in silico Plants Vol. 4, No. 1, pp. 1–16
https://doi.org/10.1093/insilicoplants/diab040
Advance Access publication 14 December 2021
Special Issue: Functional-Structural Plant Models
Technical Advance

in silico Plants
OXFORD

Towards virtual modelling environments for functional–structural plant models based on Jupyter notebooks: application to the modelling of mango tree growth and development

Jan Vaillant[1,2], Isabelle Grechi[1,2], Frédéric Normand[1,2] and Frédéric Boudon[3,4,*]

[1]CIRAD, UPR HortSys, 97455 Saint-Pierre, La Réunion, France
[2]HortSys, Univ. Montpellier, CIRAD, Montpellier, France
[3]CIRAD, UMR AGAP Institut, 34398 Montpellier, France
[4]UMR AGAP Institut, Univ. Montpellier, CIRAD, INRAE, Institut Agro, Montpellier, France
*Corresponding author's e-mail address: frederic.boudon@cirad.fr

Editor-in-Chief: Stephen P. Long

Citation: Vaillant J, Grechi I, Normand F, Boudon F. 2021. Towards virtual modelling environments for functional–structural plant models based on Jupyter notebooks: application to the modelling of mango tree growth and development. *In Silico Plants* **2021**: diab040; doi: 10.1093/insilicoplants/diab040

ABSTRACT

Functional–structural plant models (FSPMs) are powerful tools to explore the complex interplays between plant growth, underlying physiological processes and the environment. Various modelling platforms dedicated to FSPMs have been developed with limited support for collaborative and distributed model design, reproducibility and dissemination. With the objective to alleviate these problems, we used the Jupyter project, an open-source computational notebook ecosystem, to create virtual modelling environments for plant models. These environments combined Python scientific modules, L-systems formalism, multidimensional arrays and 3D plant architecture visualization in Jupyter notebooks. As a case study, we present an application of such an environment by reimplementing V-Mango, a model of mango tree development and fruit production built on interrelated processes of architectural development and fruit growth that are affected by temporal, structural and environmental factors. This new implementation increased model modularity, with modules representing single processes and the workflows between them. The model modularity allowed us to run simulations for a subset of processes only, on simulated or empirical architectures. The exploration of carbohydrate source–sink relationships on a measured mango branch architecture illustrates this possibility. We also proposed solutions for visualization, distant distributed computation and parallel simulations of several independent mango trees during a growing season. The development of models on locations far from computational resources makes collaborative and distributed model design and implementation possible, and demonstrates the usefulness and efficiency of a customizable virtual modelling environment.

KEYWORDS: Distributed 3D visualization; distributed environment; FSPM; Jupyter notebooks; mango tree.

1. INTRODUCTION

Functional–structural plant models (FSPMs) provide new opportunities to understand the complex interplays between plant growth, their underlying physiological functioning and the environment (Godin and Sinoquet 2005; Louarn and Song 2020). To do this, FSPMs combine a representation of the 3D structure of a plant with the modelling of physiological processes and environmental interactions. The architectural structure of the plant is defined on the basis of a small number of types of elementary units that are instantiated on different locations on a topological representation. The development of the plant is described

as the appearance, growth, ageing and death of the elementary units that are affected by physiological and environmental processes. Since the state and the number of units change over time, simulating plant growth corresponds to a class of problems formalized as dynamic systems with dynamic structures $(DS)^2$ (Giavitto and Michel 2001), which leads to the definition of dedicated formalisms (Godin *et al.* 2005) such as L-systems (Prusinkiewicz and Lindenmayer 1990).

During the last decades, dedicated modelling platforms (Federl and Prusinkiewicz 1999; Barczi *et al.* 2008; Hemmerling *et al.* 2008; Pradal *et al.* 2008; Boudon *et al.* 2012; de Reffye *et al.* 2021) have been developed. They allow the creation of a multitude of models, built on a series of specialized tools for the simulation of plant growth and functioning. Such integrative platforms usually rely on software components composed of multiple computer languages or formalisms. Models are created as scripts or as scientific workflows (Pradal *et al.* 2008; Lang 2019). Some platforms include a visual representation of the workflows in order to give an overview of the modelled processes and their interdependencies. However, their reuse by non-experts is usually limited to the modification of parameters, and the exploration of model outputs (extraction, transformation, export, plotting, etc.) is often cumbersome. Extending and customizing the model generally requires inputs from the authors of the original model. Furthermore, while some efforts have been made to port these tools over multiple operating systems, reproducibility and dissemination are hampered by the complexity inherent in their deployment and installation on new computers with different configurations.

On the other hand, modelling languages such as Python or R for statistical computation propose ready-to-use ecosystems of scientific packages (Millman and Aivazis 2011). For instance, the scientific Python ecosystem (Oliphant 2007) allows data processing and analysis. At the centre of these packages, multidimensional arrays allow the user to characterize sets of entities. Numerous tools make it possible to manipulate, visualize and perform statistical analyses. However, these tools are generally disconnected from FSPM simulations where flexible, graph-based structures are used to represent plants. Dedicated software tools are required for parsing model input/output and extracting homogeneous views of the data for plotting or analyses.

In a new initiative to alleviate these problems, we explored the use of the Jupyter framework (Kluyver *et al.* 2016) to create virtual modelling environments for plant models. To build the simulation environment, we combined and extended the modules of the Python scientific ecosystems, namely numpy, pandas, SciPy (Virtanen *et al.* 2020), xarray (Hoyer and Hamman 2017) and xarray-simlab (Bovy *et al.* 2021), based on multidimensional arrays to represent attributes of plant units over time. We complemented this with the igraph library (Csárdi and Nepusz 2006) to build, validate and visualize the topology of the plant using matrices that allow matricial algebraic operations to model, e.g. physiological processes involving distance relationships. Finally, we integrated the FSPM dedicated tools, L-Py (Boudon *et al.* 2012) and PlantGL (Pradal *et al.* 2009), into the Jupyter notebooks and used them to efficiently model and visualize the 3D architecture of the plants. The key points of this environment are: (i) the seamless integration into Jupyter that allows the easy collaboration, dissemination, visualization and introspection of the generated data and model processes; and (ii) a full compatibility with the Python scientific stack and, therefore, direct access to the vast numpy and SciPy ecosystem since the data are almost entirely modelled as multidimensional arrays.

To illustrate this approach, we developed a new implementation of V-Mango (Boudon *et al.* 2020), a model of mango tree development and fruit production that is composed of complex architectural and fruit growth processes sensitive to environmental (temperature, light, etc.), temporal and structural factors. The new design of the model, based on this environment, allows clear modularization of the original model, efficient computation, easy exploration of the results and simple, reliable installation for the user.

Our contribution can be summarized as follows:

-The definition of a virtual modelling environment based on the Jupyter notebooks and the Python scientific ecosystem. The use of conda, a software environment manager, or Docker, a virtualization service, allows easy deployment of the environment, both locally and remotely.
-The development of wrapping and visualization tools of FSPM modelling software modules for the Jupyter environment and for xarray-simlab-based scientific workflows, offering new possibilities to develop models in a modular manner and simulate and visualize plant development within notebooks.
-The reimplementation of the V-Mango model within this environment to illustrate its use for a complex FSPM.

While planning the requirements for a redesigned mango tree model and drawing conclusions from our results within a group of scientists with diverse scientific backgrounds, it became apparent that we were addressing some issues that were relevant to a wider audience of FSPM modellers and users. In particular, we addressed the shortcomings of current approaches related to dissemination, reproducibility, complex model handling and interoperability, and greater genericity (Louarn and Song 2020).

2. THE VIRTUAL MODELLING ENVIRONMENT

The intent of this project is to propose a virtual software environment that allows the creation and the execution of FSPM models. As reported by Capuccini *et al.* (2019), the idea of on-demand Web-based working environments on virtual infrastructures was envisioned by Candela *et al.* (2013). These working environments, dedicated to a community of practice, were originally referred to as Virtual Research Environments. While the Jupyter project and its notebooks provide a solid foundation for creating such environments, simulating and analysing complex modelling scenarios of plant growth create specific needs. In particular, specific formalisms such as growth grammar need to be integrated. 3D visualization and flexible interaction with 3D shapes in a distributed way are required. Workflow formalism, possibly with a multiprocessing computation capability, to gather and run a set of processes that define the model, is also necessary. On the basis of these concepts, we propose the definition of what we call a virtual modelling environment for FSPM. The different components of such an environment are described below.

2.1 Notebook-based environment

These last years have seen the emergence of a new way to communicate and collaboratively explore scientific computational ideas and data

analysis using notebooks. In these environments, raw code is interspersed with charts, figures, texts and equations. This allows the creation of a shareable, interactive computational narrative (Perkel 2018) where analysis or modelling scenarios can be textually described, with visual elements alongside their code. Some interactive features allow the manipulation of different parts of a model and its parameters. Notebook environments emerged from the concept of literate programming originally proposed by Donald Knuth (1984). They were first introduced in a number of commercial analysis packages such as Mathematica, Maple, Matlab, and later, in open-source software such as SageMath.

More recently, the Jupyter project, an open-source computational notebook ecosystem, has gained wide popularity. Part of its success is due to its clear and open format for its notebook representation. Moreover, thanks to a major redesign, it is possible to couple it with many programming languages. The foundational languages Julia, Python and R inspired the name of the project (Perkel 2018). Each language is introduced into the system as a specific computational kernel that is responsible for the interpretation of the code (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels). The Jupyter project is based on a distributed infrastructure and thus provides easy deployment on the Internet, e.g. with the JupyterHub project. Based on this, services such as MyBinder (Auer and Landers 2019) and Google Colaboratory (Bisong 2019) provide online services to run any public notebook with minimal configuration. Online resources for education based on the Jupyter infrastructure have been developed, like at UC Berkeley (Perez 2018), and modules such as nbgrader (Hamrick *et al.* 2017) have been developed to build online exams. Notebooks are also used to publish books or supplementary information on scientific papers.

Notebooks can be imported and run in different applications with different styles of the Jupyter project. The standard one is the display of the notebook as a simple editable Web page within the browser, composed of a sequence of cells for codes and charts, figures, texts and equations. An advanced version is the JupyterLab that makes it possible to manage different resources and to customize the display by arranging cell outputs in different ways so as to create a custom virtual modelling environment.

Furthermore, dedicated Web applications can be built from notebooks using the *voila* project (Voila 2020). Finally, notebooks can also be displayed as interactive slideshows for educational purposes, or run inside popular IDEs such as Visual Studio Code.

2.2 Integrating L-systems into notebooks

While models built with agnostic modelling languages such as Python or Java could be easily integrated within Jupyter notebooks, dedicated formalisms such as L-systems have been widely adopted by the modelling community to create FSPMs and require specific integration. As a first step to building a useful modelling environment for FSPMs, we integrated the L-systems formalism into notebooks. To do this, we reused the L-Py framework (Boudon *et al.* 2012) that combines L-systems constructs with Python. Specific notebook cells can be created with L-systems code and are executed by the L-Py interpreter. Since this interpreter is built on top of Python, the execution of this cell corresponds to the activation of a specific mode of interpretation rather than the activation of a different computational kernel. Variables are exchanged with any other Python cell. By using specific interface modules such as RPy2 (https://rpy2.github.io/), L-system cells can even be mixed with code cells from other languages such as R (see illustration in https://nbviewer.org/github/fredboudon/plantgl-jupyter/blob/isp2022/examples/r_and_py.ipynb).

This L-systems integration is illustrated in Fig. 1, which represents a notebook (https://nbviewer.org/github/fredboudon/plantgl-jupyter/blob/isp2022/examples/integration-demo.ipynb) composed of a series of cells that combine formatted text, equations and both Python and L-systems code. The goal of this model is to simulate the growth of a simple growth unit. The first cell gives the title of the notebook and mathematical details on the growth function. The next cell gives its Python implementation. The second cell of the code defines the parameters of the L-system that can be graphically controlled by the user. Using the growth function and the previously defined parameters, some L-systems rules are defined in the third cell. The cell is initiated with the magic command *%%lpy*, which makes it possible to write L-systems rules embedded in Python notebooks. Executing this cell directly generates a 3D dynamic plot in the browser. Different buttons make it possible to navigate within the simulation, allowing, e.g., the display of the entire animation of the simulation or forward or backward movement. Graphical parameters appear alongside the visualization, and the simulation is automatically updated when these values are edited. For now, parameters of the scalar, function and 2D curve type can be defined and manipulated. With such tools, it is possible to directly compare the code and its result, interact with the model and thus provide an educational experience for the audience.

2.3 The simulation framework

To organize and execute simulations, our environment is based on the Python library xarray-simlab (Bovy *et al.* 2021), a feature-rich and robust extension to the xarray library.

The xarray-simlab library provides a framework to compose complex computational models from sets of reusable components, called processes. A collection of processes can be combined to form a model, and their computational ordering is entirely deduced from process dependencies. In essence, those dependencies are created by explicitly linking processes via output variables (producing processes) and input variables (consuming processes). Variables declared within a process class may be annotated with other useful metadata like unit, description, validation functions or specific encoding settings. The set of variables declared inside a process class describes the process interface in terms of computed variables. They may be consumed by any other process in the model as long as no circular dependencies are created. However, circular dependencies are allowed if a variable is consumed with an offset over time, i.e. in the following simulation step. For the case of interdependent variables, their evolution should be estimated within a common process, for example, using an appropriate numerical solver.

The model—the predefined collection of processes—can be dynamically altered by plugging in or unplugging other processes, or by replacing a particular process with an alternative implementation. Processes may inherit from a base process class, and

A growth unit's growth defined using Lsystem rules

Growth of organs is simulated using a logistic function (Cookson et al., 2007).

$$\frac{L}{1+\exp-\frac{t-tip}{b}}$$

```
from math import exp
def growth_logistic(ttime, finalsize, tip, b):
    return finalsize / (1 + exp(-(ttime-tip)/b ))
```

Definition of graphical parameters

```
from openalea.lpy.lsysparameters import LsystemParameters
params = LsystemParameters()
try:
    params.load(open('integration.json'))
except:
    p = params.add_curve('axis')
    p = params.add_scalar('nbfolioles',3,minvalue=1,maxvalue=5)
```

```
%%lpy -a False -u dm -w 5 -s 630,400 -p params
Axiom:
    nproduce _(1) @Gc SetColor(green) F(5) [ &(40)_(0.3) F(2) SetGuide(axis,sum(LIn))
    for i in range(nbMetamers): nproduce I(i,0)[/(137*i)&(40)L(i,0)]
    nproduce I(i,0) @O ] F(5)

derivation length: int(maxTime//dt)
production:
I(i,t) --> I(i,t+dt)
L(i,t) --> L(i,t+dt)
interpretation:
I(i,t) --> nF(internode_size(i,t),0.1)
L(i,t) :
    l = leaf_size(i,t)
    dl = l/(1+nbfolioles/2)
    nproduce SetColor(colors[i])_(0.1)&(t) @v @Ge
    for i in range(nbfolioles-1):
        nproduce F(dl/2) [+(90*(1-i/nbfolioles))/(-30)~l(dl)][-(90*(1-i/nbfolioles))/(30)~l(dl)]
    nproduce F(dl/2,0.02) &(20)~l(dl)
```

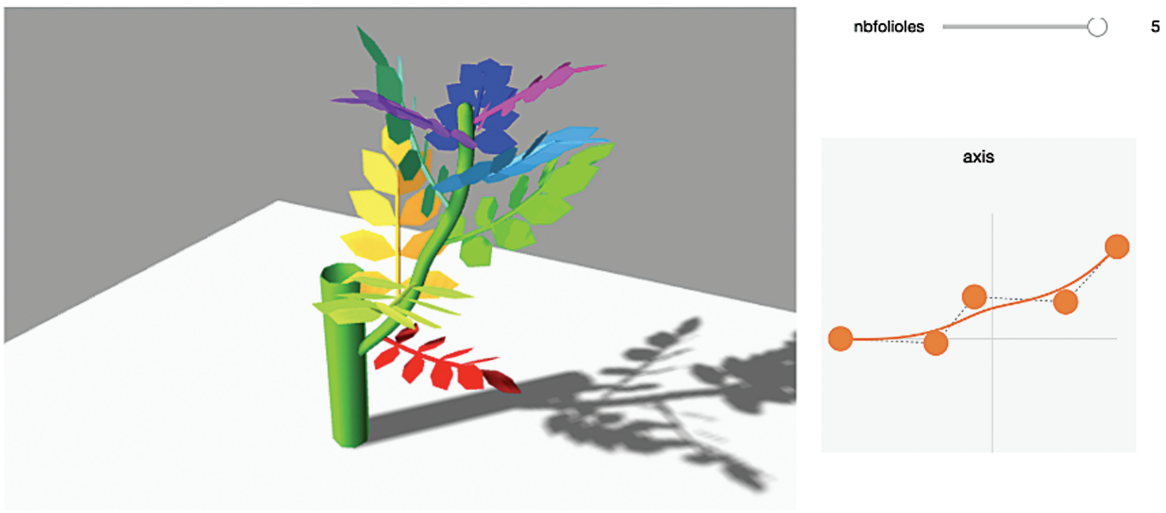

Figure 1. Integration of L-systems within a notebook. Cell code that starts with %%lpy contains L-systems rules. At the execution of the rules, a 3D visualization widget is displayed below. Graphical parameters are defined in the cell above using Python code and displayed on the right of the 3D visualization.

derived classes may implement a different model logic and provide additional output variables. Models and their processes and variables can be programmatically inspected, and the computational order, i.e. process dependency, can be easily visualized.

In order to execute a particular model, users create a model set-up in which they define the basic parameters for each simulation run. The most important are the time steps, the values of the model input variables and the name of the variables to be exported.

The xarray-simlab library provides great freedom in the design of a model and numerous ways to declare variables. We decided to separate three types of numerical data: pure constants like natural physical constants that are kept directly inside a Python module, variables that effectively vary during a simulation, and parameters that are either obtained from the literature or after a calibration process and are therefore constant within a simulation set-up. Each individual process may be parameterized with these separate files stored in the toml format. By default, an initialization function is associated with each process to load those parameter files and parameterize a process at startup. However, it is still possible to inject custom parameter values by reimplementing a process initialization function.

Simulation inputs and outputs are mostly composed of xarray data structures (i.e. labelled arrays and data sets). All features from xarray are therefore readily available for all input and output data: index- and label-based selections; interpolation and grouping of data; reshaping and combining data sets; reading and writing files; and advanced plotting.

To adapt this framework to FSPMs, we have extended the library with new features. L-systems models can be automatically integrated as processes, and the simulation runs with associated visualization (see following Section 2.4 below). While FSPMs simulate structures with a growing number of elements, xarray-simlab is originally designed to model the dynamics of structures with a fixed number of elements. We extended it so that all property arrays are automatically and consistently resized when new entities are created. The growth of the entity index over time does not make it possible to use xarray-simlab built-in parallelism capabilities. We therefore reimplemented the parallel processing of multi-model simulations using Python's standard multiprocessing library. Our solution is not as efficient as some GPU-based solutions for L-systems (Lipp *et al.* 2010), but it offers great flexibility and expressiveness to define modelling processes thanks to Python and its scientific ecosystem.

2.4 Plant representation

Plants can be seen as a collection of entities of various natures (e.g. vegetative and reproductive units). The topological relationship of those entities can be expressed as a tree graph, possibly multiscale (Godin and Caraglio 1998). Dedicated data structures such as the multiscale tree graph (MTG) (Pradal and Godin 2020) or a bracketed string representation (Prusinkiewicz and Lindenmayer 1990) have been developed to represent and manipulate these structures. Those representations have various advantages, like easy traversals through the parent–child relationships and the possibility to store items with arbitrary structures and properties of various types within the tree.

However, this flexibility comes with a cost and has several disadvantages that may make it a suboptimal choice, depending on the specific model structure, the research question and the requirements of the model environment. The main disadvantages are:

(i) No straightforward transformation to and from array structures suitable for utilizing features of the standard Python scientific stack. By default, the Python scientific stack is based on multidimensional arrays (in general, from the numpy library) and provides tools to analyse and plot such arrays. Using such tools for items stored within flexible structures requires the definition of queries to parse the structures and extract values in an appropriate order.

(ii) A lack of control over data consistency. Loosely typed data structures such as MTG do not ensure that values for a property stored within a tree are of identical types. Type checking needs to be introduced to ensure compatibility with standard scientific stacks.

(iii) Difficulties to store time series data, which are typical for most biological process models. By default, the standard plant data structures for FSPM are mainly designed to represent the current state of the simulation. Time series of structures can be created, but mapping between entities over time is challenging. Alternatively, time series of property values can be directly stored within a structure. However, trees need to be parsed each time one wants to export the time series of the properties of different structure entities in order to manipulate, analyse or plot them.

(iv) Possible computational inefficacy if operations need to be repeatedly applied on every or on a large subset of entities. In particular, structure parsing and type checking creates a performance overhead that can be avoided when working directly with multidimensional arrays. Moreover, array-based operations, optimized at the C level, generally outperform equivalent operations built using pure Python.

We therefore explored ways of expressing the properties of plant models and their topologies in multidimensional arrays by essentially creating arrays for properties and an adjacency matrix (a standard matrix representation of graphs). An additional time dimension may also be required for all properties that vary over time. A data structure, referred to as a data set, from the xarray library makes it possible to assemble the different properties and the topology in a coherent way. A constraint is that all entities have a common set of properties for their representation. L-system string representations are made compatible with the data set of properties by associating a unique id with each module of the L-systems string that represents the position in the arrays of the properties of the entities.

The main challenge is to take the growing number of entities over the simulation time into account. To do this, we extended the xarray-simlab framework to provide an automatic extension of the data structure representing the plant. Every array representing properties or topology is extended with new default values upon the appearance of new entities. By default, the NaN value is used to express the fact that a value is not set (i.e. entities that have not yet appeared or have been pruned) and should be determined by the appropriate process. As a constraint, the values are all expressed as floating-point values (the only nullable data type available).

2.5 Integrating L-systems into the simulation workflow

The definition of the processes of a simulation workflow based on xarray-simlab requires the precise definition of input and output

variables, and an execution function for each process. In the specific case of L-systems, input and output variables can be directly deduced from the model definition. Using L-Py specifications, model input variables are defined using the `extern` command. These variables are directly exported onto the process interface. As output, an L-system generates topological structures in the shape of L-strings, and 3D representations. To be compliant with the array-based plant representation of the simulation framework, the parameters of the different modules of the L-strings are grouped and stored as individual arrays. For each type of module, an index array is also generated. This makes it possible to directly access and manipulate all values of the parameters of all entities of the same type.

The integration of L-systems within the xarray-simlab scientific workflow is illustrated in Fig. 2 (https://nbviewer.org/github/fredboudon/plantgl-jupyter/blob/isp2022/examples/lpy-simlab/lpy_carbon_light.ipynb). In the first cell, the code of an L-system is defined. The `extern` commands (lines 5–7) define the input variables that control the simulation. Note that the `step_delta` is a custom variable of xarray-simlab that gives the current step duration, which is required as input for each step of the L-system simulation. The second cell of the code generates an xarray-simlab process from the L-system code, automatically exposing in and out variables. For the out variables, the `lpyprocess` function individually exports all the parameters of the custom modules as arrays. For instance, the `Metamer_t` variable stores all values for parameter t of all Metamer modules. Although it is transparent from the L-system point of view, the redirection of the module parameters into arrays is made through specific `Param` structures that are automatically generated for the simulation. For each module of an L-string, the `Param` structure mainly stores the id of the module it represents and has the possibility of retrieving or modifying its associated parameters within the array representation.

The third cell of the code defines a second process that simulates a simple carbon allocation procedure. To do this, some parameters of the L-system simulation are used, such as `Metamer_t`. Other parameters are filled in by the process, such as `Metamer_allocation`. Another process (code not shown) simulates the intercepted light (from a zenithal light source in this example). The fourth cell assembles these processes into a simulation workflow, defines its set-up and visualizes it. The workflow is then run and some of the variables are explored and visualized as 2D plots, such as the amount of intercepted light per metamer over time.

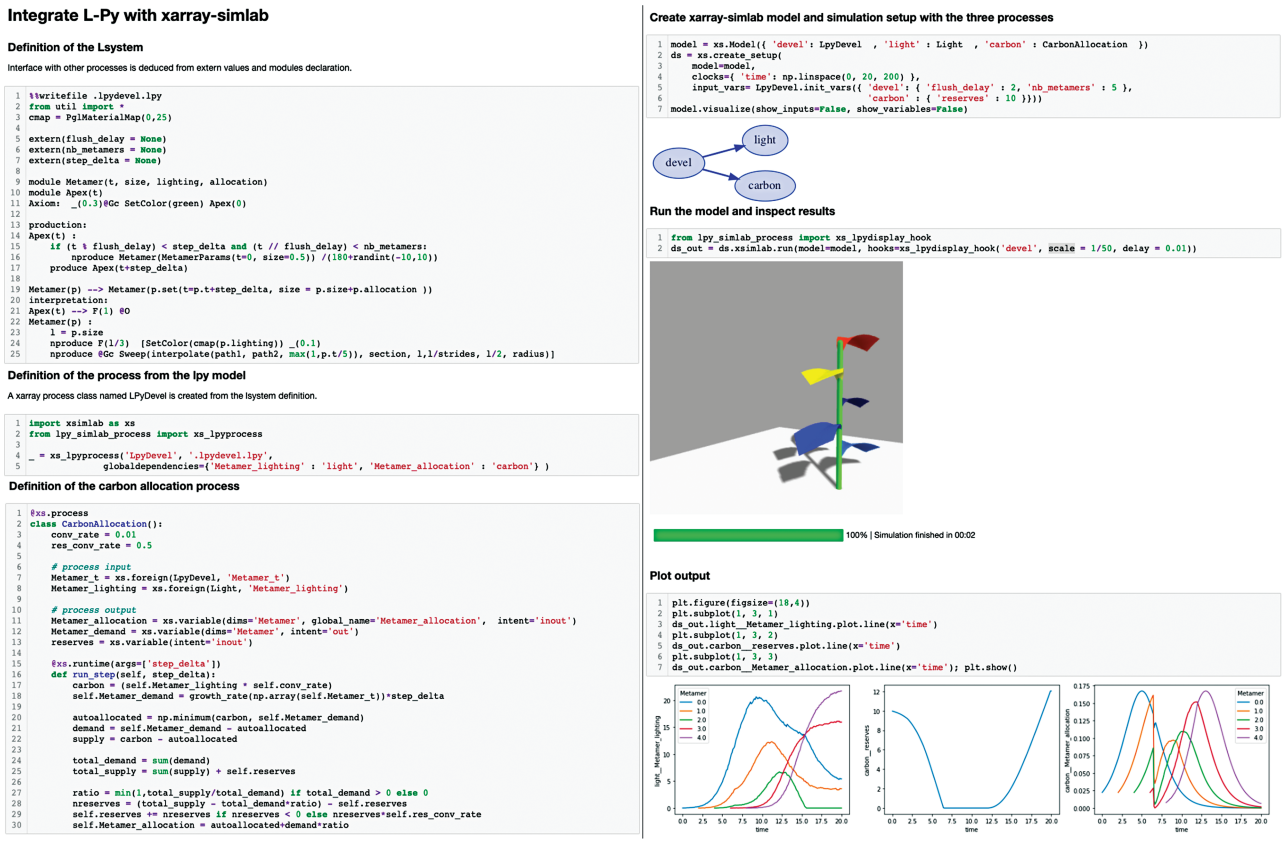

Figure 2. Integration of L-systems within a notebook-based scientific workflow.

2.6 3D visualization of plants in a client-server context

Both execution of the model and visualization of the simulation results are performed inside Jupyter notebooks using the plantgl-jupyter module (Vaillant and Boudon 2021). The Jupyter framework has a distributed architecture with a client (Web browser) and a server that contains the actual execution of the model. The result of a simulation must therefore be communicated by the server to the client using Internet protocols and be rendered by the Web client. While standard protocols for streaming and display of pictures and 2D charts have been adopted for a long time, the case of the 3D representation is less clear. Different libraries allow the definition and the streaming of 3D shapes (pythreejs, k3d, etc.). From the client side, high-level JavaScript libraries such as three.js (mrdoob 2021) allow the rendering of 3D shapes in the browser using WebGL.

To have interactive visualization, the time necessary to transmit the 3D representation has to be minimized. This time can be broken down into a time to encode the representation into a streamable representation, the actual time of streaming and the time to decode the representation, convert it into three.js objects, and the time for the rendering itself. Streaming time is proportional to the size of the encoded data, while encoding and decoding are proportional to the size of the original 3D scene. We tested different procedures to serialize and stream 3D representations with minimal encoding and streaming time, and we evaluated their benefits.

We first tested the pythreejs library (https://pythreejs.readthedocs.io/) that wraps three.js in Python and allows the streaming and display of 3D objects in a Web browser. The advantage of such an approach is that a similar representation is used both by the server and the client, minimizing data transformation. However, it does not allow a straightforward high-level representation for plants, and it leaves too little room for customizations. As such, many objects need to be stored with low-level mesh representation. Furthermore, due to its architecture, every custom user interaction with the browser's canvas (where the 3D scene is rendered) must be transmitted to the server's Python runtime, which in turn issues a signal to the renderer. This process slows down the visualization and user interaction. As an alternative, we tested the Draco mesh compression library (https://google.github.io/draco) that proposes an efficient compression algorithm that makes it possible to minimize streaming time. The drawback is that it requires converting the entire 3D representation into a single mesh, thus removing semantic information contained in the 3D scene.

Finally, we tested an alternative method that consists in directly using the serialization methods for 3D objects of the PlantGL library (Pradal *et al.* 2009) and combining it with a standard zip compression format. PlantGL representation allows the use of massive instantiation and high-level primitives with compact representation. From the client perspective, a port of the PlantGL API can be created using its original C++ code and a WebAssembly transpiler such as Emscripten (Zakai 2011). With such an approach, server and client share a high-level geometric representation. Discretization of the geometry for rendering is made by the client, allowing the server to dedicate its computational capability to the simulation. This last approach proved to be efficient with minimal coding costs since C++ routines already implemented in PlantGL could be made readily available in JavaScript. Nevertheless, minimal computational capabilities are required for the client. Our approach might be suboptimal for less well-equipped terminals such as tablets or smartphones.

2.7 Deployment of the virtual environment

In order to deliver deployable software modules for reproducible science, we rely on the conda management system. It allows us to build a software environment relatively independent of the host operating system. An environment is simply specified using a text file that lists all the required software modules and, possibly, their version. An environment can be created locally or on online services such as binder from this file.

To create custom FSPM environments, some of the software modules we developed, such as PlantGL and L-Py, were packaged for conda so as to be easily deployed. To rapidly build and assemble these packages, we defined a releasing pipeline based on online services for open software. Code is managed on the GitHub repository that is now configured to trigger automatic build on continuous integration services such as GitHub Actions or AppVeyor. If the build is successful and passes the automatic tests, a new version of the package is automatically published on the conda package online database (https://anaconda.org/), currently in the `fredboudon` channel. This allows the rapid publication of new features and bug correction, while preserving previous versions for reproducibility of previous environments.

3. APPLICATION TO THE V-MANGO MODEL

The integrative FSPM V-Mango was recently developed to simulate mango tree growth, phenology and fruit production (Boudon *et al.* 2020). Mango is an important fruit from tropical and subtropical regions and its cultivation faces several agronomic challenges. In particular, it exhibits (i) phenological asynchronisms partly due to complex interactions between vegetative and reproductive growth (Dambreville *et al.* 2013; Normand and Lauri 2018), and (ii) fruit heterogeneity at harvest in terms of size, gustatory quality and post-harvest behaviour. To understand such a complex behaviour, the precise modelling of the development of the mango tree architecture and its constituents (growth units (GUs), inflorescences and fruits) was required. A number of developmental processes were formalized and assembled. In the first version of the model, these processes were implemented as simple functions or L-system rules with no way to distinguish them from each other, except by their names. The model was implemented using Python, R and the L-Py framework. Glue-code between these different technologies was based on code developed in-house. In particular, the fruit model was implemented in R and, because of the simple interface provided, no interaction with the other submodels written in Python was possible.

In the new version of the model, called vmango-lab and published at https://github.com/fredboudon/vmango-lab/tree/isp2022 under an open-source license, we used the virtual modelling environment based on the notebooks presented above to redesign the model and reorganize its code. Most of the work consisted in defining processes and their inputs/outputs, and assigning corresponding model logic.

Code was also adapted to the use of multidimensional arrays to store parameter values. This new modular implementation allows the customization of the model for new use cases. To illustrate this, we present in the following sections the application of vmango-lab for testing carbon allocation strategies at the scale of individual GUs, and for performing parallel multi-model simulations. For each of the cases presented, a notebook with further detail is available at https://github.com/fredboudon/vmango-lab-demo/tree/isp2022.

3.1 Modularity and redesign of the V-Mango processes

V-Mango is a complex model where different processes simulate the growth and the phenology of the mango tree and its constituents of interest (GUs, inflorescences and fruits). To have a homogeneous and simple representation, the model formalizes the tree as a collection of GUs with flowering and fruiting attributes to represent information on inflorescences and fruits. During the simulation, the variables representing the different attributes of the GUs are passed between the different processes of the model and their values are updated to simulate the development of the mango tree and its constituents of interest. The model is now formalized as the sets of processes depicted in Fig. 3.

A first collection of processes, depicted in green in Fig. 3, simulates the appearance of the botanical entities in the architecture at different time steps. The appearance of these entities is broken down into elementary stochastic events that describe the occurrence, the intensity and the timing of their appearance and are modelled by binomial, Poisson and ordinal multinomial distributions, respectively. These distributions are assembled into a stochastic automaton that simulates the number, timing and fate (purely vegetative, flowering or fruiting) of new entities appearing on each terminal GU. The different distributions were parameterized from measured architectural data using generalized linear models, and probabilities were determined by architectural and temporal factors (Dambreville *et al.* 2013; Boudon *et al.* 2020). Different automata were formalized for the different types of entities (GUs, pure inflorescences and mixed inflorescences). For vegetative development, GUs appearing during the same growing cycle as their mother GU are distinguished from the ones appearing during the following cycle since different factors affect their appearance. All of this developmental information is contained in the Integrative Developmental process that updates the developmental attributes. Finally, the Topological Growth process uses this developmental information to extend the plant representation at each time step according to the simulated appearance date (feedback represented with a dashed arrow on the diagram).

A second set of processes, represented in purple in Fig. 3, simulates the development and growth of the individual entities. To do this, potential growth is estimated from empirical distribution (Organ Initiation process), thermal time models are run for the development and the growth of each entity (Organ Phenology process) and their spatial dimensions are increased accordingly (Organ Growth process).

The specific case of fruit growth in terms of dry matter is modelled based on carbon exchange. To do this, we extended and modularized the carbon balance model proposed by Léchaudel *et al.* (2005) for mango fruiting branches (represented in yellow in Fig. 3). First, carbon supplies (reserves and assimilation by photosynthesis) and demands (maintenance and growth) are estimated by different processes. Carbon reserves are estimated for the different GUs as a function of their size (Carbon Reserve process). Carbon demand for organ maintenance and fruit growth are determined in the Carbon Demand process as a function of organ size and potential growth for each fruit. Carbon assimilated by photosynthesis (Photosynthesis process) is calculated from intercepted light given by the Light Interception process.

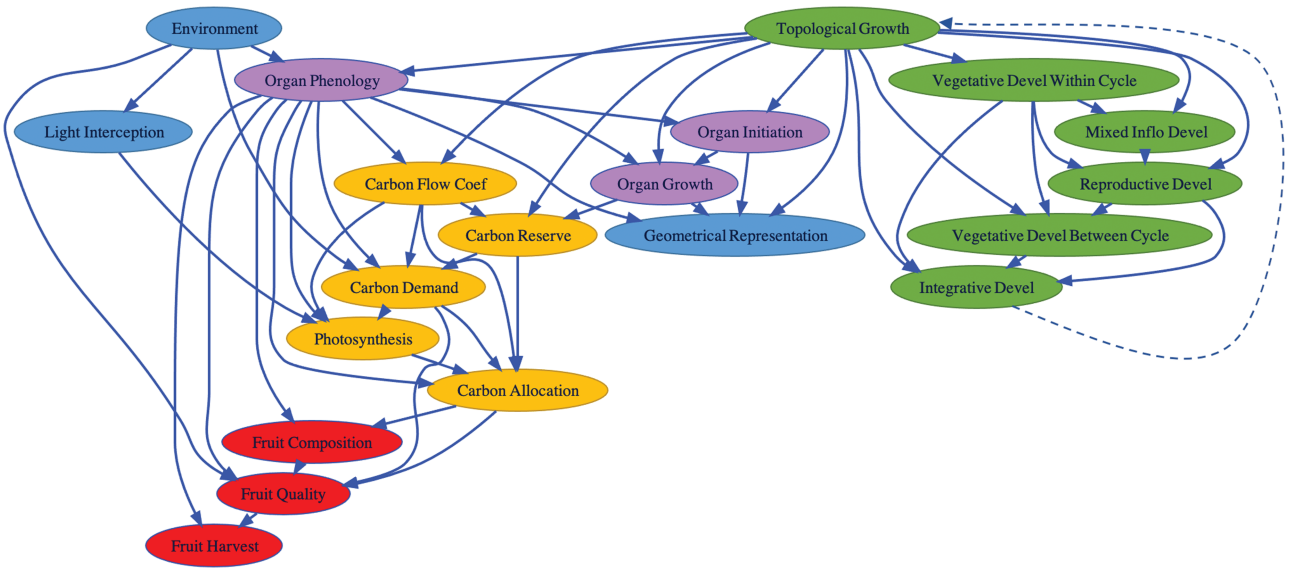

Figure 3. Workflow of the new V-Mango model. The diagram has been directly generated from the code, except for the dashed arrow. Explicit names to explain process purposes are used. Abbreviated names for processes are also used in the code (see Supporting Information—Table S1 for correspondences).

At this time, light interception is chosen from microclimate environments measured within canopies.

Based on the estimations of carbon supplies and demands, and on the exchange parameters determined from distances between GUs (Carbon Flow Coef process) and allocation rules, the Carbon Allocation process performs the allocation of carbon between organs and functions. Priority for carbon allocation is as follows: organ maintenance is first satisfied with local carbon supply; supplementary carbon is then exported to demanding fruiting GUs according to the estimated distance-based flow coefficient; and finally, fruit growth in dry mass is determined from their allocated carbon.

A last set of processes, represented in red in Fig. 3, simulates fruit growth in fresh matter and fruit quality. We integrated the model proposed by Léchaudel *et al.* (2007) for mango fruiting branches. The amount of soluble sugars, starch, acids and minerals in the fruit is empirically linked to fruit age and fruit dry mass (Fruit Composition process). Fruit growth in fresh matter is then modelled by water-related processes (water inflow into the fruit, driven by stem and fruit water potentials, and fruit transpiration), and sugar content is calculated (Fruit Quality process). The Fruit Harvest process calculates a ripeness index for each fruit. By default, the index is estimated as a threshold on the thermal time sum of the fruit.

3.2 Estimating carbon fluxes from a distance matrix

In the original model, carbon demand (for GU maintenance and fruit growth) and supply (from reserves and photosynthesis) were globally estimated for all the GUs composing a fruiting branch. As a limit, no topological change in the structure was allowed during fruit growth. In order to model more detailed exchanges, all GUs are now considered separately. Their carbon exchanges with the surrounding fruits are explicitly formalized using an exchange matrix (Carbon Flow Coef process) based on distances that can be re-evaluated at the appearance of each new GU. The distance between two GUs is expressed as the number of GUs in the shortest path between the two GUs in the tree structure. To reproduce the assessment of the level of carbon autonomy of branches according to their size found in the original version of the model, only carbon flows between GUs at a distance d from a fruit below a given threshold d_{max} are considered. The carbon flow coefficient F_{ij} from a leafy GU i towards a fruit borne by GU j, with GU j belonging to the set J of fruiting GUs, is calculated as:

$$F_{ij} = \frac{H(d_{max} - d_{ij})}{\sum_j^J H(d_{max} - d_{ij})} \quad (1)$$

where H represents the heaviside step function equal to one when distance d is less than or equal to d_{max}, and zero otherwise. This function can be seen as a simplified version of the distance-based allocation function proposed in previous studies (Lescourret *et al.* 2011; Reyes *et al.* 2020). The distance between all GUs in the structure is efficiently computed using SciPy routines and, in particular, the '`shortest_path`' function, from the adjacency matrix representing the tree structure. The distance matrix makes it possible to test more complex carbon allocation scenarios in the future.

3.3 Customization of the modelling workflow

Thanks to the introduced modularization, alternative processes can be easily defined to replace predefined ones. In the following example, changes in the Fruit Harvest process are illustrated in Listing 1. A new process to test the sucrose content of the fruit is defined. This process is defined as a Python class with the `@xsimlab.process` decorator (lines 1–2). Its `run_step` function (lines 4–8) first retrieves the variable `sucrose` that represents the sucrose content of each fruit with an array. The ripeness index is set to 1 if the sucrose content exceeds the threshold `sucrose_thresh` defined as a parameter of this process. A new model `customized_vmango` is then instantiated (line 11) as a copy of the vmango model, with removal of the `geometry` process (that computes the geometrical representation). The harvest process is replaced by the new custom process `HarvestByQuality`. This non-visual, and thus faster, model makes it possible to test for an alternative harvesting policy.

```
1  @xsimlab.process
2  class HarvestByQuality(Harvest):
3      sucrose = xs.foreign(FruitQuality, 'sucrose')
4
5      @xsimlab.runtime
6      def run_step(self):
7          sucrose_thresh = self.parameters.sucrose_thresh
8          self.ripeness_index = np.minimum(1, self.sucrose / sucrose_thresh)
9          self.harvested[:] = self.ripeness_index == 1
10
11 customized_vmango = vmango \
12     .drop_processes('geometry') \
13     .update_processes({'harvest': HarvestByQuality})
```

Listing 1. Customizing a model by replacing a process and unplugging another one (code slightly simplified for clarity).

New modelling workflows can thus be designed by partly reusing the default workflows. For instance, in the notebooks https://nbviewer.org/github/fredboudon/vmango-lab-demo/blob/isp2022/notebooks/1-0-modularity.ipynb and https://nbviewer.org/github/fredboudon/vmango-lab-demo/blob/isp2022/notebooks/1-1-arch_dev.ipynb, a workflow focussing on the architectural development alone is designed. In contrast, in https://nbviewer.org/github/fredboudon/vmango-lab-demo/blob/isp2022/notebooks/4-use_case_measure_and_simulate.ipynb, presented in more detail in the Section 3.5, a fixed architecture is considered and only phenology and fruit growth are simulated. A variety of workflows can thus be designed according to the modelling needs. Since workflows are coupled with L-systems, it can also be seen as a way to introduce modularity within L-system models.

3.4 Distributed simulations and visualization

The performance of the modelling environment and of the visualization tools is important to provide the user with an interactive and intuitive experience of the modelling process. The technological stack presented in this manuscript already provides many excellent solutions to overcome these challenges. However, certain aspects of our particular model logic and data representation, such as the inherent dynamic nature of the topology and the rendering of its geometric representation, required solutions that were not readily available. These solutions had to be implemented, in particular, the visualization of large PlantGL scenes in a notebook context and multi-model parallelization. To evaluate the usability and interactivity of the proposed visualization

system, we first assessed its performance on complex mango models. In a second step, we assessed the possibility to launch, display and analyse in parallel simulations with different initial conditions. In the model development process, these features offer interesting possibilities for calibrating a model or running a sensitivity analysis more efficiently.

On the basis of our tests, the rendering engine was able to efficiently transmit and render large scenes with great detail, even if the scene was generated on a distant system (Fig. 4). Our test machine was a laptop with a Debian GNU/Linux 10 system with 16 GB of memory and an Intel® Core™ i5-8250U CPU @1.60 GHz × 8, Intel® UHD Graphics 620. With the model execution and the visualization in the Web browser distributed locally, the transmission took several milliseconds. The frame rate (FPS) of the display varied from 40 to 10 for an average tree with 2000 to 3000 GUs (approximately 10^7 triangles), depending on the number and level of details in associated GU organs like leaves, flowers and fruits. Similarly, large scenes representing orchards (Fig. 4 and **Supporting Information—Video 1**) were smoothly visualized, and user interactions like rotation and zoom were seamlessly performed.

In a second test, a synthetic mango orchard [**see Supporting Information—Video 1**] was simulated. Each mango tree was initialized with the same structure, but different seed values controlled the random number generation used during the stochastic processes of tree development (Fig. 4).

Depending on the number of available CPU cores, multiple simulations were run in parallel with only a few lines of code (Listing 2

```
1 dataset = vmlab.run(setup, vmango,
2       batch=('seed', [{'topology__seed': seed} for seed in range(6, 10)])
3 )
4 dataset.harvest__nb_fruit_harvested.sum(dim='GU').plot(hue='seed')
```

Listing 2 : Creating and running a set-up in batch (parallel) mode and plotting of results.

The batch option (line 2) of the vmango-lab run function made it possible to pass the array of parameters used by the parallel execution. The 3D view of the parallel simulations was updated by each simulation independently, allowing a pleasant interactive view of the development and fruit production of the virtual 'orchard'. Results of the different executions were assembled into a common data set and could be directly plotted (line 4), resulting in the diagram shown in Fig. 5.

The performance of a simulation run varied with the size of the initial tree, the number of time steps, the number of geometries derived during the simulation and, last but not least, with the number and frequency of model outputs that needed to be merged into the resulting xarray data set. As a test case, we benchmarked the duration of a simulation run, both in sequential (four model runs on a single CPU core) and in parallel mode (four model runs, each on a single CPU core), over 2 years (daily time steps) with a varying number of GUs in the initial tree, and generating 3D visualization every 30 steps or only at the end of the simulation. The model was initialized with trees composed of 100–500 GUs. At the end of the simulation, final trees had 800–5500 GUs, which roughly translated into 4- to 10-year-old mango trees. The total simulation time increased linearly with the number of GUs of the final trees, with a higher slope for sequential simulation and 3D visualization generated every 30 steps (Fig. 6A). A comparison between sequential and parallel simulation shows an increasing speedup ratio with the number of cores used, and ranging from 1.5 to 2.2 for 2–8 CPU cores (Fig. 6B). The general performance was reasonable for a typical simulation set-up run on a personal computer.

However, for very large trees with several thousand GUs, the factor limiting computational time could be the available computer memory rather than the number of cores or the capacity of CPUs and GPUs. Since square adjacency and distance matrices

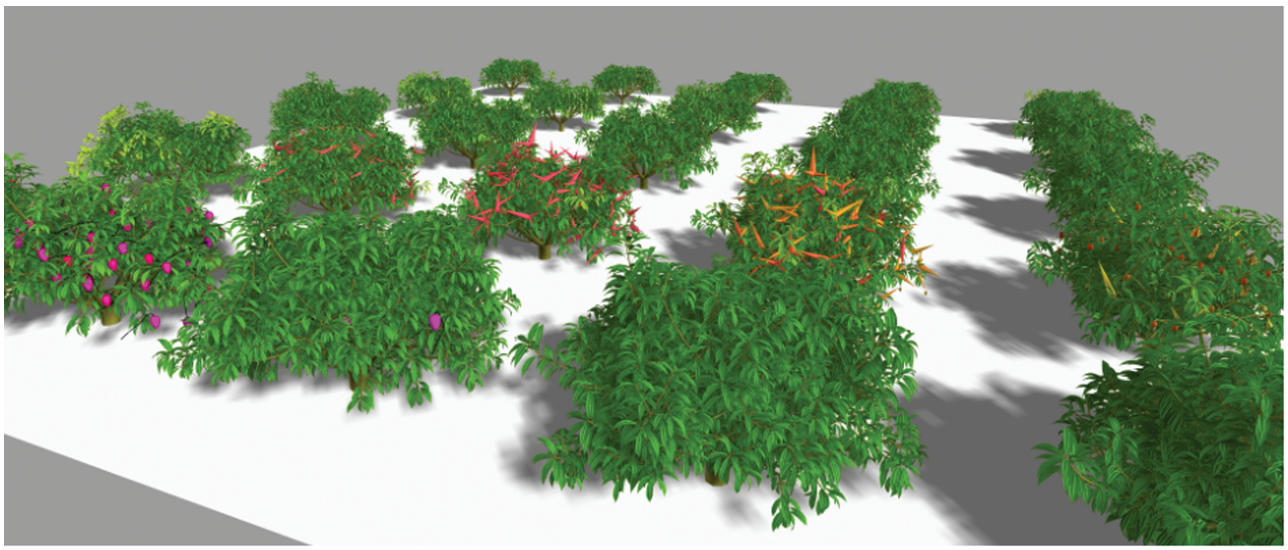

Figure 4. 3D rendering of a tree at selected simulation steps during several successive growing cycles. Simulation steps are from leftmost in the back to rightmost in the front.

grow exponentially with the number of GUs in the tree, the memory required to hold the data grows likewise. For example, allocating a single full distance matrix for a tree composed of 20 000 GUs requires 1.6 GB of memory (float32 data type). In the future, this limitation could be alleviated by integrating sparse implementations of multidimensional arrays in xarray-simlab based, for example, on a representation defined in the *sparse* library (https://sparse.pydata.org). Alternatively, a vmango-lab environment can be deployed on a virtual machine with a configurable amount of memory if required for large simulations.

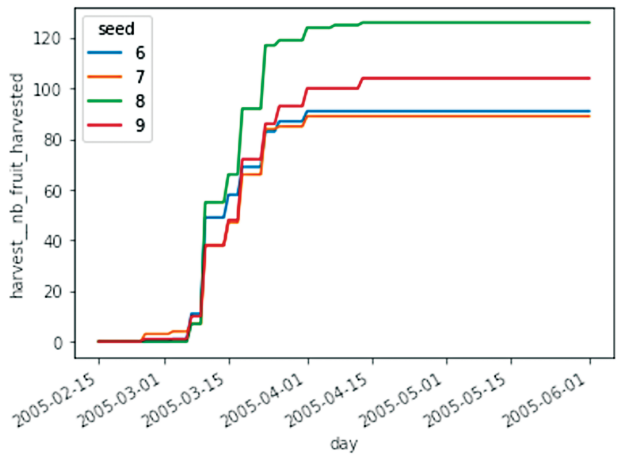

Figure 5. Cumulated number of fruits harvested per tree, plotted from the resulting data set of a parallel batch run of four simulations (four trees) with different seed values.

3.5 Study case: investigating source–sink relationships on measured architecture

The vmango-lab allows a wide range of uses and applications in agronomical research. In this section, we illustrate how the model could be used to investigate source–sink relationships, a widely studied issue in fruit production, by manipulating the Carbon Allocation process through the threshold distance d_{max} (Equation (1)). Model modularity makes it possible to run simulations for fruit growth- and fruit quality-related processes (represented in yellow and red, respectively, in Fig. 3), using observed architecture instead of architecture simulated with architecture-related processes (represented in green in Fig. 3). The corresponding notebook can be found at https://nbviewer.org/github/fredboudon/vmango-lab-demo/blob/isp2022/notebooks/4-use_case_measure_and_simulate.ipynb.

The study case consisted in a girdled branch bearing one fruit, whose architecture (topology, stem diameter, stem length and leaf number of all GUs) was measured in the field and represented with a simple drawing (Fig. 7A1 and A2). These data were easily formatted into a csv file used as model input. A 3D (mock-up, Fig. 7B) and a 2D (igraph, Fig. 7C1) representations of the observed architectural data set were generated by the model. Carbon flow between a GU and a fruit is controlled by a distance-based allocation function (Equation (1)). It was assumed that only leafy GUs, i.e. those producing photoassimilates, located below a threshold distance d_{max} to a fruit allocate carbon to support the growth of this fruit. Carbon flow from a leafy GU *i* to the fruit borne by GU *j* is proportional to the carbon flow coefficient F_{ij} that depends on the distance d_{ij} (Equation (1)). The set of leafy GUs supporting fruit growth according to d_{max} is illustrated in Fig. 7C1 using an igraph representation of the data, for d_{max} values of 4 or 10 GUs. Fruit growth in fresh mass was simulated in these two cases,

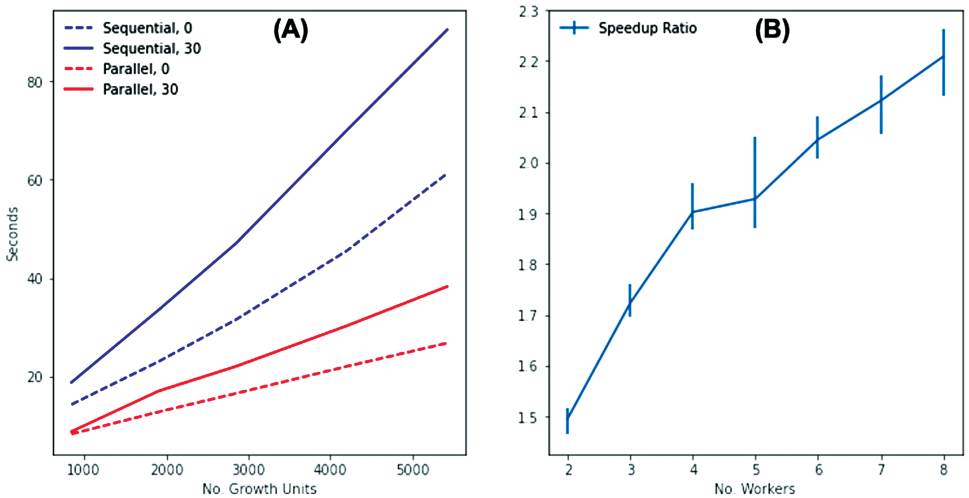

Figure 6. Evaluation of simulation performances. (A) Comparison of computational time (seconds) of four simulation scenarios: two parallel multiprocessing simulations with four models run on four cores (in red) and two sequential single process (in blue) simulations of four model runs, both with either a geometry derived at the end of the simulation (0) or a geometry processed every 30 days (30); (B) Speedup ratio: time elapsed for n models in a sequential, single process simulation over time elapsed for n workers (cores/processes) in parallel mode with 10 repetitions (mean, minimum and maximum values are displayed).

and predicted and observed fruit fresh mass at harvest were compared (Fig. 7C2). The light environment was set to an average default value for all the GUs of the branch. However, it would be possible to use observed values if the light environment was measured. Results showed that 10 GUs was certainly a more adequate d_{max} value than 4 GUs, indicating that almost all the leafy GUs of the branch might have contributed to fruit growth. By varying the value of the input parameter d_{max}, it is possible to explore different source–sink relationships within the branch through simulation. The approach developed at a local (branch) scale could further be extended at a global (tree) scale, as proposed, for example, in peach (Lescourret *et al.* 2011) or apple (Pallas *et al.* 2016) fruit crops.

4. DISCUSSION

4.1 Notebooks for FSPM

The Jupyter-based modelling environment presented in this manuscript brings together and customizes scientific and modelling tools of the Python ecosystem to build robust and shareable FSPMs. Building on such an environment makes it possible to benefit from the regular improvements of this dynamic ecosystem. At the centre of the Jupyter project, the notebook format allows the user to create a computational narrative of a modelling scenario. Such an approach is efficient because it allows collaborators and users to test a model, as illustrated by the notebooks that supplement this manuscript, thus demonstrating the reproducibility and ease of dissemination of research work on FSPMs. In

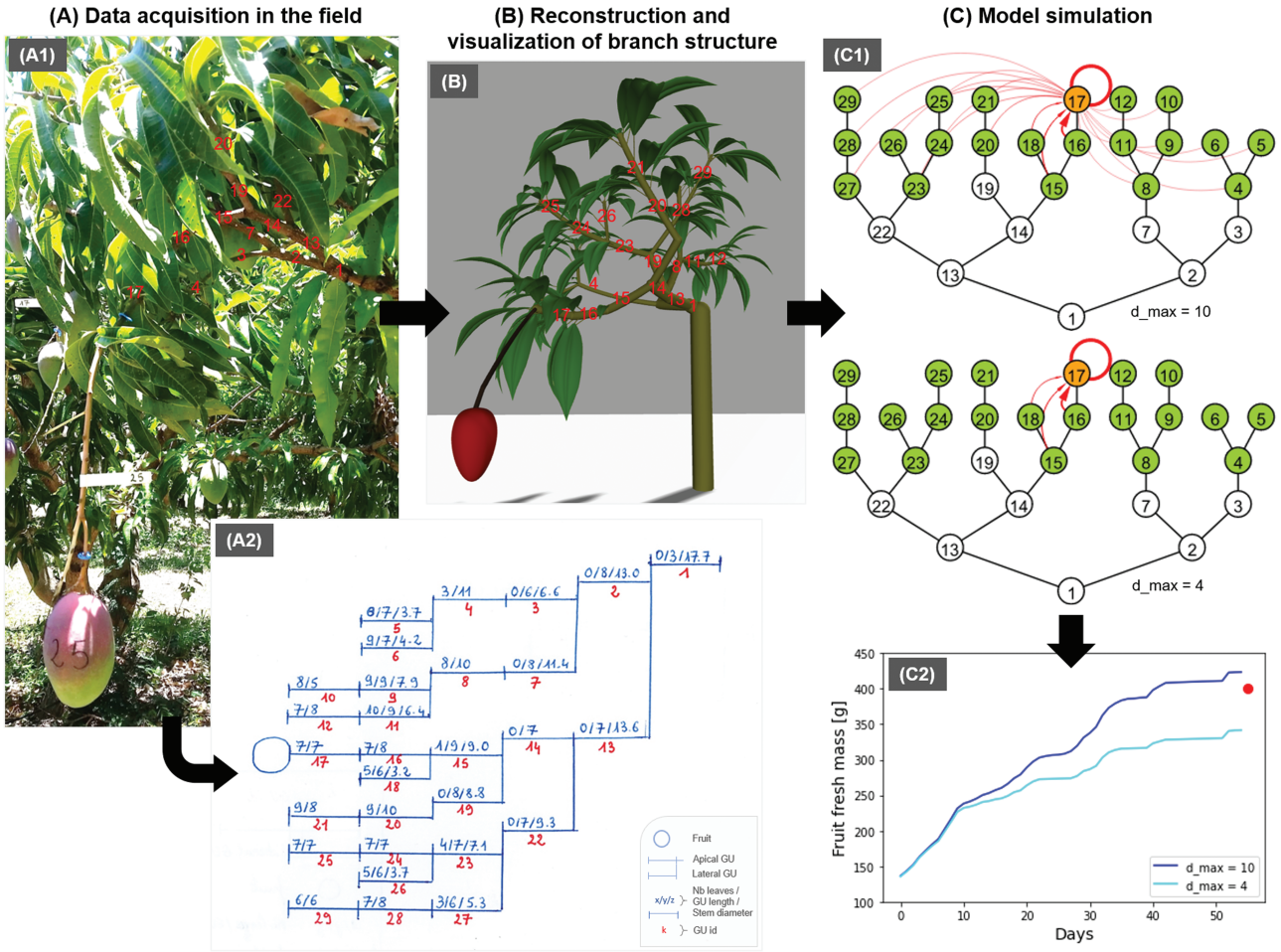

Figure 7. Protocol of use of the fruit model on measured architecture. (A) Data acquisition in the field: (A1) picture of the original branch; and (A2) drawing of the manual measurement of the branch architecture. (B) Reconstruction and 3D visualization of the branch (a fake trunk was added for visualization). (C) Fruit growth simulation with a threshold distance (d_{max}) of 4 and 10 GUs for the Carbon Allocation process, and visualization of model outputs: (C1) carbon flow between all GUs and the fruiting GU; and (C2) simulated fruit fresh mass dynamics as of the end of fruit cell division (time = 0 d). Red numbers in pictures A1, A2 and B and numbers in picture C1 are the GUs' id. Orange, green and white dots in picture C1 represent the fruiting GU, leafy GUs and GUs without leaves, respectively. Red arrows represent the distance between source and sink GUs, with arrow width inversely proportional to distance. The red point in picture C2 is the observed fresh mass of the fruit at harvest.

particular, the use of notebooks and the conda package management system make it possible to clearly specify software dependencies, hypotheses of the model and actual parameter values. There is still a possible limitation since the link to any external sources of data can be inconsistent over time and create non-reproducible experiments. A solution to this problem is to reference only the data distributed in the modelling project or to follow the FAIR guidelines (Wilkinson *et al.* 2016).

Moreover, organizing the code, the descriptive texts and illustrations in a didactic way requires special care. The notebook cells can be executed in any order. This has the advantage of allowing flexible experiments, but makes it sometimes difficult to assess the exact origin of the results. As such, notebook design policy should be followed to ensure consistency of the experiments over time (Rule *et al.* 2019).

4.2 Continuous delivery

Relying on the conda management system to access standard scientific tools and to package some of our specific modelling tools proves to be efficient and allows easy deployment over different computers with different operating systems or clusters or clouds. The continuous delivery pipeline we used, based on online services, allows us to develop new features more transparently for collaborators and users since they can have a view of the new code produced and its releasing state through the pipeline. Based on our experience, this fosters the interaction with other groups and the diffusion of the software.

4.3 FSPM modelling

The adaptation of Jupyter notebooks to FSPMs requires the integration of specific tools and formalisms such as L-system and 3D visualization capabilities. The possibility to mix cells of L-system code with standard code and detailed descriptions containing text, equations and figures makes it possible to create a modelling narrative, with different cells to compute, display and comment on the different results of the narrative. Nevertheless, the L-system code has a non-standard execution mode since it generates an entire animation. The visualization interface allows the exploration of this animation by running the model as a whole or for a reduced number of steps. Several modes of execution are thus nested in such a notebook: the execution of the L-system cells and the execution of the steps of the model. This may be particularly confusing for the beginner. Our experience revealed that computer science Master's students were at ease with such an approach, while plant science Master's students with little programming experience preferred the standard L-system interface provided by L-Py. One reason for that was the limited debugging tools provided by the notebook environment, in particular, for the execution of an L-system model. In order to improve the tools, it would therefore be important to upgrade this feature.

The 3D visualization of the growth of a plant structure modelled with an L-system requires an interactive visualization. The existing solutions were considered to be unsatisfactory since they may need extensive loading time, which runs counter to the interactivity of the tools. The approach we designed relies on transpiling the PlantGL library in WebAssembly. It was relatively simple to develop and proved to be efficient to execute for standard computers. As a drawback, it requires the maintenance of specific tools for 3D visualization.

The possibility of distributing simulation computation allows the efficient computation of multiple models with varying parameters, and thus makes it possible to perform efficient sensitivity analysis. As it comes natively within our framework, this feature is easy to use. As a future step, modelling the growth in parallel of a multitude of plants would create interesting possibilities for the study of crop growth. However, this would make it necessary to account for plant interactions, and to thus extend the modelling formalism to allow interactions between processes computed in parallel.

The virtual modelling environment presented in this manuscript is largely based on Python. Some modelling softwares of the FSPM community are built using alternative programming languages such as C++ or Java (Hemmerling *et al.* 2008; Griffon and de Coligny 2014). Jupyter notebooks are compatible with a large number of languages, including Java and C++. Communication between languages is facilitated by the Jupyter project but can still be complicated in some cases (e.g. Python-Java). As a result, integrating Java-based tools within the framework presented here would be difficult. However, similarly to what we did with Python, L-Py and PlantGL, integration of Java-based FSPM modules into Jupyter can certainly be undertaken in order to provide a complementary Java-based modelling environment in notebooks.

4.4 Collaborative design of workflows

Rather than maintaining 'second-order' documents about processes, dependencies, variables and units, we looked for an approach that would enable us to directly inspect the model code—as the single source of truth—and to visualize the model structure in order to collaboratively develop the model and its processes without the need to inspect the actual source code itself. Due to the model modularity and the choice of process granularity, the model can be easily extended and adjusted, even by collaborators with moderate proficiency in the programming language. This increases the usability of the model and its environment for a larger group of researchers.

With such features provided by xarray-simlab and our customizations, vmango-lab allows fast model development, interactive model exploration and collaborative model design—not least due to its seamless integration into the Jupyter framework. However, if required, any simulation and model can be run headless (without a Jupyter frontend or kernel) as a standard Python module. This is particularly important for efficient debugging.

4.5 Data representation

Many aspects of xarray and xarray-simlab have been designed for use in the earth sciences and thus have a strong focus on longitude–latitude grids. These libraries therefore have no generic support for modelling topological relations and graphs such as plant entities. In particular, they lack support for growing structures (i.e. growing indices along one or more dimensions), sparse representation of topologies and modelling of inherently cyclical biological processes. However, we have shown that an approach based on encoding topologies and entity properties into multidimensional arrays is viable and computationally efficient, even for large trees with several thousand GUs. This enables the user to represent the tree topology at a given moment, as well as to represent its evolution and the changes

in its property values over time. However, modelling architectures with a particularly high number of elements would require extra optimization such as the integration of sparse matrix representation. Moreover, modelling multiscale structures with GUs and their individual organs (inflorescences, fruits and leaves) may become cumbersome with multidimensional arrays.

Apart from the obvious advantage of having seamless access to the Python scientific stack, we believe that a great benefit of this approach is that all Python libraries assembled in the vmango-lab environment have excellent documentation and a comparatively large user base. This further improves dissemination of models and reinforces training and exercises.

4.6 Improvements brought to the V-Mango model by the virtual modelling environment

The quality of a model does not only rely on valid and rigorous mathematical descriptions of its processes, but on its design and ability to enable a rapid exploration of new hypotheses as well. With the redesign and reimplementation of V-Mango within the virtual modelling environment, some improvements occur directly in relation to the new structure of the model, demonstrating the practical implications of such an approach. In particular, splitting the model into multiple processes proved to be highly useful since they can be reused, recombined and adapted for different purposes. While the results of the original model were successfully reproduced (see https://nbviewer.org/github/fredboudon/vmango-lab/blob/isp2022/notebooks/vmango_archdev_evaluation.ipynb), new modelling scenarios can be easily built, like the one described in the study case, thanks to the new modular design. Changes in topology are efficiently expressed using L-system processes. On the other hand, functional processes that mainly deal with the evolution of physiological or geometrical parameters can now be directly modelled with multidimensional arrays. We found such an approach to be efficient in terms of coding and execution. Exported arrays of properties allow the exploration and visualization of the results of the simulation.

Since all processes are able to operate with a common time step of 1 day, it is possible to conveniently import the results of any process into a new process and transform its variables by introducing a new logic. For instance, stochastic processes like the forecasting of the fate of a GU can now be intercepted by other processes (e.g. future pest and disease models) in order to alter, for example, the potential fruit number into an effective fruit number per GU.

The computation of carbon exchange between non-fruiting and fruiting GUs is now dynamic during the fruit growth season. Each fruiting GU may have an individual harvest date. Moreover, new GUs that appear during the fruit growth season are included in the structure as carbon providers once they have reached their final development stage. Fruit removal or the appearance of new GUs during the fruit growth season results in a recomputation of all source–sink relationships. Carbon exchanges between GUs could also be extended to global compartments of the tree, such as fine roots, coarse roots and wood. Such a dynamic and flexible model could be used and customized to explore carbon exchange within the tree in the future.

4.7 Conclusion

Thanks to the Jupyter project, advances in Web browser technologies and the development of cluster and cloud computing, new possibilities are now available for modelling locations far from computational resources, allowing collaborative and distributed model design and implementation. The major limitations for modelling FSPM into notebooks were the lack of FSPM simulation formalisms such as L-systems in notebooks, and the limited availability of 3D visualization, for which we proposed efficient solutions.

We demonstrated the usefulness and efficiency of a customizable, virtual environment used for mango tree growth and production modelling. The environment was assembled and customized from a set of high-quality libraries. Reproducibility is enhanced thanks to clear specification of processes in the xarray-simlab framework and the documentation provided in the notebook to create the simulation narrative. Interoperability was excellent for data provided as multidimensional arrays or in any other compatible container.

Using a standard representation of multidimensional arrays to represent plant properties improves the efficiency of modelling and the coding process because it requires far less custom codes to extract, transform and visualize data. Those features are provided out-of-the-box by the Python scientific stack. Furthermore, being compliant with standard data representations reduces the risk of a lock-in into a specific modelling platform since many libraries/platforms are compatible with such representations. The modelling environment should ideally be just a set of well-integrated and exchangeable tools. Maintaining dedicated FSPM modelling platforms and their comprehensive documentation is time-consuming. The FSPM community might lack a critical mass to rely on the contributions of source codes that include features, bug fixes and documentation from the community.

As an application of the virtual modelling environment, a major redesign of the V-Mango model was performed and resulted in a modular and easily customizable model that can efficiently generate complex sets of trees. Different modelling scenarios can be investigated, such as the exploration of source–sink relationships on measured architectures presented in this manuscript. This new design will provide a solid foundation for future modelling experiments on mango trees. Additionally, we plan to use this virtual modelling environment for other complex FSPMs, such as the MappleT model (Costes *et al.* 2008), to enhance the genericity of our approach.

SUPPORTING INFORMATION

The following additional information is available in the online version of this article—

Table S1. Explicit vs. abbreviated names of processes defined in the vmango-lab model. File processnames.odt.

Video 1. 3D animation of a process-based parallel simulation of a mango orchard. A new representation of each tree is computed every 14 days. Video speed is set to be two times faster than the actual model execution.

ACKNOWLEDGEMENTS

The authors would like to thank Benoit Bovy for his support, Christophe Pradal for his useful comments and all

open-source contributors to the Jupyter project and to the libraries we used. We thank the reviewers for their constructive remarks and suggestions.

SOURCES OF FUNDING

This work was carried out as part of the CIRAD DPP COSAQ agronomical research program (activities 2015–21) funded by the European Regional Development Fund and the Conseil Régional de la Réunion (DPP COSAQ).

CONFLICT OF INTEREST

None declared.

CONTRIBUTIONS BY THE AUTHORS

Conceptualization: J.V., I.G., F.N., F.B.; Software: J.V., F.B.; Model application: I.G., J.V.; First draft of the manuscript: F.B., J.V., I.G.; Review and editing: J.V., I.G., F.N., F.B.

DATA AVAILABILITY

pgljupyter is available at https://github.com/fredboudon/plantgl-jupyter/ and vmango-lab at https://github.com/fredboudon/vmango-lab with instructions for the installation process. All examples in the Section 2 are available as notebooks in a demo repository at https://github.com/fredboudon/plantgl-jupyter/blob/isp2022/examples and can be inspected with nbviewer and reproduced either locally or on a binder instance. The notebooks described in Section 3 are available at https://github.com/fredboudon/vmango-lab-demo/tree/isp2022.

LITERATURE CITED

Auer E, Landers R. 2019. Creating reproducible and interactive analyses with JupyterLab and Binder. In: 34th Annual Conference of the Society for Industrial and Organizational Psychology. https://mybinder.org (3 January 2022).

Barczi JF, Rey H, Caraglio Y, de Reffye P, Barthélémy D, Dong QX, Fourcaud T. 2008. AmapSim: a structural whole-plant simulator based on botanical knowledge and designed to host external functional models. *Annals of Botany* **101**:1125–1138.

Bisong E. 2019. Google colaboratory. In: *Building machine learning and deep learning models on Google Cloud Platform*. Berkeley, CA: Apress. doi:10.1007/978-1-4842-4470-8_7. https://colab.research.google.com (3 January 2022).

Boudon F, Persello S, Jestin A, Briand AS, Grechi I, Fernique P, Guédon Y, Léchaudel M, Lauri PÉ, Normand F. 2020. V-Mango: a functional-structural model of mango tree growth, development and fruit production. *Annals of Botany* **126**:745–763.

Boudon F, Pradal C, Cokelaer T, Prusinkiewicz P, Godin C. 2012. L-py: an L-system simulation framework for modeling plant architecture development based on a dynamic language. *Frontiers in Plant Science* **3**:76.

Bovy B, McBain GD, Gailleton B, Lange R. 2021. benbovy/xarray-simlab: 0.5.0 (version 0.5.0). *Zenodo.* doi:10.5281/zenodo.4469813

Candela L, Castelli D, Pagano P. 2013. Virtual research environments: an overview and a research agenda. *Data Science Journal* **12**:GRDI75–GRDI81.

Capuccini M, Larsson A, Carone M, Novella JA, Sadawi N, Gao J, Toor S, Spjuth O. 2019. On-demand virtual research environments using microservices. *PeerJ Computer Science* **5**:e232.

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

Csardi G, Nepusz T. 2006. The igraph software package for complex network research. *InterJournal, Complex Systems* **1695**:1–9. http://igraph.org (3 January 2022).

Dambreville A, Lauri PÉ, Trottier C, Guédon Y, Normand F. 2013. Deciphering structural and temporal interplays during the architectural development of mango trees. *Journal of Experimental Botany* **64**:2467–2480.

de Reffye P, Hu B, Kang M, Letort V, Jaeger M. 2021. Two decades of research with the GreenLab model in agronomy. *Annals of Botany* **127**:281–295.

Federl P, Prusinkiewicz P. 1999. Virtual laboratory: an interactive software environment for computer graphics. In: Proceedings of Computer Graphics International'99: Canmore, AB, Canada. 93–100.

Giavitto JL, Michel O. 2001. MGS: a rule-based programming language for complex objects and collections. *Electronic Notes in Theoretical Computer Science* **59**:286–304.

Godin C, Caraglio Y. 1998. A multiscale model of plant topological structures. *Journal of Theoretical Biology* **191**:1–46.

Godin C, Costes E, Sinoquet H. 2005. Plant architecture modelling—virtual plants, dynamic and complex systems. In: Turnbull C, ed. *Plant architecture and its manipulation. Annual plant reviews*. Oxford: Blackwell Publishing, 238–287. doi:10.1002/9781119312994.apr0171

Godin C, Sinoquet H. 2005. Functional-structural plant modelling. *The New Phytologist* **166**:705–708.

Griffon S, de Coligny F. 2014. AMAPstudio: an editing and simulation software suite for plants architecture modelling. *Ecological Modelling* **290**:3–10.

Hamrick J, Bussonnier M, Frederic J, Granger B, Page L, Ragan-Kelley M, Willing C. 2017. nbgrader: a tool for creating and grading assignments in the Jupyter notebook. In: SciPy 2017, Austin, TX, 10–16 July 2017.

Hemmerling R, Kniemeyer O, Lanwert D, Kurth W, Buck-Sorlin G. 2008. The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. *Functional Plant Biology* **35**:739–750.

Hoyer S, Hamman J. 2017. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software* **5**:10.

Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, Kelley K, Hamrick J, Grout J, Corlay S, Ivanov P, Avila D, Abdalla S, Willing C, Jupyter Development Team. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In: *Positioning and power in Academic Publishing: players, agents and agendas*. Amsterdam: IOS Press, 87–90. doi:10.3233/978-1-61499-649-1-87

Knuth DE. 1984. Literate programming. *Computer Journal* **27**:97–111.

Lang M. 2019. yggdrasil: a Python package for integrating computational models across languages and scales. *In Silico Plants* **1**:diz001; doi:10.1093/insilicoplants/diz001

Léchaudel M, Génard M, Lescourret F, Urban L, Jannoyer M. 2005. Modeling effects of weather and source–sink relationships on mango fruit growth. *Tree Physiology* **25**:583–597.

Lechaudel M, Vercambre G, Lescourret F, Normand F, Génard M. 2007. An analysis of elastic and plastic fruit growth of mango in response to various assimilate supplies. *Tree Physiology* **27**:219–230.

Lescourret F, Moitrier N, Valsesia P, Génard M. 2011. QualiTree, a virtual fruit tree to study the management of fruit quality. I. Model development. *Trees* **25**:519–530.

Lipp M, Wonka P, Wimmer M. 2010. Parallel generation of multiple L-systems. *Computer & Graphics* **34**:585–593.

Louarn G, Song Y. 2020. Two decades of functional-structural plant modelling: now addressing fundamental questions in systems biology and predictive ecology. *Annals of Botany* **126**:501–509.

Millman KJ, Aivazis M. 2011. Python for scientists and engineers. *Computing in Science & Engineering* **13**:9–12.

Mrdoob. 2021. three.js. https://github.com/mrdoob/three.js (3 January 2022).

Normand F, Lauri P-É. 2018. Integrated mango production: objectives and challenges. *Acta Horticulturae* **1228**:377–383.

Oliphant TE. 2007. Python for scientific computing. *Computing in Science & Engineering* **9**:10–20.

Pallas B, Da Silva D, Valsesia P, Yang W, Guillaume O, Lauri PE, Vercambre G, Génard M, Costes E. 2016. Simulation of carbon allocation and organ growth variability in apple tree by connecting architectural and source–sink models. *Annals of Botany* **118**:317–330.

Perez F. 2018. Sea change: what happens when Jupyter becomes pervasive at a university? In: *JupyterCon*, New York, USA. https://www.oreilly.com/radar/sea-change-what-happens-when-jupyter-becomes-pervasive-at-a-university/ (3 January 2022).

Perkel JM. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* **563**:145–146.

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

Pradal C, Dufour-Kowalski S, Boudon F, Fournier C, Godin C. 2008. OpenAlea: a visual programming and component-based software platform for plant modeling. *Functional Plant Biology* **35**:751–760.

Pradal C, Godin C. 2020. MTG as a standard representation of plants in FSPMs. In: *Book of abstracts of the 9th International Conference on Functional-Structural Plant Models: FSPM2020*, 5–9 October 2020. Germany: University of Hannover. https://hal.inria.fr/hal-03059523/file/abstract7.pdf (3 January 2022).

Prusinkiewicz P, Lindenmayer A. 1990. *The algorithmic beauty of plants*. Berlin, Heidelberg: Springer-Verlag.

Reyes F, Pallas B, Pradal C, Vaggi F, Zanotelli D, Tagliavini M, Gianelle D, Costes E. 2020. MuSCA: a multi-scale source–sink carbon allocation model to explore carbon allocation in plants. An application to static apple tree structures. *Annals of Botany* **126**:571–585.

Rule A, Birmingham A, Zuniga C, Altintas I, Huang SC, Knight R, Moshiri N, Nguyen MH, Rosenthal SB, Pérez F, Rose PW. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter notebooks. *PLoS Computational Biology* **15**:e1007007.

Vaillant J, Boudon F. 2021. jvail/plantgl-jupyter: (version v1.2.0). *Zenodo:*. doi:10.5281/zenodo.5513337

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat İ, Feng Y, Moore EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, van Mulbregt P; SciPy 1.0 Contributors. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* **17**:261–272.

Voila: rendering of live Jupyter notebooks with interactive widgets (2020). https://github.com/voila-dashboards/voila (3 January 2022).

Wilkinson MD, Dumontier M, Aalbersberg IJ, Appleton G, Axton M, Baak A, Blomberg N, Boiten JW, da Silva Santos LB, Bourne PE, Bouwman J, Brookes AJ, Clark T, Crosas M, Dillo I, Dumon O, Edmunds S, Evelo CT, Finkers R, Gonzalez-Beltran A, Gray AJ, Groth P, Goble C, Grethe JS, Heringa J, 't Hoen PA, Hooft R, Kuhn T, Kok R, Kok J, Lusher SJ, Martone ME, Mons A, Packer AL, Persson B, Rocca-Serra P, Roos M, van Schaik R, Sansone SA, Schultes E, Sengstag T, Slater T, Strawn G, Swertz MA, Thompson M, van der Lei J, van Mulligen E, Velterop J, Waagmeester A, Wittenburg P, Wolstencroft K, Zhao J, Mons B. 2016. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data* **3**:160018.

Zakai A. 2011. Emscripten: an LLVM-to-JavaScript compiler. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, October 2011. 301–312. https://emscripten.org/ (3 January 2022).