Title: Building simulation models of developing plant organs using *VirtualLeaf*

Authors: Roeland M.H. Merks[1,2,*] and Michael Guravage[1,2]

Running head: Modeling with *VirtualLeaf*

1. Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands

2. Netherlands Consortium for Systems Biology/Netherlands Institute for Systems Biology (NCSB-NISB), Science Park 123, 1098 XG Amsterdam, The Netherlands

* Corresponding author: roeland.merks@cwi.nl

Summary:

Cell-based computational modeling and simulation are becoming invaluable tools in analyzing plant development. In a cell-based simulation model, the inputs are behaviors and dynamics of individual cells and the rules describing responses to signals from adjacent cells. The outputs are the growing tissues, shapes and cell-differentiation patterns that emerge from the local, chemical and biomechanical cell-cell interactions. Here, we present a step-by-step, practical tutorial for building cell-based simulations of plant development with *VirtualLeaf*, a freely available, open-source software framework for modeling plant development. We show how to build a model of a growing tissue, a reaction-diffusion system on a growing domain, and an auxin transport model. The aim of *VirtualLeaf* is to make computational modeling better accessible to experimental plant biologists with relatively little computational background.

Keywords: plant development; organ growth; cell division; cell growth; mathematical modeling; cell-based modelling; systems biology; computational modelling; reaction-diffusion; biomechanics; auxin

1. Introduction

Computational modeling is becoming a key tool in the study of biological development ***(1-5)***. Instead of focusing on the phenotypes of single gene knock-out lines or over-expression lines, computational models help the researcher to unravel the function of the gene in the context of the dynamics of the regulatory gene networks. In such a *systems biology* approach, the researcher starts with a hypothesis of the biological mechanism that he/she studies. The researcher then implements this hypothetical mechanism in a computational model and studies to what extent the computational model reproduces the biological observations, and where the comparison fails. These discrepancies offer the most valuable information, because they identify components missing from the model, incorrect interactions between existing components, or wrong values for the model parameters. The researcher hypothesizes a series of new components and tests if simulation models extended with these new components better reproduce the experimental observations. A next step is to experimentally test the validity of the components used to update simulation model. Typically, the experiments will falsify the new hypothesis or point at further factors that need to be considered in the model, prompting a new round of the *systems biology cycle* ***(6)***. In essence this approach does

not differ from the traditional empirical approach. The computational model replaces the hypothesis, which biologists typically express in terms of box-and-arrow diagrams (often also called 'model'). The computational model helps to check the internal logic of the hypothesis, it predicts unexpected outcomes of the component interactions, and it identifies specific experimental tests to test the hypothesis.

To build a computational model, most experimental biologists depend on computational biologists to build the models for them, even if they want to test a relatively simple, "back-of-the-envelope" kind of idea. To make constructing and simulating models of biochemical networks more accessible, a range of simulation tools has been made available (*see* **Ref.** *7* for review). These systems give useful insight in the dynamics of single cells, but they are rarely suitable for studies of developing plant organs because of their multicellularity.

Unraveling the development of a plant organ requires insight at multiple biological scales at the same time: 1) the biochemical networks regulating single cells, 2) the resulting behavior of the cells (i.e. cell expansion, cell division, cell differentiation), 3) the signals received from adjacent cells, 4) the resulting plant tissues and cellular differentiation patterns, 5) the resulting biomechanics of the whole tissue and its effects on the single cells. For developmental studies a useful technique is cell-based modeling ***(7-9)***. The input to cell-based models are the biochemical networks of individual cells, and the resulting cell behaviors; the output of such a model are the resulting tissue patterns and shapes.

VirtualLeaf makes it possible for plant biologists to design simple, cell-based models of plant development. To model tissue growth and biomechanics, it uses a Monte-Carlo approach. These tissue growth models are combined with dynamic models of intracellular regulatory networks and intercellular transport of phytohormones, which are simulated using an ordinary differential equation approach (*see* **Ref.** *7* for details). Although the details of these simulation methods typically remain hidden to the user, they can be modified if necessary without touching the model definition.

To set up a model in *VirtualLeaf*, the first step is to design a *cell-based* hypothesis: what do the cells do (i.e. the cell behaviors), what networks regulate these behaviors (intracellular dynamics), what inputs do the cells obtain from their neighbors and how do the responds to them. The next steps are to implement these rules in *VirtualLeaf*, to simulate the model, and to analyze it. Section 2 shows how to prepare your computer for running *VirtualLeaf*. Section 3 provides instructions on the basic usage of *VirtualLeaf*, and demonstrates how to build models of a growing plant tissue, a reaction-diffusion mechanism. The section ends by describing a detailed model of auxin transport.

2. Materials

Building a model in *VirtualLeaf* requires basic knowledge of the programming language C++. Basic knowledge of differential equations is useful too (*see* Note 1). Suitable tutorials can be found on the internet. All required tools have open source versions and can be downloaded for free. Download and install a C++ compiler, the graphical library Qt and the source code of *VirtualLeaf*. *VirtualLeaf* can be run on Windows, Linux and

MacOSX.

2.1. Required software: C++ compiler and libraries

Download and install the Qt Software Development Kit (Qt-SDK) from *http://qt.nokia.com/downloads*. The download page presents you first with a licence choice, GPL or commercial, and then with list of QT downloads based on operating system and machine architecture. Choose the "Complete Development Environment" appropriate for the operating system you are using. The file you download is a self-extracting archive; when executed it will display a graphical user interface that will guide you through the installation process.

On **MacOSX** also install the XCode Development environment from the MacOSX DVDs.

2.2. Source code

Download the source code of *VirtualLeaf* from:

http://virtualleaf.googlecode.com/files/VirtualLeaf-v1.0-src.zip.

Unpack the archive to the folder where you wish to install the Virtual Leaf.

Windows:

```
> c:\Documents and Settings\mge\simulations
```

Linux/MacOSX:

```
~/simulations
```

Experienced users can get the latest development version of *VirtualLeaf* using the mercurial version control system. To do so, install mercurial and type the following in a terminal shell:

```
> hg clone https://virtualleaf.googlecode.com/hg/ virtualleaf
```

2.3. Compile the VirtualLeaf framework

Windows:

Open a Qt Command prompt by choosing 'Qt Command Prompt' from the 'start' menu, then go to the folder where you have unpacked the source code of *VirtualLeaf*, e.g., (replace "mge" for your own user name)

```
> cd c:\Documents and Settings\mge\simulations
```

Change to the *VirtualLeaf* source directory.

```
> cd VirtualLeaf\src
```

Start the compilation procedure.

```
> set MAKE=mingw32-make
> ming32-make
```

Linux and MacOSX.

Open a terminal (on MacOSX: type ‘Terminal’ in Spotlight and press enter; Terminal is in /Applications/Utilities/).

Go to the directory where you unpacked the *VirtualLeaf*.

```
> cd ~/simulations
```

Change to the *VirtualLeaf* source directory.

```
> cd VirtualLeaf/src
```

Start the compilation procedure.

```
> make
```

2.4. Test VirtualLeaf

Once the compilation is complete, you will find the *VirtualLeaf* binary in `virtualleaf/bin` and the models in `virtualleaf/bin/models`. Test *VirtualLeaf* on Windows and Mac by double clicking on the binary; on Linux type `../bin/VirtualLeaf` at the terminal command prompt.

3. Methods

VirtualLeaf implements a suite of biological objects and processes required for modeling plant development, including cells, rigid cell walls, biochemical networks, and transport of chemical signals. You can define the model by describing the properties of these biological objects and the biochemical networks in a small section of C++-code that implements a model definition plugin to VirtualLeaf.

Box 1 outlines the structure of this file. The first line, #include "vleafmodel.h" reads a series of definitions required for building a *VirtualLeaf* model. Each code block defines the behaviors of one type of the model element: cells, and cell walls. *MyModel::ModelID* allows you to specify a name for your model. *MyModel::NChem* specifies the number of

chemical species (e.g. proteins, phytohormones, etc.) the model requires. *MyModel::CellHousekeeping* is used to specify mechanical changes or state changes of the cells. *MyModel::OnDivide* is used to define the rules to be executed after cell division, e.g., the redistribution of the chemicals in the parent cell. *MyModel::CellDynamics* and *MyModel::WallDynamics* define the differential equations describing the biochemical networks within the cells and within the walls. *MyModel::CelltoCellTransport* defines the rules for active and diffusive transport of chemicals over cell walls.

3.1. Basic usage of the *VirtualLeaf*

After starting *VirtualLeaf* with an existing model plugin or your own, the main window appears, and a pop-up window displays version and license information. Dismiss the information window to reveal the default model. The model's name is visible at the top of the window and is included in the list of models visible under the 'Models' menu. The simulation can be started and paused by toggling the 'Simulation paused' item in the 'Run' menu, or more easily by typing 's' on the keyboard. To run a different, predefined model, go to the 'Models' menu, and select one of the model descriptions.

To change the value of one of the predefined simulation parameters, click on the 'Edit parameters' item under the 'Options' menu, or type the keyboard shortcut 'Ctrl-E' (or 'Command-E' on Mac). A window will appear with parameter names followed by

their values. Once you have made your changes, save them by clicking the 'Write' button in the lower right corner of the screen.

You can run the model on different, initial cellular geometries, called "leaves". To select a different "leaf", choose 'Read Leaf' from the file menu. Leaves are stored as XML files in the directory 'data/leaves'. To store a cellular geometry, select 'Save Leaf'.

To store an image of the simulation results, choose "Snapshot" from the File menu. Specify the image format by using the appropriate file extension. Most common image formats are available, including PDF, PNG, and JPEG. The full list of file formats is listed in the field "File type:" of the Snapshot pop-up dialog.

To save snapshots at regular intervals, for example to make a simulation movie, in the parameter editing window change the value of "datadir" to an appropriate directory on your file system (under "Data Export"), and toggle the 'Start saving movieframes' item from the View menu. To change the export interval, modify parameter 'storage_stride' from the parameter window.

You can also export data in CSV-format (compatible with Excel) to report on the positions, size, and chemical content of cells and chemicals, and to export some basic data on the leaf morphology. To export data on a single morphology, select the 'Export cell data' from the 'File' menu. To save data at regular intervals, you can make use of the

parameters ‘export_interval’ and ‘export_fn_prefix’. A value of zero for ‘export_interval’ means that no data is exported.

3.2. Prepare an empty model plugin

In the next sections, we show how to develop your own model plugin in *VirtualLeaf* (*see* Note 2). First, you will need to prepare an empty model plugin.

1. In a terminal window, go the *VirtualLeaf* tutorial directory.

   Windows:

   ```
   > cd virtualleaf\src\TutorialCode
   ```

   Linux/MacOSX:

   ```
   > cd virtualleaf/src/TutorialCode
   ```

2. Copy the empty model template ‘Tutorial0’ to a name of your choice, e.g ‘MyModel.’

   Windows: Use the Finder or Explorer tools, or type the following at the terminal prompt:

```
> copy Tutorial0 MyModel
```

Linux/MacOSX:

```
> cp -pr Tutorial0 MyModel
```

3. Remove temporary files generated by the previous compilation.

```
> cd MyModel
> make clean
```

4. Rename all the remaining files using the name you chose.

   Windows: Use the Explorer tool.

   MacOSX/Linux:

```
> rename tutorial0 mymodel *
```

5. In all remaining files, replace occurrences of the strings 'Tutorial0' and 'tutorial0' with ones reflecting the name you chose.

6. Rename your new model. Edit the file *mymodel.cpp*, and on line 37 replace the string:

```
return QString( "0: Empty model template (does nothing)" );
```

with one reflecting the name you choose:

```
return QString( "My own model in VirtualLeaf" );
```

7. Build your new (empty) project, by typing:

Windows:

```
> set MAKE=mingw32-make
> mingw32-make
```

Linux:

```
> make
```

MacOSX:

```
> qmake -spec macx-g++ mymodel.pro; make
```

(or qmake mymodel.pro, then compile from XCode).

8. If still running, quit and restart *VirtualLeaf*. The model should now appear in 'Models' menu.

3.3. Tissue growth

We will first build a simple model of cell division driven tissue growth. Use the model template you have constructed in Section 3.2.

1. In the file *mymodel.cpp* append the following code to the `MyModel::CellHouseKeeping` method:

   ```
   c->EnlargeTargetArea(par->cell_expansion_rate);
   ```

   This statement instructs the cells to expand by the value `cell_expansion_rate` after each relaxation cycle.

2. Recompile the model plugin.

   Windows:

   ```
   > set MAKE=mingw32-make
   > mingw32-make
   ```

Linux/MacOSX:

```
> make
```

3. Restart *VirtualLeaf,* select your updated model from the 'Models' menu and start it by pressing 's'. You should see a single, expanding cell.

4. To instruct cells to divide after they doubled in size, append the following code to the `MyModel::CellHouseKeeping` method:

```
if (c->Area() > 2 * c->BaseArea() ) {
   c->Divide();
}
```

   The line starting with '`if`' is a conditional statement. When a cell's area grows larger than twice its original area. i.e. `BaseArea`, the cell will divide (see Note 3).

5. Recompile and restart *VirtualLeaf*, select your updated model and press 's' to start it. The cells should expand and divide (Figure 1).

6. By default, `Divide()` instructs cells to divide over the shortest principal axis. To add an optional division axis replace `c->Divide()` with:

```
c->DivideOverAxis(Vector(0,1,0));
```

   Recompile and start the model as instructed in steps 2 and 3.

7. Let’s make the cells divide over an axis of random orientation. Add the following two header files directly after the existing header files to define π and add functionality for random functions.

```
#include "Pi.h"
#include "random.h"
```

8. Replace the `c->DivideOverAxis(axis)` statement with:

```
double orientation = Pi*RANDOM();

Vector axis(sin(orientation),cos(orientation),0.);
c->DivideOverAxis(axis);
```

   Recompile and start the model as instructed in steps 2 and 3. The result should look similar to Figure 2.

3.4. Reaction-diffusion and cell differentiation

In example 3.3 all cells behaved exactly the same. In an actual plant tissue, cell-cell communication and pattern formation mechanisms instruct cells to behave differently according to the local signal concentrations. This experiment illustrates how to implement a classic reaction-diffusion hypothesis for pattern formation ***(10)***.

Meinhardt ***(10)*** proposed that leaf venation patterns can be formed by reactions between diffusing chemicals:

$$\frac{dY}{dt} = dA - eY + \frac{Y^2}{1 + fY^2};$$
$$\frac{dA}{dt} = \frac{cA^2S}{H} - \mu A + D_A \nabla^2 A + \rho_0 Y;$$
$$\frac{dH}{dt} = cA^2S - \nu H + D_H \nabla^2 H + \rho_1 Y;$$
$$\frac{dS}{dt} = c_0 - \gamma S - \varepsilon YS + D_S \nabla^2 S,$$

with ‘Y’ a cell differentiation factor, ‘A’ a self-reinforcing activator, ‘H’ an inhibitor ‘S’ a substrate and all the other symbols constants.

We will run these biochemical reactions in each of the cells by implementing a set of differential equations that assume mass-action kinetics (see Note 1).

1. First, construct a sufficiently large model tissue to test the reaction-diffusion equations: take the model constructed in Section 3.2 and make the cells divide over their shortest axis (step 5 in Section 3.3). That is, in the method `MyModel::CellHouseKeeping`, use the division statement (see Note 4):

```
c->Divide();
```

2. Specify the number of chemicals the model considers, by inserting the following code

in the method `MyModel::NChem()`:

```
int MyModel::NChem(void) { return 4; }
```

3. Recompile, start *VirtualLeaf* and load your model.

4. Run the model until you have obtained a model tissue with several hundred cells.

5. Switch off cell growth. Click ‘Cell growth’ under the ‘Options’ menu such that the option is unchecked.

6. Give the cells suitable initial values. Open the ‘Edit Parameters’ dialog in the ‘Options’ menu. Under the heading ‘Auxin transport and PIN1 dynamics’, change the first four values of ‘initval’ to **`0.001`** (within the text box, use the arrow keys to navigate to the front of the list). When done click ‘Write’ on the parameter dialog and then choose ‘Reset Chemicals’ from the ‘Edit’ menu. If you put your mouse pointer over a cell the values of the chemicals will be shown. Choose ‘Randomize PIN1 Transporters’ from the ‘Edit’ menu to add some noise (see Note 5).

7. Save the tissue template to the *virtualleaf/data/leaves* directory. Choose ‘Save Leaf’ from the ‘File’ menu, navigate to the directory *virtualleaf/data/leaves* and choose a suitable name for your template, e.g. *myleaf.xml*.

8. Use this tissue template as the default for your model. Edit your model's header file, e.g. *mymodel.h*, and add the following line at the end of the file just before the closing curly-brace (see Note 6).

```
virtual QString
      DefaultLeafML(void) { return QString("myleaf.xml"); }
```

   Replace *myleaf.xml* with the name you chose for your tissue template.

9. Let each cell run the reaction-diffusion equations proposed by Meinhardt (1976). Insert them into `MyModel::CellDynamics` method so it becomes:

```
void MyModel::CellDynamics(CellBase *c, double *dchem) {
     double Y = c->Chemical(0), A = c->Chemical(1),
     H = c->Chemical(2), S = c->Chemical(3);

     dchem[0] = ( par->d * A - par->e * Y + Y*Y /
                  (1 + par->f * Y*Y ) );
     dchem[1] = ( par->c * A*A*S/H - par->mu *
                  A + par->rho0*Y );
     dchem[2] = ( par->c * A*A*S - par->nu*H + par->rho1*Y );
     dchem[3] = ( par->c0 - par->gamma*S - par->eps * Y * S );
}
```

10. Color the cells according to the values of the chemicals. Insert the following code into the `MyModel::SetCellColor` method (see Note 7):

```
double red=c->Chemical(1)/(1.+c->Chemical(1));
double green=c->Chemical(0)/(1.+c->Chemical(0));
double blue=c->Chemical(3)/(1.+c->Chemical(3));
color->setRgbF(red,green,blue);
```

11. Recompile your model, restart *VirtualLeaf* and select your model from the menu.

12. In the 'Edit parameters' dialog, set suitable values for the parameters. For example *d*=`0.002`, *e*=`0.1`, *f*=`10`, *c*=`0.004`, *mu*=`0.12`, *nu*=`0.04`, *rho0*=`0.03`, *rho1*=`0.0003`, *c0*=`0.02`, *gamma*=`0.02` and *eps*=`0.4`. To save the parameters, click 'Write' on the parameter dialog. Choose the 'Save leaf' item from the 'File' menu and rewrite the template to *virtualleaf/data/leaves/mymodel.xml*. Choose 'yes' to overwrite.

13. Start your model. Some cells will turn green, others black, but not much will happen. The reason is that we have not yet implemented chemical diffusion.

14. To implement Fick's law of chemical diffusion, insert the following code into the `MyModel::CelltoCellTransport` method:

```
// Passive fluxes (Fick's law)
for (int c=0;c<NChem();c++) {

  if (w->C1()->BoundaryPolP() || w->C2()->BoundaryPolP())
     return;

  double phi = w->Length() * ( par->D[c] ) *
  ( w->C2()->Chemical(c) - w->C1()->Chemical(c) );
  dchem_c1[c] += phi;
  dchem_c2[c] -= phi;
}
```

Here ‘`w`’ indicates a cell wall separating the two cells, `w->C1()` and `w->C2()`. The cell wall’s length is given by `w->Length()`.

Recompile your model and restart *VirtualLeaf*.

15. Choose suitable diffusion parameters. In the parameter dialog, change the first four values for ‘D’ to: `0`, `0.002`, `0.018`, `0.02`. (see Note 8). Save the template to *virtualleaf/data/leaves/mymodel.xml*.

16. Set `v=1` in one of the cells to initiate the venation pattern. To do so, hover the mouse pointer over the cell whose contents you want to change to display its index number

and contents. Open the leaf template file e.g. *virtualleaf data/leaves/mymodel.xml* in a text editor and search for the line starting with `<cell index="#">` where # is the number of the cell you want to change. Near the end of this cell's definition is a `<chem>` tag containing four `<val>` tags. Change the first `<val>` tag's value to `1.0`, i.e.,

```
<chem n="4">
 <val v="1.0">
 ...
 ...
 ...
 </chem>
```

A more flexible alternative to editing the leaf template file requires a change to the *VirtualLeaf* source code (see Note 9).

Save the file and reopen it in *VirtualLeaf* .

To run the model, press 's' (see Note 10). You should now see a result similar the one shown in Figure 3.

17. It is interesting to study the behavior of this model in a growing domain. First define

an empty template of a couple of cells - it is easiest to grow it from one cell. To start with an initial single cell again, undo step 8 by inserting two forward slashes "//" (*see* Note 4) before the definition of the `QString DefaultLeafML`'in file *mymodel.h*. Recompile your model and restart *VirtualLeaf*. Define appropriate parameters and initial conditions by repeating steps 6 and 12 or you will receive an error "step size too small in odeint" because of a division by zero. A quick way to define these values is by reading only the parameters from a previous template. Choose 'Read leaf' from the 'File' menu and in the file dialog uncheck 'geometry', then proceed as usual.

18. Choose 'Cell growth' from the 'Options menu' to switch on cell growth. Start the simulation until a template of around 8 cells has grown (*see* Note 11). Repeat step 16 in order to define an initial venature cell and save your growing leaf template.

19. Run the model with the new template. You should now see the pattern develop as the leaf grows out. It may be useful to increase the simulated time per growth cycle for the reaction-diffusion equations. To do so, increase the parameter `rd_dt`.

20. *VirtualLeaf* is particularly suited for modeling mechanisms in which growth and pattern formation feed back on one another. We will implement the effects of chemical concentrations on growth in the `MyModel::CellHouskeeping` method. For example, to prevent vascular cells from expanding, wrap the statement that controls cell expansion:

```
c->EnlargeTargetArea(par->cell_expansion_rate);
```

within a conditional statement like this:

```
if (c->Chemical(0)<0.5) {
  c->EnlargeTargetArea(par->cell_expansion_rate);
}
```

21. You now have seen all functionality in *VirtualLeaf* necessary for implementing reaction-diffusion hypotheses of plant patterning and morphogenesis. You should now be able to experiment with modifications of existing hypotheses or to implement new reaction-diffusion models. As a suggestion, implement the assumption that the substrate ‘S’, i.e. ‘Chemical #3’, inhibits cell expansion. Another suggestion is to implement a reaction-diffusion hypothesis for trichome patterning ***(11, 12)***. The next section shows how to add polar auxin transport to your models.

3.5. Polar auxin transport

The previous Section implemented a reaction-diffusion hypothesis for leaf venation patterning. Although reaction-diffusion mechanisms are thought to be involved in a range

of plant patterning mechanisms, e.g. trichome patterning ***(11, 12)***. Many recent hypotheses of plant organ patterning assume a role for directed transport of auxin. This section will demonstrate how to implement directed transport mechanisms, starting from the *traveling-wave* hypothesis for leaf venation patterning ***(13).***

The traveling-wave hypothesis is a variant of the *auxin upstream pumping* hypothesis ***(14, 15)***. It assumes a membrane bound matrix protein, PIN1, exports the phytohormone *auxin* towards adjacent cells. The PIN1 recycles between the membrane and an intracellular storage, called the *endosome*, and binds preferentially to cell membranes adjacent to cells with a high concentration of auxin.

1. Start with an empty model template. Define the number of chemicals we are using in this model. We will need equations for auxin and for PIN1. Therefore, in *mymodel.cpp*, redefine `MyModel::NChem` as:

```
int MyModel::NChem(void) { return 2; }
```

2. Next we will implement the auxin upstream pumping hypothesis: PIN1 transports auxin actively to adjacent cells; a diffusion term is responsible for downstream auxin transport,

$$\frac{dA_i}{dt} = T_{active}\left(\frac{P_{ji}A_j}{k_a + A_j} - \frac{P_{ij}A_i}{k_a + A_i}\right) + T_{diffusive}\sum_j L_{ij}\left(A_j - A_i\right),$$

where the sum is over all neighbor cells, A_i is the auxin concentration in cell *i*. P_{ij} and P_{ji} are the amounts of PIN1 in cell *i* pumping auxin into cell *j* and *vice versa,* and T_{active} and $T_{diffusive}$ are active and passive transport coefficients, and L_{ij} is the length of the wall between cell *i* and cell *j*.

We will store the concentrations of auxin as 'Chemical #0' and the concentration of PIN as 'Chemical #1.' Transporter proteins and other components that localize within the membranes or within the cell wall matrix, are stored in the `Wall` class. Insert the following code into the `MyModel::CellToCellTransport` method (see Note 12):

```
void MyModel::CelltoCellTransport(Wall *w, double *dchem_c1,
                                   double *dchem_c2) {
   for (int c=0;c<NChem();c++) {

     // diffusive transport
     double phi = w->Length() * ( par->D[c] ) *
               (w->C2()->Chemical(c) - w->C1()->Chemical(c));
     dchem_c1[c] += phi;
     dchem_c2[c] -= phi;
   }
   // active transport
```

```
    // efflux from cell 1 to cell 2
    double trans12 = ( par->transport * w->Transporters1(1) *
    w->C1()->Chemical(0) / (par->ka + w->C1()->Chemical(0)));
    // efflux from cell 2 to cell 1
    double trans21 = ( par->transport * w->Transporters2(1) *
    w->C2()->Chemical(0) / (par->ka + w->C2()->Chemical(0)) );
    dchem_c1[0] += trans21 - trans12;
    dchem_c2[0] += trans12 – trans21;
}
```

3. Use suitable cell coloring rules, e.g., those defined in Section 3.4 step 10. Replace the definition for the 'blue' channel by:

```
double blue=0;
```

(*see* Note 13).

4. To test the implementation, recompile your model and restart *VirtualLeaf*. Read the leaf *tutorial3_init.xml* from the *virtualleafdata/leaves* directory and run the model by pressing 's'. This initial condition contains predefined auxin and oriented PINs.

5. Next, implement the PIN1 recycling equations. We define the flux ϕ_{ij} as the

translocation of PIN1s from the endosome of cell *i* to its cell membrane adjacent to cell j (for details, *see* **Ref.** *13*):

$$\phi_{ij} = k_1 \sum_j \frac{P_i A_j f(A_j)}{k_m + P_i} - k_2 \sum_j P_{ij} \text{, with } f(A_j) = \frac{A_j R}{k_R + A_j}.$$

Define a new function *PINflux* to calculate ϕ_{ij}. To do so, add the following line of code to the file *mymodel.h* right before the closing curly-brace:

```
virtual double PINflux(CellBase *this_cell,
   CellBase *adjacent_cell, Wall *w);
```

To implement the function, add the following code to the end of *mymodel.cpp*:

```
double MyModel::PINflux(CellBase *this_cell,
   CellBase   *adjacent_cell, Wall *w) {


   // calculate PIN translocation rate from cell to membrane

   double adj_auxin =  adjacent_cell->Chemical(0);
   double receptor_level = adj_auxin * par->r / (par->kr +
   adj_auxin);
   double pin_atwall; // pick the correct side of the Wall
   if (w->C1() == this_cell) pin_atwall = w->Transporters1(1);
   else pin_atwall=w->Transporters2(1);
```

```
   double pin_flux = par->k1 * this_cell->Chemical(1) *
      receptor_level / ( par->km + this_cell->Chemical(1) ) -
      par->k2 * pin_atwall;
   return pin_flux;
}
```

6. Next implement the following differential equations.

$$\frac{dP_i}{dt} = -\sum_j \phi_{ij} + \alpha A_i - \delta P_i \, , \; \frac{dP_{ij}}{dt} = \phi_{ij} \, .$$

The first equation sums all the net PIN1-fluxes from the membrane to the endosome, and takes it as the change per time unit of the level of PIN1 in the cell. The second equation states that the change in PIN1-level in a cell wall is the flux of PIN1 moving to it.

```
void MyModel::WallDynamics(Wall *w, double *dw1, double *dw2){
   // add biochemical networks for reactions occurring at
   // walls here
   dw1[0] = 0.; dw2[0] = 0.; // chemical 0 unused in walls
   dw1[1] = PINflux(w->C1(),w->C2(),w);
   dw2[1] = PINflux(w->C2(),w->C1(),w);
```

```
}
```

Similarly, in the method `MyModel::CellDynamics` we specify what comes back from the walls.

```
#include "flux_function.h"


void MyModel::CellDynamics(CellBase *c, double *dchem) {

// add biochemical networks for intracellular reactions here

// sum all incoming fluxes of PINs

dchem[1] = -SumFluxFromWalls( c, MyModel::PINflux )+

        //  Auxin-dependent production of PINs

        par->pin_prod * c->Chemical(0) -


        // Breakdown of PIN

        par->pin_breakdown * c->Chemical(1);

}
```

7. Merks et al. ***(13)*** assume that auxin enters the leaf primordium at its margin with a constant flux, and that all PIN1 is produced in response to auxin stimulation. To implement this assumption, we will need to add the auxin sources to a suitable initial condition.

8. Define the initial condition. Start *VirtualLeaf*, open the new traveling wave model and open the file *tutorial4_init.xml*. Remove all auxin and PIN1 values from the leaf template by changing the first two values of 'initval' to zero; click 'Write' on the parameters dialog and choose 'Reset Chemicals and Transporters' from the 'Edit' menu.

9. Next make several of the peripheral walls auxin sources. Make sure 'Show Transporters' is checked in the 'View' menu. Then, while holding down the *Control* key, click the *outside* of some peripheral walls of the template until they are purple. This indicates that they have turned into auxin sources (see Note 14). Save the template. Alternatively, download a suitable template from http://code.google.com/p/virtualleaf/wiki/Protocols. The template used for the simulations in **Ref.** *13* can be downloaded from this same site.

10. To make the auxin sources work, add the following code to end of the `MyModel::CelltoCellTransport` method in the file *mymodel.cpp*:

```
// Influx at leaf "AuxinSource"
// (as specified in initial condition)
 if (w->AuxinSource()) { // test if wall is auxin source
          double aux_flux = par->leaf_tip_source
          * w->Length();
          dchem_c1[0] += aux_flux;
          dchem_c2[0] += aux_flux;
 }
```

11. Open the parameter dialog to set the parameters used by **Ref.** *13*: `D[0]=1e-6` ($T_{diffusive}$; see Note 15); `leaf_tip_source=1e-5` (ϕ_{tip}; *see* Note 15); `transport=0.08` (T_{active}); `pin_prod =1e-5` (α); `pin_breakdown=1e-8` (δ); `r=100` (R); `kr=100` (k_R); `k1=2e-4` (k_1); `k2=5e-7` (k_2); `km=100` (k_m).

12. Start the simulation. If the calculations seem to progress slowly, increase the value of `rd_dt`. This increases the integrated time between two subsequent plotting steps or cell growth steps. A suitable value for this model would be `rd_dt=1000`.

13. You can now start experimenting with the parameters, e.g., the changes suggested in the Supplements of **Ref.** *13.* Interesting effects occur by speeding up the constitutive translocation of PIN1 (`k1=0.2`; `k2=0.005`), speeding up the production and breakdown of PIN1 (`pin_prod=0.001`; `pin_breakdown=0.001`) and by increasing the diffusion of auxin (`D=5e-06`). The result should look similar to Figure 4. Choose a large leaf template.

4. Notes

1. A discussion on differential-equation descriptions of biological networks is out of the scope of this chapter. Useful texts on the subject can be found in **Ref.** *16* and **Ref.** *17*.

2. You can skip this step by using the model template 'Tutorial0' or by downloading the template "MyModel" from http://code.google.com/p/virtualleaf/wiki/Protocols.

3. Optionally, use parameter '`par->rel_cell_div_threshold`' instead of '`2`':

```
if (c->Area() > par->rel_cell_div_threshold * c->BaseArea()){
```

```
    c->Divide();
}
```

4. In C++, you can switch off code you want to keep for later re-use, by making it a comment that the compiler will ignore. To do so, place the code between "`/*`" and "`*/`", or insert two slashes ("`//`") at the beginning of the line. In the following two line code snippet, the compiler will interpret the first statement as a comment and perform only the second statement `c->Divide()`.

```
/* c->DivideOverAxis(axis) */
c->Divide();
```

5. In future versions the name of this menu item will be changed to 'Randomize Chemicals and Transporters'.

6. Only one of these function definitions is allowed per model. So if a definition called `DefaultLeafML` is already present, do not add a second one.

7. `SetRgbF(r,g,b)` is a Qt library function. It takes red, green and blue color values between 0 (minimal) and 1 (maximal). All other Qt color library functions can be used as well. See *http://qt.nokia.com* for documentation.

8. *'D'* is under 'Auxin Transport and PIN1 dynamics'. Remember to scroll to the start of the text box using the left arrow keys).

9. As an alternative to editing the leaf XML template, you can also redefine function

`Cell::OnClick` in file `VirtualLeaf.cpp` to change cell contents interactively, i.e., to reset the value of 'Chemical #0' after you have clicked a cell, redefine `Cell::OnClick` as follows:

```
void Cell::OnClick(QMouseEvent *e){
   if (NChem()>0)  SetChemical(0,1.);
}
```

To effect this change, you will need to recompile the *VirtualLeaf* framework (see Step 2.3). Future versions of *VirtualLeaf* will move this functionality to the model definition files.

10. If *VirtualLeaf* becomes unresponsive, decrease the value of parameter `rd_dt`. If the model progresses too slowly, increase the value of `rd_dt`. `rd_dt` is the integration time per display step.

11. If *VirtualLeaf* aborts with the message 'stepsize underflow in rkqs, with `h=0,000000` and `htry=0,10000`' you are probably dividing by zero. In this example, `H=0` 'Chemical #2'. (The 'h' mentioned in the error message is an unrelated integration parameter). Use an appropriate initial condition with 'Chemical #2' set to small positive value, as explained in step 17.

12. In this code, the symbols in the equations are represented as follows. L_{ij} : `w->Length()`; P_{ij} and P_{ji}: `w->Transporters1(1)` and `w->Transporters2(1)`; diffusion coefficient (D) : `par->D[c]`; transport coefficient (T) : `par-`

`>transport` ; k_a : `par->ka`.

13. If you used the exact code defined in Section 3.4, step 10, you would attempt to read Chemical #3. Here Chemical #3 is undefined, and as a result *VirtualLeaf* may crash. In fact, reading from or writing to undefined memory locations is a common cause of software crashes.

14. In the release of *VirtualLeaf* published with our *Plant Physiology* paper ***(7)***, this task of clicking on peripheral walls is quite tedious. It helps to increase the width of the cell walls, by increasing the value of parameter `outlinewidth`. Also it helps to hide the cells by checking 'Hide cells' item in the 'View' menu. If you accidentally click the interior walls, it turns red to indicate that you have injected it with a high concentration of PIN. If this happens, simply choose 'Reset Chemicals and Transporters' from the 'Edit' menu. In more recent version of *VirtualLeaf* this issue will have been solved.

15. To calculate D and leaf_tip_source from the values of $T_{diffusive} = 1.5 \times 10^5 m^{-2} s^{-1}$ and $\phi_{tip} = 1.5 \times 10^6 m^{-2} s^{-1}$ mentioned in **Ref.** *(13)*, note that we average length of a cell wall in the template used is around 15 a.u. (arbitrary units). Assuming a cell wall has an area of around $L_{ij} = 100 \mu m^2 = 10^{-10} m^2$, we rescale $D = T_{diffusive} \times \frac{L_{ij}}{15} = 10^{-6}$ and $leaf_tip_source = \phi_{tip} \times \frac{L_{ij}}{15} = 10^{-5}$.

References:

1. Dupuy L, Mackenzie J, Rudge T, Haseloff J (2008) A system for modelling cell-cell interactions during plant morphogenesis. Ann Bot-London 101:1255-1265

2. Grieneisen VA and Scheres B (2009) Back to the future: evolution of computational models in plant morphogenesis. Curr Opin Plant Biol 12:606-614

3. Chickarmane V, Roeder AH, Tarr PT et al (2010) Computational morphodynamics: a modeling framework to understand plant growth. Annu Rev Plant Biol 61:65-87

4. Santos F, Teale W, Fleck C et al (2010) Modelling polar auxin transport in developmental patterning. Plant Biol 12 Suppl 1:3-14

5. Keurentjes JJ, Angenent GC, Dicke M et al (2011) Redefining plant systems biology: from cell to ecosystem. Trends Plant Sci 16:183-190

6. Kitano H (2002) Systems Biology: A Brief Overview. Science 295: 1662-1664

7. Merks RMH, Guravage M, Inzé D, Beemster GTS (2011) VirtualLeaf: An Open-Source Framework for Cell-Based Modeling of Plant Tissue Growth and Development. Plant Physiol 155:656-666

8. Merks RMH, Glazier JA (2005) A cell-centered approach to developmental biology. Physica A 352:113-130

9. Anderson ARA, Chaplain MAJ, Rejniak KA, eds (2007) Single-Cell-Based

Models in Biology and Medicine, Birkhaüser, Basel

10. Meinhardt H (1976) Morphogenesis of lines and nets. Differentiation 6:117-123

11. Benítez M, Espinosa-Soto C, Padilla-Longoria P, Díaz J, Alvarez-Buylla ER (2007) Equivalent genetic regulatory networks in different contexts recover contrasting spatial cell patterns that resemble those in Arabidopsis root and leaf epidermis: a dynamic model. Int J Dev Biol 51:139-155

12. Bouyer D, Geier F, Kragler F, Schnittger A, Pesch M, Wester K, Balkunde R, Timmer J, Fleck C, Hülskamp M (2008) Two-dimensional patterning by a trapping/depletion mechanism: The role of TTG1 and GL3 in Arabidopsis trichome formation. PLoS Biol 6:1166-1177

13. Merks RMH, Van de Peer Y, Inzé D, Beemster GTS (2007) Canalization without flux sensors: a traveling-wave hypothesis. Trends Plant Sci 12:384-390

14. Jönsson H, Heisler MG, Shapiro BE, Meyerowitz EM, Mjolsness E (2006) An auxin-driven polarized transport model for phyllotaxis. P Natl Acad Sci USA 103:1633-1638

15. Smith RS, Guyomarc'h S, Mandel T, Reinhardt D, Kuhlemeier C, Prusinkiewicz P (2006) A plausible model of phyllotaxis. P Natl Acad Sci USA 103:1301-1306

16. Ellner SP, Guckenheimer J (2006) Dynamic Models in Biology. Princeton University Press, Princeton

17. Fall CP, Wagner JM, Marland ES, Tyson JJ, eds (2002) Computational Cell Biology. Series Interdisciplinary Applied Mathematics, Volume 20. Springer, New York

Acknowledgements

This work was financed by the Netherlands Consortium for Systems Biology (NCSB), which is part of the Netherlands Genomics Initiative / Netherlands Organisation for Scientific Research, and by Marie Curie European Reintegration Grant PERG03-GA-2008-230974 to RM.

Figure captions:

1. Simulation result of the tissue growth model implemented in Section 3.3, steps 1-5. In this model, cells expand at a constant rate and divide over their short axis after doubling in size.
2. Simulation result of the tissue growth model implement in Section 3.3, steps 1-8. In

this model, the division axis has a random orientation, resulting in an irregular overall tissue shape.

3. Simulation result of the reaction-diffusion model of leaf venation by Meinhardt ***(10)***, as implemented in Section 3.4. The moving activator-inhibitor front (red) traces out the vein shown in green. It requires a substrate (blue), that the veins consume.
4. Simulation result of auxin travelling-wave model on a static domain ***(13)***. The auxin concentration is shown in green, the concentration of PIN in the cells and at the walls is shown in red. The white arrows indicate the polarization directions of PINs. See **Ref.** *13* for details.

```
#include "vleafmodel.h"


QString MyModel::ModelID(void) {

     // specify the name of your model here

     return QString( "My Own Model in VirtualLeaf" );

}


// return the number of chemicals your model needs

int MyModel::NChem(void) { return 0; }



// Rules in this method are executed after the cellular mechanics has

// equilibrized

void MyModel::CellHouseKeeping(CellBase *c) {

          // add cell behavioral rules here

}


// To be executed after cell division

void MyModel::OnDivide(ParentInfo *parent_info, CellBase *daughter1, CellBase *daughter2)
{

          // rules to be executed after cell division go here

         // (e.g. cell differentiation)

}


// Differential equations describing chemical reactions inside the cells

void MyModel::CellDynamics(CellBase *c, double *dchem) {

     // add biochemical networks for intracellular reactions here
```

Box 1: Model definition outline. New models are constructed by defining the functions in this model definition file.

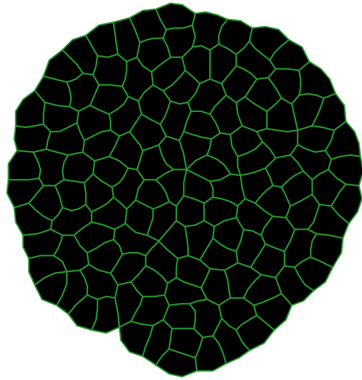

Figure 1.

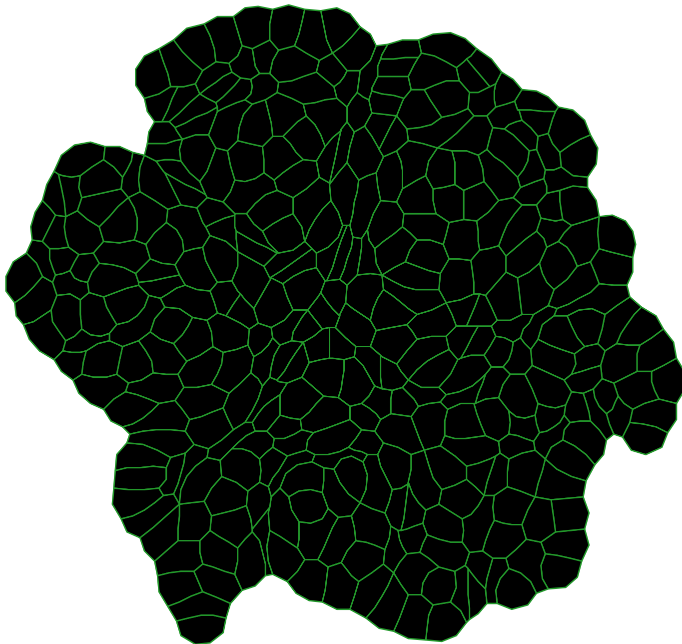

Figure 2.

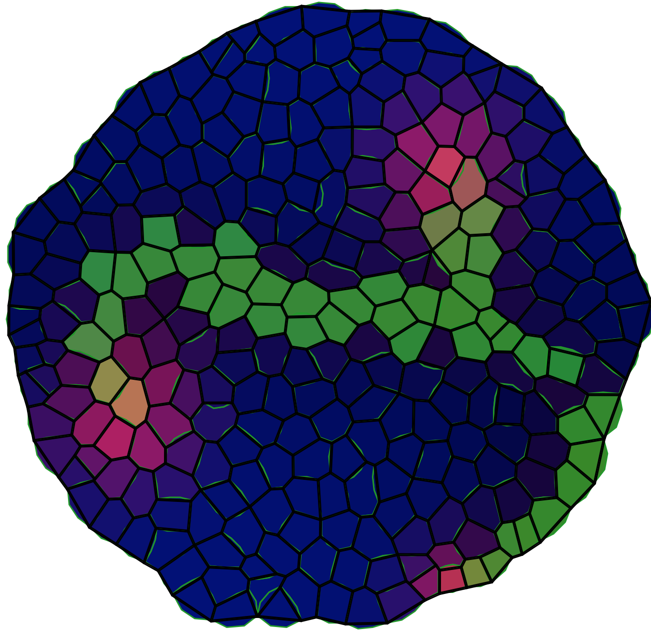

Figure 3.

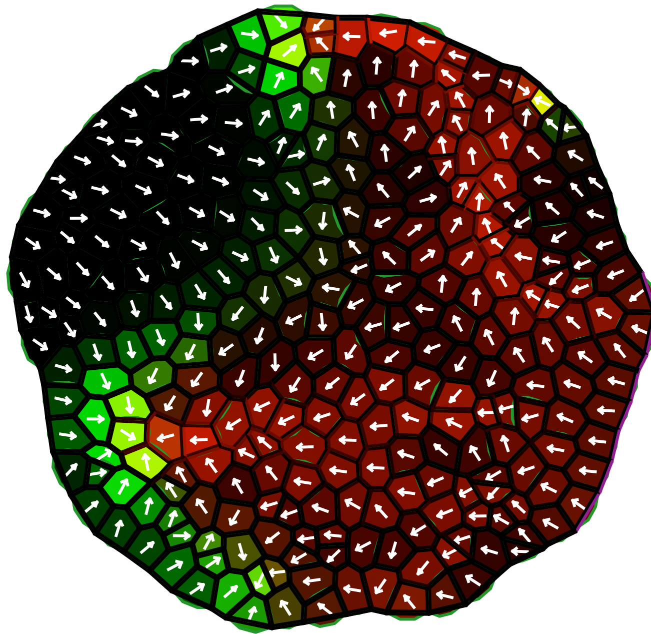

Figure 4.