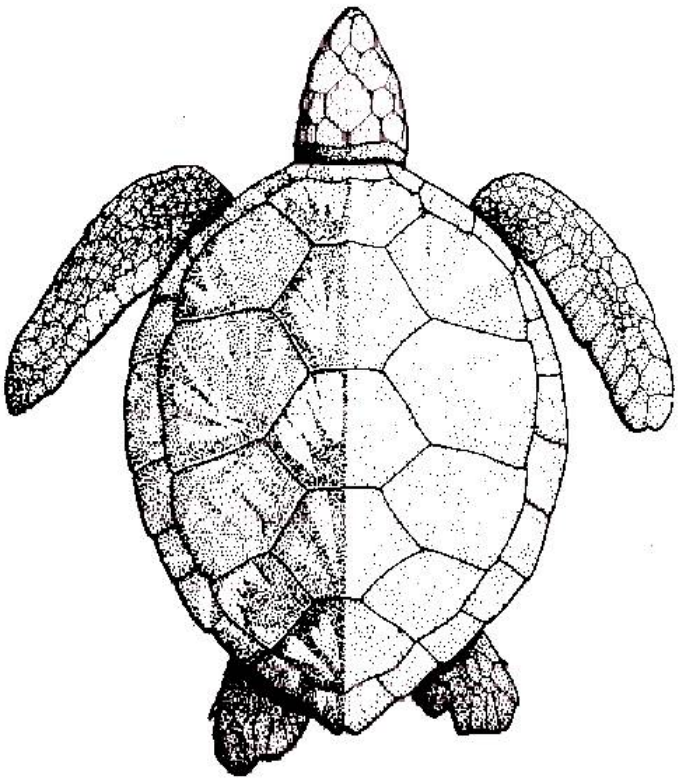
A Field Guide to

NetLogo



Steve Scott and Matt Koehler
George Mason University
Department of Computational Social Science

About the Cover:
The green sea turtle (*Chelonia mydas*) image on the cover was created by Jack Javech of the National Oceanographic and Atmospheric Administration (NOAA). NOAA's National Marine Fisheries Service (NMFS) and the U.S. Fish and Wildlife Service (FWS) work together to protect sea turtles, which are affected by numerous threats including fisheries bycatch, habitat destruction and degradation, marine pollution, and boat strikes. The NMFS, FWS, and numerous partners are actively involved in sea turtle conservation efforts, such as monitoring sea turtle nesting activity, protecting turtle nests and nesting beach habitat, and implementing and enforcing regulations to reduce bycatch of sea turtles.

Visit the NMFS website at http://www.nmfs.noaa.gov/pr/species/turtles/ and the FWS website at http://www.fws.gov/northflorida/SeaTurtles/seaturtle-info.htm for further information about sea turtles and conservation efforts.

Table of Contents

Preface

About This Guide

This guide has been developed for graduate students at George Mason University in the Department of Computational Social Science who need to develop agent-based models but are not familiar with NetLogo modeling and language syntax. This is not intended to be a comprehensive review of the NetLogo environment and language, nor to supersede the existing NetLogo documentation, but rather is intended as a teaching aid for students developing their first agent-based models. In much the spirit of a field guide from the natural sciences, this guide is intended to help the reader quickly get oriented and point out key features of what may be an initially unfamiliar landscape.

About NetLogo

NetLogo is a freely available agent-based modeling environment developed and maintained by the Center for Connected Learning (CCL) at Northwestern University. The website at http://ccl.northwestern.edu/netlogo/ describes NetLogo as "*a multiagent programming environment… used by tens of thousands of students, teachers, and researchers worldwide*." The website contains numerous resources for NetLogo model builders and is the definitive source for information about NetLogo.

Prerequisites

Readers of this guide are assumed to have some basic technical computing skills. Software development or agent-based modeling background is helpful, although not required.

Font Conventions

This guide uses the following typesetting conventions:

Italic
For emphasizing new terms when introduced, and for referencing names of NetLogo functions or procedures in the narrative text.

`Constant Width`
Used for source code, commands, and model output.

About the Programs in This Guide

The programs and program fragments included in this guide should be considered illustrations of concepts rather than examples of industrial-strength software.

The NetLogo environment includes an extensive built-in library of models spanning domains including mathematics, biology, social science, network science, and computer science, to name a few. In addition to these domain-specific models, the NetLogo library includes a number of coding examples to illustrate specific features of the NetLogo environment or programming language syntax. The models in the models library and the coding examples are excellent resources for learning about NetLogo programming.

How to Suggest Improvements

Suggestions for improvements, corrections, and general recommendations are welcome. The best way to make contact is via email:

Steve Scott
sscotta@gmu.edu

Matt Koehler
mkoehler@gmu.edu

Correspondence can be addressed to:
George Mason University
Department of Computational Social Science
Research Hall, Suite 300
Fairfax VA 22330

Other NetLogo References

NetLogo includes an online help system accessed via the Help tab on the main menu. This includes a link to the **NetLogo User's Manual**, which is a comprehensive reference of over 430 pages, covering in detail the NetLogo programming language, the NetLogo modeling environment, and additional NetLogo packages such as mathematical arrays and tables, sound, robotics, and Geographic Information System extensions. (Although accessible from within the NetLogo environment, the manual is also available online in HTML and PDF form, and can be accessed at http://ccl.northwestern.edu/netlogo/docs/NetLogo%20User%20Manual.pdf)

In addition to the NetLogo User's Manual, the Help tab includes a link to the **NetLogo Dictionary**, an online tool providing help on specific NetLogo functions and capabilities.

The **NetLogo Resources and Links** page at http://ccl.northwestern.edu/netlogo/resources.shtml contains a variety of information about NetLogo, including links to NetLogo developer and user community resources, model libraries, information about NetLogo extensions, software tools, books, research papers, and NetLogo tutorials.

Several books are available either online or from commercial publishers focusing on how to develop agent based models in NetLogo.

> Marco Janssen. 2010. Games and Gossip. Available online at http://www.openabm.org/book/1928/games-gossip.
>
> Steven F. Railsback and Volker Grimm. 2011. Agent-Based and Individual-Based Modeling: A Practical Introduction. Princeton University Press.
>
> Uri Wilensky and William Rand. In work. An Introduction to Agent-based Modeling: Modeling Natural, Social and Engineered Complex Systems with NetLogo. Publisher TBD (MIT Press).

An outstanding **Quick-Reference Guide for NetLogo Programming** syntax has been developed by Luis Izquierdo. The guide is available at http://luis.izqui.org/resources/NetLogo-5-0-QuickGuide.pdf . Students may wish to print this reference guide on heavy stock paper and laminate in plastic for convenient use while developing models.

Acknowledgements

The authors would like to thank members of the faculty and graduate students at George Mason University in the Department of Computational Social Science and the School of Public Policy for their encouragement, helpful suggestions, and recommendations for the improvement of this text. In particular, Jessica Hughes and William Kosmann provided detailed reviews of the draft manuscript and NetLogo source code examples.

NetLogo Versioning

This guide was developed under NetLogo 4.7 (and updated to NetLogo 5.1), however, NetLogo is an actively evolving product and new versions are being released on a regular basis. The technical discussions and code examples are all believed to be accurate under version 5.0 or greater; however, it is possible that there may be some incompatibilities. Where such issues are known, they will be called out in the text.

1. Getting Started

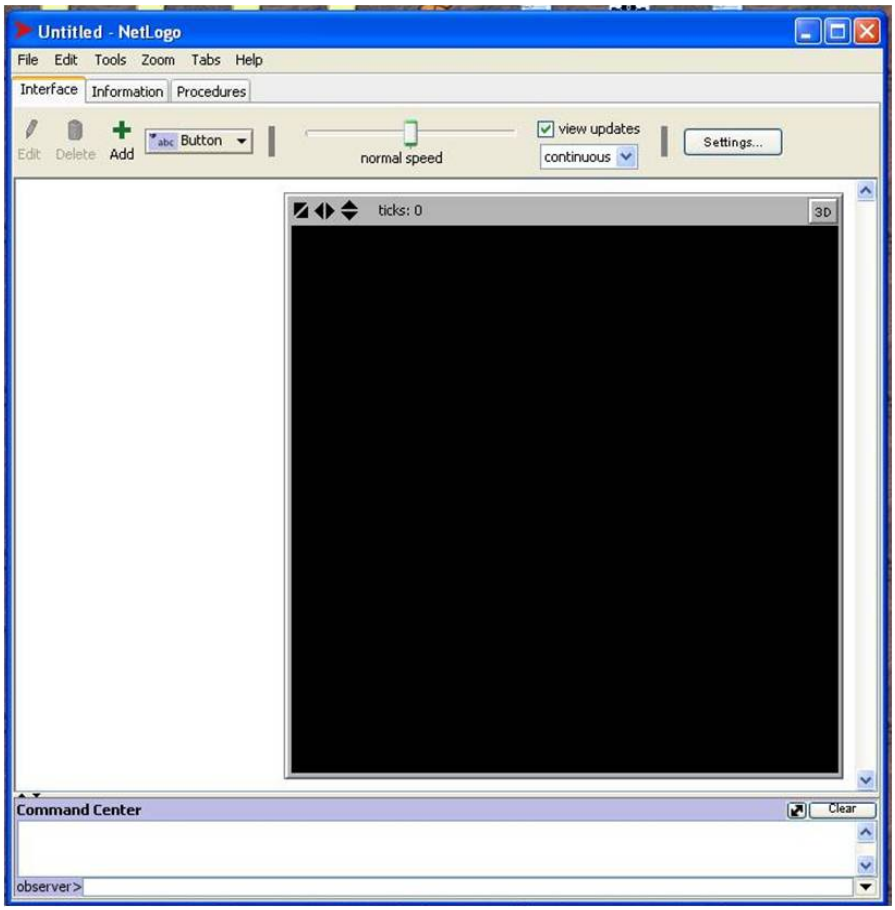
The first thing we need to do is download the latest copy of NetLogo from the home site at the Center for Connected Learning at Northwestern University http://ccl.northwestern.edu/netlogo/ . Click on *Download* to obtain the latest version of the software. While we're there, check out the site, as there are a variety of sample models and other resources for NetLogo fans.

Installing NetLogo on a Windows machine is fairly straightforward. The Windows installer supplied should take care of everything, and just using the defaults settings works fine. (All the examples in this guide were developed on a Windows environment – Mac and Linux users should not experience any major differences in functionality, but the screen layouts may not look exactly as depicted here.)

Running NetLogo

Let's get started – use either the *Run->Programs->NetLogo* option, or double click on the NetLogo icon if you've installed a desktop icon. As NetLogo is coming up, a brief splash bar display should appear looking like this.

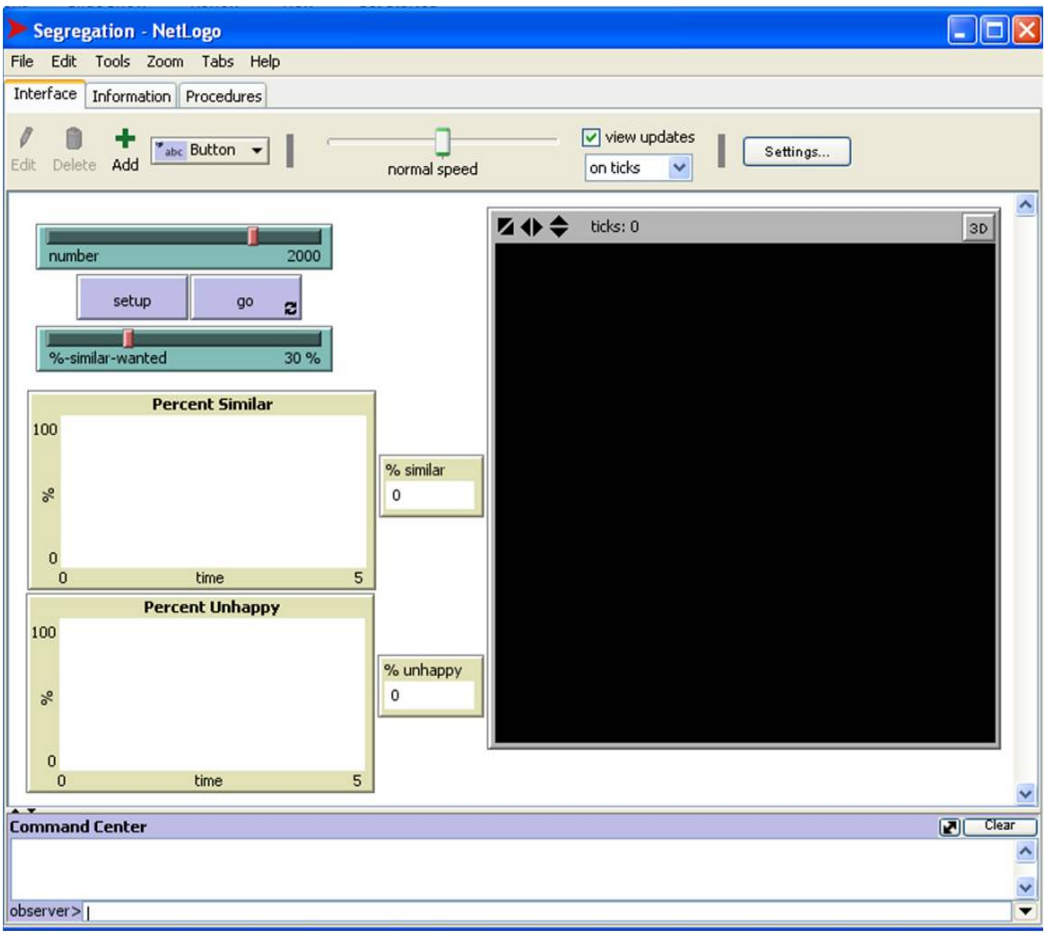
Within a few seconds, the main NetLogo window should come up looking like this.



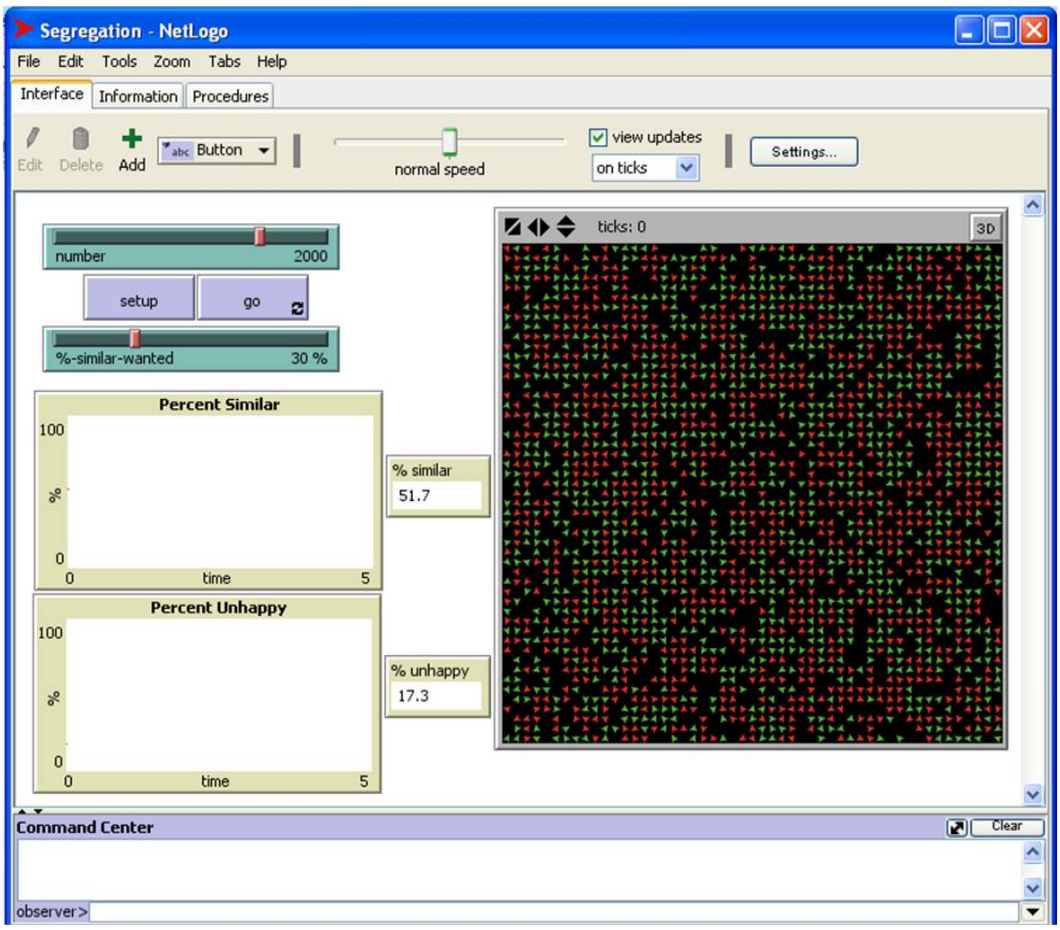
Running a NetLogo Model from the Library

Before delving into the technical details of building a NetLogo model, let's actually run a preexisting model from the built-in NetLogo models library just to get a feel for how the environment works and how we could use an agent-based model. We'll start with a classic model from social science, originally developed by Nobel Prize-winning economist Thomas Schelling in the 1960s to explore segregation preferences in urban housing.

Go to *File->Models Library*. Under the Social Science folder, select *Segregation Model*, then press the *Open* button. The system should load up the model and look like this.



There are three main tabs to a NetLogo model: Interface, Information, and Procedures. We'll cover them all eventually, but let's start here with the Interface tab. Press the *setup* button, and the model is initialized with a population of 2000 agents randomly distributed around the landscape. The model should look like this (note that your map layout and the percentages for *% similar* and *% unhappy* may vary since the model uses random values to establish the makeup and location of the initial population).



For starters, observe that we've got two different types of agents: reds and greens. We've also got some empty space on the map where there are no agents, indicated by the black background. The *number* slider controls how many agents get created when we press the *setup* button – the default was 2000. The *%-similar-wanted* slider controls the preference for being around agents of the same type: the higher this value, the more the agent wants to be nearby similar agents.

The *%-similar* value is calculated by looking at every agent and determining what percentage of agents in its immediate neighborhood of adjacent cells is of similar type. For many agent-based models, this concept of neighborhood adjacency is very important: in fact, there are specific terms for sets of adjacent neighbors. The set of four neighbors in the cardinal compass directions North, South, East, and West are called the *Von Neumann neighborhood*, and the set of all eight possible adjacent neighbors (N, S, E, W, NE, NW, SE, SW) are called the *Moore neighborhood*. The Schelling model uses the Moore neighborhood for adjacency.

The *%-unhappy* value is calculated by looking at every agent and determining if it is surrounded by the *%-similar-wanted* or a higher percentage of agents of the same type. Thus if a green agent is surrounded by a Moore neighborhood consisting of 4 reds and 4 greens, its similarity with its neighbors would be 4/8 or 50%. So in this case because the *%-similar-wanted* slider value is set to 30%, the green agent is above this value and therefore will be "happy." Conversely, if a red agent is surrounded by a Moore neighborhood of 7 greens and 1 red, its

similarity to its neighbors would be 1/8 or 12.5%. In this case, because the *%-similar-wanted* slider value is set to 30%, the red agent is below this value and thus will be "unhappy."

In the Schelling segregation model, agents calculate their happiness value based on similarity to their neighbors. If they are happy, they stay at their current location, but if they are unhappy, they move to a random open location. That's all there is to this model – there is no central planning authority to control where agents move. Despite this seemingly simplistic behavior, some surprising things can happen if we tweak the *%-similar-wanted* slider. Let's try it.

Using the default settings for the model, press the *go* button and watch what happens. The model should run briefly, then stop with the agents slightly clustered on the map, no unhappy agents, and about 70-ish % agents similar. Try this a couple of times: press *setup*, then press *go*.

Now move the *%-similar-wanted* slider up to 50%, press *setup* and then *go*. Notice that the map shows more pronounced clustering. Repeat this a couple of times as well, noting that while the final map layout varies from run to run, the *% similar* and *% unhappy* values don't change that much from run to run.

Now move the *%-similar-wanted* slider up to 75%, press *setup* and *go*. Notice that the model takes a lot longer to run, and the final map shows very pronounced segregation with large areas of similar type agents clustering together. As before, repeat a couple of times to see the variation in final layout and similarity of final percentages.

Finally, move the *%-similar-wanted* slider up just 1% more, to 76%. Press the *setup* and *go* buttons. The model is churning, with agents moving all around and never settling down. With just a slight increase in *%-similar-wanted*, the system has entered an unstable state, and the agents will wander forever trying unsuccessfully to find neighborhoods with more than ¾ similar agents. Press the *go* button to stop the model. Notice that the *%-unhappy* is very high compared to previous runs. This is not at all intuitive – why would such a very small change in *%-similar-wanted* cause such a huge difference in model performance? This is an example of the often unexpected and interesting results one can observe in agent-based modeling.

To round out our tour, let's take a look at the other tabs on the model. Press the *Information tab* under the menu bar. This will bring up the online documentation about the model. You can scroll through and read about how the model was developed, key features, and things to try. Documenting a model is considered good form in the NetLogo modeling community, and you can see why: reading this built-in documentation provides a good overview of what the model is doing.

Now press the *Procedures tab*, and take a look at the actual NetLogo model code. Don't be too intimidated if it looks complicated – the language is actually pretty straightforward once you learn a few conventions. If you've had experience developing models in other programming languages, here's something to notice: NetLogo code is very compact. The entire model we just demonstrated above can be expressed in under 100 lines of code. Hopefully this quick tour has whetted your appetite for more, so let's move on.

Diving In: A Quick Hands-On Tour

Let's dive right in and get started – we'll begin by creating our first virtual agent-based world with a pod of green sea turtles swimming in a simulated ocean.

If you haven't already done so, start up NetLogo, and *Select File->New* to create a new model. The landscape should be empty, as we have no agents and no landscape to speak of at this time. Let's start by making an ocean for our turtles. Press the tab key twice to bring up the *patches>* prompt at the bottom of the screen, then type in the following command. (Don't worry too much if the commands seem cryptic: we'll explain them in more detail later on.)

```
set pcolor blue
```

This command asks all the patches in the model, which you can think of as grid cells on a map, to set their patch color (abbreviated as pcolor) to blue. This will make the entire landscape (OK, actually a "seascape") a nice shade of light blue. This may be perfectly fine, but as any good impressionist painter knows, the ocean is not all the same color. So let's fix this by making some shades of color to suggest waves and depth. To do this, we'll use a random number generator and ask patches to adjust their color slightly above and below the standard blue shade. Don't worry too much about the exact details, just type in these commands. Pay attention to the use of parentheses and brackets – NetLogo language syntax is sensitive to such things.

```
if ((random-float 1.0) < 0.25) [ set pcolor blue - 3 ]
if ((random-float 1.0) < 0.10) [ set pcolor blue + 2 ]
```

Now we should have a smattering of various shades of blue. The first command asks NetLogo to ask all the patches in our simulation to generate a random number between 0.0 and 1.0, and if the generated number is less than 0.25, change the patch color to a darker shade of blue. This results in about ¼ of our patches being shaded darker blue. Similarly, the second command asks for some of our patches to be colored a slightly lighter shade of blue. This results in about 1/10 of our patches being toned a little brighter.

This is better, but it still doesn't look much like an ocean. We'll smooth things out using the built-in *diffuse* command, a function designed to work like mathematical sandpaper to even out a bumpy surface. However, *diffuse* doesn't work in the *patches* context, so we'll need to press tab twice to get back to the *observer* context, then type in the following command.

```
diffuse pcolor 0.50
```

This command asks each patch to average out 50% of its `pcolor` value among its eight nearest neighbors (the Moore neighborhood). Now things are starting to get a little smoother. Let's repeat the previous command, but instead of typing, use the up arrow to have the system reenter the command for us. Hit enter, and the command is executed. Repeat this a few more times until you get the ocean looking reasonable.

We've got a good-looking ocean, but it's a little bit lonely out there, so let's add some turtles. At the Observer prompt, enter the following command to create 30 turtles.

```
create-turtles 30
```

NetLogo has created a pod of 30 turtles, but right now they are all located on top of one another in a blob in the middle of the ocean. That's because without telling NetLogo otherwise, the default location for a new turtle when it is created is at the origin, which is currently defined to be right in the middle of the landscape. Also, the turtles don't look much like turtles; in fact, they look like little triangles (again, because the default turtle shape in NetLogo is a triangle).

We can improve this with a few commands. At the *observer* prompt, hit the tab key to move over to the *turtles* context. Enter the following commands.

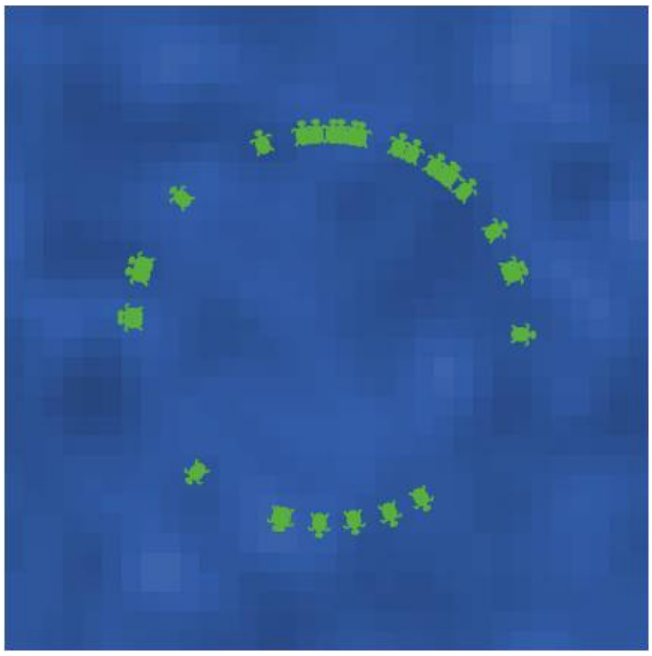
```
set shape "turtle"
set size 1.5
set color green
```

The first command changes the shape for our turtles from the default triangle to a more turtle-like shape, conveniently named "turtle". NetLogo has a wide variety of built-in shapes for agents (such as birds, cows, wolves, rabbits, people, cars, and airplanes, just to name a few), and you can also create your own shapes using your artistic talents and the handy Shape Editor under the Tools menu. The second command makes our turtles one and a half times the normal size so we can see them better. The third command makes our turtles turn green; since they are supposed to be green sea turtles, this seems appropriate.

It's still hard to see what's going on, so let's spread out our turtles and let them explore the ocean a bit. Type in the following commands (still at the *turtles*> context).

```
set heading (random 360)
forward 10
```

The first command points each of our turtles in some random direction between 0 and 360 degrees (which covers the entire span of the compass, so it's really quite random). The second command instructs our turtles to move forward 10 units. (Quick note: NetLogo is quite agnostic about scale, so if you care about things like miles versus kilometers or furlongs or fathoms, you'll need to figure out an appropriate spatial resolution for your model and be mindful of this as you develop agent behaviors). We should now have a ring of turtles, all facing away from the center of the seascape.



Now let's spread them around a bit. Type in these commands.

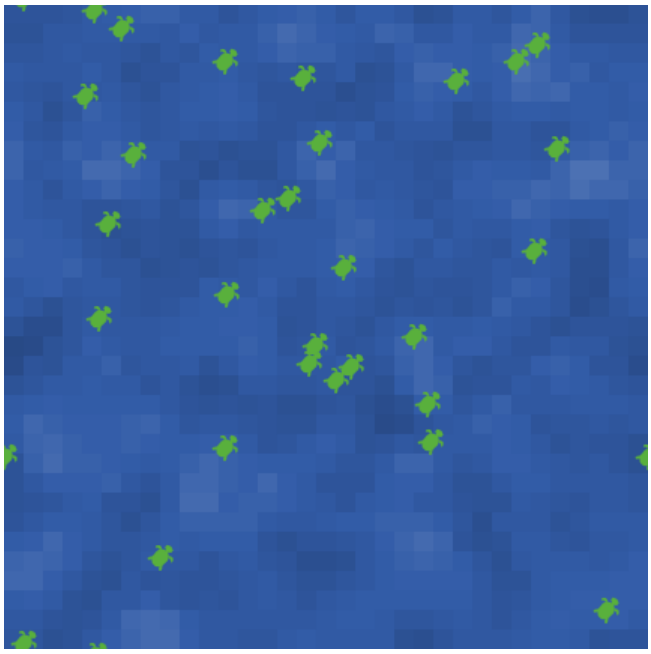
```
set heading (random 360)
forward (random 10)
```

The turtles should now be spread around the seascape fairly randomly. Now let's move them around some more. Type in the following command. (Note: if the command "fd" seems cryptic, it's just the abbreviated form for the NetLogo command "forward". Many NetLogo commands have shortened forms, and it's not at all uncommon to find abbreviations in the model code.)

```
fd (random 10)
```

Using the up arrow, you can recall previous commands and move the turtles around a bit more. To close out this demonstration, let's point all the turtles in a common direction and have them swim in formation.

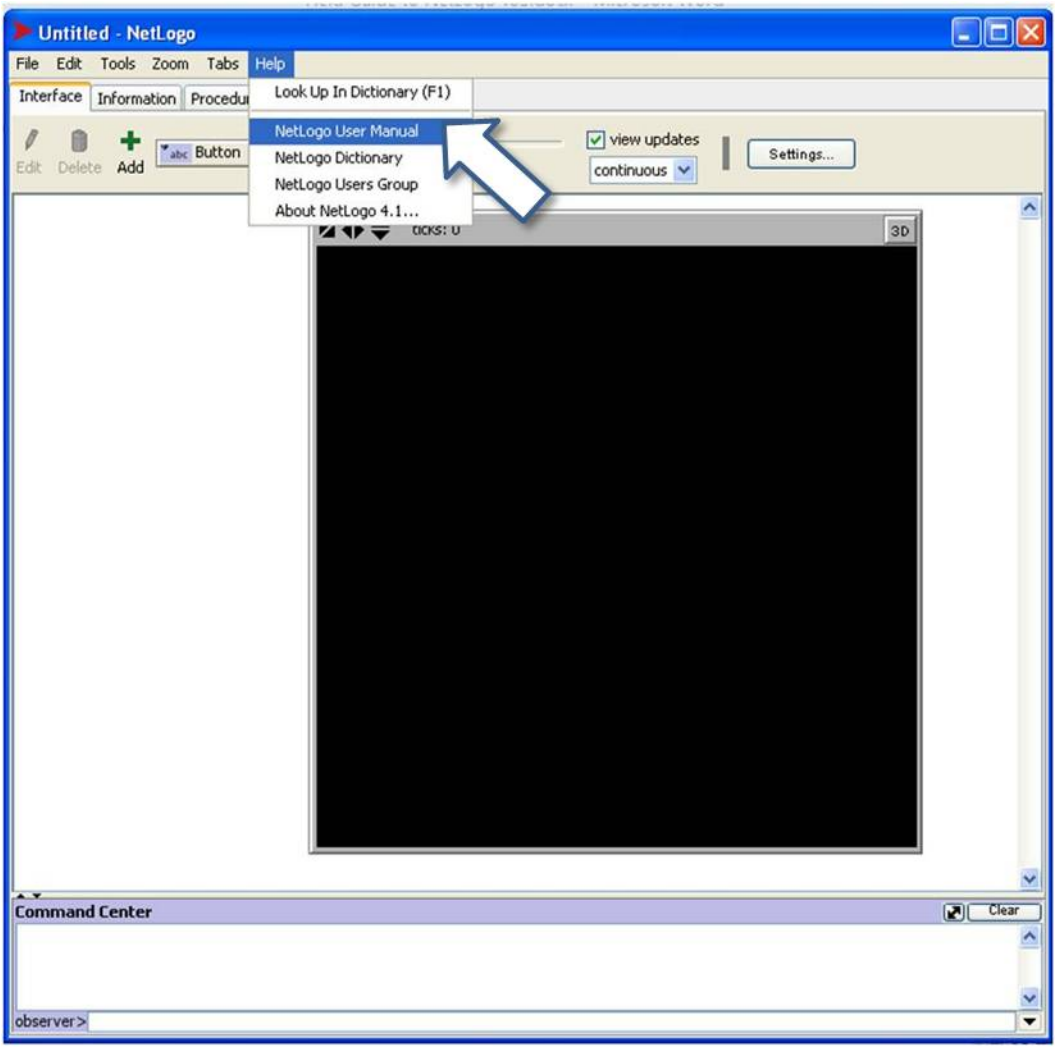
```
set heading 45
fd 1
fd 1
fd 1
```



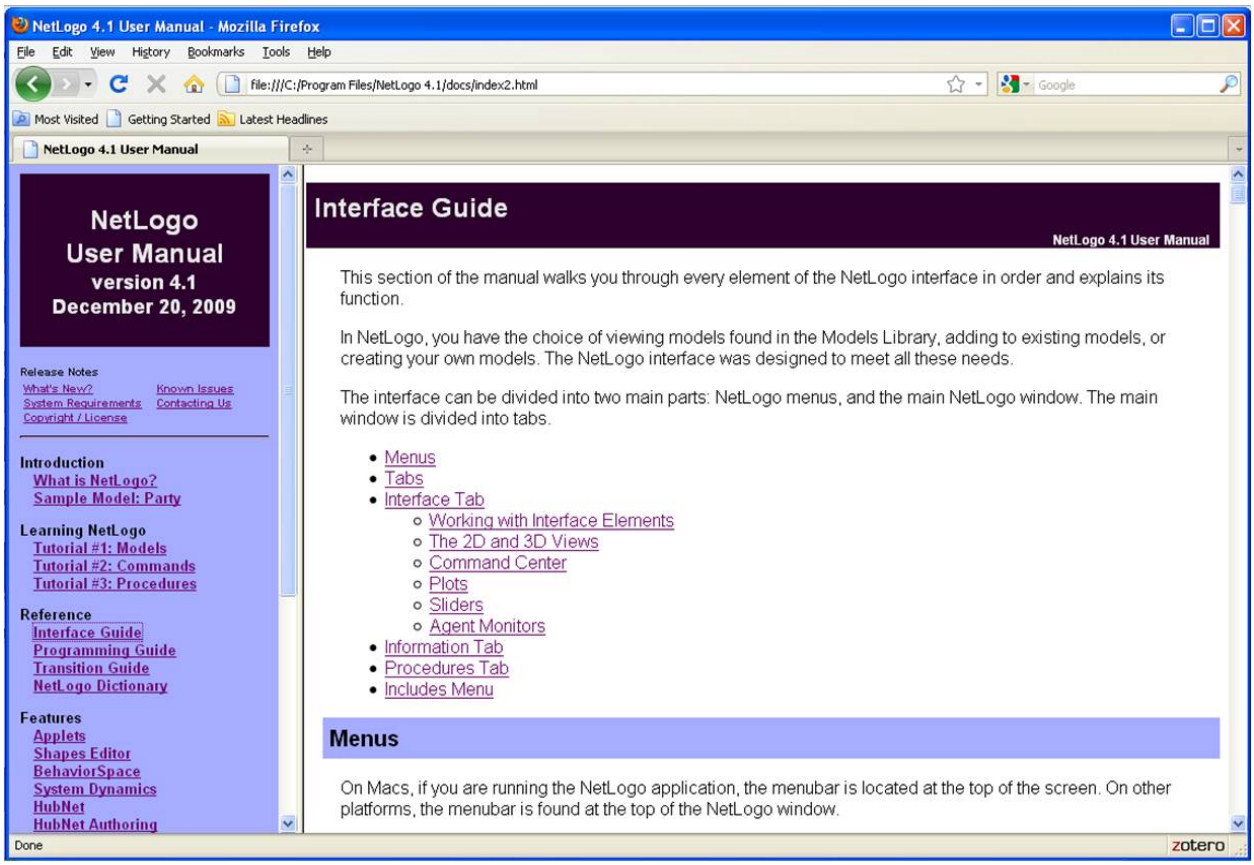
Repeat this until you or your turtles get tired of swimming.

A Guided Tour of the NetLogo Environment

There are a number of features of the NetLogo environment, and the best way to get up to speed on the environment is to use the online documentation. The following discussion is not intended to replace the existing information, but hopefully will help you get a handle on the various sections of the online documentation. To access the documentation, go to the *Menu* bar, select *Help*, and choose *NetLogo User Manual*.



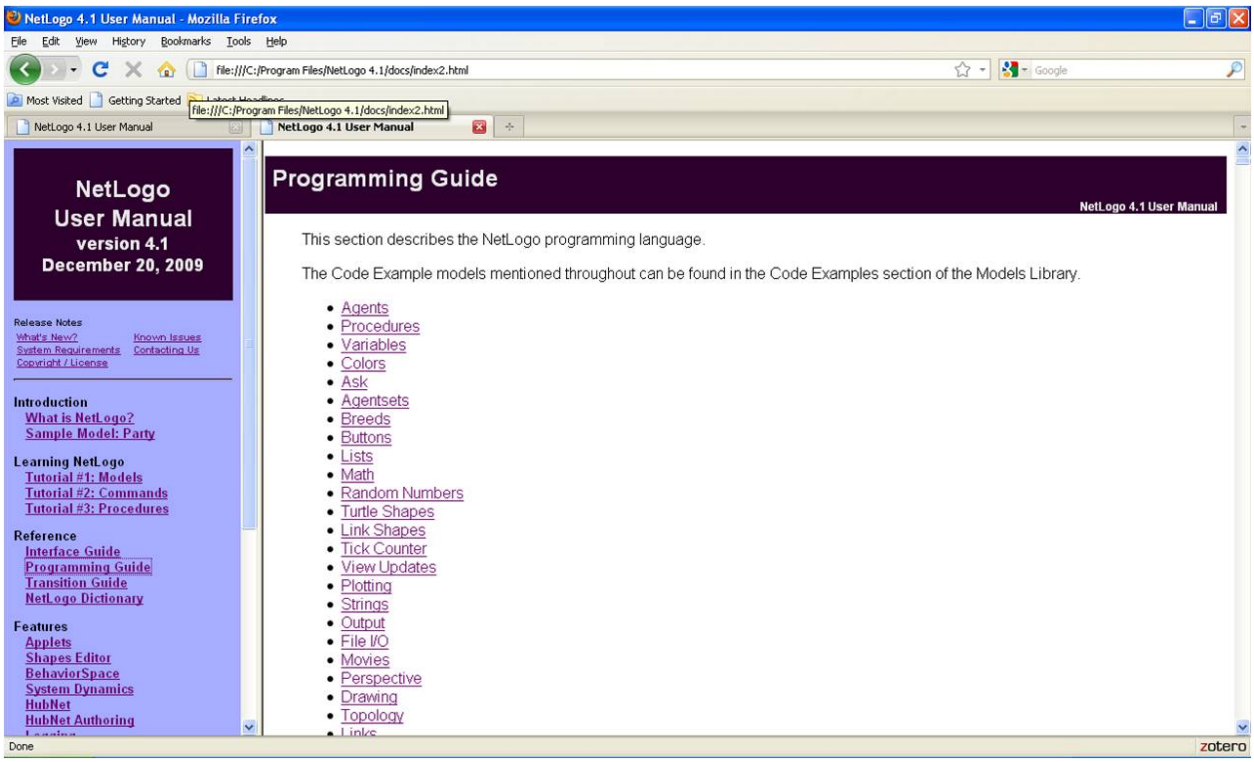
Once you've done this, the online User Manual documentation will come up in your default browser. Here's what it should look like.



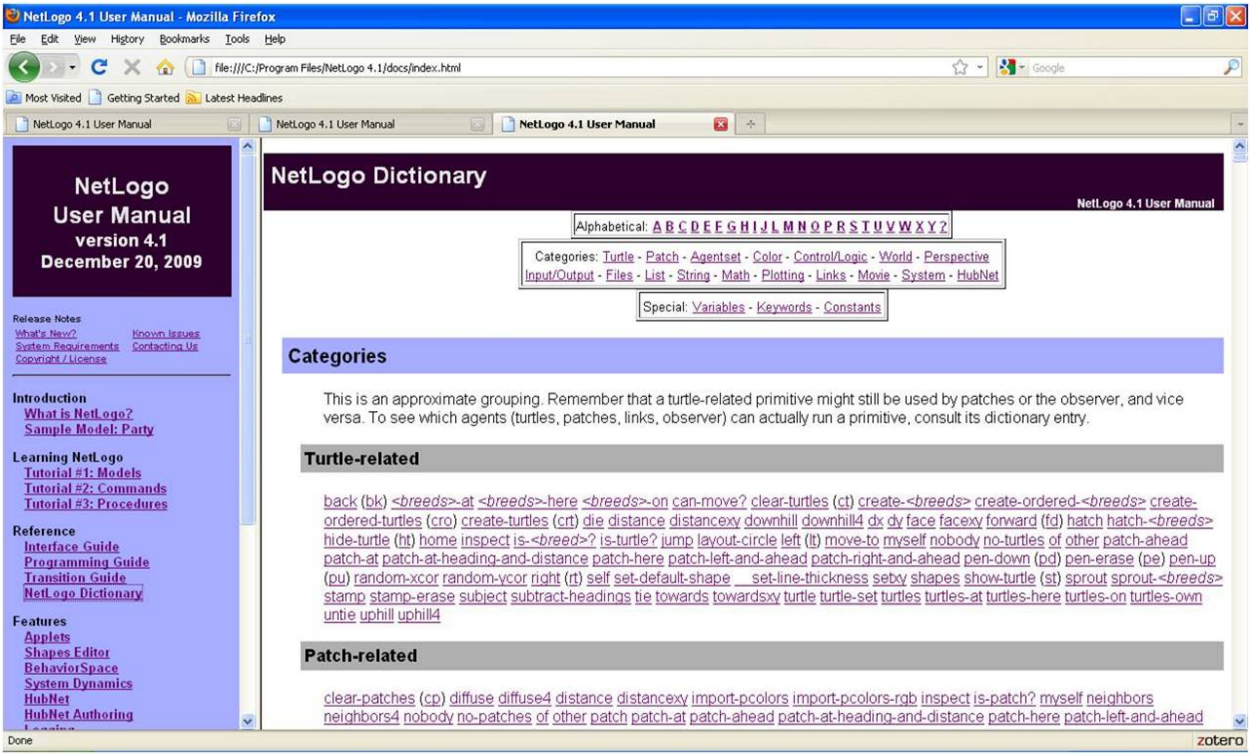
Notice that the Interface Guide comes up by default, which is the first item under the **Reference** section on the left menu area. Above this, the **Introduction** section includes materials to get you oriented on what NetLogo is and how it can be used for agent-based modeling. The **Learning NetLogo** section has three built-in tutorials to get you started: these are a good way to cover topics we just skimmed over above, and get you a more thorough survey of NetLogo capabilities. *Tutorial #1: Models* covers the basics of a model, introducing how to set up and run a model from the models library. *Tutorial #2: Commands* covers how to interact with a model from the command prompt. *Tutorial #3: Procedures* covers the basics of writing simple code procedures as well as interacting with interface components like sliders, monitors, and plots.

Under the **Reference** section, the **Interface Guide** covers all the functionality found in the menu bar, discusses the functions of the three main NetLogo tabs (Interface, Information, and Procedures), and also shows how to use components on the Interface tab to make your model more usable and interesting.

The **Programming Guide** is available by clicking on the Programming Guide on the left of this menu. This set of documentation provides details on the NetLogo programming language. Read through this to get a feel for the subject areas – you'll probably find most of the answers to your NetLogo programming questions in here.



The **NetLogo Dictionary** is available by clicking on the left menu panel, or also available by selecting *Help->NetLogo Dictionary* on the main NetLogo window. This will bring up an alphabetically and categorically organized set of online resources. The alphabetical listing goes from A-Z, covering all NetLogo functions. The categorical listing has functions related by topic area, such as Turtle-related functions, Patch-related functions, etc. Everything is hyperlinked, so it's easy to move around and find things.



Even if you're an experienced software developer, you should spend some time getting familiar with these online help resources.

Summary

We've seen how to obtain and install NetLogo, and we've demonstrated a classic agent-based model to get an idea of how these models work. We've experimented with using landscapes and agents via interactive commands. We've introduced the extensive NetLogo online help system, and hopefully you've had a chance to read through these materials and become somewhat familiar with what NetLogo is and how it works. Now let's consider how we might approach building and using an agent-based model.

2. Agent-Based Modeling Concepts

In the physical and social sciences, we generally seek to find some kind of mathematical or logical explanation for phenomena of interest, and then compare this abstraction to the data observed in the real world. The better the model explains the data, the better the model is.

In many scientific disciplines, mathematical models can provide a compact and abstract representation for events of interest. Simple linear models can be used to predict trends, systems of ordinary differential equations can be used to represent more complicated relationships, and nonlinear models can be used for inherently complex systems. All these approaches rely on the ability of the modeler to formulate the problem in terms of mathematical expressions.

Agent-based modeling is based on the idea that complex phenomena are often exhibited without some overarching control mechanism, but rather as the result of large numbers of interacting agents all doing relatively simple things. This concept is different from "top-down" mathematical modeling. Agent-based modeling relies on collections of purposeful agents, all interacting to achieve some goals, with inherent randomness and parallelism. This has some attractive features for a wide variety of problems in the social and physical sciences, offering a way to more directly model the phenomena of interest.

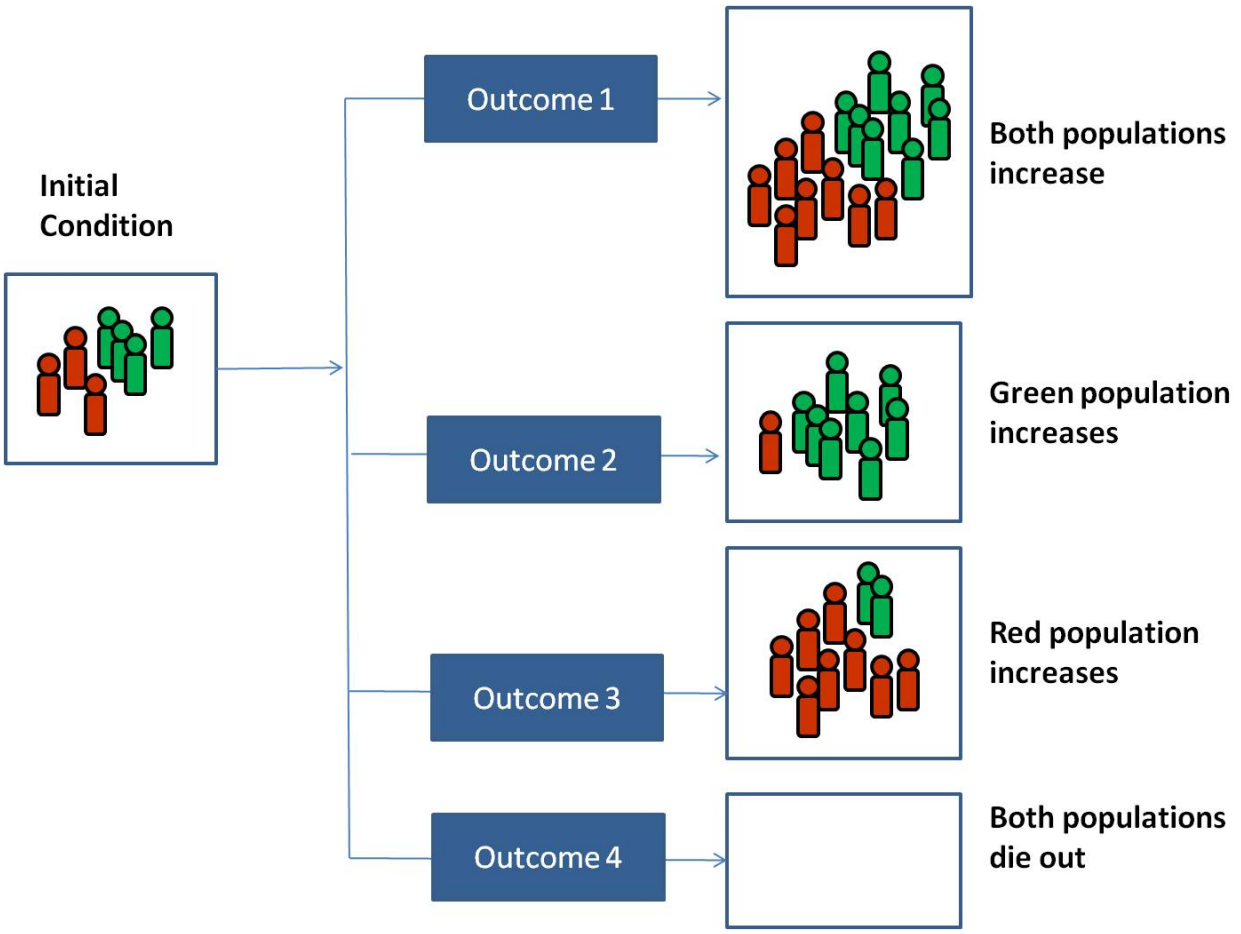
Agent-based modeling is derived from the disciplines of artificial intelligence, distributed computing, parallel computing, evolutionary computing, and object-oriented programming. An early example of the agent-based computational genre is the cellular automata (CA) model CA model, in which cells in a grid-like structure communicate with their immediate neighbors and perform simple computations. Despite this modest framework, surprisingly complex results can emerge.

More sophisticated agent-based models include mobile agents, the capability for complex behavior among agents, and also the capacity for interactions between agents and their environment. As we've already seen above, an early example of an agent-based model is the Schelling Segregation model, in which simple agents moving around a hypothetical urban environment can demonstrate results that mimic real housing segregation patterns seen in large cities.

Randomness is often incorporated by design in agent-based models in order to account for the individual and often unpredictable behavior that agents may exhibit, and also to provide a more realistic representation of many real-world situations. The outcome of any given modeling run may be different from another modeling run. This is not an error in the model or the methodology, but rather a standard approach for gaining understanding of how variability is implicit in the model.

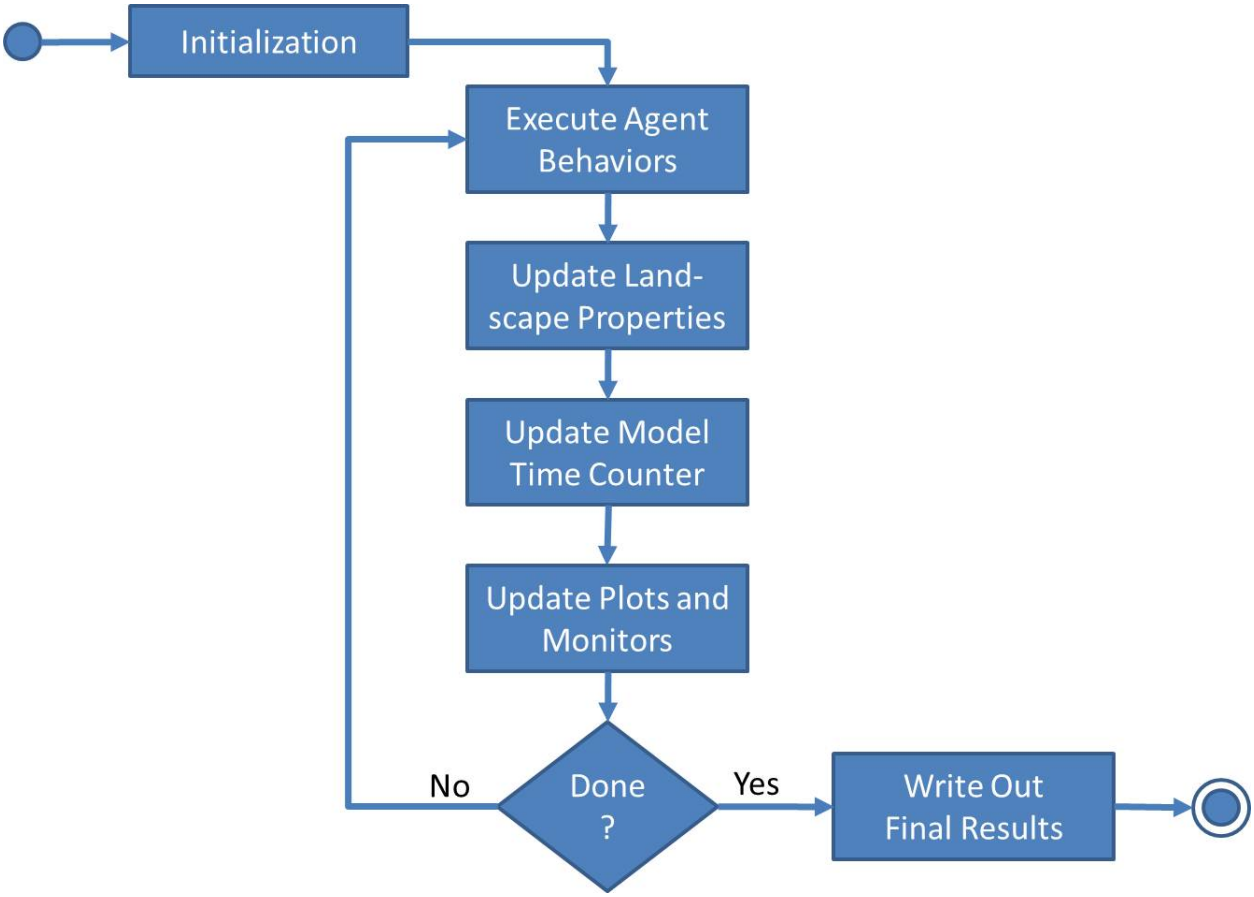
This concept is illustrated in the figure below, where if we start with a population with certain characteristics, we may have various possible outcomes when finished. By running the model

repeatedly and observing the distribution of outcomes, we can start to make inferences about the relationships between model input parameters and probable outcomes. Under some set of conditions, we may see Outcome 1, in which both population types increase in size. Under other sets of conditions, we may see Outcome 2 or Outcome 3, in which one population type increases and another decreases. Under still other conditions, we may see Outcome 4, in which both population types die out completely.



Designing a model may seem daunting at first, given all the things that need to be considered. Fortunately, there are some well-proven design patterns that can aid in the development of models. A basic pattern used in many agent-based models is illustrated in the figure below. The process begins by initializing the model and configuring any data that may be needed prior to the main simulation. The model then goes into an extended loop during which the agents execute their behaviors and the landscape properties are updated based on domain-specific modeling operations and interactions between agents. During each loop, some modeling housekeeping is performed to update monitors and plots, and update the global modeling time clock. Next, an evaluation is made to determine if the model should repeat another loop or end. If not finished,

the model repeats another loop. If finished, the model does final housekeeping such as writing out final computed values, and then ends.



This design pattern is straightforward to implement in NetLogo, as each of the main operations in the blue boxes can be coded as a procedure. The initialization is often handled by a procedure called "setup", and the iteration of the main loop is usually implemented by a procedure called "go." We'll see examples of this in the discussions to follow, and many models in the NetLogo models library follow this basic design.

3. Programming in NetLogo

The NetLogo programming language includes features found in object oriented languages and scripting languages. In this section, we will cover some important basics to get you started.

Agents

Not surprisingly, agents are the main entities in an agent-based model. An agent is a software abstraction of some real-world thing that you want to model. In an agent-based model, the agents generally have some goals to achieve, and some interaction with other agents and/or the environment. In some models, agents may have some memory of past interactions or some distinctive attributes that change over time.

Turtles (and Breeds)

In NetLogo, agents are called *turtles*, a term that is a historical reference to the earlier educational languages Logo and StarLogo that featured animated turtles in an elementary-school kid-friendly programming environment. All turtles have built-in attributes, including a unique identifier, a current position in 2-dimensional Cartesian space, and a color, shape, and size. Additional attributes can be added using the *turtles-own* command. Here's an example of adding weight, age, and gender attribute to the turtles in NetLogo.

```
;
; example of adding new attributes for all turtles
;
turtles-own [
     weight
     age
     gender
]
```

For most NetLogo applications, it is convenient to create specialized classes of agents to represent entities in the problem domain. Such classes of agents are called *breeds*, and since all breeds are descended from the turtle class, they inherit all the available turtle attributes automatically. In addition, breeds can have breed-specific attributes to indicate things of importance to the problem domain. Here's an example of how to create a breed of agents and to declare some breed-specific attributes. (Note that the *breeds* command requires the plural form of the breed first, followed by the singular form. Also note that the *breeds-own* command uses the plural form of the breed name.)

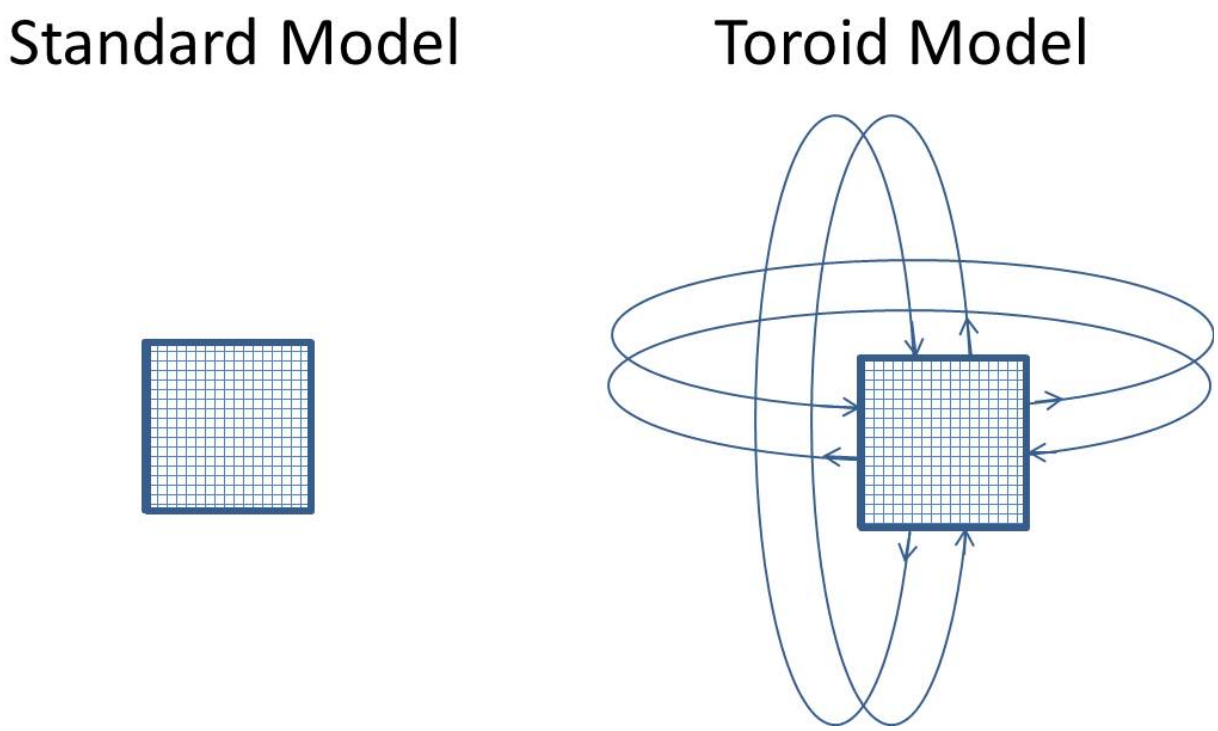
```
;
; example of declaring a new breed of agents
;    (note plural form precedes singular form)
;
breeds [ ducks duck ]

;
; example of adding attributes to the new breed
;    (note the use of the plural form in the breeds-own command)
```

```
ducks-own [
     wingspan
     weight
     age
     gender
     species
]
```

Patches

In the NetLogo environment, the world consists of a 2-dimensional grid of cells, much like a checkerboard. Each cell is called a "patch," and as we saw for turtles, the patch contains built-in attributes as well. These include the patch location in the 2-dimensional grid, and the patch color. Patches are considered agents, except they don't move – you can think of them as the (potentially active) landscape upon which your model will run. Patch attributes are defined using the *patches-own* command.

NetLogo allows the 2-dimensional patches grid to be either simple or toroid. In the simple grid model, the north, south, east, and west edges represent the extent of the model; no agent can traverse past these boundaries. In the toroid model, the grid is thought of as a 2-dimensional projection of a 3-dimensional doughnut-like shape, in which the east edge wraps around to the west edge, the west edge wraps around to the east edge, and similarly the north edge wraps around to the south edge, and the south edge wraps around to the north edge. The configuration of the patch grid model is managed on the NetLogo "Settings" button on the main screen; select the "World wraps horizontally" and/or "World wraps vertically" check boxes as needed for your model.



Just as with turtles, it is possible to add attributes to patches to represent features in the problem domain of your model. Here's an example of how to add some features to the patches as might be appropriate for an ecological simulation or land-use / land-cover change model.

```
;
; example of adding some attributes to the patches
;
patches-own [
     annual-rainfall-amount
     land-cover-category
]
```

Links (and Breeds)

In NetLogo, it is possible to define links between agents. Social network models, logistical flow models, and graph theory models rely heavily on the use of links between nodes. The NetLogo environment provides built-in support for managing links, making it easy to integrate them into models.

As with turtles, it is possible to create specialized classes or "breeds" of links. For example, certain classes of links may represent inflows or outflows, or certain types of links may represent different types of social or economic connections between agents. You can declare specialized types of links using the *create-breed[s]-to, create-breed[s]-from, create-breed[s]-with, create-link[s]-to, create-link[s]-from, and create-link[s]-with* commands.

Here's a very basic example of how to declare links between some agents. The links will be represented on the screen as thin lines connecting the agents. For this example, we'll assume that you are typing in commands at the *Observer* prompt into a blank (empty) NetLogo model. We start by creating 10 turtles, giving them random headings, and moving them forward 5 units. We then ask the first turtle (the turtle with id # 0) to create links to all other turtles. Finally, we ask all the turtles to move ahead 5 units so we can see the links being automatically managed by NetLogo.

```
;
; example demonstrating some link commands
;
create-turtles 10
ask turtles [ set heading random 360 ]
ask turtles [ forward 5 ]
ask turtle 0 [ create-links-with other turtles ]
ask turtles [ forward 5 ]
```

This discussion has really just barely scratched the surface of the Links capabilities – they are quite powerful and very useful for models that incorporate networking and social connectivity.

Observer

The *observer* in NetLogo is kind of like the referee in a soccer match – a ref is involved in the game and has full visibility into what is going on, but is not actually participating in the game. In NetLogo, the observer allows you to interact with the elements in the model and change things if you need to. We've already seen some examples above of how to use the observer to interact with agents and the patches landscape.

Data Structures and Variables

In any software system, data structures are the organizing elements used to manage the data in the system. NetLogo has some data structures that are similar to those found in programming and scripting languages (and the extensions provide many additional data structures), and it has some that are tailored specifically for agent-based modeling.

Variables in NetLogo

Variables in NetLogo software use naming conventions similar to Java or other languages. In NetLogo, variables are not case-sensitive, and can be composed of letters, numbers, underscores, hyphens, or question marks. Many NetLogo programs use hyphens to separate words, unlike the mixed or "camel case" seen in many Java or C++ programs. The following are examples of valid and invalid NetLogo variable names.

```
;
; valid NetLogo variable names
;
MAX_INTEREST_RATE              ; upper case with underscores
consumer-confidence-level      ; lower case with hyphens
Income2004                     ; mixed case and numbers
product-in-stock?              ; example of a Boolean variable
```

```
;
; invalid NetLogo variable names
;
123variable          ; begins with a number
max_interest_rate    ; not case sensitive, variable already used
```

Global Variables (and Interface Objects)

Many NetLogo models make use of global variables to represent values that need to be read or written at any place within the model code. Global variables are declared in a *globals* block at the beginning of the model. Here's an example of three global variables that we plan to use in our model code.

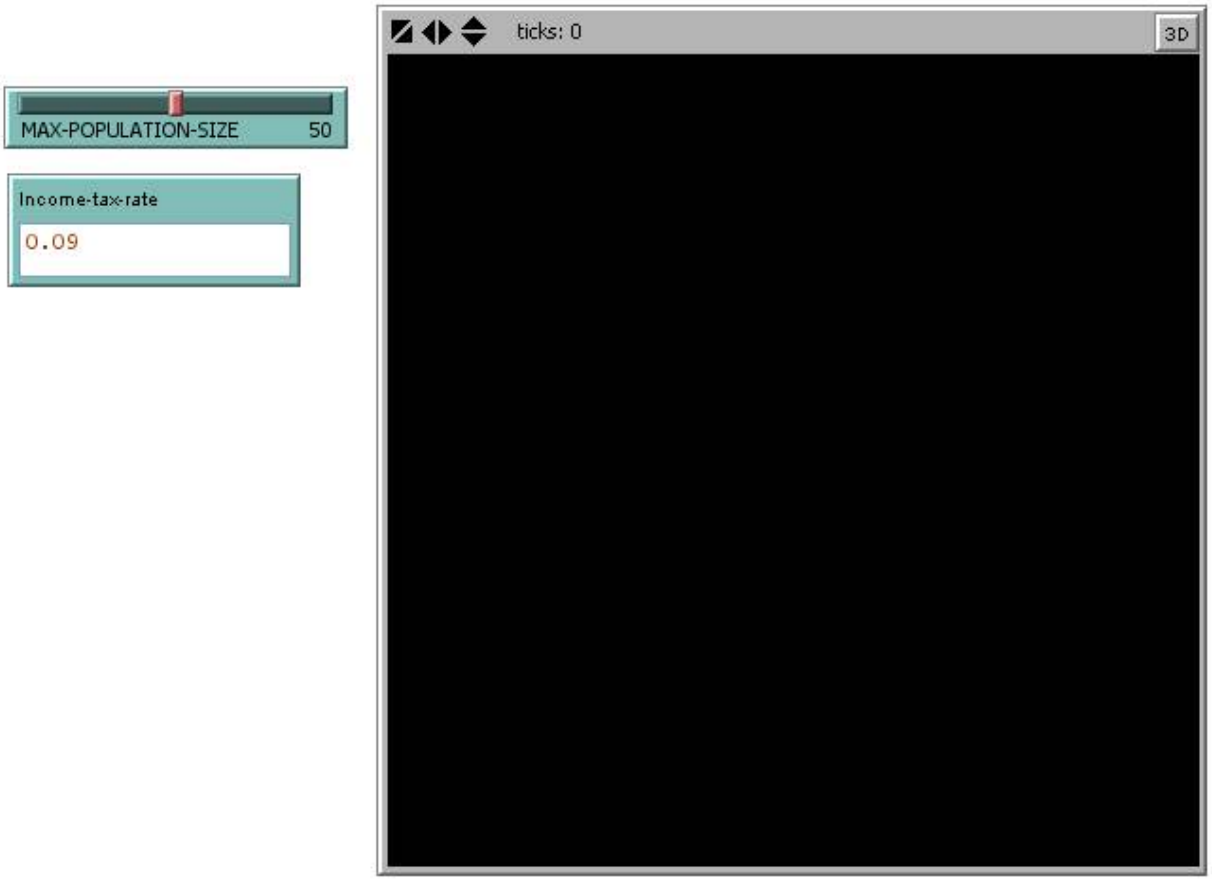
```
;
; global variable example
;
globals [
     MAX_POPULATION_SIZE
     Income-tax-rate
     average-farm-size
]
```

Global variables can be used completely within the model code, or they can be exposed via certain components on the model interface tab such as the slider, chooser, input, or switch. If used as an interface tab component, the global variable is declared within the interface component, and it is **NOT** redeclared in the model code. This can be confusing to new NetLogo programmers because it may not be obvious where the variable "lives."

Let's clarify this using the previous global variables example. Suppose after building our initial model, we decide that we'd like to have the user specify some inputs on the interface tab: *MAX_POPULATION_SIZE* will be declared in a slider, and *Income-tax-rate* will be declared in an input box on the Interface tab. The *average-farm-size* variable is a global variable that we will continue to calculate during the simulation, but it is not declared as in interface tab component. So we will have to remove *MAX_POPULATION_SIZE* and *Income-tax-rate* as globals in our code, because they have already been "declared" as elements of the interface tab components. We will, however, need to keep *average-farm-size* in our globals block. Our new globals declaration will look like this.

```
;
; updated global variable example
;
globals [
     average-farm-size    ; global variable in our model
]
```

And our corresponding new interface tab should look like this. Note that *MAX_POPULATION_SIZE* and *Income-tax-rate* are now within a slider and an input box on the interface tab.



Local Variables

NetLogo allows local variables to be declared within the scope of a procedure or function. Variables are introduced and initialized using the *let* command, and variables are assigned values using the *set* command.

You declare a variable by using *let*. You must provide an initial value for the variable when it is created. So if you write code that says `"let x 1.5"`, that means you've created a variable named "x", and it has an initial value of "1.5". If we did not supply an initial value, NetLogo will supply a default value of 0.

Here's an example of a code snippet we might find inside a procedure or function that demonstrates using *let* and *set*. We'll declare and initialize some variables using *let*, do some nonsense code as a placeholder, then we'll return to our variables and assign them some new values using *set*.

```
;
; example to demonstrate variable declaration using "let"
```

```
; and assignment using "set"
;

;
; declare some variables using let
; initialize them with starting values
;
let x 0
let y 1.5
let z "red"
let my-list [ 1 3 5 ]

;
; now do some nonsense code…
print "la de da"


; later in the procedure, we want to
; assign variables new values using set
;
set x 3
set y 2.5
set z "green"
set my-list [2 4 6 ]
```

We've declared and assigned several variables, and we've used several different data types as well, including integer number, real number, character string, and list. We've made use of the optional initialization feature for our variables when we declared them with *let*.

A quick note about lists: if you plan to use a variable as a list, then you should initialize that variable with an initial list of values, or set it to the empty list using the list declaration *[ ]* operator. If you forget to initialize your list type variables, then NetLogo will (incorrectly) assume that you mean for them to be numeric variables and will assign them a default value of 0. A later attempt to access a variable that you intended to be used as list will cause a run-time error when NetLogo attempts to use a list operator on an integer value.

```
;
; example of proper way to declare a list variable
;
; declare variable my-list for use sometime later as a list,
; and initialize it for now to the empty list
;
let my-list []
```

When a variable is declared using *let*, it assumes the type of the initial value if supplied, or integer type if no initial value is supplied. After a variable has been declared, NetLogo allows variables to be assigned new values using the *set* command. However, in addition to changing

value, the variable may change type as well based on the new assignment value. So in the example above, the initial declaration of *x* was 0, which you might think implies that *x* is bound to an integer type. However, you could reassign *x* to non-integer values if you want to – NetLogo doesn't enforce strong type checking. As mentioned above in the discussion on list types, however, if you change the underlying type from one atomic type to another via a *set* command, you'll need to remember what type your variable is and access it appropriately.

```
;
; perfectly legal variable assignments
; demonstrating type polymorphism in NetLogo
;
let x 0       ; x is declared and assigned an integer value of 0
set x 5       ; x is reassigned integer value 5
set x 3.14159 ; x is now assigned floating point value pi
set x "green" ; x is now assigned as a character string
set x []      ; x is now set to an empty list
```

Agentsets

In NetLogo, agents are the main features of agent-based models. It's very common to operate on subsets of agents in your model, and anytime you need to obtain such a set, you're dealing with an "agentset". An agentset is just what it sounds like: it's a set of agents – there is no particular ordering to the agents, and you can think of it as a "grab bag" of agents that meet some criteria.

In order to use agentsets, we need to "query" the pool of agents and find the ones that meet certain criteria based on what we're doing in our model. This is conceptually similar to doing a Structured Query Language (SQL) query in a database environment, except that in a database world we're dealing with tables, and in the agent-based world we're dealing with collections of agents. A common form for doing this in NetLogo is to use the *with* operator, combined with some type of evaluation criterion.

Here's a small procedure that we might possibly use if we were working with the Schelling Segregation model discussed earlier. The goal here is to set up two agentsets: `green-set` is the one for green agents and `red-set` is the set for red agents. We then count the number of agents within each agentset that are "happy," and report this value to the output console. (The "*to*" and "*end*" statements are part of the procedure declaration – we'll talk more about procedures and functions later. Also, *type* and *print* are built-in functions for printing data to the output console).

```
;
; example of using agentsets
;
to demonstrate-agentsets
     let green-set (turtles with [color = green])
     let red-set (turtles with [color = red])

     let green-happy (count green-set with [happy? = true])
```

```
     let red-happy (count red-set with [happy? = true])

     type "greens have " type green-happy print " happy members"
     type "reds   have " type red-happy   print " happy members"
end
```

Lists

As we've already seen from some examples, NetLogo provides support for list data structures, which are sequential collection of elements. Lists can contain constants, variables, or even other lists. There are several important differences between lists and agentsets. Lists are sequential, which means that list elements have a distinct ordering, but agentsets are unordered collections. Lists can contain sublists (and sublists of sublists, ad infinitum), but agentsets are simple "bags" that do not contain additional nested elements (agentsets should not be confused with "multi-sets" as they will NOT contain duplicates). Finally, lists can contain any type of element, but agentsets only contain agents.

There are a number of built-in list operators in NetLogo, and of course further details are available in the online documentation. Here's a short example to show some list features, including *first*, *last*, *length*, and *foreach*. Note that the *foreach* command iterates over the list, and the *?* operator accesses the current list element as the iteration proceeds. As before, *type* prints a value to the console without a line break, and *print* prints values to the screen followed by a line break.

```
;
; example to demonstrate some list operators
;
to demonstrate-lists
     let my-list [1 3 5 7 9]

     let my-first (first my-list)
     type "first =  " type my-first print " (should be 1)"

     let my-last (last my-list)
     type "last =  " type my-last print " (should be 9)"

     let my-len (length my-list)
     type "len =  " type my-len print " (should be 5)"

     let my-sum 0
     foreach my-list [
          set my-sum (my-sum + ?)
     ]
     type "sum =  " type my-sum print " (should be 25)"
end
```

This procedure will result in the following output on the console.

```
first =  1 (should be 1)
last =  9 (should be 9)
len =  5 (should be 5)
sum =  25 (should be 25)
```

If you're planning to use lists in NetLogo a lot, there are a couple of tricks you can use to make them more efficient. As mentioned above, anytime you declare a list variable, make sure to initialize it first as an empty list or an initial list of values (i.e, use a command like *let myList []* or *let myList [1 3 5]*) so NetLogo knows you want your variable to be a list type. If you don't do this, NetLogo will throw an error when you try to access your variable later. Also, remember that when you are building up a list iteratively, you need to set your list variable to be equal to the new value plus all the old values. Moreover, you need to tell NetLogo where the new value is supposed to go (the beginning of the list and the end of the list are the usual spots). Use the commands *fput* to place a new item at the front of the list (remember *fput* as "front-put") and use *lput* to place a new item at the end of the list (remember *lput* as "last-put"). The NetLogo commands necessary to do this may look a little odd, but if you keep in mind the logic, you can build up very sophisticated statements dealing with lists. Here's an example of some list operations.

```
;
; initialize a small list of numbers
;
let my-list []                 ; initially my-list is empty
set my-list (list 1 2 3 4 5)   ; my-list is now [1 2 3 4 5]
set my-list fput 0 myList      ; my-list is now [0 1 2 3 4 5]
set my-list lput 6 myList      ; my-list is now [0 1 2 3 4 5 6]
```

You can randomize the ordering of the items in a list using the *shuffle* command. You can reverse the order of a list by using the *reverse* command. For example, picking up where we just left off above.

```
;
; from our previous work, my-list is currently [ 0 1 2 3 4 5 6]
;
set my-list reverse my-list; my-list is now [6 5 4 3 2 1 0]

set my-list shuffle my-list; my-list is shuffled [3 5 2 6 0 1 4]
                           ; or something like that)
```

Remember that you must always start list manipulations by setting something to the value of the manipulated list (that's why you see the ***set myList*** *shuffle myList* above).

Finally, if using an early version of NetLogo (prior to version 5), to get to the end of a list, you start at the beginning and then iterate over each element to get to the end. This can have some performance implications, especially if you have long lists to work with. For example, if you have a lot of agents adding themselves to a list, it is far more efficient to have the agents place

themselves at the front of the list. If necessary, you can readjust the order of the list later if you need to. In NetLogo version 5, the performance issues of iterating over lists have been largely removed. Here's an example of various ways of doing list access.

```
;
; examples of efficient and not-so-efficient list operations
;

; declare a couple of global variables to
;    eventually hold some lists for us
;
globals [
     my-fast-list ;  by default set to 0, not a list
     my-slow-list ;  by default set to 0, not a list
]

;
; driver procedure to test various modes of list access
;
to do-list-example
     ; initialize variables as empty lists rather than zero
     set my-fast-list []   ; now set to empty list []
     set my-slow-list []   ; now set to empty list []

     ; now compare two ways of doing list operations
     ask turtles [do-fast-list]
     ask turtles [do-slow-list]
end

;
; access from the beginning of the list - much faster
;
to do-fast-list
     set my-fast-list fput self my-fast-list ; more efficient
end

;
; access from the end of the list - much slower
; although it should be noted that with NetLogo 5 fput and
; lput are now nearly identical in performance…
;
to do-slowlist
     set my-slow-list lput self my-slow-list ; less efficient
end
```

Another thing you may want to do is iterate over all the agents in an agentset or access the agents in an agentset multiple times in the same order. As noted earlier, agentsets are really just a "bag-

o-agents." When you access an agentset, you'll get the agents in random order every time. If you want to iterate over the agents as a group in some particular order, you will need to convert the agentset to a list. The simplest way to do that is to use the *sort* or *sort-by* commands. Here are some example code snippets to show how these operators work on lists and agentsets. (Don't worry if you don't follow all of the commands just yet – you can use the online NetLogo dictionary to look up unfamiliar commands if you want to get ahead a bit). When you want to change the list back into an agentset, you can use the *member?* command. An example of using this is shown at the end of the code block below.

First, let's declare some global variables to hold lists of turtles.

```
;
; declare several global variables
;    to hold our bag and lists of turtles
;
globals [
     bag-of-turtles
     list-of-turtles
     other-list-of-turtles
]
```

Now, let's provide a single numeric value associated with each turtle, called *my-value*.

```
;
; each turtle has a single local value called my-value
turtles-own [
     my-value
]
```

The setup procedure creates 100 turtles and assigns the *my-value* field of each turtle to a random value in the range of 0 and 99

```
;
; set up, create some turtles,
;   and initialize each turtle with a random value
;
to setup
  clear-all
  create-turtles 100 [set my-value random 100]
  ; each turtle now has a value in my-value between 0 and 99
end
```

Now, we're going to set one of our global variables (*bag-of-turtles*) to an agentset. We will use the fact that the *my-value* field of each turtle is less than 200 (since we know it has to be between 0 and 99), and this "trick" will allow us to select all the turtles in the model and put them into our agentset. The *show* method simply prints out the contents of the agentset to the console.

```
;
; create an agentset from the turtles in our model
```

```
;
to make-agentset
  set bag-of-turtles turtles with [my-value < 200]
  ; since all values of my-value < 100,
  ; all turtles are in the agentset

  show bag-of-turtles ;show is a method to print out an agentset
end
```

Now let's set one of our global variables to a list. We'll set *list-of-turtles* to a list by using the output of the *sort* method, which takes an agentset as input and returns a sorted list as output. Without any further specifications, the sort method sorts the agents in the agentset in ascending order by comparing the internal turtle variable assigned by the system when the turtle was created; this value is known as the "who" number of the agent.

```
;
; create a list from the agentset
;
to make-a-list
  set list-ofturtles sort bag-of-turtles

  ; list-of-turtles is now a list of the turtles
  ; sorted in ascending order by the turtle's unique identifier
  ; (also called its "who" number)

  show list-of-turtles
end
```

Here we're going to make another list, but this time we'll use a more sophisticated sorting method called *sort-by*. As with its sibling *sort*, *sort-by* takes an agentset as input and returns a sorted list as output. However, *sort-by* requires a comparison clause to tell it which field should be used for comparison. In this case, we are going to use the only other variable our turtles have, which is the *my-value* variable. The *sort-by* comparison clause says compare successive pairs of agents in the agentlist, and arranges them so that the *m-value* of the first one is greater than the *my-value* of the next one, and so on. If they happen to be equal, one is selected at random. This will result in a list of agents, ordered from highest to lowest value based on their *my-value* variable. Finally, we use a *show* to dump the contents of the *other-list-of-turtles* list, and we use a *foreach* method to iterate over the list and print out the *my-value* variables.

```
;
; create yet another list
;
to make-a-differentlist
  set other-list-of-turtles
     sort-by [[my-value] of ?1 > [my-value] of ?2] bag-of-
turtles

  ; other-list-of-turtles is now a list sorted
  ;    from highest to lowest by the turtles' my-value number
```

```
  show other-list-of-turtles

  ; print out the my-values for each turtle
  foreach other-list-of-turtles [ask ? [show my-value]]
end
```

To finish out our quick introduction to lists and agent sets, we can turn a list (sorted or not) back into an agentset. We'll accomplish this by declaring a new variable *new-set*, and using an *agents with* syntax, which queries the set of all agents in the simulation and returns an agentset. The "trick" here is that by using the *member? <agent> <list>* predicate when we know that *self* has to be in *other-list-of-turtles* because every turtle in the simulation is in this list, we are guaranteed that we will get every turtle in the simulation back into our agentset.

Don't worry if you're not following along completely at this point – the main thing to take away is that agentsets and lists each have different features, and that it is easy to convert from one form to another in your code so you can choose the data structure that suits your purposes best.

```
;
; to turn a list back into an agentset you can do something
; like this:

let new-set turtles with [member? self other-list-of-turtles]
```

One could also do this:

```
let new-set turtle-set other-list-of-turtles
```

Booleans

We've already seen some examples of Boolean variables, but just to review, a Boolean variable is a variable that can only be either *true* or *false*. In NetLogo, by convention we indicate such variables by appending a question mark to the end of the variable name. That way, it's easy to visually distinguish such variables from other kinds of variables. Here are a couple of examples of declaring and initializing Boolean variables using *let*. Just like other variables, Boolean variables are updated using *set*.

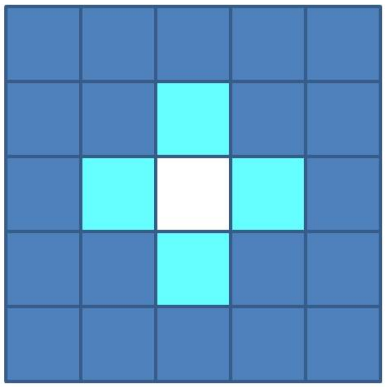
```
;
; example of declaring and initializing some Boolean variables
;
let all-done? false
let green-agents-happy? true
let red-agents-happy? true


;
; example of setting a Boolean variable after its been declared
;
```

```
set all-done? true
```

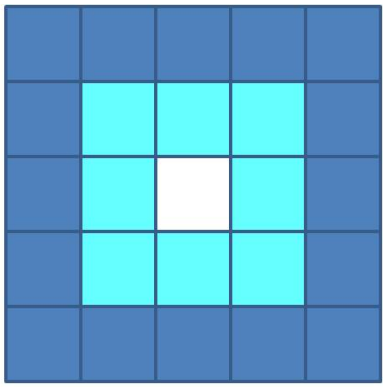
Neighbors

In many types of modeling applications in the physical and social sciences, the population of interest is treated as a homogeneous mass that changes over time. These techniques often rely on one or more equations to express population change as a function of time and other variables, such as exponential growth, logistic growth, and coupled systems of ordinary differential equations. These approaches are generally not spatial, meaning that the location of individuals is irrelevant. In many cellular automata and agent-based models, however, important effects are observed as a consequence of the spatial arrangement of patches and/or agents.

One of the key concepts in agent-based models is the notion of "neighborhood." As we briefly touched on earlier, there are two widely used neighborhood types: the *von Neumann* and the *Moore* neighborhoods. The von Neumann neighborhood consists of the immediately adjacent elements at the cardinal compass directions: North, South, East and West. The Moore neighborhood consists of all the von Neumann adjacencies, plus the Northeast, Northwest, Southeast, and Southwest adjacencies as well. The following diagram shows how this looks from the perspective of the light-colored cell in the middle: the light blue neighbors on the left figure form a von Neumann neighborhood, the light blue neighbors on the right figure form a Moore neighborhood, and the dark blue cells are outside the neighborhood boundaries.



Von Neumann Neighborhood



Moore Neighborhood

NetLogo supports both types of neighborhoods. The *neighbors4* command returns an agentset consisting of the 4 von Neumann neighbors, and the *neighbors* command returns an agentset consisting of the 8 Moore neighbors. As an example, here are some commands you could type in at the Observer prompt from a new empty NetLogo model to see how this works. The first command colors all the patches blue. The second command highlights the center patch (this is the patch located at x=0, y=0) and colors it white. The third command creates a von Neumann neighborhood of the 4 adjacent patches to the center patch, and colors them cyan. Finally, the fourth command creates a Moore neighborhood of the 8 patches adjacent to the center patch, and colors them cyan.

```
;
```

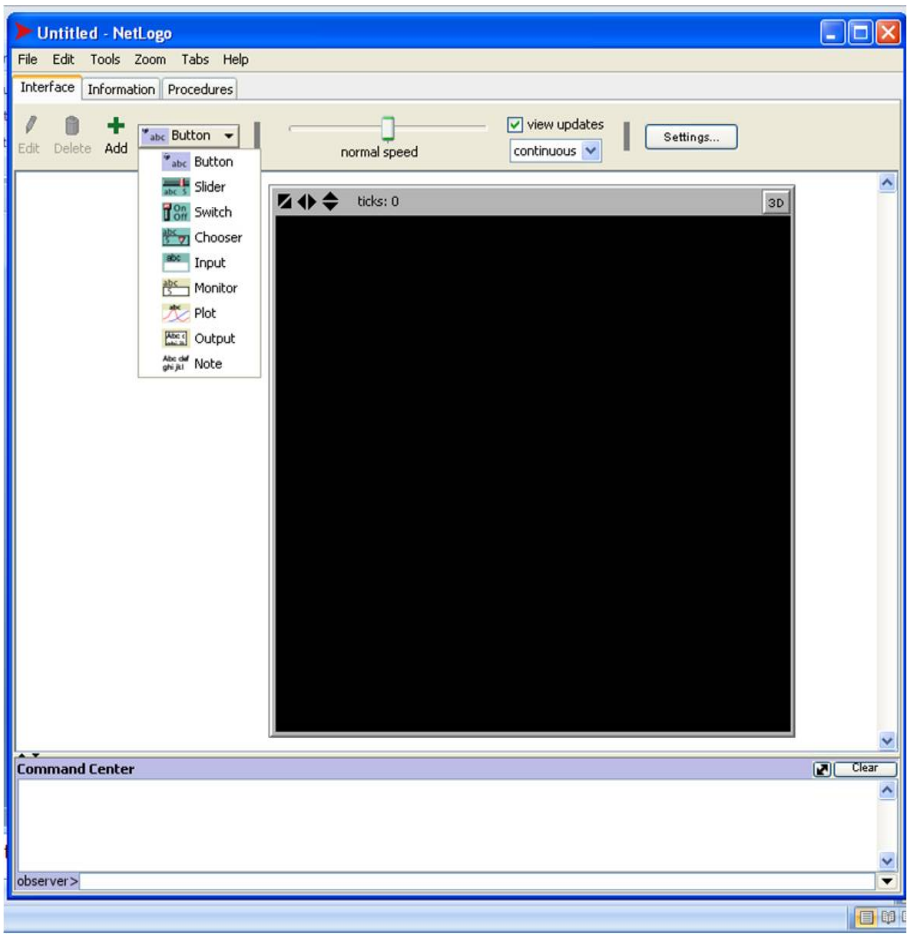
```
; example demonstrating neighborhood commands
;
ask patches [set pcolor blue]
ask patch 0 0 [set pcolor white]
ask patch 0 0 [ask neighbors4 [set pcolor cyan] ]
ask patch 0 0 [ask neighbors [set pcolor cyan] ]
```

The previous discussion has been focused on the patch context, but NetLogo supports similar types of operations for turtles (agents). Thus, if you need to locate all the agents occupying the 4 or 8 adjacent patches, the same *neighbors4* and *neighbors* commands will work. The agentset returned in this case, however, will be a collection of agents (if there are any nearby), as opposed to a set of patches nearby when used with `turtles-on` (`turtles-on neighbors` or `turtles-on neighbors4`). So this leads to a subtle point: if you query *neighbors4* or *neighbors* in a patch context, you will always get 4 and 8 patches returned in your agentset, respectively[1]. However, if you query *neighbors4* or *neighbors* in a turtle (agent) context, you'll get however many agents are occupying the adjacent patches, which could vary from none at all to a whole lot, depending on how crowded your model is.

Interface Objects: Button, Slider, Switch, Chooser, Input, Monitor, Plot, Output, Note

One of the really powerful features of NetLogo is the built-in support for various interface objects that allow the user to interact with the model. These features are available in many programming languages and simulation environments, but often require a lot of programming expertise to use. The NetLogo implementation makes it very easy to incorporate user input via buttons, sliders, choosers, inputs and switches, and also provides support for monitors, a variety of plots, and output messages. All these objects are available from the Interface tab of the NetLogo environment. We'll take an extended tour of these tools now, as they are very useful for developing and using models.

[1] OK, technically, there are some exception cases where you could get less than 4 or 8 adjacent patches, if you queried a patch on the edge of the landscape that was not using the default "wrap-around" mode.

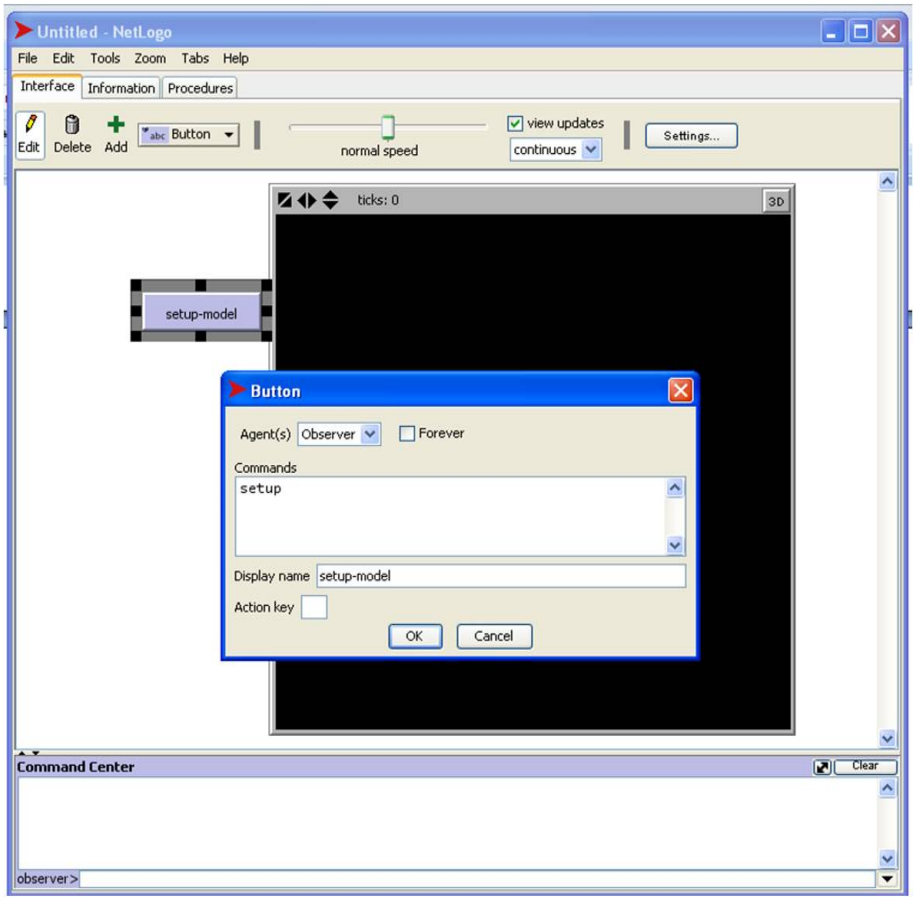


Button

A button in NetLogo is an interface object that associates a button on the interface screen with actions in the model code. A typical application of this is to define a setup button to initialize your model and prepare it for execution. To do this, you write a setup procedure in your procedures tab, then create a start button on the interface tab and associate the two. Here's some notional code to do a setup procedure.

```
;
; example setup procedure
;    assumes we have some initialization
;    procedures for our globals, patches, and turtles
;    defined elsewhere
;
to setup
     clear-all
     initialize-globals
     initialize-patches
     initialize-tutles
end
```

Now on the interface tab, we define a new button called "setup model", and associate it with our setup function. Enter "setup-model" for the Display Name, then type in the associated commands to be executed when this button is pressed, which in this case is simply to execute the "setup" procedure. When we press the "setup" button, the commands we specified will be executed in sequence, which in this case will initialize the global variables, then initialize the patches, and finally initialize the turtles in the model.

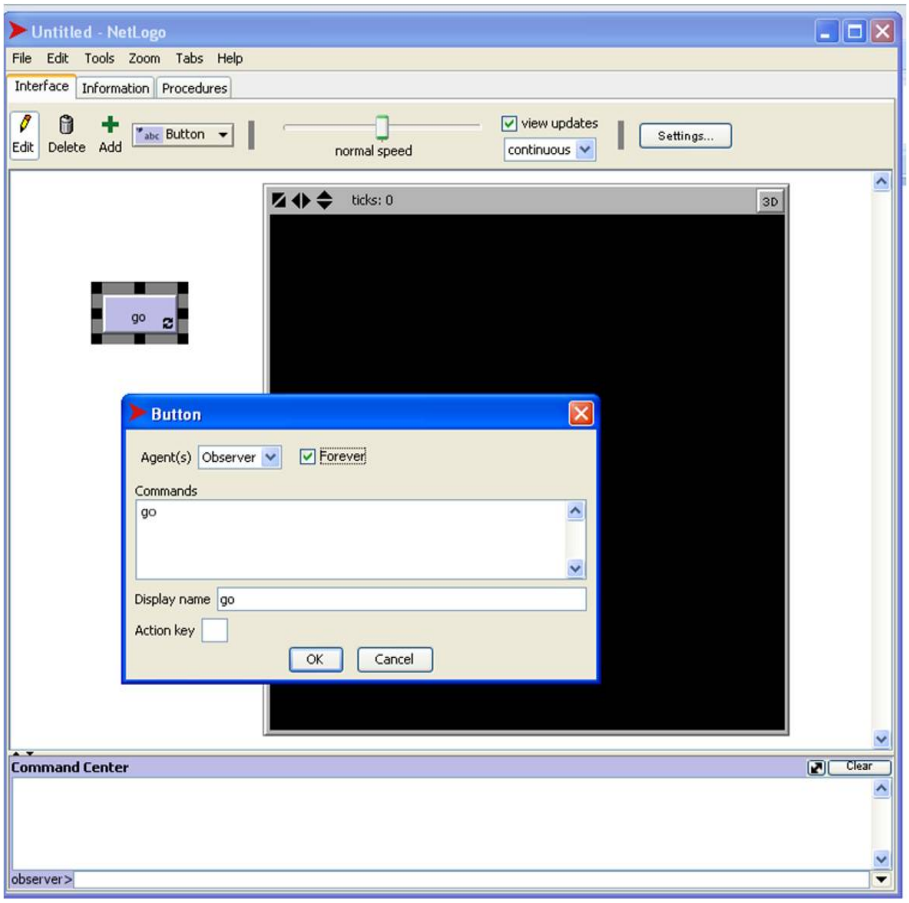


Another commonly used button in NetLogo models is the "go" button, which is associated with the "go" procedure in the model. An important point – in a case like "go" where we want the procedure to execute continuously, the "Forever" checkbox on the button must be checked. This will repeat whatever commands are associated with the button without stopping.

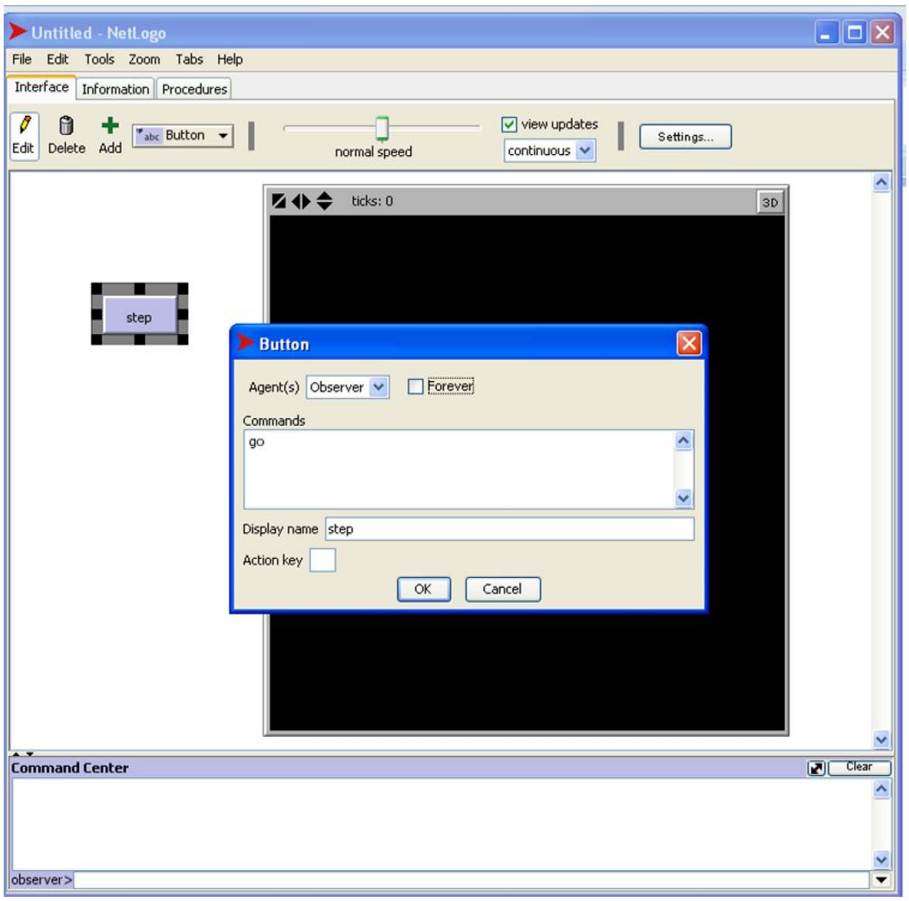
An important feature of the "Forever" property on buttons is the ability to switch between running and paused state simply clicking on the button. In effect, a button with the "Forever" box checked works like a toggle: it's either running or not running, based on how many times you have clicked the button. For the "go" button typically used to drive the simulation, this feature allows you to easily pause and restart your model. Press go once to start the simulation, press go again to pause the simulation, press go again to restart, press go again to pause.

```
;
; example go procedure
;    assumes we have some procedures we want to execute
;    that are defined elsewhere
;
to go
     do-procedure
     do-another-procedure
     do-yet-another-procedure
     tick
end
```

Here's how this would be linked to a new "go" button on the interface tab. Note the "Forever" button is checked, indicating we want the go procedure to execute indefinitely, or until we explicitly pause the simulation by pressing the button again.

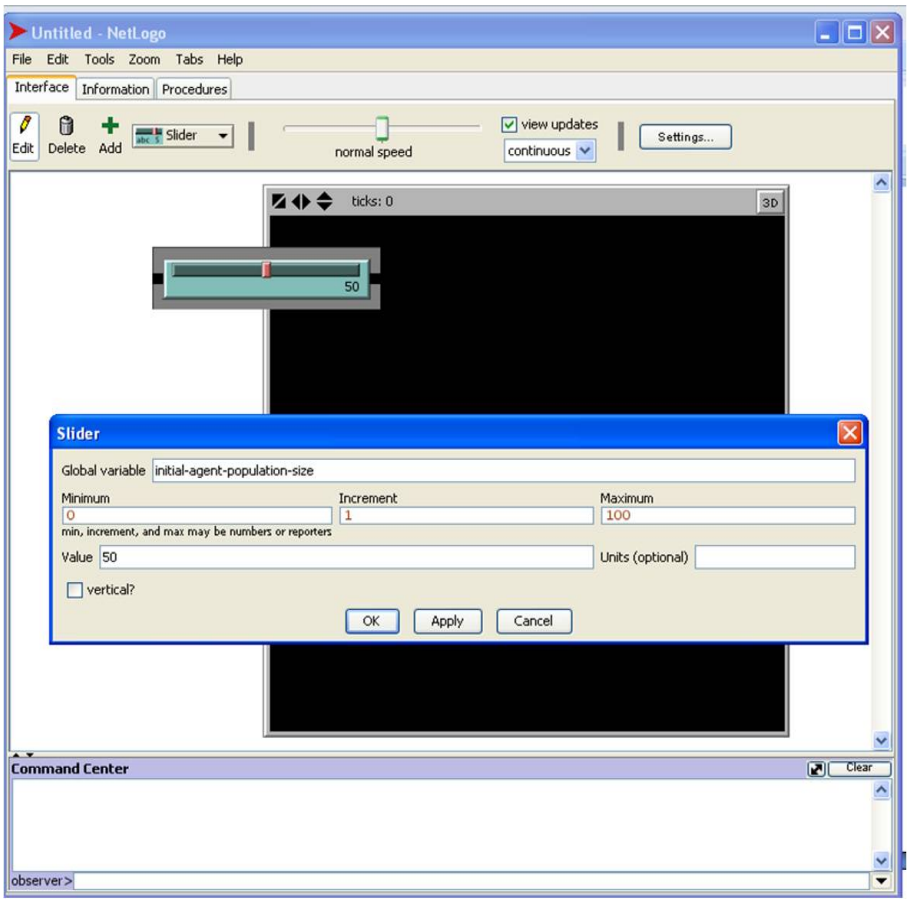


Another handy button to define is the "step" button. This is just like a "go" button, in that it uses the "go" command, but it does not have the Forever checkbox selected. This can be very helpful when debugging, as pressing the "step" button will cause the model to do a single loop through the main procedure, giving you time to examine variables, look at graphs or plots, and generally see what's going on. Here's how this to do this with a new "step" button.



Slider

A slider is a graphical widget that moves from left to right (or top to bottom). The user can adjust the value of the slider, and an associated global variable is adjusted accordingly based on the lower limit, upper limit, and increment defined in the slider. In the example below, we've defined a slider to manage a global variable called *initial-agent-population-size*, with a minimum of 0, a maximum of 100, an increment of 1, and an initial value of 50. This slider is horizontal, but we could specify a vertical slider by checking the *vertical?* checkbox on the bottom left. If the user changes the slider, the global variable is adjusted accordingly, and our simulation can react to this as appropriate. Note that we do not need to "redeclare" the *initial-agent-population-size* global variable explicitly in our code – by entering this global value in the slider definition, the variable is already declared and available for use as a global variable anywhere in the code.



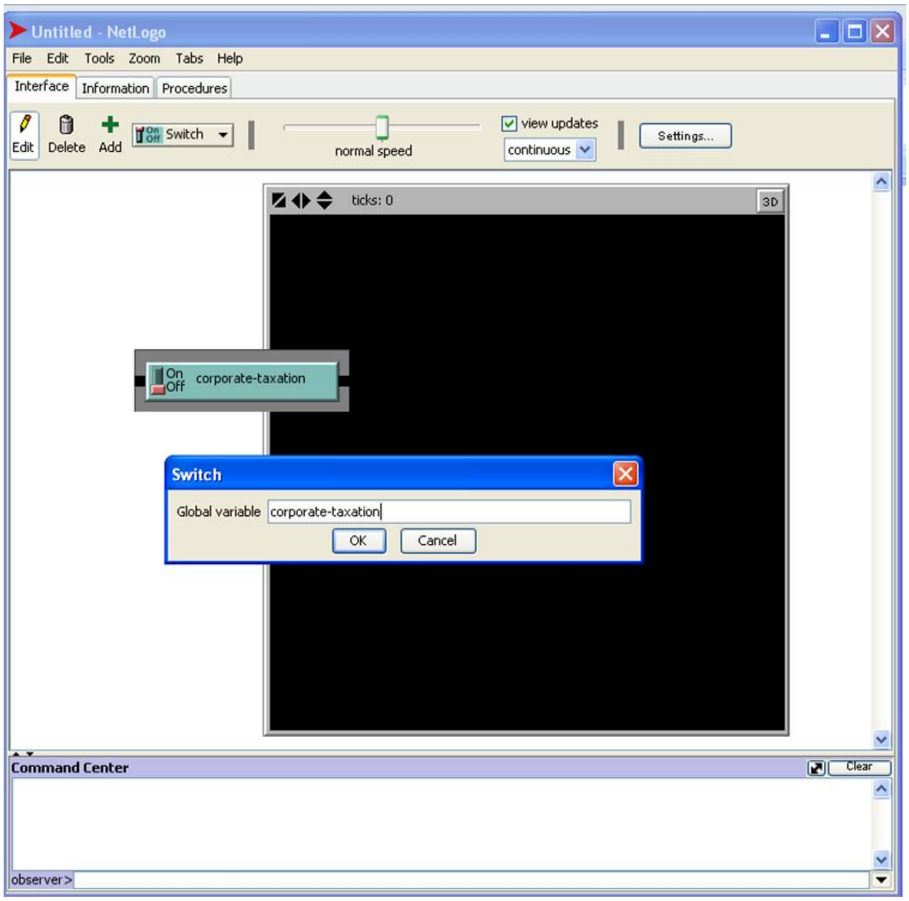
Switch

A switch is a graphical widget that looks and acts like a common light switch – it can be either on or off. Based on setting the interface tab to on or off, the global variable will assume either a value of ***true*** or ***false***, respectively, and your model code can react accordingly. Here's a code snippet to illustrate how to interact with the value set by a switch on the interface tab. Note that the evaluation is made using Boolean values of ***true*** or ***false***, not the switch string labels of "*on*" or "*off*". This is a common programming error made by new NewLogo developers.

```
;
; example of using a switch in model code
;
; switch GUI has declared a global value named
; "corporate-taxation"
;
to calculate-earnings
     ;
     ; code evaluates Boolean "true" or "false"
     ; NOT switch GUI values of "on" or "off"
     ;
     if (corporate-taxation = true) [
          calculate-earnings-using-taxes
```

```
      ]
      if (corporate-taxation = false) [
           Calculate-earnings-without-using-taxes
      ]
end
```

Here's the way to define the switch widget with the associated global variable of *corporate-taxation*.



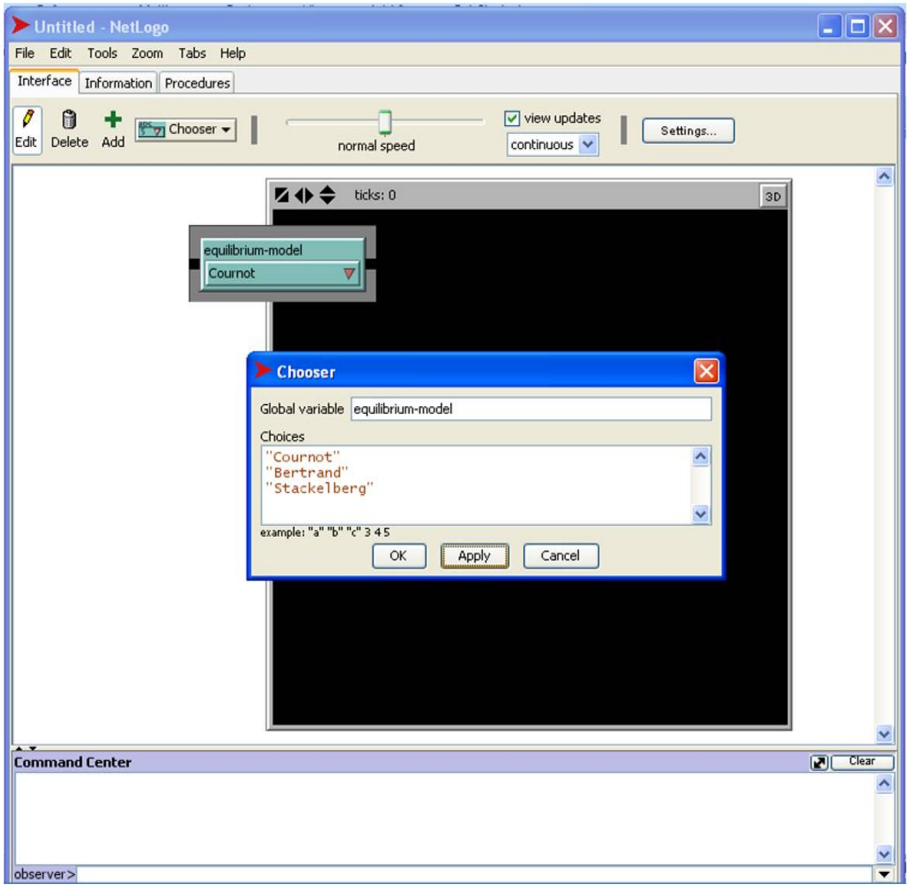
Chooser

A chooser is a graphical widget that presents the NetLogo user with a choice from a fixed menu of options. This is also known in some environments as a "pull-down menu." Here's an example of how we would use chooser inputs in our model code. Note that the comparisons are made as string variable types, so they are enclosed in double quotes in the code logic.

```
;
; example using global variable set by Chooser on Interface tab
;
to compute-equilibrium
      if (equilibrium-model = "Cournot")
           [compute-cournot-equilibrium]
      if (equilibrium-model = "Bertrand")
```

```
          [compute-bertrand-equilibrium]
     if (equilibrium-model = "Stackelberg")
          [compute-stackelberg-equilibrium]
end
```

Here's the corresponding definition for the global variable *equilibrium-model* in the chooser widget on the Interface tab.



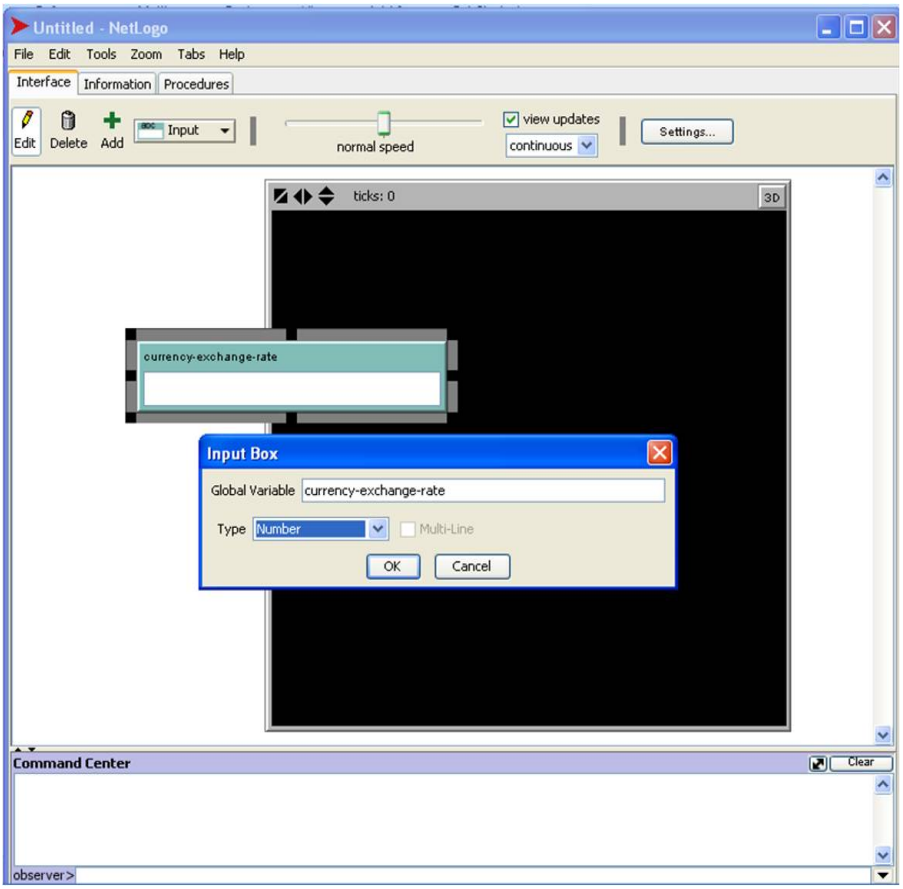
Input Box

In NetLogo, the input box is a graphical widget to allow user-specified input on the Interface tab. This is useful if the user needs to specify some value for the model, but you don't want to limit the range of values using a slider or a chooser. The code works very much the same as we have seen for other interface widgets: a global variable is declared in the widget, and the global variable is available for use in the model. Here's a code snippet to show how to use a value that is read from an Interface tab input box widget.

```
;
; example of using input box value from Interface tab
;
to assess-currency-exchange-rate
```

```
      if (currency-exchange-rate > 0.0) [
           process-with-currency-exchange
      ]
      if (currency-exchange-rate <= 0.0) [
           process-without-currency-exchange
      ]
end
```

Here's the corresponding setup for the input box for the user to enter the *currency-exchange-rate* on the model Interface tab. Note that we need to specify that the input value is a numeric type on the Type pull-down. Other data types available in input boxes include strings and colors.

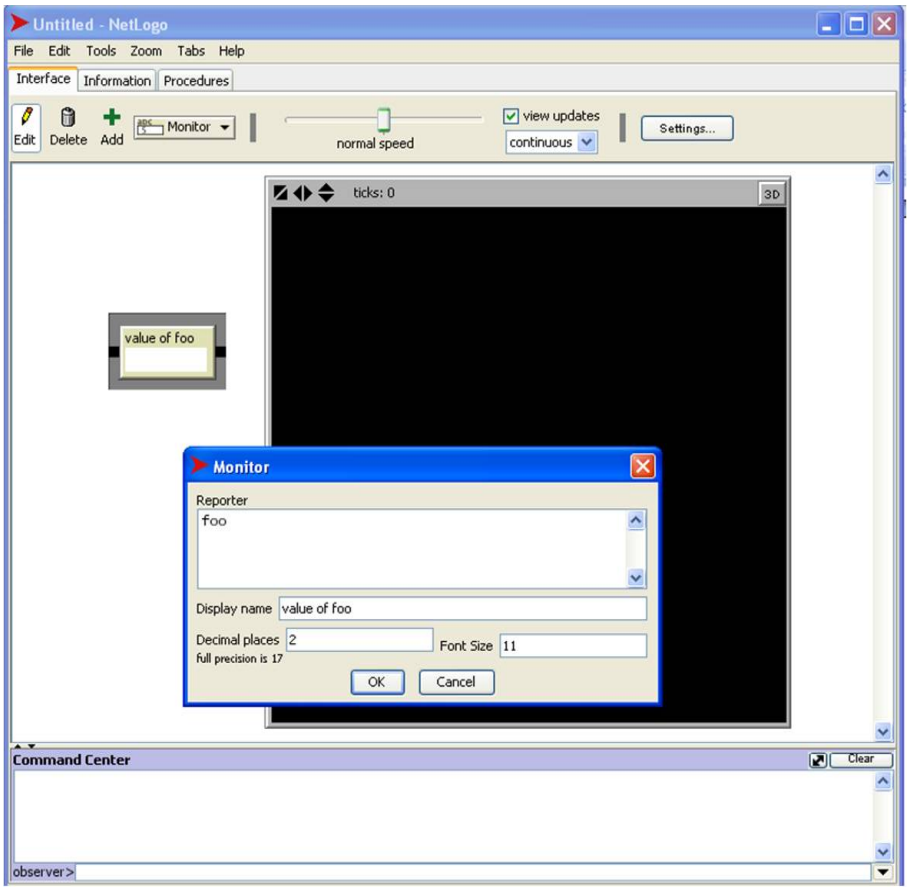


Monitor

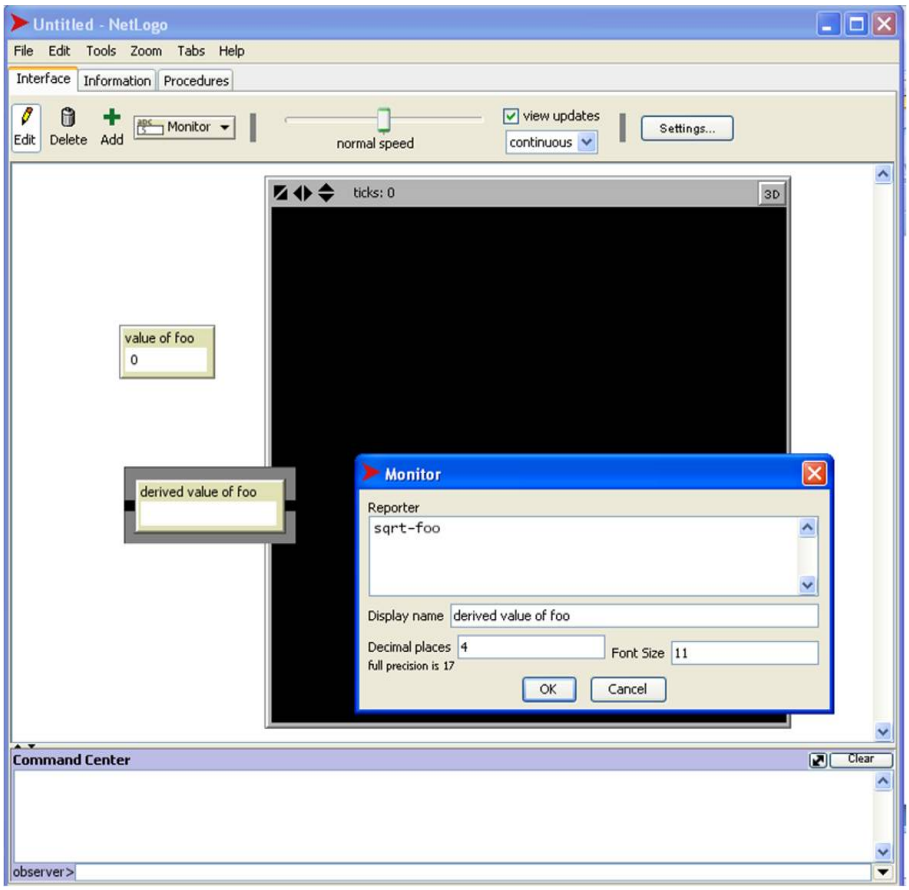
A monitor is a graphical widget that displays the value of some variable in your model. This is useful to observe as things are going on, and it can also be a valuable debugging tool. You can monitor global variables, or you can write a function (called a reporter) to calculate a value and make it available for your monitor. Here is a code snippet to demonstrate how to do both of these: we're going to monitor a global variable called *foo*, and also monitor a reported value that is derived from *foo* using the square root function. Please note, this is a simple example of the use of monitors and reporters, one could use `sqrt` directly rather than as part of a reporter.

```
;
; example code showing use of monitors
;

;
; declare a global variable called foo
;
globals [
  foo
]

;
; reporter function
;    returns square root of global variable foo
;
;
to-report sqrt-foo
  report sqrt foo
end

;
; simple go function
;    all it does is set foo to a random value
;
to go
  set foo (random 100)
end
```

On our Interface tab, we'll define two monitors. One will simply indicate the value of the global variable *foo*, and another will invoke the reporter *sqrt-foo* to indicate the computed value. To set up these monitors, we'll need to indicate the appropriate parameters in the Monitor setup dialog box. Here's the Monitor to manage the global variable *foo* – note that we set the display name for the monitor, indicate how many decimal places we need, and alter the font size if necessary.



Here's the Monitor to manage the derived value using the *sqrt-foo* reporter. Notice here we've indicated a different display name, and chosen to set the number of decimals to 4 to provide a little more precision.



Plot

The NetLogo plot capability can display 2-dimensional plots of data as your model is running. Plotting features are also found in many programming languages and modeling environments, but can be difficult to set up and use. Fortunately, the NetLogo implementation of plots is very straightforward, yet provides powerful features for visualizing and analyzing model data. There is a lot more in the online documentation in the Programmers Guide under Plotting, so check there for complete details. We'll proceed with an extended example to illustrate some of the key features.

Let's say we have a pair of variables that we'd like to keep track of in our simulation: *stock-price-one* and *stock-price-two*. We're modeling the price fluctuations of these two stocks, and we'd like to plot of these two variables over time, much like a New York Stock Exchange daily trading price chart. First, let's look at what code we need to set up in the model, and then we'll look at how to setup a plot widget on the Interface tab. Our initial model code is shown below – we don't have any plot features incorporated yet, and the stock prices are just going to assume random values between 0 and 99 units.

```
;
; example to demonstrate use of plots in NetLogo
;
```

```
globals [
     stock-price-one
     stock-price-two
]

to setup
     clear-all
     reset-ticks
end

to go
     set stock-price-one (random 100)
     set stock-price-two (random 100)
     tick
end
```

We need to define a procedure to do the plotting, and we need to invoke the procedure repeatedly in our model to update the plot. The first step is to define the plot procedure. We start by specifying which plot widget we want to write to using *set-current-plot* – in this simple case, there's only one plot (named "Stock Prices") so it's not confusing, but in a more complex model you could have several plots and you'll need to specify which one you want. Next, we need to pick out which "pen" on the plot we want to use. You can plot multiple lines on a single graph, and each line is managed by a specific "pen." Here, we've indicated via the *set-current-plot-pen* command that we want to write to the "Stock One" pen on our graph. Now we actually provide the pen with a value to write by calling the *plot* command and passing the *stock-price-one* value to plot. Since we're plotting prices for both Stock One and Stock Two, we now need to repeat some steps to indicate that we also want to plot *stock-price-two*. We call *set-current-plot-pen* and indicate our desired pen, which in this case is "Stock Two." Next we indicate what value should be plotted by this pen by calling the *plot* command with *stock-price-two*. Here's the new procedure we need to include in order to plot our stock prices.

```
;
; example code indicating how to plot some stock prices
;
to plot-prices
     ; choose the plot widget we want to write to
     set-current-plot "Stock Prices"

     ; pick out a "pen" to write on plot and then plot a value
     set-current-plot-pen "Stock One"
     plot stock-price-one

     ; pick out a "pen" to write on plot and then plot a value
     set-current-plot-pen "Stock Two"
     plot stock-price-two
```

```
end
```

Next, we need to invoke our new plotting code so that it gets called periodically and our plot will be displayed on the interface tab. To do this, we simply call the new plotting procedure from within our *go* procedure, which will run the procedure once per model time cycle. The new *go* procedure should look like this.

```
to go
     set stock-price-one (random 100)
     set stock-price-two (random 100)
     plot-prices
     tick
end
```

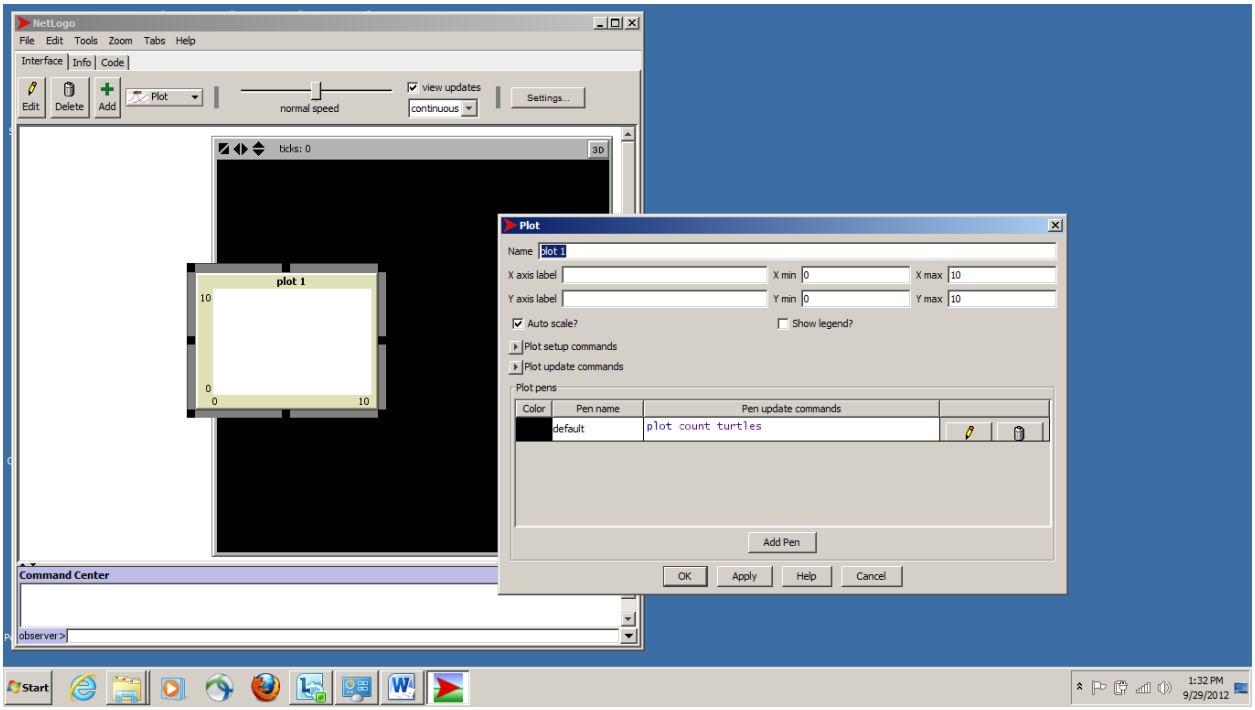
Our complete program should now look like this.

```
;
; example to demonstrate use of plots in NetLogo
;
globals [
     stock-price-one
     stock-price-two
]

to setup
     clear-all
     reset-ticks
end

to go
     set stock-price-one (random 100)
     set stock-price-two (random 100)
     plot-prices
     tick
end


to plot-prices
     ; choose the plot widget we want to write to
     set-current-plot "Stock Prices"

     ; pick out a "pen" to write on plot and then plot a value
     set-current-plot-pen "Stock One"
     plot stock-price-one
```

```
      ; pick out a "pen" to write on plot and then plot a value
      set-current-plot-pen "Stock Two"
      plot stock-price-two
end
```

Now that the code is set up, we need to go over to the Interface tab and set up the Plot widget. Choose Plot from the Interface tab. This will open up a Plot dialog box with the default settings as follows. Notice the plot widget is placed on the left, and the configuration for the plot is located on the right. We'll need to configure the plot with some new values.

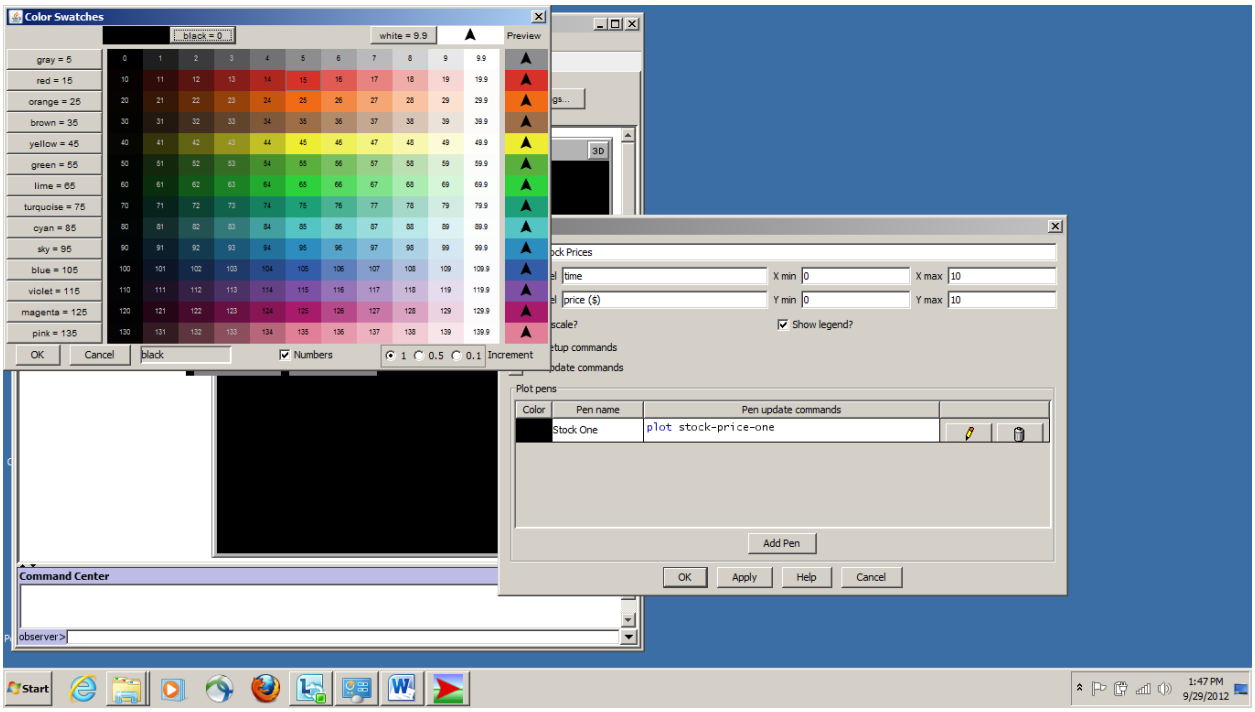
When you create your plot you'll see the dialog box shown below. You should note that one of the columns is labeled "Pen update commands." There are two ways you can setup plots in NetLogo. If you are making relatively simple plots you can place the code directly in the plot itself. In the example below you can see it starts with code there: plot count turtles. If you are going to use the plotting code found above then remove all the code in that box. However, in this relatively simple example, you could simply place the `plot stock-price-one` and `plot stock-price-two` lines from the plot-prices procedure into the plot itself and accomplish the same thing. This can be found in the next figure. For stylistic reasons we recommend not controlling your plots via the pen update commands and writing out the code in the procedures tab. We prefer this because you have more control over the plots and you always know where to find the code that is impacting your plots rather than needing to look in multiple places.



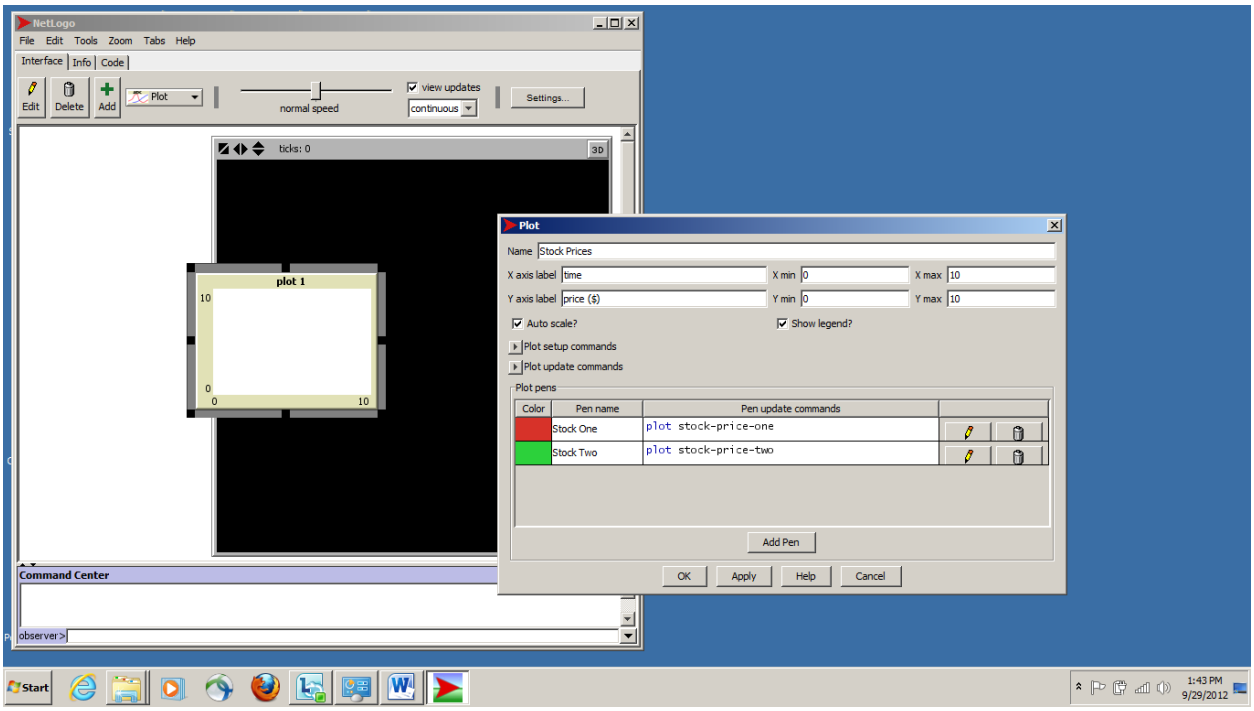
Fill in the fields as follows:

Name:	Stock Prices		
X axis label:	time	X min: 0	X max: 10
Y axis label:	price ($)	Y min: 0	Y max: 10
Check	Auto Scale?		
Check	Show Legend?		

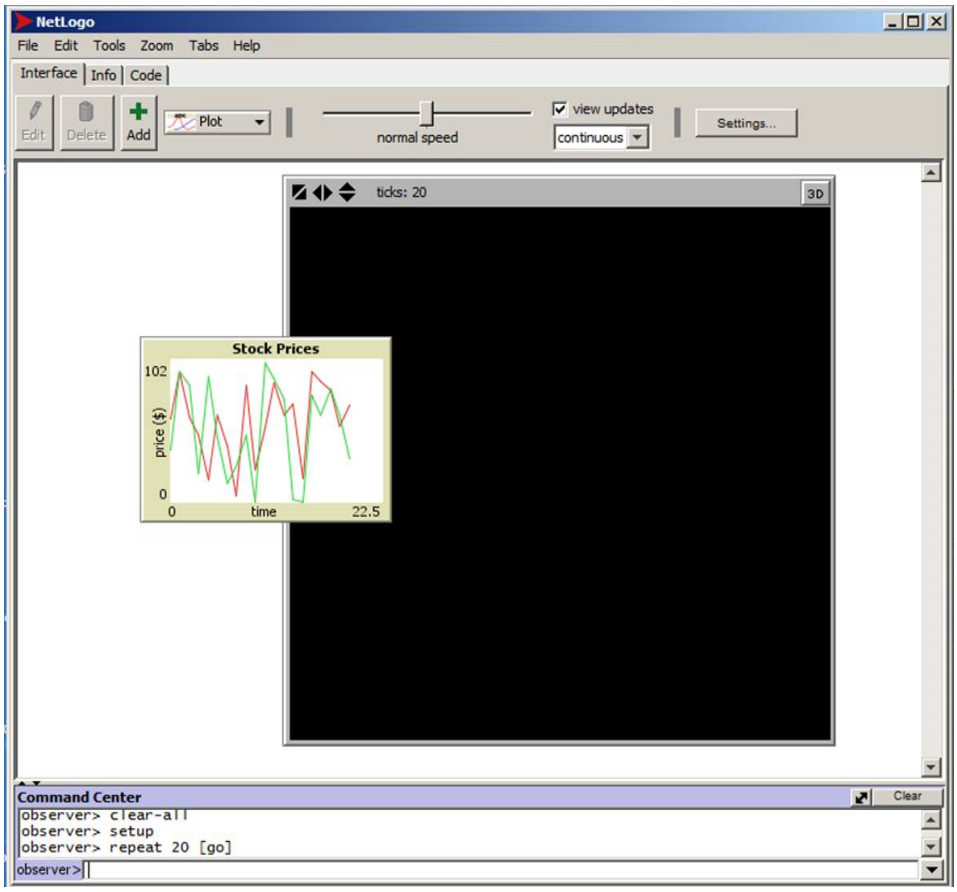
We need to use exactly the same pen names "Stock One" and "Stock Two" here that we previously specified in our plotting code, otherwise the plot widget will get confused. Click on "Pen Name", and change the default pen name "default" to "Stock One". Now click on the "Pen update command", and change the default string "plot count turtles" to "plot stock-price-one". Now click on the dark colored patch to the left of "Stock One" so we can configure the color of the pen. The color selection dialog box will come up as follows. Pick red (#15) for this pen, and press OK.



We now need to add the second pen and plot commands, so let's click the "Add pen" button at the bottom. Change the pen name from "pen-1" to "Stock Two". Change the "Pen update commands" to "plot stock-price-two". Press the color block on the left to bring up the pen color selection dialog box, and this time choose green (#65) and press OK. At this point, your Plot configuration dialog box should look like this. Press OK to finish up the configuration for your plot.



To try out your new plot, type in *setup* at the observer command prompt, then type in *repeat 20 [go]* at the command prompt. Your Stock Prices plot should be updated with values for the prices for Stock One and Stock Two, and your output should look something like this.



Hopefully this has shown you how powerful the plotting features are in NetLogo. There's a lot more to cover, so check the online documentation and review the examples in the model library and sample code library.

Output

NetLogo allows you to define a special partition of the screen called the "Output" on which you may write messages as the model is running. You can always use the command center, but there may be cases where you want to distinguish output between the command center and someplace else. Setting up an output widget is accomplished by selecting Output from the Interface tab chooser. You'll have a generic-looking window when you are done, and you can write things to this window using a special set of output commands. These are *output-show*, *output-write*, *output-type*, and *output-print*, and they work very much the same as the standard *show*, *write*, *type*, and *print* commands except that their values are written to the Output window instead of the Command center.

Note

The Note feature allows you to include static text messages and labels on your model interface. This is useful if you want to organize buttons, sliders, or other features with some informative text, or if you'd like to include credits or other metadata about your model on the interface screen itself. Note works like a text box in an editor or presentation program – it just defines a box in which you can write text.

Control Flow

NetLogo programs include the familiar control flow mechanisms like branching and loops found in other languages, and some NetLogo specific ones as well.

Ask

The *ask* operator is used when you want to request an agent or an agentset to execute a block of commands.

```
;
; examples of ask command
;
; ask one agent (in this case, agent # 17)
;    to color itself cyan
;
let my-agent turtles with [who = 17]
; one could also reference turtle 17 directly with: turtle 17
ask my-agent [set color cyan]

; ask a set of agents (the north-east quadrant)
;    to color themselves green
;
let northeast-patches patches with [(pxcor > 0) and (pycor > 0)]
ask northeast-patches [set pcolor green]
```

If and If-Then-Else

The *if-then* and *if-then-else* programming constructs in NetLogo are implemented by *if* and *ifelse* constructs, respectively. In both forms, a conditional expression is evaluated. In the basic *if-then* form, the statement group in brackets following the expression is executed if the expression evaluates to *true*, and nothing happens if the statement evaluates to *false*. As a matter of coding style, these examples enclose the conditional expression in parentheses, however, this is not required by NetLogo syntax.

```
;
; example of an if-then expression
;
;   if (expression)
;      [statement-block]
;
let done? false
let counter 0
if ((done? = false) and (counter < MAX-COUNT))
[
     do-something
     set counter (counter + 1)
]
```

In the *if-then-else* form using *ifelse*, the first statement group in brackets following the expression is executed if the expression evaluates to *true*, and second statement group in brackets is executed if the expression evaluates to *false*. The use of a pair of bracketed expressions to manage the *true* and *false* blocks may seem confusing to new NetLogo programmers, as it appears that the *false* block is just "hanging" in space.

```
;
; example of an if-then-else expression
;
; ifelse (expression)
;  [expression-true-statement-block]
;  [expression-false-statement-block]
;

let done? false
let counter 0
ifelse ((done? = false) and (counter < MAX-COUNT))
[
     ; this block executed if condition is true
     do-something
     set counter (counter + 1)
]
[
     ; this block executed if condition is false
     do-something-else-instead
     set counter (counter + 1)
]
```

While Loop

The *while* loop in NetLogo is implemented by the *while* construct. The NetLogo *while* loop evaluates a condition, and as long as the condition is true, the statements in the *while* loop will be executed. One subtle difference from the *if* and *ifelse* statements is that the conditional expression in the *while* loop requires brackets, whereas the conditional expression in the *if* and *ifelse* statements does not require brackets. As with while loops in other programming languages, you need to remember to test for the termination condition somewhere inside your statement block, otherwise the system will execute your statement block forever in an infinite loop.

```
;
; example of a while loop
;
; while [ condition ]
;    [ expression(s) ]
;

let done? false
while [ (done? = false) ] ; note [] brackets for test expression
[
```

```
     ; call a procedure to do something
     do-something

     ; now test to see if the while loop is finished yet
     if done? [ test-if-done-yet ]
]
```

Repeat Loop

Some programming languages provide an analogue to the *while* loop called a *repeat-until* loop. NetLogo has a variant of this, implemented using the *repeat* construct. The NetLogo *repeat* is used to execute commands in a block a fixed number of times. In the example shown below, the procedure *do-something* will be executed 5 times. Note that the semantics of the *repeat* command are **not** like a *repeat-until* with a conditional expression in some languages, but more like a for-loop in Java or C++ in which a block of statements are executed some fixed number of times.

```
;
; example of a repeat loop
;
; repeat n
;    [ expression(s) ]
;

repeat 5
[
     do-something
]
```

List Iteration

NetLogo provides support for a list data type, and includes a variety of functions to manage lists. The *foreach* operator can be used to iterate over the contents of a single list or multiple lists.

In the single list form, the *foreach* operator is supplied with a list, and a block of commands is executed for each element of the list. The *?* operator is used to access the current member of the list during the iteration process.

```
;
; example of a foreach on a single list
;
; foreach [ list ]
;   [ expression(s) ]
;
;

let my-value 0
foreach [ 1 3 5 7 11 13 17 19 ]
[
```

```
     ; select the i-th list element using "?" operator
     set my-value ?

     ; do something (here just print the value)
     type "processing list element " print my-value
]
```

In the multiple list form, the *foreach* operator is enclosed in parentheses. The *foreach* operator is supplied with two or more lists, each of which must be the same length. The *foreach* operator then iterates over each list simultaneously, with the *?1* operator providing access to the elements of the first list, the *?2* operator providing access to the elements of the second list, and so forth.

In the example, two short lists are supplied as input. The *foreach* operator extracts pairs of terms from each list, adds these terms together to calculate a sum, then prints the result to the console.

```
;
; example of a foreach on multiple lists
; note: expression enclosed in parenthesis
;
; ( foreach ( [ list-1 ] [ list-2 ] )
;   [ expression(s) ]
; )
;

let list-one-element 0
let list-two-element 0
let sum-of-ith-terms 0
let i 0

( foreach [1 3 5 7 9] [ 2 4 6 8 10] ; note use of parenthesis
[
     ; fetch the next element from list 1
     set list-one-element ?1

     ; fetch the next element from list 2
     set list-two-element ?2

     ; add them up
     set sum-of-ith-terms (list-one-element + list-two-element)

     ; print a message
     type "sum " type i type "-th elements = "
     print sum-of-ith-terms

     ; increment i. note that this is NOT needed for list
     ; iteration, but is only used to augment the message
     ; being printed to the console.
```

```
      set i (i + 1)
]
)
```

Procedures

There are a large number of built-in procedures and functions in the NetLogo environment. Procedures that return values are often called functions in other programming languages, although in NetLogo they are called *reporters* as they report a value to the caller.

User-Defined Procedures

In NetLogo, a user-defined procedure is a set of statements identified by a procedure name. User-defined procedures are used to accomplish things in your model. In agent-based modeling, this often involves selecting a particular set of agents or patches, evaluating some conditions, and updating the state of the model world.

Some commonly used NetLogo conventions are to include a procedure named *setup* to manage all the initialization and preparation for the model, and to include a procedure named *go* to manage the running of the model. In many models, these are linked to buttons on the main interface screen to enable the user to quickly initialize and run a model.

NetLogo procedures use the keyword *to* followed by the procedure name to indicate the start of the procedure, and the keyword *end* to indicate the end of the procedure. The form of a user defined procedure is shown below.

```
;
; example of a user-defined procedure
;
;


; create a type of agent called "firm"
breed [firms firm]

; endow the firm with certain attributes
firms-own [
      number-employees
      years-in-business
      total-assets
      total-liabilities
]

; write a procedure to randomly set up firm attributes
to firms-setup
  ask firms [
        set number-employees (random 100)
        set years-in-business (random 10)
```

```
    set total-assets (random 100000)
    set total-liabilities (random 50000)
]
end
```

User-Defined Functions (Reporters)

In NetLogo, a reporter is a user-defined procedure that returns a value to the caller. The form of a reporter is similar to a procedure except that reporters begin with the keyword *to-report* instead of the keyword *to*. In addition, a reporter **must** indicate what value should be returned to the caller using the *report* operator. The following is an example of a reporter that counts the number of agents in the model.

```
;
; example of a reporter
;
to-report get-agent-population-size
     let pop-size 0
     set pop-size (count turtles)
     report pop-size
end
```

Passing Parameters

In NetLogo, it is possible to pass parameters to both procedures and reporters. The form is very similar to what we just discussed, except that the declaration includes brackets to hold the parameter or parameters being passed in. Multiple values can be passed as parameters, but the number of parameters in the declaration must match the number passed when the procedure or reporter is being used.

In the example below, a procedure is defined that expects one parameter: the new color to paint the landscape. The procedure will accept the parameter and use this to update the patch colors in the model.

```
;
; example of a defining procedure that uses a parameter
;
to color-landscape [ new-color ]
     ask patches [ set pcolor new-color ]
end
```

Here's how to invoke this procedure in the model code somewhere. Note that the user defined procedure name *color-landscape* is followed by the parameter *green*.

```
;
; example of invoking the procedure that expects parameters
;
color-landscape green
```

If we want to use a reporter to compute a value based on some parameter(s), we use the same basic form. The reporter is defined with the parameter declaration in brackets, and the reporter is invoked with the desired parameter value(s).

Here's an example of a reporter that will examine a list of numbers and return the largest positive value in the list.

```
;
; example of defining a reporter with a parameter
;
to-report find-biggest-positive-in-list [ my-list ]
     let my-max 0
     foreach my-list [
          if (? > my-max) [
               set my-max ?
          ]
     ]
     report my-max
end
```

Here's how to invoke the reporter in the model code.

```
;
; example invoking reporter in the code somewhere
;
let my-big-value 0
set my-big-value find-biggest-positive-in-list [1 513 7 2089 27]
```

It's important to note that as coded, the *find-biggest-positive-in-list* reporter will fail if it is invoked with something other than a list as a parameter, such as an agentset for example. Due to the flexible typing in NetLogo, you have to be especially careful if you are using type-specific operations or making type-specific assumptions in your procedures or reporters.

NetLogo Extensions (Arrays, Tables, Geographic Info Systems, Sounds, etc.)

NetLogo includes a number of libraries containing software associated with certain problem domains, such as processing mathematical arrays and tables, geographic information systems, sound processing, robotics, etc. The list of currently supported libraries can be found in the online NetLogo Programmers Guide under NetLogo Extensions.

Extensions are invoked by using the extensions keyword and including in brackets the name of the extension or extensions you want to use. This declaration must occur at the beginning of your model code **before** you declare any breeds or variables. After this point, you can invoke functions from the extensions library and use these features in your model. The general form for using an extension procedure or reporter is *extension-name:procedure-name* or *extension-name:reporter-name*. Consult the Programmers Guide for specifics on the available features of

the extensions. Here's an example of how to include the extension for arrays into a model and how to invoke some of the array-specific functions.

```
; must begin with extensions keyword
; followed by list of extension(s) to be used
;
extensions [ array ]

;
; example of including an extension
; and using some extension functions
;
to array-demo
     ; declare local variable named x, populate it
     let x array:from-list (list 1 3 5 7 9 2 4 6 8)

     ; get the length of the list
     type "length of list x = " print array:length x

     ; fetch the 4th item (note array begins with item 0)
     type "4th item of list x = " print array:item x 3

     ; update first element of the list to be 137
     type "updating 1st element of x to " print 137
     array:set x 0 137

     ; print out final result
     type "updated x = " print x

end
```

Input/Ouput

NetLogo provides a variety of features for handling input and output. We've already seen several examples of writing values to the console, and NetLogo can also read and write files of various formats. As with all the NetLogo functions, these are discussed in more detail in the Programmers Guide, so check there for additional details.

Console I/O

The NetLogo *type*, *print*, and *write* operators are used to write data to the console. The methods vary with respect to whitespace formatting conventions and line breaks.

type

The *type* operator will output a single value to the console. The value can be numeric, string, or even an agent. The value is printed without leading or trailing whitespaces, and no line break occurs. This is useful if you need to concatenate several elements and/or labels to a single line of output. Here's an example (note the use of embedded white spaces in the labels: *type* will not supply whitespaces between elements).

```
;
; example of the type function
;
to demo-type-function

     ; assign some values and some labels
     ;
     let x 3.5
     let x-label "quantity of x = "

     let y 5
     let y-label "quantity of y = "

     ; now put values on the console
     type x-label             ; print label
     type x type " , "        ; print value and separator " , "

     type y-label type y type " "  ; print label, value,
                               ; and a space

end
```

Entering *demo-type-function* at the observer console results in the following message being typed on the console (without a line break). Notice that we have various numbers of *type expression* pairs, and NetLogo does not care how many of these you put on one line of code.

```
quantity of x = 3.5 , quantity of y = 5
```

print

The *print* operator works like the *type* operator, except that it includes a line break afterward. Here's an example.

```
;
; example of the print function
;
to demo-print-function

     ;
     ; assign some values and some labels
     ;
     let x 3.5
     let x-label "quantity of x = "

     let y 5
     let y-label "quantity of y = "
```

```
     ; now print these expressions to the console
     type x-label print x
     type y-label print y

end
```

Notice that our *x-label* is typed without a line break, then our *x* value is printed with a line break afterward. We do the same thing again in the next line for *y-label* and *y*. It's OK to mix *type* and *print* on the same line, although usually you'll want to *print* the last element to get the line break. Entering *demo-print-function* at the observer prompt results in the following output on the console. As with *type*, *print* does not include extra white spaces around elements, so note again the use of embedded white spaces in the labels.

```
quantity of x = 3.5
quantity of y = 5
```

write

The *write* operator works much like the *type* operator. A single value is written to the console. The main difference from the *type* operator is that *write* includes whitespaces between values and encloses strings in quotes, whereas *type* does not. The *write* operator is designed to produce output that is compatible with the *read* operators for data input.

```
;
; example of using write operator for output
;
to write-demo
     let x 3
     write "first value is " write 1.5
     write ", the x value = " write x
end
```

Entering *write-demo* at the observer prompt will produce the following output on the console. Note the automatic insertion of white spaces between elements and the automatic quoting of string data.

```
"first value is " 1.5 ", the x value = " 3
```

word

Finally, although not technically an I/O operator, the *word* operator allows you to concatenate data together and treat it as a single block for console output. This is especially useful if you have several elements that logically fit together in a message.

```
;
; example of using word to concatenate values for output
;
to word-demo
     let x 3
```

```
     print (word "the value is " 1.5 ", the x value = " x)
end
```

Entering *word-demo* at the observer prompt will produce the following output. Notice the differences with respect to quotes in this case.

```
the value is 1.5, the x value = 3
```

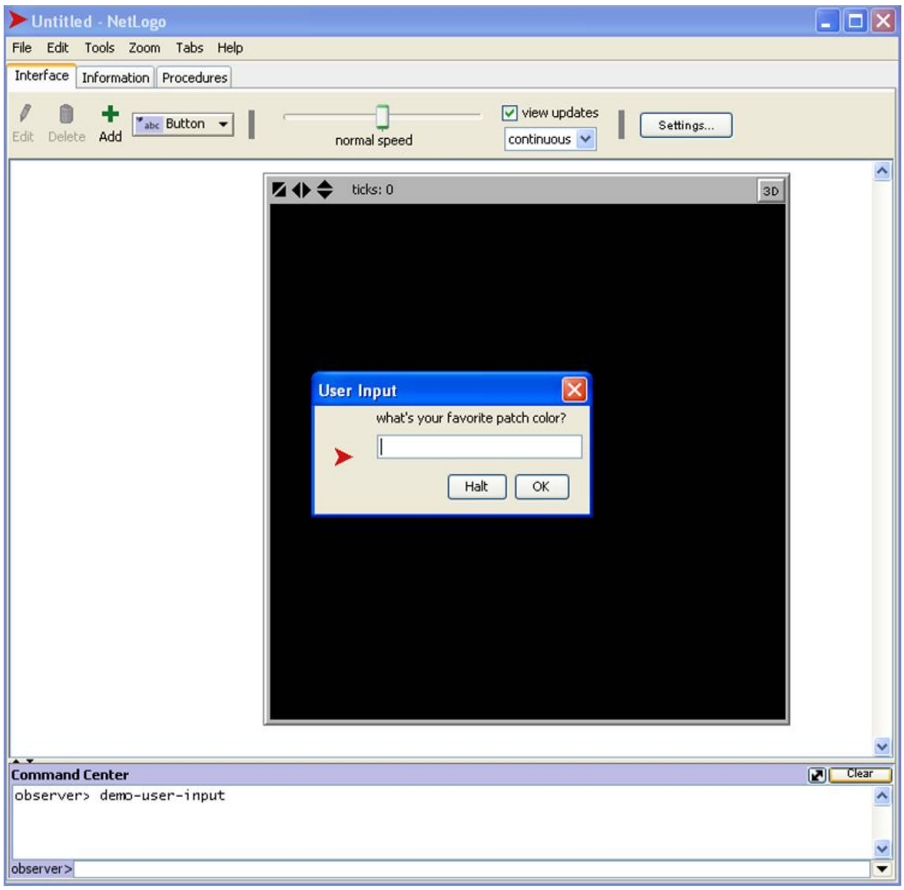
User Interaction

NetLogo provides a family of built-in operators to manage interactive user input. These include the *user-input*, *user-message*, *user-one-of*, and *user-yes-or-no?* operators. Each of these user interface operators opens a small dialog box on the screen and allows the user to specify inputs or read a message. As with all the NetLogo functions, these are discussed in more detail in the Programmers Guide, so check there for additional details. One note: these operators treat responses from the user as strings, so you'll need to pay attention to how you interpret the value returned by the *user-input* or other user interface operators.

Here's an example of how to ask the user interactively what color he or she would like to use for the default background patch color. The code is not particularly robust, since there are only a few choices of colors that that actually work, but it will illustrate the point. In the code snippet below, for example, strings returned by the user are compared with some (but nowhere near all) of the built-in color types, and if a match is found, the patches are set to the desired color.

```
;
; example of user interactive input
;
to demo-user-input

  let chosen-color black
  set chosen-color user-input "what's your favorite patch color"

     if (chosen-color = "red")
          [ ask patches [set pcolor red] ]
     if (chosen-color = "orange")
          [ ask patches [set pcolor orange] ]
     if (chosen-color = "yellow")
          [ ask patches [set pcolor yellow] ]
     if (chosen-color = "green")
          [ ask patches [set pcolor green] ]
     if (chosen-color = "blue")
          [ ask patches [set pcolor blue] ]
end
```

This will result in the following dialog box for the user.



File I/O

NetLogo has a variety of operators to manage reading and writing files. As with all the NetLogo functions, these are discussed in more detail in the Programmers Guide, so check there for additional details (in particular, see the Programmers Guide section on **Input/Output** and **File** operations). We'll summarize some of them here.

user-directory, user-file, user-new-file

The *user-directory*, *user-file*, and *user-new-file* are interactive operators that open a dialog box and allow the user to specify a directory, existing filename, or new file name, respectively. These work much as the user-input functions we discussed earlier. Here's an example of how you could ask the user to specify the filename of an existing file.

```
;
; example of user-file operator
;    want to get a data file name from the user at run-time
;
to process-user-data-file
     let file-name " "
     set file-name user-file
```

```
     ; now do something with the file, like maybe open
     ; it and read its contents…

end
```

file-read, file-read-characters, file-read-line

The *file-read*, *file-read-characters*, and *file-read-line* are used to read data from a file. The *file-read* operator will read whitespace-separated values from a file one at a time. The *file-read-characters* operator will read a specified number of characters from a file. The *file-read-line* operator will read a line of input from a file. Here's an example code snippet of how you might use some of these operators to initialize your simulation with data from a configuration file.

```
;
; example of file-read operators
;
; assume our data file is named "data-file.txt"
; assume it is formatted like this
;
; simulation name
; one-line description
; num-red-agents num-green-agents max-time-ticks
; <EOF>
;
to get-setup-data
     let sim-name " "
     let sim-description " "
     let num-red-agents 0
     let num-green-agents 0
     let max-time-ticks 0

     ; open file
     file-open "data-file.txt"

     ; do some reads from the file
     set sim-name file-read-line
     set sim-description file-read-line
     set num-red-agents file-read
     set num-green-agents file-read
     set max-time-ticks file-read

     ; close file as we're done
     file-close

     ; now do something with these setup variables…
     ;
end
```

file-write, file-print, file-type, and file-show
The *file-write*, *file-print*, *file-type*, and *file-show* operators are used to write data to a file. The *file-write* operator writes a single value to the file, without a carriage return. It is conceptually similar to the *write* operator we saw earlier for console I/O, in that it automatically includes white spaces around elements and encloses string data in quotes. The *file-print* operator prints data to a file, much as the print operator, and includes a line break at the end of the line being written. The *file-type* operator types data to a file, without a line break at the end, much as the *type* operator we saw earlier. The *file-show* operator prints data to a file and includes the identifier of the calling agent in the output (which can be useful for debugging purposes).

file-open, file-close, file-delete
The *file-open* and *file-close* operators are used to open and close files, respectively. NetLogo only allows one open file at a time, so there is no ambiguity about which file handle is being used – if you've opened a file, that's the one you can read or write. You need to close it and open another one if you have different files you need to access in your simulation.

After a file is opened, the first operation on that file determines if the access mode will be read or write. This is different from other programming languages where the file mode is specified at the time of the file open. Once a file has been established in read mode, it can only be read from until it is closed. Similarly, once a file has been established in write mode, it can only be written to until it is closed.

When opening a file for writing, NetLogo assumes by default that the file is to be accessed in append mode. If you want to start from fresh, you'll need to delete the file first using *a file-delete* operation, open it anew and then write to it. By default, NetLogo will attempt to read and write from the directory in which it is being run, so if you want to access files elsewhere, you'll need to include system-appropriate path information.

file-flush
The *file-flush* operator forces any pending I/O to be completed. It's most useful when you've been writing to a file and you want to make sure all the I/O is complete before you close it. You may not need to use this very much (or even at all) depending on your system I/O characteristics, but we've included an example here just for demonstration purposes.

The following example demonstrates all the file I/O operators we've mentioned so far.

```
;
; example of file I/O operators
;

to demo-file-io
     ; declare some values to write
     let x 1.0
     let y 3.0
     let num-agents 10
```

```
  let outfile-name "output.txt"

  ;
  ; carefully is a method to try a command block
  ; and recover gracefully if it fails. The syntax
  ; works like this:
  ;
  ; carefully [ try block cmds ] [ fail block cmds ]
  ;

  ; first try to delete existing file
  carefully
  [ file-delete outfile-name ]
  [ print (word "error deleting " outfile-name) ]

  ; now try to open output file
  carefully
  [ file-open outfile-name ]
  [ print (word "error creating " outfile-name) ]

  ;
  ; write values to the output file
  ;
  file-type "value of x = "
  file-write x
  file-print " "

  file-type "value of y = "
  file-print y
  file-print " "

  file-show num-agents
  file-print " "

  ;
  ; flush output and close file
  ;
  file-flush
  file-close
end
```

Running the *demo-file-io* procedure at the observer prompt will result in a file name "output.txt" being written, with contents as follows.

```
value of x = 1
value of y = 3
```

```
observer: 10
```

We've covered a lot of detailed technical material, so now let's see how to apply some of this to a modeling problem. Up next, we'll look at extending a classic model to study a new problem, and also apply some of the technical insights we've gained from this section.

4. Using Models for Research

When beginning a project in NetLogo, the models library is a great place to start. There are a wide variety of models across various domains, so you're likely to find something that's at least somewhat similar to what you're trying to do. As an extended case study, let's consider how we might use a well-known model for a research task.

Let's say we are doing social science research in urban housing preferences. We've already reviewed the classic Schelling Segregation model, and we've decided to use this as a starting point for studying a variant of the original problem. In particular, we'd like to know if there are segregation effects caused by class-specific similarity preferences that cause different housing effects than when using one global similarity preference. In real-life, this could correspond to differences in similarity housing preferences by age, ethnicity, or economic status, for example.

To investigate this, let's modify the existing standard Schelling Segregation model to include the ability for separate and distinct similarity preferences among the different agent classes. To keep things simple, we'll stick to just using two classes of agents: greens and reds. For each class of agent, we'll create a user-definable parameter that controls the similarity preference for that agent class. In the model code, we'll use these parameters to decide whether or not our agents are happy, and consequently if they are prone to relocate or to remain where they are. Let's go through the process to get an idea of how to proceed.

Updating an Existing Model

Start up NetLogo, choose the Models Library, go to the Social Science folder, and open up the Segregation Model. We'll need to modify the existing model interface, and we'll need to make some code changes in the procedures. Let's start by saving a new copy of the model, called *Segregation-updated.nlogo*, to our workspace. Click on File, Save-As, and specify *Segregation-updated.nlogo* as our file name. NetLogo will now save this file for us.

On the Interface tab, let's add a couple of new sliders to handle the class-specific agent preferences. Add a new slider named *%-similar-wanted-green*, and another one named *%-similar-wanted-red*. Also, to keep things cleaner in the code, let's add a switch named *model-extensions* – we'll use this switch to separate out the new code we add from the existing code in the model.

Now go over to the Procedures tab, and we'll need to make some code changes to implement the new functionality. Before we start anything, however, it's good form to introduce some comments at the top of our model and provide attribution to the original author. Here's a block of comments to start things off.

```
; -----------------------------------------------
; Name:
```

```
; Segregation-updated.nlogo
;
; Description:
; extensions to the Schelling segregation model
; to investigate variable preferences
; for similarity among different agent classes.
;
; Credits:
; This model is based on the Segregation model
; from the NetLogo library. Original code is
; Copyright 1997 Uri Wilensky. All rights reserved.
; The full copyright notice is in the Information tab.
; -------------------------------------------------------
```

Under our *globals*, let's add some additional variables to keep track of the percentage of similar agents and their happiness, distinguished by their color. We'll add *percent-similar-red* and *percent-similar-green* to measure the percentage of red agents near a red agent and similarly the percentage of green agents near a green agent. We'll add *percent-unhappy-red* and *percent-unhappy-green* to measure the happiness of red and green agent classes, respectively. Make the following changes to the globals block. To make things easier to see, new code is indicated in bold font.

```
globals [
     percent-similar     ;; on the average, what percent of a
                         ;; turtle's neighbors
                         ;; are the same color as that turtle?
     percent-unhappy     ;; what percent of the turtles
                         ;; are unhappy?

     ;
     ; variables to keep track of similarity and unhappiness
     ; by agent-class if extended model is used
     ;
     percent-similar-red
     percent-unhappy-red

     percent-similar-green
     percent-unhappy-green
]
```

Now we'll need to add some variables to each turtle to keep track of how many agents of each color are nearby. We'll add *similar-nearby-red* and *similar-nearby-green* to count the number of red and green agents nearby, respectively. We'll use *other-nearby-red* and *other-nearby-green*

to count the number of non-red and non-green agents nearby, respectively. Finally, we'll use *total-nearby-red* and *total-nearby-green* to count the total number of agents of each color nearby.

You might ask why we declared all these variables – do we really need to keep track of both red **and** green adjacency? Strictly speaking, the answer is no - we don't need all these variables. That's because our agent must be either green or red, and if it's red we don't care about green bookkeeping, and if it's green we don't care about red bookkeeping. However, to simplify some calculations and plotting later, it's easier if we declare both sets and only use one. This is a great example of software engineering trade-off: we could probably engineer a more sophisticated solution, but it would introduce greater complexity elsewhere in the model. As a NetLogo developer, you've got a great deal of flexibility in how you design your models, and it is very easy to add complexity very quickly.

Here's the updated turtles-own code we need to use.

```
turtles-own [
   happy?      ;; for each turtle, indicates
               ;; whether at least %-similar-wanted percent of
               ;; that turtles' neighbors are the same
               ;; color as the turtle
   similar-nearby   ;; how many neighboring patches
               ;; have a turtle with my color?
  other-nearby ;; how many have a turtle of another color?
  total-nearby ;; sum of previous two variables

     ;
     ; variables to keep track of color-specific agent
     ; adjacency if extended model is used
     ;
       similar-nearby-red
       other-nearby-red
       total-nearby-red

       similar-nearby-green
       other-nearby-green
       total-nearby-green
]
```

Remarkably, most of the remaining procedures are unchanged. The *setup*, *go*, *move-unhappy-turtles*, and *find-new-spot* procedures are exactly the same as before.

Our *update-turtles* procedure needs to change a bit. We need first to bracket the original code to take advantage of our *model-extensions* switch. If the user has selected Off for model extensions, then we want to run in standard mode just like it came out of the NetLogo box.

We'll isolate the original code blocks by testing the value of *model-extensions*, and executing the original code if *model-extensions* is set to false.

If the user has selected On for model-extensions, then we want to make use of our new variables and calculations. Take a look at the new code for update-turtles and see if you can follow the logic. We're basically evaluating all the red agents and all the green agents separately, and determining how many agents of each type are nearby, which we also use to determine the agents happiness.

```
to update-turtles
  if (model-extensions = false) [
    ask turtles [
      ;; in next two lines,
      ;; we use "neighbors" to test the eight patches
      ;; surrounding the current patch
      set similar-nearby count (turtles-on neighbors)
      with [color = [color] of myself]
      set other-nearby count (turtles-on neighbors)
      with [color != [color] of myself]
      set total-nearby similar-nearby + other-nearby
      set happy? similar-nearby >=
          ( %-similar-wanted * total-nearby / 100 )
    ]
  ]
  if (model-extensions = true) [
    ask turtles with [color = red] [
      set similar-nearby-red count (turtles-on neighbors)
          with [color = [color] of myself]

      set other-nearby-red count (turtles-on neighbors)
          with [color != [color] of myself]

      set total-nearby-red similar-nearby-red + other-nearby-red

      set happy? similar-nearby-red >=
          ( %-similar-wanted-red * total-nearby-red / 100 )
    ]
    ask turtles with [color = green] [
      set similar-nearby-green count (turtles-on neighbors)
          with [color = [color] of myself]

      set other-nearby-green count (turtles-on neighbors)
          with [color != [color] of myself]

      set total-nearby-green
          similar-nearby-green + other-nearby-green
```

```
      set happy? similar-nearby-green >=
          ( %-similar-wanted-green * total-nearby-green / 100 )
    ]
  ]
end
```

Next we need to make some changes to the *update-globals* procedure to account for our new variables. As before, we'll bracket the original code based on the user specified value for model-extensions, so if the user does not want to run our extensions then the model runs in the default mode just like in the NetLogo models library. If the user does want to use the new extensions, then *update-globals* will calculate the similarity counts for both red and green agents, and store these values in the new global variables we declared earlier. Here's the updated code for *update-globals*.

```
to update-globals

  if (model-extensions = false) [
    let similar-neighbors sum [similar-nearby] of turtles
    let total-neighbors sum [total-nearby] of turtles
    set percent-similar
       (similar-neighbors / total-neighbors) * 100
    set percent-unhappy
       (count turtles with [not happy?]) / (count turtles) * 100
  ]

  if (model-extensions = true) [
    ;
    ; handle red neighbors
    ;
    let similar-neighbors-red
       sum [similar-nearby-red] of turtles

    let total-neighbors-red sum [total-nearby-red] of turtles

    set percent-similar-red
       (similar-neighbors-red / total-neighbors-red) * 100

    set percent-unhappy-red
      (count turtles with [not happy? and (color = red)]) /
      (count turtles) * 100

    ;
    ; handle green neighbors
    ;
    let similar-neighbors-green
       sum [similar-nearby-green] of turtles
```

```
    let total-neighbors-green
       sum [total-nearby-green] of turtles

    set percent-similar-green
       (similar-neighbors-green / total-neighbors-green) * 100

    set percent-unhappy-green
       (count turtles with [not happy? and (color = green)]) /
       (count turtles) * 100
  ]
end
```

We're on the home stretch – we just need to fix the plotting code so we can visualize our new variables if the model is being run with the new extensions. The plotting code is found in the *do-plots* procedure. As before, we'll bracket out the existing code with a test of the *model-extensions* switch setting, and execute the new code only if the user has indicated to do so.

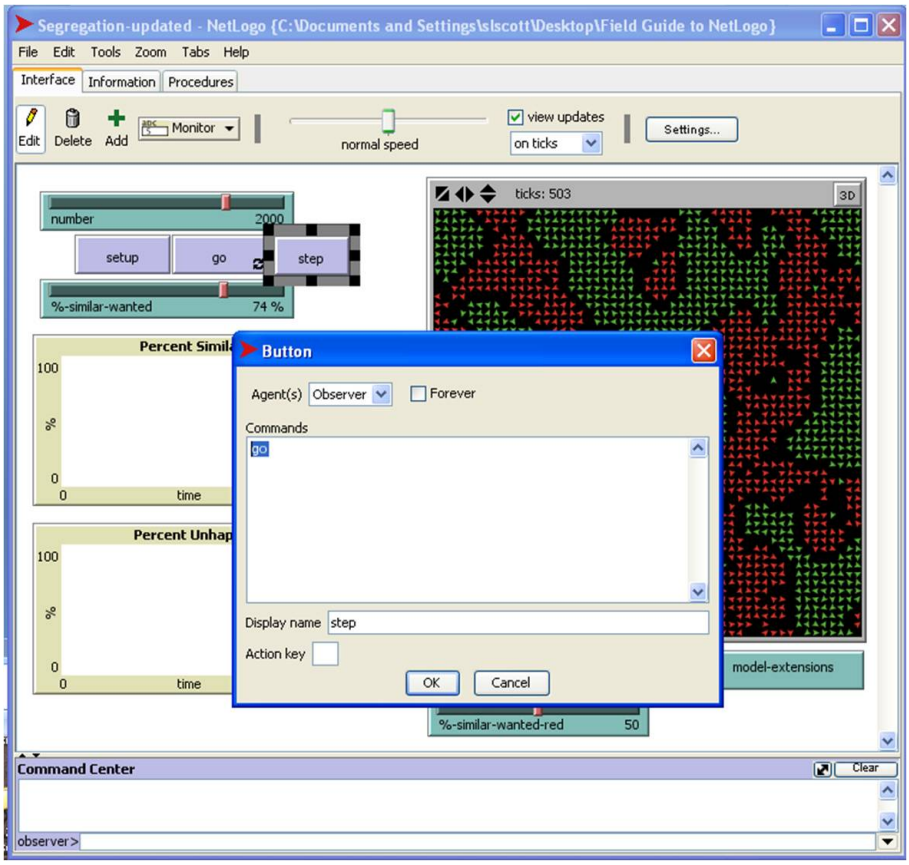
Unlike the other procedures we've seen so far, we do need to make a couple of minor changes to the existing code in this case. Since we want to use the existing Percent Similar and Percent Unhappy plot widgets on the interface tab, we need to add some new *plot pens* to these widgets to draw our new variables as the simulation runs. We need to do this so we are completely clear about which pen we want to use when, otherwise the plot widgets will get very confused. Notice that we've include a couple of *set-current-plot-pen* commands in the original plotting code block, and specified the original pens for these plots.

We include some new plot commands to handle plotting our new global variables if the user is running the model with the extensions enabled. As before, we'll specify which plot widget to use by invoking the *set-current-plot* command. We then specify which new pen we want to use in the plot widget with a *set-current-plot-pen* command, then pass the value to be plotted to the *plot* command. The logic is very similar to the original plotting code, and as before, new code is indicated in bold font.

```
to do-plots
  if (model-extensions = false) [
    set-current-plot "Percent Similar"
    set-current-plot-pen "percent"
    plot percent-similar
    set-current-plot "Percent Unhappy"
    set-current-plot-pen "percent"
    plot percent-unhappy
  ]
  if (model-extensions = true) [
    set-current-plot "Percent Similar"
    set-current-plot-pen "percent-similar-red"
    plot percent-similar-red
    set-current-plot-pen "percent-similar-green"
```

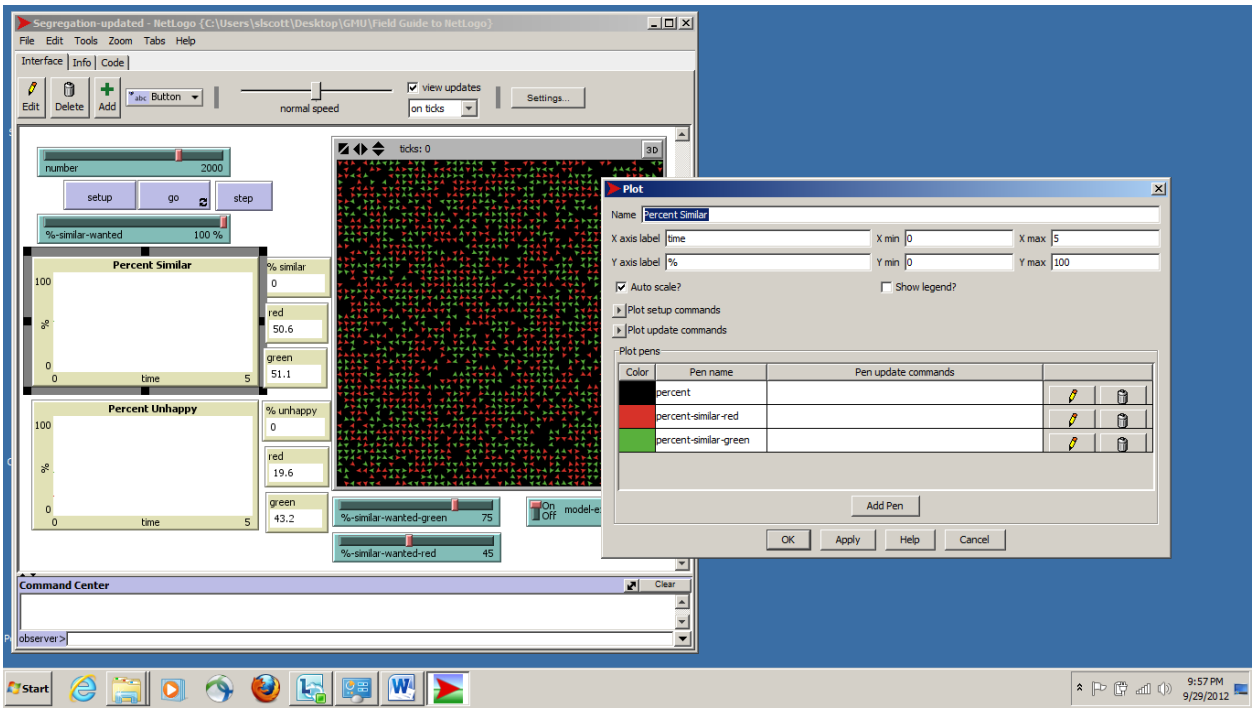
```
    plot percent-similar-green

    set-current-plot "Percent Unhappy"
    set-current-plot-pen "percent-unhappy-red"
    plot percent-unhappy-red
    set-current-plot-pen "percent-unhappy-green"
    plot percent-unhappy-green
  ]
end
```

We're almost done. Let's go back over to the Interface tab, and make a few minor adjustments and add some extra goodies. First, let's add a button to run the go procedure one time step. Choose the Button widget, and enter in *go* for the commands. Name the button "Step", and do NOT check the forever tab – we don't want this button to execute indefinitely. Press OK to add this button to the interface.

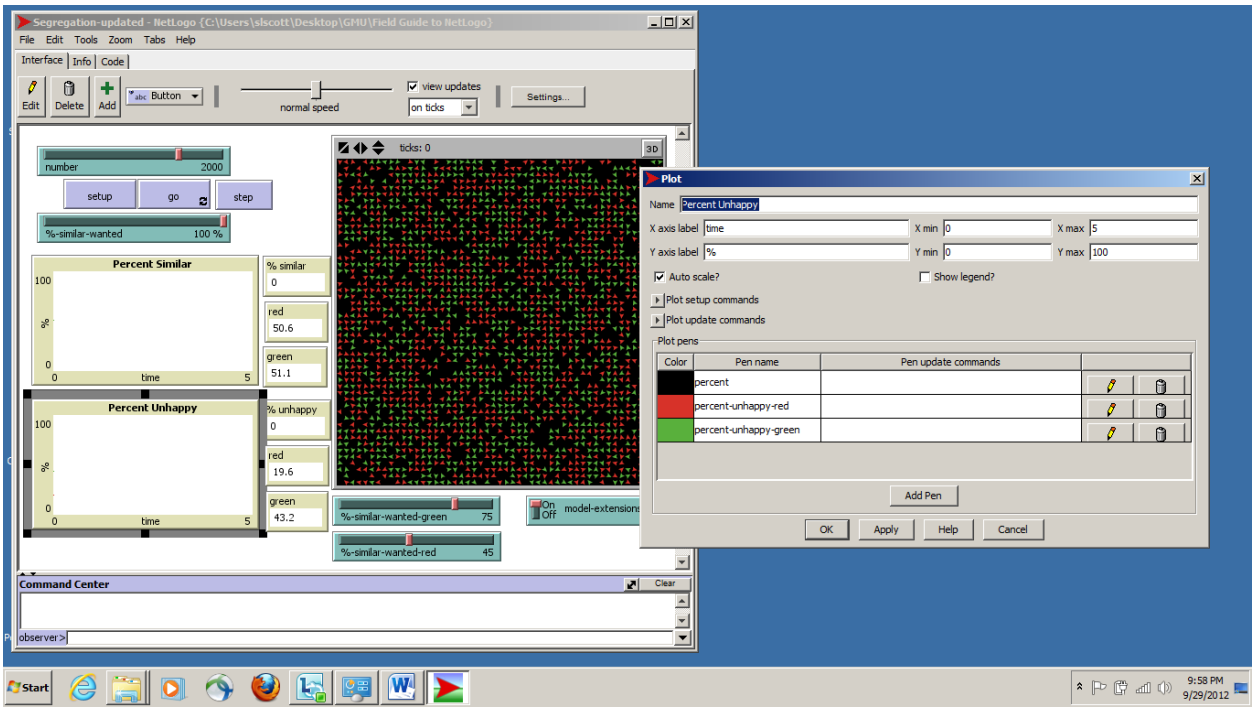


We need to edit our plot widgets to make use of the new pens we discussed in the *do-plots* code. Right-click on Percent Similar, choose "edit" and go to the dialog box that comes up. Press the "Add pen" button, add a new pen named "percent-similar-red". Under the "color" column, click on the grey box and open the color selection dialog box. Choose a nice shade of red, such as #15. Press OK to exit the color selection dialog box. Now press "Add pen" again, and add a new pen named "percent-similar-green". As before, click on the grey box under "color", and

open the color selection dialog box. Choose a shade of green, such as #55. We're going to explicitly manage the plot widget from within our code, so we don't need to include any pen update commands. Your completed plot configuration dialog box should look like this. Press OK to save the changes.



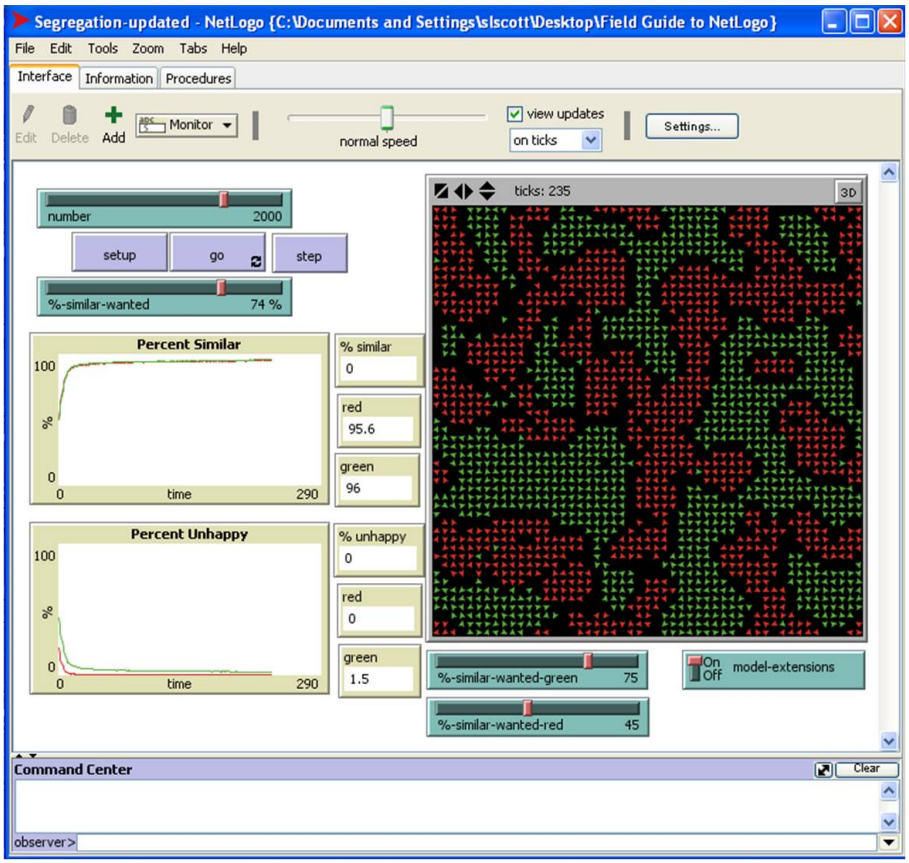
Now right-click on the Percent Unhappy plot widget, choose "edit", and go to the dialog box. Create new pens for "percent-unhappy-red" and "percent-unhappy-green", and color them red and green, respectively. Your completed plot configuration dialog box should look like this. Press OK to save the changes.



It would be nice to keep a visual tab on the red and green percent-similar values and the red and green percent-unhappy values. Let's add some monitors to do that. Choose the Monitor widget, and a blank monitor comes up. Type in "percent-similar-red" for the monitor, set the display name to "red", and set the decimal places to 1, and leave the font size at the default setting of 11. Click OK to add the monitor to the interface tab. To move it around, right-click it, choose "select", then drag it where you want (you can resize it as well), then right-click again and choose "unselect".

Choose the Monitor widget, and a blank monitor comes up. Type in "percent-similar-green" for the monitor, set the display name to "green", set the decimal places to 1, and leave the font size at the default setting of 11. Click OK to add the monitor to the interface tab.

Do similarly for "percent-unhappy-red" and "percent-unhappy-green". You should end up with four new monitors looking something like this (you may need to adjust your model to place the widgets where you want them – your layout may differ from this layout).



Just to make sure we're all on the same page, here is the complete code for the updated Schelling Segregation model.

```
; Segregation-updated.nlogo
;
; Description:
; extensions to the Schelling segregation model
; to investigate variable preferences
; for similarity among different agent classes.
;
; Credits:
; This model is based on the Segregation model
; from the NetLogo library. Original code is
; Copyright 1997 Uri Wilensky. All rights reserved.
; The full copyright notice is in the Information tab.
; -------------------------------------------------------

globals [
  percent-similar  ;; on the average, what percent of a turtle's neighbors
                   ;; are the same color as that turtle?
  percent-unhappy  ;; what percent of the turtles are unhappy?

  percent-similar-red
  percent-unhappy-red
```

```
  percent-similar-green
  percent-unhappy-green


]

turtles-own [
  happy?       ;; for each turtle,
               ;; indicates whether at least %-similar-wanted percent of
               ;; that turtles' neighbors are the same color as the turtle

  similar-nearby  ;; how many neighboring patches
                  ;; have a turtle with my color?

  other-nearby  ;; how many have a turtle of another color?
  total-nearby  ;; sum of previous two variables

  similar-nearby-red
  other-nearby-red
  total-nearby-red

  similar-nearby-green
  other-nearby-green
  total-nearby-green
]


to setup
  clear-all
  reset-ticks
  if number > count patches
    [ user-message
      (word "This pond only has room for " count patches " turtles.")
      stop ]

  ;; create turtles on random patches.
  ask n-of number patches
    [ sprout 1
      [ set color red ] ]
  ;; turn half the turtles green
  ask n-of (number / 2) turtles
    [ set color green ]
  update-variables
  do-plots
end

to go
  if all? turtles [happy?] [ stop ]
  move-unhappy-turtles
  update-variables
  tick
  do-plots
end

to move-unhappy-turtles
  ask turtles with [ not happy? ]
```

```
    [ find-new-spot ]
end

to find-new-spot
  rt random-float 360
  fd random-float 10
  if any? other turtles-here
    [ find-new-spot ]          ;; keep going until we find an unoccupied
patch
  move-to patch-here  ;; move to center of patch
end

to update-variables
  update-turtles
  update-globals
end

to update-turtles
  if (model-extensions = false) [
    ask turtles [
      ;; in next two lines, we use "neighbors" to test the eight patches
      ;; surrounding the current patch
      set similar-nearby count (turtles-on neighbors)
      with [color = [color] of myself]
      set other-nearby count (turtles-on neighbors)
      with [color != [color] of myself]
      set total-nearby similar-nearby + other-nearby
      set happy? similar-nearby >= ( %-similar-wanted * total-nearby / 100 )
    ]
  ]
  if (model-extensions = true) [
    ask turtles with [color = red] [
      set similar-nearby-red count (turtles-on neighbors)
      with [color = [color] of myself]
      set other-nearby-red count (turtles-on neighbors)
      with [color != [color] of myself]
      set total-nearby-red similar-nearby-red + other-nearby-red
      set happy? similar-nearby-red >=
            ( %-similar-wanted-red * total-nearby-red / 100 )
    ]
    ask turtles with [color = green] [
      set similar-nearby-green count (turtles-on neighbors)
      with [color = [color] of myself]
      set other-nearby-green count (turtles-on neighbors)
      with [color != [color] of myself]
      set total-nearby-green similar-nearby-green + other-nearby-green
      set happy? similar-nearby-green >=
            ( %-similar-wanted-green * total-nearby-green / 100 )
    ]
  ]
end

to update-globals
  if (model-extensions = false) [
    let similar-neighbors sum [similar-nearby] of turtles
    let total-neighbors sum [total-nearby] of turtles
    set percent-similar (similar-neighbors / total-neighbors) * 100
```

```
    set percent-unhappy
      (count turtles with [not happy?]) / (count turtles) * 100
  ]
  if (model-extensions = true) [
    let similar-neighbors-red sum [similar-nearby-red] of turtles
    let total-neighbors-red sum [total-nearby-red] of turtles
    set percent-similar-red
      (similar-neighbors-red / total-neighbors-red) * 100
    set percent-unhappy-red
      (count turtles with [not happy? and (color = red)])
      / (count turtles) * 100

    let similar-neighbors-green sum [similar-nearby-green] of turtles
    let total-neighbors-green sum [total-nearby-green] of turtles
    set percent-similar-green
      (similar-neighbors-green / total-neighbors-green) * 100
    set percent-unhappy-green
      (count turtles with [not happy? and (color = green)])
      / (count turtles) * 100
  ]
end

to do-plots
  if (model-extensions = false) [
    set-current-plot "Percent Similar"
    set-current-plot-pen "percent"
    plot percent-similar
    set-current-plot "Percent Unhappy"
    set-current-plot-pen "percent"
    plot percent-unhappy
  ]
  if (model-extensions = true) [
    set-current-plot "Percent Similar"
    set-current-plot-pen "percent-similar-red"
    plot percent-similar-red
    set-current-plot-pen "percent-similar-green"
    plot percent-similar-green

    set-current-plot "Percent Unhappy"
    set-current-plot-pen "percent-unhappy-red"
    plot percent-unhappy-red
    set-current-plot-pen "percent-unhappy-green"
    plot percent-unhappy-green
  ]
end

; Copyright 1997 Uri Wilensky. All rights reserved.
; The full copyright notice is in the Information tab.
```

Running the Updated Model

Now, we're finally done and ready to run our new model. For starters, let's verify that we haven't broken anything, so we'll set the *model-extensions* to Off, set *%-similar-wanted* to 74%, and press *setup*, then press our new *step* button. The model will advance one time tick, and we can watch updates to the plots and the landscape. Press the *step* button a few more times and

watch the model advance one tick at a time. When you get bored of doing this, press the *go* button. The model should run as we've seen before – lots of churning, but eventually large clusters of red and green neighborhoods emerge. Now, let's set *model-extensions* to On, set *%-similar-wanted-red* to 74%, and *%-similar-wanted-green* to 74%, and press *setup*, then press *go*. If all goes well, we should see very similar results to the standard model we just ran – lots of churn, then fairly large clumps of neighborhoods of red and green agents. So far, so good.

The original focus of our research was to examine what happens when there are different similarity preferences among the red and green agents. To test various combinations of similarity preferences, we simply move the *%-similar-wanted-red* and *%-similar-wanted-green* sliders to different values, hit *setup* and *go*, and watch what happens. Just to get a feel for what this looks like, set *%-similar-wanted-red* to 40%, and set *%-similar-wanted-green* to 70%. Notice that the red and green agents settle down to fairly large segregated clusters, but a small percentage of green agents are constantly flitting about. These agents are unable to find homes that meet their happiness criteria, because there is no movement from either red or green established agents. Try this with various combinations of *%-similar-wanted-red* and *%-similar-wanted-green* values. You can even experiment with changing slider values while the model is running.

So recapping, we've seen how to take an existing model and alter it to study a new problem. Although we went through a lot of discussion on the code changes, hopefully you've seen that it did not take all that much change to create some interesting new functionality. Also, you've seen how easy it is to update the graphical user interface to a model by adding new widgets to support adding new inputs or monitoring new outputs.

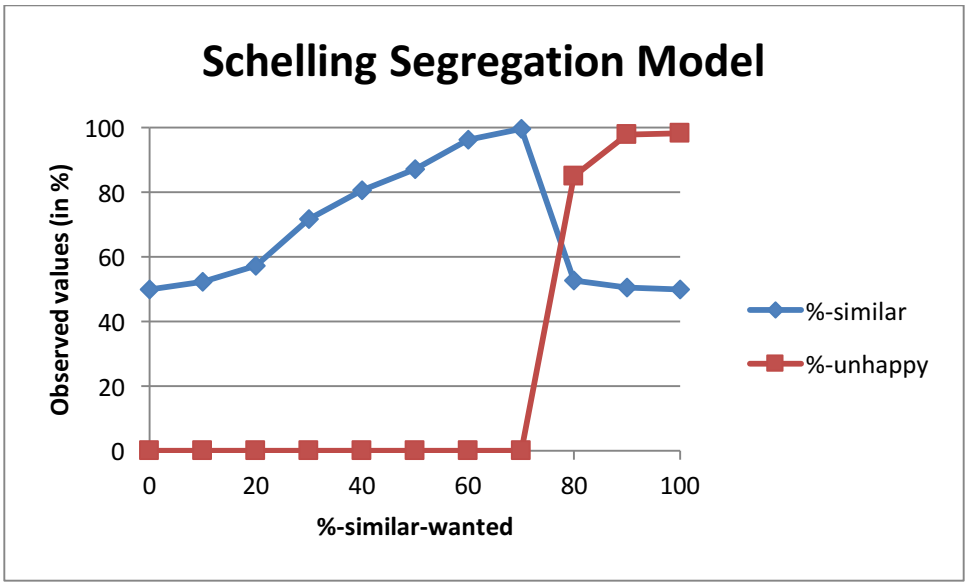
Experimenting with the Updated Model

Although running the model with different parameter settings is interesting, and it shows some visually appealing results, that's not enough to show real results. In order to make best use of an agent-based model, we need to apply a systematic approach to assessing the performance. One way that jumps out quickly is to run the model with different parameter settings, then record key output values and see if there is any discernible relationship between these quantities. As an example, we could build a table in a word processor or spreadsheet, record the parameter settings, run the model, then record selected results of interest. Here's an example showing how we could set up to evaluate the model (without extensions turned on just to keep things simple), looking specifically at the relationship between the user-supplied value for *%-similar-wanted* and the observed *%-similar* and *%-unhappy*. For each setting, we run the model once and record the values.

User supplied	**Observed**	**Observed**
%-similar-wanted	**%-similar**	**%-unhappy**

User supplied %-similar-wanted	Observed %-similar	Observed %-unhappy
0	49.9	0.0
10	52.4	0.0
20	57.3	0.0
30	71.8	0.0
40	80.7	0.0
50	87.2	0.0
60	96.3	0.0
70	99.6	0.0
80	52.7	84.9
90	50.5	97.9
100	49.9	98.3

We could then plot these results on a graph to visualize the results. Based on a quick inspection, it looks like the overall population remains happy even as *the %-similar-wanted* value increases, but only up to a critical point somewhere between 70% and 80%. Values above 80% *for %-similar-wanted* result in very high levels of unhappy agents and roughly evenly distributed neighborhoods.

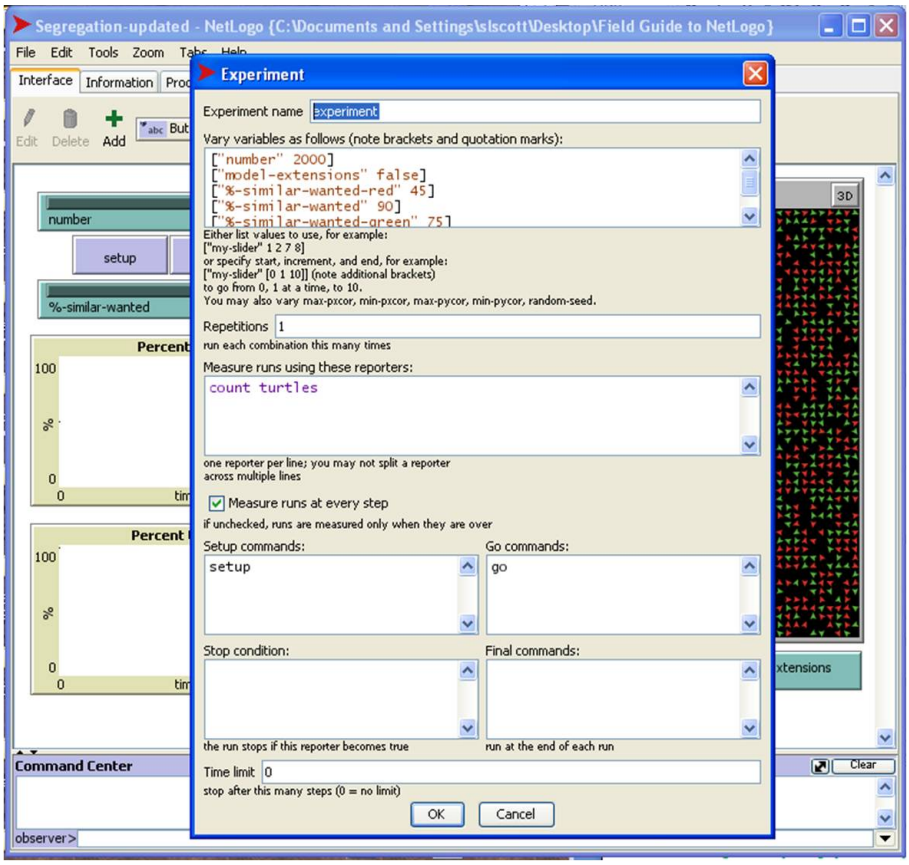


This is interesting, but since our agent-based model does make use of a lot of randomness, we have to ask the question: do these results truly reflect the characteristics of the system, or are they the result of random fluctuation? One way to test this is to run the model repeatedly using particular parameter settings, collect results, average the results obtained at each parameter setting, and then do further statistical analysis.

We could build a very large table to tabulate results, spend many happy hours configuring our model, running the model with various parameter settings, and earnestly copying down the results from each run. Fortunately, NetLogo includes an automated test management tool called BehaviorSpace that will take care of all this for us. BehaviorSpace will automatically change variables according to our specifications, and then run the model for us – and best of all, BehaviorSpace will produce a spreadsheet with the results so we can review and process the results. Let's take a quick tour of this tool just to get a feel for how it works.

Managing Multiple Experiments Using BehaviorSpace

From the main menu, choose Tools -> BehaviorSpace. A small dialog box will come up (it's probably empty unless you've already built some previous experiments). Press New to create a new experiment, and a dialog box like the following should come up.



For Experiment name, we'll just use the default name of "*experiment*" for now, but you can specify any name that you like for your experiments. The next box contains specifications for how our variables should change over the course of the experiment. This can be a little bit intimidating, but thankfully the notation is explained just below the box. Variables are enclosed in brackets and identified using double-quoted strings. Variable values are either explicitly specified as a single value, or specified as a range of values. Ranges of values are themselves enclosed in brackets, with the values indicating the start, increment, and end values. Here's a couple of examples.

["number" 2000]	this will run our model with *number* set to 2000
["model-extensions" false]	our model is run with *model-extensions* set to *false*

Now what if we want to run the model for various values of *%-similar-wanted*, say ranging from 0% to 100% in increments of 10%? No problem – we just adjust the specification for *%-similar-wanted* to indicate the desired start, increment, and end values. Here's an example (note the use of embedded brackets for the range specification).

["%-similar-wanted" [0 10 100]]	vary %-similar-wanted from 0% to 100% by 10%

NetLogo can run each experiment one or more times. You might want to run an experiment multiple times to collect a set of results for each input parameter setting, so let's specify a value here to indicate that. Replace the default value of 1 with 5, so we'll run each setting five times. You may want to run more tests to achieve greater statistical accuracy, but we'll limit ourselves to five runs for now. Be aware that the total number of runs can increase rapidly as you increase this value.

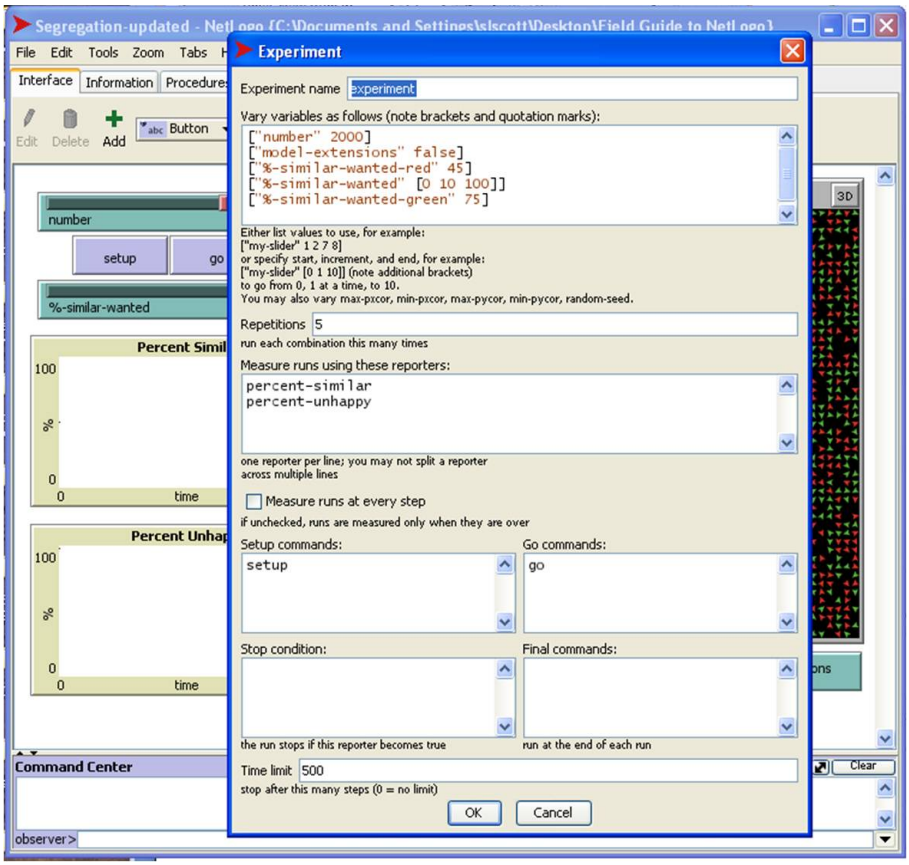
The next thing is very important: **UNCHECK** the box titled Measure runs at every step. If you check this box, NetLogo will report the state of your model at every time tick – which may be OK, but in most cases is not what you want to do and will end up with enormous output files. By unchecking the box, NetLogo only reports the final values of the state of the model when each run finishes, which is far less data and usually all you need for analysis.

Next we need to specify what data we want NetLogo to capture for us at the end of each run. Remove the default expression "count turtles", and replace it with the following two lines. This will indicate to NetLogo that we want to record the value of *%-similar* and the value of *%-unhappy* at the end of each model run.

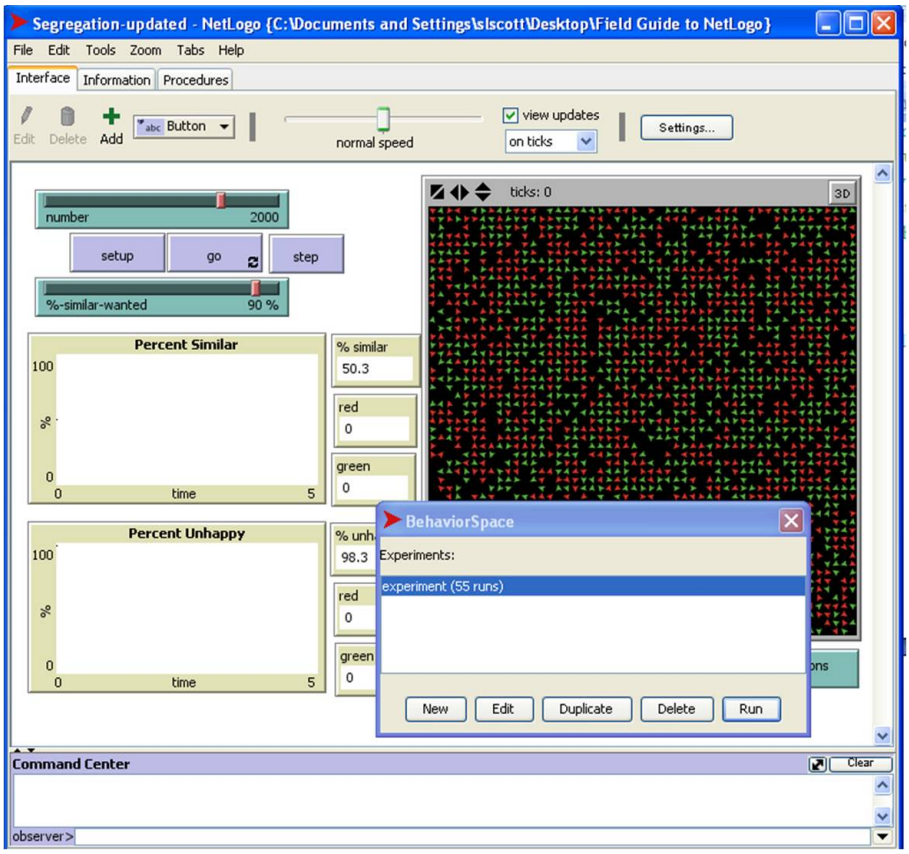
```
percent-similar
percent-unhappy
```

The next set of boxes manages setup, go, stop conditions, and any final commands we want to execute at the end of each model run. The default setup box says we want to execute the setup procedure, which is fine, so leave it as is. Similarly, the default go box says we want to execute the go procedure, which is also fine, so leave it as is.

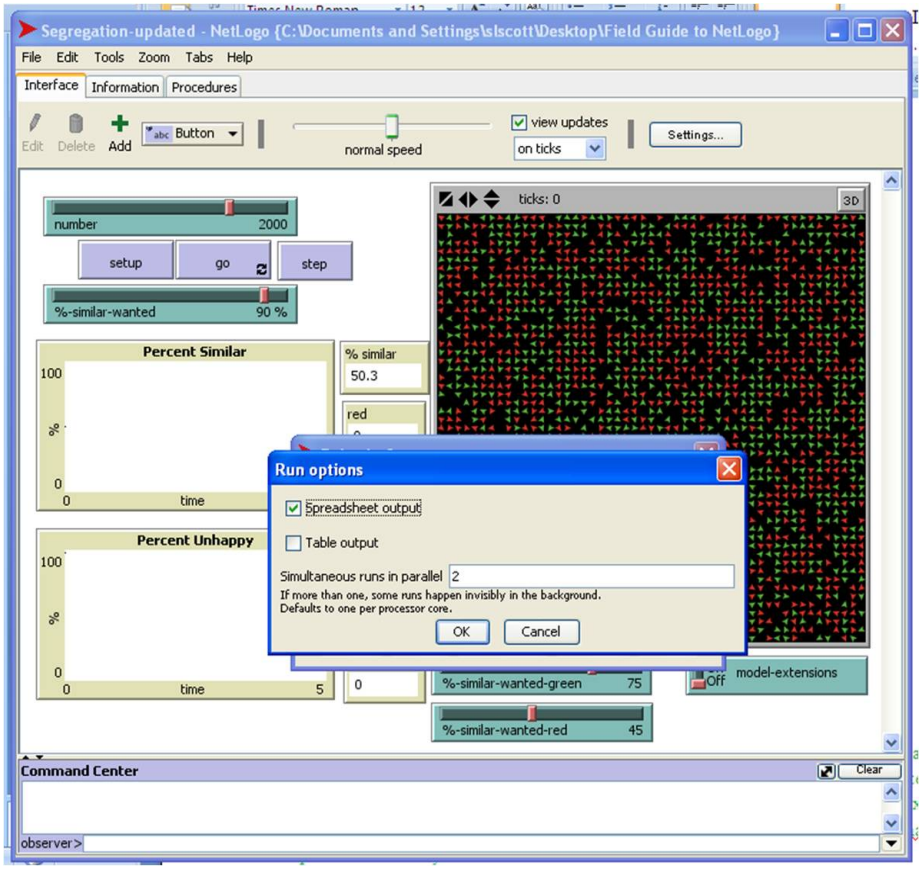
The last thing we need to do, which is very important, is to specify a time limit for the model. As we know from watching our model, under certain conditions the system will never stop, so we need to indicate a maximum number of time cycles as an emergency stop to prevent our model from running forever. Replace the default value of 0 with 500. The final setup for our BehaviorSpace experiment should look like this.



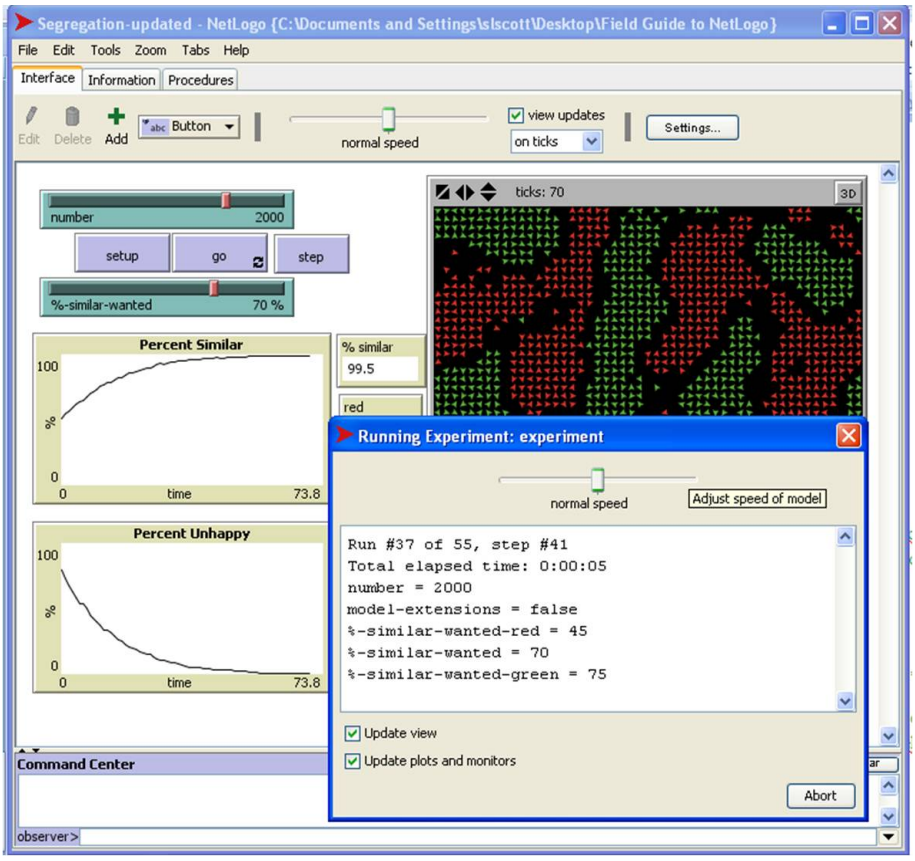
Click OK, and we should now see the original BehaviorSpace dialog box, but now our new experiment (named *experiment*) appears in the box. Also note that NetLogo has calculated the total number of runs for this set of variables and repetitions: right now, we've got 55 experiments being run. Wow – that seems like a lot! We got this by running tests with *%-similar-wanted* ranging from 0, 10, 20, …, 90, 100 % (which means 11 different settings), and for each of these we repeated the test 5 times. So 11*5 = 55, giving us 55 tests. Now hopefully you can see why BehaviorSpace is so valuable – instead of running 55 tests manually (and possibly miscopying results or forgetting something or just getting really tired of running tests), the system will automatically run this suite of tests and produce nicely formatted spreadsheet output for further analysis.



To run the experiment, press Run. Click on the Spreadsheet output option, and if you have multiple CPUs on your machine, specify how many jobs you'd like to run in parallel. Normally, NetLogo uses all available CPUs to speed up the experiment runs. Click OK.

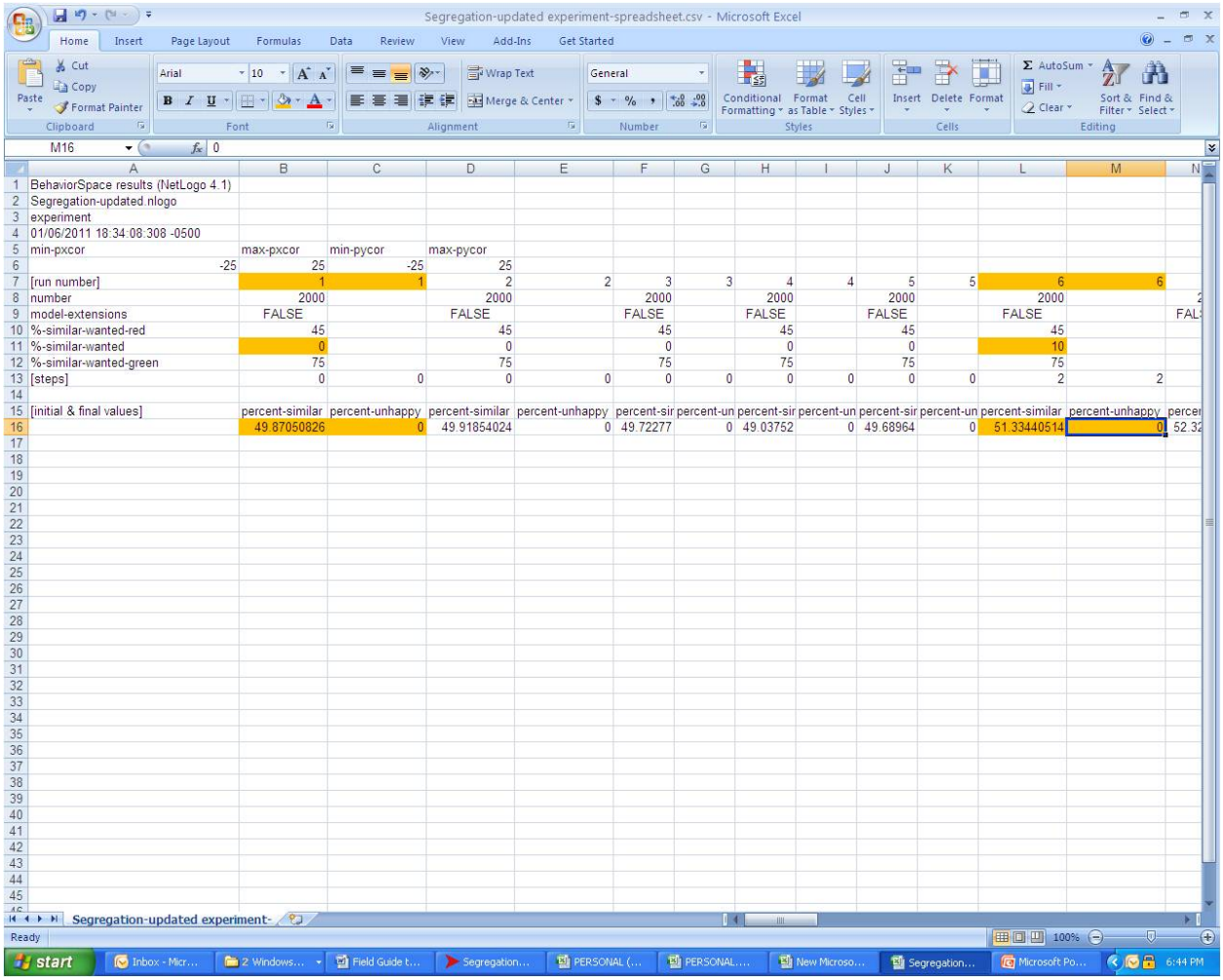


Now specify the name of the output spreadsheet file you would like to use. NetLogo supplies a default name, but you can adjust the filename to something else if you'd like. Click OK, and NetLogo will run the experiment. While it's running, you can see the progress in the dialog box. To make things run faster, you can uncheck the "update view" and "update plots and monitors" checkboxes.



When it's all done running the experiment, the main BehaviorSpace dialog box comes back. At this point, exit by clicking on the red X in the upper right corner of the BehaviorSpace dialog box.

The spreadsheet we created contains the results of our experiment. For each of the parameter settings for %-similar-wanted, we ran 5 tests, and we collected the *%-similar* and *%-unhappy* values. All this data is presented in a spreadsheet so we can review, analyze, or plot the results. Here's an example of what the results look like. In the example below, several cells have been highlighted for emphasis – we'll talk about the shortly.



On the left part of the spreadsheet, the basic experiment metatdata including the name of the model and the date and time are recorded in cells A1 thru A4. Data is collected with one column for each variable of interest, so we'll have two columns per run number with data since we asked for two variables to be recorded at each run. For run number 1, located in columns B7 and B8 on the left of the screen, notice that *%-similar-wanted* is set to 0 in cell B11. Other variable names and their settings for run 1 are found in A8 thru B13. A blank line (line 14) separates variable specifications from results. For run 1, we have the final values for *percent-similar* and *percent-unhappy* in cells B16 and C16, respectively. The results for the remaining runs with *%-similar-wanted* set to 0 are found in D16 thru K16. Starting in column L we see run 6 with *%-similar-wanted* set to 10. The corresponding results for *percent-similar* and *percent-unhappy* for run 6 are indicated in L16 and M16, respectively. You can scroll to the right to see the results for the rest of the runs.

Hopefully this gives you an idea of how to interpret the BehaviorSpace spreadsheet results: all the data from the experiment is there, and you can cut and paste or rearrange things however you'd like in order to do calculations, analysis, or make plots and graphs.

5. So, Your Model Doesn't Work...

First of all, don't panic! Agent models are very complex and there are many reasons they may not work as expected. This chapter will walk through some of the more common problems that agent models can have and some ways to understand why a model may be misbehaving.

Introduction

Before we begin the discussion about specific issues with debugging agent models, there are some rules of thumb that can help minimize the potential problems associated with building agent models and debugging them later.

Always start with a formulation. A formulation is a narrative description of the agent model you are building. This will help to ensure that you have thought through the basics of the model. The more detail you can add to this narrative, the better. This includes adding as many algorithms and equations as you can. Equally useful are UML diagrams, mind maps, or even simple flow charts. This will help you think through how information will move around the model and what agents will need to keep track of. For example, a very basic formulation of the Schelling segregation model might look something like this:

'Households will move about a landscape until they find a neighborhood that they are satisfied with. Movement will be to a random location not currently occupied. Neighborhood will be defined as the household's Moore neighbors. "Satisfied" is a parameter set at runtime that is a threshold compared to the likeness of the household's neighborhood based upon homogeneity.'

This is a very simple narrative of the model that covers most of the major points. However, it is incomplete—how is "likeness" calculated? Is "satisfaction" homogeneous or heterogeneous? How will agents be activated? and so on. It can be helpful to create a simple flowchart of the model:

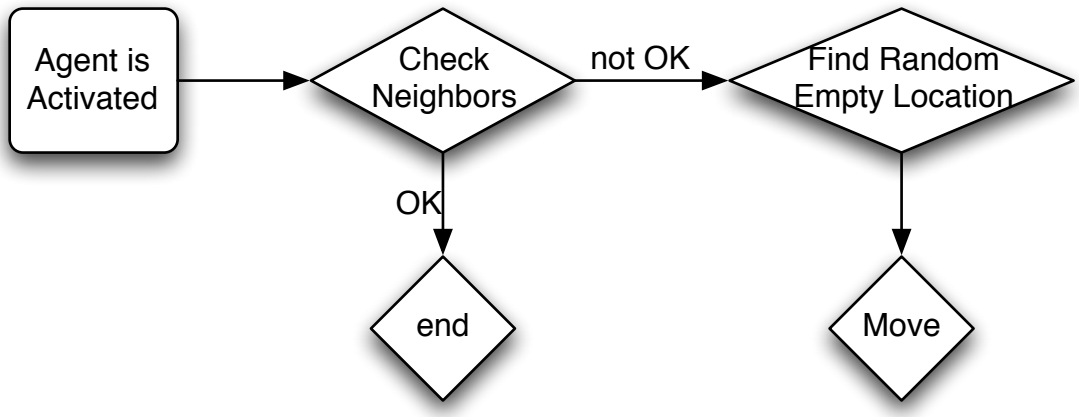


Figure 5.1 A simple flowchart of the Schelling segregation model

A formulation will help to highlight model functioning and aspects of the model that still lack detail. Any "amount" of formulation is helpful, but the more detailed the formulation, the easier the creation of the executable simulation will be. Even with the most detailed formulation, however, you are bound to run into some problems as your model executes. Now we will turn to some of these problems and ways to expose them.

In simple terms, there are three general types of problems that you may run into while building an ABM. Unfortunately, these types are not mutually exclusive, can pretend to be each other, and can actually "cancel" each other out. This makes debugging ABMs quite challenging. The types of problems are: Code, Logic, and Activation. Code problems are simply issues where you've implemented your model (the formulation) incorrectly. This can be done with typos (using a ">" when you meant to use "<"), or by calling the wrong function or using the wrong command (using *neighbors* when you meant *neighbors4*).

Code problems are the most straightforward. This isn't to say that they are easy to find and fix, but they are usually driven by typos, such as using a greater-than symbol instead of a less-than symbol. For example:

```
If random-float 1 > my-probability [do-something]
```

when you mean:

```
If random-float 1 < my-probability [do-something]
```

The issue here is that if you want the likelihood of do-something to increase as my-probability increases, then the greater-than symbol in the first case will cause the opposite to happen at runtime. In the first case, as my-probability increases, the likelihood of do-something occurring will decrease. This example also shows how a problem can fall into more than one type. If your formulation included a less-than symbol, then this is a code type problem. If, however, your formulation included a greater-than symbol, then this becomes a logic problem.

Logic problems spring up when what you have specified in your formulation does not produce the desired dynamics/outcome. These problems can be tougher to figure out. For example, suppose you want to create a simple model of follow the leader. A formulation for such a model might be as follows:

'Turtles will pick someone at random to follow and then move towards that person.'

The dynamic you are looking for is a long chain of turtles moving around the screen. Will that happen with the above formulation? A literal implementation of this formulation creates a system where agents slowly tend to cluster but in no way is it a long chain of agents. This is because during each time step the turtles pick a turtle to follow, but not necessarily the *same* turtle. By potentially picking a new turtle to follow each time step no coherent structure forms. What happens if they pick a single person at random and only follow that person? Then you end up with a few turtle chains moving about the environment and occasionally a ball of turtles (ones

that picked each other to follow). Finally, if you have all turtles pick the turtle with a who-number that is one less than theirs (this way everyone has a unique leader) and ask turtle 0 to move without regard to a leader (as there is no turtle -1), you end up with a long chain of turtles that is actually very enjoyable to watch. Code for this third case is shown below:

```
turtles-own [my-buddy]

to setup
  ca
  crt 3500 [turtle-setup]
end

to turtle-setup
set my-buddy nobody
if who > 0 [set my-buddy turtle (who - 1)]
jump random 1000
end

to go
ask turtles [move]
end

to move
ifelse my-buddy = nobody
  [set heading heading + random-normal 0 10 fd 1]
  [
    if distance mu-buddy > 1
      [set heading towards my-buddy fd 1]
  ]
end
```

This very simple example points out there are many ways to create code that functions but does not actually do what you intended. Logic problems can be some of the toughest to track down.

The last general type of problem that we will explicitly discuss here is problems with agent activation regime. Activation regime refers to how and when agents (turtles and patches) are told to do things. This usually boils down to when agents update their values. Buffered activation refers to a regime in which agents wait to share new values until after all agents complete their activities for that time step. There are also different ways to activate agents within a single time step: 1) everyone every time step in the *same* order, 2) everyone every time step in a *different* order, or 3) statistically (here agents have a probability of being activated so some agents may get many more "turns" than other agents but overall agents will get x turns per y time steps). Examples follow:

Unbuffered activation:

```
Ask turtles [gather-info do-calculations update-values]
```

Buffered activation:

```
ask turtles [gather-info]
ask turtles [do-calculations]
ask turtles [update-values]
```

By splitting the procedures out, agents will not move to the next one until everyone is done with the current one.

Activation in the *same* order each time:

```
set my-turtles sort turtles

foreach my-turtles [ask ? [do-stuff]]
```

Activation in a *different* order each time (this is the NetLogo default):

```
ask turtles [do-stuff]
```

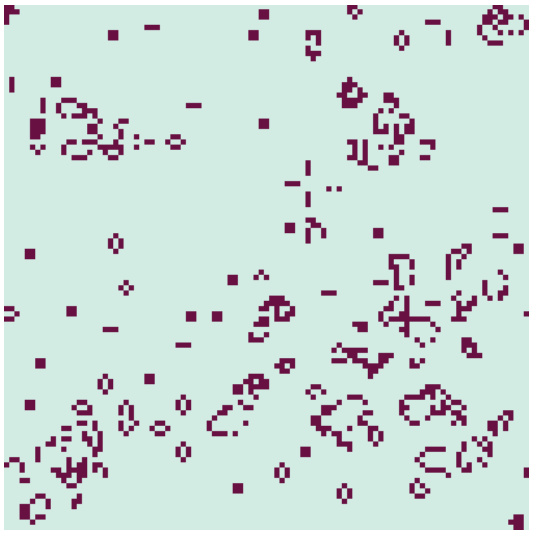
Statistical activation (here turtles have a 10% chance of being activated):

```
ask n-of (count turtles / 10) turtles [do-stuff]
```

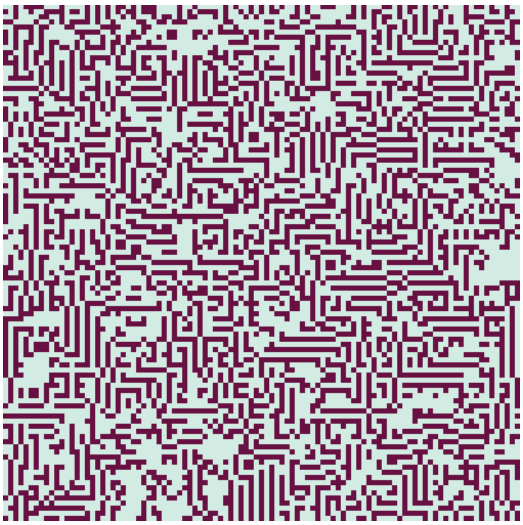
Another, potentially less efficient way to do this (one should note that while statistically these methods are equivalent, in the latter all turtles have a 10% chance of doing something; therefore, each time this is run it is possible that more or less than 10% of turtles will actually do something):

```
ask turtles [if random 100 < 10 [do-stuff]]
```

A dramatic example of the importance of activation regime can be found in the Game of Life. Figure 5-2 shows how different the dynamics are when agent activation is buffered or not. The classic Game of Life uses buffered activation.



a. classic Game of Life with buffered activation



b. the Game of Life with unbuffered agent activation

Figure 5-2. The Game of Life with buffered and unbuffered activation after about 150 time steps

Random Numbers

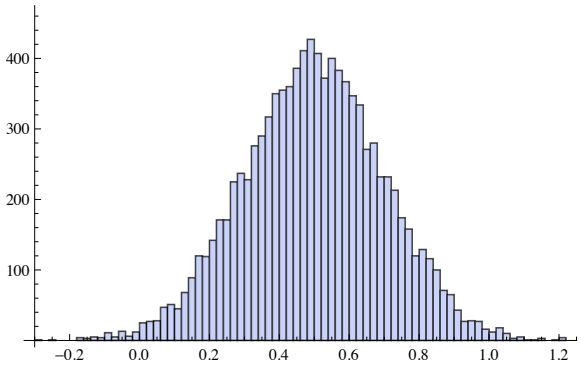
Another source of potential problems in agent models are the way random numbers are generated and used. Random numbers are often a vital piece of an agent model, but as the above simple example (< vs. >) shows, they can also be a source of problems. As you develop your model, be sure to be conscious of how you use random numbers and when. NetLogo, like all computer programs uses a pseudo-random number generator. The numbers are not truly random; they are created in a deterministic way. This can be very advantageous to the agent modeler. If you "seed" the random number generator with the same number (or starting point) then you get the same stream of pseudo-random numbers out. As you build up your model, you should try the same random seed as well as others. Using the same random number stream (from the same seed) should produce the same dynamics, meaning your model is reproducible (which is a key component to any scientific endeavor). This can also help with debugging as it allows you to control changes in model dynamics that are due to the stochastic nature of your simulation. To set the random seed in NetLogo, simply place this line of code in the beginning of your setup procedures (after you have called clear-all): *random-seed* 55. The number 55 is simply an example here, any number can be used. Keep in mind that if you have changed the way agents access the random number generator, they may get different random numbers than they did before, despite the fact that you have set NetLogo's random seed. For example: Say your fixed random number stream is: [2, 5, 3, 3, 7, 9, 1, 4, 2, 6], and your model has two agents who each need a random number. You have fixed the random seed so the activation order of the agents will be the same. So, agent 0 asks are a random number and receives the number 2; then agent 1 asks and receives number 5. This will happen every time as you have fixed the random seed. Now you make a change to what the agents do that necessitates that the agents ask for two

random numbers. Even with the fixed random seed, agent 1 will now receive different numbers. With the fixed seed, the random stream is as before, as is agent activation order, but now when agent 0 asks for two random numbers, it receives numbers 2 and 5. Now when agent 1 asks it will not receive the number 5 as before; it will receive the numbers 3 and 3. This will cause execution differences despite the fixed seed. Code for this example is shown below (to have the turtles ask for two numbers, simply uncomment the second "show random 10"):

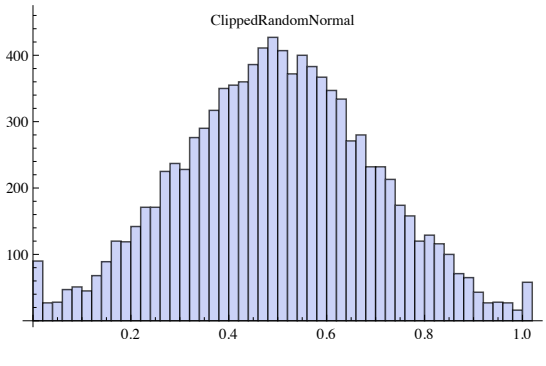
```
to setup
clear-all
random-seed 55
create-turtles 2
end

to go
ask turtles [do-something]
end

to do-something
show random 10
;;Show random 10
end
```

This can be complicated by subtle use of the random number generator. For example, when turtles are created, they are given a random color and a random heading. When you ask n-of turtles to do something and so on, all these commands use the random number generator, which complicates determining exactly which number any given agent will receive from a fixed random number stream.

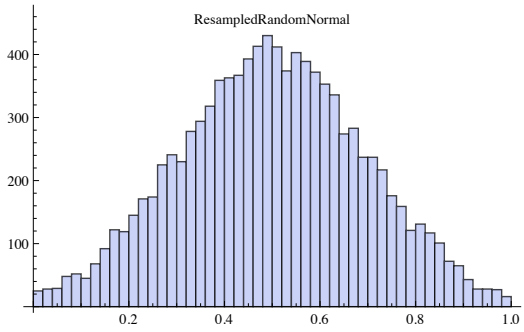
Another source of potential problems with random numbers can occur when you build up distributions of random numbers. For example, suppose you want your population of agents to have a normally distributed probability of doing some action. As this is a probability, it will need to be bounded by 0 and 1. If we assume that you want the normal distribution to be centered on 0.5 with a standard deviation of 0.2, this will produce a population of agents that will mostly have a 0.3 to 0.7 chance of performing the action. However, the normal distribution does not have hard bounds, so although about 98% of agents will have a probability between 0.1 and 0.9, some will have probabilities less than zero or greater than 1. Using NetLogo, Figure 5-3a shows the distribution of values generated from 10,000 draws using the command: (`random-normal 0.5 0.2).` As can be seen in Figure 5.3, the values drawn from the aforementioned distribution stretch beyond 0 and 1.



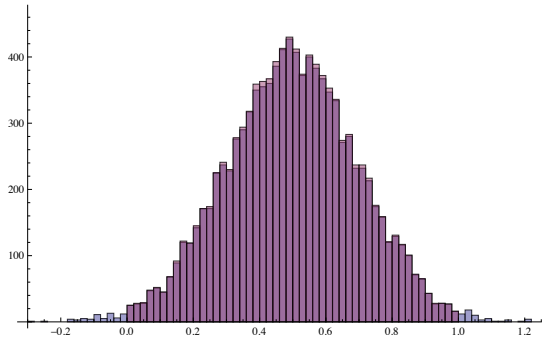
a) 10,000 draws using NetLogo's random normal command with a mean of 0.5 and a standard deviation of 0.2.



b) 10,000 draws using NetLogo's random normal command with a mean of 0.5 and a standard deviation of 0.2, where values greater than 1 were set to 1 and values less than 0 were set to 0.



c) 10,000 draws using NetLogo's random normal command with a mean of 0.5 and a standard deviation of 0.2, where values greater than 1 or less than 0 were resampled until they fell between 0 and 1.



d) Two distributions each of 10,000 draws from NetLogo's random normal distribution with a mean of 0.5 and a standard deviation of 0.2; the purple distribution has values greater than 1 and less than 0 resampled. As can be seen the blue "tails" on either side of the distribution are what must be resampled. This causes greater weight in the "shoulders" of the normal distribution and means your normally distributed population will not be truly normally distributed.

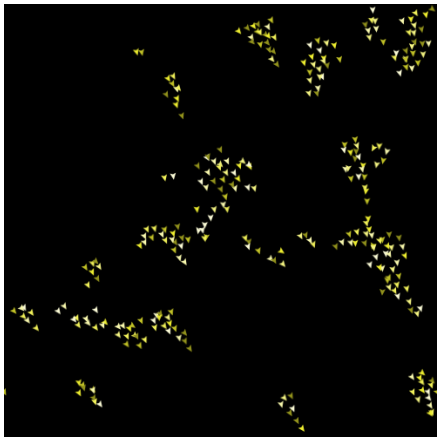
Figure 5-3. Random normal distributions from NetLogo
`( random-normal 0.5 0.2 )`

Obviously, having values less than 0 and greater than 1 will cause trouble when being used as a probability. There are a few ways to deal with this issue. One way is to simply check the value of the random number when it is generated and if it is greater than 1, set it to one, and if it is less than 0, set it to 0. This method, however, will create a trimodal distribution as can be seen in Figure 5-3b. This means that you will have a population of agents that have, potentially, an

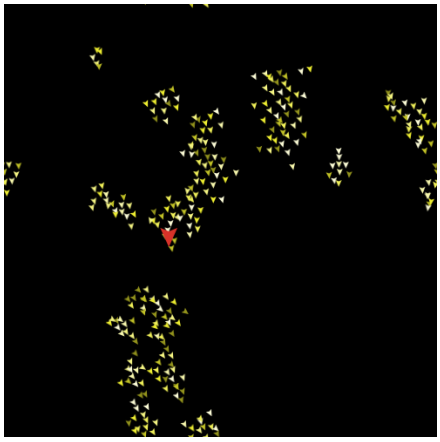
overabundance of agents that will never perform the activity and agents that will always perform the activity. Code that implements this check would look something like this:

```
set my-probability random-normal .5 .2
        if my-probability > 1 [set my-probability 1]
        if my-probability < 0 [set my-probability 0]
```

This might be fine for the model that you are creating, but it also might skew the results depending upon the path dependencies that are in it. One way to think about the potential impact of having "too" many agents that always do one thing is to play with NetLogo's flocking model. There is randomness throughout that model and you end up with many changing flocks of boids "flying" around the space in basically the same direction. If you add a "spoiler" boid you can get the flock to fly in any direction you wish. A spoiler boid is simply one boid that flies in the same direction all the time and does not do anything else. A typical flock is shown in Figure 5-4a, where most boids are heading in the same direction. This direction will change over time as the agents make *random* adjustments to each other.



a) A typical run of the NetLogo flocking model.



b) A typical run of the NetLogo flocking model with a spoiler included (here shown in red).

Figure 5-4. Examples of the flocking model from NetLogo

Even with the inclusion of only one spoiler, you can bias the flying dynamics and create any outcome you want with respect to flying direction. Figure 5-4b shows the same flocking model but with the addition of a spoiler. As can be seen, the flock is flying down. This run was started with the spoiler flying to the right, then once the boids were all flying to the right, the spoiler direction was changed to down. It did take about 30,000 time steps to change the boids' direction by 90 degrees, but it should be noted that the spoiler could only influence the boids that are within vision of it. This means that the spoiler has very little direct influence over the system. But even tiny amounts of influence applied consistently can cause great bias in the results. This example is not meant to suggest that simply resetting out-of-bound values to be at the bounds is necessarily a bad idea. It is meant to highlight the need to think through the

mechanisms and that you will likely need to test ideas out and see what, if any, impact your design decisions have. It is also worth noting that the outliers often drive these systems. Therefore, you want the system to have the "right" outliers, not outliers created simply by statistical happenstance.

There are other ways of dealing with a random distribution that may generate out-of-bound values. Returning to the normal distribution discussed above `(random-normal 0.5 0.2)`, you could resample when values that are out-of-bounds come up. In this case, if a value of 1.1 was generated, you would throw it away and continue to generate new values until a value between 0 and 1 was found. Code for this check would look something like this:

```
set my-probability random-normal .5 .2
        while [my-probability > 1 or my-probability < 0]
        [set my-probability random-normal .5 .2]
```

As this is a while loop, this is a slower way to deal with the problem than simply clipping values to 0 or 1, but it doesn't have the problem of creating a trimodal distribution (see Figure 5-3c). However, it will increase the mass of the middle of the distribution (see Figure 5-3d). This change to the mass of the middle of the distribution is not extreme, but it is there, so the impact this may have on your model should be thought through.

What all this really boils down to is truly understanding your model (what you are trying to do) and how you turned that into a simulation (the executable agent-based model that you created in NetLogo). The first step to a functioning model is a clearly articulated formulation, as discussed above. Once you move that formulation into code, things can get a bit trickier to understand. NetLogo has many features that can help you to understand what your model is doing. There is no perfectly right or wrong way to build and debug an agent model. The way you approach it will need to be the way that makes the most sense to you. The rest of this chapter will be a discussion of various options available for understanding your model, how it is working, and if it adequately represents your formulation.

How you gain an understanding of the model will be a balancing act between tractability and information loss. Except in the smallest cases, it will be nearly impossible to keep track of the complete state of the model from one time step to the next to gain a "perfect" understanding of the model. Instead, you will need to try and get the "gist" of what the model is up to and then focus in on the parts that seem to be causing trouble.

A Few Ways to Understand Your Model

Visualization

One of the most powerful ways of understanding an agent model is visualization: watching the agents do stuff. As you know, you can set agent color and size in NetLogo. Agent color does not need to be static. Agent color can be a function of the value of a parameter. The *scale-color* command is the most convenient way to do this. Going back to the Schelling segregation model,

the agents could scale their redness or greenness by the ratio of "like" to total neighbors. First we need to add a variable to keep track of whether the agent is red or green.

```
turtles-own [… my-color]
```

Then we need to add a line at the end of setup to ask the turtles to set `my-color`

```
ask turtles [set my-color color]
```

Now we are ready to use the neighborhood ratio to scale a turtle's color. This will help us see where turtles are happy and unhappy. To scale the turtle's color, use a command such as:

```
set color scale-color my-color (similar-nearby / total-nearby) 0
1
```

For this to work correctly, you should call it after the agent has calculated similar-nearby and total-nearby. If you call this before that, then it will be using the previous time step's values. With this setup the turtles' color will be lighter the happier they are. Figure 5-5 shows this version of the Segregation model. This visualization makes clear that agents inside large homogeneous blocks of similar agents tend to be happier than do agents near the edges. This is obvious in hindsight but the visualization makes it clear. (Note: to implement this in the NetLogo Segregation model you need to change all references to color to myColor so that turtles compare each other to a discrete value (red vs. green) and not a floating value, which is what we changed turtle color to be.) Of course, the same thing can be done with patches.

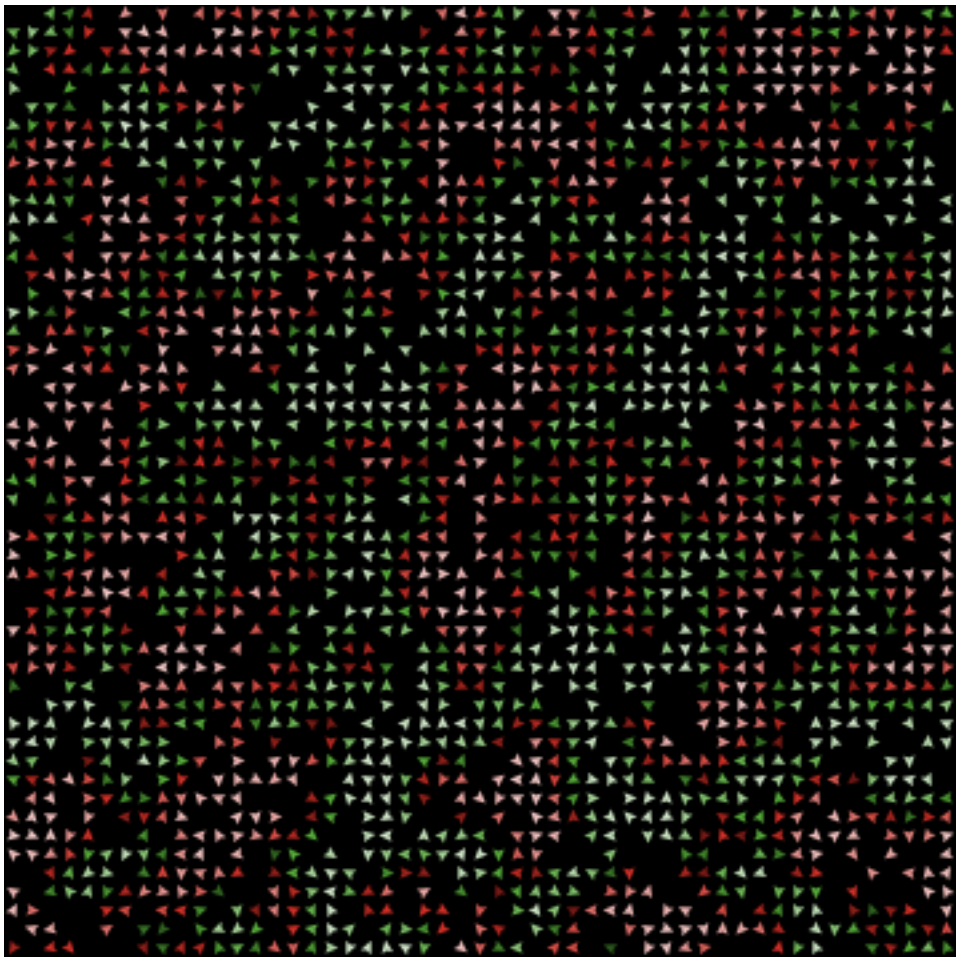


Figure 5-5. The scaled-color NetLogo Segregation model

Plotting

The plotting capabilities of NetLogo are fantastic. There are many ways to create plots, each highlighting a different window into your model. It is a very good idea to create plots as you build in new or important functionality into your model so you can understand how it is working. Moreover, as you create plots, you are automatically creating mechanisms to capture data. Then to produce data from a run, all you need to do is export-plots and NetLogo will create a file with all current settings on the user interface and the data that is displayed in every plot, all well-structured and well-labeled. NetLogo allows you to create scatter plots, line plots, and bar charts. The differences come in pen "mode." Pen modes include line (which will draw a line between points), bar, and point (no line between points). The basics of plotting were covered in an earlier chapter and will not be repeated here. This focus of this section is ways to think about plots that may reveal more of the agent model.

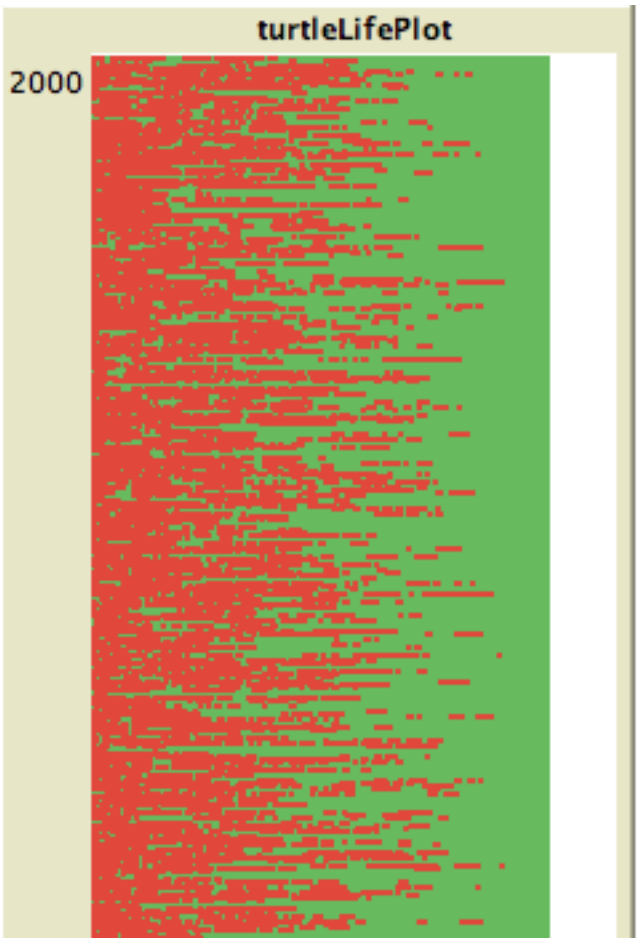
The most obvious type of chart is a time series (the value of something over time) created with a line pen mode. Looking again at the NetLogo Segregation model, we can see that it has two time series plots, one looking at the percent-similar and another that tracks the percent of unhappy turtles. These plots are very good for giving you the gist of what is going on, but they do not tell you what is happening to any given agent. This brings up the tractability vs. information loss tension again. For example, suppose you have 10 agents who all have a single number. Further, suppose that all have the same number, for instance, the number 2. The mean value of the turtles' numbers will be 2 and will be a very good characterization of the population. However, now suppose 5 agents have the number 1 and 5 agents have the number 3. Again, the mean is 2, but now it is a terrible representation of the situation. It is worth keeping this tension in mind as you design your plots.

If you don't have too many agents, you can keep a time series for each one in a plot. Going back to the Segregation model, you could set up a point plot with two pens: one pen for happy agents (for example a green pen), and one for unhappy turtles (for example a red pen). Then you ask each turtle to plot its turtle number by tick with the appropriate pen. This plot can be seen in Figure 5-6. As you can clearly see, the turtles start out unhappy but then become happy. You can also examine the individual "lives" of each agent. Many agents transition between being happy and unhappy while the system as a whole settles down. Given the structure of the model, these are the sorts of dynamics one might expect. Turtles start out randomly distributed and then move randomly if they are unhappy. Therefore, one would not expect to see clear periodic structure in the plot, and here we do not. If such structure was seen in the plot, it could signal a problem with the model's algorithms. Again, it will depend on the model. Looking at the patch "life" plot, you can see distinct structure emerge over time. This is to be expected as the turtles form isolated islands of like colors that translate into happy turtles. When making these plots, also keep in mind that you need to plot the turtles and patches in the same order every time. You will need to make an ordered list from the set of interesting agents (recall how to do that from our earlier discussion).

One thing to keep in mind is that even with the relatively small model we have here, we are looking at a data structure that is, for turtles, 2000 multiplied by the number of time steps, and for the patches it is 2601 multiplied by the number of time steps. That is a big plot, and likely you will lose resolution in its display. Look at the tip of Figure 5-6 again. You can see the whole plot there, but, even as complicated as it looks, there is information that is lost due to size. If we zoom in by about a factor of 2, now we start to see the fine, agent level structures. The loss of detail is particularly obvious in the plot of the patches. The structure associated with happy turtles is much clearer when the plot is zoomed in (the islands of happy turtles show up as stripes in the plot because the 2D space is represented in row major format (this format essentially takes each row and appends it to the end of the row above it so one ends up with one very long row) along the y-axis). When creating these sorts of low-loss visualizations, it is worth thinking about the resolution of the monitor you are using. Even a reasonably high-resolution wide-screen monitor (circa 2010) oriented vertically may only have 1920 pixels from top to bottom. The locationPlot shown below requires at least 2601 pixels to show its full resolution. This does not mean you should not use these types of visualization, only that you should be aware of the hardware you are using and the impact that it may have. These visualizations are shown in the lower portion of Figure 5-6.

a) The whole turtle life plot.

b) The whole location plot.



c) The "zoomed in" turtle life plot.

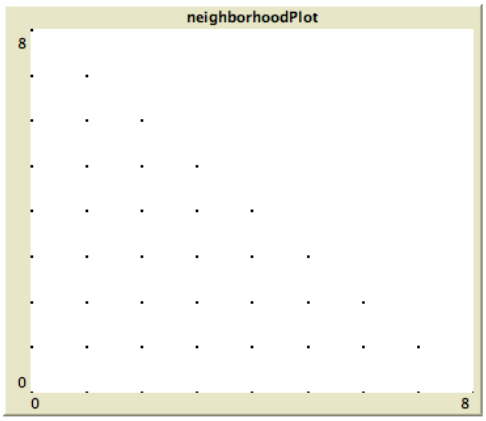
d) The "zoomed in" location plot.

Figure 5-6. The "life history" of turtles and patches in a Segregation model (2000 turtles that desire 75% similarity in their neighborhood and live on a 51x51 patch torus).

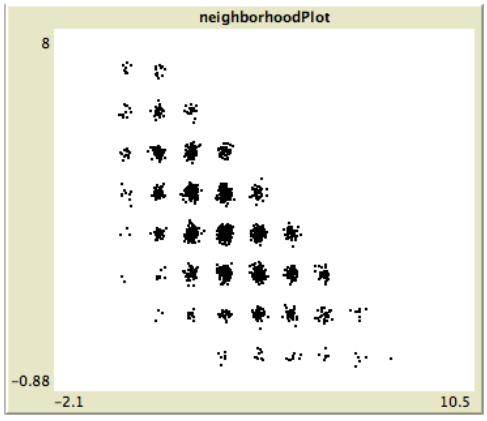
In addition to the resolution issue, asking all of the turtles and all of the patches to do something every time step can significantly degrade the performance of the model. Furthermore, if you have turtles dying and being born over the course of the run, these sorts of plots will likely not work.

You can use also scatter plots to look at the relationship between two parameters. To do this, simply use one parameter as the x-axis coordinate and the other parameter as the y-axis coordinate. Looking at the Segregation model again, we could set up a plot to look at the relationship between number of similar and number of dissimilar turtles in each turtle's neighborhood. An example of this plot can be found in Figure 5-7a. As can be seen, there is a significant problem with the plot. Although we have 2000 agents, there are only 45 points on the plot. The problem is that we have broken the space up into a small number of discrete chunks and so the points are on top of each other and we cannot tell how many turtles are in each spot. One way to get around this is to add some jitter to each point. This is simply a bit of random noise to each value. This will spread out the points and make the density much easier to see.

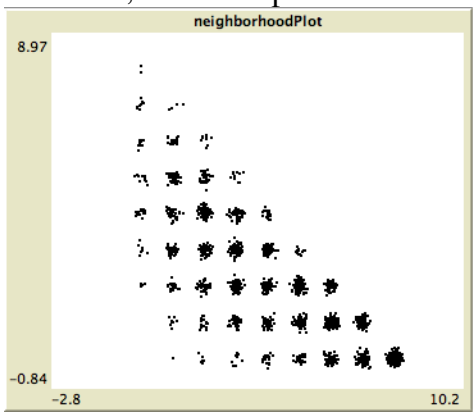
This change is shown in Figure 5-7b. The noise was added via NetLogo's *random-normal* call with a mean of 0 and a standard deviation of 0.1. This noise term was added to both the x and y coordinates. Adding the right amount of noise can be a bit of an art. You do not want to add so much noise that you mix the data together, however you do want to add enough noise to make the density of points clear. With the added noise, it is now clear that the turtles are not uniformly distributed, and most turtles are in the middle. This stands to reason because the turtles were, initially, randomly distributed. There is one other aspect to keep in mind. You may want to *clear-plot* each time step so that you get the current picture. If you do not *clear-plot* each time step then you will end up accumulating all of the points and that will obscure changes over time. In Figure 5-7c, you can see that there is a clear migration of turtles into the lower right portion of the graph. This movement is consistent with the observation that the turtles are creating islands of homogeneity. This structure can be seen in the visualization of turtle activity with the happiest turtles surrounded by other turtles like themselves and in the structure seen in the life history plots.



a) Unjittered plot with obscured data.



b) Jittered plot with clearer agent distributions, at time step 1.



c) Jittered plot with clearer agent distributions after 75 time steps, the change is agent distribution is clear, here the agent values have drifted to the lower right.

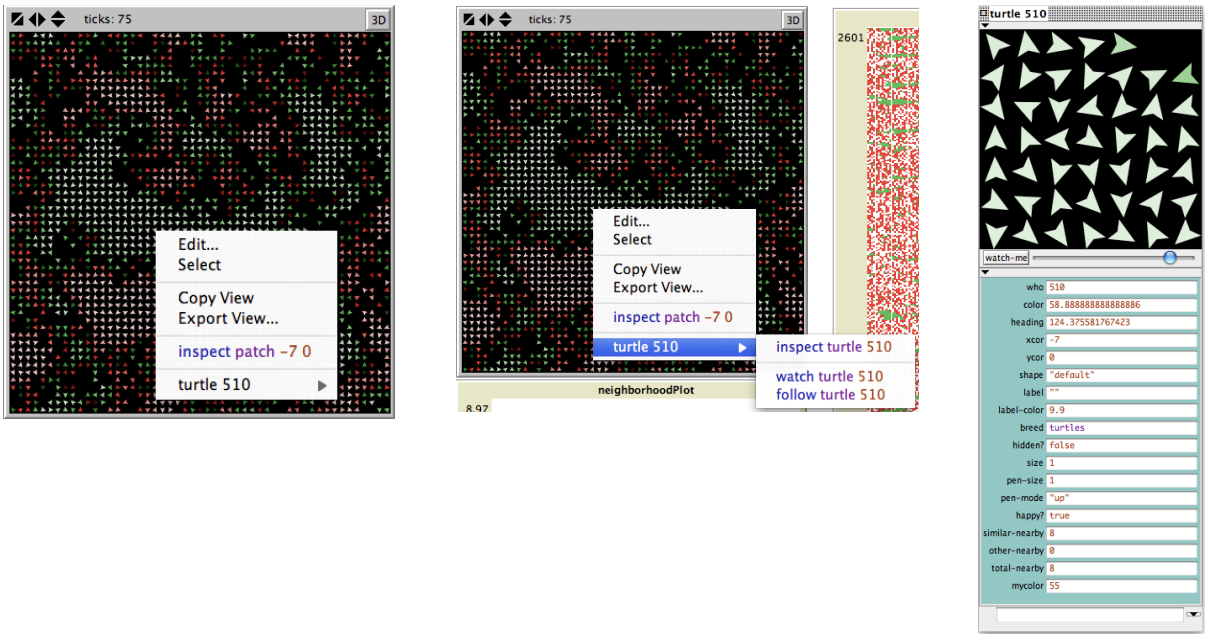
Figure 5-7. Scatter plots and jittered scatter plots

We have now created three distinct ways of understanding the dynamics of the Segregation model. Each view into the dynamics is different and none shows any unexpected structure. This provides evidence that our model is functioning the way we anticipated. Moreover, we have only scratched the surface of how to use plots for debugging and understanding. You can also add reference lines to show significant thresholds on the plots and so on.

Stepping and Print Statements

One of the simplest ways of debugging an agent model is to simply slow things down and print out everything that's going on (locking down the code and the random seed first, of course). NetLogo makes this fairly easy. Assuming the main runtime loop of your model is called *go,* then you can make a forever button on the interface to run your model that simply calls the *go* procedure over and over again. If you make a button that calls go and is **not** a forever button, you will now only advance the model one step at a time each time you press that button (we'll call this button the step button). Advancing the model one step at a time will give you time to review what happened and begin to build up an understanding about how the model behaves and how what happens accumulates as the model runs forward. Using the Segregation model as an example, one could advance that one step at a time and watch the change in turtle location and color, as well as the change to the various plots we created and gain a reasonable understanding of the model. However, all of this is at a fairly aggregated level.

Often, stepping the model will be coupled with following a small number of agents. This is done with the inspection capability within NetLogo. You can inspect a turtle or a patch in two ways. The first way is to right-click on the patch or turtle you want to inspect. This can be hard as the number of turtles and patches increases. The second way to inspect a turtle or patch is to input that in the command center. This way you can inspect a specific turtle (by number) or patch (by coordinates). More importantly, you can also ask to inspect a random turtle or patch that meets one or more characteristics. For example, you could ask to inspect a turtle with total-nearby more than 9 (as this would be a malfunctioning turtle): `inspect one-of turtles with [total-nearby > 9]`. This will create an inspection window for a randomly chosen turtle that meets the stated criteria. The inspection window also has a command line so you can interact directly with that turtle/patch. One might ask that the turtle increase its size so you can find it more easily. You can also ask that the turtle/patch show you the values of particular variables with the `show` command. NOTE: This will only work with declared variables and will **not** work with local variables. Local variables can be "inspected" with *print* or *show* (the advantage of the *show* statement is that it lists the turtle/patch doing the "showing") statements in the procedure where they exist. NOTE ALSO: As soon as a report command is executed, the reporter is exited. This means that if you have a *show*/*print* statement after the report command in a reporter, it will never be executed. The progression of clicking to turtle inspection is shown in Figure 5-8.



a) Initial right click on a turtle brings up this context menu.

b) Choosing the last option reveals the "inspect turtle" option.

c) The inspected turtle view.

Figure 5-8. Inspecting a turtle

Along with inspecting one or a small number of turtles or patches, you can set up your model to include statements that *show*/*print* values to the command center. This allows you to keep track of what is going on in as much detail as you would like. This can get to be a bit overwhelming if you have a lot of turtles and are running the model for a long time. In the Segregation model, for example, we have 2000 agents and it typically takes them about 125 time steps to get happy (when they are looking for 75% similarity). If we asked every turtle to print something out every time step, we would end up with 250,000 lines of output. Furthermore, it is very time consuming to write to the screen, so this will dramatically increase your run time.

Sometimes there is no way around it, you just need to print everything out and then dig around in it to understand what is going on. Often, however, you only need to see when things are going wrong. In these cases, you can combine an if-statement with the *show*/*print* statement. For example:

```
if my-probability < 0 or my-probability > 1 [show my-
probability]
```

Now, we will only see a message from a turtle when its value of `my-probability` is less than 0 or greater than 1, which, obviously, would indicate a significant problem as probabilities should not be greater than 1 or less than 0. Better yet, these problems will not be hidden among, potentially, thousands of lines of *print*/*show* statements from properly functioning agents. You can also add other constraints onto the *print*/*show* statements to have greater control of what is

being written to the screen. This is done by building up additional tests in the if-statement containing the *print*/*show* statement(s). For example:

```
if debug? and my-robability < 0 or my-probability > 1 [show my-
probability]
```

Now you can turn the printing on/off with a switch called "debug?". This can be further developed with other Booleans such as show-turtles? and show-patches?. Adding these switches will let you turn debugging statements off and on generally and also for turtles and patches individually. Finally, you can create "levels" of debugging statements. "Levels" here refers to how deep into the details of the code you wish to go. For example: level 1 debugging might be some simple system-wide aggregated values such as the minimum, maximum, and mean value for my-probability. Level 2 debugging might be the above example where any turtle with a problematic my-probability value *prints*/*shows* it. Finally, level 3 might include turtles printing/showing not only their problematic values for my-probability but also the values that went into the calculation. A way to implement this is as follows:

On the interface create a chooser that is called debug-level and has the following choices [0, 1, 2, 3].

Then, in your program, your if-statement containing the *print*/*show* statements would look something like this (*print* is used in the first case as the values are not turtle-specific):

```
if debug-level > 0 [print (word "min = " min [my-probability] of
turtles "; max = " max [myProbability] of turtles "; mean = "
mean [myProbability] of turtles)]

if debug-level > 1 and my-probability < 0 or my-probability > 1
[show my-probability]

if debug-level > 2 and my-probability < 0 or my-probability > 1
[show (word my-probability " calculated by: " numerator "
divided by " denominator)]
```

With this structure, you would choose 0 for debug-level to turn the *print*/*show* statements off, 1 for aggregated statistics, 2 for problematic values of my-probability, and 3 for everything. This particular setup makes debugging cumulative, of course. You could set it up with equals to pick only a single level at a time.

Another option is to structure the execution of the model so that you have a debugging "system" created as part of the step button with code to ask the patches and turtles to print/show values of interest. This will work well for declared variables that have dynamics of interest that do not occur, typically, within a time step. For example, if you reset the value of a variable at the end of each time step, you need to make sure that your print/show commands happen before that reset. However, you need to make sure that you also do **not** change how the model is executing

by splitting up procedures (see the discussion on buffered and unbuffered agent activation), etc., or you will be debugging a **different** model!

To do this, one might structure the step button as such:

```
go
ask turtles [show-turtle-values]
ask patches [show-patch-values]
```

With `show-turtle-values` and `show-patch-values` being procedures for showing values, that might be problematic. For example:

```
to show-turtle-values
  if my-probability < 0 or my-probability > 1 [show my-
probability]
end

to show-patch-values
  if count turtles-here > 1 [show (word "count turtles-here, "
count turtles-here)]
end
```

Again, the notion here is that turtles have a probability (`my-probability`) that should stay between 0 and 1 and you want to know when it does not. Additionally, you are trying to keep no more than one turtle per patch, so you want the patches to let you know when that constraint is violated. Of course, this can be combined with the debugging levels to make a very flexible system for digging into your model.

Checkpointing and Initialization Control

Just like controlling the random number generator within your model, to really dig into the dynamics of a model, change very specific things, and find what might be very subtle bugs, you may want to completely control and replicate the initialization of your model or be able to start the model "further in time." When you create a population of turtles, you can give them a set of variables that are initiated when they are created, something like this:

```
turtles-own [happy? my-probability my-size]

to setup
  ca
  create-turtles 10 [turtle-setup]
end

to turtle-setup
  set happy? True
  set my-probability (random 100) / 100
```

```
  set my-size (random 10) + 1
end
```

This will create a population of 10 turtles. Once the procedures are complete, all turtles will be happy, and they will have a value of my-probablity between 0 and 0.99 and a value for mySize between 1 and 10. Every time you run setup, you will create a different population. As discussed above, if you add a line after the `clear-all` in the setup procedure to set the seed (`random-seed 55`, for example) you will create the same population every time. But, what if you want specific turtles to have specific values? You could create the setup procedure to create each turtle individually but that would be time-consuming and painful if you had more than a few turtles. In these cases, you often have a file that represents the population you wish to create in your model. For example, if trying to validate a Schelling-type model, it could be ethnicity, income level, and location of households from census data. This input file might look something like this (column order is record, ethnicity, income, x, y):

```
1 1 100 20 30
2 1 110 21 30
3 2 45 50 67
4 2 44 49 66
etc.
```

Now what you want to do is read the file line by line and create a turtle for each line of data. (Note: NetLogo's standard delimiter is a space. The input file used in this example, therefore, uses a space between values.) Again, for a neighborhood, you could probably hand-jam this, but if you are modeling a city or a metro region, you will need to automate the process. There are many ways to do this in NetLogo. The process described next is only one way to do it. Basically, what you are going to do is do all the setup procedures and then make a procedure that goes through your input file line by line, and for each line makes a turtle and gives it the appropriate variable values. Note: the file extension of "input" is just a random extension used for this example. Any extension is fine so long as you use spaces as the delimiter.

```
to setup
  ca
  do-other-setup-stuff
  make-turtles
end

to make-turtles
  file-open "myInputFile.input"
  while [not file-at-end?]
  [
    create-turtles 1 [turtle-setup file-read file-read file-read
file-read file-read]
  ]
end

to turtle-setup [a b c d f]
```

```
  set my-index a
  set my-ethnicity b
  set my-income c
  set xcor d
  set ycor f
end
```

The above example will create a turtle for every line in the input file, read in each value on the line, and pass those values into the `turtle-setup` procedure. The NetLogo command `file-read` will grab the next chunk of data in the file until it hits a space (which you will recall is NetLogo's delimiter). This will allow you to create very specific populations of turtles or patches based upon an input file that you have created.

Checkpointing the Model

Along with wanting very specific control over the initialization of your model, you may want to "checkpoint" your model. This will allow you to restart your model at any arbitrary point during a run. This can be particularly useful if your model has a significant transient period when it first starts running. As can be seen in Figure 5-9, the dynamics of your run can change significantly after a transience "burn-in" period. If your model runs slowly, being able to restart your model after the transient period can save significant time during development.

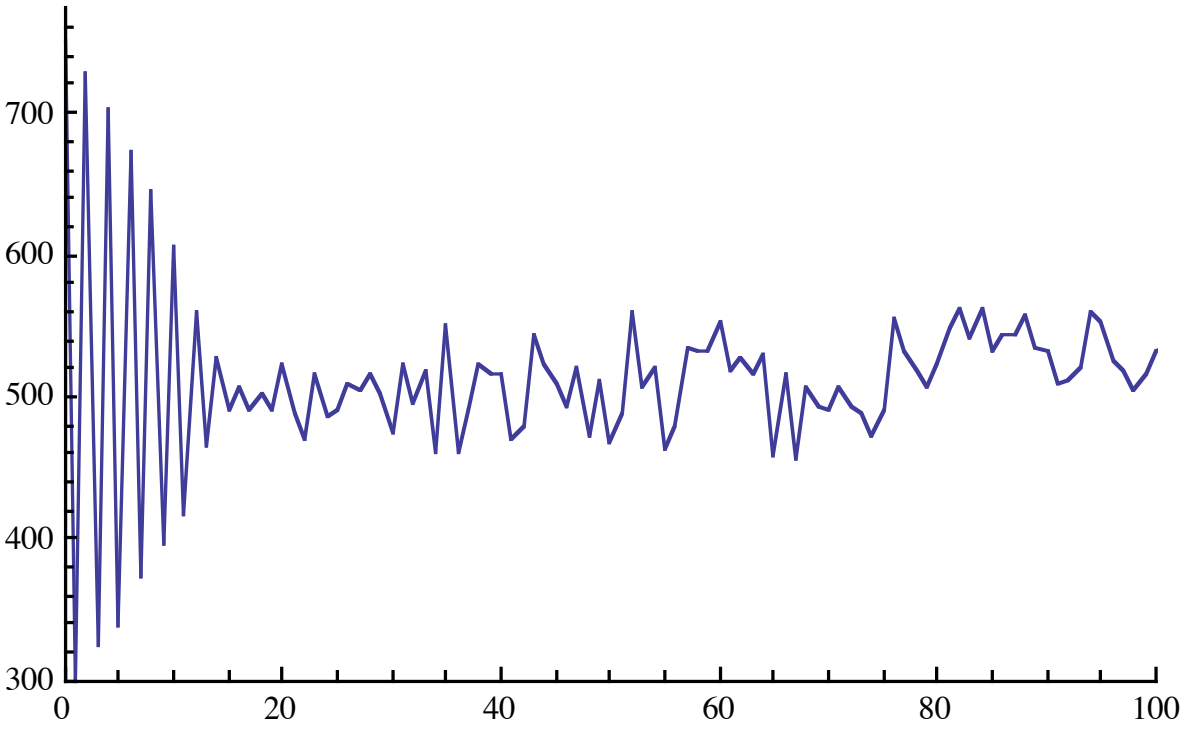


Figure 5-9. An example of the transience burn-in period that can happen when an agent-model first starts running.

To save the complete state of the model at any point, or checkpoint, use the `export-world` command. This command will export the complete state of your model into a .csv file. The model can then be reloaded with the `import-world` command. Note: this will create a very large file, depending upon the size of your model, and it will take a fairly long time to create. The same is true of reading it back into NetLogo. It can take ten or more minutes to read a large world file back into NetLogo. Note also: This will save the state of the random number generator, which means that if you want to see something different happen in the model from that point on, you will need to re-seed the random number generator or change something else manually. What is good about this is that you can reload a simulation run and then change one thing to see if it has any impact on the subsequent execution of the model. For example, you could load in a particular settlement pattern during the run of a Schelling model and then manually change the location of a turtle and see if that had any impact on the final settlement pattern.

Output Data

For very large numbers of turtles or patches, the above process of writing values to the command center, even when limited to values of particular interest, may be untenable. In these cases, you may need to write these statements to a file and then use a different program to sort through and understand the dynamics represented by the output. In this case, you will need to change the *print/show* statements to file-print and explicitly include the printing turtle/patch's identification. For example:

```
If debug-level > 1 and my-probability < 0 or my-probability > 1
[file-print (word self "," my-probability)]
```

This will generate a line in your output file such as this:

```
(turtle 33),0.8
```

This is a file that is easy to import into other programs like R and Excel. If the "(turtle " and the closing ")" is hard to deal with, as it can be, change the *self* to *who*. Now you would have a line of output like this:

```
33,0.8
```

This can be much easier to deal with for statistics and sorting, etc., but is not as human-understandable.

Breaking Your Model

This chapter has been about understanding how your model is working and finding potential problems. However, just because your model has some problems does not necessarily mean it is buggy and needs to be fixed. What you have created is a model that has a specific purpose, a specific question it was created to answer. That means it *should* break when you push it beyond

its intended purpose. It is always a good idea to test your model's performance in "corner cases." These are cases where input values are set to their extremes and the model is run testing various combinations of high and low extremes. Often times this will cause the model to break.

The point of this exercise is to understand where and how your model breaks. If it breaks suddenly and completely, say with a runtime error, then you have some confidence that as the model is pushed past the boundary of the theory that underlies it, it will fail clearly. That's a good thing. If the model fails in very subtle ways then it is much more difficult to know when the model moves out of its theoretic "comfort zone" and is potentially creating meaningless, or at least poorly understood, output.

It is also good to understand how the inputs and output relate to each other. One can do a formal design of experiments (DOE) to accomplish this but there are also other methods to do this, such as John Miller's Active Nonlinear Tests (ANTs) methodology. This methodology combines genetic algorithms and other heuristic search techniques with ideas from DOE to create a system that lets you find regions that break your model in an automated way. One of John Miller's students (Forrest Stonedahl) created a Java wrapper for NetLogo to do ANTs on NetLogo models. It is available here: www.behaviorsearch.org.

Understanding your agent model can be a nontrivial endeavor. However, there are many tools and techniques available to help you. If you start with a good formulation, then you can also, *a priori*, specify what are acceptable values for most if not all variables within your model. This will let you build debug statements as you code up the overall model. Moreover, with a strong formulation, you can specify interactions and relationships that can be used to develop very insightful visualizations that will help you not only debug your model, but also present it to others in a clear and compelling manner.

Additional Reading

Axtell, Robert. (2000). "Effects of Interaction Topology and Activation Regime in Several Multi-Agent System". *Santa Fe Institute Working Paper*. Santa Fe, NM.

Axtell, Robert, and Joshua. Epstein, (1994). "Agent Based Modeling: Understanding Our Creations". *The Bulletin of the Santa Fe Institute,* Winter (1994), pp. 28-32.

Fortner, B. and T. Meyer. (1996). Number by Colors: A Guide to Using Color to Understand Technical Data. Springer.

Miller, J. (1998). "Active Nonlinear Tests (ANTs) of Complex Simulation Models". *Management Science*, vol. 44, no. 6, June 1998, pp. 820-830.

Montgomery, Douglas. (2005). Design and Analysis of Experiments, 6th edition. John Wiley & Sons, Inc.

6. Integrating NetLogo with Java

Up to this point, we've been working exclusively within the NetLogo environment, defining models and running them using the NetLogo Graphical User Interface (GUI). For many modeling situations, this is just fine – you can develop models, run them, and analyze results visually using NetLogo display widgets like graphs and histograms, or quantitatively by exporting data collected at run time using tools like Excel or R. However, NetLogo models don't have to always be run from within the NetLogo GUI – you can embed your NetLogo model inside a Java class, compile it, and run it just like any other Java application. Since it's just a Java class, you can write code to access NetLogo variables at runtime and do whatever type of analysis, data exchanges, or special processing you'd like.

There are at least a couple of situations where this feature is quite handy. For example, if you would like to exercise fine control over the execution of your NetLogo model, such as taking an alternative course of action if some event is detected, then writing a Java class to manage the execution of your NetLogo model is a good way to go. Another situation involves integrating your NetLogo model with other models or libraries written in Java – having a Java class to manage your NetLogo model allows you to quickly integrate NetLogo with your other Java tools.

This chapter is a bit more technical than most of the previous chapters, and it is very much an "optional" discussion. If you're not a Java programmer, or you don't have close friends you could motivate to write Java code for you, then consider this a free pass and move on to other parts of the book. However, if you think you might need to run a NetLogo model from within a Java environment, then read on.

We're not going to go into the details of setting up your Java development environment here – this is a technical topic that is covered elsewhere, and since there are a number of options for Java development, it's not possible to address them all. The examples that follow will show how to integrate a NetLogo model into Java using the popular Eclipse environment (www.eclipse.org), however, any modern integrated development environment (IDE) should work just as well. No Eclipse-specific options or features will be used – everything is fairly generic.

Also, before going much further, we should note that the NetLogo online documentation provides an excellent discussion of how to integrate NetLogo into Java (and Scala also), so be sure to check out the link http://ccl.northwestern.edu/netlogo/docs/controlling.html . The NetLogo web page has a lot of examples, and we'll only look at a bare minimum here just to get started.

Before going much further, we need to include the NetLogo Java Archive File (JAR file) in our environment. The NetLogo jar file we need is conveniently enough named *NetLogo.jar*; put this jar file into your Java compiler's build path according to the conventions of your Java IDE.

Since we've already introduced a NetLogo model earlier (remember the modifications to the Schelling Segregation Model), let's build on that and try some Java integration.

Let's assume that we want to run this model as a Java application instead of running the model from within the NetLogo environment. In addition, let's assume that we want to run the model repeatedly and record the results of each run in a comma-separated-value (CSV) file that we can import into our favorite statistical number crunching package for later analysis. (Note: although we can accomplish both these things quite handily from within the NetLogo environment, we're trying to show Java integration here, so a little indulgence for the purpose of exposition is in order.) We'll consider two separate options: one with a NetLogo GUI, and one without a NetLogo GUI environment.

Running with a NetLogo GUI

The following code shows how to configure and run the modified Segregation model as a Java application. A couple of things are worth discussing here before going much further. Note that this code will actually launch a NetLogo environment and then manage the execution of the modified Segregation model from within the NetLogo system. You'll see the model being loaded, and then a series of commands are executed, just as if you were typing the commands at the NetLogo command prompt or pressing buttons on the NetLogo user interface GUI (although unless you are an extraordinarily fast typist, these things will be happening much quicker than in a normal NetLogo session!).

The model will run, looking exactly like a normal NetLogo model run. The "magic" happens because the command to make the model run, in this case the line *App.app.command("repeat 50 [go]")* is no different from any other Java statement. Although it causes a lot of action to take place on the NetLogo side, it is in fact simply a Java statement, and when it finishes, the program moves to the next Java statement in the program. This allows us to follow the run statement by other statements, such as polling the state of the model for key variables and printing out nicely formatted output. Follow along the code example below – it's commented at each of the major sections to guide you along.

```
/** SchellingGUI
 *
 * example demonstration of how to incorporate a NetLogo
 * model into a Java class so it can be run as a Java
 * application
 *
 */

import org.nlogo.app.App;

public class SegregationGUI {

public static void main(String[] argv) {
      //
      // setup the path to the model we want to run
      //
      final String netlogoHome =
       "C:/Documents and Settings/NetLogo";
      final String modelPath = "Segregation Research Project";
      final String modelName = "Segregation-updated.nlogo";
```

```
final String appString = netlogoHome + "/" + modelPath + "/"
      + modelName;

//
// setup how many times we want to run model here
//
final int numExperiments = 5;

//
// build the thread we want to use to run the model
//
App.main(argv);
try {
      java.awt.EventQueue.invokeAndWait(new Runnable() {
      public void run() {
            try {
            App.app.open(appString);
            } catch (java.io.IOException ex) {
            ex.printStackTrace();
            }
      }
      });

      //
      // repeat the experiment and record results
      //
      for (int i = 0; i < numExperiments; i++) {
      //
      // do the setup commands here
      //
      App.app.command("clear-all");
      App.app.command("setup");
      App.app.command("set model-extensions false");
      App.app.command("set number 2000");
      App.app.command("set %-similar-wanted 50");
      App.app.command("random-seed 76543");

      //
      // run the model here
      //
      App.app.command("repeat 50 [ go ]");

      //
      // when each run completes, print data here
      // if this is the first run, print CSV column headers
      //
      if (i == 0) {
            System.out
            .println("run-number, percent-similar, percent-unhappy");
      }
      //
      // print CSV values
      //
      System.out.print(String.valueOf(i) + ", ");
      System.out.print(App.app.report("percent-similar") + ", ");
      System.out.print(App.app.report("percent-unhappy"));
      System.out.println();
```

```
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        }
}
```

Running without a NetLogo GUI

Now that we've seen how to run a model in a way that looks pretty much like the standard NetLogo environment, let's try a different approach, this time without invoking a NetLogo GUI at all. Called running in "headless" mode, this version will fire up the model, run it without any apparent user interface, and print out the results as before. This is actually not much different from building and executing a simple, non-GUI command line Java application – the program runs and prints out results on the console.

Notice that the key difference from the prior Java example is that here we are using a HeadlessWorkspace object as the main workhorse in our program. As before, follow along the code below – comments are included at key points.

```
/** SegregationHeadless.java
 *
 * example demonstration of how to incorporate a NetLogo
 * model into a Java class so it can be run as a Java
 * application
 *
 * this version runs "headless", or without a NetLogo
 * GUI environment.
 *
 */

import org.nlogo.headless.HeadlessWorkspace;

public class SegregationHeadless {

      public static void main(String[] argv) {
      //
      // setup the path to the model we want to run
      //
      final String netlogoHome =
       "C:/Documents and Settings/NetLogo";
      final String modelPath = "Segregation Research Project";
      final String modelName = "Segregation-updated.nlogo";
      final String appString = netlogoHome + "/" + modelPath + "/"
            + modelName;

      //
      // setup how many times we want to run model here
      //
      final int numExperiments = 5;

      //
      // build the thread we want to use to run the model
```

```
        //
        HeadlessWorkspace workspace = HeadlessWorkspace.newInstance();
        try {
              workspace.open(appString);

              //
              // repeat the experiment and record results
              //
              for (int i = 0; i < numExperiments; i++) {

              //
              // do the setup commands here
              //
              workspace.command("clear-all");
              workspace.command("setup");
              workspace.command("set model-extensions false");
              workspace.command("set number 2000");
              workspace.command("set %-similar-wanted 50");
              workspace.command("random-seed 76543");

              //
              // run the model here
              //
              workspace.command("repeat 50 [ go ]");

              //
              // when each run completes, print data here
              // if this is the first run, print CSV column headers
              //
              if (i == 0) {
                    System.out
                    .println("run-number, percent-similar, percent-unhappy");
              }
              //
              // print CSV values
              //
              System.out.print(String.valueOf(i) + ", ");
              System.out.print(workspace.report("percent-similar") + ", ");
              System.out.print(workspace.report("percent-unhappy"));
              System.out.println();

              }

              //
              // all done, clean up
              //
              workspace.dispose();

        } catch (Exception ex) {
              ex.printStackTrace();
        }
        }
}
```

Going On From Here

So we've only just scratched the surface of what you can do with integrating NetLogo into a Java environment, and we haven't even touched Scala (which is another programming language used by NetLogo internally). Hopefully this brief introduction will give you a feeling for how you can incorporate NetLogo models into your Java systems. As we mentioned in the beginning of this section, integrating NetLogo and Java is not a requirement – you can do a tremendous amount of modeling within the NetLogo environment and toolset, so don't feel a need to migrate over to Java unless you need to. However, if your project needs to exchange data between Java classes and a NetLogo model, the above examples and the NetLogo manual pages should get you well on your way.

7. Networks in NetLogo

May interesting structures in social systems can be represented using networks. In the simplest terms, networks are simply a collection of nodes (some "things") and some designation of how the nodes are connected. There are, essentially, no constraints on what a node can be or how these nodes can be connected. The same holds true in NetLogo. Anything that can be a turtle can act as a node, and any turtle can be connected to any other turtle, though a node cannot be connected to itself via a link. Furthermore, if there are different ways turtles can be connected, you can explicitly represent that feature in NetLogo using different types (or breeds) of links. This is an especially powerful feature that allows for very rich modeling of social phenomena.

Before we start to work through some examples of networks, it should be noted that there are two ways to use networks within NetLogo. The first is the topic of this chapter, explicit representation of networks via links connecting turtles (nodes). In this approach, nodes are represented as turtles, and links are used to represent connections among nodes. This is the "default" way of building networks in NetLogo, and it's the approach used for the examples in the NetLogo models library.

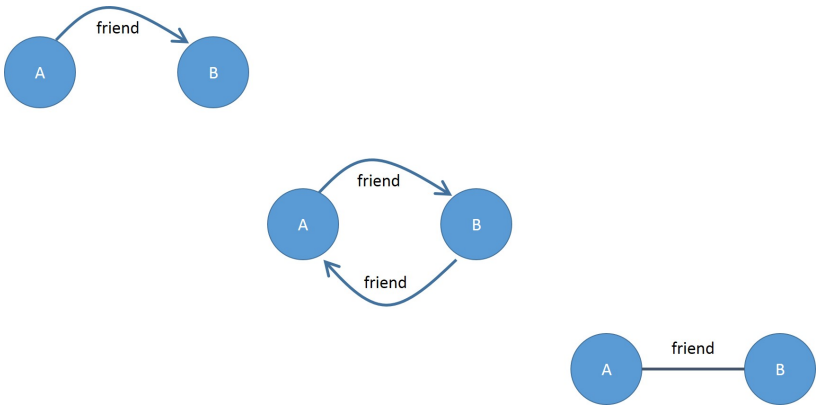
The other way to represent networks is to implicitly represent networks without explicit connections via links. When utilizing this method each turtle keeps track of their connections via lists or some other data structure. This method might be preferable to explicit representation as it will require fewer agents (since links are a type of agent), and likely will use less memory and fewer resources to refresh the screen when watching the model run. However, this method makes it much less obvious which nodes are connected, since all the links are internally managed within each node and not readily visible. Having said that, we will now turn our attention to using links to explicitly represent networks within NetLogo (although there will be an example of using plots to depict your network rather than links).

Building networks in NetLogo

Within NetLogo you build networks by connecting turtles together with links. Links are a type of turtle. As such, they can be "plain" or "breeded", although plain and breeded links cannot coexist in the same model. It will save you some significant heartache if you decide upfront how you want to structure your model and if you will need different kinds of links. Also, links may be either undirected or directed. Directed links have an explicit source and target node, whereas undirected links simply connect two nodes.

For example, turtle A may consider turtle B to be their friend but the feeling may not be mutual. In such case, we say there is a directed "friendship" link between turtle A and turtle B, but there is no reciprocating link from turtle B to turtle A. If turtle B decides to befriend turtle A, then there would be directed "friend" link from turtle B to turtle A, and we would have two distinct directed links in this simple network. In NetLogo if two agents make directed links of the same kind to each other you will have two directed links, one in each direction rather than a single undirected link. If the two turtles were friends from the start, we could represent their mutual

friendship with a single undirected "friend" link between them. These situations are shown in the figure below.



Let's try that out to make sure, typing the following commands at the NetLogo Observer prompt…

```
create-turtles 2 [forward random 10]
ask turtle 0 [create-link-to turtle 1]
show count links
ask turtle 1 [create-link-to turtle 0]
show count links
```

The first line of code, `create-turtles 2`, creates two new turtles. By convention, NetLogo sequentially assigns each turtle a guaranteed unique internal ID number (called it's "who" number), which in this case means the first turtle is assigned who ID number 0 and the second is assigned who ID 1.

Running this code, you should have two links now, right? The first time `show count links` is called it will show a count of 1, the second time it is called will show a count of two. You should also see what looks like a double-headed arrow linking the two turtles. This double-headed arrow is actually two directed links one directed link from turtle 0 to turtle 1 and vice versa. The term `create-link-to` will create a directed link from the calling turtle to the referenced turtle. So, in the above example the line: `ask turtle 0 [create-link-to turtle 1]` will create a directed link from turtle 0 to turtle 1. If we instead use `create-link-with` then turtle 0 will create an undirected link with turtle 1. You can also use `create-link-from` to create a directed link to the calling turtle from the referenced turtle.

There are a number of built-in functions of which you can take advantage of when using a network with links. Turtles can ask their link neighbors to do things, that can be extended to include a turtles out-link neighbors and in-link neighbors. Also, you can use all the usual agentset commands with a turtle's link neighbors, such as `one-of` or `n-of`.

Since links are considered full-fledged objects (technically they are a type of turtle), each link can also have it's own variables (though links DO NOT have position in the same way as turtles: for example you cannot get a distance from a patch to a link). This means you can add weights,

for example, to the connections between your turtles. With this capability you can implement Granorvetter models in NetLogo quite easily. Granorvetter models are threshold models that take place on a network. These can be powerful ways to think about social processes. For example, an agent may not do something unless they experience "enough" peer pressure. In the example we're going to develop shortly, agents are connected via their ability to exert peer pressure on each other. This requires directed links that contain a weighted value representing the amount of peer pressure one agent can exert on another.

Let's now work up a more complicated model of social influence. We will create a set of agents linked via a social influence network. This time we will used breeded links, where each link manages a local variable. The specific type of link we will create is social influence and the variable it will have will be `myInfluence`.

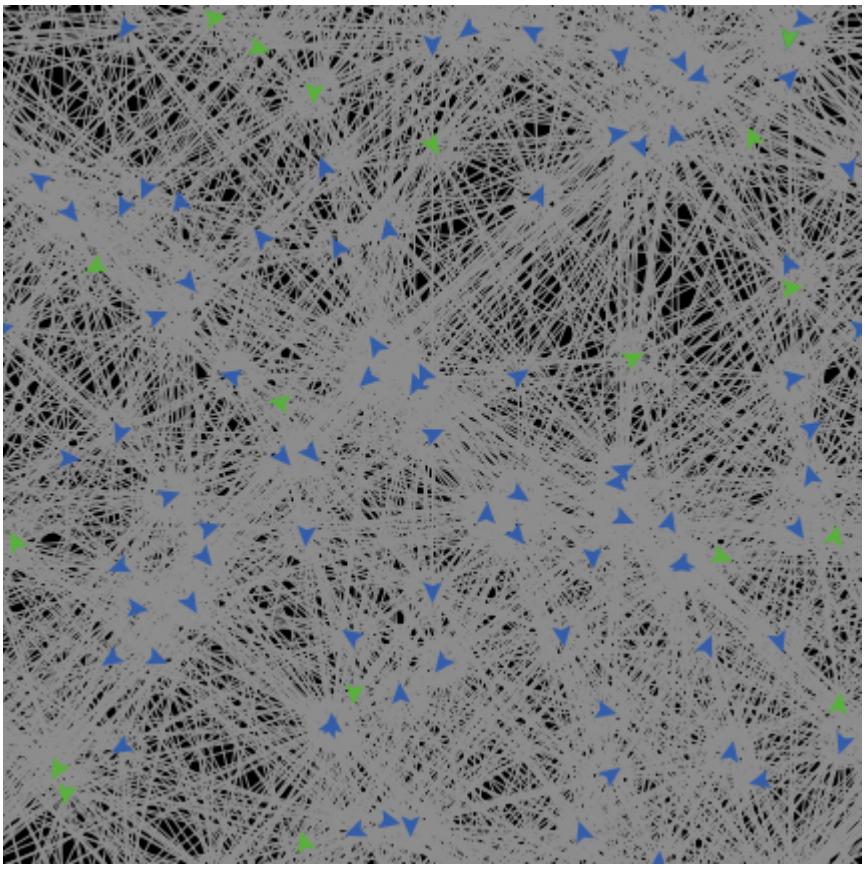
First we need to specify our breeds. Here we only need one breed, in this case that of a breeded directed link. That can be created as follows (note that if we were creating an undirected link that would be done analogously via the `undirected-breed-link` command). Here's the code to implement the model.

```
; social influence model
;
; demonstrates how to use directed links in NetLogo
;
;

; declare a breed for the directed links
directed-link-breed [socialInfluence a-socialInfluence]

; declare some turtle-specific attributes
turtles-own [
     adopted?       ; flag to indicate if influence was adopted
     myThreshold    ; level for my threshold of being influenced
     myInfluence    ; level indicating how much influence I have
]

socialInfluence-own [] ; list to hold social influence of
turtles


;
; setup
;
; sets up the model
;
to setup
  clear-all
  create-turtles 100
```

```
  ask turtles [turtle-setup]
  ask one-of turtles [set adopted? True]
  reset-ticks
end

;
; set up the turtles
;
to turtleSetup
  ;; move the turtles around so there is something to look at
  forward random 100

  ;; initialize the turtle variables
  set adopted? false
  set my-influence random 100
  set my-threshold random 100

  ;; build out the network
  ;; first pick a number of turtles to be influenced by
  let t random ((count turtles * .5) - 1)
  let x nobody
  ;; iterate over the list of turtles and make links from them
  ;; to you until you have the number that you picked before
  while [count in-a-socialInfluence-neighbors < t]
  [
    set x one-of other turtles
    if not in-link-neighbor? x [create-a-socialInfluence-from x]
  ]
end

;
; main go method for the simulation
;
to go
  ask turtles [check-on-peer-pressure]
  tick
end

;
; this method implements the agent checking on peer pressure
;
to check-on-peer-pressure
  ;;
  ;; here is where we'll do the threshold stuff...
  ;;
  if sum [my-influence] of in-a-socialInfluence-neighbors
     with [adopted?] > my-threshold
```

```
  [set adopted? True]

  ;;
  ;; color code turtles based on whether or not adopted
  ;;
  ifelse adopted?
  [set color blue]
  [set color green]
end
```

In this example we didn't need to have breeded links but wanted to use them. Here's what the model looks like.



Creating Specific Kinds of Networks

The following code examples show how to create specific kinds of networks. The examples include random graphs, small worlds graph, lattes, ring, and scale-free. For a thorough examination of the characteristics of these networks and why/when you might want to use one over the other, see (Newman, 2010). [2]

[2] M.E.J. Newman (2010). Networks: An Introduction. Oxford University Press.

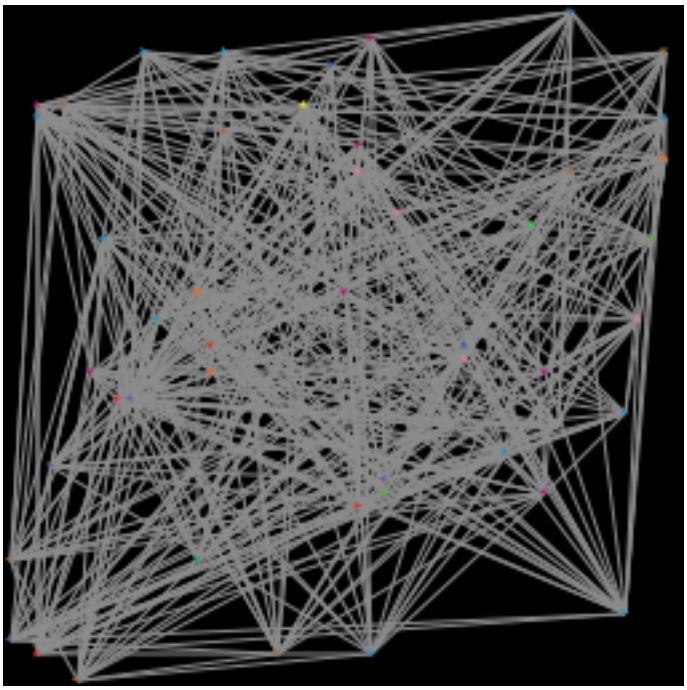
The following block of code shows how to build particular kinds of networks, including: an Erdos-Renyi random network, a Watts-Strogatz small-worlds network, a Barabasi-Albert scale-free network, a regular lattice, and a ring network. It should be noted that these examples are just that, examples, just something to get you pointed down the right path. When you implement your network code you should test the network statistics to ensure you are creating the network you wish to create. For example, the Barabasi-Albert network example is a proportional attachment algorithm (Simon, 1955).[3] As with other algorithms that generate as with other algorithms that generate scale-free distributed degree distributions, it will only be scale-free in the limit.

The Erdos-Renyi network generates a "random" network in which each node has a probability p of being connected with another node. In this example, the value of p has been set arbitrarily to 0.20, but this can of course be adjusted as needed.

```
; ================================
; build random Erdos-Renyi network
; ================================
to build-random-ER-network
  clear-all
  create-turtles 100
  let p 0.20
  ask turtles [
    let me self
    let nn sort other turtles
    foreach nn [
      let chance random-float 1.0
      if chance <= p [
        ask self [
        create-link-with ?
          ]
        ]
      ]
    ]
end
```

The network that this code creates (here shown with 50 turtles) can be found below:

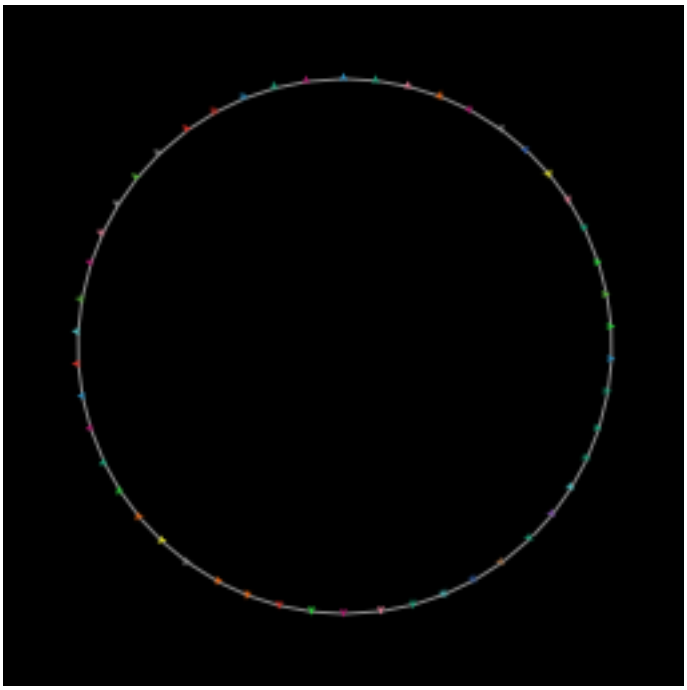
[3] Herbert Simon (1955) "On a Class of Skew Distribution Function", *Biometricka*, vol. 42, issue 3/4, pp. 425-440



A ring network is a network in which each node is connected to two neighbors. For this implementation, we rely on the NetLogo built-in agent identifier "who" value, which is a sequentially assigned unique identifier assigned to an agent when it is created. Also note that since the links we create are undirected, we don't have to create an explicit link to two neighbors. An undirected link from agent i to agent i+1 by definition creates a symmetric link from agent i+1 back to agent i, so we only have to make one call to `create-link-with` in the code example below. If the links were directed, however, this symmetry would not exist and we would need to explicitly create links in both directions. Here's the code for a ring network.

```
; ========================================
; build ring network
; ========================================
;
to make-ring-network
  clear-all
  create-turtles 100

  ask turtles
  [create-link-with turtle ((who + 1) mod (count turtles))]
end
```

The network that this code creates (with 50 turtles) can be found below:

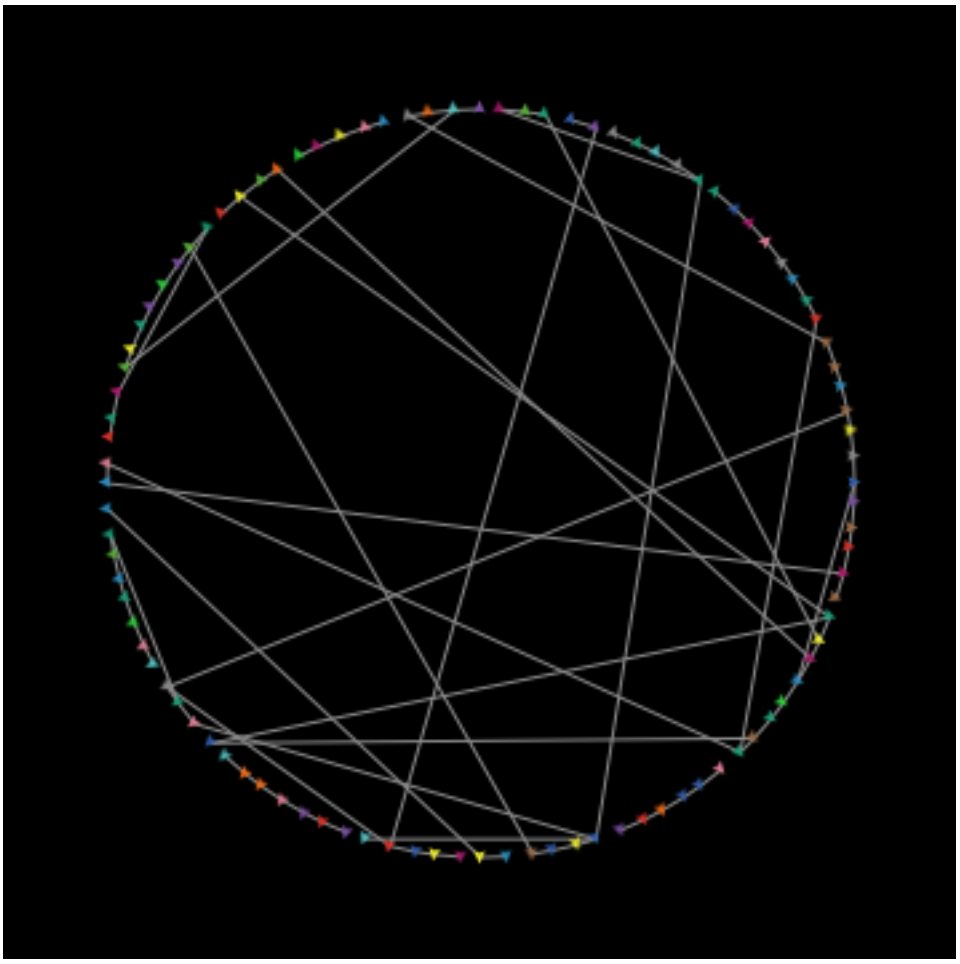


A Watts-Strogatz network creates a network in which small groups of nodes are well connected into small cliques, but certain nodes also maintain links outside their immediate groups to other cliques. The code operates by building a ring network initially, then allowing some nodes to attempt to rewire by connecting with distant nodes according to a probability p, which in this case is arbitrarily set to a value of 0.05.

```
; ========================================
; build Watts-Strogatz small-world network
; ========================================
;
to make-small-world-network
  clear-all
  create-turtles 100
  ask turtles
  [create-link-with turtle ((who + 1) mod (count turtles))]
  make-long-jumps
  ask turtles [try-to-rewire]
end

to make-long-jumps
  foreach sort turtles
  [
    ask ? [try-to-rewire]
  ]
end

to try-to-rewire
```

```
  let rewire-prob .05

  let temp sort link-neighbors
  foreach temp
  [
    if random-float 1.0 < rewire-prob
    [
      let x one-of other turtles
          with [abs (who - [who] of myself) > 2]
      if is-turtle? x
      [
        ask link-with ? [die]
        create-link-with x
      ]
    ]
  ]
end
```

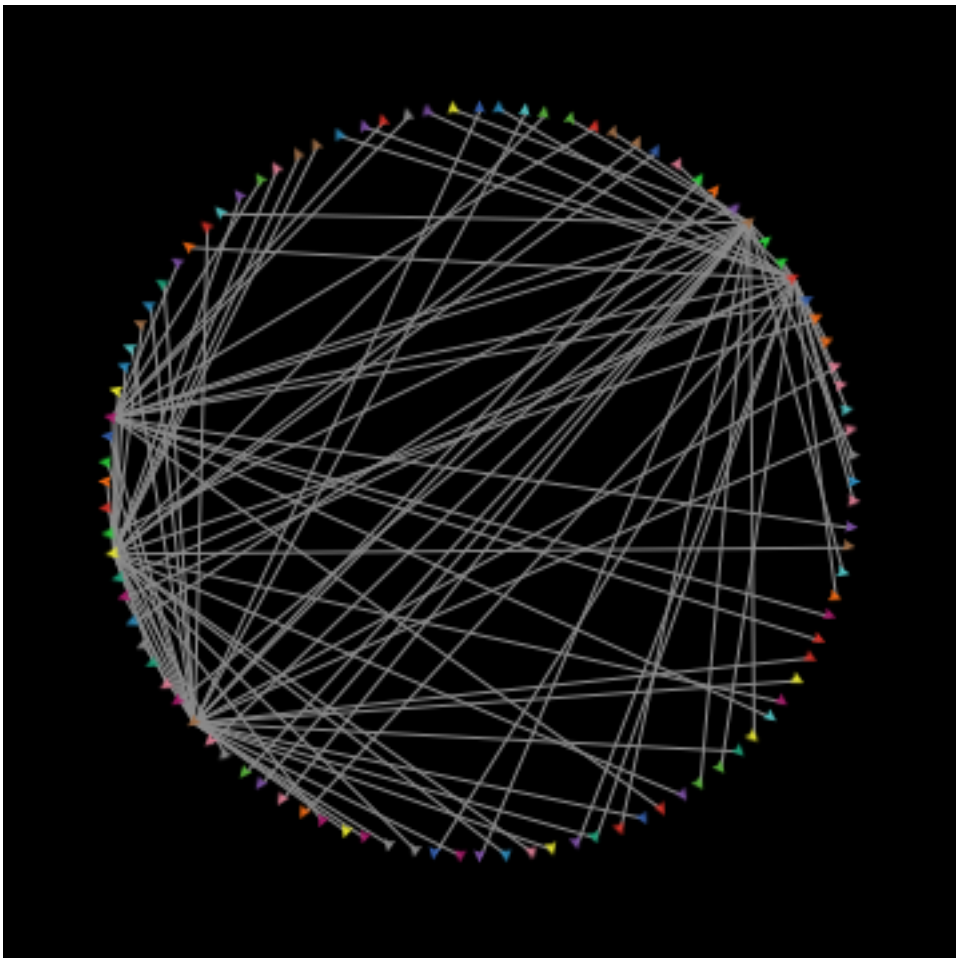
The network that this code creates (with 50 turtles) can be found below:



The Barabas-Albert scale-free network is designed to exhibit "preferential attraction" as the network grows, a feature in which new nodes joining the network are more strongly attracted to existing nodes of high degree. In this code example, the value m determines the upper limit of how many neighbors a node considers making a link with, and the value p governs the probability of actually making a link with a neighbor. The last bit of code cleans up any isolates that may accidentally be hanging around and attaches them to a high degree node.

```
; =======================================
; build Barabasi-Albert scale-free network
; =======================================
to make-scale-free-network
  clear-all
  create-turtles 100
  let m 5
  let p .05
    ask turtles [
      ;
      ; find high degree distributions nodes
      ;
      let me self
      let degrees max-n-of m turtles [count link-neighbors]
      foreach (sort degrees) [
        let chance random-float 1.0
        if (? != self) and (chance < p) [
          ask self [
            create-link-with ?
          ]
        ]
      ]
    ]
    ;
    ; now catch any isolates and attach to high degree nodes
    ;
    ask turtles with [(count link-neighbors) = 0] [
      let degrees max-n-of m turtles [count link-neighbors]
      let t one-of degrees
      ask self [create-link-with t]
    ]
 end
```

The network that this code creates can be found below:

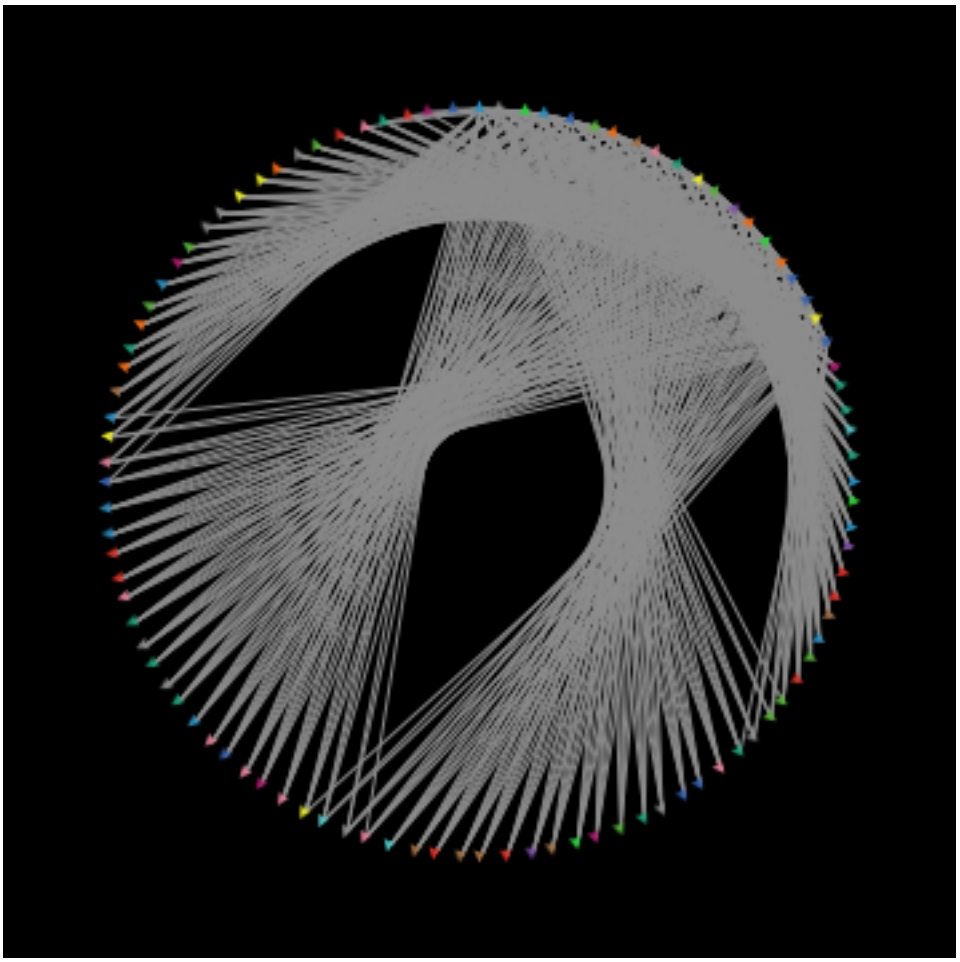


A Random Uniform Link network creates a network in which each node is assigned a constant number of links. In this example, each node is assigned k=5 links, and the network consists of N=20 nodes.

```
  ; =====================================
  ; build random uniform link network
  ; =====================================
to make-random-uniform-network
  clear-all
  create-turtles 100
  let k 5
  let n 20
  ; assign each node k neighbors
    ;
    ask turtles [
      let knn []
      let i 0
      let j 0
      let my-i [who] of self
      let t 0
```

```
      ;
      ; get k neighbors
      ;
      set i 1
      set j 0
      while [ i <= k ] [
        set j ((my-i + i) mod n)
        set knn fput j knn
        set i (i + 1)
      ]

      ;
      ; connect the links
      ;
      foreach knn [
        set t turtle ?
        create-link-with t
      ]
    ]
end
```

The network that this code creates can be found below:

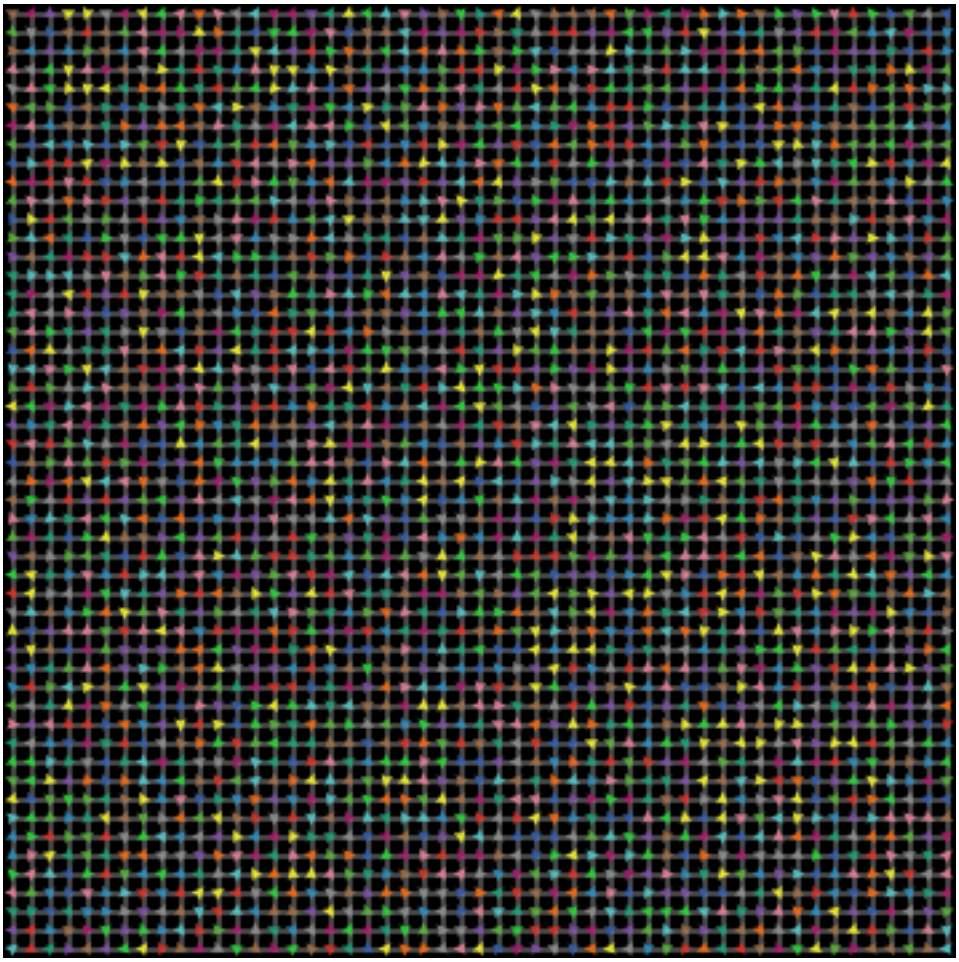


A Regular Lattice network creates a very regular lattice structure in which each node is assigned links in a geometric pattern. In this code example, we ensure that there is one and only one turtle on each patch, then we assign a link from that turtle to its four Von Neumann neighbors (N, S, E, and W).

```
;==================================
; build a lattice network
;==================================

to make-regular-lattice-network
  ;; this assumes there is one and
  ;; only one turtle on each and every patch
  clear-all
  ask patches [sprout 1]
  ask turtles [
     foreach sort turtles-on neighbors4 [
          create-link-with ?
     ]
  ]
```

```
end
```

The network that this code creates can be found below:



Now that you know how to make a few types of networks don't be limited by these. Social networks are dynamic, part of the fun and challenge of modeling social networks is trying to figure out what rules agents can use that will produce networks that have properties similar to (sometimes even identical to) these canonical networks shown above.

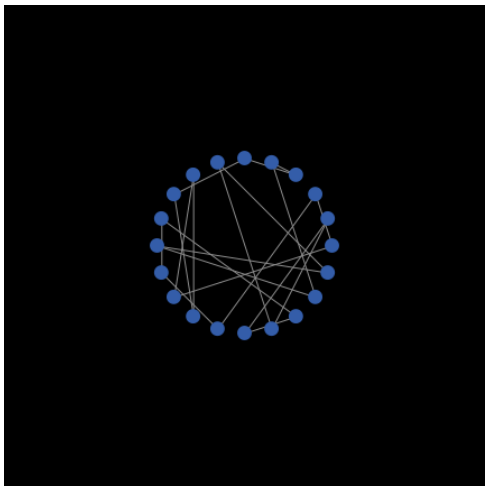
Visualizing Networks in NetLogo

Careful visualization the network(s) you have created can be very helpful in understanding and presenting the model that you created. NetLogo has a number of built-in functions for laying out a network in a logical way. Do not be tied to a particular network layout, however. Different networks will give you different understanding about the network and which nodes/links are the most important. It may also be useful to make the network layout an "offshoot" of the main dynamics of your model--remember often the model is an outgrowth of the dynamics of the model, as such, don't let the network visualization drive your model dynamics. Meaning, build your model and let the agents run around and do their thing, then freeze the model and visualize

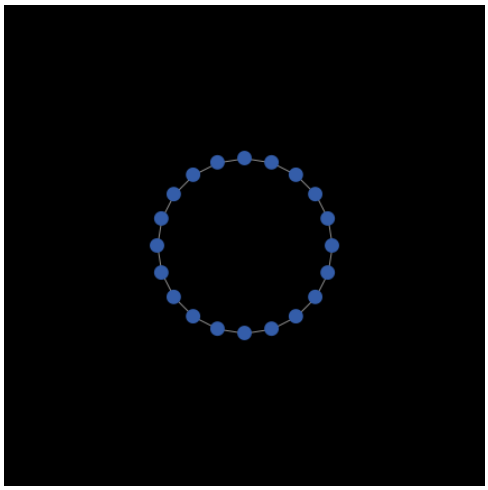
the network that the agents have created, then go back to the model. To do this you will need to create a separate procedure and give the turtles a few additional variables. We'll explore how to do this next.

Now we will turn our attention to the display of networks in NetLogo. When using links to depict the network structure within your model you can use a number of built in functions to automatically display your network. This will change the location of your agents so DO NOT USE these algorithms if the precise location of the agent is important within your model. You should also be aware of the computational demand needed for the visualization. For example, the radial spring layout can take a long time to run as it is iterative and involves significant computation on all links. If you have a large number of links, this layout, while very good at showing the clustered structure of your network, can take a significant amount of time to run. You may need to experiment with different layouts to find the best one to highlight the structure of your agents' network.

One of the most basic network layout algorithms is to simply lay the agents out in a ring. This is very simple within NetLogo. The command is `layout-circle`, which expects two parameters: an agentset or list of agents, and a radius for the circle. If you use an agentset then the agents will be arranged around the circle in random order. If, however, you pass `layout-circle` a list, then the agents will be arranged around the circle in their list order starting at the top of the circle moving around clockwise. For example, in the figure below on the left is an example of a ring network that is displayed using the layout-ring algorithm with an agentset so the turtles are arranged around the ring in random order. On the right, we use a sorted list so now the ring network appears as a ring.



layout-circle turtles 6

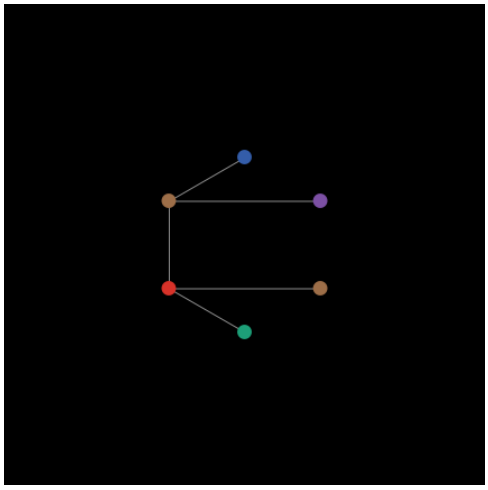


layout-circle sort turtles 6

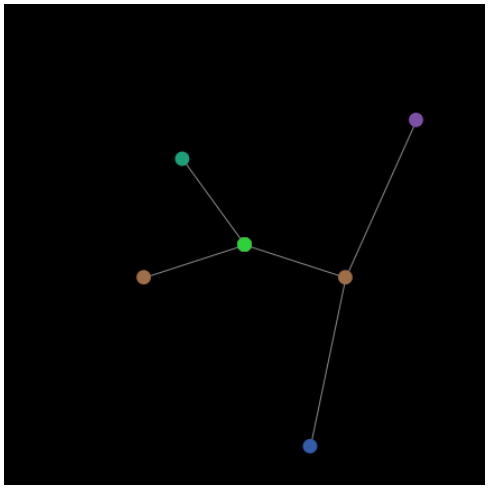
If we move to a random network we can explore other network layouts. If your agents have a tree or hierarchical structure to their network, then the `layout-radial` algorithm might be a good choice for the visualization. If we create the very simple tree network as described in the NetLogo manual we can display it as both a circle and as a tree.

```
to make-a-tree
  set-default-shape turtles "circle"
  create-turtles 6

  ask turtle 0 [
    create-link-with turtle 1
    create-link-with turtle 2
    create-link-with turtle 3
  ]
  ask turtle 1 [
    create-link-with turtle 4
    create-link-with turtle 5
  ]

  ; do a radial tree layout, centered on turtle 0
  ifelse do-layout?
     [layout-radial turtles links (turtle 0)]
     [layout-circle turtles 6]
end
```

Although this is a very simple example, in the figure below you will notice that the circle layout (left side of the figure) obscures the network structure that is highlighted with the radial layout (right side of the figure). To create the radial layout use the command `layout-radial`. The `layout-radial` command expects three parameters: an agentset, the set of links to use, and an "anchor" agent. The anchor agent will be placed in the center of the world and the network will grow out radially from there.



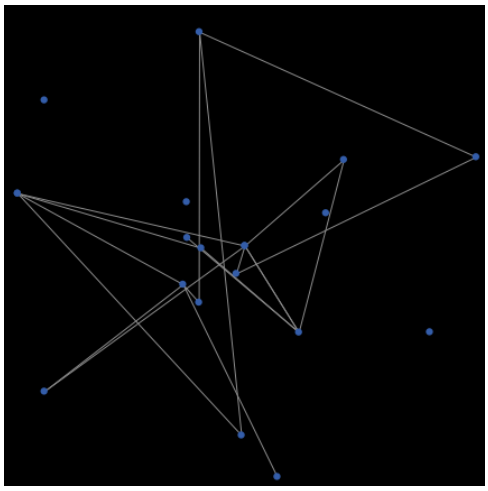
Circle layout



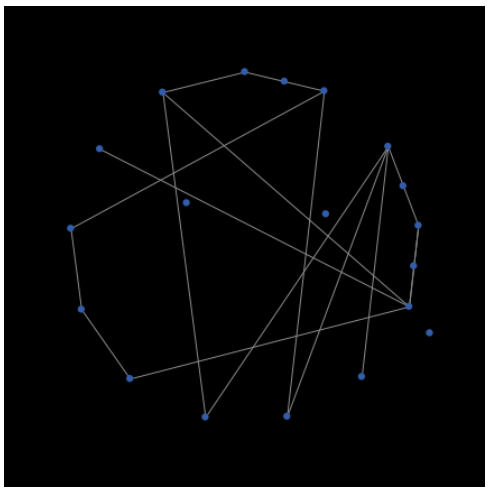
Radial layout

Now we will turn our attention to two iterative layout algorithms: tutte and spring. An informal way to think about the tutte layout is as a combination of radial and circle. For tutte you define a set of anchor agents that are placed around a circle, other agents are then placed inside that circle

based upon polygons defined by the link neighbors. This can be seen in the figure below. The left picture is a random network randomly laid out. The right side is the same random network but with a tutte layout. To create a tutte layout use the `layout-tutte` command. The layout-tutte command expects three parameters: a turtle set, a link set, and a radius.

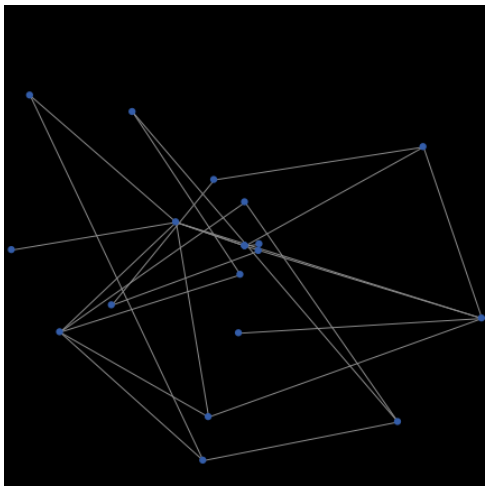


Random layout of a random graph

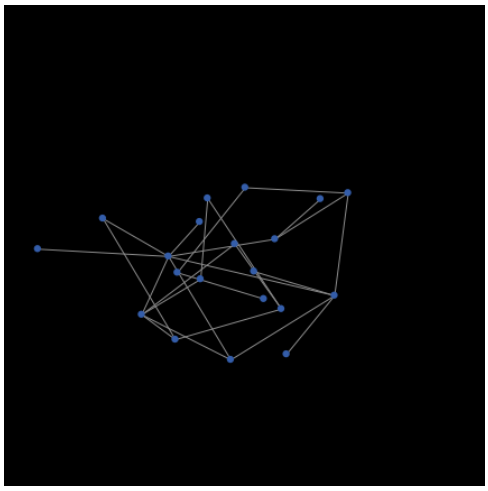


Tutte layout of the same random graph

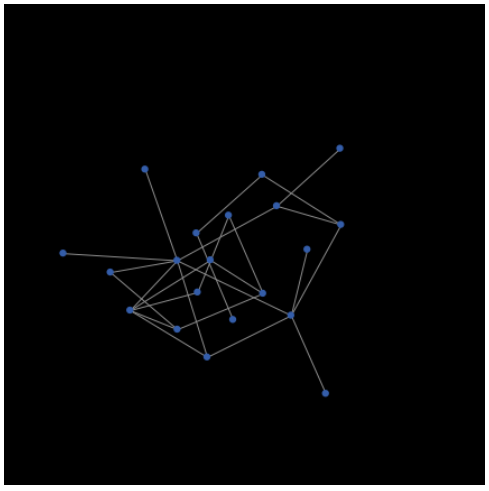
The final network layout algorithm is the spring algorithm. This is probably the most entertaining to watch but also the most computationally intensive so use caution if you have a large network! The spring layout algorithm treats links like rubber bands that are trying to achieve their relaxed length by pushing nodes away or pull nodes together, and nodes repel each other. Like the tutte layout, this is an iterative algorithm so you need to repeatedly call it. To use this algorithm, call `layout-spring`. The `layout-spring` expects five parameters: a turtle set, a link set, a spring constant, a spring length, and a repulsion constant. The turtle set is the set of turtles that you would like to be moved by the algorithm, turtles that are not part of this set will not be moved and act like anchors. The link set is simply which links to use as springs. The spring constant is how "hard" a spring will try to achieve its zero force length that is designated by the spring length parameter. Finally, the repulsion constant is the force nodes use to push each other away. There is no "best" setting for the spring and repulsion constants. What will work best will depend upon your network, so experiment with it. In the figure below you can see the progression from an initial random layout to 5000 iterations of the spring layout algorithm using all turtles, all links, a spring constant of 0.2, spring length of 5, and a repulsion constant of 1.



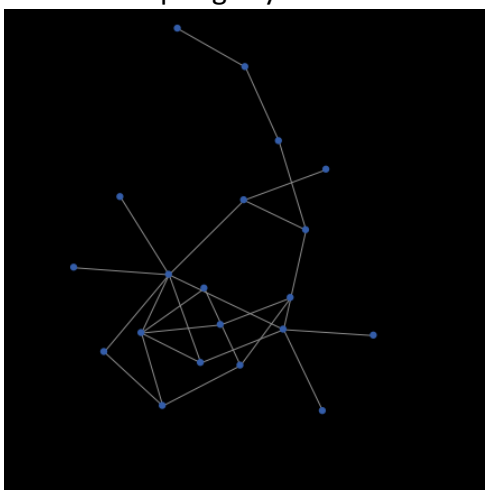
Initial Random Start



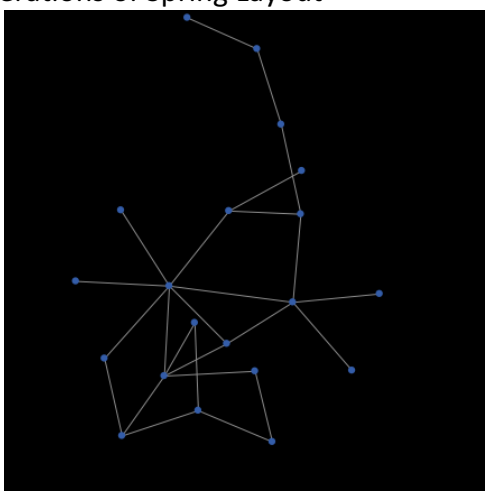
10 Iterations of Spring Layout



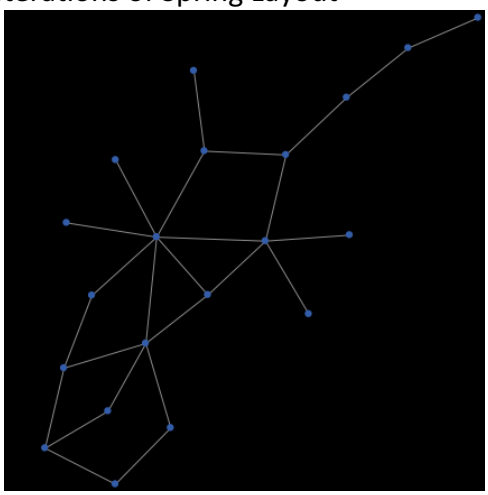
50 Iterations of Spring Layout



500 Iterations of Spring Layout



1000 Iterations of Spring Layout



5000 Iterations of Spring Layout

As can be seen in the above figure, the spring layout does a nice job of showing network structure that is largely hidden by a random layout. Unfortunately, it takes 500 iteration of the algorithm before that structure starts to emerge.

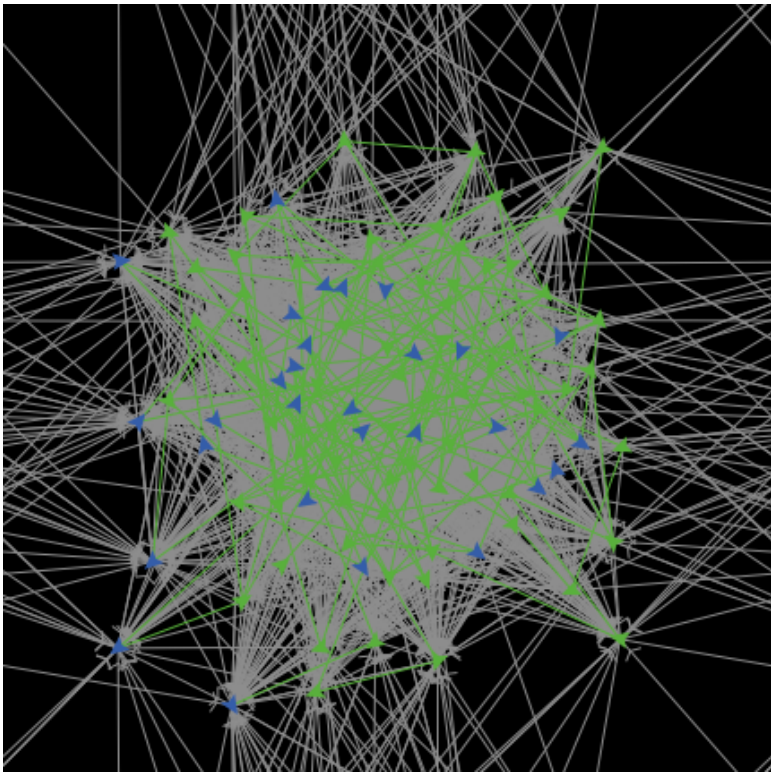
As one last example, let's turn back to the influence example that we started originally. Now we will add an undirected link called friendship (remember breeded and unbreeded links cannot exist together in the same model). We'll use these links to build up a separate friendship network from the influence network. Then we'll use the `layout-spring` on the friendship network to highlight the structure of that network while also keeping the influence links around so you can see how folks can be influenced. Things to notice: use of undirected link breed, and the use of layout-spring that uses the friendship ties rather than the influence links for the layout. Since this is a random network the layout-spring algorithm does not reveal much structure.

```
; social influence model
;
; illustrates use of links in a social network
; and also propagation of influence
;

; declare breeds for directed and undirected links
directed-link-breed [socialInfluence a-socialInfluence]
undirected-link-breed [friendship a-friendship]

turtles-own [
   adopted?
   my-threshold
   my-influence
   my-previous-x
   my-previous-y
]
socialInfluence-own []

;
; setup
;
; setup the model
to setup
  clear-all
  create-turtles 100
  ask turtles [turtle-setup]
  ask one-of turtles [set adopted? true]
  ask friendship [set color green]
  reset-ticks
end

;
; turtle-setup
```

```
;
; setup the turtles
;
to turtle-setup
  ;; move the turtles around so there is something to look at
  forward random 100

  ;; initialize the turtle variables
  set adopted? false
  set my-influence random 100
  set my-threshold random 100

  ;; build out the network
  ;; first pick a number of turtles to be influenced by
  let t random ((count turtles * .5) - 1)
  let x nobody
  ;; iterate over the list of turtles and make links from them
  ;; to you until you have the number that you picked before
  while [count in-a-socialInfluence-neighbors < t]
  [
    set x one-of other turtles
    if not in-link-neighbor? x [create-a-socialInfluence-from x]
  ]
  ;; build the social network
  ;; and make sure that everyone has at least one buddy
  let f random 5 + 1
  let af nobody
  ;; iterate over the list of turtles to find your friends
  while [count a-friendship-neighbors < f]
  [
    set af one-of other turtles
    if not link-neighbor? af [create-a-friendship-with af]
  ]
end

;
; go
;
; main driver for simulation
;
to go
  ask turtles [check-on-peer-pressure]
  repeat 10 [layout-spring turtles friendship .2 5 1]
  tick
end

;
```

```
; check-on-peer-pressure
;
; method to model peer pressure among agents
;
to check-on-peer-pressure
  ;; here is where we'll do the threshold stuff...
  if sum [my-influence] of in-a-socialInfluence-neighbors
     with [adopted?] > my-threshold [set adopted? true]

;
; color code agents
;
  ifelse adopted?
  [set color blue]
  [set color green]
end
```

Here's what the model looks like when run.



Remember we mentioned that sometimes you may want the network visualization to be an "offshoot" of the main model? Using the above network visualization algorithms will change the position of your agents. This may or may not matter. If it does matter you will want to be able to "unwind" the visualization so that your agents can get back to where they started. Now we will make use of the turtle variables `my-previous-x` and `my-previous-y`.

```
to do-network-layout
  ask turtles [
    set my-previous-x xcor
    set my-previous-y ycor
  ]
  ;; your network visualization algorithm goes here
end

to reset-agent-positions
  ask turtles [
    set xcor my-previous-x
    set ycor my-previous-y
  ]
end
```

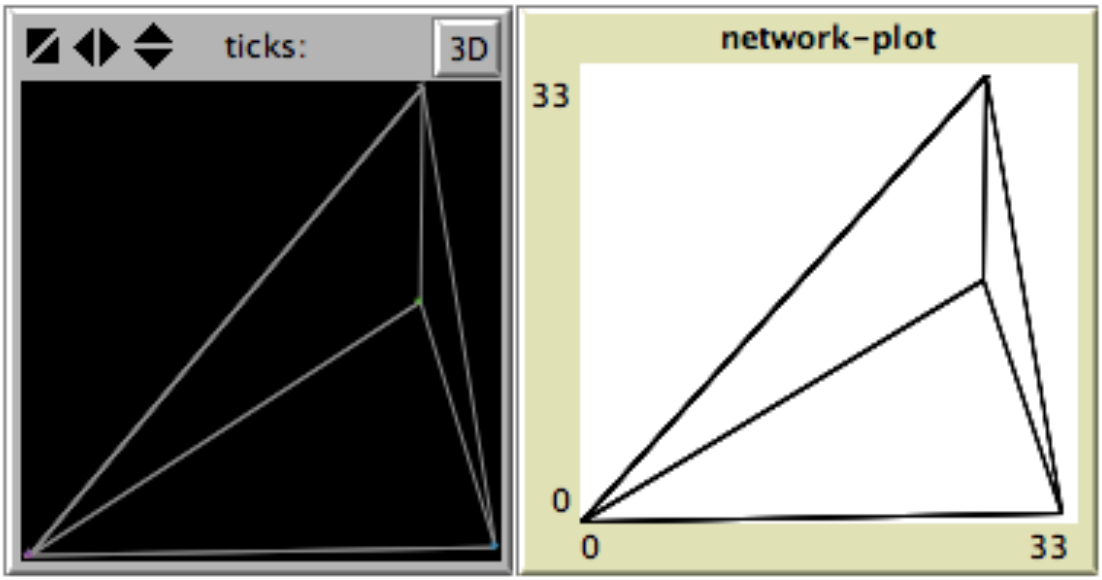
For more advanced modeling, it should be noted that the above procedure does not maintain the state of the random number generator so running your model with the same initial random seed and then using this procedure or not will likely produce different results.

Don't forget you can also be clever about using plots to display network information. We can write a simple routine to iterate over our agents and display a graph as follows, rendering the network inside a plot widget instead of on the main screen.

```
;
; update-network-plot
;
; draw a network inside a plot widget named "network-plot"
;
to update-network-plot
     set-current-plot "network-plot"
     clear-plot
     foreach sort turtles
     [
       ask ? [ plot-my-network ]
     ]

end

;
; plot-my-network
;
to plot-my-network
set-current-plot "network-plot"
foreach sort link-neighbors
[
     pen-up
     plotxy xcor ycor
```

```
      pen-down
      plotxy [xcor] of ? [ycor] of ?
]
end
```

When run on a small random network this plot will look like this, see below (the network is also depicted using links to show that the plot worked).



Doing something like this will let you see the network, albeit depicted somewhat simplistically, with out cluttering up the main agent view. Moreover, you can tune the refresh rate of the plot based upon how dynamic your network is. This can improve performance as you are not creating link agents or sampling a network every time step regardless of its actual dynamics. Moreover, the x,y space of the plot does not necessarily need to correspond to the x,y coordinates of the agents. You can use the plot to show network structure in other 2D space, e.g., wealth and intelligence or influence and social power.

As you become more comfortable using networks in your models you should investigate using the NetLogo NW extension. The NW extension contains a set of network specific primitives for creating, analyzing, and importing/exporting networks. The most important thing to remember when using it is to make sure that you have set the context (which nodes and links to pay attention to) correctly and updated it as necessary. Updating the context is also necessary for exporting a network. Not setting up the context correctly will mess up your metrics and could mean you do not export the network you expect. The current functionality of the network extension (nw:____) can be found in the NetLogo documentation.

8. Using Geospatial Data in NetLogo

The default patch landscape in NetLogo is great for building abstract models. However, there are many applications where it is useful to have a much more literal representation of space. NetLogo provides several ways to add spatial detail to your models. The approaches vary in terms of complexity and realism, so you'll need to exercise judgment to determine which way is best for your model. We'll start by looking at simple ASCII data and bitmapped image files, then we'll take a tour of NetLogo's GIS capabilities and use raster and vector files.

Using Flat ASCII Data

The simplest way to add spatial realism to your model is to populate the patches in your model with an ASCII file containing a single data value for each of the spatial locations of interest. For example, imagine a small model of say 5x5 cells, where each cell contains a single value that represents one of three possible land cover types. The ASCII data file, here named "land-cover.txt", could look like this:

1 1 2 2 2
1 2 2 2 3
2 2 2 3 3
2 2 3 3 3
2 3 3 1 1

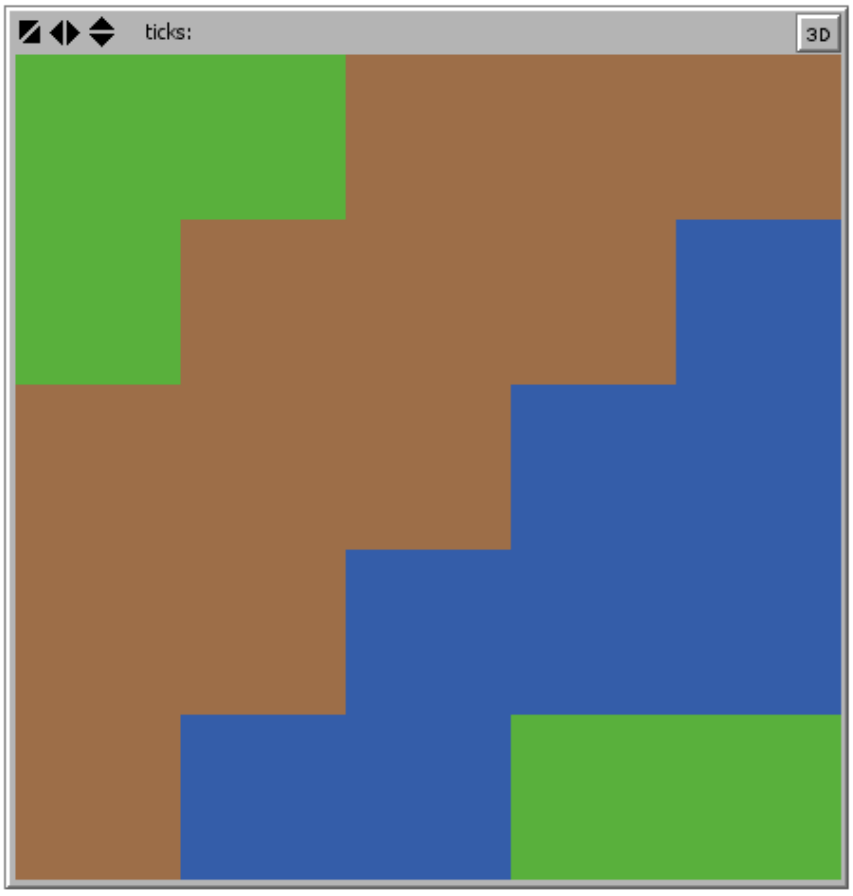
The data file needs to reside in the directory in which you started up NetLogo, or else you need to provide an operating-system specific path indicator for your data file. In the example that follows, we assume that the data file resides in the same folder as the NetLogo executable for simplicity.

To get going, we need to declare that each patch has a land-cover-type variable to store the data, then we need some code to manage reading the file. We read the data file into NetLogo and assign the data values to each patch. Here's a code snippet that does the trick.

```
;
; each patch needs a land-cover-type
;
patches-own [
     land-cover-type
]

;
; this code reads in the data file and assigns the
; value to each patch, then color codes the patch
; based on the value just read in
;
to setup-patches
   file-open "land-cover.txt"
     foreach sort patches [
     ask ?
     [
          ; read a data element into a patch variable
           set land-cover-type file-read
```

```
          ; color the patch accordingly
          if (land-cover-type = 1) [
               set pcolor green
          ]
          if (land-cover-type = 2) [
               set pcolor brown
          ]
          if (land-cover-type = 3) [
               set pcolor blue
          ]
     ]
  ]
  file-close
end
```

The result is a patch landscape initialized based on our ASCII values.

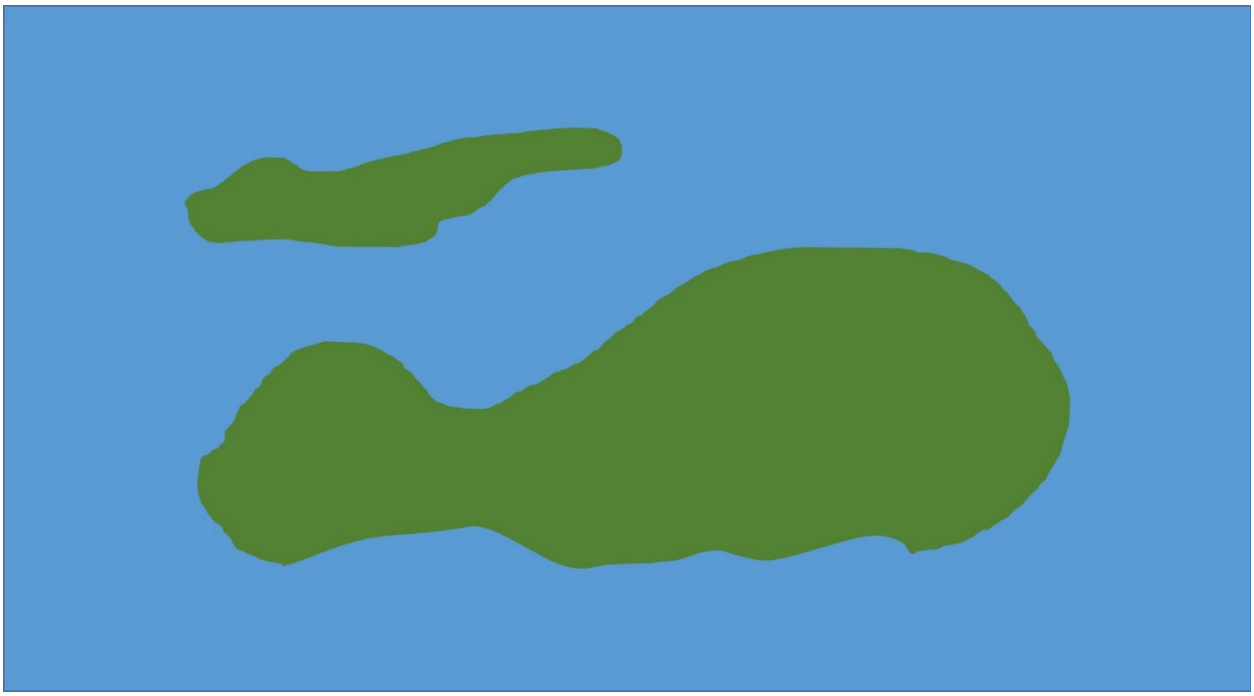


One more thing to note: for this example, we were assuming that the NetLogo patch space was 5x5 cells in total area, but the default NetLogo size is typically 33x33. To be safe, it's best to go into the Settings tab on the Interface and set the model size to match the data set size before you read in the data file.

Using Image File Data

Another technique you can try is using an image file as the basis for your patch landscape. NetLogo allows you to import image file data and map the pixel values read from the file into data values for a patch set. This makes use of the NetLogo extension **bitmap**, which does most of the heavy lifting for accessing binary image files. The bitmap extension supports several popular binary file formats including BMP, PNG, and JPEG. They all work similarly, so we'll just use PNG for the examples to follow.

As an example, let's create a stylized landscape containing some islands located in an ocean. For this demonstration, we've used a standard office presentation software package to draw the picture and save it in an image file format. You don't have to use an office tool to do this: any graphical editor tool that can create binary image files will work fine. Draw an image, and save it as a PNG file format. Note that the number of pixels is important: the original image below is sized at 1620x891 pixels. Here's our image that's been saved as a PNG file named "islands.png".

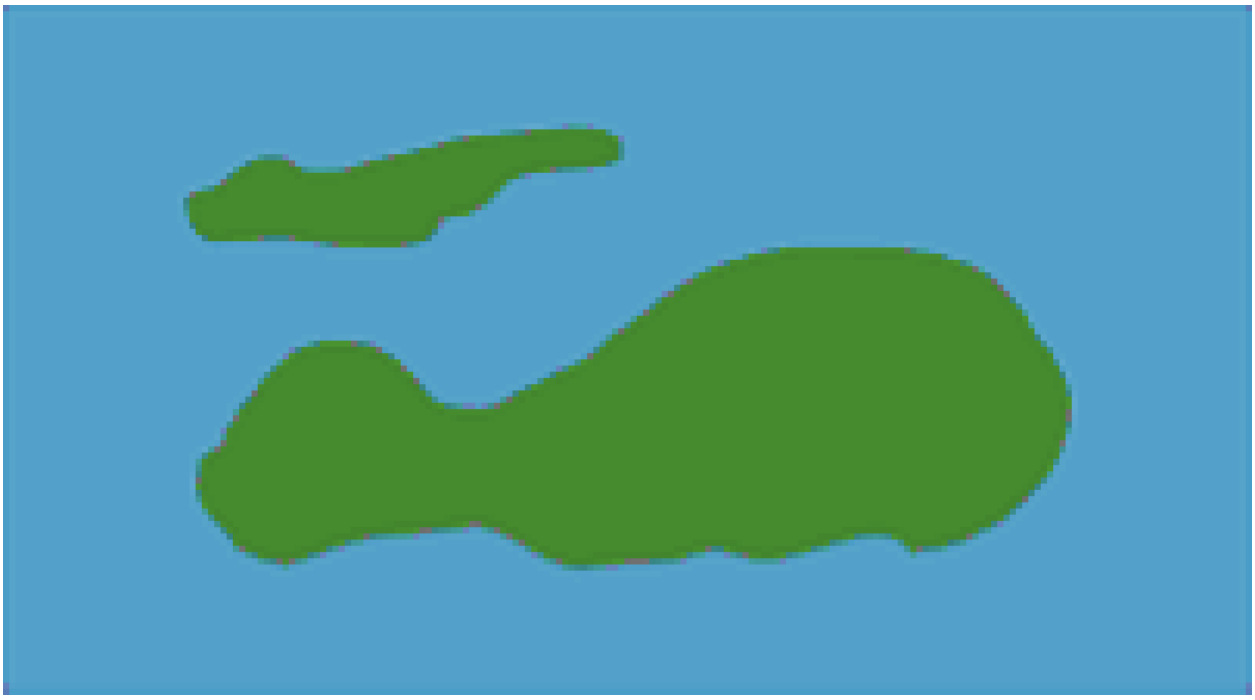


Here's some code to import the PNG file into a NetLogo patch set.

```
;
; the first line of code needs to list any extensions being used
;
extensions [ bitmap ]

; read-png
;
; read in a PNG formatted image file, load the pixels into the
NetLogo
; patch space
;
```

```
to setup-patches-PNG
  ; clear everything to start
  clear-all

  ;
  ; read in the bitmap PNG into a data set
  let data-set bitmap:import "islands.png"

  ;
  ; print out some debug info about the PNG file
  ;
  let data-width bitmap:width data-set
  let data-height bitmap:height data-set

  print (list "image width  = " data-width)
  print (list "image height = " data-height)

  ;
  ; scale the NetLogo world to match the PNG data
  ;
  resize-world 0 data-width 0 data-height

  ;
  ; make a smaller patch size so everything fits on the screen
  ;
  set-patch-size 4

  ;
  ; assign the pcolors based on the PNG data values
  ;
  bitmap:copy-to-pcolors data-set true
end
```

Our example has created a landscape with 1620x891 = 1443420 patches, because when the binary data gets imported into NetLogo each pixel becomes a patch. This means that the number of patches in your model can be very large very quickly, and consequently performance can take a big hit. If you can get away with less detail, you should resize the image using an image editing tool so that it has fewer pixels. Here's what it looks like in NetLogo when you've imported a reduced size image.



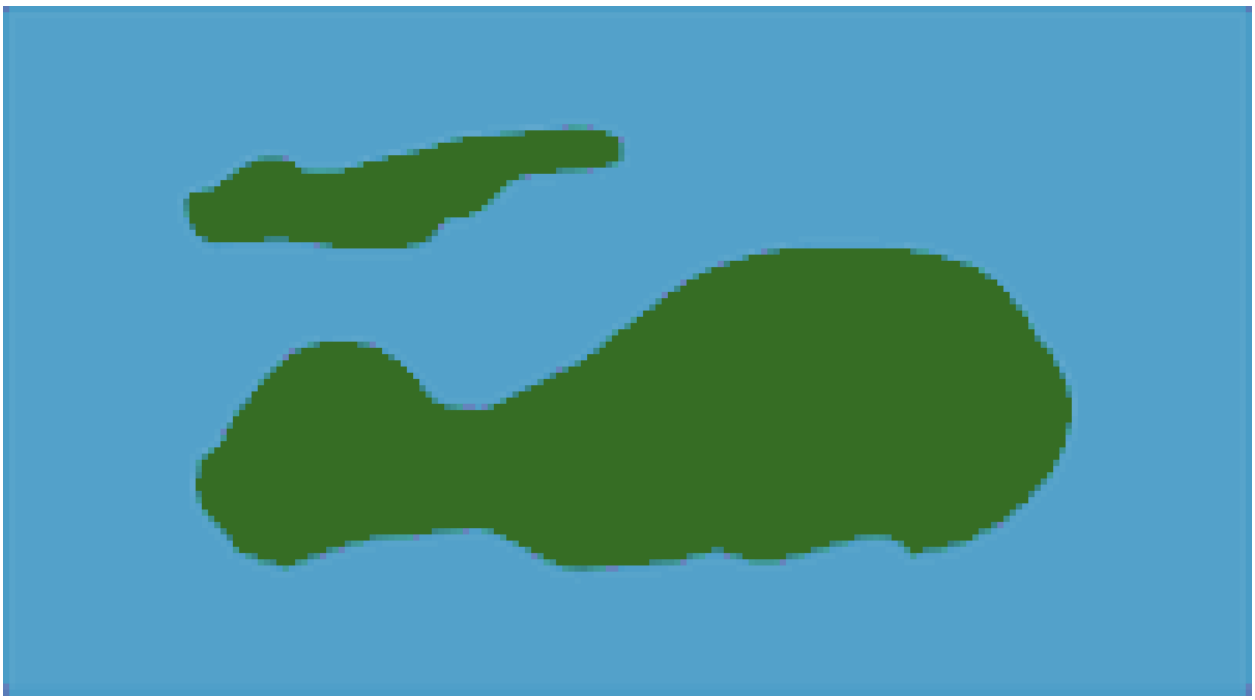
If you look really carefully, you'll see that the process of reducing the number of pixels has resulted in some distortion of the original image. In particular, the edges of the shapes have been "smoothed", which results in some pixels being shaded different colors than in the original image. If you intend to make use of the patch color values as a proxy for land use, for example, you'll need to deal with the issue, otherwise your agents will get confused about what to do with the various slightly different values of the patches.

One solution is to cluster similar valued patch color values into bins, which reduces the overall number of different patch color values. Here's some code to demonstrate the idea.

```
; cluster-pcolors
;
; analyze the patch colors, consolidate
; into a smaller set of patch colors
;
to cluster-pcolors
  let color-list sort [pcolor] of patches
  let i 0
  let value 0
  let delta-list []
  while [ i < (length color-list) - 1 ]
  [
    set value (item (i) color-list) - (item (i + 1) color-list)
    set delta-list (lput value delta-list)
    set i (i + 1)
  ]
```

```
  ;
  ; now find the spikes, adjust patch colors into bins
  ;
  set i 0
  let new-color 0
  let n 0
  while [ i < (length delta-list) ] [
    ifelse abs (item i delta-list) > 10.0
    [
      print (list "DEBUG: spike occurs at location " i )
      set new-color round (new-color / n)
      ask patches with [(pcolor - new-color) < 10.0] [
        set pcolor new-color
      ]
    ]
    [
      set new-color (new-color + (item i color-list))
      set n (n + 1)
    ]

    set i (i + 1)
  ]
end
```

This results in a cleaner image with fewer distinct patch color values, although the color scheme is slightly different from the original.

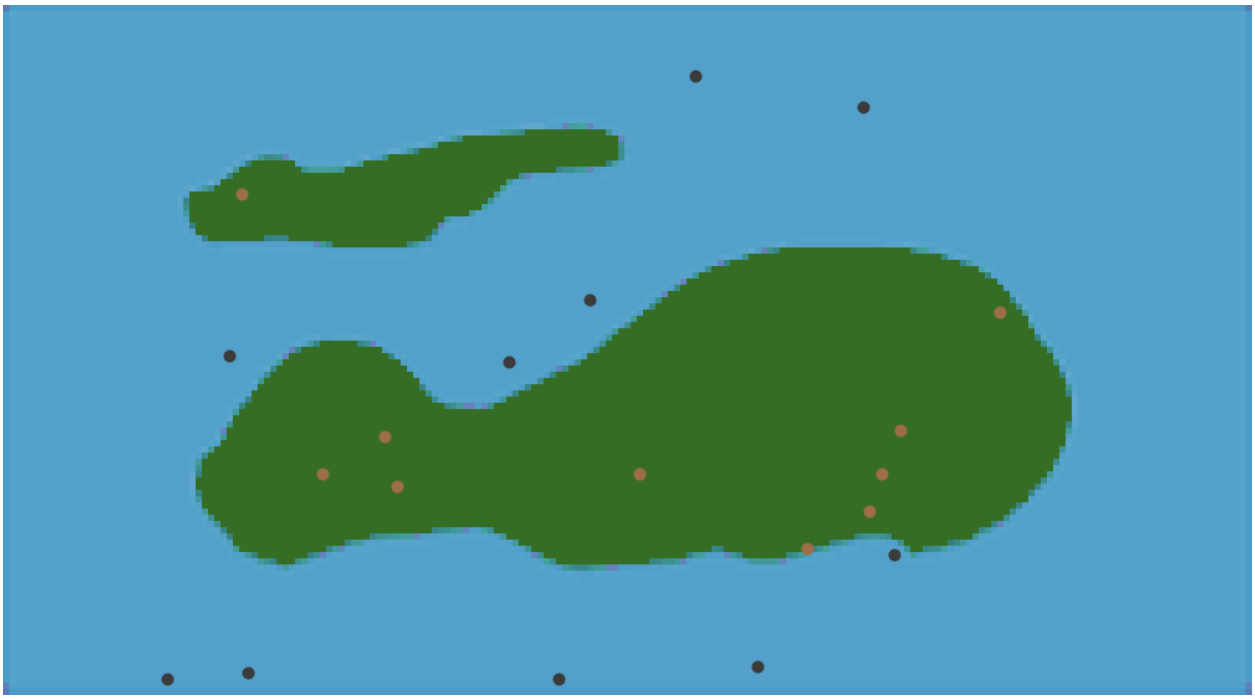


Next, let's make use of our patch set by introducing two classes of agents. One class lives on land, and another class lives in the ocean. We need to locate our agents in the appropriate venue, and we'll decide where they go by examining the patch pcolor value. By using the built-in NetLogo inspect-patch tool, we find out that the blue ocean patches have a pcolor of about 95.9, and the green land patches have a pcolor of about 53.0. We'll ignore the patches at the coastline, as they have include some intermediate values in between, and we don't really need them for this example.

Here's some code to create a set of agents and locate them on land and sea patches.

```
; setup-agents
;
; creates agents, puts them on the land and sea
;
; note that we had to run cluster-pcolor
; and we had to manually determine what the patch
; colors were for land and sea.
;
to setup-agents
  let land-color 53.0
  let ocean-color 95.9

  create-turtles 10 [
    set shape "circle"
    set color brown
    set size 2
    let spot one-of patches with [(pcolor = land-color)
     and (not any? turtles-here) ]
    if (not (spot = nobody)) [
      setxy [pxcor] of spot [pycor] of spot
    ]
  ]

  create-turtles 10 [
    set shape "circle"
    set color gray - 3
    set size 2
    let spot one-of patches with [(pcolor = ocean-color)
     and (not any? turtles-here) ]
    if (not (spot = nobody)) [
      setxy [pxcor] of spot [pycor] of spot
    ]
  ]
end
```

This results in a set of agents and a model that looks something like this. The brown dots show our land-based agents, and the dark gray dots show our sea-based agents.



Hopefully this has given you some ideas about how to work with bitmap images for your models. Now let's turn our attention to real geospatial data sets using raster and vector files.

Raster Data

In the world of Geographic Information Systems, the two main types of data are raster and vector. Raster data consists of tables of measurements, usually automatically collected from aircraft or satellites by scientific instruments. We'll discuss vector data in the next section. A raster data file is conceptually the same as what we've discussed earlier, in that we have a "picture" of some geographic area of interest. However, instead of simply making up values using a color palate as we did in the islands example, the data actually have some kind of real-world interpretation, such as land cover type, elevation, temperature, ice coverage, vegetation, hydrology, wave-height, and so forth. The digital values in the file represent coded numeric quantities from the sensing platform.

In order to get a feel for how to use raster data, let's load in some real raster data and make a small example NetLogo model.

Point your browser at http://www.maine.gov/megis/catalog/ , scroll to the Imagery, Basemaps and Landcover section, and download the Landcover - MELCD 2004 (MELCD) data set. This data set is a land cover assessment of the US state of Maine from 2004. The data set is delivered as a ZIP file, so you'll need to unpack it to a directory somewhere.

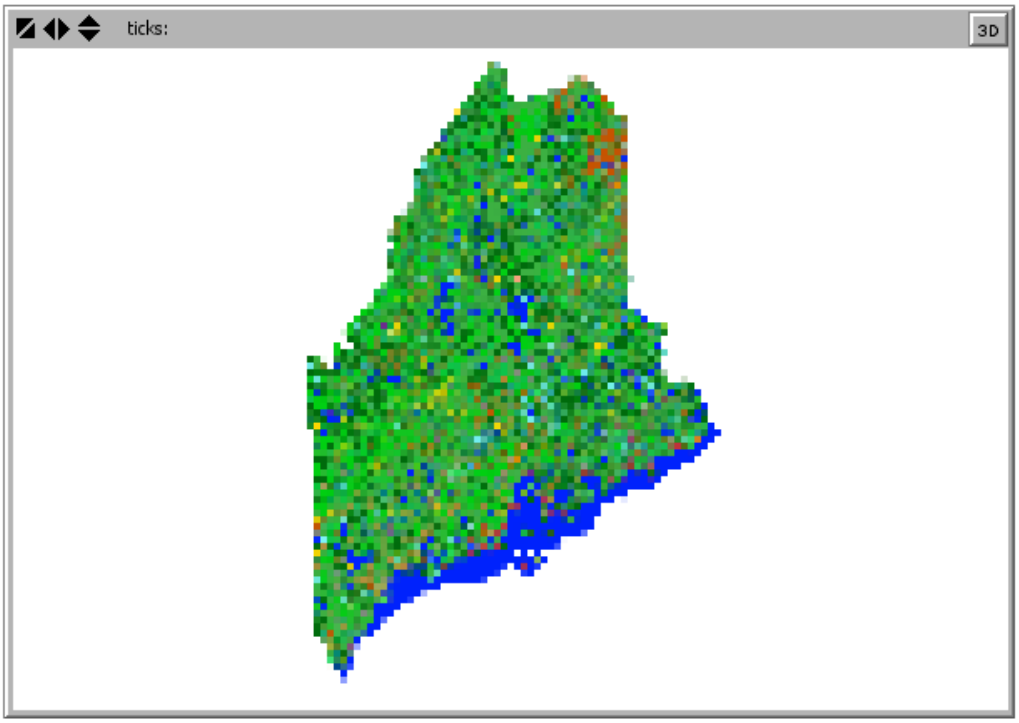
The data set includes several supporting technical files, explaining how the data was collected and how to interpret the data set. We want to import the image of the Maine land use into a NetLogo model.

Right off the bat, we've got a problem – the image file is shipped as a Tagged Interchange File Format (TIFF) file, and that's not one of the NetLogo supported bitmap file types. In addition, the file is fairly large, at over 250MB. We'll need to convert the file from TIFF format into something that NetLogo can read. When the file size is fairly small, one option is to use an office presentation tool like OpenOffice or similar, read in the TIFF file, then "save as" using a bitmap format like PNG, JPEG, or BMP. In this case, however, the file is too big to use this approach, so we'll need to read the file into a real GIS and then export the raster as an image file. I use the open-source QuantumGIS tool, but other products may work as well. In QuantumGIS, read in the TIF raster file, then under File-Save as Image, save the file as a PNG.

Now that we've got a descent sized PNG file, we can load it into a NetLogo model. Here's some code to read in a PNG file and set up our patches with data.

```
to setup-patches
  let bmap bitmap:import
     (word "GIS Data Sets/Maine Land Cover Raster
2004/melcd.png")
  let width bitmap:width bmap
  let height bitmap:height bmap
  print (list "debug: image width = " width ", height = "
height)
  let k 16
  resize-world
     (-1 * (width / k)) (1 * (width / k))
     (-1 * (height / k)) (1 * (height / k))
  set-patch-size 4
  bitmap:copy-to-pcolors bmap false
end
```

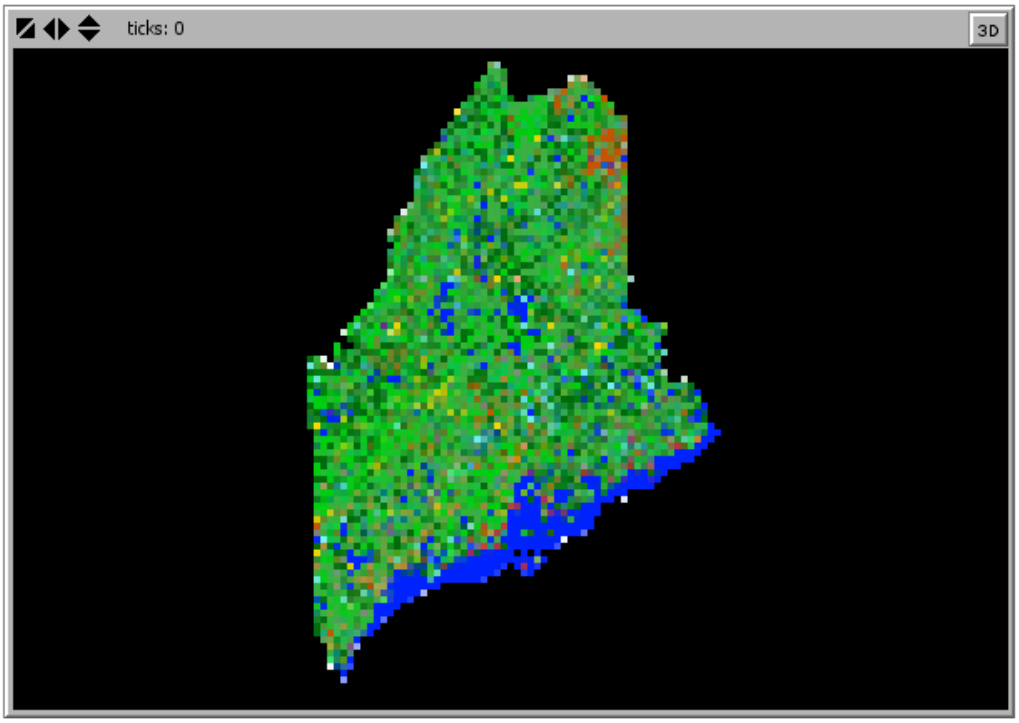
The bitmap:import reads a data file into a NetLogo internal data structure – note that the code is looking for data files in a subdirectory named "GIS Data Sets" which is under the location where the NetLogo model is running. The declarations for width and height are used to display a message to the console, and also to resize the world so that the patch boundaries are more manageable. We call set-patch-size to make the patch sizes smaller than the default which reduces the total amount of screen real estate needed. Finally, the bitmap:copy-to-pcolors does a mapping from the data in the NetLogo internal bitmap data structure to the colors on the patch landscape. Running "setup-patches" from the observer prompt will produce a NetLogo interface that looks something like this.



So far, so good – we've got some data loaded into the NetLogo patch space. As a practical matter, however, we've got some issues. First, there is a lot of unused white space surrounding the patches of interest. Also, we have a lot of different land use types in the original data, which has resulted in a bunch of different colors for patches. Let's deal with these issues one at a time. We can ignore the bounding white space by telling NetLogo that any patch with a pcolor of white (defined as RGB code 255 255 255) is outside our scope. Here's some code to declare a flag within each patch that determines if it is in scope, and some code to ignore all the patches that are out of scope and shade them black.

```
patches-own [
     in-scope?
]

to ignore-out-of-scope
  ask patches [
    ifelse ( pcolor = [ 255 255 255 ] ) [
      set in-scope? false
      set pcolor [ 0 0 0 ]
    ]
    [
      set in-scope? true
    ]
  ]
end
```

Running this code gives us a new map that looks like this, and also sets the in-scope? flag for the peripheral patches to false so we can ignore them.



Now let's deal with the multiple land use categories. From the original data ZIP file, we can consult the documentation and determine that there are 26 categories of land use in the TIFF file. When we converted the TIFF file into a PNG, we should have preserved the variety in 26 different shades of color. In turn, our patches should now have 26 different colors, each representing a distinct type of land use.

Unfortunately, if we carefully examine the contents of our patch data set, we have lots more than 26 shades of colors represented. For example, two cells may look like they have the same shade of green, but they actually have slightly different RGB code values. It's virtually impossible for the eye to distinguish among these fine shadings, but the computer will consider different RGB values as distinct, no matter how close they might appear visually. What we'd like to have is a way to cluster patches that are similar in color together, adjusting their RGB values to a small set of standardized RGB codes, then use this smaller set as a proxy for land use category.

This process is called raster reclassification in the GIS community, and one option available to us is to make use of a real GIS package. We would import the PNG file, do the raster reclassification, save the updated PNG file, then use the new PNG as our data file. In the open-source QuantumGIS package, for example, this process can be accomplished by importing the PNG into the workspace, and updating the properties of the raster map layer using the Style tab, changing the Rendering type to "Singleband psuedocolor", generating a new color map using

one of the many built-in color map schemes, and specifying say 5 equal interval classification bins. Here's an example of the output from a raster reclassification using 5 equal interval bins in QuantumGIS. Notice that the image has far fewer colors than the original PNG, indicating that pixels with similar values have been reclassified together.

Another option is to write some software in NetLogo that does raster reclassification. For this example we'll use the popular K-Means clustering approach from the computer science machine learning community. For this method, the data is portioned into K distinct bins, and then each data point is systematically compared to each of the centroids of the K bins, and assigned to the bin to which it is closest. When all the data points have been assigned, then the centroid for each of the K bins is recomputed by examining all the points in the bin. The process repeats until all the points have been classified to some arbitrary degree of satisfaction. Here's some code to do a K-Means clustering for our raster file image data.

```
;
; analyze-land-use
;
; method to assess the pcolor values in a data
; set and classify them into K bins
;
to analyze-land-use
  let category-list []
  let category-count []
```

```
  let i 0
  ask patches [
    set ptype []
    ifelse (is-list? pcolor) [
      set ptype pcolor
    ]
    [
      set ptype (list pcolor)
    ]

    ifelse member? ptype category-list [
      set i 0
      while [ not ((item i category-list) = ptype) ] [
        set i (i + 1)
      ]

      let t item i category-count
      set t (t + 1)
      set category-count (replace-item i category-count t)
    ]
    [
      set category-list lput ptype category-list
      set category-count lput 1 category-count
    ]
  ]

  ;
  ; report results
  ;
  print (list "total number of patches    = "
     count patches with [in-scope? = true] )
  print (list "total number of categories = "
     length category-list)

  ;
  ; k-means cluster
  ;
  let num-categories 10
  print (list "clustering patches into "
     num-categories " categories..." )
  k-means-cluster-patches num-categories

end

;
; euclidian-distance
;
```

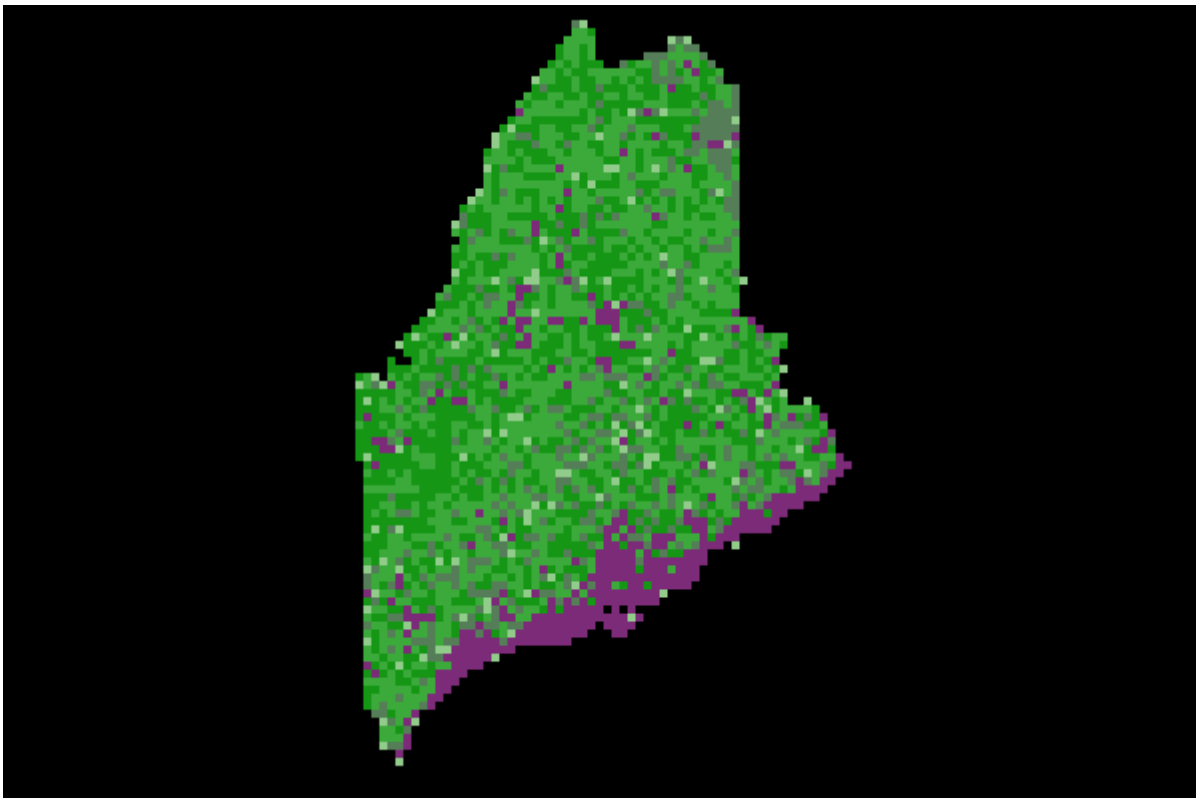
```
; compute euclidian distance between vectors p and q
;
to-report euclidian-distance[ p q ]
  let value 0
  let delta[]
  let n length p
  let i 0
  while [ i < n ] [
    let dv 1.0 * ((item i p) - (item i q)) ^ 2.0
    set delta (fput dv delta)
    set i (i + 1)
  ]
  set value sqrt (sum delta)
  report value
end

;
; vector addition
;
; elementwise add vector p and q
;
to-report vector-addition [ p q ]
  let value []
  let delta []
  let n length p
  let i 0
  while [ i < n ] [
    let dv (item i p) + (item i q)
    set delta (fput dv delta)
    set i (i + 1)
  ]
  set value delta
  report value
end

;
; vector-multiplication
;
; multiply vector q by the scalar K
;
to-report vector-multiplication [ K q ]
  let value []
  let delta []
  let n length q
  let i 0
  while [ i < n ] [
    let dv K * (item i q)
```

```
    set delta (lput dv delta)
    set i (i + 1)
  ]
  set value delta
  report value
end

;
; k means cluster for patches
;
to k-means-cluster-patches [ n ]
  ;
  ; randomly generate starting locations
  ; in the 3-space [0..255] [0..255] [0..255]
  ;
  ; make sure they are unique
  ;
  let centroid-list []
  let ok? false
  let i 0
  while [ i < n ] [
    set ok? false
    while [not ok?] [
      let tmp (list (random 255) (random 255) (random 255))
      ifelse member? tmp centroid-list [
        set ok? false
      ]
      [
        set ok? true
        set centroid-list (fput tmp centroid-list)
      ]
    ]
    set i (i + 1)
  ]

  ;
  ; initially all patches have centroid #1
  ;
  ask patches with [in-scope? = true] [
    set centroid-id 0
  ]

  ;
  ; loop thru and compute the k-means clusters
  ;
  let distance-vector []
  let dv 0
```

```
let num-loops 10
while [ num-loops > 0 ] [

  ;
  ; ask all the patches to find which
  ; centroid they are closest to
  ;
  ask patches with [in-scope? = true] [
    ;
    ; reset the distance vector
    ;
    set distance-vector []

    ;
    ; compute distance from each patch to each centroid
    ;
    foreach centroid-list [
      set dv euclidian-distance ? ptype
      set distance-vector (lput dv distance-vector)
    ]

    ;
    ; find index of smallest distance observed
    ;
    let min-index position

    ;
    ; assign this to our patch
    ;
    set centroid-id min-index
  ]

  ;
  ; update all the centroids based on latest set of patches
  ;
  set i 0
  let tmp [0 0 0]
  while [ i < n ] [
    set tmp [0 0 0]
    let myset patches with
        [(in-scope? = true) and (centroid-id = i)]
    let mylist (sort myset)
    if (length mylist > 0) [
      foreach mylist [
        set tmp (vector-addition tmp ([ptype] of ?))
      ]
      set tmp (vector-multiplication
```

```
          (1.0 / (length mylist)) tmp)
        set centroid-list (replace-item i centroid-list tmp)
      ]

      set i (i + 1)
    ]

    set num-loops (num-loops - 1)
  ]

  ;
  ; report results
  ;
  print (list "debug: final centroid list = ")
  set i 0
  foreach centroid-list [
    let r precision (item 0 ?) 2
    let g precision (item 1 ?) 2
    let b precision (item 2 ?) 2
    print (list r " " g " " b " " ", ")
  ]
  print " "

  ;
  ; now update the patches
  ;
  ask patches with [in-scope? = true]
  [
    let tmp item ([centroid-id] of self) centroid-list
    ifelse (is-list? tmp and (length tmp = 3)) [
      set pcolor tmp
    ]
    [
      set pcolor [0 0 0]
    ]
  ]
end
```

Here's an example of applying these methods to our original PNG data file, with K=5 classes.

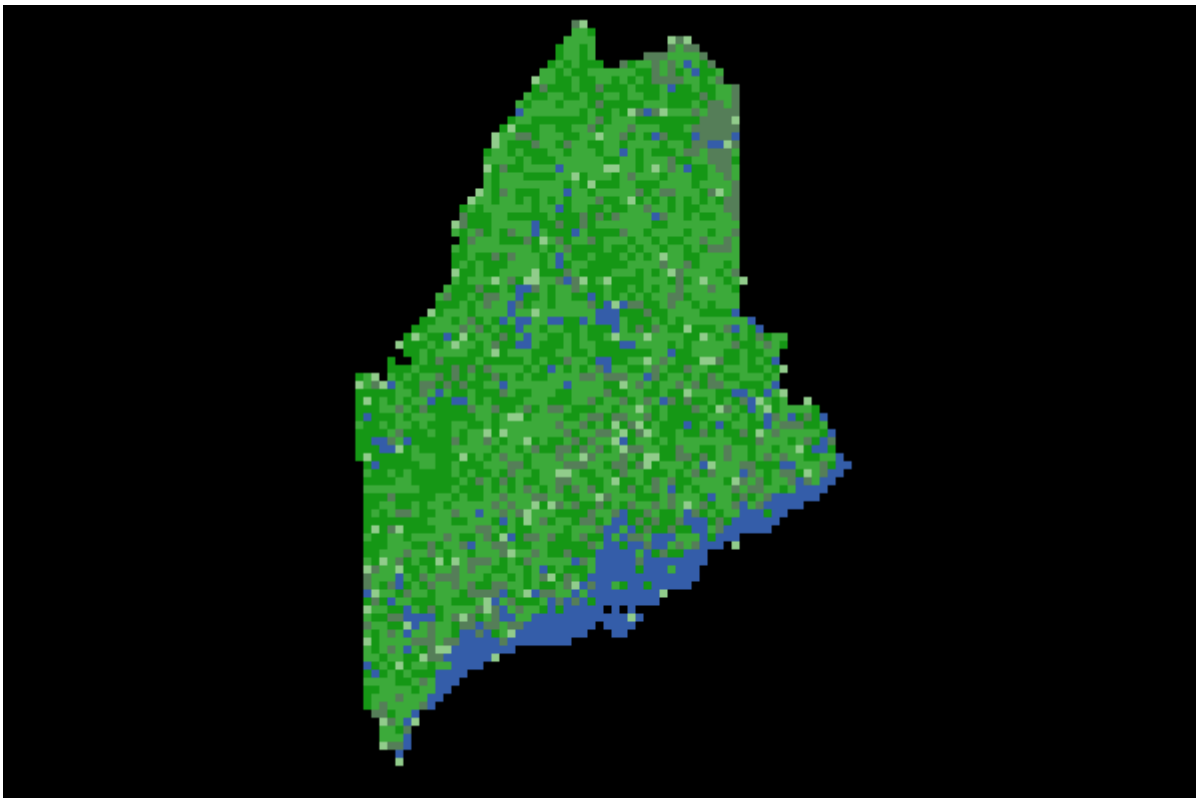


The results are very similar to the output of the GIS raster reclassification approach, although visually the NetLogo screen rendering is a bit more "pixelated" due to lower resolution than the GIS example presented earlier. The underlying categorization of the patches is conceptually the same however: in both methods, we have a map representation with a relatively small number of land use categories.

If you don't like the color scheme that resulted from the K-Means clustering results, you can of course adjust the pcolor values interactively in NetLogo. Let's say we really want to convert the Atlantic Ocean from purple to something more oceanic like blue, for example. We need to determine what the NetLogo color for the patches depicted as purple is by right clicking on a patch and reading the pcolor value. For my example, the shade of purple on the image was represented by the RGB triple = [124.20215633423182 43.7789757412399 121.65498652291106]. So to convert this shade of purple to blue, we need to go to the Observer prompt and type in the following command (you may want to cut/paste the RGB values instead of typing them manually!)

```
Observer> ask patches with [pcolor = [124.20215633423182
43.7789757412399 121.65498652291106]] [set pcolor blue]
```

This yields a more aesthetically pleasing image. You can adjust the other colors accordingly.



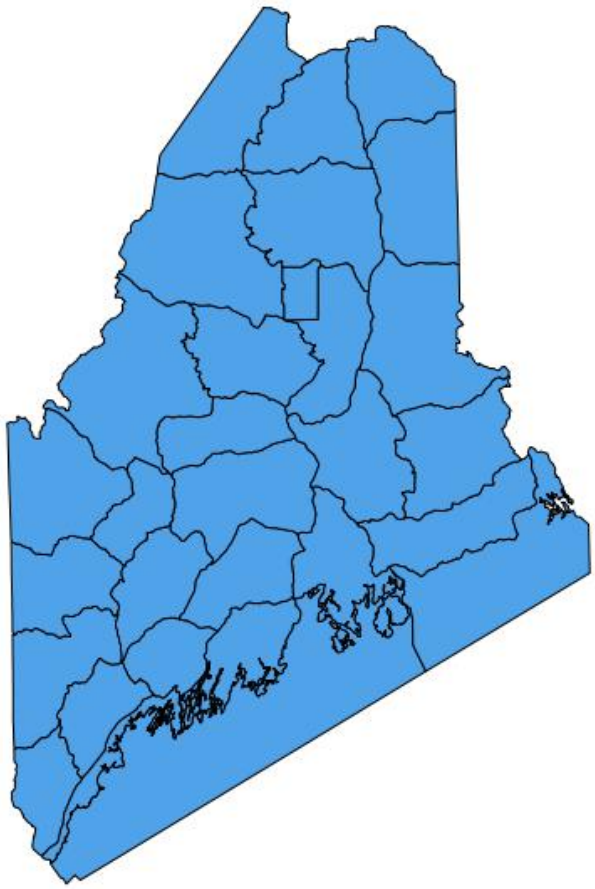
Hopefully this discussion has shown you how to get started using raster data files in NetLogo. The example may seem a bit complex, but it's representative of real-world data that you're likely to encounter when doing analysis in NetLogo.

Vector Data

The second major type of GIS data used in the community is vector data. Vector data files contain a mathematical model of a geospatial area of interest. The most popular format for vector data is the shapefile, which is a data file containing spatial data and metadata. Shapefiles can contain point, line, or polygon data, and often a GIS project includes multiple shapefiles of each of these types. NetLogo provides the capability to read in shapefiles directly, allowing you to import mapping data in your models. NetLogo allows you to import more than one shapefile, although to keep things simple for this example we'll just use a single polygon file.

Let's continue with our example from Maine, this time importing a vector data file. Point your browser back to http://www.maine.gov/megis/catalog/ , scroll to the Biologic and ecologic/ Environment and conservation section, and download the Wildlife Management Districts (2/7/2012) data set. This data set defines the boundaries of wildlife management districts for the US state of Maine as of 2012. As before, the data set is delivered as a ZIP file, so you'll need to unpack it to a directory somewhere.

To get a feel for what the data looks like, you can load it into a real GIS such as QuantumGIS and visualize the shapefile. Here's what it looks like.



We can clearly see the various wildlife management district boundaries, as well as some arbitrary regions of coastal waterways. If we examine the properties for this layer, we'll note that the data is using a North American Datum 1983 (NAD 83) datum and a Universal Transverse Mercator (UTM) zone 19N projection. Don't worry if you're unfamiliar with these GIS terms, as we'll introduce them when they touch NetLogo GIS capabilities as needed. Now let's load this vector file into a NetLogo model and see what we get. Here's some code to read in the vector data file and overlay the data onto the NetLogo patch space.

```
globals [
  maine-dataset  ; NetLogo container to hold vector data
]

to setup-vector-data
  ; restore default sizes
  resize-world -25 25 -25 25
  set-patch-size 10

  ; load in a polygon shape file for the map boundary
  gis:load-coordinate-system
     (word "GIS Data Sets/Maine Wildlife Mgmt
Vector/wildlife_mgmt_districts.prj")
```

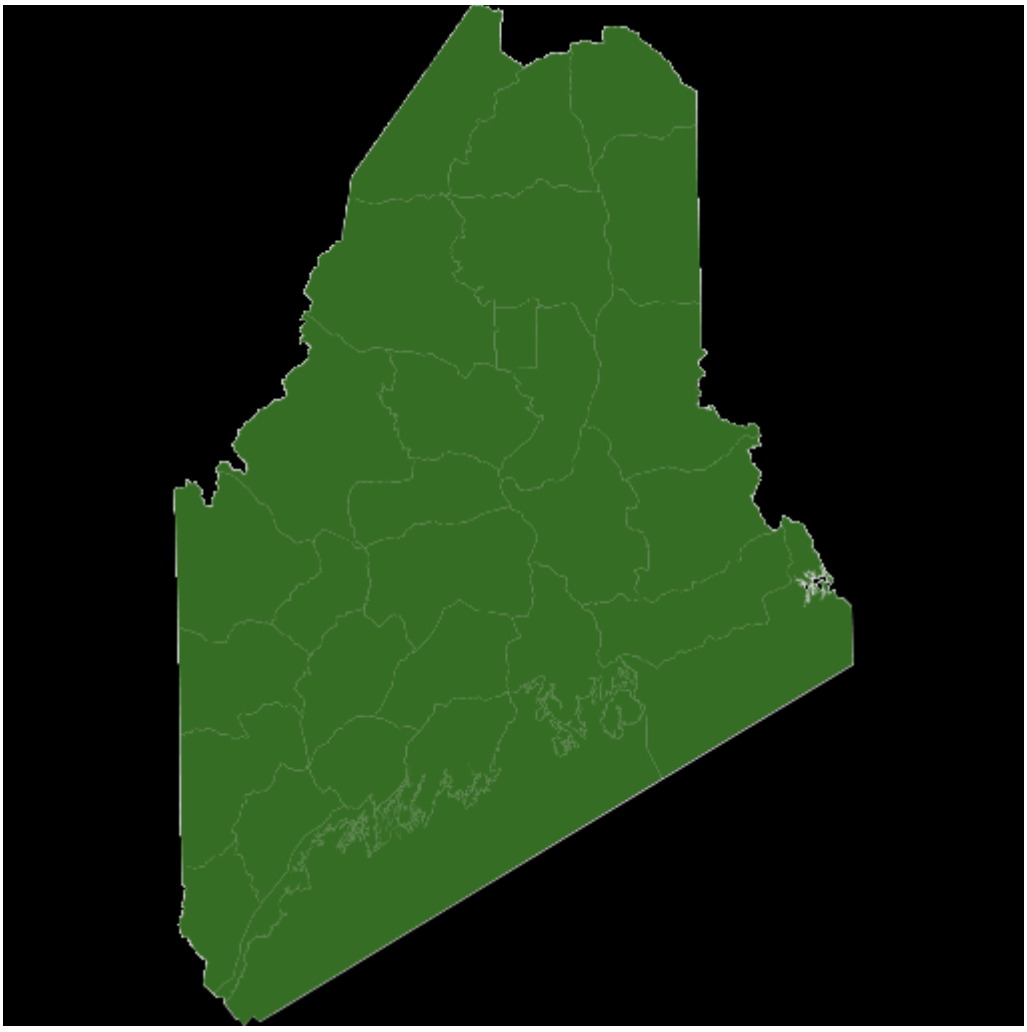
```
  set maine-dataset gis:load-dataset
"GIS Data Sets/Maine Wildlife Mgmt
Vector/wildlife_mgmt_districts.shp"

  ;; color in the district borders
  gis:set-drawing-color gray + 3
  let pen-width 1
  gis:draw maine-dataset pen-width

  ;; shade the land areas
  gis:set-drawing-color green - 2
  gis:fill maine-dataset 2
end
```

The first thing we do is resize the world to some reasonable defaults, here set to -25 to 25 in both the vertical and horizontal dimensions. We set the patch size to 10 pixels, again to give a reasonable default size. Of course these can be adjusted to your liking.

The next set of methods are used to setup the coordinate system and read in the map data. The method `gis:load-coordinate-system` reads in a PRJ file, which is a standard GIS data file containing the geographic projection information for a shapefile. In this case, the projection is UTM, so the PRJ file contains the details of the UTM projection. Next we read in the data from the vector file using the `gis:load-dataset` method. Note that we read the data into a NetLogo data structure, here called "maine-dataset".

We now need to draw the data onto our NetLogo patch space. The `gis:set-drawing-color` method determines which color to use when drawing borders, and the `gis:draw` method actually draws the borders for us. Similarly, the `gis:set-drawing-color` method is used again to determine which color to fill our polygons with, and the `gis:fill` method fills the polygons.

The result of this yields a NetLogo patch space that looks like this.



Notice that by comparison to the raster data set, the resulting map image has a lot more fine detail especially around the borders.

To use our data, let's add some code to create an agent that wanders around the various wildlife management districts in Maine and reports information about each district. Here's some code to move an agent to the centroid of each polygon, then report all the metadata fields about that polygon to the console.

```
extensions [ gis ]

globals [
  maine-dataset    ; this is the NetLogo container to hold
vector data

  shape-type       ; what type of shapefile did we read
  property-names   ; list of properties from the shapefile
  feature-list     ; list of features (VectorFeatures) from
shapefile
```

```
  vertex-lists     ; list of lists of vertices

]

to setup-vector-data
  ; restore default sizes
  resize-world -25 25 -25 25
  set-patch-size 10

  ; load in a polygon shape file for the map boundary
  gis:load-coordinate-system (word "GIS Data Sets/Maine Wildlife
Mgmt Vector/wildlife_mgmt_districts.prj")
  set maine-dataset gis:load-dataset "GIS Data Sets/Maine
Wildlife Mgmt Vector/wildlife_mgmt_districts.shp"

  ;; color in the state borders
;
  gis:set-drawing-color gray + 3
  let pen-width 1
  gis:draw maine-dataset pen-width

  ;; shade the land areas
  gis:set-drawing-color green - 2
  gis:fill maine-dataset 2

  ;
  ; now set up the properties for the main shapefile
  ;
  set shape-type gis:shape-type-of maine-dataset
  set property-names gis:property-names maine-dataset
  set feature-list gis:feature-list-of maine-dataset
  set vertex-lists gis:vertex-lists-of item 0 feature-list
end


;
; visit
;
; turtles visit features at random and print out
; metadata about the feature where the are located
;
to visit
  ask turtles [

    ;
    ; pick a GIS feature at random
    ;
```

```
    let chosen random (length feature-list)
    let my-feature (item chosen feature-list)

    ;
    ; translate the centroid of the chosen feature from
    ; GIS space to NetLogo patch space, then move there
    ;
    let new-location gis:location-of (gis:centroid-of my-
feature)
    let new-x (item 0 new-location)
    let new-y (item 1 new-location)
    setxy new-x new-y

    ;
    ; print out data on where we are now located
    ;
    foreach property-names [
      print (list ? " = " gis:property-value my-feature ?)
    ]
    print " "

  ]
End
```

Let's walk thru the code to make sure you follow what's going on. First, as before, we need to include the GIS extensions package, which is accomplished by the "extensions [gis]" statement.

As a convenience, we've added a few more global variables to hold data about the Vector data file: shape-type, property-names, feature-list, and vertex-lists. These are NetLogo GIS extension structures that we'll used later on when we traverse the landscape. Let's go thru them one at a time to see what they do.

The shape-type will indicate what type of data our shapefile holds. Recall that there are only three basic shapefile data types (point, line, and polygon). Point files contain data represented as a series of points, such as city or towns. Line data contain linear data, such as roads, rivers, power lines, and railroads. Polygon data are sets of polygons, such as county, state, or national administrative boundaries.

The property-names will hold a list of strings indicating the metadata property names associated with our shapefile. Every shapefile can have a set of properties and attributes, which you can think of as a small database table with data about the features in the shapefile. For example, if we have a polygon shapefile, then the property-names will contain the "column" names for the table, and each "row" of the table corresponds to a polygon in the shapefile.

The feature-list holds a list of lists, where the inner lists are the property:value pairs for each polygon. Finally, the vertex-lists holds a list of lists as well, but the inner lists in this case are the X:Y positions of the polygon points used to define the line segments for each polygon.

Notice that in order to use our new global data structures, we need to make a few calls to initialize them. This is done in some new code we've added at the end of the setup-globals method, which is listed below for convenience.

```
  ;
  ; now set up the properties for the main shapefile
  ;
  set shape-type gis:shape-type-of maine-dataset
  set property-names gis:property-names maine-dataset
  set feature-list gis:feature-list-of maine-dataset
  set vertex-lists gis:vertex-lists-of item 0 feature-list
```

The interesting part comes up in a new method called "visit". Here, we go thru our agents, and randomly locate a wildlife management district, relocate ourselves there, and then print some data about that district. We start by randomly selecting a feature from the data set.

```
    ;
    ; pick a GIS feature at random
    ;
    let chosen random (length feature-list)
    let my-feature (item chosen feature-list)
```

The local variable my-feature is the chosen district. Next, we need to locate our agent at the location in NetLogo space that corresponds to the location in GIS space of the centroid of the chosen district. The call to gis:centroid-of my-feature will return the coordinates of the centroid for my-feature. These coordinates are in GIS space, so we need to translate into NetLogo space because our agent doesn't operate in GIS space. The call to gis:location-of will return a 2-element list in NetLogo space corresponding to the point in GIS space supplied as an input. Since it's a 2-element list, we need to pick out the first and second items, which will correspond to our X and Y locations respectively, before assigning our agent to the new location on the patch space. Having done all this, we can now set our agent to the NetLogo X-Y location that corresponds to the centroid of the district of interest using the setxy method. Work thru the code below to make sure you're following.

```
    ;
    ; translate the centroid of the chosen feature from
    ; GIS space to NetLogo patch space, then move there
    ;
    let new-location gis:location-of (gis:centroid-of my-
feature)
    let new-x (item 0 new-location)
    let new-y (item 1 new-location)
    setxy new-x new-y
```

The next interesting thing to do is to poll the system and extract metadata about the district we just moved to. Recall that we previously set up a global data structure called property-names to

hold all the labels for the metadata, so we can use this data structure here to get the various labels. Using a foreach iterator, we can ask to obtain the value for this metadata field using the gis:property-value method, passing the current feature (my-feature) and the desired property label (using the "?" operator). This will iterate over all the metadata fields, and ask our current location for the record associated with that label. Here's the code to do this.

```
;
; print out data on where we are now located
;
foreach property-names [
  print (list ? " = " gis:property-value my-feature ?)
]
print " "
```

The resulting output looks something like this.

```
[IDENTIFIER  =  29]
[OBJECTID  =  13]
[MOOSEMGMT  =  Recreation Mgmt Area]
[HUNTSEASON  =  ]
[SHAPE_LENG  =  699.3680135]
[AREASQMI  =  0.00844356]
[SHAPE_AREA  =  21868.8440261]
[SHAPE_LEN  =  699.367983483]
```

Summary

We've covered a lot of ground in this chapter, but to be fair we have only just scratched the surface on what NetLogo can do with GIS data. We've introduced a simple model of spatial data using ASCII files, and we covered using standard image files like PNG and JPG as sources of spatial data. We covered using real GIS data such as raster and vector data sets, and introduced how to employ various methods from the NetLogo GIS extensions package. Hopefully this will get you interested in using real GIS data for your NetLogo models.

9. Creating NetLogo Extensions

Extensions are user-defined functions and procedures, written in Java and "compiled" to a JAR file that are accessible from within a NetLogo program. NetLogo includes a variety of extensions, including arrays, hash tables, matrixes, MIDI sound generation, robotics/NetLogoLab interfaces, performance profiling, GIS capabilities, bitmap data structure I/O, and QuickTime movie support.

This chapter goes over how to create your own NetLogo extensions. We'll assume that you have a successfully installed Eclipse Java development system, and that you already know enough about Java development to create Java classes for applications. Just to be clear, this is an advanced topic, but if you've got experience working in a Java, Python, or Scala development environment, you should be able to pick it up quickly.

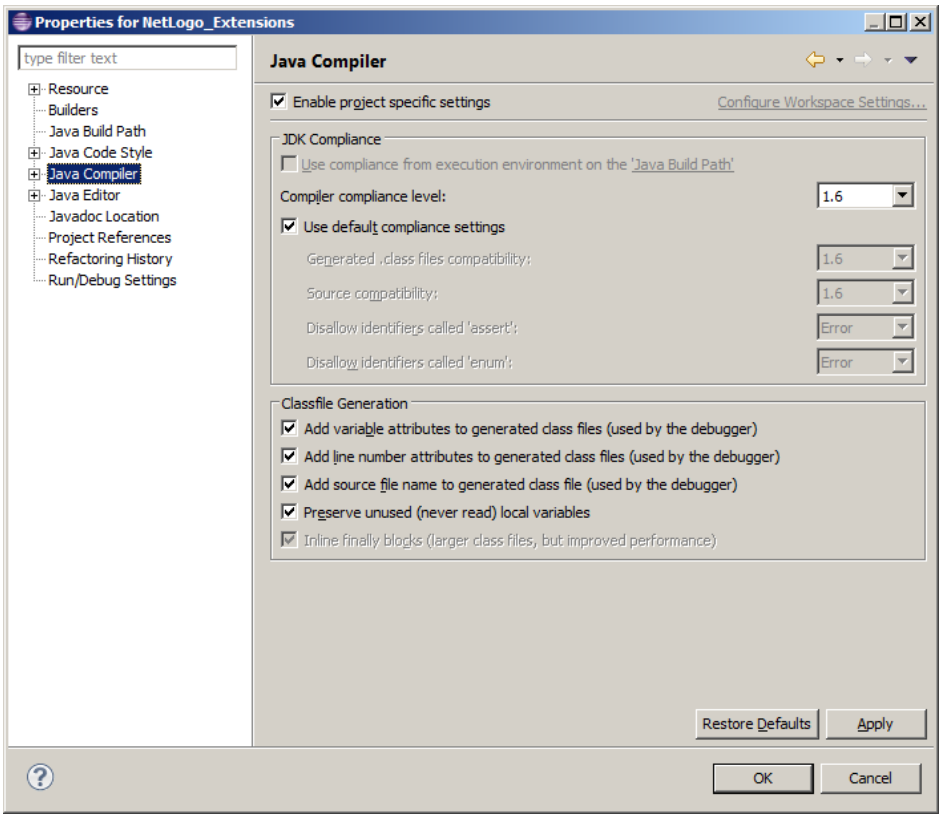
Important Prerequisite

You'll need to determine the version of the Java compiler that was used to create your copy of NetLogo. To determine this, start up NetLogo, go under the Help -> "About NetLogo 5.0.1" menu item, and choose "System" tab. Listed on the Java HotSpot™ Server VM row you'll find the version number. For NetLogo 5.0.1, my Java environment is:

```
Java HotSpot(TM) Server VM 1.6.0_30(Sun Microsystems Inc.; 1.6.0_30-b12)
```

The string “1.6.0_30-b12” means my NetLogo was compiled using Java version 1.6.0_30, so I need to use Java 1.6. This **does not** mean I have to uninstall a later version of Java such as 1.7, but rather that the Java development environment I use needs to generate code compliant at the Java 1.6 release level.

Set your Eclipse project so that it will generate code according to the same Java build number as the NetLogo build. To do this, go under Project -> Properties, click to "Enable project specific settings", and use the pulldown to select the appropriate version of Java in the "Compiler compliance level" field. For this example, we'll use Java 1.6.



This step is critical: if you don't setup the Java version in the Eclipse environment to match the Java version in the NetLogo environment, your extensions won't work.

Building your Extension

Within a NetLogo extension, you can have one or more new capabilities implemented – they will look just like build-in methods in NetLogo, but you'll access them with the extension prefix. For example, if we are creating a statistics package called 'stats' and we want to implement the Anderson-Darling test for normality, our new capability could be accessed from within NetLogo as "stats:anderson-darling".

The first thing to decide is if your new capability will be a command or a reporter. A command does some computation and generates side effects on NetLogo objects. A reporter can do computation and can generate side effects as well, but reporters always explicitly return a value. For example, you might write a command called "compute-agent-value" that performs some calculation on a field within an agent and updates the value of the agent. You could also write a reporter called "convert-miles-to-kilometers" that converts distance measurements from English to Metric units and reports a value.

NetLogo requires you to create your classes as implementations of the Java interface org.nlogo.api.Command or org.nlogo.api.Reporter classes. When creating your code, you'll need to figure out if you are implementing a command or a reporter, and write your code as an extension of the appropriate Java base class.

NetLogo requires you to create a ClassManager, as an extension of the class the Java interface org.nlogo.api.ClassManager. Whichever type of extension you write, you will have to implement a ClassManager to manage the new extension.

NetLogo requires you to deploy your new extension as a Java Archive (JAR) file. To build a JAR file, you need to create a manifest file, which is a small text file used during the creation of a Java JAR file used to specify important metadata. As an example, let's say you are writing a new NetLogo package called "stats", and the Java file implementing the NetLogo class manager is named "StatsClassManager.java". Your manifest file should look like this. Let's go through each of the items in the manifest file.

```
Manifest-Version: 1.0
Extension-Name: stats
Class-Manager: stats.StatsClassManager
NetLogo-Extension-API-Version: 5.0
```

The Manifest-Version indicates which version number to use; 1.0 is fine.

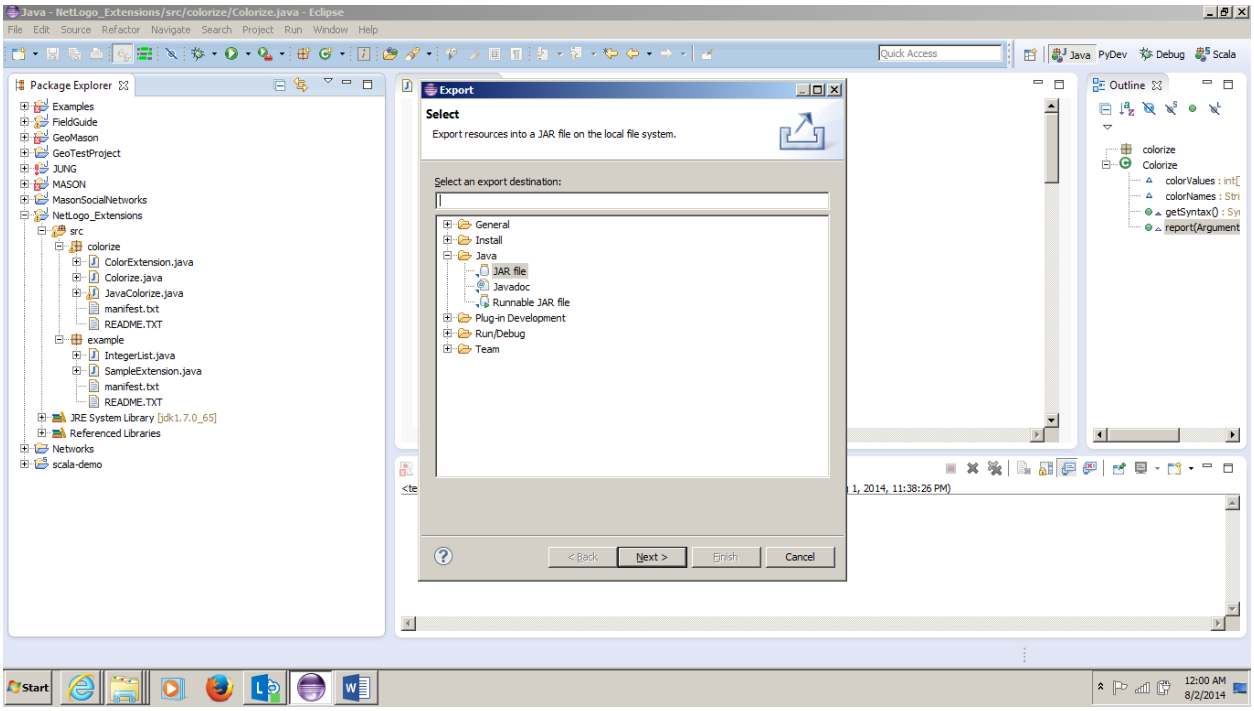
The Extension-Name is the name you plan to use in NetLogo when accessing your new capabilities. This must also be the name of the JAR file itself that you will eventually deploy in your NetLogo installation directory.

The Class-Manager is the fully package name qualified name of the class that will be doing the class manager function. Note that in this example, the StatClassManager resides in the "stats" package, so the fully qualified name is "stats.StatsClassManager". It's important to make sure that the package name sequence is correct, otherwise NetLogo will be confused when loading your extension JAR file.

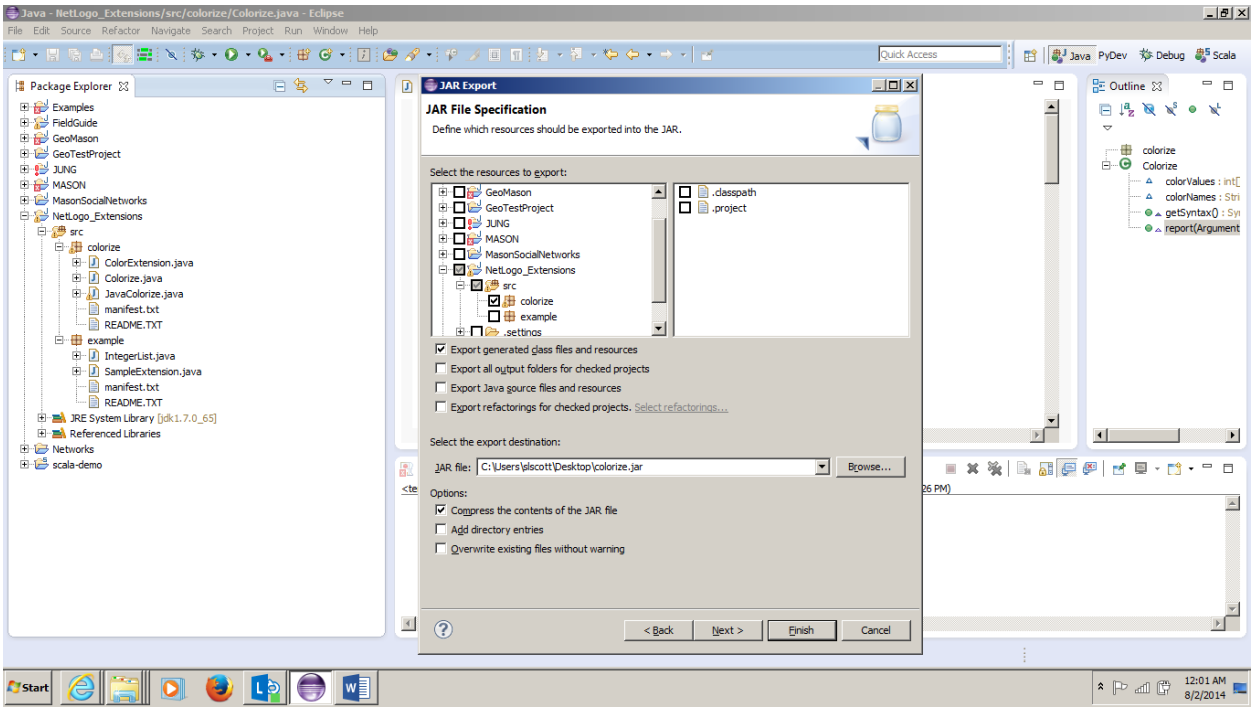
Finally, the NetLogo-Extension-API-Version tells which version of the NetLogo Extension API we're using: 5.0 is fine for this example. Note that the manifest file must include a <cr><lf> at the end of the last line of text.

Compiling your Extension

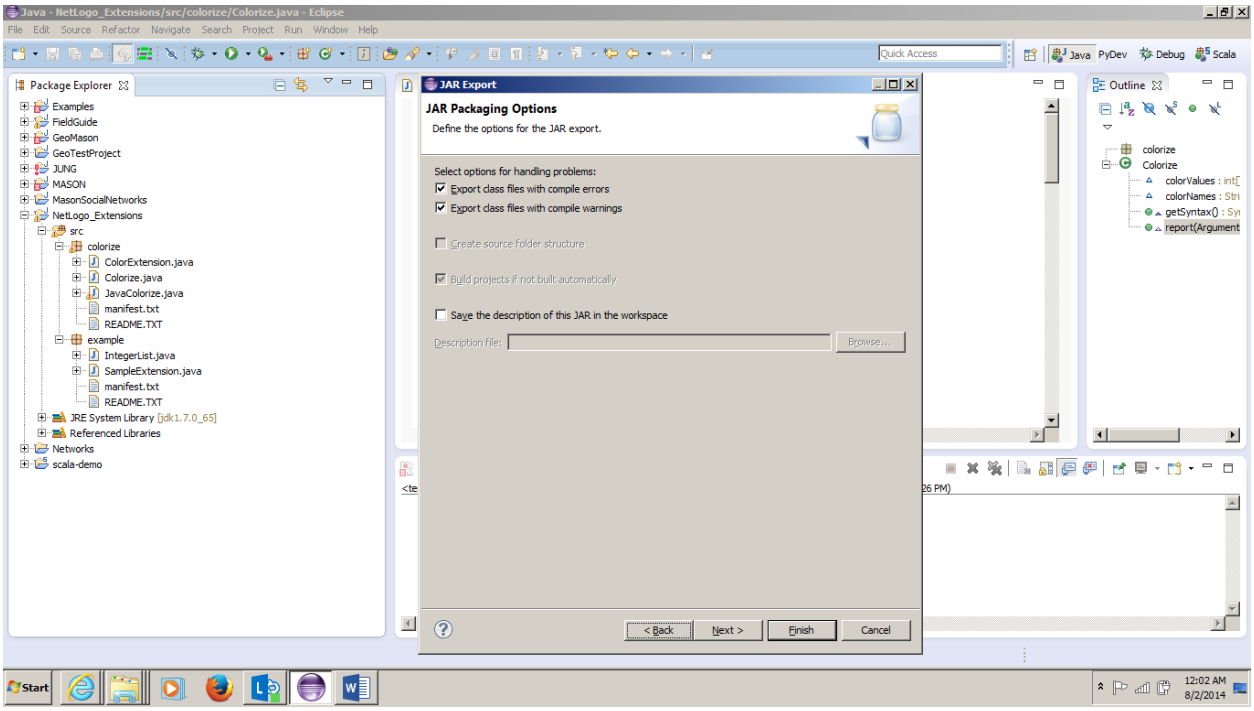
NetLogo requires you to build a JAR file. The easiest way to do this in Eclipse is to go under File-Export, and select "Jar file" option.



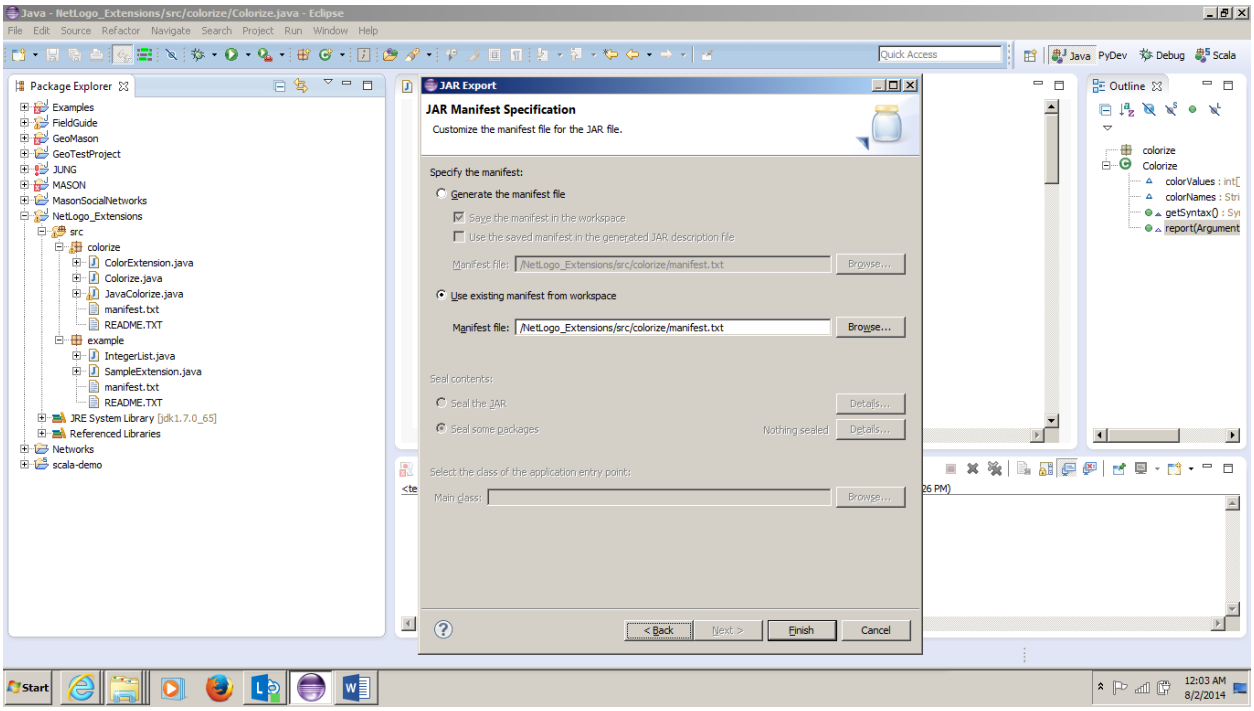
At the Next button, advance to the JAR file specification dialog, and click thru to the appropriate directory for your source code. Click the Next button.



At the JAR Packaging Options, use the default settings. Click on Next.



Finally, make sure to select the "Use existing manifest from workspace" radio button. Click Finish to generate your JAR file.



Installing your Extension

Once the JAR file is built, place it in the extensions folder of your NetLogo installation. Note that the folder name and the JAR file name need to be the same.

Using your Extension

To use your extension, add the keyword **`extensions [ extension-name ]`** as the first line of code in your NetLogo model. You can of course include other extensions, including built-in NetLogo extensions or other user-developed extensions. To invoke your extension, use the **`extension::method-name`** syntax. Here's an example of the first few lines of an example NetLogo program that uses a couple of extension packages: the "GIS" package is built-in to NetLogo, and the "stats" package is something you've built or obtained from another NetLogo coder.

```
; my-netlogo.logo
;
; code for my model
;
extensions [ gis stats ] ; includes the built-in extension for
                           gis
                           ; and my stats extension as well
```

Example: Adding W3C Standard Color Codes for patches and agents

For this example, let's build a Java extension that allows us to specify colors for either patches or agents using the extended color name scheme defined by W3C for HTML and CSS (see http://www.w3.org/TR/css3-color/). The idea for this extension is that we'd like to be able to set the color for an object using the well-known text name for the color instead of trying to remember the hex code or the RGB values. The W3C standard specifies lots of color names over and above the basic set that is built in to the Java Color class, so we could add a lot more interesting colors to our NetLogo agents and patches in our models.

To do this, we need a Java class to map the color names to RGB values, and some basic string matching software to look up our requested colors. Here's the data we want to use as the lookup table. I've abbreviated the table to just the first few rows and the last few rows – you'll get the idea.

Color Name	HEX CODE	Red	Green	Blue
White	#FFFFFF	255	255	255
maroon	#800000	128	0	0
Red	#FF0000	255	0	0
purple	#800080	128	0	128
fuchsia	#FF00FF	255	0	255
Green	#008000	0	128	0
Lime	#00FF00	0	255	0

Color Name	HEX CODE	Red	Green	Blue
Olive	#808000	128	128	0
yellow	#FFFF00	255	255	0
Navy	#000080	0	0	128
Blue	#0000FF	0	0	255
Teal	#008080	0	128	128
Aqua	#00FFFF	0	255	255
aliceblue	#F0F8FF	240	248	255
antiquewhite	#FAEBD7	250	235	215
Aqua	#00FFFF	0	255	255
aquamarine	#7FFFD4	127	255	212
Azure	#F0FFFF	240	255	255
Beige	#F5F5DC	245	245	220
bisque	#FFE4C4	255	228	196
Black	#000000	0	0	0
blanchedalmond	#FFEBCD	255	235	205
Blue	#0000FF	0	0	255
blueviolet	#8A2BE2	138	43	226
brown	#A52A2A	165	42	42
burlywood	#DEB887	222	184	135
cadetblue	#5F9EA0	95	158	160
chartreuse	#7FFF00	127	255	0
chocolate	#D2691E	210	105	30
Coral	#FF7F50	255	127	80
… a bunch more				
teal	#008080	0	128	128
thistle	#D8BFD8	216	191	216
tomato	#FF6347	255	99	71
turquoise	#40E0D0	64	224	208
violet	#EE82EE	238	130	238
wheat	#F5DEB3	245	222	179
white	#FFFFFF	255	255	255
whitesmoke	#F5F5F5	245	245	245
yellow	#FFFF00	255	255	0
yellowgreen	#9ACD32	154	205	50

We'll need to write three pieces of software: a Java class to manage the actual methods and implementation, another Java class to manage the interface, and a Java manifest file to provide

the compiler with instructions for how to build the Java Archive (JAR) file. We'll go through these one at a time.

Let's start with the actual code to do the work: in this case, we need a Java class to accept a user-supplied string specifying a color name as input, and return a list of integers representing the RGB values for the specified color. For example, let's say we want to color all the patches in our model "yellowgreen". Instead of having to remember that yellowgreen consists of the RGB triple [154, 205, 50], we'd like to just specify "yellowgreen" and let the software look it up for us. Here's some Java code to do this operation.

```
/**
 * JavaColorize
 *
 * this class is an example of a Java extension to the NetLogo
 * agent-based modeling environment.  The code in this class
 * is designed to extend the base functionality of NetLogo
 * by allowing the user to specify the full set of W3C color
 * names for use in coloring NetLogo patches and agents.
 *
 * This class must be compiled into a JAR file and the resulting
 * JAR file must be deployed into the NetLogo extensions folder
 * before the feature can be used in NetLogo code.
 *
 * usage in NetLogo code:
 *
 * ask patches [ set pcolor colorize:lookup-rgb "CadetBlue" ]
 * ask turtles [ set color colorize:lookup-rgb "azure" ]
 *
 */
package colorize;

import java.awt.Color;
import java.util.ArrayList;
import java.util.Locale;

import org.nlogo.api.Argument;
import org.nlogo.api.Context;
import org.nlogo.api.DefaultReporter;
import org.nlogo.api.ExtensionException;
import org.nlogo.api.LogoException;
import org.nlogo.api.LogoListBuilder;
import org.nlogo.api.Syntax;

public class JavaColorize extends DefaultReporter {
	//
	// here are the W3C standard color definitions
	//
	Object[][] JColorList = {
			{ "aliceblue",0xf0f8ff,240,248,255} ,
			{ "antiquewhite",0xfaebd7,250,235,215} ,
			{ "aqua",0x00ffff,0,255,255} ,
			{ "aquamarine",0x7fffd4,127,255,212} ,
			{ "azure",0xf0ffff,240,255,255} ,
```

```
{ "beige",0xf5f5dc,245,245,220} ,
{ "bisque",0xffe4c4,255,228,196} ,
{ "black",0x000000,0,0,0} ,
{ "blanchedalmond",0xffebcd,255,235,205} ,
{ "blue",0x0000ff,0,0,255} ,
{ "blueviolet",0x8a2be2,138,43,226} ,
{ "brown",0xa52a2a,165,42,42} ,
{ "burlywood",0xdeb887,222,184,135} ,
{ "cadetblue",0x5f9ea0,95,158,160} ,
{ "chartreuse",0x7fff00,127,255,0} ,
{ "chocolate",0xd2691e,210,105,30} ,
{ "coral",0xff7f50,255,127,80} ,
{ "cornflowerblue",0x6495ed,100,149,237} ,
{ "cornsilk",0xfff8dc,255,248,220} ,
{ "crimson",0xdc143c,220,20,60} ,
{ "cyan",0x00ffff,0,255,255} ,
{ "darkblue",0x00008b,0,0,139} ,
{ "darkcyan",0x008b8b,0,139,139} ,
{ "darkgoldenrod",0xb8860b,184,134,11} ,
{ "darkgray",0xa9a9a9,169,169,169} ,
{ "darkgreen",0x006400,0,100,0} ,
{ "darkgrey",0xa9a9a9,169,169,169} ,
{ "darkkhaki",0xbdb76b,189,183,107} ,
{ "darkmagenta",0x8b008b,139,0,139} ,
{ "darkolivegreen",0x556b2f,85,107,47} ,
{ "darkorange",0xff8c00,255,140,0} ,
{ "darkorchid",0x9932cc,153,50,204} ,
{ "darkred",0x8b0000,139,0,0} ,
{ "darksalmon",0xe9967a,233,150,122} ,
{ "darkseagreen",0x8fbc8f,143,188,143} ,
{ "darkslateblue",0x483d8b,72,61,139} ,
{ "darkslategray",0x2f4f4f,47,79,79} ,
{ "darkslategrey",0x2f4f4f,47,79,79} ,
{ "darkturquoise",0x00ced1,0,206,209} ,
{ "darkviolet",0x9400d3,148,0,211} ,
{ "deeppink",0xff1493,255,20,147} ,
{ "deepskyblue",0x00bfff,0,191,255} ,
{ "dimgray",0x696969,105,105,105} ,
{ "dimgrey",0x696969,105,105,105} ,
{ "dodgerblue",0x1e90ff,30,144,255} ,
{ "firebrick",0xb22222,178,34,34} ,
{ "floralwhite",0xfffaf0,255,250,240} ,
{ "forestgreen",0x228b22,34,139,34} ,
{ "fuchsia",0xff00ff,255,0,255} ,
{ "gainsboro",0xdcdcdc,220,220,220} ,
{ "ghostwhite",0xf8f8ff,248,248,255} ,
{ "gold",0xffd700,255,215,0} ,
{ "goldenrod",0xdaa520,218,165,32} ,
{ "gray",0x808080,128,128,128} ,
{ "green",0x008000,0,128,0} ,
{ "greenyellow",0xadff2f,173,255,47} ,
{ "grey",0x808080,128,128,128} ,
{ "honeydew",0xf0fff0,240,255,240} ,
{ "hotpink",0xff69b4,255,105,180} ,
{ "indianred",0xcd5c5c,205,92,92} ,
```

```
{ "indigo",0x4b0082,75,0,130} ,
{ "ivory",0xfffff0,255,255,240} ,
{ "khaki",0xf0e68c,240,230,140} ,
{ "lavender",0xe6e6fa,230,230,250} ,
{ "lavenderblush",0xfff0f5,255,240,245} ,
{ "lawngreen",0x7cfc00,124,252,0} ,
{ "lemonchiffon",0xfffacd,255,250,205} ,
{ "lightblue",0xadd8e6,173,216,230} ,
{ "lightcoral",0xf08080,240,128,128} ,
{ "lightcyan",0xe0ffff,224,255,255} ,
{ "lightgoldenrodyellow",0xfafad2,250,250,210} ,
{ "lightgray",0xd3d3d3,211,211,211} ,
{ "lightgreen",0x90ee90,144,238,144} ,
{ "lightgrey",0xd3d3d3,211,211,211} ,
{ "lightpink",0xffb6c1,255,182,193} ,
{ "lightsalmon",0xffa07a,255,160,122} ,
{ "lightseagreen",0x20b2aa,32,178,170} ,
{ "lightskyblue",0x87cefa,135,206,250} ,
{ "lightslategray",0x778899,119,136,153} ,
{ "lightslategrey",0x778899,119,136,153} ,
{ "lightsteelblue",0xb0c4de,176,196,222} ,
{ "lightyellow",0xffffe0,255,255,224} ,
{ "lime",0x00ff00,0,255,0} ,
{ "limegreen",0x32cd32,50,205,50} ,
{ "linen",0xfaf0e6,250,240,230} ,
{ "magenta",0xff00ff,255,0,255} ,
{ "maroon",0x800000,128,0,0} ,
{ "mediumaquamarine",0x66cdaa,102,205,170} ,
{ "mediumblue",0x0000cd,0,0,205} ,
{ "mediumorchid",0xba55d3,186,85,211} ,
{ "mediumpurple",0x9370db,147,112,219} ,
{ "mediumseagreen",0x3cb371,60,179,113} ,
{ "mediumslateblue",0x7b68ee,123,104,238} ,
{ "mediumspringgreen",0x00fa9a,0,250,154} ,
{ "mediumturquoise",0x48d1cc,72,209,204} ,
{ "mediumvioletred",0xc71585,199,21,133} ,
{ "midnightblue",0x191970,25,25,112} ,
{ "mintcream",0xf5fffa,245,255,250} ,
{ "mistyrose",0xffe4e1,255,228,225} ,
{ "moccasin",0xffe4b5,255,228,181} ,
{ "navajowhite",0xffdead,255,222,173} ,
{ "navy",0x000080,0,0,128} ,
{ "oldlace",0xfdf5e6,253,245,230} ,
{ "olive",0x808000,128,128,0} ,
{ "olivedrab",0x6b8e23,107,142,35} ,
{ "orange",0xffa500,255,165,0} ,
{ "orangered",0xff4500,255,69,0} ,
{ "orchid",0xda70d6,218,112,214} ,
{ "palegoldenrod",0xeee8aa,238,232,170} ,
{ "palegreen",0x98fb98,152,251,152} ,
{ "paleturquoise",0xafeeee,175,238,238} ,
{ "palevioletred",0xdb7093,219,112,147} ,
{ "papayawhip",0xffefd5,255,239,213} ,
{ "peachpuff",0xffdab9,255,218,185} ,
{ "peru",0xcd853f,205,133,63} ,
```

```
            { "pink",0xffc0cb,255,192,203} ,
            { "plum",0xdda0dd,221,160,221} ,
            { "powderblue",0xb0e0e6,176,224,230} ,
            { "purple",0x800080,128,0,128} ,
            { "red",0xff0000,255,0,0} ,
            { "rosybrown",0xbc8f8f,188,143,143} ,
            { "royalblue",0x4169e1,65,105,225} ,
            { "saddlebrown",0x8b4513,139,69,19} ,
            { "salmon",0xfa8072,250,128,114} ,
            { "sandybrown",0xf4a460,244,164,96} ,
            { "seagreen",0x2e8b57,46,139,87} ,
            { "seashell",0xfff5ee,255,245,238} ,
            { "sienna",0xa0522d,160,82,45} ,
            { "silver",0xc0c0c0,192,192,192} ,
            { "skyblue",0x87ceeb,135,206,235} ,
            { "slateblue",0x6a5acd,106,90,205} ,
            { "slategray",0x708090,112,128,144} ,
            { "slategrey",0x708090,112,128,144} ,
            { "snow",0xfffafa,255,250,250} ,
            { "springgreen",0x00ff7f,0,255,127} ,
            { "steelblue",0x4682b4,70,130,180} ,
            { "tan",0xd2b48c,210,180,140} ,
            { "teal",0x008080,0,128,128} ,
            { "thistle",0xd8bfd8,216,191,216} ,
            { "tomato",0xff6347,255,99,71} ,
            { "turquoise",0x40e0d0,64,224,208} ,
            { "violet",0xee82ee,238,130,238} ,
            { "wheat",0xf5deb3,245,222,179} ,
            { "white",0xffffff,255,255,255} ,
            { "whitesmoke",0xf5f5f5,245,245,245} ,
            { "yellow",0xffff00,255,255,0} ,
            { "yellowgreen",0x9acd32,154,205,50} };

// JColor()
//
// helper class – used to encapsulate data in the
// color list data structure.
//
class JColor {
       String name;
       int hexvalue;
       int red;
       int green;
       int blue;
}

// lookup()
//
// method to do a linear scan of the color list and
// either return the RGB values if found
// or return RGBs for the color “white” if not found
//
public JColor lookup(String colorName) {

       JColor value = new JColor();
```

```
        value.name = "White";
        value.hexvalue = 0xffffff;
        value.red = 255;
        value.green = 255;
        value.blue = 255;

        boolean found = false;
        int i = 0;
        int numColors = JColorList.length;
        colorName = colorName.toLowerCase();

        while ((i < numColors) && (found == false)) {
                if (JColorList[i][0].equals((String) colorName)) {

                        found = true;
                        value.name = (String) JColorList[i][0];

                        value.hexvalue = (int)
                        Integer.parseInt(JColorList[i][1].toString());

                        value.red = (int)
                        Integer.parseInt(JColorList[i][2].toString());

                        value.green = (int)
                        Integer.parseInt(JColorList[i][3].toString());

                        value.blue = (int)
                        Integer.parseInt(JColorList[i][4].toString());
                }
                else {
                        found = false;
                        i++;
                }
        }
        return(value);
}


// helper methods: retrieve specified components of
// RGB triple
//
public int getRed(String colorName) {
        int value = 0;
        JColor jColor = lookup( colorName );
        value = jColor.red;
        return(value);
}

public int getGreen(String colorName) {
        int value = 0;
        JColor jColor = lookup( colorName );
        value = jColor.green;
        return(value);
}
```

```
public int getBlue(String colorName) {
	int value = 0;
	JColor jColor = lookup( colorName );
	value = jColor.blue;
	return(value);
}


// getSyntax()
//
// this class defines the syntax for the method call
//
// input is a string representing a Java Color name,
// output is the RGB integer values in a list
//
public Syntax getSyntax() {
	return Syntax.reporterSyntax(
			new int[] { Syntax.StringType() },
			Syntax.ListType());
}

// report()
//
// implements the report method
//
public Object report( Argument args[], Context context)
		throws ExtensionException {

	// create a NetLogo list for the result
	LogoListBuilder list = new LogoListBuilder();

	int R, G, B, alpha;
	String selection;

	// fetch the user-supplied string containing the
	// desired color name.
	//
	// use typesafe helper method from
	// org.nlogo.api.Argument to access argument
	//
	try {
		selection = args[0].getString().toLowerCase();
	}
	catch(LogoException e) {
		throw new ExtensionException( e.getMessage() ) ;
	}

	//
	// check to make sure string is not empty
	//
	if (selection.length() == 0) {
		// signals a NetLogo runtime error to the modeler
		throw new ExtensionException
		("error: must supply a nonzero length Java color name");
	}
```

```
            //
            // create a new JavaColorize object
            // to manage the lookup
            //
            JavaColorize jc = new JavaColorize();
            JColor j = jc.lookup(selection);

            //
            // build the return list
            //
            // note that we use Double objects; NetLogo
            // numbers are always Doubles
            //
            list.add(Double.valueOf( j.red ));
            list.add(Double.valueOf( j.green ));
            list.add(Double.valueOf( j.blue ));

            // return the list
            return list.toLogoList();
      }

}
```

Let's walk through the code to see what's going on. After some imports, we start first by declaring that we are extending the DefaultReporter class in NetLogo: this is the class that manages methods that return values to the caller.

```
      public class JavaColorize extends DefaultReporter {
```

Next we declare a data structure to hold the color name strings, their hexadecimal values, and their decimal values for their red, green, and blue components. This is done using a 2-dimensional array of Java Objects, here called JColorList.

```
      //
      // here are the W3C standard color definitions
      //
      Object[][] JColorList = {
                  { "aliceblue",0xf0f8ff,240,248,255} ,
                  { "antiquewhite",0xfaebd7,250,235,215} ,
```

We've declared a small helper class named JColor to hold an instance of a one color extracted from the JColorList table. This is simply a convenience for coding – the attributes of the class map directly to the components of a row in the JColorList table.

```
      class JColor {
            String name;
            int hexvalue;
            int red;
            int green;
            int blue;
```

The real work of the class gets done by the `lookup()` method. Here we're implementing a very basic linear search of the JColorList table, comparing the user specified color string with all the known color strings in the table. If we get a match, then we return the data about that color. If not, however, we return data for a default color of "white". Here's the code for the linear lookup method.

```
public JColor lookup(String colorName) {

            JColor value = new JColor();
            value.name = "White";
            value.hexvalue = 0xffffff;
            value.red = 255;
            value.green = 255;
            value.blue = 255;

            boolean found = false;
            int i = 0;
            int numColors = JColorList.length;
            colorName = colorName.toLowerCase();

            while ((i < numColors) && (found == false)) {
                  if (JColorList[i][0].equals((String) colorName)) {

                        found = true;
                        value.name = (String) JColorList[i][0];

                        value.hexvalue = (int)
                        Integer.parseInt(JColorList[i][1].toString());

                        value.red = (int)
                        Integer.parseInt(JColorList[i][2].toString());

                        value.green = (int)
                        Integer.parseInt(JColorList[i][3].toString());

                        value.blue = (int)
                        Integer.parseInt(JColorList[i][4].toString());
                  }
                  else {
                        found = false;
                        i++;
                  }
            }
            return(value);
      }
```

We've declared three small helper methods to extract the red, green, and blue components of a JColor object. These are pretty straightforward: all they do is peform a lookup, then return the appropriate attribute from the JColor resulting from the lookup. Here's the `getRed()` method as an example.

```
      public int getRed(String colorName) {
            int value = 0;
```

```
            JColor jColor = lookup( colorName );
            value = jColor.red;
            return(value);
      }
```

The next two methods are more complicated and more interesting. These are the required methods for the implementation of the NetLogo DefaultReporter class. The first of these is the `getSyntax()` method. This one returns an instance of Syntax class, which is used to understand how our new method will interact with the rest of the NetLogo environment. In this case, we declare that our method will accept a Syntax.StringType as input, and return a Syntax.ListType as output. When you are designing your NetLogo extensions, this is where you will define the input and output data types.

```
      public Syntax getSyntax() {
            return Syntax.reporterSyntax(
                        new int[] { Syntax.StringType() },
                        Syntax.ListType());
      }
```

The next method is the `report()` method, which implements the reporter functionality in NetLogo. We begin by declaring that our return type will be a list, here implemented as a LogoListBuilder class. The user supplied color name will be assigned to the String selection, which we extract by pulling the args[0] parameter.

After some type checking and error checking, we can now invoke the lookup for the specified color string, which will return a JColor object. Using that JColor object, we can then extract the RGB components, and put each of these as elements of a list. We then return the list to the caller, and the method is complete. Here's the code to do this.

```
public Object report(Argument args[], Context context)
                  throws ExtensionException {

            // create a NetLogo list for the result
            LogoListBuilder list = new LogoListBuilder();

            int R, G, B, alpha;
            String selection;

            // fetch the user-supplied string containing the
            // desired color name.
            //
            // use typesafe helper method from
            // org.nlogo.api.Argument to access argument
            //
            try {
                  selection = args[0].getString().toLowerCase();
            }
            catch(LogoException e) {
                  throw new ExtensionException( e.getMessage() ) ;
            }
```

```
            //
            // check to make sure string is not empty
            //
            if (selection.length() == 0) {
                  // signals a NetLogo runtime error to the modeler
                  throw new ExtensionException
                  ("error: must supply a nonzero length Java color name");
            }

            //
            // create a new JavaColorize object
            // to manage the lookup
            //
            JavaColorize jc = new JavaColorize();
            JColor j = jc.lookup(selection);

            //
            // build the return list
            //
            // note that we use Double objects; NetLogo
            // numbers are always Doubles
            //
            list.add(Double.valueOf( j.red ));
            list.add(Double.valueOf( j.green ));
            list.add(Double.valueOf( j.blue ));

            // return the list
            return list.toLogoList();
      }
```

Now we need to define how our extension will fit into the NetLogo environment using a ClassManager class. This class lets NetLogo interpret our newly created method as a primitive (built-in) command. Fortunately, this one is fairly simple. We begin by declaring a class named ColorExtension as an extension of the DefaultClassManager. We only need to declare a "load" method, which in turn will add the primitive we specify to the primitive manager in NetLogo. Our NetLogo command will be called "lookup-rgb", and it is implemented using an instance of the JavaColorize class we just wrote, so these are the parameters we use for the `addPrimitive()` method. Here's the code to do this.

```
package colorize;
import org.nlogo.api.*;

public class ColorExtension extends DefaultClassManager {

      public void load(PrimitiveManager primitiveManager) {
            //
            // lookup-rgb definition here
            //
            primitiveManager.addPrimitive(
                        "lookup-rgb",
                        new JavaColorize());
      }
}
```

Finally, we need to connect everything together with a manifest file so the Java compiler knows how to build our new code. The manifest version should be set to 1.0, as shown. The Extension-Name is the NetLogo name that we'll use in our NetLogo models to reference our new capabilities. Here we're using the name "colorize", so that goes on line 2. Next we need to tell the compiler which class is our Class-Manager. As previously introduced, this is done by our ColorExtension class. Note that we have to precede the name of this class by the fully qualified package name "colorize". Thus, we have to specify "colorize.ColorExtension" rather than just the class name by itself. This can be a difficult "gotcha" to debug, as everything compiles fine but the code will raise run time exceptions because it can't find the main class file. Finally, the NetLogo-Extension-API-Version is set to 5.0, as this is the current version for NetLogo Extensions.

One more "gotcha" when working with manifest files. The last line of the manifest file must include a blank line. The manifest parser will get confused if your end-of-file is at the end of the 4th line of text, so be sure and add an extra <CR> to the end of the last line. Here's an example manifest file.

```
Manifest-Version: 1.0
Extension-Name: colorize
Class-Manager: colorize.ColorExtension
NetLogo-Extension-API-Version: 5.0
```

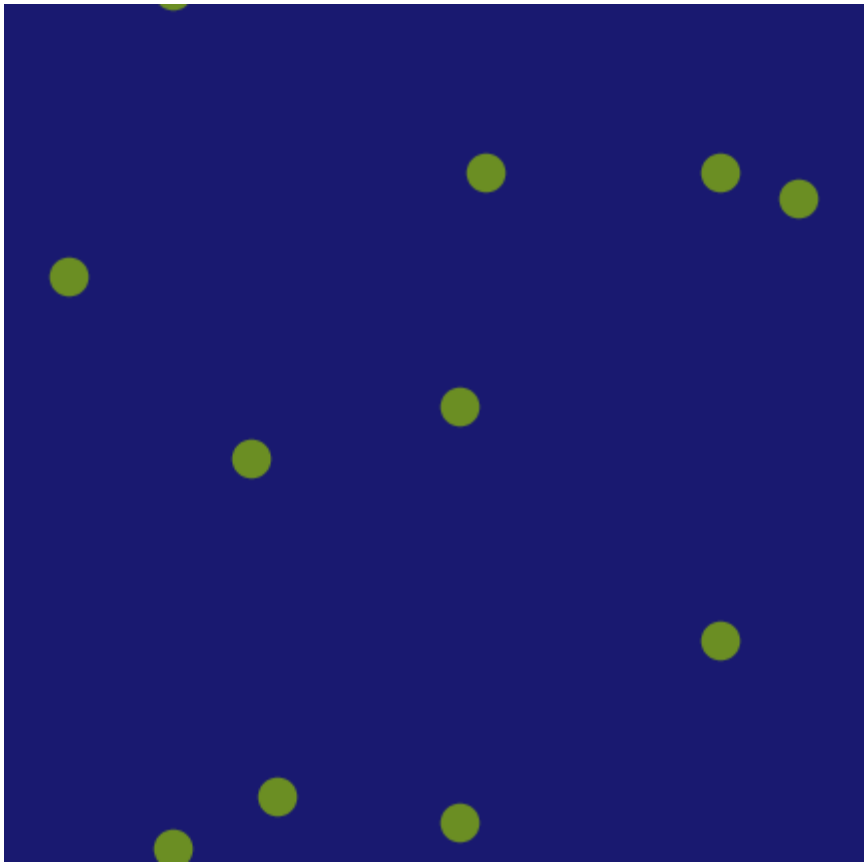
Using our Extension

Now that we've built the Java code and created and installed a JAR file into the NetLogo extensions folder, let's use our new code. Here's a small NetLogo model that uses the new "colorize" extension.

```
extensions [ colorize ]

to setup-patches
  ask patches [
    set pcolor colorize:lookup-rgb "MidnightBlue"
  ]
end

to setup-turtles
  create-turtles 10 [
    set size 1.5
    set shape "circle"
    set color colorize:lookup-rgb "OliveDrab"
    setxy random-pxcor random-pycor
  ]
end

to setup
```

```
  clear-all
  setup-patches
  setup-turtles
  reset-ticks
end

to step
end

to go
  step
  tick
end
```

If we startup the NetLogo environment and enter "setup" at the observer prompt, we'll get a landscape and some turtles that look like this.



Wrapping Up

Hopefully this discussion has got you interested in extending NetLogo using the Java APIs. The example here is just a starter, and there's a lot more you can do to incorporate either your own Java code or libraries developed elsewhere into your NetLogo models.

As a next step, you may want to check out some of the excellent tutorials on integrating Java and NetLogo listed below.

http://scientificgems.wordpress.com/2013/12/11/integrating-netlogo-and-java-part-1/
http://scientificgems.wordpress.com/2013/12/12/integrating-netlogo-and-java-2
/http://scientificgems.wordpress.com/2013/12/13/integrating-netlogo-and-java-3/

Another good overview of the NetLogo extension building process can be found here: https://github.com/NetLogo/NetLogo/wiki/Extensions-API

For further discussion of the JAR file creation and Manifest configuration options in Eclipse, see: http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftasks-35.htm

10. Next Steps

We've covered a lot of ground in this field guide, yet there is a whole lot more to NetLogo. Hopefully you're better prepared now to dive in to the NetLogo ocean and swim on your own. Happy modeling!

Index