Intermediate and Advanced Modeling Techniques in *Nova*

Richard M. Salter

April 3, 2014

1 Introduction

The *Nova* modeling system is a powerful platform for building cross-paradigm simulations with high-impact visualizations; success requires, however, that the user build up his or her ability to work with *Nova*'s model design "language", comprised of both visual and textual elements. Many interesting models can be built solely using the visual language and a small amount of text. More complex models will require a larger amount of text-based programming to specify the flow of data through the model's components. Fortunately, *Nova*'s Codechip component allows you to compartmentalize this code and maintain a visual structure to the model's design.

This document will show you how to build models using *Nova*'s aggregating components (CellMatrix, AgentVector, NodeNetwork, SimWorld) where there is significant interaction among the constituents. It will also demonstrate the use of Clocked Chips for sensitivity analysis and data collection over multiple runs. To start, only basic programming skills are assumed, as outlined below. Some of the later sections will expect a deeper understanding of program design, and references are provided for further reading on these subjects.

2 Prerequisites

Before reading this document you should be familiar with the techniques of stock-and-flow systems dynamics modeling in *Nova*, and the use of chip-based submodels, as presented in the tutorials in Sections 1 and 2 of the Nova Tutorial, `http://www.novamodeler.com/tut/`. It will also be useful to look at Tutorial Section 3, even if it is not completely clear; much of that material will be included here.

3 Background

This section introduces the key ideas and terminology used in Nova model programming. It will prepare you for the examples that follow.

3.1 Operational Semantics

This refers to the behavior of the *Nova* simulating engine. To fully appreciate *Nova* design concepts, you should first know something about how it works.

Each simulation uses a *clock* to sequence its steps. The clock maintains a current *model time*, starting at a specific *start time* (usually 0) and *end time*, and incremented at each step by a *delta value* called *dt*.

Each step, or *iteration* represents the progress of the system from its state at time t to its state at time $t + dt$. The state of the system is comprised of the values of all Stocks and local variables in all Capsules used by the simulation. The user must choose an *integration method*, which determines the process by which Stocks representing continuous functions are updated.

At the beginning of the iteration model time is t, and by the end it has been updated to $t+dt$. During the iteration the computation "bootstraps" by drawing on previously computed values to compute the next generation. Depending on the integration method, model time may be updated incrementally through several substeps, however in systems dynamics models all processing is complete once model time has reached $t + dt$. This may not be the case for other simulation types. Consequently, you will notice that some components (Commands and Codechips) provide a choice of *pre-update* or *post-update* for when they are to be executed. Those selected for pre-update use component values at t (or intermediate points, depending on the integration method), while those selected for post-update use the newly computed values for $t + dt$.

An iteration consists of a sequence of *strobes*, which are actions taken at a step or substep, followed by *post-processing*, which is performed once at the end of the iteration when the clock has been updated to $t + dt$. Here is a simplified summary of Capsule iteration:

Strobe

- Strobe aggregates and chips
- Strobe stateful plugins (explaned below)
- Strobe Stocks (i.e. compute their next values)
- Strobe pre-update code chips, Commands and converter plug-ins
- Update clock

Post Processing

- post-process aggregates and chips
- Strobe post-update Codechips and Commands

- Update displays and display plug-ins
- Perform any cleanup

3.1.1 Plug-in types

In *Nova* plug-ins are used to extend the basic component platform with special functionality. In order for plug-ins to function properly we must distinguish 3 different types:

display: Some plugins, such as Raster and AgentViewerX, are used for visualization; they will want a post-update strobe to receive the latest data.

stateful: A second group have state values that are used to compute the values of Stocks. They need to be updated before any processing of Stocks during the cycle. An example of the latter is the Perceptron plug-in, which models a multiple layer neural network.

converter: A third class use the current state to produce the next, like Terms and Flows. These need to be strobed pre-update.

Plug-in type is determined by the plug-in designer and cannot be changed.

3.2 Programming

All running *Nova* simulations are expressed in a language called *NovaScript*. Even in the simplest stock-flow model, you are already writing *NovaScript* code when you enter expressions for initial Stock values, Flows, and Terms (we'll refer to these as *component definitions*). For these models, however, the code is generally restricted to simple numerical expressions. You will see that to express the relationships required of complex models your definitions will necessarily include a broader set of expressions.

NovaScript is an extension of a well-known and widely used language called *JavaScript*. This means that *NovaScript* uses JavaScript syntax for all of its code; moreover, any legal JavaScript program is also a *NovaScript* program. This includes the code used for initial Stock values, Flows and Terms. Fortunately, there are many good sources for learning to program in JavaScript. We will review a few important ideas about programming as it applies to building *Nova* models below.

3.3 Statements, Commands, Expressions

Programs consist of one or more *statements* that operate on data values. The latter are expressed either as constants or variables. Statements are divided between *expressions*, which compute a value using operators of various sorts (e.g., the arithmetic operators $+, -, *$ and $/$); and *Commands*, which change the state of the program by assigning values to variables, or by performing some side-effecting operation (e.g. "print").

The values used in a program are categorized according to their *datatype*. Virtually all programming languages include one or more numerical types (e.g., integer and reals, also called *floating point*). Another common datatype consists of textual data, which are called *character strings*, or just *strings*. Finally, languages generally provide some means of combining multiple values into a single entity. These include *arrays*, which are sequences of values indexed by integers; and *structures* or *objects*, in which the constiuents are labeled by string *property names*[1].

3.4 JavaScript

JavaScript[2] has become a prominent language if for no other reason than its role as the standard for programming the behavior of Web browsers. Here is a brief list of JavaScript highlights which will be of particular use in *NovaScript*. While this review is not enough to teach you how to program, it will help you to understand much of the later content of this document.

3.4.1 Variables

Variables are declared using the *var* keyword:

```
var x = 17, y = "hello";
var y;
```

In the latter case y is initialize to *undefined*. Variables are not restricted by datatype and can be assigned a value of any JavaScript type.

3.4.2 Datatypes

- **floating point** (i.e. real) numbers: `1, 3.14, 2.78e01`
- **character strings**: `"This is a character string"`, `"So is this"`.
- **arrays:**
  In JavaScript arrays are lists, or sequences, of values indexed by an integer argument. An empty array is created using the *new* keyword as follows:

  ```
  var a = new Array();
  ```

  Array constants are denoted using square brackets; e.g., `var b = [1,2,3,4]`. Similarly, array components use square brackets to denote the index for assignment and access; e.g., `b[0]`, `b[1]`. Unlike most languages, however, the index set may contain holes; e.g., `b[0]`, `b[1]`, and `b[7]` may be defined while `b[2]` through `b[6]` are not.

[1] also sometimes called *fields*

[2] Don't confuse JavaScript with *Java*, which, though similarly named, is a completely different language.

- **objects:**
  Every other datatype in JavaScript is an *object*. The simplest of these is the `Object` type, which consists of a list of *fields*. Each field contains a property name labeling a value. A property name must be a string, but the corresponding value can be of any type. When an object field value is a function (see below) we call that function a *method*. Once again, the *new* keyword can be used to construct an empty object, to which name-value pairs can be added:

  ```
  var c = new Object();
  ```

  Object constants are denoted using curly brackets enclosing the name-value pairs, and may included embedded object constants as values:

  ```
  var a_person = {name: "Steve",
                  date_of_birth: {month: 1, day: 28, year: 1980}
  }
  ```

- **functions**:
  Functions are unique in that they are both a datatype and contain actual executable code. JavaScript is one of a small number of computer languages that allow you to treat functions this way (sometimes called *first class* treatment). Consequently you can create a function, store it in an object or array, and pass it as data to another function where it can subsequently be executed. This kind of expressiveness turns out to be very useful for constructing complex *Nova* models.

  Functions are defined using one of two similar syntaxes:

  ```
  var double = function(x){return 2 * x;}
  function triple(y){return double(y) + y;}
  ```

  The first case allows you to create a function without the necessity of providing a name. For example, if

  ```
  var apply_to_two = function(f){return f(2);}
  ```

  Then invoking `apply_to_two(function(z){return z * z;})` will result in 4. This also shows that functions may be passed as parameters to other functions.

3.4.3 Program structures

JavaScript adapts the structure of the C and C++ languages for its code. Common control structures (i.e. coding structures that direct the flow of the program) include conditionals, for-loops and while loops. One particularly useful version of the for-loop specific to JavaScript has the following form, assuming variable a refers to either an array or object:

```
for (var i in a) {
   print(a[i]);
```

```
}
```

The index variable i will cycle through all defined indices in a.

Another important feature involves the conditional, or "if" statement. Suppose we want to assign the maximum of x and y to z. One way would be

```
if (x > y) {
   z = x;
} else {
   z = y;
}
```

This familiar statement is called a *conditional command* because it provides a pair of alternative commands, only 1 of which is actually executed, depending on the outcome of the conditional test.

An alternate way of doing the same thing uses the *conditional expression*:

```
z = (x > y) ? x : y;
```

The right hand side expression produces one of a pair of values again depending on the outcome of the conditional test. This form is particularly useful in creating the *Nova* expressions that appear in component definitions.

3.4.4 Primitive operators

Actual computation (i.e., combining and manipulating data to create new values) is performed by JavaScript's *primitive operators*, or *primops*. The most familiar primops to most people are the standard arithmetic operators ($+$, $-$, $*$, $/$). JavaScript provides additional mathematical operators via the `Math` object. This is a special object whose properties are bound to various mathematical functions. For example, `Math.sin(x)` computes the triganometric sine of the value of x. A reference to the complete listing of available `Math` primops is provided in Section 14.

A second large set of primops is provided by the *underscore.js* library. These functions are all properties of a special object denoted by the underscore character, `_`, and are documented on the library's Web page `http://underscorejs.org`. For example, `_.last(a)` will return the last element of the array a.

In addition to these, *NovaScript* adds a set of primops specifically germaine to modeling. These will be discussed more fully below.

3.4.5 Environments

The body of a JavaScript function may contain variables other than the formal parameters used to pass data into the function. These so-called *free* variables must obtain their values from some place other than

from within the function itself. We call the source of variable values used during evaluation of a function the *environment* in which that function is evaluated. JavaScript has a complex scheme for looking up variable values, however it is worth noting that the JavaScript object, with its property-name/property-value structure, is a perfect candidate to serve as an environment data structure, and JavaScript makes ample use of this fact. We will discuss environments further as needed.

3.5 Programming and *Nova*

As mentioned above, a complete *Nova* model is expressed in *NovaScript*. You create the program for this model when you click the **Capture** button. *Nova* is able to construct the scaffold for your model using the structure of the visual elements placed on the canvas. As author you are expected to provide the details connecting the various components through the component expressions. In systems dynamics models component expressions generally operate solely on numerical data; i.e., they are simple arithmetic expressions.

As you extend your use of *Nova* into more complex applications the coding used to link components will also become more complex. This complexity manifests along the following dimensions:

- **The data structures used.** Simple applications can rely on simple numerical data. For more advanced use you will need to introduce arrays and objects, and may also want to create your own functions.
- **The program structures required.** Multi-line computations involving loops and conditionals generally accompany the use of data structures such as arrays and objects.
- **The use of primitive operators.** Models using *Nova*'s aggregating components rely heavily on primops. You will need to become familiar with these primops and how they provide necessary information to the constituents of the aggregators.

3.6 What is *NovaScript*?

As mentioned above, *NovaScript* is embedded in JavaScript. Precisely, this means that *NovaScript* is JavaScript with additional functionality and special objects. The concept of extending the JavaScript core with special objects is not new. In fact, we've seen that JavaScript itself does this with the `Math` object, used to bundle together a substantial set of mathematical functions[3]. In order for JavaScript to control the behavior of a Web browser, the JavaScript environment is extended with a `Document`[4] object that contains fields specifically designed for Web browser functionality.

NovaScript similarly extends JavaScript with a set of object specifically engineered to express the design of *Nova*. Thus there are *Stock*, *Term* and *Flow* objects corresponding to those components in a visually

[3]Similarly, there is a JavaScript `Date` object used to contain date and day-of-week data

[4]For this reason, the JavaScript Web browser environment is often called the *document object model*, or DOM.

rendered *Nova* model. We call these *component objects*. Moreover, there is a *Capsule* representing an entire model or submodel, referencing all of its constituent parts. There are also *Table*, *Graph*, *Slider* and *Spinner* objects for input/output, and a single *NovaScript* object type, *VPlugin*, acting as a surrogate for every type of plug-in.

3.6.1 Simulators

The Capsule is one type of *simulator*. A simulator is a container whose components are programmed to interact, creating a runnable simulation. The other *NovaScript* simulators are the four aggregating components described in Section 6; A Capsule can contain Stocks, Flows, Terms, Commands, Codechips, etc., but also any of the simulator types (a chip is just a Capsule contained in a parent Capsule). Aggregator components (called *members*), however, must be Capsules.

Note that components such as Stocks, Terms and Flows do not contain other components as constituents. We'll use the term *base components* to distinguish Stocks, Terms and Flows from simulators such as Capsules, CellMatrices and AgentVectors.

Except for the top level Capsule, every simulator is the component of some other simulator. For example, Capsule A may contain a chip containing an instance of Capsule B. Capsule A may also contain one of the aggregators, and that aggregator will contain instances of Capsule C. We use the terms *container* and *component* to describe this relationship. *Every simulator in a model has a container except for the top-level Capsule.*

In order to actually carry out a simulation, a simulator must be associated with a *clock*. With two exceptions a simulator's clock is used by all of its components, so that only a single top-level *system clock* is required. However, new clocks are introduced with Clocked Chips, and when running in batch mode. See Section 5.

3.6.2 Component Objects and State Objects; the *Self* property

As mentioned above, components such a Stocks, Terms, Floats; but also simulators such as Capsules, CellMatices, etc. are represented in *NovaScript* as JavaScript objects called component objects. Component objects provide the computational mechanism for carrying out simulations. As a simulation proceeds each of these objects provide access to its current value through the `value()` method.

Each type of simulator object provides a way of accessing the component objects of its constituents. In the simplest case, a Capsule points to each of its components by name. CellMatrices, AgentVectors, SimWorlds and NodeNetworks also have methods that produce the component objects of their constituents. Consequently the complete state of any simulator can be found by traversing its structure to extract the `value()` at all the Stocks, Flows and Terms.

NovaScript however, provides a simpler approach: each simulator has a *Self* property that references a

special *state* object, in which base component names are bound to the current values of those components. The values in these state objects change as the simulation progresses. If a simulator is not at the top-level, then the state object of its container is the value of the property *Super*[5]. One can follow the chain of container state objects all the way to the top-level by using `Super, Super.Super`, etc.

A Capsule's `Self` object binds the name of each base component to that component's current value. This `Self` object also binds the name of each simulator component contained in the Capsule to that simulator's `Self` object. The `Self` object of a simulator provides methods to access the `Self` objects of its components.

These state objects are the most convenient way to view the current state of a simulation, and they play a crucial role in supplying values for component computations used to update that state at each iteration. Consequently, most of the primops used to discover values of components in aggregators do so by supplying state objects for those components rather than component objects. If an actual component object is required, it can be retrieved as the value of the property *self*. You should rarely need to use this, however.

3.6.3 Scenarios

When you capture a visual model *Nova* creates a set of special objects called *scenarios* that define every element of the current project. Each scenario describes either a Capsule, graph, table, cell matrix, or some other complex entity. Since we are actually writing JavaScript, each scenario is in fact a JavaScript object with various fields containing the descriptions of constituent parts. Included are the component definitions that you provide when creating the model. When a project is loaded, these scenarios provide the blueprints for construction the actual *NovaScript* objects. When a *NovaScript* program is run, the objects come to life and produce the expected simulation.

For those familiar with object oriented programming, the scenario serves as a class definition for the creation of *NovaScript* objects. As we shall see below, the *Nova* Codechip component is a tool for method definition in such a system.

3.6.4 Universal properties and primops

An *NovaScript property* is a symbol that represents a single constant value throughout the execution of a *Nova* program. Properties can be defined in several ways, as detailed below.

In Section 6 we introduce aggregating components, which are distinguished by their introduction of special properties and primops. Those properties and primops may be used in Capsules contained by the aggregator, but are generally not meaningful outside of the aggregator environment, and their use

[5]These names are chosen for historic reasons, and also because they are not likely to conflict with the names of some actual components.

may cause errors. To distinguish those from properties and primops that may be used anywhere, we'll refer to the former as *restricted* and the latter as *universal*. When describing a primop, we will always distinguish it as either universal or restricted.

Lists of all properties and primops, both universal and restricted to one or more aggregator, are found in Appendix B and at `http://www.novamodeler.com/wiki/help-2/ns/ref/primops/`

3.6.5 Special *NovaScript* objects

NovaScript also implements several special objects, including the Clock object (Section 5.1), Coord object (which models a pair of matrix coordinates with fields for row and column) and RunData object, which contains the history of a Stock's value over an entire run. These objects will be discussed as they are introduced in the sequel, and summarized in Appendix C.

4 Extending the model with code

In simple models your coding responsibility is generally limited to the arithmetic expressions used for component expressions; all other programmatic content is expressed through the *semantics* of the components themselves; i.e., what they represent and how they operate. One part of extending Nova is introducing more complex components, which we will do in Section 6. However, a second requirement is the addition of algorithms and other programmatic computations that go beyond simple arithmetic expressions. Nova has several ways of integrating new code into the existing visual structure. The most important of these is the *Codechip*. A Codechip is like a new component designed to operate on the content of a particular model. However, many Codechips express computations as universal as *Nova*'s hard-wired components, and so can be reused in multiple settings. We introduce Codechips in Section 4.2 and show some simple designs.

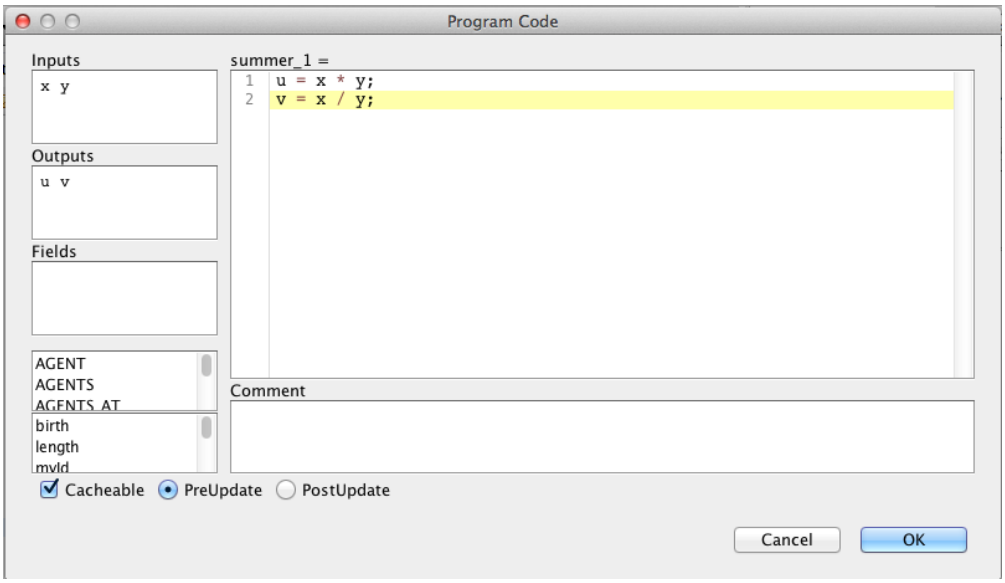
Other ways of extending the model with code by using more elaborate code for component expressions (Section 4.1), and with code entered into the *Nova* Programming Window. The latter is used to define global constants, variables, and functions; and local properties, variables and methods. This will be discussed in Section 4.3.

4.1 Component expressions

Component expressions are often a single line of mathematical code; e.g.,

```
rate * population
(TIME() < 100) ? x : y
```

You may, however, use any sequence of commands, separated by semicolons, followed by an expression:

a. 

b. 

Figure 1: Codechip *prodquo*: a) Program Code Dialog; b) Chip with pins

```
<command 1>; <command 2>; ... <command n>; <expressions>
```

This is most often seen when using `print` statements for debugging (See Section 13):

```
var ans = rate * population;
print(ans);
ans
```

4.2 Codechip overview

A Codechip is added to *Nova*'s design canvas like any other component; Unlike other components, a new Codechip is a blank slate. It has no built-in content. Any benefit must come from the programming that it includes.

Right-clicking on a Codechip produces a menu with two options: `Program Code` and `Inputs/Outputs`. If you select the former you will open a dialog box that closely resembles the `Set Property` dialog of other components. The `Program Code` dialog includes 3 panes not present elsewhere: `Inputs, Outputs` and `fields`. Let us consider only the former two for now. The basic mission of a simple Codechip is to compute 1 or more output values given a set of input values. In these two panes you respectively list the names of input and output variables; these can be any legal variables and are only meaningful in the context of the Codechip.

Figure 1a shows the Program Code dialog for a simple Codechip called *prodquo* that accepts 2 numbers and outputs their product and quotient. Note that the inputs and outputs are separated by spaces (not commas).

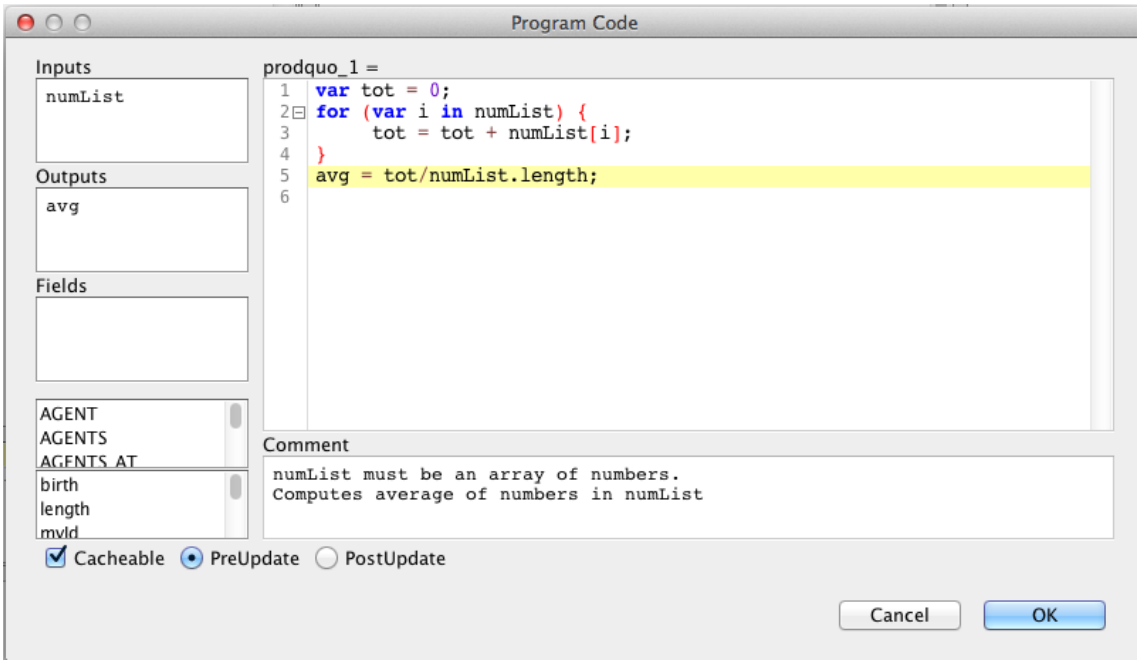


Figure 2: Codechip *average* Program Code

When you close this dialog the Codechip resembles Figure 1b. The 4 pins correspond to the 2 inputs and 2 outputs and are readily connected using the `Inputs/Outputs` dialog, as you would connect an Capsule chip.

When you open the Codechip pallet notice that prodquo is listed. Dragging from this list to the canvas will produce a second instance of prodquo. (The components use sequence numbers to distinguish Codechip instances). Editing either of these will cause the changes to appear in both – they are two instances of the same function.

This type of Codechip, called a *functional Codechip*, is the most common and easiest to write. The code window many contain any legal *NovaScript* code. **Important:** make sure you assign results to the output variables.

Figure 2 shows a typical application of Codechips: computing the average of a list of numbers. The input *numList* must be an array consisting of numbers only. This Codechip could be used in several places in a given model, and exported for reuse in other models.

In addition to the functional Codechip form, there are two others: *object Codechips* and *function-builders*. These will be discussed in Section 12.

4.3 Using the Programming Window

Each submodel places its own content in the Programming Window (just as it does for the Modeling Canvas and Dashboard). This content may contain up to 4 *segments* of code, as shown in Figure 3. The

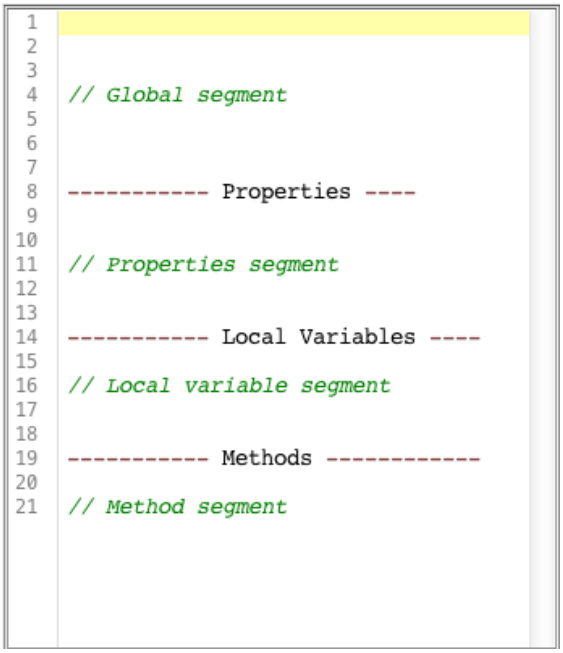


Figure 3: Programming Window Segments

segments are determined by the placement of the labeled dashed lines, as shown (4 or morel dashes to the left of the label are required; any number may follow). Not all segments are required, and, except for the Global segment, they may be in any order.

Any legal *NovaScript* code can appear in the Global segment. The remaining segments *must* follow a strict form that resembles the definition of an object constant:

```
<name 1>: <value 1>,
<name 2>: <value 2>,
...
```

Note that the final comma is required. Examples will be given below.

Global segment: Code in this section transfers exactly as written into the *NovaScript* program upon capture. This code is executed when that program is loaded, and so should consist of global constants, functions and any required initialization steps. Global segment code can appear with any Capsule, but since the code is global it can be referenced from any Capsule. Stylistically, it is probably best to only include global code in the top level model.

Properties segment: This and the remaning segments introduce *local bindings* that are only visible in the current submodel. Properties are identifiers that are bound to values at the beginning of a run *and do not change* throughout the run. Here is an example properties segment:

```
init_x: cols * RANDOM(),
init_y: rows * RANDOM(),
population: 50,
```

Properties may also be created using Term components in which the *Property* box has been checked.

Local variable segment: Local variables are similar to properties, however their values may be changed during the run of the program. The format of their declaration is the same as that of properties:

```
u: 100,
v: 3.14,
```

In this case 100 and 3.14 are initial values for u and v, respectively.

Stock components generally play the role of local variables in model designs, however it can be convenient to define a few local variables to facilitate communication among interacting Capsules. *Local variable assignment should only occur in a post-update Command.*

Methods: Methods are functions local to the Capsule. An example definition would be

```
fact: function(n){if (n == 0) return 0; else return fact(n-1);},
```

Methods may refer to Capsule components, properties or local variables.

The role of method has been subsumed by the Codechip, and so it is only being included here for completeness. You should not need to define any.

5 Clocks and Clocked Chips

In Section 3.6.1 we discussed the need to associate a clock with each simulator. The clock's function is to keep track of model time and to sequence the cycle of strobes that update the simulation. The clock is programmed with values for start and end times, update interval (dt), and integration method. Most of the time a single system clock suffices, maintaining a uniform synchronized processing environment.

A chip designated as a *Clocked Chip* introduces a new clock with its own parameter settings for use with the chip's Capsule. Each strobe on the clock of the chip's container corresponds to a complete run of the chip's clock. The effect is to synchronously subdivide the container's update interval.

Clocked Chips are particularly useful for sensitivity analysis, and to help facilitate this *Nova* provides a *batch mode*, which creates the necessary Clocked Chip structure for repeated runs over sets of parameters. Clocked Chips and batch mode are illustrated in Section 12.1.

5.1 Clock object

It may be necessary to read clock parameters and actually perform clock operations from within the program. The most common example of clock access is the TIME primop, which returns the current model time on the current clock and within a Clocked Chip, the SUPERTIME() primop will do the same for the clock of the container. There are also primops SIMSTART, SIMEND and SIMMETHOD for

obtaining the other clock parameters.

Clock objects for the current and container clocks are returned using the CLOCK() and SUPERCLOCK() primops. Method calls supported by this object are described in Appendix C.2

6 Nova Aggregating Components

Abstraction is the process of extracting a set of interacting elements which together create a well-defined computation over a given a set of inputs, and providing the ability to access that computation with different inputs from multiple points within the overall project. The simplest example of this in *Nova* is the Capsule which is used in one or more chips to implement the instances of a particular submodel.

Chips are less useful when large numbers of submodels are required. In such cases it is more efficient to use some form of container to hold a set of Capsule elements. This is analogous to using arrays to manage large sets of data. Like the array, an organizing structure (i.e., the index set) is required to provide a uniform means of access to these constituents.

We can actually take this one step further by adding a set of properties and primitive operators that enforce a topological structure on the Capsules. For example, if we organize the Capsules into a two-dimensional lattice, each could represent a single cell in a cellular automaton. In order for this to be of any value, however, each cell must be able to identify its own coordinates and have some means of communicating with other cells in the lattice.

This is the role of *Nova*'s aggregating components: 1) organize and provide access to a (possibly large) set of constiuent Capsules; and 2) provide a set of properties and primops that foster information transfer among those constituents. Using this fundamental design the four aggregating components currently available with *Nova* provide different topological organizations for their elements. Here are brief descriptions (note: all example properties and primops are restricted):

CellMatrix Organizes its constiuent set into a 2 dimensional matrix or cells, with each assigned a row-column coordinate pair. Example property: `Coords`, which is bound to a Coord object[6] containing the row and column of the caller[7]. Example primop: `RING(n)`, which returns the array containing the coordinates of all neighbors n units away from the caller.

NodeNetwork Organizes its constiuent set into a 1-dimensional array of *nodes*. Each node is assigned a node number and contains a Capsule instance and a set of weighted pointers referencing other nodes. Example property: `myId`, which returns the caller's node number; Example primop: `CONNECTIONS_IN` returns a list of Objects, each containing the id and weight of a connection to this node.

AgentVector Organizes its constiuent set into a 1-dimensional array of *agents*. Each agents is assigned

[6] A *Coord object* is one containing the properties `row` and `col`.

[7] by *caller* we mean that Capsule instance that is making the reference.

an agent id and contains a Capsule instance. Each agent is equipped to record its location as a pair of x-y coordinates. Example property: `myId`, which returns the caller's agent id; Example primop: `MOVE(x, y)` sets the current location of its caller to (x, y).

SimWorld Combines the topological spaces created by the CellMatrix and AgentVector aggregators into one that maps the (x, y) location of each agent to position within a cell. In this topology each CellMatrix is the size of a unit square, and so an agent with x-y coordinates (x, y) is mapped to the cell with row-column coordinates (r, c), where $r = \text{Math.floor}(y)$ and $c = \text{Math.floor}(x)$[8]. Example property: `rows` is bound to the number of rows in the underlying CellMatrix (can be called from either a cell or an agent). Example primop: `MYAGENTS()` returns the list of agents currently within the calling cell.

7 Using CellMatrices

We are now ready to build a model using a CellMatrix. We will use the Firespread model found in the Model Library as our example, however our implementation will differ slightly from the one in the Model Library. Recall that this model views each cell either as a tree or firewall. Trees can be in one of 3 possible states: unburned, burning or burned. The firewall is represented as the fourth cell state.

At each iteration each tree cell determines its next state based on the state of its 8 (Moore[9]) neighbors and are summarized with the following rules:

1. If the current cell is unburned, determine if any of its neighbors are burning. If so, then the current cell will randomly either burn or remain unburned;
2. If the current cell is burning, it becomes burned;
3. If the current cell is burned, it remains burned
4. A cell that is part of the firewall remains unchanged.

This model has a main level and one submodel called *Treecell*. The latter will be the Capsule type used to populate the Forest CellMatrix. Create this submodel by clicking the New Sub Model button on the toolbar. Save the project as Firespread, so that the main model has that name.

7.1 Creating the cell submodel

Next we will develop the Treecell submodel, which represents a single tree. We'll let the 4 possible cell states be represented by the integers 0 through 3. It is convenient to assign these values as constants

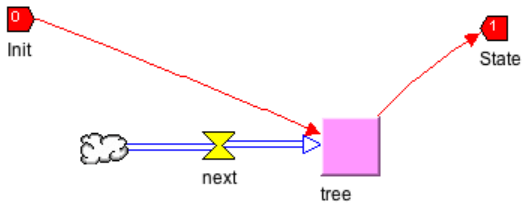
[8] Math.floor maps a real number to the largest integer less than that number; e.g., Math.floor(3.5) = 3.

[9] Moore neighborhoods consist of the 8 immediate neighbors of a cell in a cartesian lattice; the Von Neumann neighborhood only contains neighbors directly above, below, or to the left or right.

so that we can refer to them in a meaningful way. In the Programming Window insert the following code:

```
const unburned = 0,
      burning = 1,
      burned = 2,
      firewall = 3;
```

Drag a Stock called `tree` onto the canvas. This Stock will assume one of these 4 values throughout the simulation. Now drag a Flow called `next` and attach its output to `tree`. Right-click on `tree` and check the Discrete checkbox. Next drag an input pin called `Init` and an output pin called `State`, attaching them to `tree` as shown:



Right click on `tree` and make its initial value equal to `Input`; similarly make `State`'s value equal to `tree`.

It remains to program the logic that determines how `tree`'s state changes. This is complicated by the need to consult with the cell's neighbors to determine if any are burning. Consequently, for each cell we will use a Term called `neighbors` to hold an array consisting of state objects from that cell's immediate neighborhood. Fortunately a cell's neighborhood doesn't change, so we only need compute this array once, and so we can check the Property checkbox in `neighbors`[10].

Add the Term `neighbors` and check its Property box. The list of neighbors can be created a single line of code:

```
_.map(RING(1), function(coords){return CELL(coords);});
```

Let's take a moment to analyze this:

- The primop call `RING(1)` produces the list of coordinates for the 8 Moore neighbors of the calling cell.[11]
- The primop CELL takes a pair of coordinates and returns the state object of the cell at those coordinates.
- The primop `_.map` applies a function to each element of an array and returns the array of results. Example:

[10] Recall that by checking the Property checkbox in a Term it becomes a property; i.e., it computes its value only once.
[11] This version does not wrap; if the caller lies on the boundary it will have fewer neighbors.

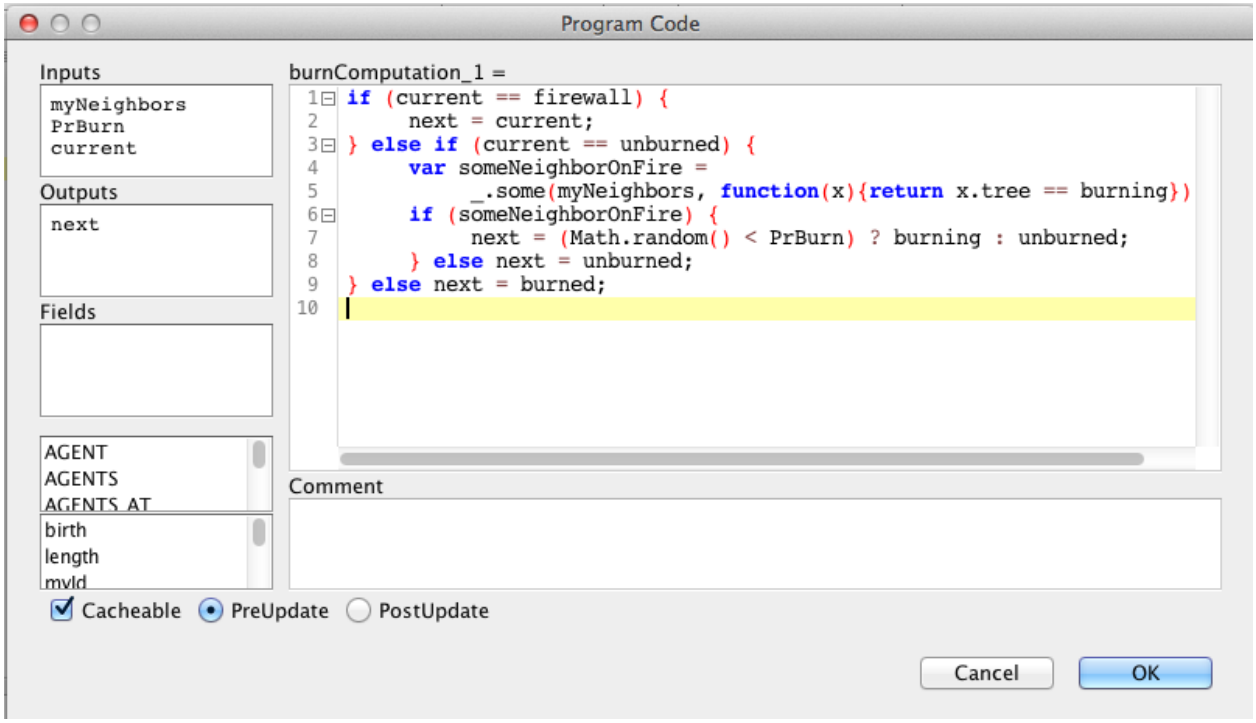


Figure 4: Codechip *burnComputation* Program Code

```
_.map([1,2,3,4,5], function(x) {return 2 * x;})
```

produces

```
[2,4,6,8,10]
```

Consequently, our use of `_.map` produces a list of state objects from the list of coordinate objects.

Now that we have this list of neighbors can program the state logic. Add a Codechip called `burnComputation` to the canvas and set up its inputs, outputs and code as shown in Figure 4.

Inputs `myNeighbors` and `PrBurn` will be connected respectively to the neighbors Property we just created, and to an input containing the probability of burning. `current` and `next` are the current and next states, respectively.

The logic makes use of the function `_.some`, which applies a function to each element of a list and returns true if at least one of the results is true. Here we're testing to see if any of the neighbor trees by checking the `burning` field of their state objects.

Add a datainput `PrBurn` and connect it, `neighbors` and `tree` to the appropriate pins on `burnComputation 1`. Initialize `PrBurn` to 0.3, which will be the default burn probability.

The completed Treecell should look like Figure 5. We will return to the main level to complete the program.

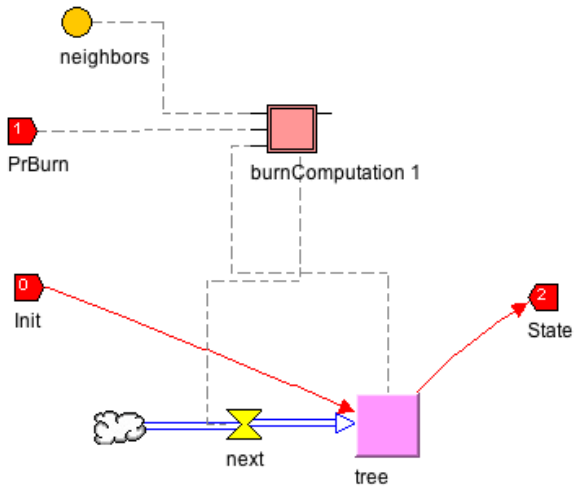


Figure 5: Completed Treecell Capsule

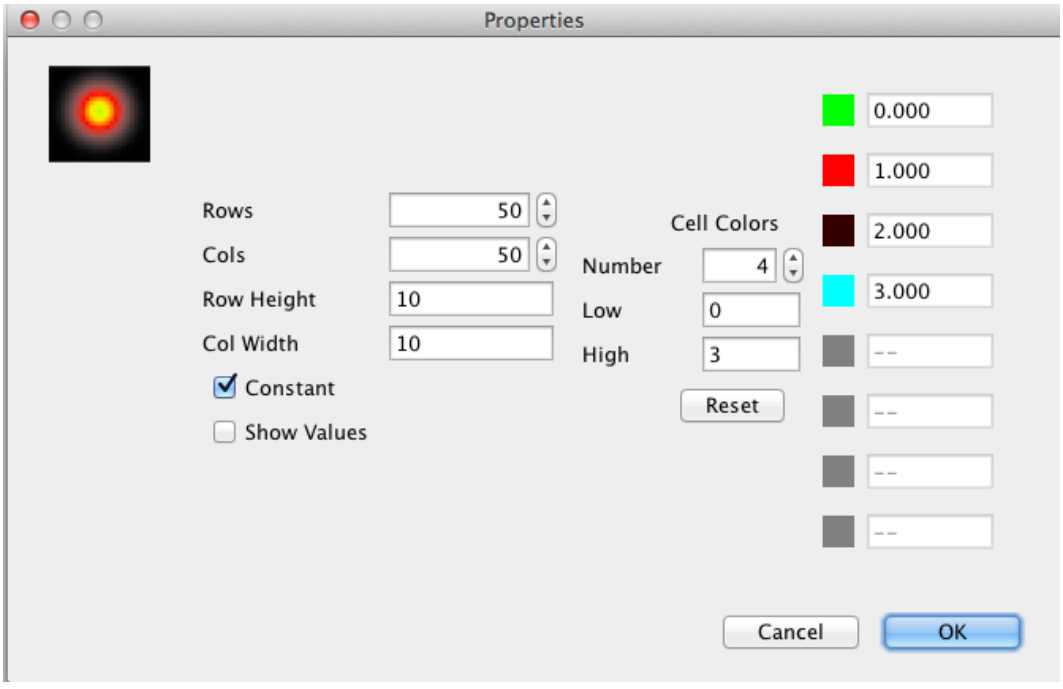


Figure 6: Forest Viewer Properties

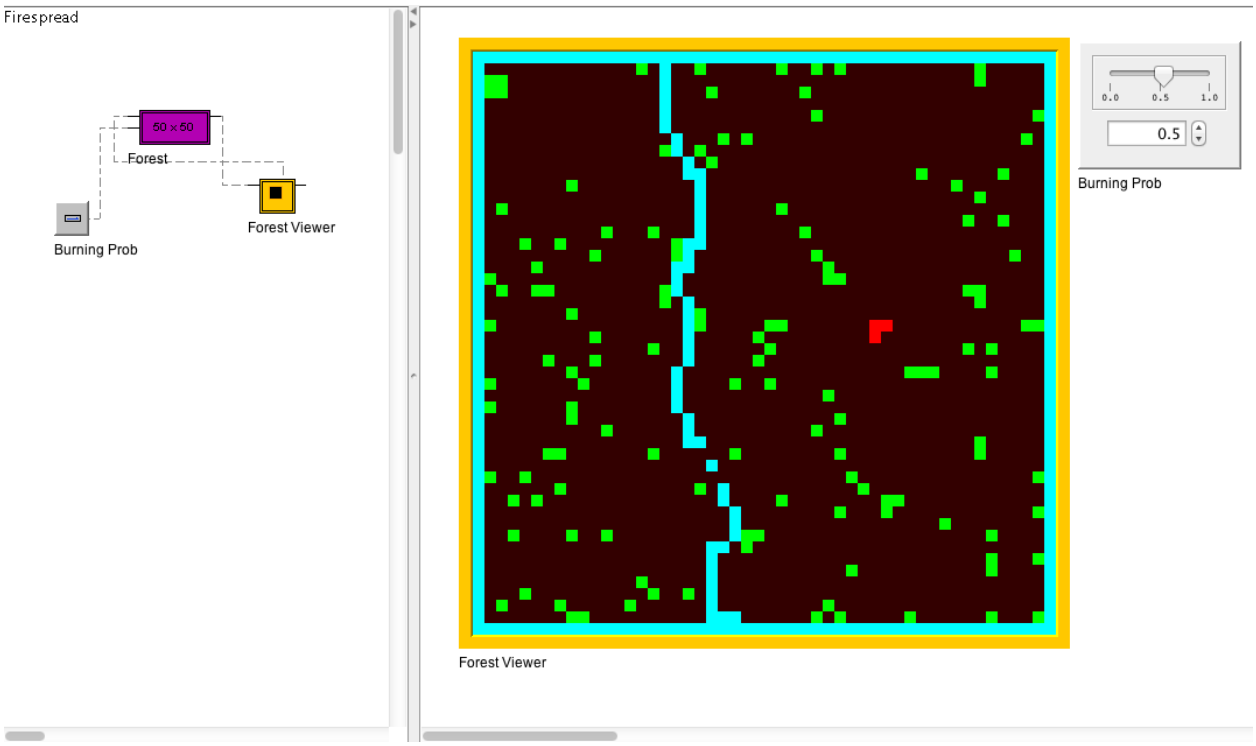


Figure 7: Final Version of the Main Capsule

7.2 Completing the Main Capsule.

To complete the program:

1. Add a CellMatrix called `Forest` from the component pallet to the canvas.
2. Set the number of rows and columns in `Forest` to 50.
3. Add a Raster called `Forest_Viewer` from the plug-in pallet to the canvas.
4. Connect the `State` output from `Forest` to the `In` input on `Forest_Viewer`.
5. Connect the `Out` output from `Forest_Viewer` to the `Init` input on `Forest`.
6. Initialize the properties of `Forest_Viewer` as in Figure 6.
7. Add a burn probability slider, and initialize your program as in Figure 7. It is now ready to be run.

********* To be completed; other topics may be added ********

8 Using AgentVectors

9 Using SimWorlds

10 Using NodeNetworks

11 Using the Console to Interact with *Nova*

12 Advanced Programming Techniques

12.1 Playing with the clock

13 Debugging

14 References

Appendix – Quick Reference

A JavaScript

A.1 Datatypes

Numbers: `3.1, 17, 2.78e2`

Strings: `"A string is text enclosed by quotes"`

Objects: `obj = {x: 0, y: "text"}`

`obj.x == 0, obj.y == "text"`

Each entry is called a *field* or *property-name/property-value* pair.

Arrays: `a = [6, "howdy", {x: 0, y:1}],`

`a[0] == 6, a[1] == "howdy", a[2] == {x:0, y:1}`

Functions: `f = function(x, y) {return 2 * x + y}`

A.2 Program structures

Loops:

```
var x = 0
for (var i = 0; i < 10; i++) {
    x = x + i;
}

var a = new Array();
...
for (var i in a) {
    a[i] = a[i] + 1;
}
```

Conditional Command

```
if (x > y) {
   z = x;
} else {
   z = y;
}
```

Conditional Expression

```
z = (x > y) ? x : y;
```

B Property and Primop Reference

In the following the arguments $s, s_1, ...$ denote string values, n, m denote integer values, and t, x, y, z denote real values. Additionally, *time* denotes current model time, and *dt* the current delta value.

Arguments enclosed by brackets are optional.

Some of these are deprecated, which means they will be dropped in future versions of Nova. In each case a replacement property/primop is provided.

B.1 Universals

These may be used anywhere.

Primops

`ALERT(s)`	Displays message s in a dialog box.
`BASEDIR()`	Returns the current model directory. *Available as of Release 12.*
`BINOMIAL(n, p)`	Returns a random number from the binomial distribution with n trials and success probability p.
`COS(x)`	Returns the trigonometric cosine of x. *Deprecated*: use `Math.cos`.
`COLUMNSPLIT(tab)`	*tab* is a 2-dimensional array derived from a table, where the first row contains column headers. Returns an object in which each property name is a column header with property value an array comprising the corresponding column.
`COSWAVE(x y)`	Returns: $x * cos(\frac{2\pi t}{y})$, where t is the current time.
`COUNT(f, a)`	f is a function of one argument that returns a boolean value and a is an array. Applies f to each element of a and returns the number of times f returns true.
`CLOCK()`	Returns the current clock as an object. Clock objects are discussed in Section 5. *Available as of Release 12.*
`CVSTOMAT(csv)`	*csv* is string consisting of a sequence of lines, each of which is a comma-separated string of values. Returns a matrix (i.e., 2-dimensional array) containing the values.
`DELAY(c, x, [y])`	Returns the value of component c delayed by x time units (i.e. at $time - x$). c is a string naming a Stock. Optional y is returned if the current time is less than x. If y is omitted then 0 is returned.

`DERIVN(c, n)`	c is a string naming a Stock or Term. Returns the value of the nth derivative of c at the current time, with precision based on the value of dt.
`DISTANCE(x0, y0, x1, y1)`	Returns Euclidean distance between points (x_0, y_0) and (x_1, y_1).
`DT()`	Returns the current delta (i.e., dt) value
`LOAD(l)`	l is a list of JavaScript or *NovaScript* filenames contained in the current model directory. Each is loaded into the runtime system. Should be part of simulation initialization.
`Math.XXX`	The JavaScript Math functions and constants; see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math`.
`NORMAL(x, y)`	Returns a random number from the normal distribution with mean x and standard deviation y.
`POISSON(lambda)`	Returns a random number from the Poisson distribution with density *lambda*.
`OPENREAD(filename)`	Opens *filename* for reading and returns a Java `BufferedReader` object. The latter contains methods `read` and `readLine` to perform input. See `http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html` for complete details. *Available as of Release 12.*
`OPENWRITE(filename)`	Opens *filename* for writing and returns a Java `PrintWriter` object. The latter contains methods `print` and `println` to perform output. See `http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html` for complete details. *Available as of Release 12.*
`PULSE(x, y, z)`	Returns x when current time is y, $y + z$, $y + 2 * z$, etc.; 0 otherwise
`RANDOM()`	Returns a uniformly distributed random number between 0 and 1. *Deprecated*: use `Math.random()`.
`READFILE(path)`	*path* is a string designating a textfile. if *path* begins with "/" it is treated as an absolute pathname; otherwise it is treated as relative to the current model directory. Returns the content of the file path as a string.

`RECONFIGURE(agg, map)` — *agg* must refer either to an AgentVector or SimWorld, and *map* must be an object with property names matching initializable components in the *agg* agent Capsule (i.e., properties or Stocks). Each property name must be bound to an array of values, with each array at least as long a the initial agent count in *agg*. `RECONFIGURE` reconfigures the initial agent vector in *agg* by rebinding each of the properties named in *map*, matching the *agg* agent vector to the values in the array bound to that property in *map*. *Available as of Release 12.*

`ROWSTOOBJS(tab)` — *tab* is a 2-dimensional array derived from a table, where the first row contains column headers. Returns an array of objects, one for each non-header row. In each object properties are column headers bound to the entry for that column in the corresponding row.

`SEED(x)` — Returns nothing; sets the seed of the random number generator. Should be part of simulation initialization.

`SIN(x)` — Returns the trigonometric sine of x. *Deprecated*: use `Math.cos`.

`SINWAVE(x, y)` — Returns: $x * sin(\frac{2\pi t}{y})$, where t is the current time.

`SIMEND()`
`SIMMETHOD()`
`SIMSTART()` — Returns simulation start time, end time and integration method, respectively.

`STEP(x, y)` — Returns the value of x if the current time is y or greater; 0 otherwise.

`TIME()` — Returns the current model time

`TRANSPOSE(mat)` — *mat* is a matrix (i.e. 2-dimensional array). Returns the transpose of *mat*.

`TOTAL(f, a)` — f is a function of one argument that returns a number and a is an array. Applies f to each element of a and returns the sum of values returned.

`UNIFORM(x, y)` — Returns a uniformly distributed random variable between x and y.

`_.XXX` — The `underscore.js` library of useful functions. See `http://underscorejs.org`.

B.1.1 Primops by category

Here is an index to the Universal primops according to a set of categorical types:

Control: `ALERT, COUNT, DELAY, PULSE, STEP, TOTAL`

Input/Output: `BASEDIR, LOAD, OPENREAD, OPENWRITE, READFILE`

Matrix: `COLUMNSPLIT, CSVTOMAT, ROWTOOBJS, TRANSPOSE`

Mathematical: `BINOMIAL, COS, COSWAVE, DERIVN, DISTANCE, Math.XXX, NORMAL, POISSON, RANDOM, SEED, SIN, SINWAVE, UNIFORM`

Simulation: `CLOCK, DT, SIMEND, SIMMETHOD, SIMSTART, TIME`

Misc: `RECONFIGURE, _.XXX`

B.2 Clocked Chip Primops

The following may only be used by a Capsule in a Clocked Chip.

`SUPERCLOCK()`	Returns the clock associated with the container of the Clocked Chip. *Available as of Release 12.*
`SUPERTIME()`	R!eturns the model time of the container of the Clocked Chip. *Available as of Release 12.*

B.3 Cell Properties and Primops

The following may only be used by Capsules contained in a CellMatrix, or cell Capsule in a SimWorld. `row` and `col` are integers representing row-column coordinates within the CellMatrix. `coords` is a Coords object (see Section C.1). *As of Release 12 all primops expecting coordinate pairs as parameters can either be called with separate row-column arguments or with a Coords object.*

Cartesian coordinates will generally be automatically *wrapped*; i.e. if the coordinate space has dimension $rows \times cols$, then a negative value for row or col is treated respectively as $rows + row$ and $cols + col$. Values that exceed the dimensions respectively become $row - rows$ and $col - cols$. This type of space is topologically equivalent to a *torus*.

Neighborhood function `BLOCK` and `RING` come in two versions, one that includes wrapping (`WBLOCK` and `WRING`), and one that treats the edges as boundaries (`BLOCK` and `RING`).

Properties

`coords`	Object whose `row` and `col` properties respectively reference the caller's row and column coordinates within the CellMatrix.
`rows` `cols`	Bound to the total number of rows and columns, respectively.

Primops

`BLOCK(n)` `WBLOCK(n)`	Returns a list (array) of coordinate objects (i.e., containing `row` and `col` properties) comprising the square block of cells n units away from the caller or less. (`WBLOCK` is the "wrapped" version, which treats the surface as a torus). *Available as of Release 12.*
`CELL(row, col)` `CELL(coords)`	Returns the state object for the cell at the given coordinates, which may be passed either as separate parameters or in a `coords` object (See Section 3.6.1 for a discussion of state objects.) The fields of the state object bind each of the Stocks, Terms and Flows in the cell to the current value of the corresponding component.
`CELLS()`	Returns the entire matrix (i.e., 2-dimensional array) of state objects for the CellMatrix. `CELLS()[row][col]` == `CELL(row, col)`.

`CELL_VALUE(row, col, comp)` `CELL_VALUE(coords, comp)`	*comp* is a String naming a Stock, Flow or Term in the cell Capsule. Returns the current value of that component in the cell with the given coordinates. `CELL_VALUE(row, col, comp)` == `CELL(row, col)[comp]`
`RING(n)` `WRING(n)`	Returns a list (array) of coordinate objects (i.e., containing `row` and `col` properties) for all cells comprising the square exactly n units away from the caller. (`WRING` is the "wrapped" version, which treats the surface as a torus). *Available as of Release 12.*
`WRAP(row, col)` `WRAP(coords)`	Performs a "wraparound" of the coordinates if they exceed the size of the matrix (or are negative). `WRAP(row, col)` Returns an array containing the new row and column; `WRAP(coords)` returns a Coords object containing the new row and column.

B.3.1 Hexagonal primops

The following should only be used if the cell matrix is in hexagonal mode.

`HEXRING(n)`	Returns a list (array) of coordinate objects (i.e., containing `row` and `col` properties) for all cells comprising the hexagon exactly n units away from the caller.
`HEXBLOCK(n)`	Returns a list (array) of coordinate objects (i.e., containing `row` and `col` properties) comprising the hexagonal block of cells n units away from the caller or less.
`HEXPATH(dir, dist)`	Returns a list of coordinates comprising the path of length *dist* in the direction indicated by *dir*. Directions are denoted by compass directions; i.e., `"N", "NE", "SE", "S", "SW", "NW"`.

B.4 Agent Properties and Primops

The following may only be used by Capsules contained in an AgentVector. `id` is an integer representing agent id of an agent.

Properties

`myId`	Bound to the caller's agent id.
`birth`	Bound to the time at which the caller was created.
`rows` `cols`	Bound to the total number of rows and columns in the space occupied by this AgentVector.

Primops

`AGE(id)` `MYAGE()`	Returns the age (i.e., time since birth) of agent *id*, or of the caller, respectively.
`AGENT(id)`	Returns a state object for agent *id*.
`AGENTS_AT(row, col)` `AGENTS_AT(coords)`	Returns the list of agents located at specified cell coordinates.
`AGENT_COUNT()`	Returns the current total number of agents.
`AGENT_IDS()`	Returns the array of ids for currently living agents.
`AGENT_VALUE(id, comp)`	*comp* is a String naming a Stock, Flow or Term in the agent Capsule. Returns the current value of that component in agent *id*. `AGENT_VALUE(id, comp)` == `AGENT(id)[comp]`
`AGENTS()`	Returns the array of agent state objects.
`CELL_COORDS(id)`	Returns the `row`, `col` cell coordinates (of *id*, or of the caller if *id* is omitted) in an object.
`CREATE([init], [n])`	Schedules the creation of a n new agents (n is assumed to be 1 if omitted). *init* is an initializer object containing bindings for properties in the new agent. These may include `init_x`, `init_y` and `init_heading`, to indicate the new agent's initial position and direction. It may also include alternate initialization expressions for Stocks, local variables and properties. If *init* is omitted, the new agent is a clone of the caller. All new agents are created at the end of the cycle.
`KILL(id)`	Schedules the elimination of agent *id*. All agent eliminations are carried out at the end of the cycle.

`LOCATION(id)`	Returns the cartesian location (of *id* or the caller if *id* is omitted) in an object with properties x and y. Also includes the heading as the value of property *theta*.
`MOVE(x, y)`	Moves the caller to cartesian coordinates (x, y).
`SET_HEADING(theta)`	Sets the directional heading of the caller to *theta* (given in radians).
`WRAP(row, col)` `WRAP(coords)`	Identical to CellMatrix `WRAP`

B.5 SimWorld Properties and Primops

Cell and agent Capsules in a SimWorld may respectively use the CellMatrix and AgentVector properties and primops described above.

The following may only be used by Capsules contained in a SimWorld.

B.5.1 Cell Capsules in a SimWorld

Cell Capsules may call the following agent primops:

`AGENT`

`AGENTS_AT`

`AGENT_COUNT`

`AGENT_IDS`

`AGENT_VALUE`

`AGENTS`

`CREATE`

`KILL`

Cell Capsules may call the following additional primops:

`MYAGENTS()`	Returns the list of agents currently contained in the caller.
`MYAGENT_COUNT()`	Returns the number of agents currently contained in the caller.

B.5.2 Agent Capsules in a SimWorld

Agent Capsules may call the following cell primops:

`CELL`

`CELLS`

`CELL_VALUE`

Agent Capsules may call the following additional primop:

`MYCELL()`	Returns the state object of the cell containing the caller.

B.5.3 Hexagonal primops

The following should only be used by an agent if the cell component of the SimWorld is in hexagonal mode.

`HEXMOVE(dist, dir)` Moves the calling agent distance *dist* in the direction *dir*. Directions are denoted by compass directions; i.e., `"N"`, `"NE"`, `"SE"`, `"S"`, `"SW"`, `"NW"`.

B.6 NodeNetwork Properties and Primops

The following may only be used by Capsules contained in a NodeNetwork. `id` is an integer representing node id of a node.

Properties

`myId`	Bound to the caller's node id.
`count`	Bound to the number of nodes in the NodeNetwork.

Primops

`CONNECTIONS_IN([id])`	Returns the array of connections into the node *id* (if *id* is omitted, it is assumed to be the caller). Each connection is an object with 3 properties: *id*, the node id of the source; *strength*, the raw strength of the connection; and *n_strength*, the normalized strength of the connection, where the total strength of all connections into the caller is 1.
`CONNECTIONS_OUT([id])`	Returns the array of connections from the node *id* (if *id* is omitted, it is assumed to be the caller). Each connection is an object with 3 properties: *id*, the node id of the target; *strength*, the raw strength of the connection; and *n_strength*, the normalized strength of the connection, where the total strength of all connections from the caller is 1.
`NODE(id)`	Returns a state object for node *id*.
`NODE_COUNT()`	Returns the total number of nodes.
`NODE_VALUE(id, comp)`	*comp* is a String naming a Stock, Flow or Term in the node Capsule. Returns the current value of that component in node *id*. `NODE_VALUE(id, comp)` == `NODE(id)[comp]`
`NODES()`	Returns the array of node state objects.
`INFLOW([id])`	Returns the total strength of connections into node *js* (if *id* is omitted, it is assumed to be the caller).
`OUTFLOW([id])`	Returns the total strength of connections from node *js* (if *id* is omitted, it is assumed to be the caller).

C *NovaScript* Objects

The properties and methods of special *NovaScript* objects are described below.

C.1 Coords object

Contains a pair of matrix coordinates for a cell (usually in a CellMatrix).

Properties

`row`	the row value of the cell
`col`	the column value of the cell

C.2 Clock object

This object provides properties and methods for managing a simulation clock. It is most useful in Clocked Chips for getting information about clocks other than one associated with the chip (retrieved using `SUPERCLOCK()`). The system clock is bound to the universal property `$clock$`.

The user should seldom need to refer to this object directly, as there are universal methods for retrieving clock values. It is provided primarily for Clocked Chips that need to probe their super clocks.

Properties

`high, low, dt, method, current`: clock parameters.

D Glossary

A

active level The capsule of a model project currently selected in the Capsule Set pane and displayed in the Model Canvas.

AgentVector An aggregating component that manages its members as agents moving over a cartesian or hexagonal plane.

aggregating component Refers to CellMatrices, AgentVectors, SimWorlds and NodeNetworks. Members of aggregating components must be Capsules.

B

base component Components such as Stocks, Terms, Flows, Commands, Codechips, etc., that do not have sub-components. They can only be members of Capsules.

C

Capsule Prototype for a simulation unit. Capsules contain interacting base and aggregating components, and chips, and may contain inputs and outputs.

capsule set The window of the Application Interface where the capsules of a model are listed.

capture A button that converts the visual representation of a Nova model into a script.

CellMatrix An aggregating component that creates a two-dimensional cartesian or hexagonal topology with its members.

cellular automaton A type of spatially explicit model where space is represented as a two-dimensional finite grid and each cell has a discrete state.

Chip A "wrapper" component which contains a single Capsule for membership in a parent Capsule.

Clock A special object for maintaining model time and providing strobe signals to the components.

Clocked Chip A Chip with which a new Clock has been associated. Each strobe on the Chip produces a complete run of the enclosed Capsule instance based on the parameters of the associated Clock.

Codechip A programmable component with user-specified inputs and outputs.

Command A *Nova* component containing executable code that changes the state of the program.

component expression One or more lines of code included as a component property that defines the value of that component.

console The window of the Application Interface where you can enter NovaScript commands one at a time.

container Simulator of which a component is a member (i.e., if A is the container of B, then B is a member of A). Also called a *parent*.

converter plug-in A plug-in used to compute values used in updating the current state.

Coords Refers to a *JavaScript* object that contains fields `row` and `col`, representing matrix row and column values.

D

delta value The amount of time between state updates; also called *dt*.

deterministic model A model where the outcome is fully predictable from the initial state (i.e., no random effects).

display plug-in A plug-in used only for visualization.

dynamic systems model A model of a system that changes over time.

E

Euler Method A method of numeric integration that estimates P(t) as P(t-t) + P(t-t)t, where t is the change in time. Pronounced "Oiler method".

F

field A property-name/property-value pair in a *JavaScript* object. Also refers to a stateful variable in a CodeChip.

G

global segment Section of the Programming Window containing global definitions.

I

identifier A text string (beginning with a letter) used as a property or local variable.

integration method Procedure used to iterate from t to $t + dt$ when considering continuous functions.

iteration Step from time t to $t + dt$ in the simulation.

L

local variable An identifier used as a variable within a specific Capsule instance.

local variable segment Section of the Programming Window containing local variables.

M

member Constituents of simulators.

method An object field that contains a function.

method segment Section of the Programming Window containing local methods.

model canvas The window of the Application Interface where the model is graphically designed and built from components.

model time Local simulation time in units determined by the model.

Monte-Carlo A model involving an element of chance (i.e., randomness).

N

NodeNetwork An aggregating component that creates a network (i.e. mathematical graph) topology in which its members are nodes.

NovaScript A scripting language that was created specifically for designing and running models. NovaScript is an extension of JavaScript.

P

parent Another name for a component's container.

plug-in An extension to the basic component functionality.

post-processing (post-process) Actions required during the post-update phase.

post-update Actions taken after the current state is changed during an iteration.

pre-update Actions taken before the current state is changed during an iteration.

primop Short for *primitive operator*; a built-in *JavaScript* or *NovaScript* function.

Programming Window The section of the *Nova* interface in which the user may add code.

project All of the capsules, functions, clock settings, etc. associated with a model. When you open Nova, you are working on a project.

property An identifier whose value is fixed throughout the simulation.

property segment Section of the Programming Window containing local properties.

R

RunData A special *NovaScript* object that contains all of the output from a Stock during a complete run. Used to accumulate statistics.

Runge-Kutta 2 Method A method of numeric integration that employs a correction to each Euler method estimate.

Runge-Kutta 4 Method A method of numeric integration, where each approximation is weighted average of four estimates.

S

scenario *NovaScript* object used for defining a component. Acts like a class declaration for *NovaScript* objects.

self state object binding referencing the component object.

Self Pointer to the state object of a simulator.

simulation A sequence of state transitions from a start time to an end time using a fixed time increment, dt.

simulator Capsules, CellMatrices, AgentVectors SimWorlds and NodeNetworks, all of which have constitutent members.

SimWorld An aggregating component containing a CellMatrix and AgentVector, in which the CellMatrix serves as the cartesian or hexagonal space in which the AgentVector's agents exist.

start time Point in model time when the simulation starts (usually 0).

state object A special object referenced from *Self* in a simulator. For Capsules, it contains the current value for each member; for aggregating components it provides methods for obtaining the state object of members.

stateful component A component that keeps track of its value over time (e.g., Stock).

stateful plug-in A plug-in with state-values that persist between iterations.

stateless component A component that is only aware of its current value.

stochastic model A model that exhibits random effects.

strobe Action taken by each component at each iteration.

super component object binding referencing an object's container.

Super Pointer to the state object of the container of a simulator.

U

underscore.js A library of very useful functions included in *NovaScript*; see `http://underscorejs.org`.

W

wrap The practice of treating coordinates outside the dimension of a cartesian space as continuing from the opposite boundary. The resulting space becomes a torus topologically.