Cellular Automata: Introduction to NetLogo

Disclaimer: To get you oriented to the NetLogo platform, I've put together an in-depth step-by-step walkthrough of a NetLogo simulation and the development environment in which it is presented. For those of you with significant programming experience under your belt, this walkthrough may be a bit of overkill. For those of you without significant programming experience, it may still be a bit of overkill, considering that StarLogo and NetLogo are frequently used by junior-high school students; i.e., the interface and language are very intuitive. Nonetheless, the detail is there for those who may need it. For those of you who wish to read less carefully (i.e., feel they can learn fine by exploring on their own), I've highlighted sections that you really should not skip in blue, such as "exercises" that need to be answered.

Getting to Know NetLogo: All About Ants

NetLogo is a *development environment*. For you non-computer science people, that means that it is a program that allows you to write code, debug code, translate that code into instructions the computer understands (compile and interpret), and run the instructions (link and execute). In other words, it allows you to do all the steps necessary to write and execute a computer program in one convenient software package. As already discussed, it is also a programming language. To use NetLogo effectively, therefore, we need to understand both the software and the language. Let's start with the software.

First we should assume that NetLogo is installed on your machine. However, it may not be. If not, download an installer from the link provided on the class website and install it. NetLogo will run on any platform (Windows, Mac, Linux).

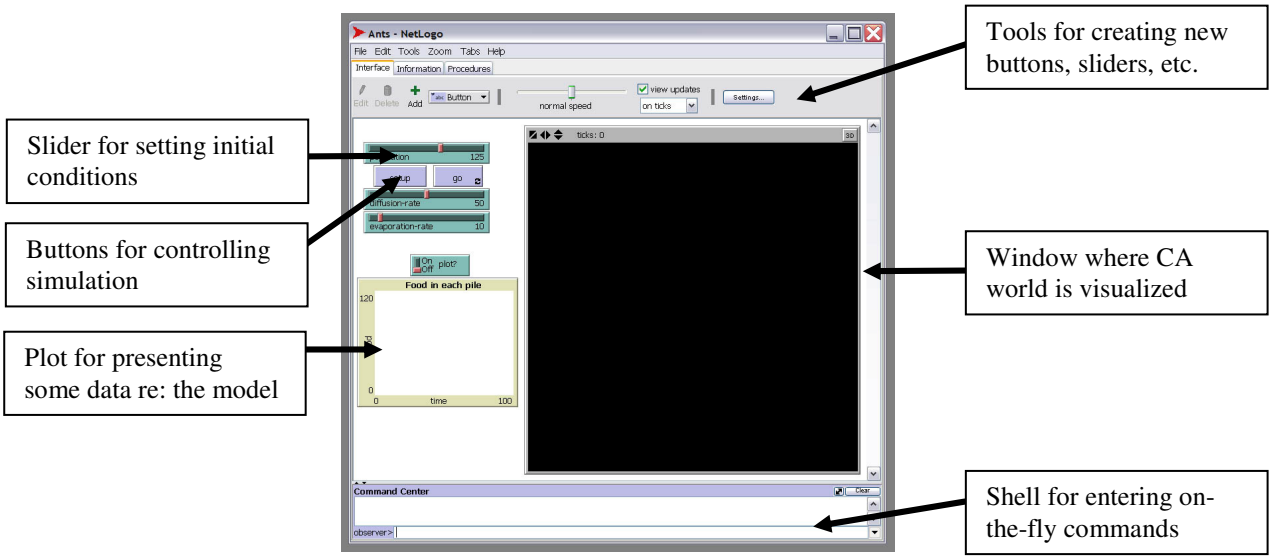
Good, now launch the program!

Welcome to the NetLogo environment! When you load the software, it presents you with an interface screen (i.e., **Interface** tab is selected and on top). The interface screen allows you to 1) design and modify the interface for a simulation using a nice Graphical User Interface (GUI), 2) interact with a simulation that is already written and view it behavior, and 3) use the interpreter aspect of NetLogo to run on-the-fly commands. We will look at all of these options in a bit. For now, let's continue to see what else NetLogo has to offer.

Click on the **Information** tab. Doing this brings the following to the front an editable help file. If you are running a simulation written by someone else, it should contain a description of the model and instructions on its use. If you have written a simulation, it is your responsibility to edit this file so that others are informed of the purpose and use of your simulation. Now select the **Procedures** tab. It is this window that you will write and edit NetLogo code.

Enough chit-chat; let's get started! From the **File Menu**, select **Models Library**. This brings up a library of all the pre-existing simulations that have been written in NetLogo and incorporated in the software distribution. As you can see, there are quite a few and

we will explore many of them over the next couple of weeks. For now, browse to the **Biology** folder and select the **Ants** simulation. Then select **Open** to load the simulation into NetLogo. When loaded NetLogo switches back to the Interface window, which as you can see, is now filled with stuff:



This stuff includes: sliders for setting the initial conditions (e.g., population size, rate at which pheromones diffuse or evaporate, and even an on/off "switch" for the output plot), buttons that allow you to setup or run the simulation, and a plot that presents some data concerning the behavior of the simulation, as well as the other aspects of the interface window (tools for adding other controls and the interpreter for on-the-fly interaction).

Exercise 1: Play with Ants!!! Press the **setup** button. What happens? Then press the **go** button. What happens? Given these initial conditions and the behavior of the system, to which class of cellular automata do you think this model belongs? Press the **go** button to stop the simulation. Change the initial conditions by using the sliders. Press the setup button to **setup** the model with the new conditions and then run again. Do you still like your classification choice?

Exercise 2: So, what is this a simulation of? Press the **Information** tab to read a bit about the system this simulation is modeling. Now that you understand the system, go back to the Interface tab and play with the model some more. Turn on the plot. What insight does the plot give you into the behavior of the system? Is this system emergent. Why or why not?

Just from running this little simulation, I hope you can see that NetLogo is a pretty cool development environment that lets you play with cellular automata models in a very sophisticated manner (I mean, there were actually *ants* running around on your computer screen!!!). Certainly, this simulation is a lot more interesting than Conway's Game of Life! And, as you saw when you looked at the model's library, there are many, many simulations that we could happily spend hours playing with. But...we are here to learn some biology and some computer science, which means we need to understand not only

the output of the simulation, but how the simulation was created in the first place. So, let's explore a little further and look at the code for the Ant simulation by switching to the **Procedures** tab.

Below are segments from the code for the Ant simulation, cut and paste directly from the original and then annotated to help you read through and understand what is going on. The color scheme is also the same:

```
patches-own [
  chemical             ;; amount of chemical on this patch
  food                 ;; amount of food on this patch (0, 1, or 2)
  nest?                ;; true on nest patches, false elsewhere
  nest-scent           ;; number that is higher closer to the nest
  food-source-number   ;; number (1, 2, or 3) to identify the food sources
]
```

A comment

The ant program starts with a block of code that defines the variables that all patches can use. In other words, this code bit establishes the characteristics of a patch for the model, where each variable saves an aspect of the patch's state (e.g., a patch could have 5 g of chemical, 1 food, no nest, 0.32 nest scent and a food source of 3). We know this because it begins with the word `patches-own`. This word is a ***reserve word***, or a phrase that is an integral part of the NetLogo programming language. The reserve word is followed by square braces ([]), and any code statements in those braces are affiliated with the word. There is also a reserve word, `turtles-own`, that can be used to define any variables associated with turtles. Both `patches-own` and `turtles-own` must be used first; i.e., when necessary, these definitions are the first things to appear in a NetLogo program.

Also worthy of note is the text followed by double semi-colons (`;;`) or comments. Comments are annotations to source code that help explain the purpose and workings. Anything followed by the `;;` to the end of the line is ignored when a program is built from code.

Let's look at the next code block:

```
to setup
  clear-all
  set-default-shape turtles "bug"
  crt population
  [ set size 2         ;; easier to see
    set color red  ]   ;; red = not carrying food
  setup-patches
  do-plotting
end
```

Here we see two more reserve words: `to` and `end`. These two words mark the start and end of a command procedure or **function** (i.e., a block of code that performs a specific task). In general, the syntax for a command procedure is:

```
to procedure-name
      do something
            :
      do something else
```

```
end
```

The particular procedure illustrated on the previous page is standard and most NetLogo simulations will have one. This is the **setup** procedure and is the list of commands that happen when the **setup** button is pressed on the interface. This setup procedure does the following:

1. Clears the display world of all patches and turtles (`clear-all`).
2. Establishes how turtles will be drawn (`set-default-shape turtles "bug"`)
3. Creates turtles, which belong to the breed "population" (`crt`). The settings between the square braces following the `crt` statement establish what "population" turtles look like.
4. The setup function then calls two function written by the programmer, one the sets up the patches and the other that does the plotting. If you scroll down through the program, you will see that these functions are defined in the same way as was the setup procedure.

This setup function has introduced us to many NetLogo commands. Commands are words that perform specific tasks in NetLogo (i.e., built-in functions). NetLogo `commands` appear in blue. In addition to commands we can also see that there are other words that are colored. These include `values` (in red) and standard NetLogo `identifiers` and `operators` (in purple).

The standard **setup** procedure is followed by three more user-defined setup functions: setup-patches, setup-nest, and setup-food. If you look carefully, you will see that setup-patches initializes the world so that some patches are the ants' nest and some their food. It then recolors the patches (using the user-defined recolor-patch function) to reflect whether a patch contains food or is the nest. All of this patch setting-up happens in the **setup** function, when the setup-patch command is invoked. ***Take home message***: you can define a procedure (or function or block of code that completes a task) using the keywords `to` and `end` and the syntax introduced earlier. Once you define a procedure, you can ***invoke*** it (i.e., cause it to be executed) by just typing the **`procedure-name`** elsewhere in your code.

Once setup, the next step is to actually run the simulation. This is done with another standard, user-defined function: **go**. Below is the go procedure from the Ants simulation:

```
to go  ;; forever button
  ask turtles
  [ if who >= ticks [ stop ] ;; delay initial departure
    ifelse color = red
    [ look-for-food  ]       ;; not carrying food? look for it
    [ return-to-nest ]       ;; carrying food? take it back to nest
    wiggle
    fd 1 ]
  diffuse chemical (diffusion-rate / 100)
  ask patches
  [ set chemical chemical * (100 - evaporation-rate) / 100
    recolor-patch ]
  tick
  do-plotting
end
```

This **go** procedure does the following:

1. Asks the turtles to perform all the commands in between the square braces that follow the `ask turtles` command, including looking for food, returning to the nest, wiggling, and moving forward one patch.
2. Tells each patch to share some of its chemical (see the patch definition) with all eight of its neighboring patches (`diffuse`)
3. Asks all patches to update their amount of chemical.
4. Update the tick-counter (`tick`; a way of keeping track of time).
5. Plot the change in the system on the plot.

After the **go** function is a whole bunch of other code...all of which are additional procedures that accomplish or help accomplish the tasks outlined in the **go** procedure, which is the meat of the simulation.

Now, I could, in excruciating detail, work through the remainder of the code for the Ant simulation...but I am not going to. Mainly because there are a lot of built-in commands and a lot of details that can't be simply explained in one sitting. Instead, I suggest you read through the code on your own. I think you will find it to be mostly intuitive. Discuss it with your neighbor if you get confused.

After you are fairly comfortable or fairly frustrated by the Ant simulation, I'd like you to do the following:

Exercise 3: Familiarize yourself with NetLogo by doing Tutorial's 1-3 in the NetLogo User's Manual (found through the Help Menu).
Exercise 4: Now that you are a NetLogo expert, go back to the Ant simulation. Let's go out on a limb and change the model! Alter something....it could be something big or something little. Well, not too little. Changing the forward statement in the go procedure from `fd 1` to `fd 2` is TOO little. The goal is to change the actual system being modeled by modifying one of the procedures that govern its behavior (i.e., one of those functions following the **go** procedure). Now run your newly adapted model. Is the effect of your tampering visible? How so? Explain your modification and e-mail me a copy of your program.