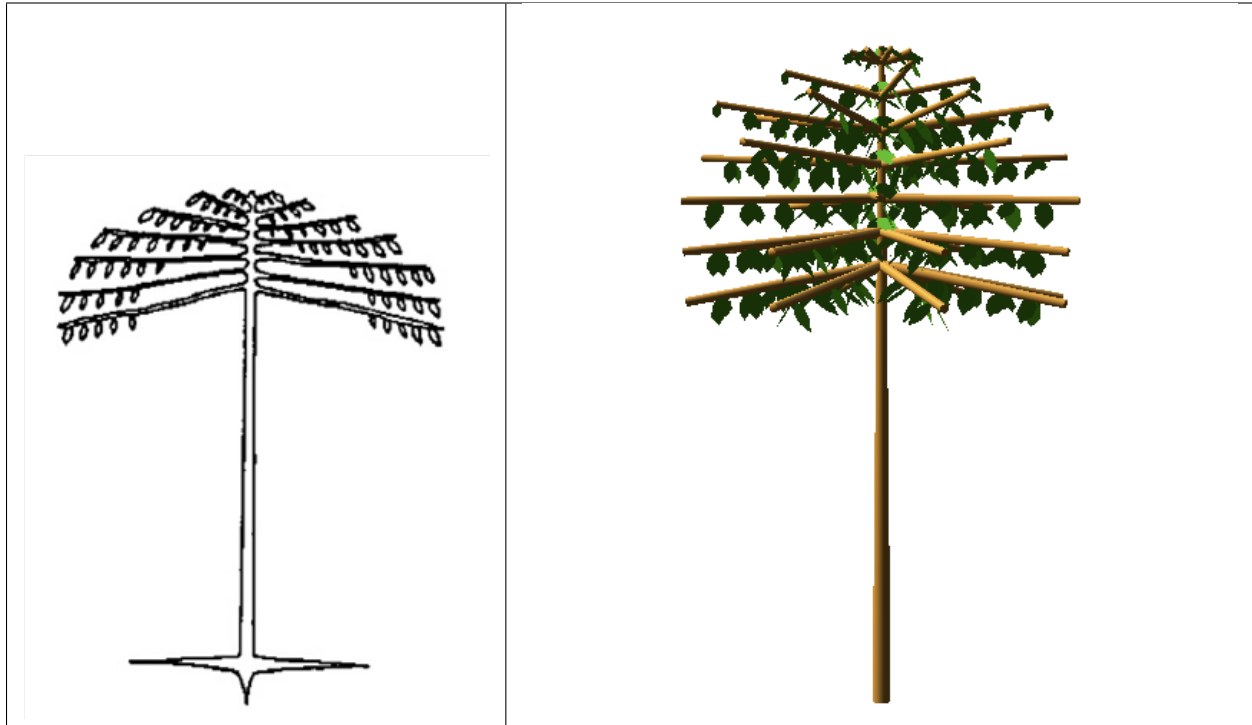
See the [schoute.lpy](#)

## Nozeran



See the [nozeran.lpy](#)

## Cook



See the [cook.lpy](#)

## 1.8 Subtleties with L-Py

### 1.8.1 Axiom parameters that change after simulation

Python makes it possible to create complex structure for instance to contains parameter of a module:

```
class ApexParameters:
    def __init__(self, age=0):
        self.age = age
    def __str__(self):
        return 'ApexParameters (age='+str(self.age)+')'
```

It can then be instantiated in the axiom and modified during the simulation:

```
module Apex
Axiom: Apex(ApexParameters(age=0))
dt = 1
derivation length: 2
Apex(p) :
    p.age += dt
    produce Node() Apex(p)
```

Creating the lsystem and running it the first time will produce the expected result.:

```
>>> l = Lsystem('mycode.lpy')
>>> lstring = l.derive()
>>> print lstring
Node() Node() Apex(ApexParameters(age=2))
```

However, a strange behavior occurs when asking for the axiom.:

```
>>> print l.axiom
Apex(ApexParameters(age=2))
```

In fact, this behavior is due to the way python manage object. By default, complex objects are not copied but simply shared between variables.:

```
>>> a = ApexParameters(age=0)
>>> b = a
>>> b.age = 2
>>> print a.age
2
```

The variables `b` and `a` point toward the same object in memory. This one will be destroyed only when no variable will be pointing to it. This is the same for the module of the string. In rule of lines 10-12, the parameter object of the `Apex` module of the string at derivation step `t` (line 10) is actually shared with the new `Apex` module of the produced string for step `t+1` (line 12). In the middle (line 11), it is modified and thus both string of step `t` and `t+1` will be affected. This produces side effects that lead to modification of the axiom in our case. With the `L-Py` graphic interface, this can also occur when running 2 times the same simulation. The second simulation can look strange since it was started with an axiom with modified parameters values.

To avoid that, an explicit copy of the parameter should be done. Rules will have the following shape.:

```
Apex(p) :
    from copy import deepcopy
    p = deepcopy(p)
    p.age += dt
    produce Node() Apex(p)
```

## 1.8.2 sproduce and undeclared modules

The `sproduce` function makes it possible to produce modules generated algorithmically.:

```
m = ParamModule('toto')
sproduce (Lstring([m]))
```

Advantage is that modules can be created on the fly. However, if you then try to math the produced module with rules such as the following one, it may not work.:

```
toto --> F
```

The reason is that by default `L-Py` assume module names of minimal length and thus the previous rule is understood as the 4 modules `t`, `o`, `t` and `o` should be transformed into one `F`. If it was used with production of the type:

```
produce toto
```

it would have work since consistently, the previous production would have been understood by `L-Py` as produce the 4 modules `t`, `o`, `t` and `o`. However if you use `ParamModule('toto')`, then you declare that only one module exists

and it has a composed name ‘toto’. toto being different from t, o, t and o, matching will not occur. To resolve this, you have to make L-Py understand that toto is actually a module name and not four letters. For this module declaration are available.:

```
module toto
```

Thus the rule:

```
toto --> F
```

will be correctly interpreted as one module toto being transformed into one F.

## 1.9 L-Py Helpcard

L-Py is based on the specification of Lstudio/cpfg-lpfg defined by P. Prusinkiewicz et al. (<http://algorithmicbotany.org/lstudio>).

### 1.9.1 Predefined Symbols

Here is a recap of the predefined symbol used in L-Py with their turtle interpretation:

	None	None Module.
--	------	--------------

#### Structure

[	SB	Push the state in the stack.
]	EB	Pop last state from turtle stack and make it the its current state.

## Rotation

Pinpoint		Orient turtle toward (x,y,z) . Params : 'x, y, z' or 'v' (optionals, default = 0).
PinpointRel		Orient turtle toward pos+(x,y,z) . Params : 'x, y, z' or 'v' (optionals, default = 0).
@R	SetHead	Set the turtle Heading and Up vector. Params: 'hx, hy, hz, ux, uy, uz' or 'h,v' (optionals, default=0,0,1, 1,0,0).
EulerAngles		Set the orientation of the turtle from the absolute euler angles. Params: 'azimuth, elevation, roll' (optionals, default=180,90,0).
.	Left	Turn left around Up vector. Params : 'angle' (optional, in degrees).
.	Right	Turn right around Up vector. Params : 'angle' (optional, in degrees).
^	Up	Pitch up around Left vector. Params : 'angle' (optional, in degrees).
&	Down	Pitch down around Left vector. Params : 'angle' (optional, in degrees).
/	RollL	Roll left around Heading vector. Params : 'angle' (optional, in degrees).
	RollR	Roll right around Heading vector. Params : 'angle' (optional, in degrees).
iRollL		Roll left intrinsically around Heading vector. Params : 'angle' (optional, in degrees).
iRollR		Roll right intrinsically around Heading vector. Params : 'angle' (optional, in degrees).
	TurnAround	Turn around 180deg the Up vector.
@v	RollToVert	Roll to Vertical : Roll the turtle around the H axis so that H and U lie in a common vertical plane with U closest to up
@h	RollToHorizontal	Roll to Horizontal : Roll the turtle so that H lie in the horizontal plane
LeftReflection		The turtle change the left vector to have a symmetric behavior.
UpReflection		The turtle change the up vector to have a symmetric behavior.
HeadingReflection		The turtle change the heading vector to have a symmetric behavior.



## Position

@M	MoveTo	Set the turtle position. Params : 'x, y, z' or 'v' (optionals, default = None for not changing specific coordinates).
MoveRel		Move relatively from current the turtle position. Params : 'x, y, z' or 'v'(optionals, default = 0).
@2D	StartScreenPro- jection	The turtle will create geometry in the screen coordinates system.
@3D	EndScreenPro- jection	The turtle will create geometry in the world system (default behaviour).

## Scale

@Dd	DivScale	Divides the current turtle scale by a scale factor, Params : 'scale_factor' (optional, default = 1.0).
@Di	MultScale	Multiplies the current turtle scale by a scale factor, Params : 'scale_factor' (optional, default = 1.0).
@D	SetScale	Set the current turtle scale, Params : 'scale' (optional, default = 1.0).

## Primitive

F		Move forward and draw. Params: 'length', 'topradius'.
f		Move forward and without draw. Params: 'length'.
nF		Produce a n steps path of a given length and varying radius. Params : 'length, dlength [, radius = 1, radiusvariation = None]'.
@Gc	StartGC	Start a new generalized cylinder.
@Ge	EndGC	Pop generalized cylinder from the stack and render it.
{	BP	Start a new polygon.
}	EP	Pop a polygon from the stack and render it. Params : concavetest (default=False).
.	PP	Add a point for polygon.
LineTo		Trace line to (x,y,z) without changing the orientation. Params : 'x, y, z, topdiameter' or 'v, topdiameter' (optionals, default = 0).
OLineTo		Trace line toward (x,y,z) and change the orientation. Params : 'x, y, z, topdiameter' or 'v, topdiameter' (optionals, default = 0).
LineRel		Trace line to pos+(x,y,z) without changing the orientation. Params : 'x, y, z, topdiameter' or 'v, topdiameter' (optionals, default = 0).
OLineRel		Trace line toward pos+(x,y,z) and change the orientation. Params : 'x, y, z, topdiameter' or 'v, topdiameter' (optionals, default = 0).
@O	Sphere	Draw a sphere. Params : 'radius' (optional, should be positive, default = line width).
@B	Box	Draw a box. Params : 'length', 'topradius'.
@b	Quad	Draw a quad. Params : 'length', 'topradius'.
@o	Circle	Draw a circle. Params : 'radius' (optional, should be positive, default = line width).
@L	Label	Draw a text label. Params : 'text', 'size'.
surface		Draw the predefined surface at the turtle's current location and orientation. Params : 'surface_name' (by default, 'l' exists), 'scale_factor' (optional, default= 1.0, should be positive).
~		Draw the predefined surface at the turtle's current location and orientation. Params : 'surface_name' (by default, 'l' exists), 'scale_factor' (optional, default= 1.0, should be positive).
@g	PglShape	Draw a geometry at the turtle's current location and orientation. Params : 'geometric_model', 'scale_factor' (optional, should be positive) or 'shape' or 'scene' or 'material'.
Frame		Draw the current turtle frame as 3 arrows (red=heading,blue=up,green=left). Params : 'size' (should be positive), 'cap_height_ratio' (in [0,1]), 'cap_radius_ratio' (should be positive).
Arrow		Draw an arrow. Params : 'size' (should be positive), 'cap_height_ratio' (in [0,1]), 'cap_radius_ratio' (should be positive).
SetContour		Set Cross Section of Generalized Cylinder. Params : 'Curve2D [, ccw]'.
Section-Resolution		Set Resolution of Section of Cylinder. Params : 'resolution' (int).
SetGuide		Set Guide for turtle tracing. Params : 'Curve[2D 3D], length [, yorientation, ccw]'.
EndGuide		End Guide for turtle tracing.
Sweep		Produce a sweep surface. Params : 'path, section, length, dlength [, radius = 1, radiusvariation = None]'.
PositionOnGuide		Set position on Guide for turtle tracing.

## Width

_	IncWidth	Increase the current line width or set it if a parameter is given. Params : 'width' (optional).
!	DecWidth	Decrease the current line width or set it if a parameter is given. Params : 'width' (optional).
SetWidth		Set current line width. Params : 'width'.

## Color

;	Inc-Color	Increase the current material index or set it if a parameter is given. Params : 'index' (optional, positive int).
,	Dec-Color	Decrease the current material index or set it if a parameter is given. Params : 'index' (optional, positive int).
SetColor		Set the current material. Params : 'index' (positive int) or 'r,g,b[,a]' or 'material'.
Interpolate-Colors		Set the current material. Params : 'index1', 'index2', 'alpha'.

## Tropism

@Ts	Elasticity	Set Branch Elasticity. Params : 'elasticity' (optional, default= 0.0, should be between [0,1]).
@Tp	Tropism	Set Tropism. Params : 'tropism' (optional, Vector3, default= (1,0,0)).

## Request

?P	GetPos	Request position vector information. Params : 'x,y,z' or 'v' (optional, default=Vector3, filled by Turtle).
?H	GetHead	Request heading vector information. Params : 'x,y,z' or 'v' (optional, default=Vector3, filled by Turtle).
?U	GetUp	Request up vector information. Params : 'x,y,z' or 'v' (optional, default=Vector3, filled by Turtle).
?L	GetLeft	Request left vector information. Params : 'x,y,z' or 'v' (optional, default=Vector3, filled by Turtle).
?R	GetRight	Request right vector information. Params : 'x,y,z' or 'v' (optional, default=Vector3, filled by Turtle).
?F	Get-Frame	Request turtle frame information. Params : 'p,h,u,l' (optional, filled by Turtle).

## Texture

TextureBase-Color		Set the base color of the texture. Params : 'index' (positive int) or 'r,g,b[,a]' or 'material'.
Interpolate-TextureBase-Colors		Set the base color of the texture from interpolation of 2 predefined material. Params : 'index1', 'index2', 'alpha' .
TextureScale		Set the scale coefficient for texture application. Params : 'uscale, vscale' (default = 1,1) or 'scale'.
TextureUScale		Set the u-scale coefficient for texture application. Params : 'uscale' (default = 1).
TextureVScale	TextureVCo-eff	Set the v-scale coefficient for texture application. Params : 'vscale' (default = 1).
Texture-Translation		Set the translation for texture application. Params : 'utranslation, vtranslation' (default = 0,0) or 'translation'.
TextureRotation		Set the rotation for texture application. Params : 'angle, urotcenter, vrotcenter' (default = 0,0.5,0.5) or 'angle, rotcenter'.
Texture-Transformation		Set the transformation for texture application. Params : 'uscale, vscale, utranslation, vtranslation, angle, urotcenter, vrotcenter' (default = 1,1,0,0,0.5,0.5) or 'scale, translation, angle, rotcenter'.

## String Manipulation

X	MouseIns	Module inserted just before module selected by user in visualisation.
%	Cut	Cut the remainder of the current branch in the string.
new	newmodule	Create a new module whose name is given by first argument.

## Pattern Matching

=]		Match exactly a closing bracket
•	any	Used to match any module in rules predecessor. First argument will become name of the module.
x	reexp, all	Used to specify matching of a repetition of modules.
or		Used to specify an alternative matching of modules.
?I	GetIterator	Request an iterator over the current Lstring.
\$	GetModule	Request a module of the current Lstring.

### 1.9.2 Predefined commands

Here comes the python commands that control the simulation.

The following commands can be redefined to initialize simulation state:

def Start(lstring)	is called at the beginning of the simulation. One argument can be optionally defined to receive the input lstring. A modified lstring can be returned by the function to modify the axiom of the simulation.
def End(lstring, geometries)	is called at the end of the simulation. One or two arguments can be optionally defined to receive the input lstring and its geometric interpretation. A modified lstring or scene can be returned by the function to change output of the simulation.
def StartEach(lstring)	is called before each derivation step. One argument can be optionally defined to receive the input lstring. A modified lstring can be returned by the function to modify input lstring of the current iteration.
def EndEach(lstring, geometries)	is called after each derivation step. One or two arguments can be optionally defined to receive the input lstring and its geometric interpretation. Returning an lstring or (lstring, geometries) will be used for next iterations and display. If frameDisplayed() is False, geometries is None.
def Start-Interpretation()	is called at the beginning of the interpretation. Interpretable modules can be produced to generate extra graphical elements.
def End-Interpretation()	is called at the end of the interpretation. Interpretable modules can be produced to generate extra graphical elements.
def Post-Draw()	is called after drawing the representation of a new lstring.

Python commands that control the rule application:

Stop()	Stop simulation at the end of this iteration.
forward()	Next iteration will be done in forward direction.
backward()	Next iteration will be done in backward direction.
isForward()	Test whether direction is forward.
getIterationNb()	Return the id of the current iteration.
useGroup(int)	Next iteration will use rules of given group and default group 0.
getGroup()	Gives which group will be used.
frameDisplay(bool)	Set whether a frame will be displayed at the end of the iteration. default is True in animation and False except for last iteration in run mode.
isFrameDisplayed()	Tell whether a frame will be displayed at the end of the iteration.
isAnimationEnabled()	Return the current simulation is in an animation.
requestSelection(caption)	Wait selection in the viewer before next iteration. Set frameDisplay to True.

### Lpy specific declaration:

module <i>name</i>	Declaration of module name.
consider: <i>name</i>	Symbol to consider.
ignore: <i>name</i>	Symbol to ignore.
group <i>id</i> :	Following rules will be associated to group <i>id</i> .
Axiom: <i>Lstring</i>	Declaration of the axiom of the Lsystem.
produce <i>Lstring</i>	Produce an <i>Lstring</i> and return.
nproduce <i>Lstring</i>	Produce an <i>Lstring</i> whithout returning.
nsproduce( <i>LstringStruct</i> )	Produce a given <i>Lstring</i> data structure whithout returning.
makestring( <i>Lstring</i> )	Create an <i>LstringStruct</i> from <i>Lstring</i> .
InLeftContext(pattern, argdict)	Test a left context. argdict contains value of all parameter of the pattern.
InRightContext(pattern, argdict)	Test a right context. argdict contains value of all parameter of the pattern.
derivation length: <i>value</i>	Number of derivation to do (default=1).
initial_view= <i>*value*</i>	Number of derivation for bounding box evaluation (default=derivation length).
production:	Start of the production rules declaration.
homomorphism:	Start of the interpretation rules declaration.
interpretation:	Start of the interpretation rules declaration.
decomposition:	Start of the decomposition rules declaration.
maximum depth:	Number of decomposition or interpretation recursive call to do (default=1).
endgroup	Reactivate default group 0.
endsystem	End of Lsystem rules declaration.

### These commands have been added to the original cpfg-lpfg specification:

context()	Get context of execution of the L-system. To use with care.
-----------	---

### The following objects and commands are also accessible from within the lpy shell:

lstring	Contains the last computed lsystem string of the current simulation.
lsystem	Reference to the internal lsystem object representing the current simulation.
window	Reference to lpy widget object.
clear()	To clear the shell.

All these functions are imported from openalea.lpy module. Other data structures and fonctionnalities are available in the module. You can check them with **help(openalea.lpy)**

## 1.9.3 References

For More details, see:

- F. Boudon, T. Cokelaer, C. Pradal and C. Godin, L-Py, an open L-systems framework in Python, FSPM 2010.
- P. Prusinkiewicz et al., 89, The algorithmic Beauty of Plants, Springer-Verlag.
- P. Prusinkiewicz. Graphical applications of L-systems. Proceedings of Graphics Interface '86, pp. 247-253.
- P. Prusinkiewicz, R. Karwowski, and B. Lane. The L+C plant modelling language. In Functional-Structural Plant Modelling in Crop Production, J. Vos et al. (eds.), Springer, 2007.

## 1.10 L-Py in scripts or in third party applications

### 1.10.1 Manipulation of Lsystems

Using the `openlea.lpy` module, Lsystems can be manipulated directly using the ‘Lsystem’ object. Main actions are the creation and the derivation of an Lsystem.

#### Creation

To create an Lsystem, a file or a string containing the code can be used.

```
from openlea.lpy import *

l = Lsystem("myfile.lpy")
```

or

```
l = Lsystem()
l.setCode(mycode)
```

To configure the creation of an Lsystems by setting predefined variables, it is possible to pass as an argument a dictionary of variables.

```
variables = {'VAR1' : VALUE1 , ... }
l = Lsystem("myfile.lpy", variables)
```

In such case, the variables contained in the dictionary may overwrite graphically defined variables if they have similar names or variable in the model defined using the `extern` command.

The `extern` command can be used to set variables that are defined with default value in the Lsystem but can be redefined externally. For instance, in the following example, the “VAR1” is redefined externally.

```
# myfile.lpy
extern(VAR1 = VALUE1)
Axiom: F(VAR1)
```

```
# application of the lsystem
l = Lsystem("myfile.lpy", {'VAR1' : VALUE2})
```

#### Derivation

The standard way to apply derivation on an Lstring using a Lsystem is to use the function `derive`. Its parameters are the lstring on which derivation is applied, the iteration number to which the derivation corresponds and the number of iterations to apply. Default values for lstring, iteration number and number of iterations are the axiom, 0 and derivation length respectively. An extra optimization parameter makes it possible to indicates if the lstring has already been used for interpretation (and thus environmental modules are completed).

Thus an Lsystem can be simulated with the following code

```
l = Lsystem("myfile.lpy")
lstring = l.derive()
```

To have all intermediate lstrings, the following code can be used

```
l = Lsystem("myfile.lpy")
lstring = l.axiom
for i in xrange(l.derivationLength):
    lstring = l.derive(lstring, i, 1)
```

A more compact version using iterator is

```
lssystem = Lsystem("myfile.lpy")
for lstring in lssystem:
    pass
```

Note that the function `interpret` of the `Lsystem` makes it possible to apply interpretation rules on an `Lstring` and return the resulting interpretation string.

```
lssystem = Lsystem("myfile.lpy")
for lstring in lssystem:
    ilstring = lssystem.interpret(lstring)
```

## Graphical output

A visual interpretation can be made using a 3D turtle. For this the function `turtle_interpretation` of the `Lsystem` can be used. Custom turtle deriving from `PlantGL Turtle` class can be defined and used. By default, a `PglTurtle` is used which output `PlantGL` primitives. A computation of the `PlantGL` representation at each step can thus be defined in the following way:

```
from openalea.lpy import *
from openalea.plantgl.all import *

lssystem = Lsystem("myfile.lpy")
for lstring in lssystem:
    t = PglTurtle()
    lssystem.turtle_interpretation(lstring, t)
    scene = t.getScene()
```

Note that an interpretation of the `lstring` as a `PlantGL` scene can be computed directly with the `sceneInterpretation` of the `Lsystem` object.

```
from openalea.lpy import *

lssystem = Lsystem("myfile.lpy")
for lstring in lssystem:
    scene = lssystem.sceneInterpretation(lstring)
```

Plotting directly the 3D scene from the `Lstring` is also possible with the function `plot` of the `Lsystem` object.

```
from openalea.lpy import *

lssystem = Lsystem("myfile.lpy")
for lstring in lssystem:
    lssystem.plot(lstring)
```

In such case, the viewer used to plot the 3D scene can be parameterized (By default it is the `PlantGLViewer`). For this the function `registerPlotter` of the `lpy` module can be used. It allows to register a plotter whose following interface are expected



```

class Plotter:
    def __init__(self):
        pass
    def plot(self, scene):
        pass
    def save(self, fname, format):
        """ Save the view of the 3D scene in fname with the given format (PNG, JPG) """
        ↪
        pass
    def selection(self):
        """Should return a list of id of selected elements """
        pass
    def waitSelection(self, txt):
        """ Wait for selection of elements with the following text """
        pass

```

From the Lsystem point of view, it is possible to use the function `animate` or `record` to plot at each step the 3D interpretation of the Lsystem using the defined `plotter`.

### 1.10.2 Graphical Parameters Manipulation

The L-Py GUI makes it possible to define graphical parameters by the user and usable within the simulation. For this, some code are defined at the end of the Lsystem code. In addition to the variables defined in its namespace, it is possible to have access to the predefined graphical parameters of an Lsystems using some globals variables:

```

lsystem = Lsystem("myfile.lpy")
# A list of scalar object that defined type, value and bounds
# of every graphical scalar parameters
print lsystem.__scalars__

# A list of information on graphical objects.
# It is a list of panel.
# A panel is a tuple with panelinfo and list of objets.
# Objects are defined as a tuple with a type and the object.
# Panelinfo is a dictionary of properties.
print lsystem.__parameterset__

```

To write parameters at the end of an Lsystem code, it is possible to use function defined in `openalea.lpy.simu_environ`

```

from openalea.lpy.simu_environ import initialisationFunction
from openalea.lpy.gui.scalar import IntegerScalar

lc = LsysContext()
# Setting the options of Lsystem execution
lc.options.setSelection('Module declaration',1)

# Defining graphical scalar parameter
scalars = [IntegerScalar('default_scalar', 1, 0, 100)]

initcode = initialisationFunction(lc, scalars = scalars,
                                visualparameters = None,
                                colorlist = None,
                                referencedir = savedir)

```

Modules and objects included in Lpy are also described in `lpy_reference`.

**Warning:** This Guide is still very much in progress. Many aspects of Lpy are not covered.

## CHAPTER 2

---

### References

---

- F. Boudon, C. Pradal, T. Cokelaer, P. Prusinkiewicz, C. Godin. L-Py: an L-system simulation framework for modeling plant architecture development based on a dynamic language. *Frontiers in Plant Science*, Frontiers, 2012, 3 (76), doi: [10.3389/fpls.2012.00076](https://doi.org/10.3389/fpls.2012.00076).

For more details on Lsystems, see also:

- F. Boudon, T. Cokelaer, C. Pradal and C. Godin, L-Py, an open L-systems framework in Python, FSPM 2010

L-Py was inspired by [Lstudio/cpfg-lpfg](#) defined by P. Prusinkiewicz et al. See also

- P. Prusinkiewicz et al., 89, *The algorithmic Beauty of Plants*, Springer-Verlag.
- P. Prusinkiewicz. Graphical applications of L-systems. *Proceedings of Graphics Interface '86*, pp. 247-253.
- P. Prusinkiewicz, R. Karwowski, and B. Lane. The L+C plant modelling language. In *Functional-Structural Plant Modelling in Crop Production*, J. Vos et al. (eds.), Springer, 2007.