# What is Ingeneue?

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Elements of an Ingeneue Model</td>
<td>4</td>
</tr>
<tr>
<td>About Ingeneue Tutorials</td>
<td>5</td>
</tr>
<tr>
<td>Using Ingeneue</td>
<td>6</td>
</tr>
<tr>
<td>Loading a Network File</td>
<td>6</td>
</tr>
<tr>
<td>Running a Network</td>
<td>6</td>
</tr>
<tr>
<td>Displaying and Hiding Nodes in the Cell View</td>
<td>7</td>
</tr>
<tr>
<td>Changing Initial Concentrations</td>
<td>7</td>
</tr>
<tr>
<td>Viewing and Changing Node Concentrations</td>
<td>8</td>
</tr>
<tr>
<td>Running Models Automatically with an Iterator</td>
<td>8</td>
</tr>
<tr>
<td>Examining Output Files from an Iterator</td>
<td>9</td>
</tr>
<tr>
<td>Network Files</td>
<td>10</td>
</tr>
<tr>
<td>General Information</td>
<td>11</td>
</tr>
<tr>
<td>Defining Nodes</td>
<td>11</td>
</tr>
<tr>
<td>Parameters</td>
<td>13</td>
</tr>
<tr>
<td>Affectors</td>
<td>14</td>
</tr>
<tr>
<td>Specific Parameter Values</td>
<td>16</td>
</tr>
<tr>
<td>Initial Conditions</td>
<td>17</td>
</tr>
<tr>
<td>Iterators</td>
<td>18</td>
</tr>
<tr>
<td>An Overview of an Iterator file</td>
<td>18</td>
</tr>
<tr>
<td>Restricting Parameter Values Relative to Each Other</td>
<td>19</td>
</tr>
<tr>
<td>Searching Multiple Times using the UberIterator</td>
<td>19</td>
</tr>
<tr>
<td>Stoppers</td>
<td>23</td>
</tr>
<tr>
<td>Combining Stoppers</td>
<td>23</td>
</tr>
<tr>
<td>Interface</td>
<td>25</td>
</tr>
<tr>
<td>Interface for Constructing and Running Models</td>
<td>25</td>
</tr>
<tr>
<td>Graphing and Calculating Statistics on Ingeneue Output Files</td>
<td>27</td>
</tr>
<tr>
<td>Programming Ingeneue</td>
<td>28</td>
</tr>
<tr>
<td>Making a New Affecter</td>
<td>28</td>
</tr>
<tr>
<td>Making a New Iterator</td>
<td>28</td>
</tr>
</tbody>
</table>
Introduction

The biology community is accumulating information about genes and their interactions at an astonishing rate. At a small scale, such as the function of particular genes and the pathologies that result when those genes are defective, this information has already been put to great use. At the slightly larger scale of a network of genes, however, it is often difficult to assimilate and process all the available information well enough to make complete hypotheses and good predictions. These difficulties are particularly evident for network-level properties such as the ability of a specific network to perform different functions, or the robustness of the network to various perturbations. One way of dealing with complicated biological systems is through computer models in which each component of the system is specified by an equation that represents what will happen to that component over time. One can represent a network of several interacting things as a system of "coupled" differential equations, and then solve the equations to simulate how the network behaves over a period of time, given some initial state. The practice of representing complex biological systems using differential equations has a long and respected history in many fields of biology, but for the most part has not yet been adopted by the genetics and molecular biology communities.

Ingeneue is a computer program built specifically for modeling networks of genes interacting in a field of cells. There is nothing particularly tricky about these types of models from a mathematical point of view, and there is no reason why one couldn't do the same job using standard math modeling software. In practice, though, Ingeneue is much more convenient and efficient than a general-purpose program for making and running genetic models. For example, many of the most interesting genetic networks make patterns in fields of cells, and this means there needs to be a separate implementation of the network's equations for each cell in the field. Ingeneue takes care of making copies of the network and placing a copy in each cell, saving you this additional work. We have designed parts of equations which represent particular types of interactions, such as translation, transcriptional activation, and so on, and have made these in such a way that they can be mixed and matched together. This means that to specify a network, you needn't type in any equations at all - you just assign one of the pre-built formulas to each interaction in the network in a pseudo-english text file. Finally, exploring models usually means running them many times, varying the parameters associated with each interaction or varying the initial patterns. Ingeneue includes a mechanism for running genetic models automatically and varying different things on each run, saving information out about each run for further analysis.

In addition to its utility in making genetic models easy to construct and explore, the Ingeneue interface is geared towards displaying and exploring the dynamics of genetic models in an intuitive way. One window shows the currently-loaded network in a view reminiscent of the arrow diagrams in genetics papers. Another window shows the concentrations of different components of a network in each of a field of cells as the model runs. Other windows let you change various components of the model through text fields, check boxes, and the like, without any need for programming. Finally, for those who wish to do novel things with Ingeneue, we have written the code for the program cleanly enough that adding new pieces (such as new formulas we haven't thought of) is relatively easy.

This manual explains Ingeneue at three different levels. First comes an explanation of how to use Ingeneue with pre-existing network files, for instance to reproduce results from a
paper that used Ingeneue. Next is a description of how to construct and explore your own networks. At the end is a brief description of the parts of Ingeneue's code that you might want to modify in order to expand the program.

Ingeneue is still very much a work in progress and nowhere near a version 1.0. Thus everything is rough around the edges and pieces that should be there are missing or hard to use (that includes this manual). Ingeneue will certainly crash occasionally as you use it, and will be frustrating to use at other times. Nevertheless, we have been successfully using Ingeneue to generate interesting results for over a year. We have also now used it successfully in several courses, including a two-week workshop on evolution and development. With this manual and a bit of patience you ought to be able to use the program to recreate our results and perform similar experiments on your own networks.
Elements of an Ingeneue Model

Ingeneue network models consist of four kinds of object. The two basic elements are called Nodes and Affectors. A Node is a mRNA, protein, dimerized protein, or any other molecular species in the network whose concentration you want to keep track of. Nodes come in two basic varieties: intracellular Nodes are internal to a cell and the concentration is assumed to be uniform across the cell; membrane-bound Nodes sit on or in the membrane, and the concentration may be different on each face of the cell. There are no strictly extracellular Nodes, because there is no model of an extracellular volume for them to inhabit. Instead, it is assumed that everything associates with a cell surface to some degree, and intercellular "diffusion" means transfer from one cell surface to another (more on this below). For elements of the network which have more than one state we make a separate Node for each state. For instance, an unbound receptor might be one Node, and that same receptor protein bound to its ligand another Node. As another example, if one wanted to model regulated import of a transcription factor into the nucleus, one would use two Nodes, one for the cytoplasmic population and one for the nuclear. (Since DNA concentrations don't change over developmental time in most contexts, we don't keep track of DNA explicitly.)

Nodes are strung together into a network by Affectors, which store pieces of equations governing Node interactions (an Affector represents how Nodes "affect" each other). Most Affectors implement an additive term in the differential equation specifying the change in concentration of one of the Nodes over time, usually as a function of the concentration of other Nodes and of a few free parameters. A particular Node can have several Affectors, each of which specifies a different process affecting that Node. For instance, a mRNA Node generally has at least two Affectors, one representing synthesis of the message by regulated transcription and another representing first-order decay of the message. As described later, there are special encapsulating Affectors that enable nesting or other complicated combinations of Affectors into a single term.

The Nodes, with their complement of Affectors, are collected together into a Network object. The Network contains a full representation of the part of the model contained within a single cell. Ingeneue then makes an array of Cell objects, arranges the neighbor relations between them, and puts a copy of the Network into each Cell. Thus each Cell stores its own concentrations of each Node and calculates the changes to its Nodes' concentrations over time using each Node's ability to compute its own differential equation from its list of Affectors. For membrane-bound Nodes, each Cell keeps track of separate concentrations of that Node in each face of the cell (most commonly 6 faces per cell). Since the concentration of some Nodes can depend upon other Nodes in neighboring cells (receptors depending on ligand expressed on the surface of a neighboring cell, for instance), an Affector can have a connection to a Node in a neighboring cell. Ingeneue automatically establishes these connections based on the geometric arrangement of cells.

The process of modeling a gene network in Ingeneue involves specifying the geometry of the Cells, specifying the Nodes, specifying the Affectors connecting together the Nodes, and giving values to each Affector's parameters. This is currently done in a rather unwieldy text file. The next sections of this manual explain how to make one of these text files.
About Ingeneue Tutorials

A growing number of tutorials are available from the Ingeneue website (http://www.ingeneue.org/). These cover both conceptual issues and practical aspects of using Ingeneue. They are designed to be used either as individual tutorials or for a small group, as in a classroom. Most tutorials focus on models we are ourselves already fairly familiar with. Presently available tutorials include an introductory one based on the segment polarity network model described in von Dassow et al. (00), a tutorial that guides one through the process of building a simple network, and one that covers programming a new Affector type.
Using Ingeneue

In our work with Ingeneue we have built files that define and explore several different genetic networks. This section explains how to load and run our pre-built Network files.

**Loading a Network File**

To load in an already written network file, do the following:

1. Select 'Load…' from the 'File' menu inside the main Ingeneue window.
2. From the standard file dialog that appears, select the network file.

If there are any errors in the network file, they will be displayed in the Console window (usually found at the bottom of the screen). Otherwise, you will see a graphical representation of the network defined by the file in the Network View window (the main window of Ingeneue, which changes its title according to a network name embedded in the network file). The Cell View window will appear and show you one or more rows of cells.

**Running a Network**

After loading a network, you will want to run it to see what patterns it generates. Mathematically, running a network means integrating the differential equations defining the network over time. Ingeneue measures time in simulated "minutes". You can either have Ingeneue run as fast as it can until it reaches a fixed stop time, currently set at 1000 minutes, or you can "step" an Ingeneue model for one integration timestep at a time (usually a few minutes of time but this changes as a model runs and depends a great deal on the equations and their parameters).

- To run a model continuously for 1000 minutes, select the 'Run Model' command from the 'Run' menu.
- To run a model for a short time, select 'Step Model' from the 'Run' menu. This runs the model for one integration timestep. Usually this is a fraction of a minute to several minutes, with the exact length depending on how quickly concentrations of the components of the model are changing.
- To stop a model that is running, select the 'Stop' command in the 'Run' menu. You can continue running from the point where you stopped using the 'Step Model' command.
- To reset a model back to time 0, in preparation for another run, select the 'Reset Model' command from the 'Run' menu. 'Run Model' will also automatically first reset the model.

When a network is running, the Cell View window displays what is happening to Node concentrations in each Cell over time. Only selected Nodes are displayed, and for each of these, a clump of hexagonal Cells is shown in the Cell View, with the brightness of the color in each Cell reflecting the concentration of the Node in that Cell. A Cell in which the Node's concentration is 0 (or close to 0) will be drawn black. The second Node being displayed will have its own clump of cells directly below the first one, and so on. To the left of each clump is a label telling you what Node that clump displays. The colors used for each Node are arbitrary and there is no significance to having the same color used for two different
Nodes. The next section tells you how to change which Nodes are displayed and what colors they are drawn with.

In the top left of the Cell View window you'll also see the number of simulated minutes that have passed since the model started running.

**Displaying and Hiding Nodes in the Cell View**

If you want to display another Node in the Cell View (so you can see concentrations of that Node in each Cell as the model runs), do the following:

1. Click on the Node you want displayed in the Network View window (the window showing each Node in the network with lines connecting the Nodes).

This will bring up the Node Inspector window, which lets you change various characteristics of each Node.

2. Check the 'Show' checkbox in the Node Inspector window.

As soon as you check the 'Show' checkbox, should see a new clump of cells for that Node appear in the Cell View window.

If you want to hide a Node, follow the same steps but uncheck the Show checkbox.

If you want to change the color used to draw a Node, do the following:

1. Click on the Node whose color you want to change in the Network View window.

2. Click on the 'Color' button in the Node Inspector window.

3. In the standard color selector that appears, select the new color you want to use and click OK.

4. Click on the Set button in the Node Inspector window.

**Changing Initial Concentrations**

At the beginning of each run, every Node in a model is given an initial concentration in each Cell. You can think of this initial concentration as the pre-pattern from which this network starts its own patterning process. For most networks, the initial concentrations will determine what the final pattern looks like, and sensitivity to initial concentrations is one important measure of the robustness of a particular patterning mechanism. You can change the initial concentration of a Node in one particular Cell, or in all Cells. To change initial concentrations, do the following:

1. In the Network View window, click on the Node whose initial concentration you want to change. This will bring up the Node Inspector window, which lets you change various characteristics of each Node.

2. To change the Node's initial concentration in a single Cell, click once on that Cell in one of the clumps of Cells in the Cell View window. It doesn't matter if the clump is displaying the Node you are changing or not, so long as you click on the proper Cell in the clump. However, if the clump of Cells you use does not correspond to the Node, be careful not to double-click on the Cell, as double-clicking will change the Node as well as selecting a Cell.
3. To change the Node's initial concentration in all Cells at once, find the 'Cell' text field in the Node Inspector window and type in –1. In this case a –1 indicates that the operation refers to all Cells rather than to a particular Cell.

4. Type your new initial concentration into the 'Init Value' field in the Node Inspector window. Although all positive values are legal initial concentrations, the mathematical equations in Ingeneue models are scaled so that the maximum steady state concentration of most kinds of Nodes should never be above 1. So in general, you should confine yourself to initial concentrations between 0 and 1.

5. Click on the 'Set' button in the Node Inspector window.

**Viewing and Changing Node Concentrations**

We often find that we want to know the concentration of a Node in a particular Cell while the model is running, and to see how this concentration changes quantitatively over time. You can both view and change current concentrations in the Node Inspector window. To find the concentration of a Node in a Cell, do the following:

1. If the Cell View window is showing a clump of Cells for the Node whose concentration you are interested in, double-click on the Cell in that clump where you want to see the concentration.

2. If the Cell View window is not showing the Node of interest, click once on the Cell you want in any of the clumps in the Cell View, and then click once on the Node you want in the Network View window.

3. With either method, look at the 'Curr Value' field in the Node Inspector window. It will now be showing the concentration of the selected Node in the selected Cell. Concentrations should almost always lie between 0 and 1.

If you want to change the concentration of a Node in a particular Cell, do the following:

1. Follow the previous set of instructions to view the current value of the Node in the Cell you want to perturb.

2. Type your new concentration into the 'Curr Value' text field.

3. Click on the Set button in the Node Inspector window.

**Running Models Automatically with an Iterator**

Once you are convinced that a model is operating correctly you may want to automate running it. In some cases, you will want to use some optimization technique to try and find optimal parameter values for the network to perform some task well. In other cases, you will want to sample parameter space to see how often the network performs a task well, or to see what range of patterns the network can generate. Ingeneue includes a set of objects called Iterators that will automate the process of running a model and searching parameter space. The task that the network should perform is generally not specified by the Iterator itself, but rather by another class called a Stopper. Stopper objects usually define a pattern that the network is supposed to generate. Separating the pattern and the algorithm for running the model means the same Iterator can be used to search for many different patterns.

As with networks, Iterators have their own text files. Most of the network files available from the Ingeneue website also have one or more associated iterator files which were used...
to generate results from that network. To load and run one of these iterators, do the following:

1. Select 'Load...' from the 'File' menu.
2. Select the Iterator file from the standard file dialog which appears.

You should see some text in the Console window indicating whether the Iterator loaded successfully.

3. Run the Iterator by selecting 'Run Iterator' from the 'Run' menu.

Most Iterators will send some output to the Console window, for instance the number of runs they have performed and whether the current run was successful or not. Most Iterators will also output data to a file. The base name of the output file is given near the top of the Iterator file, and a timestamp is added at the end of each output files name so output files don't overwrite each other. This output file is a "Cam" file, and can be loaded back into Ingeneue or accompanying analysis programs as described below and in the section of the manual on Cam files.

Iterators are one of the major features of Ingeneue, and are discussed further in their own section below, as well as in one of the programming sections at the end of this manual.

**Examining Output Files from an Iterator**

Most Iterators write out a text file containing sets of parameters values, and scores associated with each parameter set saying how well the model, with that set of parameter values, matched some desired behavior. To see the sets of parameters in an output file, open the file using the 'Load...' item in the 'File' menu. A new window will appear with a small circle in the middle, a large circle on the outside, and spokes going between the two circles. (You will probably need to expand the window to see everything clearly.) Each spoke represents the value of one of the parameters, as labeled on the outside end of the spoke, and a set of parameters makes a polygon each of the spokes at a single point. The controls at the bottom of the graph let you flip from one parameter set to another, plot all parameter sets that meet some criteria, and select certain parameter sets yourself to plot together. You can also load the values from a particular parameter set back into the model, letting you rerun the model and see the pattern generated with those parameters. Finally, the output window has a 'Stats' menu which will calculate several different statistics on the currently-selected parameter sets, including average parameter values, cross-correlations, and simply dumping the parameter values to a columnar file for use outside Ingeneue.

We also have a separate program which can perform more sophisticated analyses on Cam files. This second program ("Gatherer") is not yet ready for general use, but the last part of this manual explains some of the analyses we are working on.

That completes the quick tour of Ingeneue, and should be enough to get you started in playing with networks constructed by others. The rest of this manual describes how to construct your own Ingeneue models.
Network Files

We currently use text files to specify networks. We are moving towards a more graphical way of constructing networks, but at present to make a new network you need to run a text editing program and make a new network file. It is most convenient to use a text editor designed for programming, of which there are free varieties for all computer platforms. The network file is divided into sections specifying different parts of the network. An outline of these sections is as follows:

```
&Model
    &width  4
    &height 3
    &numsides 6

&Network my_net

&Genes
    ...
&endGenes

&Interactions
    ...
&endInteractions

&InitLevels
    ...
&endInitLevels

&ParameterValues
    ...
&endParameterValues

&endNetwork
```

The "&" character is used throughout our files to delineate tags that indicate which pieces of information are coming next. The top few lines in the file give some overall information, such as the dimensions of the array of cells to use (width, height) and the shape of each cell (numsides). Next comes a "Genes" section where the names (and some other features) of the Nodes are listed. The "Interactions" section is where one specifies which Affectors govern each Node. The "InitLevels" section is the place to specify initial concentrations of each Node. Finally, in the "ParameterValues" section one can specify values for each parameter (this is required but sometimes meaningless because often the problem is to find those values), as well as the range. We discuss each of these sections in detail below.
**General Information**

The top few lines in the Network file contain tags specifying information about the whole model. Currently there are 5 of these tags. The first thing in a network file must be the tag &Model. This is used by Ingeneue to automatically recognize that this file specifies a network. The other tags in this initial section are as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;width</td>
<td>The width of the array of cells where the model will run.</td>
</tr>
<tr>
<td>&amp;length</td>
<td>The length of the array of cells where the model will run.</td>
</tr>
<tr>
<td>&amp;numsides</td>
<td>The number of sides each cell has. Can currently be set to 2, 4, or 6. We always use 6 sides, making our cells hexagonal. The others are relics from when we were just getting things to work.</td>
</tr>
<tr>
<td>&amp;Network</td>
<td>This is the name of the network. It's not important what you name it, but it is often used as the root for iterator output filenames.</td>
</tr>
</tbody>
</table>

Model Cells are arranged in a two-dimensional grid, linked to their neighbors at their edges. Thus a square cell will be connected with four other cells, whereas a hexagonal cell will be connected with six other cells. These connections are important to membrane-bound Nodes that interact with Nodes in neighboring cells. The grid we use wraps the edges around a torus, so that Cells on the top edge of the grid are connected to Cells on the bottom edge, and those on the right are connected to those on the left. This avoids edge effects, but is only suited to certain problems. Future versions of Ingeneue will offer alternative options for boundaries.

Note that if you are modeling a network whose dynamics are to be completely internal to each cell, with no cell-cell communication, you can use a 1 x 1 grid for the model. Alternatively you can make a larger grid but start each cell in that larger grid with different initial conditions to get many runs of your network in a single run of the model. This turns out to be more efficient time-wise than running a single cell multiple times.

**Defining Nodes**

Recall that a Node is a mRNA, protein, or other molecular species whose concentration you want to keep track of. Two typical Nodes, one for a membrane bound protein (Notch in this case) and the other for its mRNA, are specified (in the "&Genes" block) as follows:
As with the sections of the file, individual components in the file are tagged with an "&". Within the Genes block each Node is defined in a sub-block, starting with "&X" and ending with "&endX", where X is the Node name. In between appear tags that define Node characteristics. The table below lists the tags that can be used in defining a Node, along with the default setting if a tag is left out. All these tags are optional – a Node can be declared just by giving it a name followed directly by the &end_ tag.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;Location</td>
<td>Whether these molecules reside on the cell surface or interior. Legal values are &quot;membrane&quot; or &quot;cytoplasmic&quot; (&quot;cyto&quot; is also legal).</td>
<td>&quot;cytoplasmic&quot;</td>
</tr>
<tr>
<td>&amp;Color</td>
<td>The color to draw the Node in the Cell View.</td>
<td>pink</td>
</tr>
<tr>
<td>&amp;Show</td>
<td>Whether to show it in the Cell View (&quot;on&quot; or &quot;off&quot;).</td>
<td>&quot;off&quot;</td>
</tr>
<tr>
<td>&amp;Scale</td>
<td>The concentration at which the Node is drawn at full brightness in the Cell View window. Should be a value between 0 and 1.</td>
<td>1</td>
</tr>
<tr>
<td>&amp;Type</td>
<td>The type of molecule this Node represents. Currently can be &quot;rna&quot;, &quot;protein&quot;, &quot;complex&quot; (i.e. of two proteins) or &quot;input&quot; (poorly defined as yet).</td>
<td>&quot;protein&quot;</td>
</tr>
<tr>
<td>&amp;XPos, &amp;YPos</td>
<td>x and y positions at which to draw the Node in the Network View.</td>
<td>Defaults to placing Node on a circle.</td>
</tr>
</tbody>
</table>
Parameters

The system of differential equations, specifying how Node concentrations change over time, is assembled using Affectors, and each Affector may have one or more parameters. These parameters are each given a name in Ingeneue and referred to throughout the program by that name. We use standard prefixes for naming different types of parameters. Some of the common ones are shown in the following table:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>K_Xy</td>
<td>Level at which a node reaches half its maximal activity in some dose-response relationship.</td>
</tr>
<tr>
<td>nu</td>
<td>nu_Xy</td>
<td>Hill coefficient (&quot;cooperativity&quot;) of a dose-response curve.</td>
</tr>
<tr>
<td>H</td>
<td>H_X</td>
<td>Time constant for non-specific degradation of a Node. The time constant, or decay time, is the inverse of the decay rate, and is more or less the same as the half-life (which is ln(2) times the time constant).</td>
</tr>
<tr>
<td>max</td>
<td>maxX</td>
<td>The maximum concentration of a Node, in molecules per cell. This parameter falls out from the scaling of concentrations that we do, and it is required for stoichiometric reactions.</td>
</tr>
<tr>
<td>alpha</td>
<td></td>
<td>Factors for scaling two different Affectors relative to each other when they are both involved in a single process, for instance in an enhancer that has two regions which both promote transcription.</td>
</tr>
<tr>
<td>r</td>
<td>r_X_Y</td>
<td>Rate constants for dimerization reactions</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>Rate constants for other reactions</td>
</tr>
</tbody>
</table>

There is no magic to these prefixes. Ingeneue does not inherently recognize any particular prefixes. If you have another class of parameters that you want to give a set prefix, you are free to do so; if you prefer you are entirely welcome to name your parameter after childhood pets, but it probably won't be very mnemonic. We have adopted the convention of making parameter names by adding an underscore and then the names of the Nodes it concerns to the end of one of the above prefixes. For instance, the half-life of the Notch protein has the name H_N. Where a single Node acts in more than one equation, we make the parameter names for each equation as above with the name of the Node being acted upon added to the end. The half-maximal activation of Su(H) by Notch protein would have the name K_N_SUH. These conventions, though in no way mandatory, may help avoid mistakes.
**Affectors**

Affectors are building blocks for the differential equations that govern how Node concentrations change over time. Whereas the ensemble of Nodes stores the present state of the model (that is, all the concentrations in all the Cells and Cell surfaces), the ensemble of Affectors can compute the derivative at that point. A Node typically contains at least two Affectors, one of which is a production term and the other a non-specific decay term. Some Nodes have more than one production term, and some also have other terms representing binding of different molecules to each other, diffusion, and so on. Ingeneue includes many Affectors, encompassing formulas for a wide range of typical genetic/molecular interactions. You will likely find, however, that you need to model some interaction using a formula not yet represented in the Affectors library. In that case you will need to do a little programming, as discussed later in the manual.

Affectors have the following format in the input file:

```plaintext
&Interactions
    ...
    &N
        &Tln1Aff n
        &DecayAff N H_N
    &endN
    ...
&endInteractions
```

All Affectors for a given Node are grouped together in between an "&X" and an "&endX" tag, where X is the name of a Node defined in the &Genes section. To add an Affector to a Node you give the name of the Affector followed by the names of the Nodes and parameters that Affector uses. The order and type of input each Affector requires is given in a web page on each Affector in Ingeneue's online documentation.

The change in concentration of a Node is the sum of the values of all its Affectors. This works well for Affectors that represent kinetically independent processes. Often, though, the processes that regulate a Node's level are not independent of each other and cannot simply be summed. For instance, two different regions of an enhancer might each be capable of driving transcription of a gene at its maximum rate, but the gene clearly will not be transcribed at twice its maximal rate if both enhancer regions are activate concurrently. In general, all the processes that regulate transcription of a particular gene must be grouped together into a function that saturates at that gene's maximal transcription rate. One can either 1) make use of a few pre-built Affectors that represent various simple cases, 2) write new, specifically tailored Affectors, or 3) make use of special nested Affector classes called EnhancerRegionAff, MultiEnhancerAff, and ProductAff to compose the necessary formula (these are discussed below).

The online documentation for Ingeneue contains a list of Affectors currently included, along with descriptions of each Affector's formula, uses, and parameters. You will notice that occasionally the list of parameters given to an Affector doesn't quite make sense (for instance, the decay time constant is used by many Affectors). This is a consequence of our non-dimensionalization scheme, which is discussed further in the section on programming Affectors. Here are some of the most commonly used Affectors.
Decay

Most Nodes exhibit first-order decay, and this is represented by the DecayAff. DecayAff takes two inputs, the Node it governs and the half-life (actually the time constant, the inverse of the first-order reaction rate) for that Node:

\[
&DL
&\text{DecayAff} \quad DL \ H\_DL
...
&endDL
\]

Translation

Nodes that represent primary protein products have at least a DecayAff and an Affector representing translation. Often, that's all that happens to a protein; it may do things, even appear in many other terms, but its own concentration merely tracks that of its mRNA. The TlnAff is our standard translation Affector, taking as input the mRNA to be translated and the half-life of the protein. Translation of delta mRNA to Delta protein would go inside the Delta protein block within the &Interactions section as follows:

\[
&DL
&\text{TlnAff} \quad dl \ H\_DL
...
&endDL
\]

Transcription

Transcription is much more complicated than translation because enhancers are by their very nature regulated, often by multiple regulators. If the enhancer for a particular gene has only one activator and no inhibitors then the Txn1Aff can suffice for that gene. Txn1Aff takes as input the activator Node, the half-maximal activation parameter for the activator, the cooperativity coefficient associated with this interaction, and the mRNA's decay time:

\[
&\text{Txn1Aff} \quad AC \ K\_ACdl \ nu\_ACdl \ H\_dl
\]

A few other simple cases are encapsulated in the current Affector library; for instance, Txn2aAff encapsulates the case of a single activator competitively inhibited by another species. Most applications, however, will require a bit more sophistication. A set of nested Affectors help one build complex enhancers. The top-level Affector for this nesting is called EnhancerRegionAff. It corrects the values of what's inside of it to correspond to the way we have non-dimensionalized our equations (if that doesn't mean anything to you, don't worry, just use it when building up complicated transcriptional terms). We have three more Affectors designed to go inside of an EnhancerRegion, and which in turn wrap normal Affectors. SumAff adds together the values of all the Affectors it contains. ProductAff multiplies together the values of the Affectors it contains. MultiEnhancerAff combines its sub-Affector values as if they were probabilities: two probabilities 0.5 and 0.5 don't combine
to yield probability 1.0; instead, $1 - (1 - 0.5)(1 - 0.5) = 0.75$, the probability of one or both events. In this case the event in question could be the binding of a regulator to a site in an enhancer. Here's an example using both a ProductAff and a MultiEnhancerAff to model the \textit{achaete} enhancer, whose activation depends on both Achaete (AC) and Scute (SC) proteins and which is inhibited by Enhancer of Split (ES):

\begin{verbatim}
&ac
 &EnhancerRegionAff H_ac
 &ProductAff
 &MultiEnhancerAff
 &TxnSiteActivatorAff AC K_AC_ac nu_AC_ac
 &TxnSiteActivatorAff SC K_SC_ac nu_SC_ac
 &endMultiEnhancerAff
 &TxnSiteInhibitorAff ES K_ES_ac nu_ES_ac
 &endProductAff
 &endEnhancerRegionAff
 &DecayAff ac H_ac
 &endac
\end{verbatim}

Notice how the MultiEnhancerAff is nested inside of the ProductAff, which in turn is nested inside of the EnhancerRegionAff. This means that values of the two TxnSiteActivator Affectors will be combined probabilistically, then multiplied by the TxnSiteInhibitor Affector, and finally corrected by EnhancerRegionAff to give the final transcriptional activity level for ac. Note that the bottom-level Affectors used inside EnhancerRegion (TxnSiteActivator, TxnSiteInhibitor) are different than the terms you would use outside an EnhancerRegion. The online documentation explains when to use each. The way we have chosen to model multiple activators and inhibitors on this enhancer is just one choice out of several we could have made, and we have yet to explore the consequences of making different choices. At present we have no reason to believe that this particular form is any better (or any worse) than others.

\textbf{Specific Parameter Values}

Once you specify all the Affectors for your model, you will have a collection of parameters. These parameters will all get default values and ranges if the &DefaultParameterValues section of the input file is complete. This may be sufficient, but sometimes one may want to give a different default value and/or range for a particular parameter. One must also specify values and ranges for any parameters whose prefix is not listed among the defaults. These assignments belong in the &ParameterValues section of the input file. The structure of this section looks identical to the &DefaultParameterValues section. Here's an example:

\begin{verbatim}
&ParameterValues
 ... &H_N 10 1 100 Linear
 ... &K_N_SUH 0.1 0.01 1.0 Logarithmic
 ... &endParameterValues
\end{verbatim}
Each line starts with the name of the parameter. The first number is the default value for that parameter. This default value is rarely used, since normally some search and optimization algorithm picks parameter values, but it will be used if you simply run the model. Thus, if you have found, know, or can guess values for all your parameters, you can specify those values as the defaults. The next two numbers are the minimum and maximum values that a parameter of this class can have. The final item indicates whether the parameter should be sampled along a linear or logarithmic range. Other tags may be available in future versions if we run across a use for them. The same line format (&name, value, min, max, range_type) is used to specify parameter values throughout all Ingeneue files.

**Initial Conditions**

In the final part of the network file each Node is assigned an initial value. Each Node should have at least a default initial value given in this section, and some Nodes may need more complicated initial conditions. One can specify patterns of initial values for a Node using an InitialCondition object, of which Ingeneue has several (including ones that can set up stripes, single cells on or off, bullseye patterns where the Node is on strongest in one cell and fades out in neighboring cells, and others). As with other Ingeneue objects, it is relatively straightforward to program your own InitialCondition if none of the existing ones makes the pattern you want. Programming InitialConditions is described in a separate chapter. The pre-built InitialConditions included with Ingeneue are all described in the online documentation. Here is an example of what the initial condition section of a network file looks like:

```
&InitialConditions
 &DefaultValue N 0.5
 &DefaultValue n 0.2
 &DefaultValue Dl 0.01
 &BullseyeIC
   &Node   Dl
   &Value  1.0
 &endBullseyeIC
 &endInitialConditions
```

Each Node should be given a default value with the DefaultValue tag. The Node will start with this value in all cells that are not set otherwise using another Initial Condition object. To specify other Initial Conditions type an & followed by the name of the Initial Condition, as shown with BullseyeIC. Within the declaration of an Initial Condition object go any parameters for that Initial Condition; close with an end tag as shown. Multiple Initial Conditions may be specified for each Node.
Iterators

In Ingeneue's lingo, Iterators are algorithms for running a model multiple times, often searching for a particular pattern. Ingeneue contains both simple Iterators which find one set of parameters that matches their goal, and also an ÜberIterator which allows a search to be repeated multiple times, perhaps with different starting parameters. Some Iterators encode so-called optimization routines which try to search parameter space in a smart way to find parameters that will generate a particular pattern. Others will sample parameter space randomly or systematically. Still others will do more specialized tasks that were needed to explore one particular aspect of some network. In most cases, the Iterator needs to know how closely the pattern generated by the model matches the desired pattern, and this is encoded by another Ingeneue object called a Stopping Condition. In more mathematical terms, the Stopping Condition is the scoring function used by the Iterator.

An Overview of an Iterator file

Iterator files are text files that specify an Iterator object to use in running models. The basic structure of these files is as follows:

```plaintext
&Iterator IteratorName
 &An_Iterator_Parameter
 &Another_Iterator_Parameter

&ParamsToVary
   &K_X 0.1 0.01 1.0
&endParamsToVary

&Stopper StopperName
 &A_Stopper_Parameter
&endStopper
&endIterator
```

The Iterator parameters and what follows them will depend on the particular Iterator but often include things like step sizes, tolerances, and so on. The ParamsToVary section specifies which of the model's parameters the Iterator will work with. For instance, an Iterator that is trying to optimize a model towards some condition will only change the parameters in the ParamsToVary section during its search, even if the model itself contains many more parameters. The ParamsToVary section has the same format as the ParameterValues section in the network file discussed above.

The scoring function that the Iterator uses is given in the Stopper section. Stoppers are generally pattern recognition objects which score how well the behavior of a model (as it runs) matches a target pattern. They get their name both because they determine when to stop the current run, i.e. when some condition is met such as the amount of time passed or the achievement of a particular stable pattern. Since the scoring function is separate from the Iterator, you can mix and match different Iterators with different ways of scoring a pattern. The base Stopper class (SimpleStop) is also sometimes used simply to stop a model from running after a certain amount of time has passed. See the next part of the manual for further discussion of Stoppers.
Descriptions of the Iterators currently included in Ingeneue are given in the online documentation.

**Restricting Parameter Values Relative to Each Other**

There is a slightly-less-than-mature mechanism (ParameterRule objects) for enforcing mutual constraints on parameters. See iterator files provided by tutorials for some examples.

**Searching Multiple Times using the ÜberIterator**

Although it may be useful to run an Iterator a single pass at a time (in order to check that it's working), you will most often want to run an Iterator many times. This can get tedious to do by hand, so Ingeneue includes a super Iterator (called the ÜberIterator) which can run any number of Iterators and StoppingConditions again and again, each time starting from a different initial set of model parameters, and saving the results to a file. A simple use of this UberIterator is to pick random parameter combinations, run the model with a StoppingCondition that recognizes a given pattern, and save each parameter combination and the score it got to an output file. More complex uses might involve using an Iterator to optimize the score from that initial parameter set, feeding the same parameter set to multiple Iterators, or even using a series of Iterators and StoppingConditions in which the set of parameters that emerges from the first one gets fed to the second one, and on down the line. The UberIterator is a complicated thing to use and at some point we'll probably clean it up or replace it with a macro language. In the meantime, here's a brief explanation of some of its capabilities.

An ÜberIterator file has the following basic structure:

```plaintext
&Iterator ÜberIterator  "UI.txt"
 &OutfileName  acsc11_15.txt
 &RunMode Random
 &MaxStepFraction  1.0

&Evaluators
   &Iterator  I1 StartAtBeginning Savefinalpars
     &< IteratorName >
     < IteratorParameters >
   &endIterator
 &endEvaluators

&ParamsToVary
  < Parameters >
&endParamsToVary

&endIterator
&endHeader
```

The top part of the file sets a few flags and parameters that govern the behavior of the UberIterator. These can include the following:
&RunMode | Specifies where to get sets of parameter values. Set this to one of the following:

Random – pick parameter sets randomly from a uniform distribution (for parameters that are Linear) or a log-uniform distribution (for parameters that are Logarithmic).

FromCam – use the currently selected parameter sets in the most recently loaded Cam file.

FromFileList – not sure what (or whether) this does.

&MaxStepFraction | This is probably a relic, but set it to 1.0

&OutpathName | This tells the ÜberIterator where to write its output file; specify it as a Unix-style path name (i.e. directories separated by a '/')

&OutfileName | Alternative to the tag above. This is supposed to give a base name for the output from the ÜberIterator. Sometimes it works, so give it a good name.

&RandomSeed | Seed for the random number generator; this can be useful in debugging because you'll get a reproducible series of values.

Only the first of these (RunMode) is necessary. You can leave the rest out if you are not using them.

The next part of the ÜberIterator file describes the gauntlet of other Ingeneue objects you would like the ÜberIterator to run each parameter set along. The ÜberIterator can run regular Iterators and can also run the loaded model with a given StoppingCondition or Experiment (not covered here). You can make a whole list of objects that you would like the ÜberIterator to run, and you can have each object start from the same initial parameter set or you can have each object use the parameter set returned by the previous object (e.g. from an optimizer which changes the values of the parameters). You also have choices about when to save parameter sets to the output file.

You choose each of these options following the '&Evaluators' tag, which indicates to the ÜberIterator that a list of different objects for it to run will follow. You tell the ÜberIterator what this list of objects is using the '&Iterator', '&Stopper', and '&Experiment' tags. Next to each object tag you can give a set of flags. The flags are as follows:
<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt; id_tag &gt;</code></td>
<td>The first thing following the object tag must be a name that will identify output from this evaluator in the output file. We usually use I1, I2, I3, … for iterators, S1, S2, S3, … for stoppers.</td>
</tr>
<tr>
<td>StartAtBeginning</td>
<td>Indicates that the ÜberIterator should pass the parameter set from the beginning of the round of evaluations to this evaluator. If this tag is missing, the evaluator will get the parameter set output by the evaluator just preceding it. Of course if the evaluators are merely Stoppers which don’t alter the parameter set, the tag is meaningless.</td>
</tr>
<tr>
<td>Savestartingpars, Savefinalpars, Saveboth</td>
<td>Tells ÜberIterator whether to save parameter sets to the output file. ÜberIterator will save the parameter set that was input to this evaluator, the parameter set as returned by this evaluator, or both, respectively for the three different flags. Use one or none of these flags. In any case, parameter sets will only be saved if the evaluator says it completed successfully.</td>
</tr>
<tr>
<td>FixedPoint</td>
<td>Tells ÜberIterator to use a fixed-point iterator instead of integrating the equations. If this flag is present you must also have, on the next line, a &amp;FPStabilizer tag followed by the stabilization value for the fixed point iteration (try 0.2).</td>
</tr>
</tbody>
</table>

The name of the actual evaluator goes on the following line. Here are some typical evaluator declarations:

```plaintext
// Run the RandInitRunIterator starting from the unmodified parameter set // and save the parameter set that the iterator returns to the output // file following the I1 tag.
&Iterator I1 StartAtBeginning Savefinalpars
   &RandInitRunIterator
   < parameters for iterator >
&endIterator

// Run the MetaFPStripeStop stopper, starting from the unmodified // parameter set and save the parameter set to the output file // following the S1 tag if the stopper found a good pattern. // Use the fixed point integration scheme with a stabilizer of 0.3
&Stopper S1 StartAtBeginning Savefinalpars FixedPoint
   &FPStabilizer 0.3
   &MetaFPStripeStop
   < stopper parameters >
&endStopper
```

After the declaration of the name of the actual object to run (RandInitRunIterator and MetaFPStripeStop above), you put the parameters for that object into the file. The end of
the object is marked by an end tag such as 'endIterator' or 'endStopper'. When you have completed the list of objects to run, finish with the 'endEvaluators' flag.

As with any other Iterator, you must also give the ÜberIterator a set of parameters to vary. These are the parameters that will be picked randomly or set from the Cam file (depending on the RunMode). Unlike other Iterators, you do not need to give an ÜberIterator a StoppingCondition.

The ÜberIterator is a finicky, confusing beast. The best way to use it is to find another file that uses it, copy that file, and modify it to suit your needs. That's what we do.
**Stoppers**

As discussed in the Iterators section above, the pattern recognition and scoring objects in Ingeneue are called StoppingConditions or Stoppers. A Stopper provides two things when a model is running. First it provides a score which says how closely the current pattern matches the pattern which the Stopper wants to see. Our Stoppers return low scores for good patterns, with 0 indicating a perfect pattern, and higher scores for poor patterns (often maxing out at 1 for a terrible pattern). A Stopper also provides a boolean signal to the integration code telling it whether to keep running or to stop. All Stoppers have a maximum length of time for which they will run (and tell the model to stop after that time has elapsed) which you set with the '&StopTime' tag. A Stopper can also tell the model to stop if its score has gotten better than or worse than some threshold value, or for another criteria which indicates that its not worth spending more time running the model.

Because pattern recognition is a hard problem, these Stoppers tend to be fairly complicated. We currently have a small collection of stoppers for recognizing spots, stripes, on/off patterns, and steady states. The spot and stripe Stoppers try to avoid stopping when concentrations are oscillating. The currently-included Stoppers are all described in the online documentation. Rather than making a new Stopper for every particular pattern, we instead have constructed a "meta-stopper" which can combine together several simple Stoppers. The next section describes that.

**Combining Stoppers**

To make specifying a complicated pattern more convenient, Ingeneue includes a way of combining together several Stoppers into a single scoring function. You do this using a Stopper called "MetaStop". The standard MetaStop can combine the scores of several regular Stoppers, either taking the maximum score from the group (so you would be minimizing the worst part of the pattern) or adding all the scores together. It can also signal that the model should top running when all its subsidiary Stoppers are ready to stop, or when any single one of the subsidiary Stoppers are ready to stop. The format for MetaStop is quite simple. In the part of the Iterator file where you would normally put your Stopper, use this instead:

```plaintext
&Stopper MetaStop
 &StopTime 500
 &Cutoff 0.2
 &ValueMode Max
 &StopMode And
 &Stopper < a subsidiary stopper >
   < subsidiary's parameters >
 &endStopper
 &Stopper < another subsidiary stopper >
   < subsidiary's parameters >
 &endStopper
 ...
 &endStopper
```
The top four parameters determine the maximum time that this MetaStopper will let a model run, the threshold below which the MetaStop will consider a score to be "good", and then the way MetaStop calculates its score (either "Max" or "Add") and the way it decides whether to stop (either "And" or "Or"). Below these are the list of subsidiary Stoppers, each of which is indicated with an &Stopper tag followed by the name of the Stopper to use and then the parameters for that Stopper. You can nest MetaStoppers if you want.

MetaStop is one of a few "meta"-Stoppers we currently have available. Others include MetaStopTwo and MetaStripeStop. Look at their documentation to see how each one works. As with Iterators, it's often best to start with a file that already works, copy it, and modify it to suit your needs.
Interface

Our eventual interface is still under construction, and right at the moment is caught between different strategies. We are aiming towards having a modern, mouse-and-graphics based interface that allows the user to set up a model from within the program, visually represents that model, and lets the user run the model, iterate the model, and conveniently view and analyze the results. The complete version of this goal is still a year or two away, but small parts of it are appearing in the program along with holdovers from our first couple tries at interfaces. On top of this confusion, the interface code is not as well debugged as the rest of the program.

Interface for Constructing and Running Models

The main window around which the rest of the interface revolves is the Network View. The Network View automatically appears whenever you load a new network file, and you can always make it reappear or bring it to the front of the screen by selecting 'Inspect Network' from the 'Other' menu. The Network View shows a representation of the currently loaded network file, including various geometric shapes showing all the Nodes and arrows showing the Affectors connecting the Nodes. Clicking on any of the Nodes highlights the Affectors constituting that Node's equation (in green) and the Affectors (for other Nodes) in which that Node participates (in red). The shape used for each Node indicates what kind of thing it is: ovals for mRNAs, rounded squares for proteins, and diamonds for complexes.

A second window, the Cell View, is used while running models to visually see the changing concentrations of selected Nodes. This window also automatically appears when you load a network file, and you can always re-open and bring it to the front of the screen by selecting 'Inspect Cells' from the 'Other' menu. The Cell View window shows a grid of Cells for each Node whose concentrations you want to view, drawn one above the next with labels to the right side. You can draw each Node in a different color as a cue to which Node is which. This view is essential when running a new model or testing an iterator to get a feel for the model and iterator behavior. Visually watching the output of complicated models such as these is one of the most powerful ways of catching mistakes. When you are sure everything is working with an automated search, and you are tired of watching your model run, closing this window can slightly speed up the search.

Clicking on a Node or Affector in the Network View or a Cell in the Cell View will bring up an Inspector window. This floating window lets you examine and change various values and parameters in the model. The exact items in the Inspector window will change depending on what you select, but there are two main panels.

One panel lets you view and change values associated with a Node, including its initial and current value within a particular Cell, whether to draw that Node in the Cell View, and the color to use in drawing that Node. To modify any of these values, click on one of the Nodes in the Network View and on a Cell in the Cell View, in either order. The Inspector window will then show the values for that Node in that Cell. When you click on a Cell in the Cell View, it doesn't need to be in the grid of Cells associated with your selected Node. It just needs to be a Cell in the appropriate position in the grid (although if you double-click in the Cell View the Inspector will switch Nodes too). You select the Node itself in the Network View. To change any of the values, first make the change in the Inspector view and then click the 'Set' button.
There are a few short-cuts you can use to view Cell/Node values. If you want view the value of a particular Node in several Cells, first select the Node in the Network View, then click once on each Cell in turn in the Cell View. You don't need to reselect the Node for each new Cell. Similarly, to view the values of several Nodes in a single Cell, first select the Cell, then click on each Node in turn in the Node view. For those Nodes whose concentration is being shown in the Cell View, you can also just double-click on the Cell/Node combination you want to see to simultaneously do both selections. You can also change both the Cell and the Node by typing into the appropriate fields in the Inspector window. Finally, if you'd like to change the initial value of a Node in all Cells at once, click in the Cell View outside of any of the Cells, then click on the Node. You will notice that the Cell field in the Inspector window becomes \(-1\), indicating that no Cell is selected. Now if you change the initial value and click the 'Set' button, this will affect all Cells.

The second Inspector panel lets you see and change the parameter values of an Affector. To bring up this panel, find the arrow representing the Affector you want to modify in the Network View. Click on the little circle in the middle of this arrow. The Inspector window will now contain a panel showing all the parameters associated with this Affector. Clicking on any of the parameters will allow you to change its current value.
Graphing and Calculating Statistics on Ingeneue Ouput Files

The files spit out by the ÜberIterator are plain text files in a format that Ingeneue can read back in and display. You can load these back into Ingeneue to see them displayed on a wheel plot and to perform some simple statistics. We are also developing a separate analysis program (“Gatherer”) to accompany Ingeneue which does more extensive (and experimental) analyses. This program will be available soon.

To load an output file back into Ingeneue, simply use the Load command in the File menu. Ingeneue will recognize that is an output file and make a new wheel plot window to display the results from the file. Clicking on the left and right arrows will move you back and forth from one parameter set to the next. If you check the 'Hold' box, the current parameter set will continue being displayed as you move to other parameter sets. This way you can overlay many on top of each other; for example, you might flip through sets, running each one, and keeping a hold on each one that matches some criterion. Click the 'Clear' button to eliminate all holds, or the 'Plot All' button to do just that.

If you want to run a model with one of the parameter sets in an output file, find that parameter set in the wheel plot. Then click the 'Load Cams' button to impose those parameter values on the currently-loaded model. The next time you run the model, it will use these parameters.

The text boxes labeled 'Threshold' allow you to limit the display to a subset of scores. For example, you might like to see only the sets that earned the best scores. Also, you can construct iterator files such that the output includes parameter sets caught by several different evaluators. These sets will be tagged in the output file according to who caught them. The pull-down menu above the 'Clear' button allows you to choose to display only the sets caught by a particular evaluator.

The 'Statistics' menu provides a few options for dumping digests of the currently-selected battery of parameter sets. 'Averages' does what you might expect. 'CrossCorr' computes a pairwise cross-correlation matrix (Pearson coefficients – ?), and, in the output file, lists below the matrix the ten largest-magnitude coefficients. 'Dump' spits out the parameter sets, along with the score and the name of the evaluator that caught them. 'DumpValueFields' writes out just the scores. The fifth item doesn't presently seem to do anything useful.
Programming Ingeneue

Ingeneue puts together models in a very modular way to allow the user to mix-and-match pieces in constructing models. We have tried to keep the code for each piece as simple as possible, and for this early versions of the software we expect most uses of Ingeneue will involve some programming to encode new Affectors, new InitialConditions, StoppingConditions, and Iterators. The Affectors are particularly simple to write. StoppingConditions and Iterators also have simple forms, but in practice they require much more code. The basic structure of StoppingCondition and Iterator classes are described in subsequent sections.

Making a New Affector

A tutorial available on our website explains how to make a new Affector. Warning: the surest way to deceive yourself with models is to write bogus equations. Ingeneue makes it easy to add any formula you want to the Affectors library; while this is a "feature", nothing will prevent you from introducing and using completely unrealistic formulas. It is up to you, the user, to check your math, rationalize it carefully, and then, before trusting your code, make sure that it does what you think it does.

Making a New Iterator

Iterators are objects that can run a model multiple times. You may want to write an Iterator in order to implement a new optimization routine or to do some other task that involves running models automatically. For instance, we have written Iterators that scan through different initial conditions to see how a model's pattern-forming behavior changes with initial conditions. We've also written Iterators to systematically vary parameter values to map out the scoring surface for a given model and Stopper combination.

Iterators descend from the base class ModelIterator. You can find a template of a new Iterator in the file called TemplateIterator.java in the iterators folder. If you look over that template, you'll see quite a few comments explaining what each of the functions should do. Here are more instructions for the two functions among these that will take the most modification, loadParameter() and doRun().

loadParameter() is called to load parameters of the Iterator class. Iterator parameters are items that are loaded from a file or otherwise set by the user to control the functioning of that Iterator. (Don't confuse these with the model's parameters discussed throughout the rest of this manual.) An optimizer, for instance, might have a parameter determining the size of the steps it should take or the number of steps to take before giving up. Other parameters might select between different modes that the Iterator can use, or specify particular model parameters to work with, or specify any other thing about that Iterator's operation that should be set by the user.

As described in the Iterator section, Ingeneue uses the '&=' sign and then a text string tag to name each parameter. The loadParameter() function is called for each parameter in the input file and receives both the tag string and the tokenizer object for the input stream. We usually use a set of if() / else if() statements to determine which parameter is being loaded, and then call the tokenizer to actually load it. For instance, to load a floating point number parameter called Tolerance we would use code like:
protected void loadParameter(String info, BetterTokenizer tokenizer) throws Exception {
    if (info.equals("Tolerance")) {
        tokenizer.nextToken();
        tolerance = (float)tokenizer.nval;
    } else if (info.equals( ... else super.loadParameter(info, tokenizer);

Note that you should always call super.loadParameter() at the end, as in the code snippet above, to call the ModelIterator's version of this method. The ModelIterator takes care of loading the Stopper, the Function, the OutFileName where output from the Iterator is stored, and several other parameters as given in the ModelIterator documentation pages.

The second method you will always override is doRun(). The doRun() method contains the code to perform whatever it is that you want your Iterator to do. doRun() has no direct inputs, but there are several class variables that you will want to use. An array called p[] stores the current values of all the model parameters that your Iterator is supposed to be working with. The parsTV variable points to a ParameterArray object (see below) that gives more information about each of these parameters, such as its name and range. The nParsTV variable is set to the number of parameters in p and parsTV. Your Iterator can do whatever it wants with the values in p[]. When its done, it should set the values in p[] to the final model parameter values you want returned from your Iterator. You should also give this set of parameters a score and set finalScore equal to this score.

Your Iterator will want to run the model and receive the score that the model with gets with the parameters in p[]. To do this, call F( p ). The F() method sets the model's parameters to the values in the array it receives, runs the model, and returns a floating point value equal to the score which the model received (as scored by the current Stopper). To run the model with different sets of parameters, change the values in the p[] array or make a new floating point array with different parameter values and then pass the changed values to the F() method.

If you examine the code for an iterator, you will notice that the F() method itself does nothing but call another module, an object of the "Function" class, to evaluate the parameter set in p[]. You can sub-class the Function class to implement any approach you want to evaluating a score for each parameter set. For example, the "ICVaryingFunction", instead of simply running the model with the given parameter set, runs it several times, each time with a different initial pattern.