vlab

The Virtual Laboratory

Getting Started with *vlab*

Last updated: November 30, 2021

vlab was developed in the labs of Przemyslaw Prusinkiewicz at the University of Regina and the University of Calgary, Canada

Contents

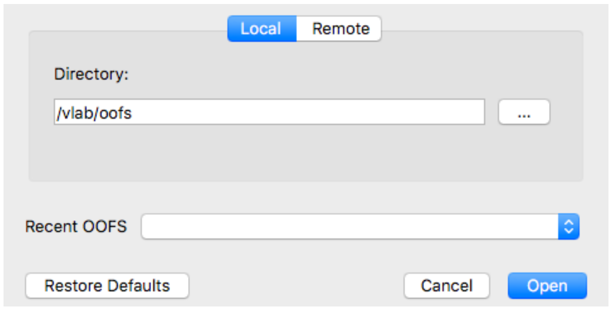


Figure 1: The *browser* dialog box to indicate the location of the *oofs* folder.

1 Introduction

This manual is a guide to setting up *vlab*, and using the existing tools that it offers. There is also a description of a series of examples in both *cpfg* and *lpfg*, highlighting some of the basic features of modeling using L-systems.

All examples in this manual have been run on a Mac iOS, and the installation instructions are also for this platform. However, the software can also be configured for a Linux system and the file structure (*oofs*) containing the example objects will work there as well.

We hope you enjoy using *vlab* as much as we have. Good luck!

1.1 Installing *vlab*

The *vlab* distribution consists of the following:

- browser.app - The main application.
- oofs - The object oriented file structure, containing sample objects.
- raserver.app - The remote access server application for use with an *oofs* structure on another machine.
- README.mac.txt - Installation instructions
- Gifts - A folder containing versions of the *vim* editor with syntax highlighting specific to *cpfg* and *lpfg*. A Readme file is included with instruction on how to install.

Review the README.mac.txt file and install the *browser* application. The instructions indicate how to add it to the Applications folder. It can also be installed in a personal directory, if that is more convenient. The *raserver* application should be installed in the same location, if being used.

The *oofs* folder should be in a separate location, however, and should only be installed once - i.e. if there is an updated version of *vlab* do not re-install the *oofs* folder, since that would overwrite any objects you had created and/or experimented with earlier. When the *browser* is opened, it will display a dialog box to ask for the location of your *oofs* directory (Figure 1).

See the **VLAB Framework** manual for information on installing the *oofs* structure on a remote server.

1.2 Running *vlab*

Since *browser* is not an "identified" App Store application, you may receive an error message when attempting to run it. To override this security, right-click on the application, and select Open from the

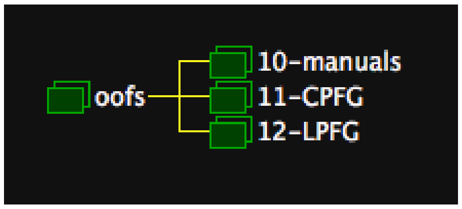


Figure 2: The first view of the distribution version of the *oofs* data structure in the browser.

menu. A warning message will pop-up: click Open to run the browser. This override is only required once; the application can be run afterwards simply by double-clicking on it.

An alternative is to run the browser from a terminal window. The advantage is that error messages from the browser, as well as from other *vlab* tools, will be displayed in the window, which can be helpful for debugging. The command to run the browser would be:

```
/Applications/browser.app/Contents/MacOS/browser
```

Terminal application icon

assuming the browser has been installed in the Applications folder.

Once the browser is started, and the location of the *oofs* directory has been specified, a graphical view of *oofs* will be displayed (Figure 2).

1.3 Setting up *oofs*

The *oofs* distribution contains all the objects referenced in the *vlab* manuals, as well as a range of sample objects for *cpfg* and *lpfg*. Double-click on the name, 10-manuals, to see the list of manuals, and again on GettingStarted to find the first object included in this manual. (The *cpfg* and *lpfg* sections reference objects in 11-CPFG and 12-LPFG respectively.)

To see a reference picture (*icon*) for an object, right-click on its name. Alternatively, if there are several objects:

1. Click on parent to select it.
2. From the View menu, select Show all icons.

This feature can be turned off again with the Hide all icons menu item.

1.3.1 Create a project

In order to keep the distribution objects intact, create your own project and copy some of the objects to your project. To create the project:

1. Click on oofs to select it.
2. From the Object menu, select New object.
3. In the dialog box, enter a name for your project (e.g. Sandbox), and click OK.

The project name (Sandbox) will appear as an extension of oofs. Note that names should not include any spaces.

Now, copy the object, 01-lychnis, from GettingStarted to Sandbox:

1. Click on the object, 01-lychnis, to select it.
2. From the Object menu, select Copy object.

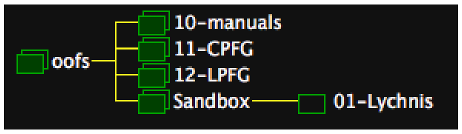


Figure 3: The object, 01-lychnis, copied to the Sandbox project.

3. Click on your project, Sandbox, to select it.
4. From the Object menu, select Paste.

The object will be displayed under Sandbox (Figure 3).

1.4 More information

Refer to the **VLAB Framework** manual for more detailed information. Specifically,

Section 2: The *oofs* structure
Section 3: The Browser
Section 4: The Object Manager (discussed below)

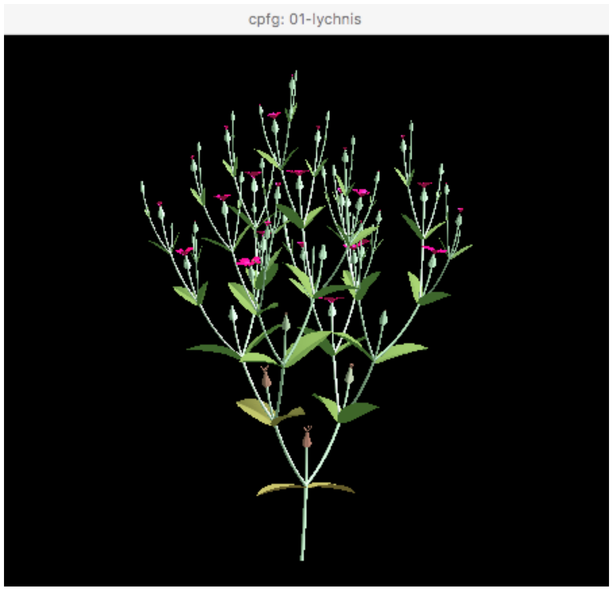


Figure 4: The *cpfg* model of lychnis, run from the 01-lychnis object.

2 Experimenting with an object

See object:
01-Lychnis

To experiment with an object, double-click on the folder next to its name. This will open a new window for the *Object Manager*, displaying the object's icon. It also copies the object to the *lab table*: a temporary location for experimenting so that the original object remains intact.

Read a description of the object by right-clicking on the object icon, and selecting description > edit from the menu.

2.1 Running the simulation

To run a simulation, right-click on the object icon, and select image > generate from the menu. This will display a new window for the simulation program (Figure 4).

This is a 3D model of a lychnis plant that can be seen from different viewpoints:

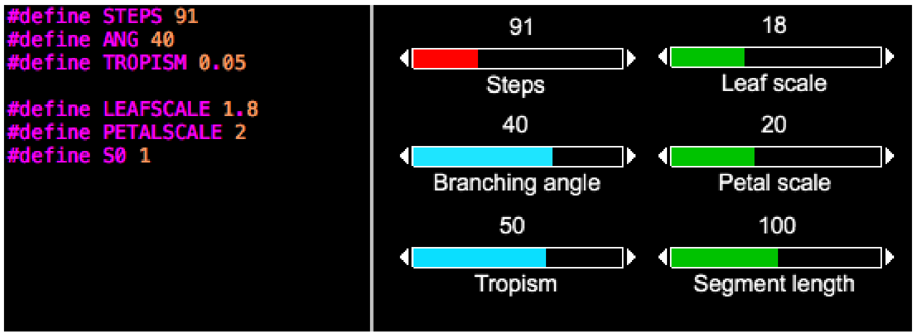
- To rotate the plant: click and drag the mouse.
- To zoom in and out: hold the Command key down, and click and drag the mouse.
- To move the plant (pan): hold the Shift key down, and click and drag the mouse.

To animate the growth of the plant:

1. Right-click on the simulation window, and select Animate from the menu.
2. Right-click again, and select Run.

To return to the beginning of the animation, select Rewind from the menu. You can also use the Step menu item to walk through the simulation a step at a time.

The *vlab* distribution includes two simulation programs: *cpfg* and *lpfg*. Both programs require an L-system model, a *view* file containing parameters for viewing it, and (optionally) an *animate* file containing animation parameters. The content of these files is specific to each simulation program. The lychnis model is written in *cpfg*. To see the details of the model, right-click on the object icon and select L-system, view file, or animate file, respectively, and then the edit option.



```
#define STEPS 91
#define ANG 40
#define TROPISM 0.05

#define LEAFSCALE 1.8
#define PETALSCALE 2
#define S0 1
```

91 Steps
18 Leaf scale
40 Branching angle
20 Petal scale
50 Tropism
100 Segment length

Figure 5: Some of the `#define` statements in the L-system for the lychnis model (left), and part of the panel for changing them (right). Note that slider values must be integers, which are scaled before updating the `#define` statements.

2.2 Changing parameters

Experimenting with an object involves changing the parameters of the underlying files. The lychnis object has several parameters defined at the start of the *cpfg* program. It is possible to edit the modelling program directly to change the parameters. However, *vlab* also provides a *panel* tool to interactively manipulate them (Figure 5). To open the panel, right-click on the object icon and select L-system > panel from the menu.

Try adjusting a slider by clicking and dragging within the white outline. The plant in the *cpfg* window changes accordingly. You can get an intuitive understanding of the parameters by playing with them using the sliders. Remember that you are working on the *lab table*: none of your changes affect the stored version of the object.

To reset the panel to its original values, right-click on a blank space within the panel, and select Reset from the menu.

2.3 Refresh mode

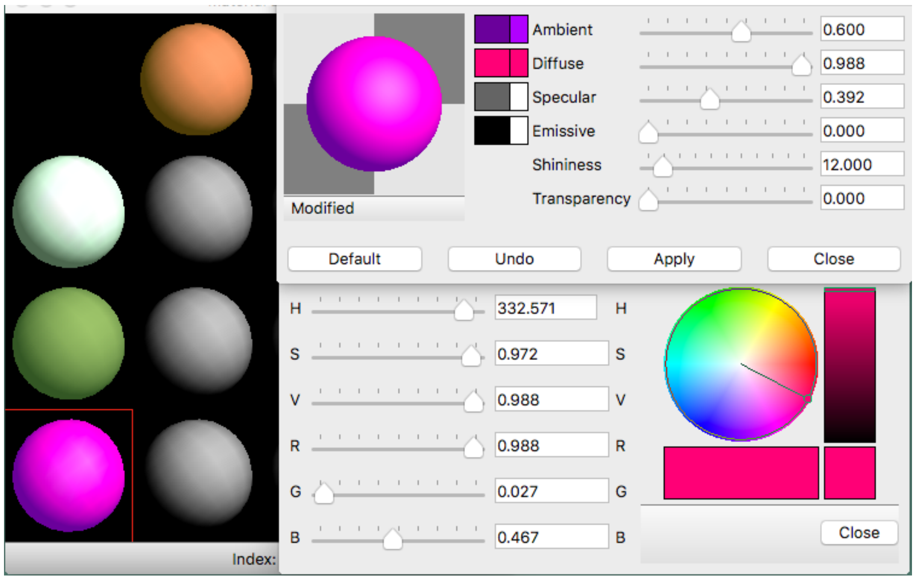
The panel is acting on the modeling program, a text file that is read by *cpfg* to create the simulation. In general, the communication between *vlab* tools is through files, with one tool writing to a file and another reading the results from the file. The tools have the ability to monitor the files, which is how changes in the panel above are immediately seen in the model: *cpfg* is monitoring the modeling program file and, therefore, when *panel* makes a change to the file it is immediately picked up and a new view of the model is displayed.

For more control over when a file is written to/read from, the *refresh mode* can be changed to:

- Explicit: Explicitly save or read the file
- Triggered: Write only when finished an action, such as when the mouse is released on a slider (i.e. only the final value of the slider will be written to the file).

Try changing the refresh mode on the panel: right-click and select Refresh mode > Triggered from the menu. Then click and drag on a slider, and notice how the plant does not change until the mouse is released.

Then change the refresh mode in the *cpfg* window: right-click and select Refresh mode > Explicit from the menu. Notice that changes made on the panel no longer affect the plant in the *cpfg* window. They are being written to the file, however, since the panel is still in Triggered mode. Re-read the modeling program by right-clicking and selecting New model from the menu: all the changes made to the sliders will be incorporated into the model at once.


Ambient 0.600
Diffuse 0.988
Specular 0.392
Emissive 0.000
Shininess 12.000
Transparency 0.000

H 332.571
S 0.972
V 0.988
R 0.988
G 0.027
B 0.467

Figure 6: The components of the material editor: the main window showing each of the colors (left), the details of the selected color (top right), and the color wheel for changing one of the four components (bottom right).

Finally, change the refresh mode in the *cpfg* window back to Continuous, so that you will be able to see the results of further manipulations immediately.

2.4 Manipulating colors

There are two tools within *vlab* for defining and manipulating colors: *colormap* and *medit*. The lychnis model uses the latter. For more information on the *colormap* tool, see the **VLAB Tools** manual.

To manipulate the colors in the lychnis model, select materials > edit from the object manager menu. This will display the colors used in the model (Figure 6, left). The first color, index 0, defines the background. The remaining colors are defined within the modeling program. To change a color:

1. Double click on the colored ball.
2. Click on the rectangle beside Diffuse, to display a color wheel.
3. Click and drag on the color wheel to change the color.
4. Then click on the rectangle beside Ambient, and change the color with the color wheel.
5. Close the main window when done.

Try changing the color of the lychnis flowers using the color wheel. If the refresh mode in the *cpfg* window is set to Continuous, you should see the flowers change colors as you manipulate the color wheel. Zoom in for a closer look.

2.5 Function parameters

The parameters in the `#define` statements within the L-system have a single value. However, in some cases, there are parameters that should change over the course of the simulation (as a plant ages, for example). These parameters are defined as *functions* of one variable. In the case of the lychnis model, there are several functions that are grouped into a *gallery*.

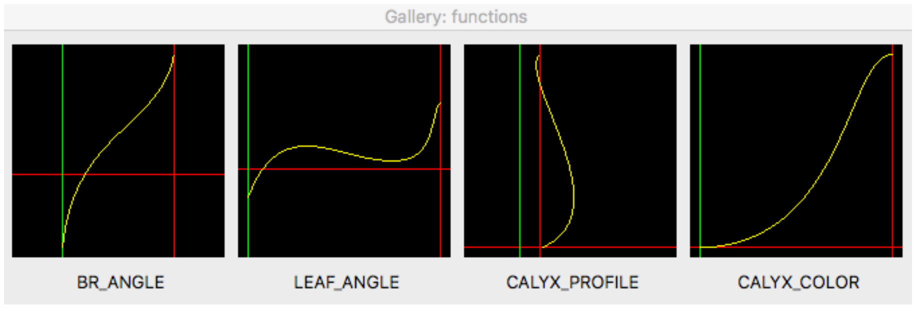


Figure 7: Part of the gallery of functions used by the lychnis model.

To experiment with a function in the gallery:

1. Right click on the object icon, and select functions > gallery to display the gallery (Figure 7).
2. Double click on the function's icon (e.g. LEAF_ANGLE) to open the function editor.
3. Click and drag on the control points to adjust the function.
4. Close the function editor and gallery windows when done.

Note that since the refresh mode for the gallery is Continuous, there is no need to save changes: the underlying file is updated as changes are made – which is why you can see the changes reflected immediately in the *cpfg* window.

2.6 Manipulating shapes

Shapes can be defined using two different tools: a *contour* editor that defines the outer edge of an object; and a *surface* editor that describes the shape of the entire object. In the lychnis model, the calyx (the organ below the flower) is described using a contour, whereas the petals and leaves are defined as complete surfaces that are scaled depending on their age. These editors are similar to the function editor described above: they all manipulate control points to change the shape.

2.6.1 Contours

If a model has several contours, they may be defined in a *gallery*, like the functions above. In the case of the lychnis model, where there is only one contour, the contour editor is called directly:

1. Right click on the object icon, and select contours > calyx to display the contour editor (Figure 8).
2. Click and drag on the control points to adjust the contour.
3. Close the contour editor when done.

Since contour editor is in Continuous mode, the underlying file is updated as changes are made and the changes reflected immediately in the *cpfg* window. Zoom in to get a closer look at the calyx of a flower.

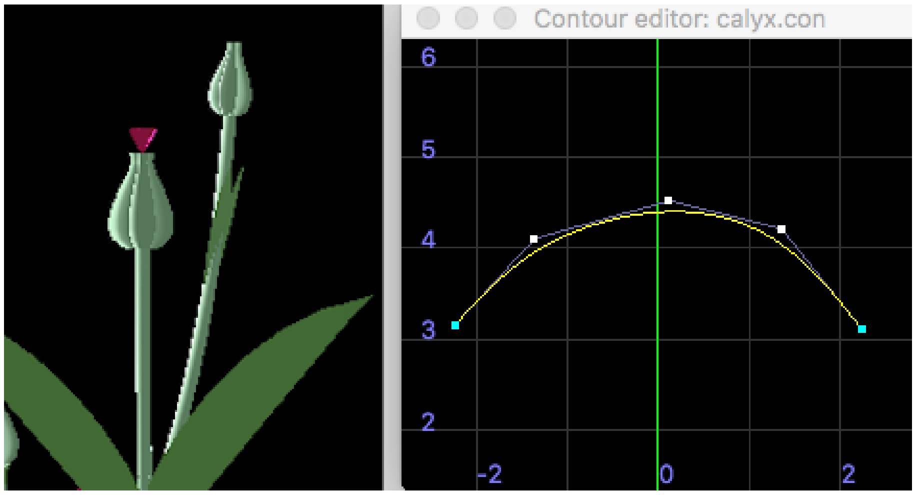


Figure 8: The lychnis calyx (left) and the contour editor (right) for adjusting it. Note that the contour is not describing the vertical outline of the calyx, but the horizontal outline of each segment of the calyx.

2.6.2 Surfaces

A surface is a 3D representation of an element of the model. In the case of the lychnis model, the elements are the petals and leaves. There is one surface for each element, which is sized within the model depending on its age (using the functions, LEAF_GROWTH and PETAL_SIZE, respectively).

To adjust the shape of a surface:

1. Right-click on the object icon, and select surfaces > leaf > bezieredit to display the surface editor (Figure 9).
2. Rotate the image by holding down the Shift key and using the mouse.
3. Zoom in and out by holding down the Command key and using the mouse.
4. Click and drag on the control points to adjust the surface.
5. Click the Save button (on the panel to the left) to save your changes and update the model.
6. Close the surface editor when done.

The *bezieredit* tool does not have a Continuous mode, so changes must be explicitly saved. However, once they are saved to the surface file, the changes are received immediately by *cpfg* if it is in Continuous mode.

There are actually two surface editors in *vlab*: *bezieredit* used in this model for both the leaf and the petal; and *stedit* that includes functionality for superimposing a texture on the surface. They both use the same underlying surface file. See the **VLAB Tools** manual for detailed information on each of these tools.

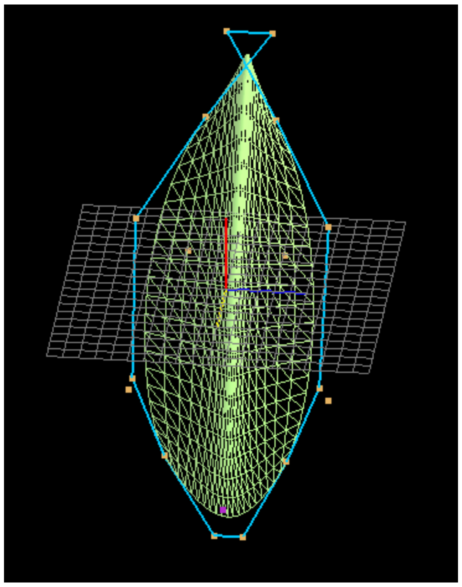


Figure 9: The lychnis leaf as seen in the surface editor, *bezieredit*.

2.7 Completing the experiment

The object manager also provides a set of standard utilities for maintaining objects. In this section we will only look at the ones needed to complete an experiment. See the **VLAB Framework** manual for details on all the utilities.

2.7.1 Saving changes

Once an experiment is finished, the object should be removed from the *lab table*, the temporary location used only for experimentation. There are three options:

- Close the object without saving any of the changes you have made. The stored version of the object will retain its original data, as it was presented when you began the experiment.
- Overwrite the stored version of the object with the changes you have made. Next time you open the object, it will contain the changes.
- Create a new version of the object with the changes you have made, leaving the stored version intact. The browser will display the new version of the object as a child of the original object.

The first two options are straightforward: simply select Quit from the object's menu. A dialog box will be displayed listing the files that have been modified. Click the Ignore and Quit button to close without saving your changes, or click the Save and Quit button to overwrite the stored version of the object with your changes. This can be done any time during experimenting.)

To create a new version of the object, select Utilities > New version from the object's menu. Enter a name for the new object, and click the OK button. Look in the browser window to see the new object (Figure 10.

Changes to an object can also be saved while experimenting to ensure they are not lost. To explicitly overwrite the stored version of the object with your changes, click on the object's menu and select Utilities > Save changes. To create a new version while you are working, use Utilities > New version and click the Point to new position checkbox after providing a name for the new object. This will update the Object Manager to point to this new object as you continue experimenting.

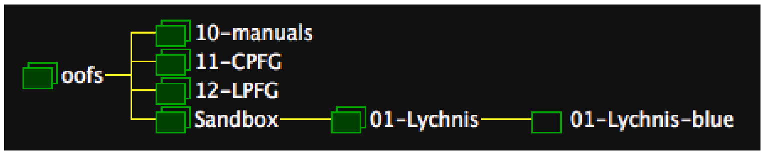


Figure 10: The browser window showing another lychnis object created using Utilities > New version. Note that the parent object now has a double file folder next to it to indicate that it has child objects.

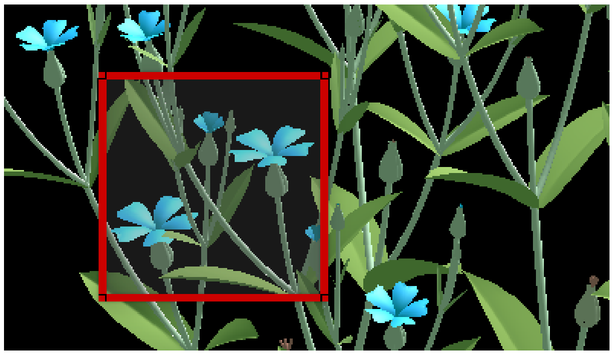


Figure 11: A portion of the simulation with the *snapicon* window (outlined in red) superimposed.

2.7.2 New icon

To distinguish the new object from the original one, it should have a different icon. This is especially relevant when both objects are open on the lab table. To create a new icon:

1. Open the object (double click on the file folder beside it in the browser window).
2. Run the simulation (image > generate).
3. Zoom in and rotate the image to find its best side.
4. From the Utilities menu, select Icon > Snap.
5. Click and drag on the *snapicon* window (outlined in red) to place it on a portion of the image (Figure 11).
6. Right-click on the same window, and select Snap, save and exit from the menu.
7. Right-click on the object icon, and select Utilities > Icon > Reread to update the icon.

2.8 Inheritance

A new version of an object "inherits" data from its parent object. Only the files that have actually changed are stored with the child object. This means that a subsequent experiment on the parent object may affect a child object as well: if a data file is changed in the parent object that was not changed in the child (and, therefore, inherited), you will see the change in the child object. Experiment with both the parent and child version of the lychnis model to understand how inheritance works.

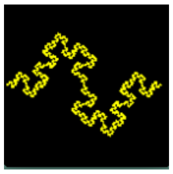
3 Modeling in *cpfg*

The *oofs* distribution includes a series of *cpfg* sample objects in the folder 11-CPFG. This section introduces some of these examples, to provide an understanding of:

- The basic components of a *cpfg* model: the L-system file and view file.
- The main programming constructs in the *cpfg* modeling language: the axiom, productions, decomposition rules, and interpretation rules.
- Geometric (turtle) interpretation of L-systems.
- Biological notions, such as modules, apices, internodes, and metamers.

For more information on programming in *cpfg*, see the **CPFG Reference Manual**.

3.1 Basic components of a *cpfg* model


01-quadratic-Koch

We begin in the 01-Introduction section with the quadratic Koch curve, a simple fractal popularized by Benoît Mandelbrot [1]. Open the object, and select L-system > edit from the menu to see the *cpfg* program:

```
Lsystem: 1
derivation length: 4
Axiom: -F
F --> F+F-F-FF+F+F-F
endlsystem
```

The five lines of this program show the overall format of a *cpfg* program:

- The mandatory opening, `Lsystem:`. The number is arbitrary.
- The number of *derivation steps*.
- The *axiom*.
- The *production(s)*.
- The mandatory closing, `endlsystem`

The axiom is the initial L-system string. Each derivation step produces a new string based on the production rules. In the example above, each instance of `F` (the *predecessor*) is replaced with `F+F-F-FF+F+F-F` (the *successor*). Therefore, the following string is output after 0, 1, and 2 derivation steps (spacing has been added for clarity).

```
-F

- F+F-F-FF+F+F-F

- F+F-F-FF+F+F-F + F+F-F-FF+F+F-F - F+F-F-FF+F+F-F - F+F-F-FF+F+F-F F+F-F-FF+F+F-F
+ F+F-F-FF+F+F-F + F+F-F-FF+F+F-F - F+F-F-FF+F+F-F
```

The image produced by *cpfg* is the result of the *turtle interpretation* of the symbols in the string. The symbols in this example are interpreted by *cpfg* as:

Command	Description
`-`	Turn right
`+`	Turn left
`F`	Move forward one step, drawing a line.

The parameters used in the interpretation of these commands are defined in the *view file*. Select View > edit from the object menu to display the content of the file:

```
angle increment: 90
initial line width: 1 pixels
initial color: 1
scale factor: 0.9000
```

In particular, the `angle increment` defines how far the turtle should turn to the right and left.

3.2 Colormap

The `initial color` in the view file above is an index into the colormap. To see the colormap, select Colormap > palette from the object menu. To change a color, click on it and then use the sliders above to adjust the red, green, and blue components of the color. *cpfg* always uses index 0 in the colormap or materials file (Section 2.4) as the background color.

3.3 Modifying a model

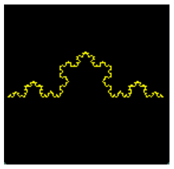
In general, it is convenient to develop new models by modifying existing ones. So let's take the quadratic Koch curve and create another classic fractal, the snowflake curve (Figure 12).

With each derivation step, every line `F` should be replaced by Figure 13. There are four lines in this building block (therefore, four `F` symbols), which turn left, then right, and then left again. This equates to:

```
F --> F+F-F+F
```

But what angle should be used? The first left turn is 60°, the right turn is 120°, and the next left turn is again 60°. So we can set the `angle increment` in the view file to `60`, and then modify the production to turn right 60° twice by adding another minus sign:

```
F --> F+F--F+F
```


03-snowflake

Try making these modifications to the quadratic Koch curve L-system and view files, and generate the image. You should see the snowflake in Figure 12 turned 60°. This is because the default starting position for the turtle is the bottom of the window, pointing upwards, and the axiom, `-F`, has turned the turtle 60° as specified by the change made to the `angle increment` in the view file – not the original 90° to horizontal.

Therefore, the angle in the axiom must be adjusted. One option would be to change the `angle increment` to 30°, and add more turns before each line is drawn. Another option is to use a parameter, as demonstrated in the next section.

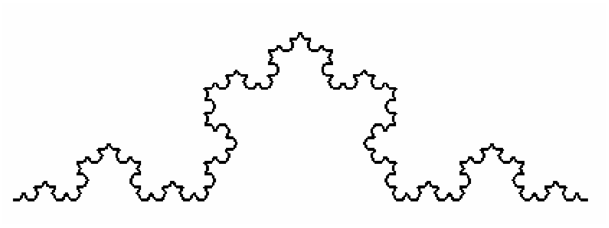


Figure 12: The snowflake curve.

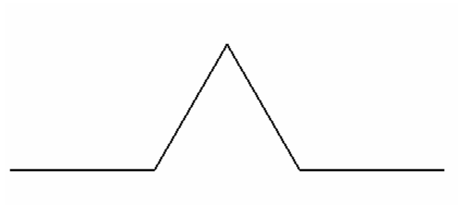


Figure 13: The building block of the snowflake curve.

3.4 Modules with parameters

Many symbols within *cpfg* have parameters, with default values used if the parameter(s) are not present. All three of the symbols we have used so far can have a parameter. A symbol with parameter(s) is called a *module*.

To turn the snowflake curve horizontal, use the right turn symbol in the axiom with its parameter: an angle in degrees:

```
Axiom: -(90)F
```

This will override the default `angle increment` value in the view file. The remaining turn symbols (in the production) that do not have an associated parameter will continue to use the default value.

We could also specify the angle of every turn symbol in the production:

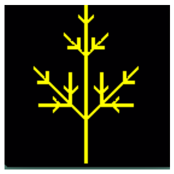
```
F --> F+(60)F-(120)F+(60)F
```

The value of `angle increment` in the view file would then be irrelevant. However, it can make the production less readable.

3.5 Branching structures

Now let's look at a simple branching structure. Open the object 04-compound, and look at the L-system:

```
Lsystem: 1
derivation length: 4
Axiom: A
A --> F[+A][-A]FA
F --> FF
endlsystem
```



04-compound

The axiom defines an *apex*, `A`, indicating the tip of a branch. The first production replaces an apex with the following components:

- a line, F
- a branch to the left consisting of an apex, [+A]
- a branch to the right consisting of an apex, [-A]
- another line, F
- a new apex, A

The second production, replaces each line, F, with two lines, thus lengthening the existing line. Note the new square bracket symbols, [], that begin and end a branch.

The symbol A is *user-defined*. It does not have a predefined turtle interpretation and, therefore, is not visualized.

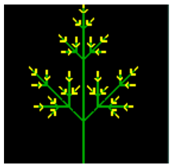
3.6 Interpretation rules - Homomorphisms

We can assign a turtle interpretation to a symbol using *interpretation rules*. These rules have the same format as a production, but they do not alter the string – they are only used by the turtle to visualize the string. They are placed after all of the productions, using the keyword homomorphism. For example, we can add an interpretation rule for the apex after the last production, such that it also draws a line:

```
...
F --> FF
homomorphism
A --> F
endlsystem
```

Interpretation rules make it possible to write L-system more clearly by using symbols defined for specific modules of a plant. For example, this branching structure can be described in terms of apices (A) and internodes (I), with interpretation rules used to define how each of these modules is visualized.

```
Lsystem: 1
derivation length: 4
Axiom: A
A --> I[+A][-A]IA
I --> II
homomorphism
A --> F
I --> F
endlsystem
```


05-compound-homo

3.6.1 Changing colors

To indicate the difference between an apex and an internode, let's change the color. In the view file, the initial color index is 1. To use color index 2 for the apex, we add the semicolon symbol (;) before the F symbol which increments the color index. Thus the interpretation rule for an apex is:

```
A --> ;F
```

3.7 Parametric productions

In the above example, the internode is elongated by duplicating the symbol (`I --> II`). Another approach is to use parametric productions. In our model, we can add a parameter to symbol `I`, starting with 1:

```
A --> I(1)[+A][-A]I(1)A
```

To increase the value, we use a *formal parameter*, `x`, in the next production:

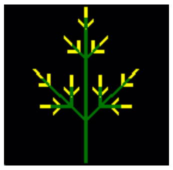
```
I(x) --> I(2*x)
```

and also a formal parameter in the homomorphism to indicate the length of the line to be drawn:

```
I(x) --> F(x)
```

The full L-system is now:

```
Lsystem: 1
derivation length: 4
Axiom: A
A --> I(1)[+A][-A]I(1)A
I(x) --> I(2*x)
homomorphism
A --> F
I(x) --> F(x)
endlsystem
```


06-compound-param

With each derivation step, the productions set *actual parameter* values in the string. For example, the L-system above produces the following strings for the first 3 derivation steps, with actual parameter values for `I`:

```
I(1)[+A][-A]I(1)A


I(2) [+ I(1)[+A][-A]I(1)A ] [- I(1)[+A][-A]I(1)A ] I(2) I(1)[+A][-A]I(1)A


I(4) [+ I(2)
            [+ I(1)[+A][-A]I(1)A ] [- I(1)[+A][-A]I(1)A ] I(2) I(1)[+A][-A]I(1)A ] ]
     [- I(2)
            [+ I(1)[+A][-A]I(1)A ] [- I(1)[+A][-A]I(1)A ] I(2) I(1)[+A][-A]I(1)A ]  ]
     I(4) I(2) [+ I(1)[+A][-A]I(1)A ] [- I(1)[+A][-A]I(1)A ] I(2) I(1)[+A][-A]I(1)A
```

3.8 Conditional productions

Parameters are a powerful construct, and can be used for a variety of modeling purposes. The following example introduces three new concepts:

1. `#define` statements to specify constants.
2. Conditional statements.
3. Branching delay.

07-fractal-ferns

Open the object 07-fractal-ferns, and view the L-system:

```
#define STEPS 18
#define D 3
#define R 1.28

Lsystem: 1
derivation length: STEPS
Axiom: A(0)
A(t) : t<D --> A(t+1)
A(t) : t==D --> I(1)[+A(0)][-A(0)]I(1)A(t)
I(x) --> I(R*x)

homomorphism
A(t) --> ;F(t)
I(x) --> F(x)
endlsystem
```

3.8.1 `#define` statements

C-type `#define` statements are used to set constants in the program. These statements make it easy to change numeric values without looking for them within the code, and also allows them to be changed by a panel such as the one in Figure 5 (as described in Section 2.2). Constants can be used anywhere a value is required: e.g. to specify the derivation length (`STEPS`), in conditional statements (`D`), or as parameters in modules (`R`).

3.8.2 Conditional statements

In the L-system above, there are now two productions for the apex, each with a *conditional statement*. In addition to matching the symbol in the string with the predecessor `A(t)`, the condition must also evaluate to true before replacing the symbol with the successor. The general format of this type of production is:

predecessor `:` *condition* `-->` *successor*

The colon (`:`) is mandatory to separate the conditional statement from the predecessor.

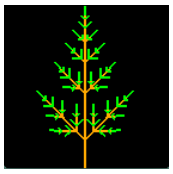
3.8.3 Branching delay

A parameter has been added to the apex symbol (`A`) that is used as a time counter. The first production increments the time (`t`) by 1. The apex is replaced by internodes and branches (production 2) only when `t` reaches the delay factor `D`. Therefore, the L-system will produce the following strings in the first 5 derivation steps:

```
A(1)

A(2)

A(3)

I(1)[+A(0)][-A(0)]I(1)A(3)

I(1.28) [+A(1)] [-A(1)] I(1.28) I(1)[+A(0)][-A(0)]I(1)A(3)
```

Note that the larger the delay, the more derivation steps are needed. Select Animate from the *cpfg* window, and then Run to see the effect of the delay.

3.9 Continuous models


08-continuous-time

In the previous models, each derivation step corresponded to a *plastochron*: the length of time between the creation of new lateral branches. In contrast, the following model operates in *real time*, advancing by a small interval, `dt`:

```
#define STEPS 400
#define dt 0.02

#define D 2
#define R 1.5

Lsystem: 1
derivation length: STEPS
Axiom: A(0)

A(t) : { t1=t+dt; } t1<D --> A(t1)
A(t) : { t1=t+dt; } t1 >= D { t2=t1-D; t3=t1-1; } -->
     I(0.5*R^t2)[+A(t2)][-A(t2)]I(0.5*R^t2)A(t3)
I(x) --> I(x*R^dt)

homomorphism

A(t) --> ;F(t)
I(x) --> F(x)

endlsystem
```

Note that the number of derivation steps has also been increased significantly since each step is very small.

3.9.1 Statement blocks

In the above L-system, productions have been extended again to include statement blocks before and after the conditional statement. The general form is:

predecessor : { *before-stmts* } *condition* { *after-stmts* } --> *successor*

where *before-stmts* are evaluated before the condition, and *after-stmts* are only evaluated if the condition is true.

The statements in each block use C syntax, including a semicolon (;) after each statement. The {} brackets are required, even if there is only one statement. Each of the components (*before-stmts, condition*, and *after-stmts*) are optional. However, if one or both statement blocks is present, the condition must also be present to distinguish them. The condition can be 1 or *, if needed. For example:

```
A(t) :  1 { t=t+dt; } --> A(t)
```

New variables within statement blocks are *local*; they are undefined outside of the production.

3.10 Metamers

As a model becomes more complex, more levels of abstraction can make it easier to read. Here, we introduce the concept of a *metamer*, a plant unit that consists of an internode and its auxilliary branches. In the original branching structure model (Section 3.5) where the productions were:

```
A --> F[+A][-A]FA
F --> FF
```

the metamer would be F[+A][-A]F; the final A is the new apex. Similarly, in our current model, the metamer is the successor in the second production:

```
A(t) : { t1=t+dt; } t1 >= D { t2=t1-D; t3=t1-1; } -->
     I(0.5*R^t2)[+A(t2)][-A(t2)]I(0.5*R^t2)A(t3)
```

excluding the A(t3) at the end of the production.

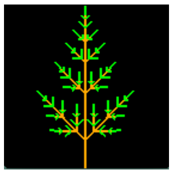
All the modules in the metamer use the same local variable t2. Therefore, we can replace the modules with a single metamer module M(t2), which will then have its own production rule. The two productions would be:

```
A(t) : { t1=t+dt; } t1 >= D { t2=t1-D; t3=t1-1; } --> M(t2)A(t3)
M(t) --> I(0.5*R^t)[+A(t)][-A(t)]I(0.5*R^t)
```

We can further simplify the metamer production by putting the rate calculation in a statement block:

```
M(t) : 1 { x=0.5*R^t; } --> I(x)[+A(t)][-A(t)]I(x)
```

3.11 Decomposition


09-continuous-decomp

Another method of clarifying a complex model is to use *decomposition rules*. Like production rules, these rules affect the modules in the output string, but are applied after the productions. Therefore, they provide a means of separating the main algorithm from the details.

Decomposition rules are specified after productions but before the homomorphism statement, and begin with a decomposition statement. For example, our model can be written as:

```
#define STEPS 400
#define dt 0.02

#define D 2
#define R 1.5
```

```
#define RATE (R^dt)

Lsystem: 1
derivation length: STEPS

Axiom: A(0)

A(t) --> A(t+dt)
I(x) --> I(x*RATE)

decomposition
maximum depth: 2

A(t) : t >= D --> M(t-D) A(t-1)
M(t) : 1 { x=0.5*R^t; }  --> I(x)[+A(t)][-A(t)]I(x)

homomorphism

A(t) --> ;F(t)
I(x) --> F(x)

endlsystem
```

Decomposition rules can be evaluated recursively; the `maximum depth` statement safeguards against infinite recursion. However, the statement is optional and the default, when no `maximum depth` statement is present, is 1 (i.e. no recursion).

3.11.1 Macros

The L-system above also includes a macro: a calculation within a `#define` statement:

```
#define RATE (R*dt)
```

It must be possible to fully evaluate the macro when it is defined. Therefore the values used within it, `R` and `dt`, must be defined before the macro itself.

3.12 Recursion

10-recursive

At the limit, the entire structure can be obtained by recursive application of decomposition and/or interpretation rules. Below is our model specified using recursive interpretation (homomorphism) rules:

```
#define T 8

#define D 2
#define R 1.5

Lsystem: 1
derivation length: 1
Axiom: A(T)

homomorphism
maximum depth: 100
```

```
A(t) : t >= D --> M(t-D) A(t-1)

M(t) : 1 { x=0.5*R^t; } --> I(x)[+A(t)][-A(t)]I(x)

A(t) : t < D --> ;F(t)
I(x) --> F(x)

endlsystem
```

Note that the **derivation length** is 1, since all the rules are recursively applied to the axiom until done. This means the model cannot be animated, making this a *structural model* only.

Models can be constructed this way only if they do not require context-sensitive productions.[1]

3.13 Alternating branches

The L-system in Section 3.11 has an *opposite* branching pattern. To modify it to produce *alternating* branches, we will need two apex types, `A` and `B`, that produce two different metamers, `M` and `N`, where metamer `M` issues a branch to the left, and metamer `N` issues a branch to the right. The terminal apex will alternate between `A` and `B`.

Each of the `A(t)` rules in the L-system should have a similar `B(t)` rule:

```
A(t) --> A(t+dt)
B(t) --> B(t+dt)
...
decomposition
A(t) : t >= D --> M(t-D)B(t-1)
B(t) : t >= D --> N(t-D)A(t-1)
...
homomorphism
A(t) --> ;F(t)
B(t) --> ;;F(t)
```

Notice the alternating terminal apex: `A` produces an `M` metamer and a `B` apex; and `B` produces an `N` metamer and an `A` apex.

There must also be an `N` metamer rule that produces an apex to the right. And the original `M` metamer should produce an apex to the left only:

```
M(t) : 1 { x=0.5*R^t; } --> I(x)[+A(t)]I(x)
N(t) : 1 { x=0.5*R^t; } --> I(x)[-B(t)]I(x)
```

12-alternating

Try modifying the object 09-continuous-decomp using the rules above to obtain an alternating branching pattern.

[1]See the **CPFG Reference manual** for information on context sensitive productions, and oofs > 11-CPFG > 04-Signaling for examples.

3.14 Further examples

Explore the remaining *cpfg* examples in the *oofs* distribution on your own. Read the description of each model, run the simulation, and play. You will find examples of:

- 3D models, functions, and surfaces (02-Mod-vis-3D)
- Functions and contours (03-Inverse-modeling)
- Context-sensitivity (04-Signaling)
- Information flow (05-Quantitative-info-flow)
- Macros and query modules (06-Enviro-sensitive)
- The use of environmental programs (07-Open)

4 Modeling in *lpfg*

The *cpfg* language is very concise but can become cryptic as the complexity of the model increases. At that point it is generally preferable to switch to *lpfg*, which is more verbose and, therefore, less elegant for short L-systems, but is easier to read and has more functionality when creating complex models [2]. It uses the L+C modeling language, a hybrid of L-systems and constructs from the C programming language.

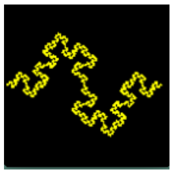
In this section, however, we will revisit the same concepts as the previous section, highlighting the differences between *lpfg* and *cpfg*. The *lpfg* example objects can be found in the *oofs* distribution under 12-LPFG > 01-Introduction.

For more information on programming in *lpfg*, see the **LPFG Reference Manual**.

4.1 Basic differences between *lpfg* and *cpfg*

The simple quadratic Koch curve model in *cpfg* (Section 3.1) was:

```
Lsystem: 1
derivation length: 4
Axiom: -F
F --> F+F-F-FF+F+F-F
endlsystem
```


01-quadratic-Koch

Comparing this with the same model in *lpfg*:

```
#include <lpfgall.h>

#define ANGLE 90

derivation length: 4;
Axiom: Right(90) F(1.0);

F(v) :
{
     produce F(v) Left(ANGLE) F(v) Right(ANGLE) F(v) Right(ANGLE) F(v)
             F(v) Left(ANGLE) F(v) Left(ANGLE)  F(v) Right(ANGLE) F(v);
}
```

we can see the following:

- The mandatory opening has changed, from `Lsystem:` to `#include <lpfgall.h>`.
- Both can have `#define` statements.
- The *lpfg* language is based on the C programming language and, as such, statements always end with a semicolon (`;`), including the `derivation length` and `axiom` statements.
- The format of the production is different. Whereas the syntax of *cpfg* is:

  predecessor `-->` *successor*

  the syntax of the *lpfg* production is:

  predecessor `:` `{` `produce` *successor* `;` `}`

- Rather than using symbols such as + and -, *lpfg* uses module names such as `Left()` and `Right()`, and there is a space separating each module.
- Modules in *lpfg* have a fixed number of parameters. Therefore, `F` alone is not sufficient – it cannot default for length 1. Similarly, the `Left` and `Right` modules cannot use a default angle specified in the view file.

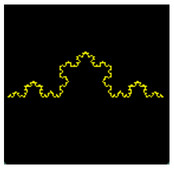
The process, however, is the same: a new string is output after each derivation step, replacing each instance of `F(v)` with its successor. And the image results from the *turtle interpretation* of the modules in the string.

Given what we have learned about *lpfg*, what is required to change the quadratic Koch curve into the snowflake curve? Since the *lpfg* modules used above must have parameters, we can revisit the *cpfg* snowflake model that also uses parameters (Section 3.4), where the axiom and production were:

```
Axiom: -(90)F
F --> F+(60)F-(120)F+(60)F
```

In *lpfg*, these two lines equate to:

```
Axiom: Right(90) F(1.0);
F(v) :
{
     produce F(v) Left(60) F(v) Right(120) F(v) Left(60) F(v);
}
```


03-snowflake

Try making these modifications to the quadratic Koch curve L-system, and generate the image. Note that the `ANGLE` value is no longer needed.

4.2 User-defined modules

As in the C programming language, each user-defined module must be declared before it is used in order to define the number of parameters and their types.

For example, to create a simple branching structure consisting of internodes `I(x)`, and apices `A(v)`, we must declare these two user-defined modules at the top of the L-system:

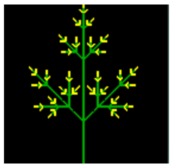
```
module I(float);
module A(float);
```

We can then use the modules to create a simple branching structure:

```
Axiom: A(1.0);

A(v) :
{
     produce I(v)
             SB() Left(ANGLE) A(v) EB()
             SB() Right(ANGLE) A(v) EB()
             I(v) A(v) ;
}
I(v) : { produce I(v*2) ; }
```

The predefined modules `SB()` and `EB()` replace the square brackets used in *cpfg* to indicate the start and end of a branch, respectively.

4.2.1 Interpretation rules


05-compound-interp

As in *cpfg*, there must be rules for the turtle interpretation of the user-defined modules. (If no rule is found, the module is ignored during interpretation.) These rules are placed after the production, using the heading `interpretation:`. For example, the two modules above could have interpretation rules:

```
interpretation:
A(v) : { produce IncColor() F(v); }
I(v) : { produce F(v); }
```

where the predefined module `IncColor()` replaces the semicolon (`;`) in *cpfg* to move to the next color in the colormap or materials file.

4.3 Conditional statements

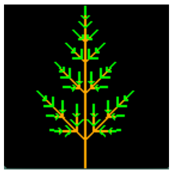
07-fractal-ferns

Conditions are specified directly within productions, using the `if-then-else` construct. For example, to create the branching delay for a fractal fern a single production can produce both the delay and the new metamer:

```
A(v) :
{
     if (v<D)
          produce A(v+1);
     else if (v==D)
          produce I(1)
                  SB() Right(ANGLE) A(0) EB()
                  SB() Left(ANGLE)  A(0) EB()
                  I(1) A(v);
}
```

It is also possible to create loops using the C constructs `for` and `while`.

4.4 Local variables


08-continuous-time

As is the case for modules, local variables must be declared. This is done within the production body. For example, to create the continuous fern using a time increment of `dt`, the production would be:

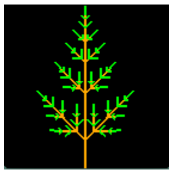
```
A(t) :
{
     float t1 = t+dt;

     if(t1<D)
          produce A(t1);
     else {
          float t2 = t1-D;
          float t3 = t1-1.0;

          produce I(0.5*pow(R, t2))
                  SB() Right(ANGLE) A(t2) EB()
                  SB() Left(ANGLE) A(t2) EB()
                  I(0.5*pow(R,t2)) A(t3);
     }
}
```

The above code mimics the before and after statement blocks in the *cpfg* program in Section 3.9.1. Another option is to declare the variables all at once, and then assign them values are needed. For example:

```
A(t) :
{
     float t1, t2, t3;

     t1 = t+dt;
     if(t1<D)
          produce A(t1);
     else {
          t2 = t1-D;
          t3 = t1-1.0;
          produce I(0.5*pow(R, t2))
                  SB() Right(ANGLE) A(t2) EB()
                  SB() Left(ANGLE) A(t2) EB()
                  I(0.5*pow(R,t2)) A(t3);
     }
}
```

A production can have any number of C statements throughout.

4.5 Decomposition


09-continuous-decomp

As in *cpfg*, decomposition rules can be placed between productions and interpretation rules to separate the basic components of the model from some of the details. For example, the model above can be rewritten as:

```
A(t): { produce A(t+dt); }
I(x): { produce I(x*RATE); }

decomposition:
maximum depth: 2;

A(t):
{
     if (t >= D) produce M(t-D) A(t-1);
}

M(t):
{
     float x = 0.5*pow(R,t);

     produce I(x)
             SB() Right(ANGLE) A(t) EB()
             SB() Left(ANGLE) A(t) EB()
             I(x);
}

interpretation:

A(t) : { produce IncColor() F(t); }
I(x) : { produce F(x); }
```

This example also includes the concept of a metamer (Section 3.10), to provide another level of abstraction. A fully recursive version of the model can be found in the object 10-recursive.

4.6 Further examples

Explore the remaining *lpfg* examples in the *oofs* distribution on your own. Read the description of each model, run the simulation, and play. You will find examples of:

- 3D models, functions, and surfaces (02-Mod-vis-3D)
- Context-sensitivity (04-Signaling)
- Information flow (05-Quantitative-info-flow)
- The use of environmental programs (07-Open)

Compare these models to their *cpfg* equivalents.

5 Credits

All models were developed by Przemyslaw Prusinkiewicz. See the individual reference manuals for more information on each application:

- The *vlab* Framework includes the browser and object manager
- *vlab* Tools includes control panel, color, function, contour, and surface tools
- CPFG Reference Manual
- LPFG Reference Manual
- *vlab* Environmental Programs

6 Document revision history

Date	Description	By
2021	First version. Incorporates *cpfg* teaching notes by Przemyslaw Prusinkiewicz.	Lynn Mercer Przemyslaw Prusinkiewicz

References

[1] Benoît Mandelbrot. *The fractal geometry of nature*. W.H. Freeman and Co., 1982.

[2] Przemyslaw Prusinkiewicz. Art and science for life: Designing and growing virtual plants with L-systems. *Acta Horticulturae*, 630:15–28, 2002.