L-Py, an open L-systems framework in Python

Frédéric Boudon[1], Thomas Cokelaer[2], Christophe Pradal[1] and Christophe Godin[2]
[1]CIRAD/[2]INRIA, Virtual Plants INRIA Team, UMR DAP, TA A-96/02, 34398 Montpellier Cedex 5, France,
frederic.boudon@cirad.fr

Keywords: L-systems, Python, Plant Simulation Software, Development Environment

Introduction

L-systems were conceived as a mathematical framework for modeling growth of plants. In this paper, we present L-Py, 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.

The L-Py development environment

The central idea of L-systems consists of the rewriting of a string of modules representing the structure of the plant. The rewriting rules (productions) express the creation and changes of state of the plant modules throughout time. Such rules can be expressed using a dedicated programming language (Prusinkiewicz et al., 99) or by incorporating L-system-based language constructs into existing languages, such as C++ (Karwowski and Prusinkiewicz 03) or Java (Kniemeyer and Kurth, 08). In this work, we explore the use of the Python language as support for L-systems. Indeed, Python offers a powerful modeling language with high level structures and constructs. It is interpreted and makes it possible to explore easily and interactively the various complex structures involved in a program.

Integrating L-systems in Python – We followed similar methodology than *lpfg* (Karwowski and Prusinkiewicz, 03). L-systems constructs are integrated into regular python code. In a first part of the L-Py code of a model, modules can optionally be declared possibly with parameter names and global properties using the following syntax.

```
module Apex(age) : scale = 1
```

Declaring module has the advantage of making string representation more compact (i.e. `Apex` for instance is now understood as one module instead of the four modules `A`, `p`, `e` and `x`). Moreover, it makes it possible to annotate modules for future use in external modules (for instance naming parameters makes their role explicit). Note that module parameters are simply python objects which can be of any type. They thus do not require any type declaration. In the code of an lpy model, the `axiom` is then declared. The keywords `production` initiates block for rules declaration. Rule predecessor includes specification of possible contexts. Rules can be expressed using two conventions. Simple rules can be written in a compact mathematical style similar to *cpfg* (Prusinkiewicz et al., 99). For more complex rules, successor specifications are declared using the `produce` statement, as in *lpfg*, embedded into regular python code with indentation-based syntax. A typical L-Py code will look like

```
module Apex(age), Internode(length,radius)
MAX_AGE, dr = 10, 0.01 # constants
Axiom: Apex(0)
production:
Apex(age) :
  if age < MAX_AGE:
     produce Internode(1+random(),0.1)/(180)[+(20)Apex(age+1)] Apex(age+1)
Internode(l,r) --> Internode(l,r+dr)
```

When an L-system is built, its code, which contains both python code and special L-system constructs, is first transformed in pure python code by the L-Py language parser. At execution, python functions corresponding to rule code are called each time a rule is applied.

Compatibility with both *cpfg* and *lpfg* has been kept making it easy to switch from one to another according to user concerns (language preference, external modules compatibility, required performance, etc.)

Geometry - L-Py is built upon the graphical library PlantGL (Pradal, Boudon et al. 09). A PlantGL scene graph is computed for each required geometrical representation. Similarly to *cpfg/lpfg*, a representation is computed with a *turtle interpretation* of the string with predefined modules encoding turtle actions. Predefined module names are kept compatible with both *cpfg* and *lpfg*.

Additionally, two new modules have been integrated. The first one, namely `@g(geom)`, appends any primitive of PlantGL at current turtle location and orientation. The second new module concerns global geometry of branches. Expressing the detailed irregular geometry of a branch may be difficult with simple turtle commands. A first option is to express branch geometry according to a given tropism. Using such an option, a model simply contains commands for straight elements that are bended according to a global tropism direction. This first approach is difficult to use for complex branch curvature. Alternatively, it is possible to express branch geometry according to a given 3D curve and looking at local changes of direction along the curve using differential geometry (Prusinkiewicz et al., 01). This method makes it possible to express in a flexible way complex geometries but requires expressing explicitly all the turtle changes of direction along a branch in the L-systems, thus polluting the overall L-system code with technical details. To ease modeler work, we introduced a hybrid approach where the user can predefine the path of the turtle with a graphically defined curve and the command `SetGuide(path,length)`. Successive turtle forward commands will follow curvature of the given path until it reaches a total length of `length`.

An open system – Thanks to Python, L-Py can reuse tools from the entire python scientific community (*Scipy*, *Rpy*), in particular the ones from the OpenAlea platform (Pradal et al., 08). L-Py can thus mix structures defined in different applicative contexts possibly with different languages (C, C++, Fortran, Java, *etc.*). In particular, complex data flows can be built graphically in OpenAlea and reused as python functions in L-Py.

L-Py defines a number of object oriented structures (module, string, rules, etc) accessible and modifiable from Python. After each iteration, modelers have access to the resulting string structure and corresponding scene graph through the `EndEach` function, thus allowing global post-processing using regular python code. For instance, direct illumination of the plant can easily be computed using *Fractalysis* (Da Silva et al., 08) or *Caribu* (Chelle and Andrieu, 98) modules of OpenAlea.

Conversely, the L-Py kernel is built as a python module that can also be integrated into any python compatible application. For instance, it is available in the OpenAlea platform as various processing nodes (creation of an L-system, iteration, run, display, *etc*. see Fig. 1) and the resulting structures can be post-processed independently by other modules. The different global variables that are used within a model are accessible externally and easily modifiable using an *introspection* mechanism. Thus, execution of a model can be simply parameterized by setting externally the values of the different variables controlling the model without changing anything in its code (see variable `dr` in first example and Fig. 1).

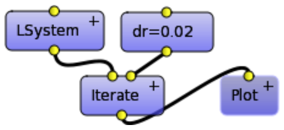

Fig 1: L-Py nodes in *Visualea*

To be able to apply a number of tools, the central L-Py string structure has been made compatible with the *MTG* structure. MTG is a central integrative structure of the OpenAlea platform which represents the plant architecture at different scales. Numerous modules are compatible or make use of it. Translation mechanisms in both directions have been defined. Annotation of modules is particularly useful since it makes it possible to define parameter names and scales of the different module types. Such compatibility is of particular interest to mix simulation and analysis processes without interference between them. As an illustration, a post-processing is made on the structure generated in first example. The goal of this process is to compute an *aggregate variable*, in this case volume of branches, from the structure defined at internode scale. This is difficult to achieve using L-systems which have a local view on the structure but is nevertheless important for FSPM. For this, the L-string structure with its associated geometric representation is converted into an MTG (line 1). A *quotient* operation that computes groups of internodes and adds corresponding macroscopic nodes in the MTG, is performed to identify branches (line 2). Volumes of each branch are estimated as sum of component internode volumes (lines 4-6) and plotted as a histogram using the *Matplotlib* module (line 7 and Fig 2.).

The advantage is that it can be performed in an exploratory manner without re-launching simulation. Note that similar analysis can be done in GroImp by carrying query on the support graph (Kniemeyer and Kurth, 2008).

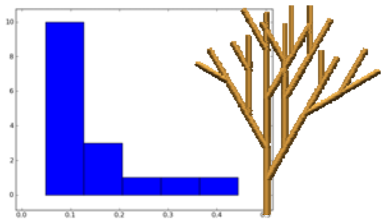
Fig 2: Structure and histogram of branch volumes

```
def EndEach(lstring,geometries):
  g = lstring2mtg(lstring,geometries)
  g,axis_scale = create_axis_scale(g,node_scale=1)
   def ivolume(l,r) : return l * pi * (r**2) # internode volume computation
  def avolume(axe) : return sum([ivolume(i.length,i.radius) for i in g.components(axe)])
  axis_volumes = [avolume(axe) for axe in g.vertices(scale=axis_scale)]
  pylab.hist(axis_volumes)
```

IDE - In general, L-system models are complemented with a number of facilities to express visual high-level attributes such as functions, materials, *etc*. Upon the L-Py kernel, an *integrated development environment* has been built that includes code and visual parameter editors. It contains a code editor with dedicated syntax highlighting and visual editors for 2D functions, scalars, curves, patches, materials, etc. This environment is inspired and bridges the gap between integrated environment such as *L-studio* (Prusinkiewicz, 04) and modular one such as *VLab* (Federl and Prusinkiewicz, 99). Indeed, modern interface concept have been used such as *dockable widget* which make it possible to display utility panels floating (see Fig 3) or docked on one of the borders of the main window. Users can thus highly customize their working environment. A python *shell* has been also integrated and makes it possible to inspect and manipulate procedurally the L-systems and their results. Similarly to *L-studio*, L-Py also contains a *continuous modeling* mode in which interactions with parameters are immediately propagated and model automatically updated. This allows simple interaction with the model.

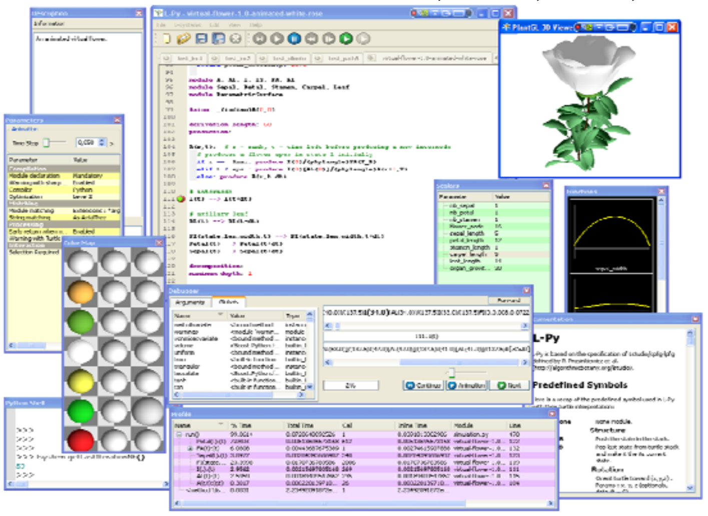
Fig. 3: The L-Py IDE with its utility windows for parameter edition.

Optimization – As for programming with classical language, performance and robustness are key points of a model. This makes it possible to have convenient interaction and explore parameter space in reasonable time. The main drawback of the Python language is its performances that are usually slower than compiled languages. To keep acceptable performance, L-Py integrates two tools in its IDE that help optimizing models. First, a *debugger* shows to the user the successive rule applications that occur during a derivation step with actual parameters and global variables values. The debugging can be constrained to only particular rules that the user can select in the code editor. In parallel, a *profiler* provides the user with the time spent in each rules and functions of its models and makes it possible to identify bottlenecks in the execution.

Conclusion

L-Py is an integrated modeling framework based upon L-systems and Python. It is now a mature software that can be easily integrated into complex modeling scenario. FSPM are currently being developed with L-Py by different research groups from the OpenAlea community. An example is the conversion of a complex growth model of apple tree (Costes et al., 08) previously written with *L-studio/lpfg*. This model mixes stochastic topological construction with bio-mechanics model for the geometry. Thanks to syntax compatibility between L-Py and *lpfg*, the code portage mainly consisted in translating and simplifying the C++ instructions into Python. Although performances decreased a bit because of Python, scientific tools from Python and OpenAlea communities were immediately accessible within the model (for instance, 2D plot with *Matplotlib*). L-Py is also used as a teaching tools for high school lectures thanks to its intuitive interface, pythonic syntax and easy to read L-system rules.

Acknowledgments

The authors are grateful to P. Prusinkiewicz and his collaborators for inspiration with L-studio/VLab and for insightful discussions about this project. This project is partially supported by Agropolis Foundation.

References

M. Chelle and B. Andrieu: The nested radiosity model for the distribution of light within plant canopies. *Ecological Modelling 111*, 1998, pp. 75-91.

E. Costes, C. Smith, M. Renton, Y. Guédon, P. Prusinkiewicz and C. Godin: MAppleT: simulation of apple tree development using mixed stochastic and biomechanical models, *Functional Plant Biology 35(9&10),* 2008, pp. 936-950

D. Da Silva, F. Boudon, C. Godin and H. Sinoquet, Multiscale Framework for Modeling and Analyzing Light Interception by Trees, *Multiscale Modeling and Simulation* 7(2), 2008, pp. 910-933

P. Federl and P. Prusinkiewicz: Virtual Laboratory: an Interactive Software Environment for Computer Graphics. *In Proceedings of Computer Graphics International 1999*, pp. 93-100.

O. Kniemeyer and W. Kurth, 2008. The Modelling Platform GroIMP and the Programming Language XL. In *Applications of Graph Transformations with industrial Relevance: Third international Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and invited Papers, A. Schürr, M. Nagl, and A. Zündorf, Eds. Lecture Notes In Computer Science, vol. 5088. Springer-Verlag,* 570-572.

R. Karwowski and P. Prusinkiewicz: Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science* 86 (2), 2003, 19 pp.

C. Pradal, F. Boudon, C. Nouguier, J. Chopard and C. Godin: PlantGL: A Python-based geometric library for 3D plant modelling at different scales, *Graphical Models 71(1)*, 2009, pp. 1-21

C. Pradal, S. Dufour-Kowalski, F. Boudon, C. Fournier and C. Godin, OpenAlea: A visual programming and component-based software platform for plant modeling, *Functional Plant Biology, 35 (9 & 10),* 2008, pp. 751-760.

P. Prusinkiewicz: Art and science for life: Designing and growing virtual plants with L-systems. *In C. Davidson and T. Fernandez (Eds:) Nursery Crops: Development, Evaluation, Production and Use: Proceedings of the XXVI International Horticultural Congress. Acta Horticulturae* 630, 2004, pp. 15-28.

P. Prusinkiewicz, J. Hanan and R. Mech. An L-system-based plant modeling language. In: M. Nagl, A. Schuerr and M. Muench (Eds): *Applications of graph transformations with industrial relevance. Proceedings of the International workshop AGTIVE'99,* Lecture Notes in Computer Science *1779, Springer, Berlin,* 2000, pp.395-410.

P. Prusinkiewicz, L. Mündermann, R. Karwowski and B. Lane: The use of positional information in the modeling of plants. *Proceedings of SIGGRAPH 2001*, pp. 289-300.