5 The Components of Agent-Based Modeling

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

—Rene Descartes

The whole is more than the sum of its parts.

—Aristotle

It's turtles all the way down.

—Author unknown

Now that we have had some experience with extending agent-based models and building ABMs on our own, we can take a step back and more comprehensively examine the individual components of such models. This gives us a chance to reflect on some issues that arise when implementing new ABMs. We begin this chapter by laying out an overview of the components of an ABM; in some cases, this will be a review of features discussed in previous chapters. We will then discuss each of the components in turn. At the end of the chapter, we will discuss how all of these components come together to create a full set of tools for the construction of ABMs.

Overview

As we have described in chapter 1, the raison d'être of agent-based modeling is the idea that complex systems can be productively modeled and explained by creating agents and environment, describing their behavior through agent rules, and specifying agent-agent and agent-environment interactions. This description of an ABM is itself a simplified picture, a model of agent-based modeling, if you will. The main components of any ABM are *agents*, *environment*, and *interactions*. Agents are the basic ontological units of the model, while the environment is the world in which the agent lives. The distinction between agents and environment can be fluid, as the environment can sometimes be modeled as agents. Interactions can occur either between agents or between agents and the environment. Agent actions can also occur internally, directly affecting only the agent's internal state. Such is the case when an agent is trying to determine which action to take, as when the residents in the Segregation model are deciding whether or not they are unhappy. The environment is not merely passive; it can also act autonomously. For instance, in the models we discussed in chapter 4, the grass regrows on its own.

To this list of three basic components, we will also add two additional components. The first is called the Observer/User Interface. The observer is an agent itself,[1] but one that has access to all of the other agents and the environment. The observer *asks* agents to carry out specific tasks. Users of agent-based models can interact with the agents by way of the User Interface, which enables the user to tell the observer what the model should do. The second component, the Schedule, is what the observer uses to tell the agents when to act. The Schedule often involves user interaction as well. In NetLogo models, the interface typically includes SETUP and GO buttons. The user presses SETUP and then GO to schedule these events to occur.

We will now describe each of these five components in further depth. Throughout this chapter, we will use a variety of models from the NetLogo models library to illustrate our examples. We start by examining the Traffic Basic model, a simple model of traffic flow (found in the Social Science section of the library). An early version of this model was designed by two high school students (Resnick, 1996; Wilensky & Resnick, 1999) to explore how traffic jams form. The students thought that they would have to include traffic accidents, radar traps, or some other form of traffic diversion to create a traffic jam, but they built an initial model without any such hindrances. To their surprise, traffic jams still formed despite the lack of impediments. This happens because as cars speed up and approach the cars in front of them, they eventually have to slow down, causing the cars behind them to slow down creating a ripple effect backward. Eventually, the car at the front of the jam will be able to move again, but by that time, there are many cars behind that car that cannot move, causing the traffic jam to move backward even as the traffic jam moves forward.[2] This simple model (figure 5.1) again illustrates that emergent phenomena often do not correspond to our intuitions. As we discussed in chapter 0, one common pitfall in making sense of emergent phenomena is the tendency toward levels confusion. In this case, it is easy to misattribute the properties of the individual agents (the cars) to the jam. Hence, it seems paradoxical to us that the cars would move forward while the jam moves backward.

1. In NetLogo, the observer is conceptualized as an agent with a point of view. In some other ABM packages, it is conceptualized as a "disembodied" controller.

2. This is an example of how levels affect perception, as we discussed in chapter 0.

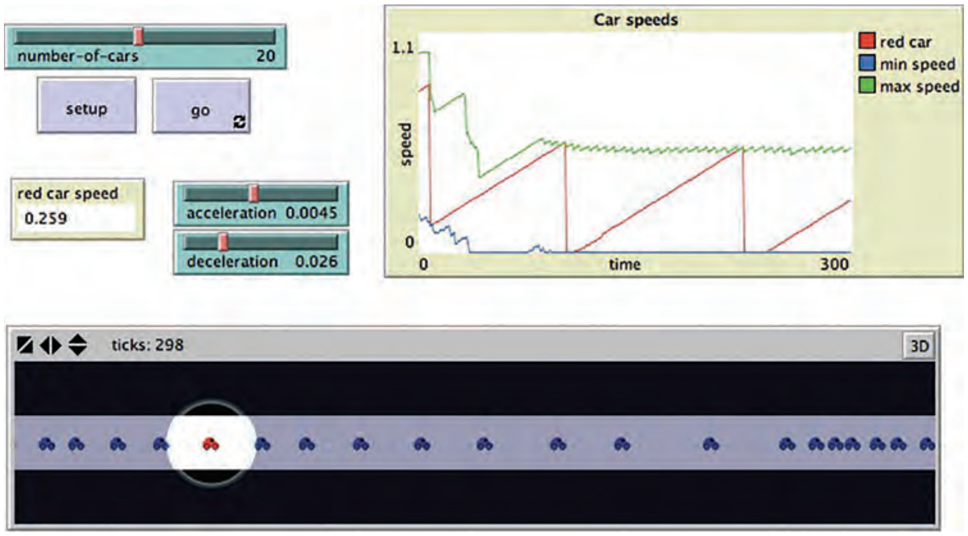


Figure 5.1
Traffic Basic model (Wilensky, 1997b).

Agents

Agents are the basic units of agent-based modeling. As such, it is important to choose the design of your agents carefully. The two main aspects that define agents are the *properties* that they have and the *actions* (sometimes called behaviors or methods) that they can execute. *Agent properties* are the internal and external state of the agents—their data and description. *Agent behaviors or actions* are what the agents can do.

Besides these two main agent attributes, there are also several issues that are related to agent design. First is the issue of agent "grain-size": which is most effective for the chosen model? For example, if you are modeling a political system, do want to model the individual actors or, instead, the political institutions or even each nation's government as a single entity? A second factor to consider is agent cognition. How much capability do the agents have to observe the world around them and make a decision? Do they act in a stimulus-response fashion? Or do they plan out their actions? Finally, we discuss some special types of agents: proto-agents, which are not fully specified agents; and meta-agents, composed of other agents.

Properties

Agent properties describe an agent's current state, the items that you see when you inspect an agent. In chapter 2, we briefly described how to use a patch monitor to see the current state of a patch, but you can also use monitors to inspect turtles and other types of agents as well. In this example, we are going to explore agent properties using a turtle monitor, but in NetLogo, you can monitor links and patches as well.

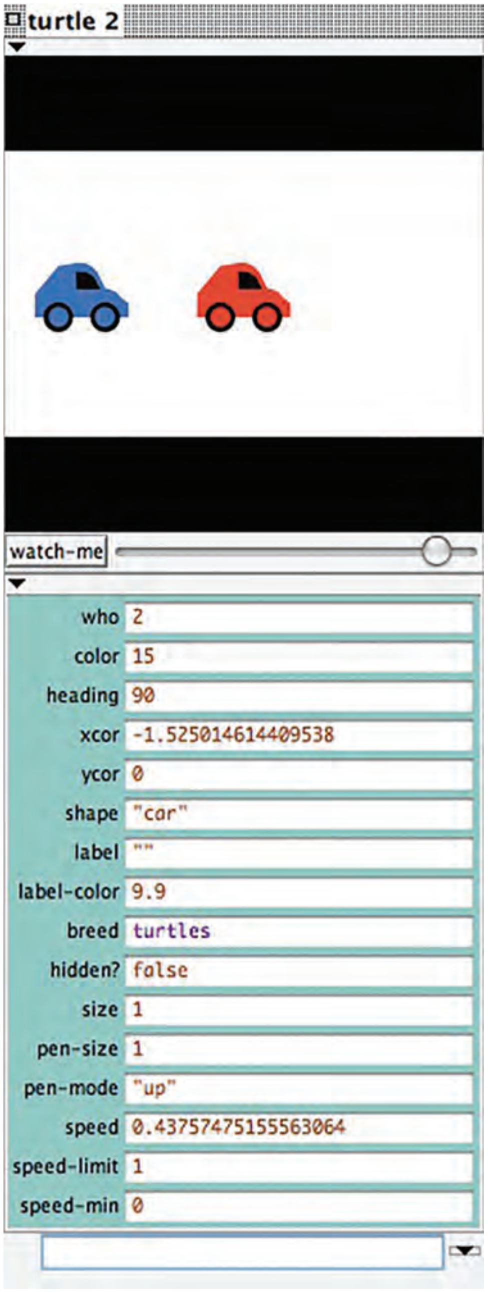
If you inspect one of the cars in Traffic Basic, you see (under the graphical image of the agent's local environment) the list of properties described in figure 5.2.[3] This list contains two sets of properties. The first set is a standard set of properties of every turtle created in NetLogo: WHO, COLOR, HEADING, XCOR, YCOR, SHAPE, LABEL, LABEL-COLOR, BREED, HIDDEN?, SIZE, PEN-SIZE, and PEN-MODE. Patches and links also have a default set of properties. For patches, these are PXCOR, PYCOR, PCOLOR, PLABEL, and PLABEL-COLOR, while for links, these are END1, END2, COLOR, LABEL, LABEL-COLOR, HIDDEN?, BREED, THICKNESS, SHAPE, and TIE-MODE. (All of these properties are described in more detail in the NetLogo User Manual.)

In NetLogo, turtles and links have COLOR as a property, while patches have PCOLOR as a property. Likewise, turtles have XCOR and YCOR as properties, while patches have PXCOR and PYCOR. To make things simpler, turtles can directly access the underlying properties of their current patch. For instance, for a turtle to set its color to the color of the underlying patch (effectively making it invisible), the turtle can just execute SET COLOR PCOLOR. If the patch property had the same name as the turtle property, this would not work, since there would be a confusion over which COLOR the code was describing. Only turtles can access patch properties directly, since a turtle can only be on one and exactly one patch. Links can span multiple patches and are always connected to multiple turtles, so you have to specify which patch you are referring to if you want to use a patch property in a link procedure. Similarly, you have to specify which link you are referring to when you want to use a link property in a turtle or patch procedure. Moreover, patches cannot directly access turtle or link properties, because a patch could have 0, 1 or many links or turtles on it. Therefore, you must specify which turtle or link you are referring to when accessing their properties.

In the inspector window, the default properties of the agent appear first. These are followed by the properties that the model author has specifically added to the agents for the model (e.g., in figure 5.2, SPEED, SPEED-LIMIT, SPEED-MIN). These author-defined properties should be described in the Info Tab associated with the model as well as in the comments in the Code Tab. For this model, SPEED describes the current speed of the car, SPEED-LIMIT is the maximum speed of the car and SPEED-MIN, its minimum speed.

When the Traffic Basic model starts up, all of these properties are set in the SETUP-CARS procedure:

3. You can inspect an agent by right-clicking on it or by using the NetLogo INSPECT primitive.



who	2
color	15
heading	90
xcor	-1.525014614409538
ycor	0
shape	"car"
label	""
label-color	9.9
breed	turtles
hidden?	false
size	1
pen-size	1
pen-mode	"up"
speed	0.43757475155563064
speed-limit	1
speed-min	0

Figure 5.2
Agent properties for the Traffic Basic model.

```
set speed 0.1 + random-float .9
set speed-limit 1
set speed-min 0
```

The SPEED-LIMIT and SPEED-MIN are set to constant values, which means that all of the cars in this model will have the same SPEED-LIMIT and SPEED-MIN. However, the SPEED is set to a constant value plus a randomly drawn value. This will cause all of the cars to have at least a speed of 0.1 and at most a speed of just less than 1.0, but each car will (probably) have a different speed. It is possible for two cars to have the exact same speed, but that would require both of them to generate the same random floating-point number, and that is very unlikely.

The current method for initializing the speed of the cars uses a uniform distribution. As a result, it is just as likely that there will be as many cars with a speed of 0.2 as there will be cars with a speed of 0.9. However, often we want all the agents to all have roughly the same value for a property, but with some variation. A common way to achieve this result is by setting the property with a *normally* distributed random variable instead of a uniformly distributed random variable. For instance, in the Traffic Basic model, we could rewrite the example of setting the speed to work like this:

```
set speed random-normal .5 .1
```

Box 5.1
Basic Properties of Agents

In NetLogo, there is a standard set of properties that all agents have, some of which are common across toolkits and are essential properties for ABM. Here are some basic properties of NetLogo's turtle agents (for a complete list of properties, refer to the NetLogo documentation):

WHO Calling this property a "who number" is unique to NetLogo, but all ABM toolkits have some unique identification number or string that is used to track each agent over time.

XCOR and YCOR These are the coordinates of the agent in the world. They are useful for determining where the agent is in the world, and how the agent relates to other agents.

HEADING The heading of an agent—the direction it is facing—is an intrinsic property of NetLogo agents. In some toolkits agents have no heading built-in, but if the model includes motion then it is often useful for the agents to have a heading property.

COLOR The color used to display the agent in the ABM visualization. In NetLogo, both turtles and links have COLOR, and patches have a PCOLOR property. (In some ABM toolkits color is not a standard agent property, when model visualization is less closely tied to model mechanics.)

RANDOM-NORMAL takes the mean of the distribution you want to draw from and the standard deviation. This code will give each car a different speed, but it will be close to 0.5. The 0.1 specifies how distributed the speed is around the mean value of 0.5. If we change the standard deviation from 0.1 to 0.2, then there is a higher probability that one of the agents will have a speed much greater or less than 0.5. With 0.1, 67 percent of the cars will have a speed between 0.4 and 0.6; 97 percent will have a speed between 0.3 and 0.7; and 99 percent of the cars will have a speed between 0.2 and 0.8.

So far, we have initialized agent properties by setting them to constant values or to statistical distributions. Other ways to initialize agent properties are to set the values from a list or from a data file. If, for example, we were trying to replicate a particular instance of a traffic pattern, then we might know the initial speeds of all the cars that were involved. We could store this in a list and then initialize each agent from this list. This method enables us to recreate particular empirical examples and applications.

We can also change agent properties during the running of the model. For instance, in Traffic Basic, the speed parameter mentioned before gets modified throughout the running of the model to increase and decrease the speed as the car is able to move. In the SPEED-UP-CAR procedure, the speed is increased:

```
set speed speed + acceleration
```

Thus properties define an agent's current state, but they can also change to reflect how the state changes as the model progresses.

Behaviors (Actions)

Besides items that define the state of an agent (properties), it is also necessary to define how the agent can behave (the actions it can take). An agent's *actions* or *behaviors* are the ways in which the agent can change the state of the environment, other agents, or itself. In NetLogo, there are many behaviors that are predefined for the agents. The list of all these predefined behaviors is too large to iterate through here, but it includes actions like FORWARD, RIGHT, LEFT, HATCH, DIE, and MOVE-TO. (For a complete list of the predefined behaviors in NetLogo, you can look in the NetLogo Dictionary under the categories Turtle, Patch, and Link.) Unlike properties, which can be viewed through the inspector, this is the only way to find out what predefined behaviors an agent possesses.

It is also possible to define new behaviors for agents to carry out within a particular model. For example, in the Traffic Basic model, the agents have two additional behaviors that they can carry out—SPEED-UP-CAR and SLOW-DOWN-CAR—both of which modify the agents' speed:[4]

4. In NetLogo, it is customary to note that an action applies to a particular agent type by adding a comment that describes the action as a "turtle procedure."

```
;; turtle procedure
to slow-down-car
  set speed speed - deceleration
end

;; turtle procedure
to speed-up-car
set speed speed + acceleration

end
```

In addition to accelerating or decelerating, each car adjusts its speed based on the speed limits and always moves forward according to its speed. In the Traffic Basic model, the code for that action is in the main GO procedure:

```
    let car-ahead one-of turtles-on patch-ahead 1
    ifelse car-ahead != nobody
[slow-down-car ]
      ;; otherwise, speed up
      [ speed-up-car ]
      ;; don't slow down below speed minimum or speed up beyond speed limit
    if speed < speed-min [ set speed speed-min ]
    if speed > speed-limit [ set speed speed-limit ]
    fd speed ]
```

Each car first checks to see if there is another car/turtle ahead of it. If there is, it then slows down to a speed below the car ahead, making sure that it does not run into the car ahead of it. Otherwise, it speeds up (but never exceeds the speed limit). This agent action is an *interaction,* as a car interacts with another car by sensing that car's speed and changing its own state based on the other car. Conversely, if a car slows down, this will then

Box 5.2
Basic Behaviors/Actions of Turtle Agents

Most ABM toolkits have a standard set of actions that agents have. Many of these are common across toolkits and are essential properties of any ABM. Here are some common actions that NetLogo's turtle agents can take:

FORWARD/BACKWARD Enables the agent to move forward and backward within the world.

RIGHT/LEFT Enables the agent to change its heading within the world.

DIE Tells an agent to destroy itself and removes the agent from all appropriate agent sets.

HATCH Creates a new agent that is a copy of the current agent with all of the same properties as the current agent.

force the cars behind it to slow down as well. The action of each agent affects the actions of other agents.

So far, we have seen cars changing their own internal state and affecting the state of other cars. You might also imagine how cars might affect the road that they are driving on, thereby affecting the environment of the model. For example, the more traffic there is on a road, the more worn down the road might become, so we could add a WEAR property to the patches of the road and then add a WEAR-DOWN procedure in which the cars wear down the road over time. This, in turn, might affect the top speed that the cars can reach on that section of road.

Behaviors are the basic way for agents to interact with the world. We will discuss some traditional interaction mechanisms later on in this chapter.

Collections of Agents

Types of Agents Agents are typically divided into three main types: mobile agents that can move about the landscape; stationary agents that cannot move at all; and connecting agents that link two or more other agents. In NetLogo, turtles are mobile agents, patches are stationary agents, and links are connecting agents. From a geometric standpoint, turtles are generally treated as shapeless area-less points, although they may be visualized with various shapes and sizes). Thus, even if a turtle appears to be large enough to be on several patches at the same time, it is only contained by the patch at the turtle's center (given by XCOR and YCOR). Patches are sometimes used to represent a passive environment and are acted upon by the mobile agents; other times, they can take actions and perform operations. A primary difference between patches and turtles is that patches cannot move. Patches also take up a defined space/area in the world; thus, a single patch can contain multiple turtle agents on it. Links are also unable to move themselves. They connect two turtles that are the "end nodes" of the link, but the visual representations of the links do move when their end nodes move. Links are often used to represent the relationship between turtles. They can also represent the environment—e.g., by defining transportation routes that agents can move along, or by representing friendship/communication channels. Despite these differences, all three share the ability to behave, that is, to take actions and perform operations on themselves.

In some ABM toolkits, the environment is passive and does not have the full capabilities of an agent. Giving the environment, the ability to directly perform operations and actions allows an easier representation of many of the autonomous environmental processes. For instance, in the Wolf Sheep Simple model in chapter 4, we could have the grass regrow by just adding to the grass variable instead of asking all the patches in the environment to execute a grass-growth behavior. Though computationally this may not be that different, representing the environment as a collection of agents allows users to reason spatially locally—e.g., they can take the viewpoint of a patch of grass, which can make the model easier to understand.

Breeds of Agents Besides these three predefined agent-types, modelers can also create their own types of agent called breeds. The breed of an agent designates the category or class to which the agent belongs. In the Traffic Basic model, all of the agents are of the same type, so it is not necessary to distinguish between different agent breeds. Instead, we used agents of the "turtles" breed, the default breed in NetLogo. Different breeds of agents are required if different agents have different properties or actions.

In chapter 4, we had two breeds of agents, the "wolf" the "sheep" breeds. Even though these two breeds had the same set of properties, it was useful to create two breeds because each had different characteristic actions—wolves eat sheep, while sheep eat grass. We defined the breeds at the beginning of the model:

```
breed [sheep a-sheep]
breed [wolves wolf]
```

At the same time, we can define the properties that breeds have:

```
turtles-own [energy]
```

In this case, both the wolves and the sheep have the same property of "energy," but we could also give them separate properties. For instance, we could give the wolves a "fang-strength" property and the sheep a "wooliness" property. The fang-strength could be used to determine how successful a wolf is at killing sheep. Wooliness, by contrast, could be an indication of how well the sheep was able to fend off the wolves' attack, if we assume that the wolves sometimes get a mouthful of wool instead of flesh. If this were the case, we would add two additional lines:

```
sheep-own [ wooliness ]
wolves-own [ fang-strength ]
```

These properties are in addition to the properties that all agents have, so that the sheep would have "wooliness" and "energy" just as the wolves have "fang-strength" and "energy."

Sets of Agents Breeds are particular collections of agents where the collections are defined by the kinds of properties and actions of their agents. We can also define collections of agents in other ways. NetLogo uses the term *agentset* to designate an unordered collection of agents, and we will also use this term/definition throughout this textbook. Usually we construct agentsets either by collecting agents that have something in common (e.g., agent location or other properties) or by randomly selecting a subset of another agentset.

In the Traffic Basic model, we could create a set of all the agents that have a speed over 0.5. In NetLogo, this is usually done with the WITH primitive. Here is an example of how we could create such an agentset that we could insert into the GO procedure in this model:

```
let fast-cars turtles with [speed > 0.5]
```

However, we usually want to ask these turtles to do something specific. For instance, we can ask all of the turtles that have a high speed to set their size bigger so they are easier to see.

```
let fast-cars turtles with [speed > 0.5]
ask fast-cars [
  set size 2.0
]
```

The “let” statement above collects the fast cars into an agentset. If we don’t plan to “talk to” this agentset again, we can do without the LET and instead construct the agentset “on the fly” and ask the turtles directly:

```
ask turtles with [speed > 0.5] [
    set size 2.0
]
```

If you insert this code into the model, you will soon see that all of the cars are big. This is because we never told the turtles to set their size to small again, once their speed has dropped below 0.5. To do that we would need to add some more code. One way to accomplish this would be to use another ask.

```
ask turtles with [speed > 0.5] [
    set size 2.0
]
ask turtles with [speed <= 0.5] [
    set size 1.0
]
```

Another way to accomplish the same goal, without creating agentsets, is to ask all the turtles to do something, but choose what actions they take based on their properties. In the previous example, all of the faster turtles took their actions before any of the slower turtles took their actions. In the following example, all of the agents are asked with the same ASK command, so the order in which the turtles take actions will be random. In this case the result of running the code will be the same, but depending on what actions the agents take (for instance, if they were to change their speed, instead of their size), the results could be different.

```
ask turtles [
    ifelse speed > 0.5 [
set size 2.0
]
[
    set size 1.0
    ]
]
```

Another method for creating agentsets is on the basis of their location. The Traffic Basic model does this in the GO procedure:

```
let car-ahead one-of turtles-on patch-ahead 1
         ifelse car-ahead != nobody
             [ slow-down-car ]
```

TURTLES-ON PATCH-AHEAD 1 creates an agentset of all of the cars in the patch in front of the current car. It there is at least one such car, it then selects one of these cars at random, using the ONE-OF primitive, and sets the speed of the current car to a little less than the speed of that car ahead. There are many other ways to access a collection of agents based on their location, such as using NEIGHBORS, TURTLES-AT, TURTLES-HERE, and IN-RADIUS.

It is also often useful to create a randomly selected set of agents. These agents are not related in any particular way, but rather, are a collection of agents that we want performing some action. In NetLogo, the primary procedure for doing this is N-OF. For example, the Segregation model that we talked about in chapter 3 used N-OF to ask a random set of patches to create a set of red turtles and then asked all of those turtles to turn green or red randomly:

```
;; create turtles on random patches
ask n-of number patches
    [ sprout 1
         [ set color one-of [red green] ]
```

Agentsets and Lists Throughout our examples, we have used both agentsets and lists, sometimes explicitly and sometimes implicitly. This may be a good time to stop and clarify what they are, how they work, how they differ, and under what circumstances we would use one over the other. Agentsets and lists are both variables that can contain one or many other variables. We can specify that a variable is an agentset or a list when we create them, by using their respective constructor reporters. If we want to create an empty list, we can write

```
let a-list []
```

If we want, we can then add numbers, strings, and even turtles to the list.

```
;; we put items at the start of the list
set a-list fput 1 a-list
set a-list fput “and” a-list
set alist fput turtle 0 a-list
show a-list
;; prints [(turtle 0) “and” 1
```

In contrast to lists, which can hold any type of item, an agentset can hold only agents. Moreover, it can hold only agents of the same type, such as turtles, patches, or links. NetLogo has special reporters for empty agentsets, no-turtles, no-patches, and no-links.

```
;; this creates an empty agentset of turtles
let an-agentset no-turtles
```

no-turtles is an empty turtle agentset (i.e., an agentset containing no turtles), so by setting an-variable to no-turtles, we specify that this variable is of the type agentset. Now that we have an empty agentset, we can then add turtles to it by using the turtle-set reporter:

```
;; this adds turtle 0 to the agentset
set an-agentset (turtle-set turtle 0)
;; this adds turtle 1 and turtle 2 to the agentset
set an-agentset (turtle-set an-agentset turtle 1 turtle 2)
show an-agentset
;; prints (agentset, 3 turtles)
```

We can only put agents (turtle-breeds, patches and links) in agentsets, but we can put anything we want, including agents, in lists. There are two important properties that set agentsets and lists apart from each other.

First, we can “ask” agentsets to do things, but we cannot ask lists to do things. So for instance, when we use

```
ask turtles [] ;; do stuff
```

what we are really doing is first creating an agentset of all turtles, and then asking all those turtles, in a random order, to do whatever we put in the brackets. If we try this with lists, we will simply get an error.

Second, agentsets are unordered. This means that whenever we invoke them, they will output a list of agents in a random order. Notice how showing a list shows both what is inside the list and the order in which it appears, but showing an agentset shows only what is inside the agentset.

So when we want to interact with turtles, when would we use one or the other? Unless we have a very good reason to, we always use agentsets. The primary reason is that we usually want our turtles to do things in a random order. That is because there could be threats to model validity if the agents always execute in the same order. For instance, in the Wolf Sheep Predation model, some sheep would have an unfair advantage if they are always first to check if there is grass on their patches.

But sometimes we do want to determine the order in which turtles do things. For instance, it could be that we *want* some turtles to have an advantage. In that case we would need to create an ordered list, and order them by whatever parameter we think determines their advantage. If turtles have different information, we might, for example, want the turtles with more information to take their turns later.

```
turtles-own [information] ;; information determines how late they get their turn
;; (later is better because then they will know what everyone else did
;; before making their decision)

;; create an empty list
let a-list []
;; sort-on reports a list containing turtles sorted by the parameter specified
;; in the brackets
set a-list sort-on [information] turtles
```

a-list now contains all turtles sorted by information in ascending order. But, as we just discussed, we cannot ask lists of agents to do things. Rather, we must iterate through the lists, and ask each agent in the list to do what we ask. For this we can use the foreach command:

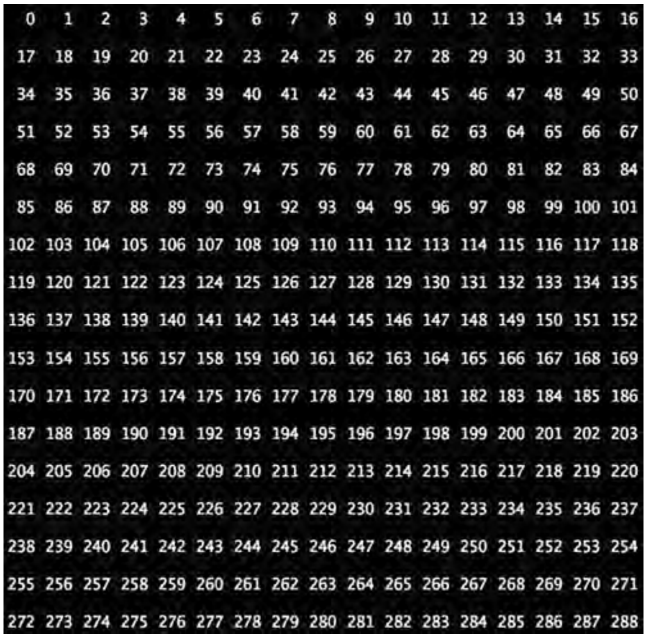
```
foreach a-list [
    ask ? [ ] ;; do stuff here
]
```

The "?" is a special variable that takes on the value of each element of the list. So this code will iterate through each turtle in the list, and ask each in that particular order, to do what we specify in the brackets. We will further discuss the "?" variable later in this chapter.

Similarly we can create lists of patches. Suppose we want to label the patches with numbers in left-to-right, top-to-bottom order. We can create a sorted list of patches and then label them using the code below:

```
;; patches are labeled with numbers in left-to-right,
;; top-to-bottom order
let n 0
foreach sort patches [
    ask ? [
       set plabel n
       set n n + 1
    ]
]
```

Agentsets and Computation Before we end our discussion of agentsets, it should be noted that once you ask an agentset to perform an action, all the agents collected at that moment (and only those agents) will perform the action. If one agent's (agent A) action causes another agent (agent B) in the agentset to no longer satisfy the collection criteria, B will



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101
102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118
119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152
153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186
187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203
204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237
238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254
255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271
272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288

Figure 5.3
Patches labeled in ascending order.

still perform the action. Likewise, if A's action causes agent C, which is not in the agentset, to meet the collection criteria, C will still not perform the action.

As an illustration, let's consider this code snippet from the Agentset Ordering model in the chapter 5 subfolder of the IABM Textbook folder in the models library:

```
to setup
    clear-all
    create-turtles 100 [
       set size random-float 2.0
       forward10
    ]
end

to go
    ask turtles [
       set color blue
       ]
    ask turtles with [ size < 1.0 ] [
       ask one-of turtles with [size > 1.0] [
           set size size - 0.5
       ]
       set color red
    ]
    print count turtles with [ color = red ]
    print count turtles with [ size < 1.0 ]
end
```

The bolded code has the potential to change the set of agents with size < 1.0 by making some agents smaller, thereby changing the agentset defined in the first ASK. For example, if we started with a total of ten agents, with eight of those having a size less than 1, and the other two, are size 1.2, then any big turtle that was selected by at least one small turtle would shrink to a size below 1.0. However, this newly small turtle will not execute the inner ask, since the first agentset, with 8 members, is held constant as soon as it is created; therefore, the agents taking actions will not change until the next time you execute the GO loop and recreate the agentset. The net result is that, at the end of the execution of the GO procedure, it is possible to have a turtle that has a SIZE less than 1.0 and is still blue. The print statements confirm that these two values, the counts of red turtles and small turtles, are not equal. This can be confusing when first thinking about agentsets, but really, this is a variant of the same issue you run into whenever you take action based on the condition of a variable and that action affects the value of the same variable.

Another problem that often comes up when working with agentsets pertains to computational efficiency. Sometimes it is more efficient to compute an agentset before performing a set of operations that involve the agentset. For instance, look at the GO-1 procedure below from the Agentset Efficiency model in the chapter 5 subfolder of the IABM Textbook folder of the models library:

```
;; GO-1 sets red patch labels to a small random number and
;; green patch labels to a larger random number
to go-1
    if any? patches with [ pcolor = red] [
       ask patches with [ pcolor = red] [
              set plabel random 5 ;; red patches are labeled 0-4
       ]
    ]
    if any? patches with [ pcolor = green] [
       ask patches with [ pcolor = green] [
              set plabel 5 + random 5 ;; green patches are labeled 5-9
       ]
    ]
    tick
end
```

This code computes each of the two agentsets PATCHES WITH [PCOLOR = RED] and PATCHES WITH [PCOLOR = GREEN] twice.

The procedure GO-2 below has the same behavior as GO-1, but it is more efficient in that it computes each of those agentsets only once. Computing agentsets can be an expensive operation, so it is often advisable to compute them first and then ask them to behave.

```
;; GO-2 has the same behavior as go-1 above. But it is more efficient as it
computes each of the patch agentsets only once.
to go-2
 let red-patches patches with [ pcolor = red ]
 let green-patches patches with [ pcolor = green ]
 if any? red-patches [
     ask red-patches [
    set plabel random 5
    ]
]
if any? green-patches [
    ask green-patches [
    set plabel 5 + random 5
    ]
]
 tick
end
```

In the code above, GO-2 is twice as efficient as GO-1. The efficiency problem can get even worse and the gain correspondingly bigger when the duplicated agentset construction is inside an ASK PATCHES. For example, if the GO loop, asked each patch to check and construct a neighbors agentset, then each patch would be executing code that is half as efficient as possible, and so computing the agentset ahead of time would provide an even larger gain in performance.

However, it is important that your code be readable, so others can understand it. In the end, computer time is cheap compared to human time. Therefore, it should be noted that,

Box 5.3
Computational Complexity

Agent-based models can be computationally intensive. It is often necessary to make sure that model code is as efficient as possible. Computing agentsets before iterating over them is one example of how you can make your computation more efficient, but there are many other ways as well. A standard way of describing the efficiency of an algorithm is Big-O notation. Big-O notation indicates how much time must be spent on an algorithm as a function of its inputs. For instance, the time it takes to complete a function that is $O(n)$ is linear in the size of its inputs, while the time it takes to complete a function that is $O(n^2)$ grows as the square of its inputs. If the number of inputs is large, then an $O(n^2)$ algorithm will take much longer to run than an $O(n)$ algorithm. For instance, if we compute the distance between each agent and the center of the world, that will only take $O(n)$ time, since we only need to examine each agent once. However, if we want to find the distance between each agent and every other agent in the world, that will take $n \cdot (n - 1)$ operations, which means it will run in $O(n^2)$ time. There is a whole field of computer science called computational complexity devoted to creating more efficient algorithms (Papadimitriou, 1994). In practice, Big-O notation is used for conveniently comparing two algorithms: if one algorithm runs in $O(n^2)$, and another operates in $O(n)$ time, then the $O(n)$ algorithm is faster and will usually be used.

whenever there is a possibility of trade-off, clarity of code should be preferred over efficiency.

A more pernicious issue arises if we change just one line of GO-1 and GO-2:

```
;; GO-3 shows what happens if patch colors are changed “on the fly”
to go-3
    if any? patches with [ pcolor = red] [
       ask patches with [ pcolor = red] [
              set pcolor green
       ]
    ]
    if any? patches with [ pcolor = green] [
       ask patches with [ pcolor = green] [
              set pcolor red
       ]
    ]
    tick
end
```

If you run this code, you might expect to get a picture that looks like figure 5.4; that is, you expect red patches to turn green and vice versa. In fact, this is not what happens. Instead, this code will result in a picture like figure 5.5, where all the patches are red. Why does this unexpected behavior happen? This is another example of the ordering issue we just discussed above. The first “if statement” turns the patches green, so by the time the

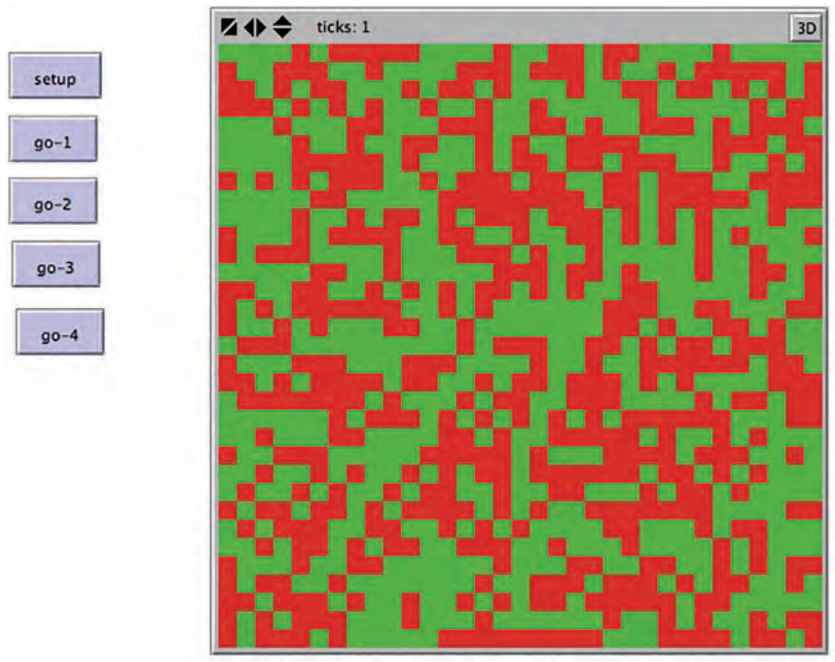


Figure 5.4
Constructing agentsets at the outset results in expected behavior.

second "if statement" is executed, all the patches are green and therefore then turn red. By computing the agentsets ahead of time, as in GO-4, you are not only using more efficient code but are also ensuring that the green-patches agentsets you ask to execute the instructions are the same green-patches agentsets as at the start of the procedure and not the set of green patches that result from the first "if-statement," resulting in the expected picture of figure 5.4.

```
;; GO-4 shows what happens if you keep track, at the outset, of which patches
;; are red and which are green
to go-4
         let red-patches patches with [pcolor = red]
         let green-patches patches with [pcolor = green]
         if any? red-patches [
             ask red-patches [
                 set pcolor green
             ]
         ]
         if any? green-patches [
             ask green-patches [
                 set pcolor red
             ]
         ]
    tick
end
```

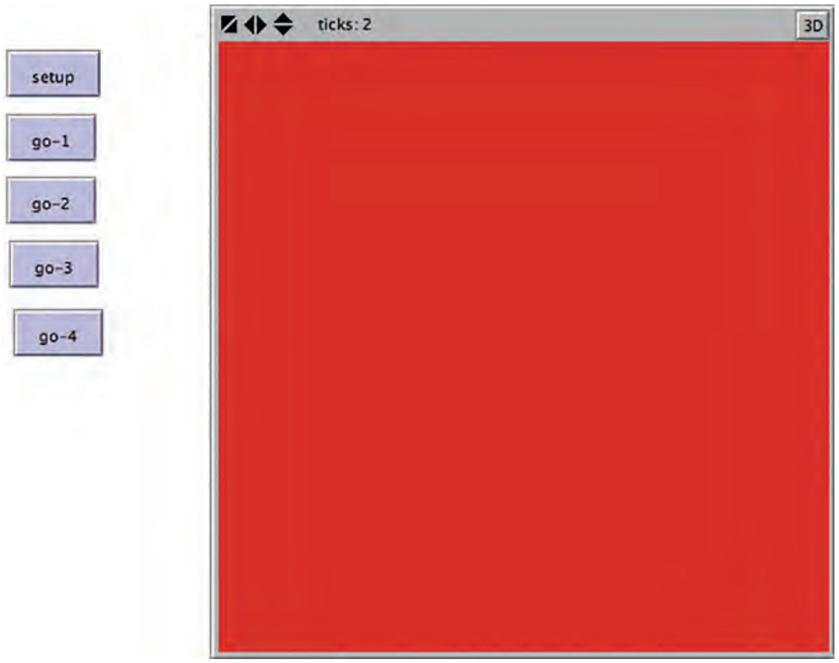


Figure 5.5
Constructing agentsets after the model state has changed, results in unintended outcome.

The Granularity of an Agent

One of the first considerations when designing an agent-based model is at what granularity should you create your agent—at what level of complexity is the agent you are modeling. In chapter 0, we discussed one prominent hierarchy of levels of complexity, e.g., atoms, molecules, cells, humans, organizations, and governments. Thus, what we mean by granularity of the agent is the point in such a hierarchy that we select the agents in the model. This point may seem obvious if you characteristically think of agents in an agent-based model as people, i.e., that there should be an agent for every individual person in the model. But that "obvious" answer is often not the best choice. For example, if you are modeling the interactions of national governments, you probably do not want to model each individual in each of the governments. Instead of the individuals within a government, it often makes more sense to model each government as an agent.

So how do you choose the level of complexity for the agents in your model? The guideline is to choose agents such that they represent the fundamental level of interaction that pertains to your question about the phenomenon. For instance, if you look at the Tumor model (shown in figure 5.6) in the Biology section of the NetLogo models library, the agents are cells within the human body, since the research question the model examines

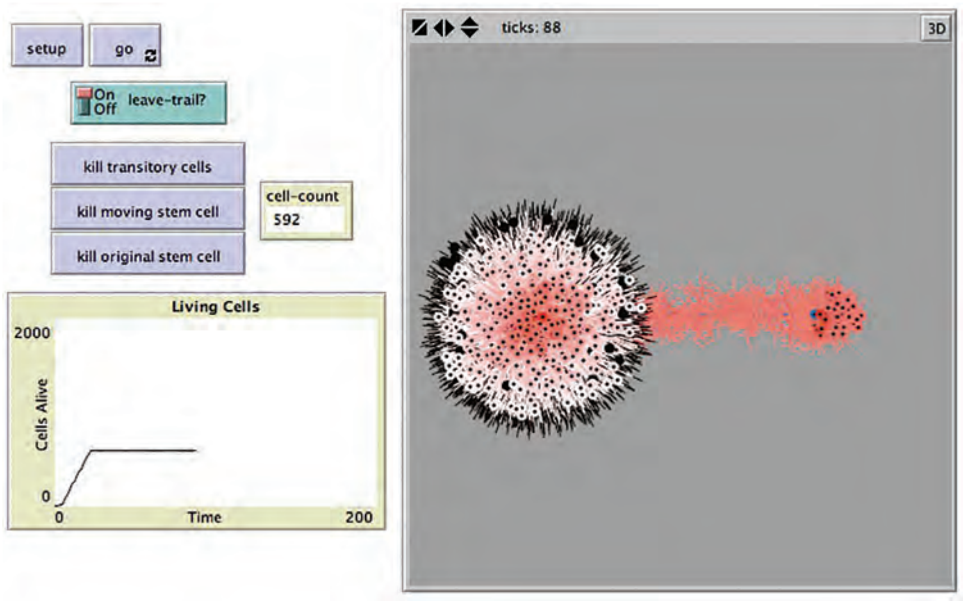
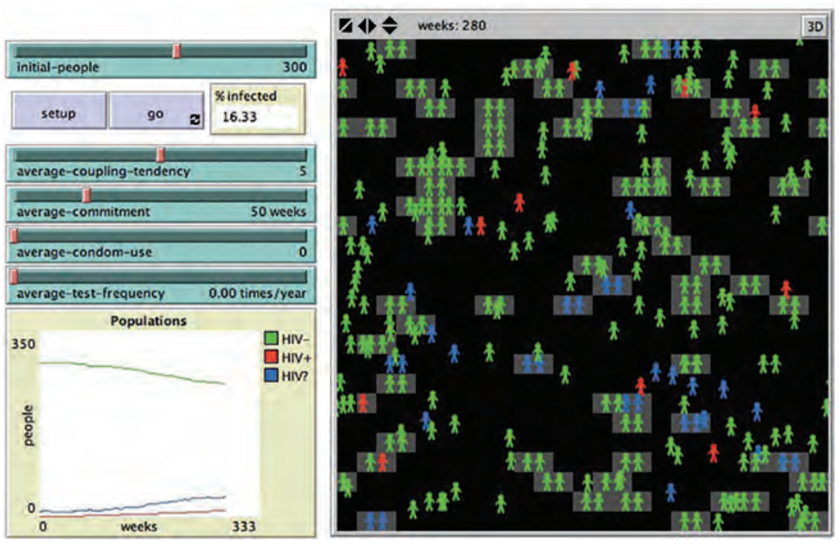


Figure 5.6
NetLogo Tumor model. http://ccl.northwestern.edu/netlogo/models/Tumor (*Wilensky, 1998b*).

is how tumor cells spread. However, even though the AIDS model in the Biology section of the NetLogo models library (see figure 5.7) is also concerned with disease, the basic agents are humans instead of cells. This is because the AIDS model is more concerned with how the disease spreads between humans instead of how it spreads within the human body. In the Tumor model, the question of concern is how cells interact to create cancer. Thus, the most important interactions happen at the cell level, and this makes cells a good choice for agents In the AIDS model, the question of concern is how humans interact to spread disease, with humans as the agents being the most appropriate choice. These two models highlight the relationship between the question being investigated, the interactions being modeled, and the resulting choices of granularity for the agents.

It is not always as easy to determine the most productive granularity for your agents, as these two cases might suggest. In some other cases, there may be more than one appropriate grain-size for the agents. For instance, in the Tumor model, it may be appropriate to move up to a larger level of aggregation. Besides looking at cells, it may also be useful to have agents represent organs so that the tumor's effect throughout the body can be examined. Within such models, it is often possible to view a group of cells as operating homogenously, meaning that you can view a few dozen cells as your core agent. This has the advantage of decreasing the computational requirements for your model. However, this has the trade-off of losing detail about the interactions that occur within this new grouping



initial-people 300

% infected 16.33

average-coupling-tendency 5

average-commitment 50 weeks

average-condom-use 0

average-test-frequency 0.00 times/year

weeks: 280

Figure 5.7
NetLogo AIDS model. http://ccl.northwestern.edu/netlogo/models/AIDS (*Wilensky, 1997a*).

of agents. Both individual cells and groups of cells can be valid choices for the level of complexity of these agents, and deciding between them will depend on details of the question you are asking as well as considerations of computational complexity.

Choosing the appropriate granularity for your agents can make your models much easier to create, as it allows you to ignore unnecessary aspects of the phenomena being modeled and instead focus on the interactions of interest. For instance, in the AIDS model, by not representing the actual AIDS virus, we can assess the spread of the disease through a population rather than its development within a single human. By contrast, by looking at cells in the Tumor model we are able to examine cellular interactions at a greater resolution than we could if we tried to use molecules as our agents. Selecting an appropriate grain-size for your agents is a critical early step in the development of a successful ABM.

It is also important that the granularity of your agents is comparable relative to each other within the model. This is sometimes known as the scale of the model. In other words, agents should operate on roughly the same time scales, and should have roughly the same physical presence within the model. If agents are not at the same scale it is still possible to include them within the same model, but it requires adjusting the model behavior so that they can interact appropriately.

Agent Cognition

As we have described, agents have different properties and behaviors. Still, how do agents examine their properties and the world around them to decide what actions to take? This question is resolved by a decision-making process called agent cognition. We will discuss several types of agent cognition: *reflexive agents*, *utility-based agents*, *goal-based agents*, and *adaptive agents* (Russell & Norvig, 1995). Often, these types of cognition are thought of in increasing order of complexity, with reflexive agents being the simplest, and adaptive agents the most complex. In reality, though this is not a strict hierarchy of complexity. Instead, these are descriptive terms that help us talk about agent cognition and can be mixed and matched; for instance, it is possible to have a utility-based adaptive agent.

Reflexive agents are built around very simple rules. They use if-then rules to react to inputs and take actions (Russell & Norvig, 1995). For instance, the cars in the Traffic Basic model are reflexive agents. If you look at the code that controls their actions, it looks like this:

```
let car-ahead one-of turtles-on patch-ahead 1  ;;choose a car on the patch ahead
ifelse car-ahead != nobody [  ;; if there is a car ahead
    slow-down-car car-ahead ;; set car speed to be slower than car ahead
]
[   ;; otherwise, speed up
    speed-up-car ;; increase speed variable by the acceleration
]
;; ….
fd speed
```

Put in words, this code says that if there are cars ahead, then slow down below the speed of the car in front; if there are not any cars ahead, then speed up. This is a reflexive action based on the state of the program, hence the name *reflexive agent*. This is the most basic form of agent cognition (and often a good starting point), but it is possible to make the agent cognition more sophisticated. For instance, we could elaborate this model by giving the cars gas tanks and fuel efficiencies based on the speed they are going. We could then have the cars change their speeds in order to improve their fuel efficiencies. As a result, the agents might have to speed up and slow down at different times than they currently do to minimize their gas usage while still not causing accidents. Giving the agents this type of decision-making process would give them a *utility-based* form of agent cognition in which they attempt to maximize a utility function—namely, their fuel efficiency (Russell & Norvig, 1995). To implement this model of agent cognition, we need to start by replacing our SPEED-UP-CAR procedure with a new procedure that accounts for the car's fuel efficiency.

```
;; choose a car on the patch ahead
let car-ahead one-of turtles-on patch-ahead 1
ifelse car-ahead != nobody [ ;; if there is a car ahead
    slow-down-car car-ahead  ;; set car speed to be slower than car ahead
]
[ ;; otherwise, adjust speed to find ideal fuel efficiency
    adjust-speed-for-efficiency ]
```

We want the ADJUST-SPEED-FOR-EFFICIENCY procedure to maintain the same speed if the car is at the maximally fuel efficient speed. If it is not at its most efficient speed, the logic in this procedure should have the car speed up if it is moving too slow and slow down if the car is moving too fast. Additionally, the car would still need to slow down if it was about to crash into the car in front of it. As such, the true utility function includes an exception that gives a utility of 0 to any action that results in a crash. As a result, we can leave the first part of the code that slows the car before it crashes and just add ADJUST-SPEED-FOR_EFFICIENCY when there is no car directly ahead, as illustrated in the code below from the Traffic Basic Utility model, which is also in the chapter 5 subfolder of the IABM Textbook folder of the NetLogo models library.

```
;; car procedure
to adjust-speed-for-efficiency
 if (speed != efficient-speed) [ ;; if car is at efficient speed, do nothing
        if (speed + acceleration < efficient-speed) [
;; if accelerating will still put you below the efficient speed then accelerate
            set speed speed + acceleration
        ]                                      ;;
;; if decelerating will still put you above the efficient speed then decelerate
        if (speed - deceleration > efficient-speed) [
            set speed speed - deceleration
        ]
    ]
end
```

In the language of utility functions, each car agent is minimizing a function f, defined by:

$$f(v) = |v - v^*|$$

where v is the current velocity of the car and v* is the most efficient velocity. The constraint that is described in the code is that the v cannot be adjusted arbitrarily, but instead, can only be changed by a limited increment every time step. By trying different values for EFFICIENT-SPEED, you can obtain quite different traffic patterns from the original model. For low values of EFFICIENT-SPEED, the system can achieve a free-flow state (with no jams) on a consistent basis.

Note that the original model code can be viewed also as maximizing a utility function—the simple utility function of the car's speed subject to the speed limit. However, this is such a simple utility function that we do not categorize the agent as a utility-based agent but rather as a reflexive agent. Such judgments can be subjective. While this second implementation of the car agent's cognition is more sophisticated than our initial design, it is still built on an underlying assumption that can be further refined. The modified code is also simplified in that it assumes we have predefined EFFICIENT-SPEED. But what if we

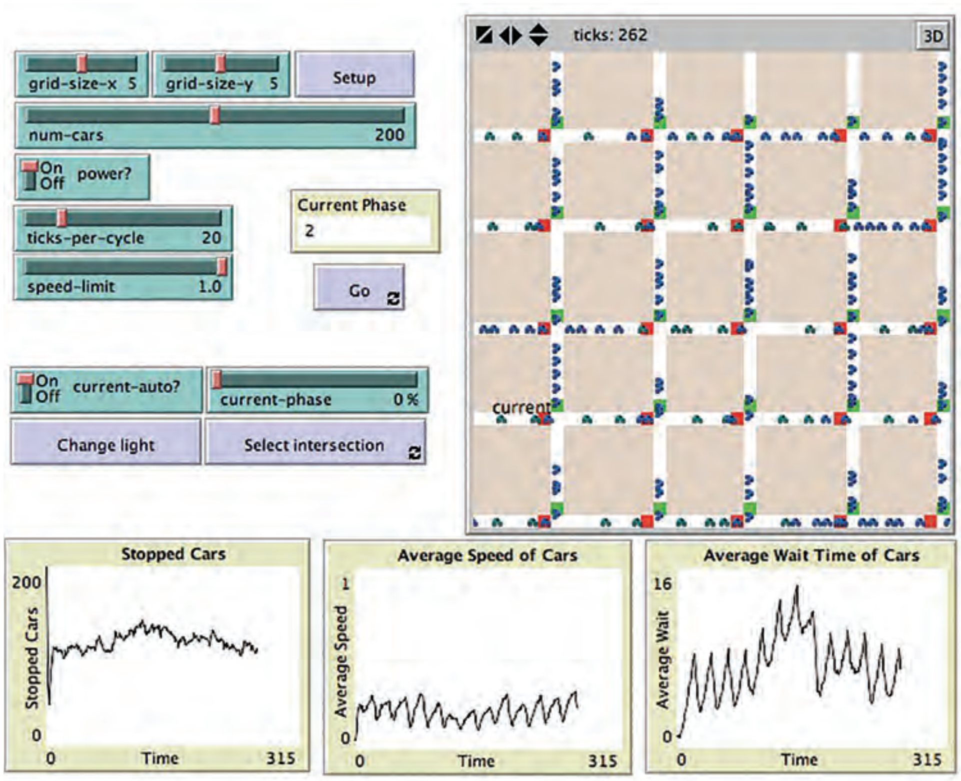

grid-size-x 5
grid-size-y 5
num-cars 200
power? On Off
ticks-per-cycle 20
speed-limit 1.0
Current Phase 2
current-auto? On Off
current-phase 0 %
ticks: 262

Figure 5.8
Traffic Grid model. http://ccl.northwestern.edu/netlogo/models/TrafficGrid (*Wilensky, 2002b*).

did not know what the most efficient speed for a car to travel at was? If we had access to the instantaneous gas mileage of the car, then we could actually learn these values over time. This addition would make our cars into simple *adaptive agents*.

The third type of agent cognition we will discuss is goal-based cognition. Imagine that each car in the Traffic Grid model (see figure 5.8) from the social sciences section of the NetLogo models library has a home and a place of work, with the goal of moving from home to work in a reasonable amount of time. Now, not only do the agents have to be able to speed up and slow down, but they also have to be able to turn left and right. In this version of the model, the cars are *goal-based agents*, since they have a goal (getting to and from work) that they are using to dictate their actions.

We will need to modify the Traffic Grid model so that each car has a house and a work location. We will want both of these locations to be off-road, but adjacent to the road. As the background of the grid is a shade of brown (color 38), we first create an agentset of all brown patches adjacent to a road (roads are white). These patches are the possible houses and work locations of the agents.

```
let goal-candidates patches with [pcolor = 38 and any?
neighbors with [pcolor = white]]
```

Then when we create the cars, each car sets its house to a random patch in that agentset and its work location to be any other patch in the agentset.

```
set house one-of goal-candidates
set work one-of goal-candidates with [ self != [ house ] of myself ]
```

The part of the GO procedure that is relevant to the cars looks like this:

```
;;set the cars' speed this tick, cars move forward an amount equal to their speed
;; record data for plotting, and set the color of the cars to either dark blue or cyan
;; based on their speed
ask turtles [
       set-car-speed
       fd speed
       record-data       ;; Record data for plotting
       set-car-color  ;; Set color to indicate speed
    ]
```

To implement the goal-based version of the model, we start by defining the procedure that the cars will use to navigate between their two desired destinations. Instead of just moving straight all the time, as with the cars in the original model, the new model should have the car at each step deciding which of its neighboring road patches is the closest to its destination, subsequently proceeding in that direction. We do this in the procedure called NEXT-PATCH. First, each car checks if it has arrived at its goal, and if so, switches its goal:

```
;; if I am going home and I am on the patch that is my home
;; I turn around and head towards work (my goal is set to "work")
    if goal = house and patch-here = house [
       set goal work
    ]
;; if I am going to work and I am on the patch that is my work
;; I turn around and head towards home (my goal is set to "home")
    if goal = work and patch-here = work [
       set goal house
    ]
```

The code above doesn't quite work. That's because the house and work are off-road and the cars remain on the road, so the condition "patch-here = house" or "patch-here = work" will never be satisfied. As we placed the house and work adjacent to the road, we can fix this by having the car check if it is right next to its goal.

```
;; if I am going home and I am next to the patch that is my home
;; I turn around and head towards work (my goal is set to "work")
if goal = house and (member? patch-here [neighbors4] of house) [
       stay set goal work
    ]
;; if I am going to work and I am next to the patch that is my work
;; I turn around and head towards home (my goal is set to "home")
    if goal = work and (member? patch-here [neighbors4] of work) [
        stay set goal house
    ]
```

Having established a goal, each car then chooses an adjacent patch to move to that is the closest to its goal. It does this by choosing candidate patches to move to (adjacent patches on the road) and then selecting the candidate closest to its goal.

The resultant NEXT-PATCH procedure is:

```
;; establish goal of driver
to-report next-patch
;; if I am going home and I am next to the patch that is my home
;; I turn around and head towards work (my goal is set to "work")
if goal = house and (member? patch-here [neighbors4] of house) [
        stay set goal work
    ]
;; if I am going to work and I am next to the patch that is my work
;; I turn around and head towards home (my goal is set to "home")
    if goal = work and (member? patch-here [neighbors4] of work) [
        stay set goal house
    ]
;; CHOICES is an agentset of the candidate patches which the car can
;; move to (white patches are roads, green and red patches are lights)
let choices neighbors with [pcolor = white or pcolor = red or pcolor = green]
          ;; choose the patch closest to the goal, this is the patch the car will move to
    let choice min-one-of choices [distance [goal] of myself]
;; report the chosen patch
    report choice
end
```

With this procedure in place, we can modify the GO procedure to ask each of the cars to now move between their homes and their work.

```
ask turtles
    [
;; head towards the patch that is closest to your goal
       face next-patch
        set-car-speed
        fd speed
        set-car-color
    ]
```

Now, we have goal-directed car agents in the grid. However, the agents' approach is rather naïve and ineffective, since agents are measuring the nearness to their goal by direct distance (as the bird flies) as opposed to the distance by following the road. Thus, the cars will often get stuck going back and forth on the same stretch of road, instead of going around a block. In the explorations at the end of this chapter, we challenge the reader to come up with more intelligent, sophisticated planning so that the agents can more reliably reach their goals.

One of the powerful advantages of agent-based modeling is that agents can change not only their decisions but also their strategies (Holland, 1996). An agent that can change its strategy based on prior experience is an *adaptive agent*. Unlike conventional agents, which will also do the same thing when presented with the same circumstance, adaptive agents can make different decisions if given the same set of inputs. In the Traffic Basic model, cars do not always take the same action; they operate differently based on the cars around them by either slowing down or speeding up. However, regardless of what has happened to the cars in the past (i.e., whether they got stuck in traffic jams or not) they will continue to take the same actions in the same conditions in the future. To be truly adaptive, agents need to be able to change not only their actions in time, but also their strategies. They must be able to change how they act because they have encountered a similar situation in the past and can react differently this time based on their past experience. In other words, the agents learn from their past experience and change their behavior in the future to account for this learning. For instance, the agent could have observed that in the past that, when there was a car five patches ahead of it, it braked too quickly, even though it could have waited longer to brake without hitting the other car. In the future, it could change its rule for braking to brake only when a car is four patches ahead. This agent is now an *adaptive agent* because it not only modifies its *actions*, but also, its *strategies*.

Another example of adaptive cognition in our traffic model would be to have the agents learn the best rate of acceleration to maintain the highest velocity. We can implement this form of agent cognition by changing the code in the GO procedure in Traffic Basic as follows. (To get this code to work we would also have to change the SETUP procedure and some global properties but that is left as an exercise for the reader.) This code is also in the Traffic Basic Adaptive model in the chapter 5 subfolder of the IABM Textbook folder of the NetLogo models library:

```
to adaptive-go
    ;; check to see if we should test a new value for acceleration this tick
    let testing? false
    if ticks mod ticks-between-exploration = 0 [
        set testing? true
;; choose new value for acceleration, slightly different from current acceleration
        set acceleration acceleration + (random-float 0.0010) - 0.0005
    ]

 ;; run the old go code
    go

;; check to see if our new speed of turtles is better than the previous speeds
;; if so, then adopt the new acceleration
    ifelse mean [ speed ] of turtles > best-speed-so-far and testing? [
        set best-acceleration-so-far acceleration
        set best-speed-so-far mean [ speed ] of turtles
    ]
    [
        set acceleration best-acceleration-so-far
    ]
    if not testing? [
;; you don't want to take one data point as a measure of the speed. Instead you
;; calculate a weighted average of past observed speed and the current speed.
       set best-speed-so-far (0.1 * mean [speed] of turtles) + (0.9 * best-speed-so-far)
    ]
end
```

Though this code may appear complicated, it is straightforward. Essentially, cars use the best acceleration they have found so far unless they are in a tick where they are exploring a new acceleration value, as specified by TICKS-BETWEEN-EXPLORATION. Over time, the cars keep a weighted average of the speed they are able to maintain at the best acceleration; if the new acceleration allows for a faster speed, the cars will then switch to using that new acceleration. This average for the best acceleration weighs the past historical speeds higher (0.9) than the present speed (0.1), accounting for the fact that you can occasionally get spurious results (noise). Thus, it is better to rely on a large amount of data than one particular data point. However, the code still allows the best acceleration to change, which means even if the environment were to change (e.g., more cars on the road, longer road, etc.) the car could adapt to the new situation.

If you run the Traffic Basic Adaptive model, you will notice the cars eventually stop forming traffic jams, though it may take them quite some time to arrive at the free-flow state. Eventually, the model "learns" that a high rate of acceleration results in the best speed overall. In this case, the model learns one acceleration value for all agents. Another change we could make to this model would be to allow the individual agents to learn different best accelerations for themselves. (This is left as an exercise for the reader.) Since

Box 5.4
Machine Learning

Machine learning is the study of computer algorithms that use previous experience to improve their performance. For example, in the 1980s, American Express had a procedure that would classify credit applications into three states: applications that (1) should be immediately approved, (2) should be immediately denied, and (3) need to be examined by a loan officer (Langley & Simon, 1995). However, it turned out that the loan officers were able to correctly determine whether the borderline applicants would default on their loan only 50 percent of the time. Michie (1989) used a machine-learning algorithm to examine these borderline cases. This algorithm was able to correctly classify 70 percent of these cases, and American Express decided to use this algorithm instead of human loan officers.

In this example, the algorithm was trained on a group of test cases. These test cases consisted of inputs based on the credit application and an output based on whether or not the applicant defaulted on the loan. The algorithm was then tested on cases that it had not been shown during the training period, with the algorithm correctly predicting 70 percent of these cases. Because the algorithm is continuously learning, the algorithm continues to improve itself with use. This is just one example of the many possible uses of machine learning. For a general overview, see Mitchell (1997).

the model now learns acceleration, another exercise would be to learn the deceleration values as well.

Besides the basic types of cognition that we have discussed, there are also more advanced methods of giving cognition to agents. One of the best ways to do this is by combining agent-based modeling with *machine learning*. Machine learning is an area of artificial intelligence that is concerned with giving computers the ability to adapt to the world around them and learn what actions to take in response to a given set of inputs. By giving agents the ability to use various machine-learning techniques such as neural nets, genetic algorithms, and Bayesian classifiers, they can change their actions to take into account new information (Rand, 2006; Rand & Stonedahl, 2007; Holland, 1996; Vohra & Wellman, 2007). This can result in agents with quite sophisticated levels of cognition.

Other Kinds of Agents

We have already discussed breeds and various types of agents, but there are two other special kinds of agents that deserve at least a brief mention. The first type are meta-agents, agents made up of other agents; the second is proto-agents, placeholder agents that allow you to define interactions for your fully defined agents with other entities that have not been fully developed.

Meta-Agents Many things that we would consider agents in reality are actually composed of other agents. In reality, all "agents" are composed of other agents; to cite the oft-told parable, "It's turtles all the way down."[5] In other words, there is always a lower level of detail that you can use to describe an agent in your model, at least until you reach the most basic level so far described by physics. For instance, if we have selected a human to be the agent, he or she is actually composed of many subagents, with these subagents in turn being different depending on how you view the human. You could view the subagents of a human as the systems of the body—e.g., the immune and the respiratory systems—or you could view the subagents of a human as psychological aspects like the intellect and emotion. Moreover, these subagents are not the most basic level; in our example, the systems of the body are made up of organs, tissues, and cells.

At each of these levels, we are describing the relationship between *meta-agents* (agents composed of other agents) and *subagents* (agents which compose other agents). Still, agents can also be both meta-agents and subagents at the same time. For instance, organs are composed of cells and are meta-agents. However, they also play a compositional role in the systems of the human body and are thus subagents. We can use meta-agents in our ABMs by defining the subagents that make up our agents and providing them with their own actions and properties.

From the perspective of another meta-agent, a meta-agent appears to be a single agent. If a person meets another person, they do not (usually) directly interface with the heart or the lungs (unless the meeting takes place at the surgical table). Instead, they interface with the person as a whole. The same is true in ABM. Suppose you have a meta-agent of a human that is composed of an intellect and an emotion. Everything that meta-agent "says" is the result of a dialogue between intellect and emotion. When another human meta-agent speaks to that meta-agent, the second meta-agent will only be aware of the utterance resulting from that dialogue, and not the individual factors that made it up.

When we think of modeling agents and their interactions, we have to determine what level of granularity we want to describe the agent behaviors. However, we always have the option of refining our model by converting agents into meta-agents that describe the agents that constitute other agents. Sometimes, it can be useful to represent the agents in our models not as autonomous individuals, but instead, as meta-agents composed of other agents. NetLogo doesn't include explicit language support for these meta-agents, though there are commands for locking together the movement of several agents (e.g., the TIE command) that may be useful in some circumstances. However, there is nothing to prevent you from designing models that have groups of agents representing single agents.

5. As the story goes, a visitor asks the wise man what keeps the earth from falling. The wise man says that the earth is held up on the back of a giant turtle. The visitor asks what is holding up the turtle, and the wise man replies, "Yet another turtle." The visitor smiles and asks, "What is holding up the first turtle?" The wise man responds, "It's turtles all the way down." The source of this quote is disputed, although William James seems the likeliest candidate.

Proto-Agents To truly be an agent within the agent-based modeling framework, an entity must have its own properties or actions. Sometimes, though, it can be desirable to create agents that, rather than having their own properties or behaviors, are instead placeholders for future agents. We call these agents *proto-agents*, and their primary purpose is to enable us to specify how other agents would interact with them if they were fleshed out into full agents. For instance, if you were creating a model of residential location decision making, you would have residents as agents; however, since where a resident lives is greatly influenced by places of employment and services (e.g., grocery stores and restaurants) you might also want to include these "service centers" as agents as well. The same level of detail necessary for the residents, who are the focal agents of concern, may not be necessary for the service centers. Instead, they might be rendered as placeholders to represent where residents might potentially find jobs and transact business. However, as you continue to refine the model, you might give the service centers additional decision-making abilities. For instance, they might have a more elaborate model of market demand and decide where to locate using their own properties and beliefs about the future growth of the world. Keeping these agents as proto-agents early on means that when you add in the more richly detailed versions to the model, you do not have to go back and revise your resident agents. Instead, you can use the interaction events that they had already been using to interface with the service center proto-agents. There are no special NetLogo commands to create proto-agents; rather, agents such as patches or turtles may serve the role of proto-agents depending on how they are treated in your model.

Both meta-agents and proto-agents provide a way to initially create a simpler model and late elaborate upon that model as we continue to refine it.

Environments

Another early and critical decision is how to design the *environment* of the agent-based model. The environment consists of the conditions and habitats surrounding the agents as they act and interact within the model. The environment can affect agent decisions, and, in turn, can be affected by agent decisions. For instance, in the Ants model from chapter 1, the ants leave pheromone in the environment that changes the environment, and in turn changes the behavior of the ants. There are many different kinds of environments that are common in ABMs. In this section, we will discuss a few of the most common types of environments.

Before we discuss the types of environments, it is important to mention that the environment itself can be implemented in a variety of ways. First, the environment can be composed of agents such that each individual piece of the environment can have a full set of properties and actions. In NetLogo, this is the default view of the environment—the environment is represented by the agentset of patches. This allows different parts of the environment to have different properties and act differently based on their local interactions. A second approach represents the environment as one large agent, with a global set of properties and actions. Yet another approach is to implement the environment outside of the ABM toolkit. For instance, it could be handled by a geographic information systems (GIS) toolkit, or it could be handled by a social network analysis (SNA) toolkit, and the ABM can interact with that environment. All of the "types" of environments we discuss in what follows are, in principle, independent of this implementation decision; still, some types may be easier to create using different implementations.

Spatial Environments

Spatial environments in agent-based models generally have two variants: discrete spaces and continuous spaces. In a mathematical representation, in continuous spaces between any pair of points, there exists another point, while in discrete spaces, though each point has a neighboring point, there do exist pairs of points without other points between them, so that each point is separated from every other point. However, when implemented in an ABM, all continuous spaces must be implemented as approximations, so that continuous spaces are represented as discrete spaces where the spaces between the points are very small. It should be noted that both discrete and continuous spaces can be either finite or infinite. In NetLogo, however, the standard implementation is a finite space, although it is possible to implement an infinite space (we explain how later in the chapter).

Discrete Spaces The most common discrete spaces used in ABM are lattice graphs (also sometimes referred to as mesh graphs or grid graphs), which are environments where every location in the environment is connected to other locations in a regular grid. For instance, every location in a toroidal square lattice has a neighboring location up, down, to the left, and to the right. As mentioned, the most common representation of the environment in NetLogo is *patches*, which are located on a 2D lattice underlying the world of the ABM (see figure 5.9 for a colorful pattern of patches whose code is simply ASK PATCHES [SET PCOLOR PXCOR * PYCOR]). This uniform connectivity makes them different from network-based environments, which we will discuss in more detail later in this chapter.

The two most common types of lattices are square and hexagonal lattices, which we will discuss in turn later. There is one other type of regular polygon that tiles the plane (i.e., covers the plane with no "holes"), namely the triangle, but the triangle is usually not as useful for representing environments as squares and hexagons are. In addition, there are eight other (semiregular) tilings involving combinations of triangles, squares, hexagons, octagons, and dodecagons (Branko & Shephard, 1987). However, since these tilings

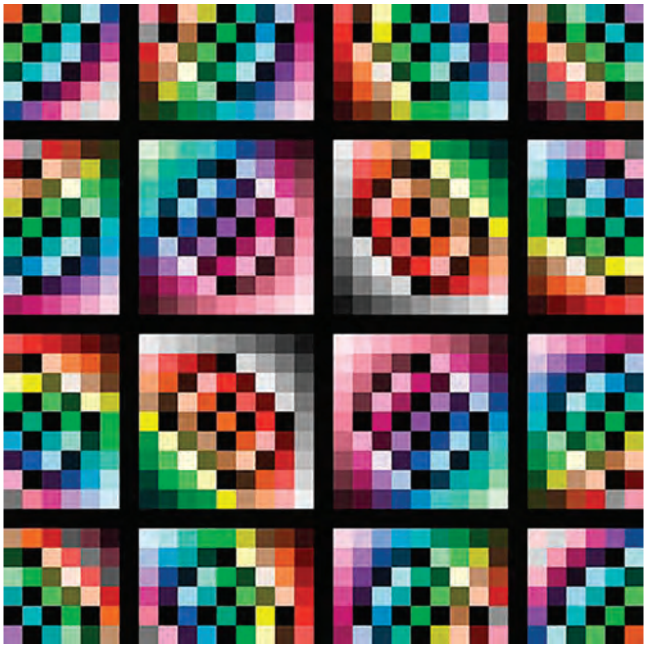


Figure 5.9
Patches displaying a range of colors.

have different cell shapes in different locations, they make the environment nonuniform; thus, they are not commonly employed in ABM, though nothing precludes their use.

Square Lattices The square lattice is the most common type of ABM environment. A square lattice is one composed of many little squares, akin to the grid paper used in mathematics classrooms. As we discussed in chapter 2, there are two classical types of neighborhoods on a square lattice: the *von Neumann neighborhood*, consisting of four neighbors located in the cardinal directions (see figure 5.10a); and the *Moore neighborhood*, comprising the 8 adjacent cells (See figure 5.10b). A von Neumann neighborhood (named after John von Neumann, a pioneer of cellular automata theory among other things) of radius 1 is a lattice where each cell has four neighbors: up, down, left, and right. A Moore neighborhood (named after Edward F. Moore, another pioneer of cellular automata theory) of radius 1 is a lattice in which each cell has eight neighbors in the eight directions that touch either a side or a corner: up, down, left, right, up-left, up-right, down-left, and down-right. In general, a Moore neighborhood gives you a better approximation to movement in a plane, and since many ABMs model phenomenon where planar movement is common, it is often the preferred modeling choice for discrete motion.

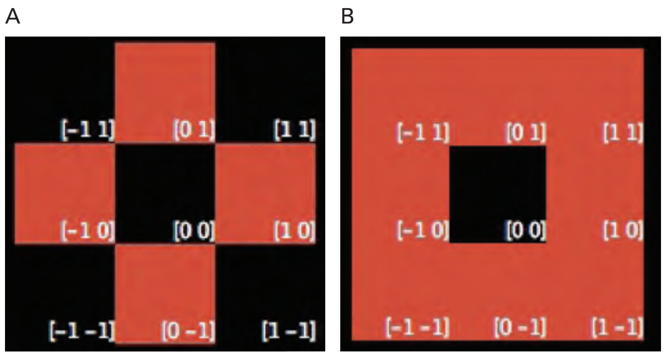


Figure 5.10
(a) Von Neumann Neighborhood. (b) Moore Neighborhood. The black cell in the center is the focal cell, and the red cells comprise its neighborhood.

These neighborhoods can be extended for neighborhoods with radii greater than 1; for example, a Moore neighborhood of radius 2 would have twenty-four neighbors, while a von Neumann neighborhood of radius 2 would have eight neighbors. An easy way to remember the difference between Moore and von Neumann neighborhoods is that a Moore neighborhood has “more” neighbors than a von Neumann neighborhood. We will discuss how neighborhoods affect interactions in the Interaction section later in this chapter.

Hex Lattices A hex lattice has some advantages over square lattices. The center of a cell in a square grid is farther from the centers of some of the adjacent cells (the diagonally adjacent ones) than other such cells. However, in a hex lattice, the distance between the center of a cell and all adjacent cells is the same. Moreover, hexagons are the polygons with the most edges that tile the plane, and for some applications, this makes them the best polygons to use. Both of these differences (equidistance between centers and the number of edges) mean that hex lattices more closely approximate a continuous plane than square lattices. But because square lattices match more closely a Cartesian coordinate system, a square lattice is a simpler structure to work with; even when a hexagonal lattice would be superior, many ABMs and ABM toolkits nevertheless employ square lattices. However, with a little effort, any modern ABM environment can simulate a hexagonal lattice in a square lattice environment. For instance, you can see a hex lattice environment in the NetLogo Code examples, Hex Cells example, and Hex Turtles example (see figures 5.11 and 5.12). In the Hex Cells example, each patch in the world maintains a set of six neighbors, with the agents located on each patch having a hexagonal shape. In other words, the world is still rectangular, but we have defined a new set of neighbors for each patch. In the Hex Turtles example, the turtles have an arrow shape and start along headings that

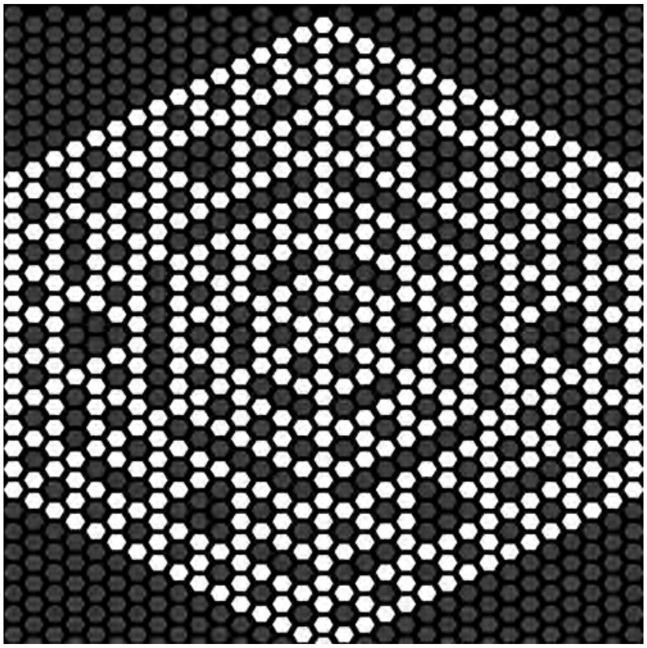


Figure 5.11
Hex Cells example from the NetLogo models library.

are evenly divisible by 60 degrees; when they move they only move along these same 60-degree angles.

Continuous Spaces In a continuous space, there is no notion of a cell or a discrete region within the space. Instead, agents within the space are located at points in the space. These points can be as small as the resolution of the number representation allows. In a continuous space, agents can move smoothly through the space, passing over any points in between their origin and destination, whereas in a discrete space, the agent moves directly from the center of one cell to another. Because computers are discrete machines, it is impossible to exactly represent a continuous space. We can, however, represent it at a level of resolution that is very fine. In other words, all ABMs that use continuous spaces are actually using a very detailed discrete space, while the resolution is usually high enough that to suffice for most purposes.

One of the interesting innovations of NetLogo absent in many other ABM toolkits is that the default space is a continuous space, with a discrete lattice laid on top of it. As a result, agents can move smoothly through the space, but they can also determine what

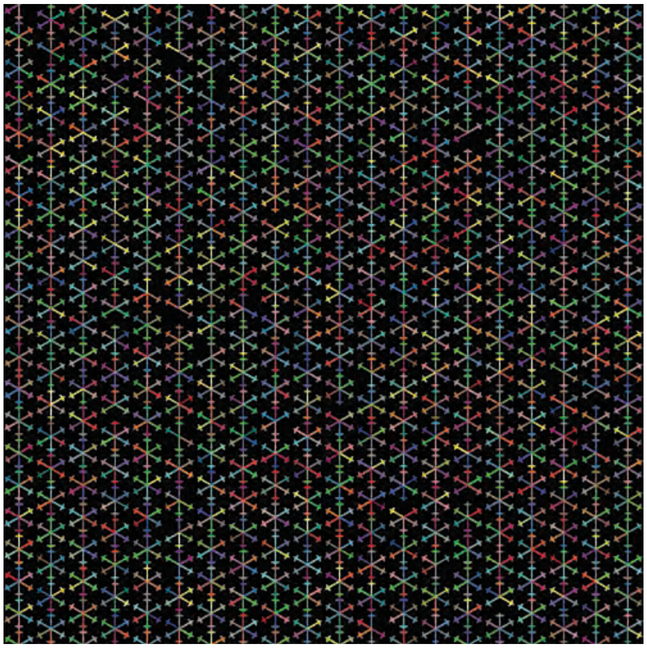


Figure 5.12
Hex Turtles example from the NetLogo models library.

cell they are in and interact with the lattice. To make this work, there has to be a mapping between the discrete space and the continuous space. In NetLogo, this is done by having the center of each patch be the integer coordinates of the patch, with each patch extending 0.5 units around this center. For instance a patch at –1, –5 has those coordinates at the center of the patch and includes every point from (–1.5, –5.5) to (–0.5, –5.5) to (–0.5, –4.5) to (–1.5, –4.5). However, this is not exactly true since the points along the edges can only belong to one patch: a patch at 0, –5, for example, includes the points along the edge from (–0.5, –5.5) to (–0.5, –4.5) including the points at the corners and the patch at –1, –4 includes the points along the edge from (–0.5, –4.5) to (–1.5, –4.5) including the points at the corner. In other words, a patch contains its bottom and left edges, but not its top and right edges.

NetLogo does not require that you specify whether you are using a continuous or discrete space ahead of time; model developers can write code that takes advantage of either spatial form. Many of the NetLogo sample models actually use both the discrete rectangular lattice and the continuous plane at the same time. For instance, the Traffic Basic model represents the cars as existing at points in the space but uses patch-ahead (i.e., the rectangular lattice) to determine if the car should speed up or slow down.

Box 5.5
Topologies

A topology is a class of environments with the same connectivity structure. For example, a sheet of paper represents the topology of a bounded plane. No matter how big that sheet of paper is, it still has fixed rectangular boundaries. A torus is the topology of a doughnut. Take a sheet of paper and roll it up. Now attach one open end to the other—you will get a doughnut-shaped (torus) topology. What makes topologies interesting is their connectivity properties and how these properties enable or block the movement of agents. When an agent on a bounded plane topology comes to the edge of the plane, there is nowhere else for it to go. By contrast, an agent on the torus topology encounters no edge to the world.

Boundary Conditions One other factor that comes into play when working with spatial environments is how to deal with boundaries, an issue for Hex and Square lattices as much as for continuous spaces. If an agent reaches a border on the far left side of the world and wants to go farther left, what happens? There are three standard approaches to this question, referred to as *topologies* of the environment: (1) it reappears on the far right side of the lattice (toroidal topology); (2) it cannot go any farther left (bounded topology); or (3) it can keep going left forever (infinite plane topology).

A *toroidal* topology is one in which all of the edges are connected to another edge in a regular manner. In a rectangular lattice, the left side of the world is connected to the right side, while the top of the world is connected to the bottom. Thus, when agents move off the world in one direction, they reappear at the opposite side. This is sometimes called *wrapping*, because the agents wrap off one side of the world and onto another. In general, using a toroidal topology means that the modeler can ignore boundary conditions, which usually makes model development easier. If the world is nontoroidal, then modelers have to develop special rules to handle what to do when an agent encounters a boundary in the world. In a spatial model, this conflict is one of whether the agent should turn around simply take a step backward. Indeed, the most commonly used environment for an ABM is a square toroidal lattice.

A *bounded* topology is one in which agents are not allowed to move beyond the edges of the world. This topology is a more realistic representation of some environments. For example, if you are modeling agricultural practices, it is unrealistic for a farmer to be able to keep driving the tractor east and then end up back on the west side of the field. Using a torus environment may affect the amount of fuel required to plow the fields, so a bounded topology might be a better choice depending on the questions the model is trying to address. It is also possible to have some of the limits of the world be bounded while others are wrapping. For instance, in the Traffic Basic model, where the cars only drive from left to right, the top and bottom of the world are bounded (cars do not go up and down in this

Box 5.6
Exploring Environments and Topologies

Find a phenomenon that is better modeled on a hex grid than on a square grid and vice versa. Implement a model of the phenomenon in both kinds of grids and demonstrate your results. Find a phenomenon to model for which each of the three topologies (bounded, toroidal, and infinite) works best. Choose one of these three phenomena and implement it in all three topologies. Compare and contrast your experiences in the three different topologies.

model) while the left and right are wrapped (in effect, a cylindrical topology). This gives the world the appearance of an infinitely long piece of road. In NetLogo, you can specify whether each boundary set (north–south or east–west) is bounded or wrapping in the Model Settings dialogue.

Finally, an *infinite-plane* topology is one where there are no bounds. In other words, agents can keep moving and moving in any direction forever. In practice, this is done by starting with a smaller world such that whenever agents move beyond the edges of the world, the world is expanded. At times, infinite planes can be useful if the agents truly need to move in a much larger world. While some ABM toolkits provide built-in support for infinite plane topologies, NetLogo does not. However, it is possible to work around this limitation by giving each turtle a separate pair of x and y coordinates from the built-in ones. Then, when an agent moves off the side of the world, we can hide the turtle and keep updating this additional set of coordinates until the agent moves back onto the view (see the Random Walk 360 coding example for an implementation of this). In most cases, however, a toroidal or bounded topology will be the more appropriate (and simpler) choice to implement.

Network-Based Environments

In many real-world situations, especially in social contexts, interactions between agents are not defined by physical geography. For instance, rumors do not spread between individuals in a strictly geographical manner. If I call a friend of mine in Germany and tell her a rumor, it spreads to Germany without passing through all of the people between Germany and me. In many cases, we want to represent the way individuals communicate by using a network-based environment (See figures 5.13 and 5.14). Using a network-based environment, we can represent the fact that I called my friend in Germany by drawing a link between the agents that represent each of us in our model. A *link* is defined by the two ends it connects, which are frequently referred to as *nodes*. These terms are used in the rapidly growing field of *network science* (Barabási, 2002; Newman, 2010; Watts & Strogatz, 1998). Note that mathematical graph theory literature uses different terminology

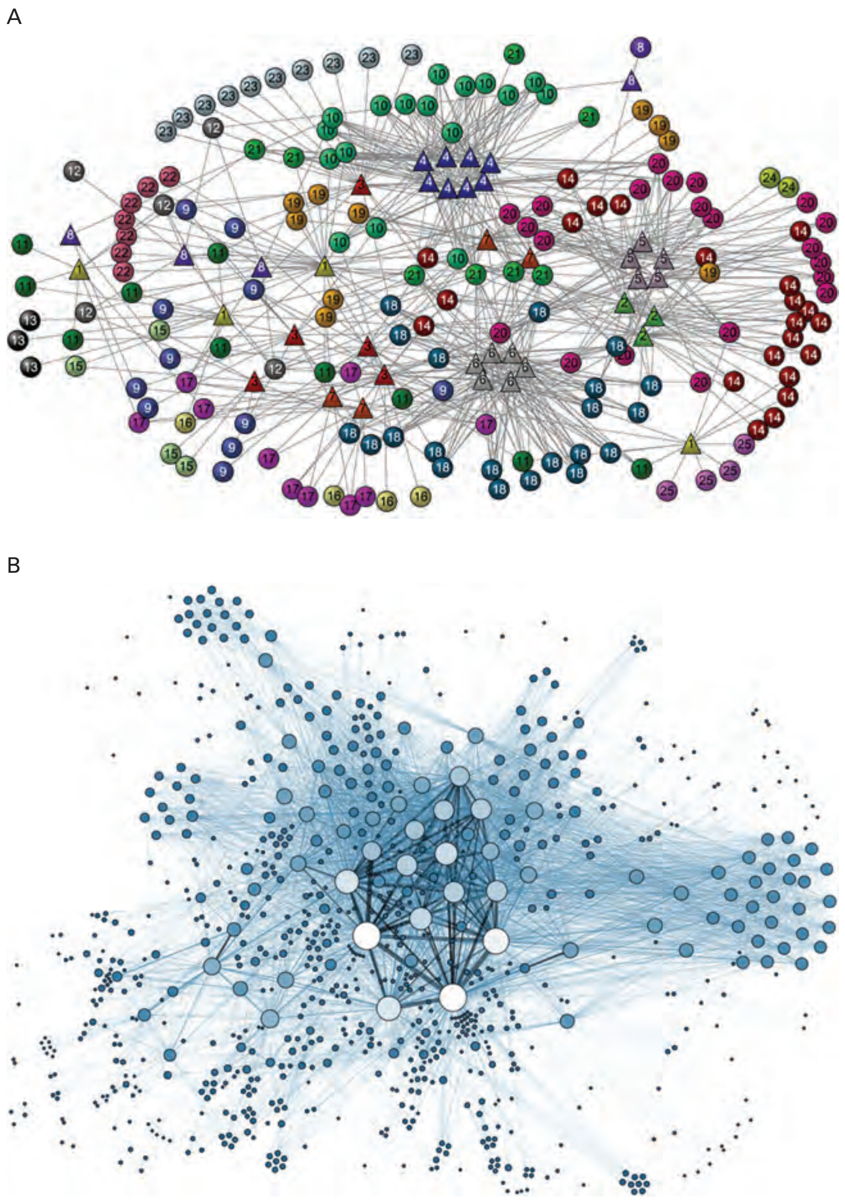


Figure 5.13
(A) Bipartite network of cancers and protein complexes. (B) Network of archive documents shared by League of Nations personnel.

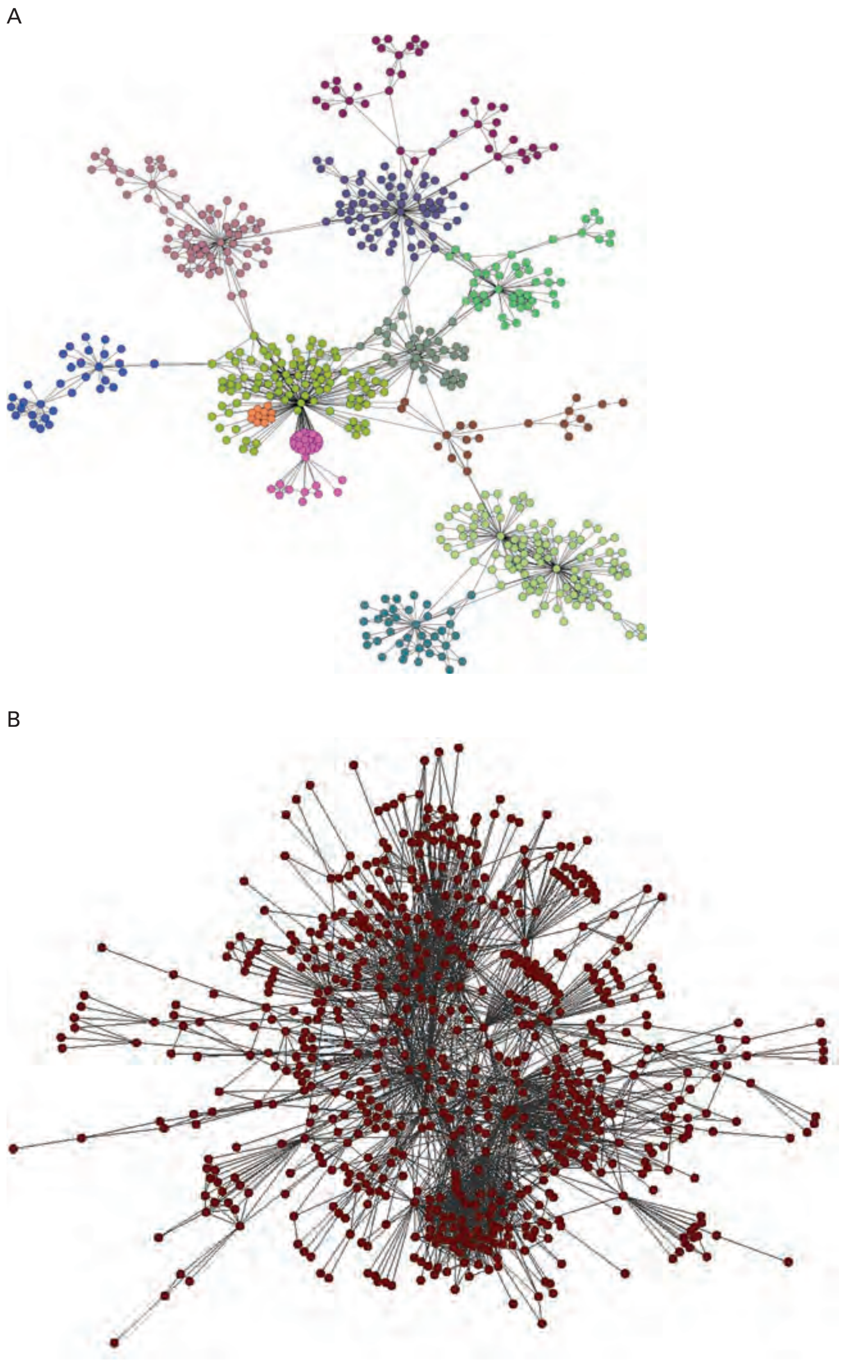


Figure 5.14
(A) Collaborations between researchers at McCormick School, Northwestern University. (B) Flights between cities in Asia and Middle East.

to refer to essentially these same objects, whereby a graph (network) consists of vertices (nodes) that are connected by edges (links). However, we will use the network/node/link vocabulary throughout this text.

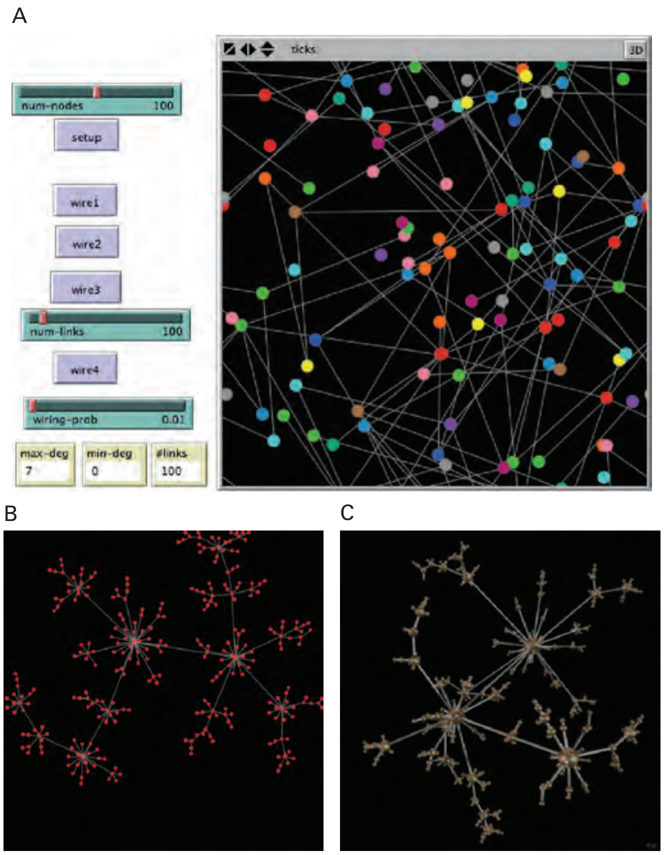
In NetLogo, links are their own agent-type. Much like patches, links can either be passive conduits of information and descriptions of the environment or else full-blown agents with properties and actions all their own. The lattice environments described earlier can be thought of as special cases of network environments, with patches being nodes connected to their grid neighbors. In fact, lattice graphs can also be called *lattice networks*, with the property that each position in the network looks exactly like every other position in the network. However, ABM environments usually do not implement lattice environments as networks for both conceptual and efficiency reasons. Additionally, using patches as the default topology allows for either discrete or continuous representations of the space, whereas a network is always discrete.

Network-based environments have been found useful in studying a wide variety of phenomena, such as the spread of disease or rumors, the formation of social groups, the structure of organizations, or even the structure of proteins. There are several network topologies that are commonly used in ABM. Besides the regular networks described earlier, the three most common network topologies are *random*, *scale-free*, and *small-world*.

In *random networks*, each individual is randomly connected to other individuals. These networks are created by randomly adding links between agents in the system. For example, if you had a model of agents moving around a large room and connected every agent in the room to another agent based on which agent had the next largest last two digits of their social security number, you would probably create a random network. The mathematicians Erdös and Rényi (1959) pioneered the study of random networks and described algorithms for generating them. We show one simple methods for creating a random network. This code is also in the Random Network model in the chapter 5 subfolder of the IABM Textbook folder of the NetLogo models library:

```
to setup
    ca
    crt 100 [
       setxy random-xcor random-ycor
    ]
end

to wire1
    ask turtles [
       create-link-with one-of other turtles
    ]
end
```

In this code, we create a group of turtles and place them randomly on the screen. We then ask each turtle to create a link to another turtle chosen randomly. If we want the



num-nodes 100
num-links 100
wiring-prob 0.01
max-deg 7
min-deg 0
#links 100

Figure 5.15
(A) Random Network model showing classic Erdös-Rényi random network. (B, C) Scale-Free Network from Preferential Attachment model. http://ccl.northwestern.edu/netlogo/models/PreferentialAttachment (Wilensky, 2001).

turtles to have more than one link we can ask each turtle to REPEAT this process several times. This code produces a network where each node has at least one link, meaning there are no isolates (i.e., nodes with no links). The Random network model shows several other ways to create random networks, including the classic Erdös-Rényi network. We will talk more about random networks in chapter 6.

Scale-free networks have the property that any subnetwork of the global network has the same properties as the global network. A common way to create this type of network is by adding new nodes and links to the system so that extant nodes with a large number of links are more likely to receive new connections (Barabási, 2002). This technique is sometimes called *preferential attachment*, since nodes with more connections are attached to preferentially. This method for network creation tends to produce networks with central nodes that have many radiating links; because of the resemblance to a bicycle wheel, this network structure is sometimes also called hub-and-spoke. Many real-world networks such as the Internet, electricity grids, and airline routes have similar properties to a scale-free network.

To create a scale-free network, we start by creating a couple of nodes and linking them. This code is also in the Preferential Attachment Simple model in the chapter 5 subfolder of the IABM Textbook folder of the NetLogo models library:

```
to setup
    ca
    set-default-shape turtles "circle"
    crt 2 [ fd 5 ]                              ;; create two turtles and space them out
    ask turtle 0 [ create-link-with turtle 1 ]  ;; create a link between them
end
```

This code clears the world and then changes the default shape of turtles to circles so that they look more like abstract nodes. After that, it creates two nodes and draws a link between them. Our GO procedure systematically adds nodes, one at a time, using the existing links to choose an endpoint. The code to add new nodes (up to the limit NUM-NODES) is:

```
to go
    if count turtles > num-nodes [stop]
    ;; choose a partner attached to a random link
    ;; this gives a node a chance to be a partner based on how many links it has
    let partner one-of [both-ends] of one-of links
    ;; create new node, link to partner
    crt 1 [ fd 5 create-link-with partner ]
tick
end
```

The crux of this code is determining a partner node that connects to the new node. One way to think of this code is that the partner node is determined by giving each node in the network a number of lottery tickets equal to the number of connections it has. Then a random ticket is drawn and whichever node has that lottery ticket is the partner node. This code is based on the Preferential Attachment model (see figure 5.15) in the Networks section of the NetLogo models library, which in turn is based on the Barabási-Albert network model (1999).

The final standard network topology is known as a *small-world network. Small-world* networks are made up of dense clusters of highly interconnected nodes joined by a few long distance links between them. Due to these long distance links, it does not take many links for information to travel between any two random nodes in the network. Small-world networks are sometimes created by starting with regular networks, like the 2D lattices described before, then randomly rewiring some of the connections to create large jumps between occasional agents (Watts & Strogatz, 1998). The example of a rumor spreading could be modeled using a small world, since much of it will happen in a local geography (i.e., friends in a nearby physical location), but there will be occasional long jumps (like a distant friend). For an example of how to create a small-world network, see the Small-World model in the Networks section of the NetLogo models library (Wilensky, 2005a).

There are many ways to characterize networks. Two commonly used ways are average path length and clustering coefficient. The *average path length* is the average of all the pairwise distances between nodes in a network. In other words, we measure the distance between every pair of nodes in the network and then average the results. Average path length characterizes how far the nodes are from each other in a network. The *clustering coefficient* of a network is the average fraction of a node's immediate neighbors who are also neighbors of the node's other neighbors. In other words, it is a measure of the fraction of my friends who are also friends with each other. In networks with a high average clustering coefficient, any two neighboring nodes tend to share many of their neighbors in common, while in networks with a low average clustering coefficient there is generally little overlap between groups of surrounding neighbors.

Random networks have low average path lengths and low clustering coefficients, indicating that that it does not take long to get from any particular node to any other node, since the nodes all have some connections to other nodes and there is no regularity to their connections. Completely regular networks, like the lattice-based environments, have a high average path length and a relatively high clustering coefficient. It takes a long time for information to travel around a regular network, but neighbors are very tightly connected. Small-world networks have a low average path length despite having a higher clustering coefficient. Neighbors tend to be tightly clustered, but since there are a few long-distance links, information can still flow quickly around the network. Scale-free networks also tend to have low average path lengths, since those nodes that have many neighbors serve as hubs for communication. We will present an example of creating and using network-based environments in chapter 6. NetLogo also includes a special extension, the network extension, for creating, analyzing, and working with networks. This extension enables full integration of network theory methods into ABM. We present uses of the network extension in chapter 8.

Special Environments

The two methods we have demonstrated for defining the environment, two-dimensional (2D) grid-based (i.e., lattice) and network-based, are both instances of "interaction topologies." Interaction topologies describe the paths along which agents can communicate and interact in a model. Besides the two interaction topologies we have looked at so far, there are several other standard topologies to consider. Two of the most interesting topologies involve the use of 3D worlds and Geographic Information Systems (GIS). 3D worlds allow agents to move in a third dimension as well as the two dimensions in traditional ABMs. GIS formats enable the importation of layers of real-world geographical data into ABMs. We will discuss each of these in turn.

3D Worlds

An unspeakable horror seized me. There was a darkness; then a dizzy, sickening sensation of sight that was not like seeing; I saw a Line that was no Line; Space that was not Space: I was myself, and not myself. When I could find voice, I shrieked loud in agony, "Either this is madness or it is Hell."

"It is neither," calmly replied the voice of the Sphere, "it is Knowledge; it is Three Dimensions: open your eye once again and try to look steadily."

—Edwin A. Abbott, Flatland: A Romance in Many Dimensions

The traditional ABM uses a 2D rectangular lattice of square patches. However, many of the systems we are interested in studying operate in 3D space. In many cases, it is okay to simplify the model to 2D space, because the systems we are working with only move in two dimensions, e.g., most humans cannot fly or the extra dimension of movement does not serve an important role in the model's outcomes. Sometimes, however, it can be important to incorporate a third dimension into our models. In general, 3D environments enable model developers to explore complex systems which are irreducibly bound up with a third dimension as well as sometimes increasing the apparent physical realism to their models.

There is a version of NetLogo called NetLogo 3D (it is a separate application in the NetLogo folder) that allows modelers to explore ABMs in three dimensions. There are many implementations of classic ABMs that have been developed for this environment in the NetLogo 3D models library (*Wilensky, 2000*). For instance, there is a three-dimensional version of the Percolation model that we discussed in chapter 3 (see figure 5.16).

Some elements of ABM remain the same in 3D, but others change. For instance, we can still use the concept of a square lattice in 3D to describe the environment, but we need to extend it in one more dimension, to a cubic lattice. Working with 3D worlds is not much different from working with 2D worlds. Although, using a 3D world does require additional data and commands. For instance, agents now have a Z-coordinate, and new commands are needed to manipulate this new degree of freedom.

In 3D, the orientation of an agent can no longer be described by its heading alone, we must also use *pitch* and *roll* to describe the orientation of the agent. If you think about an agent as an airplane, then the *pitch* of the agent is how far from horizontal the nose of the airplane is pointing. For instance, if the nose of the airplane is pointing straight up then the pitch is 90 degrees (see figure 5.17). Using the airplane metaphor the *roll* of an agent is how far the wings are from horizontal. For instance, if the wings are pointing up and down then the roll of the agent is 90 degrees (see figure 5.18). In many 3D systems,

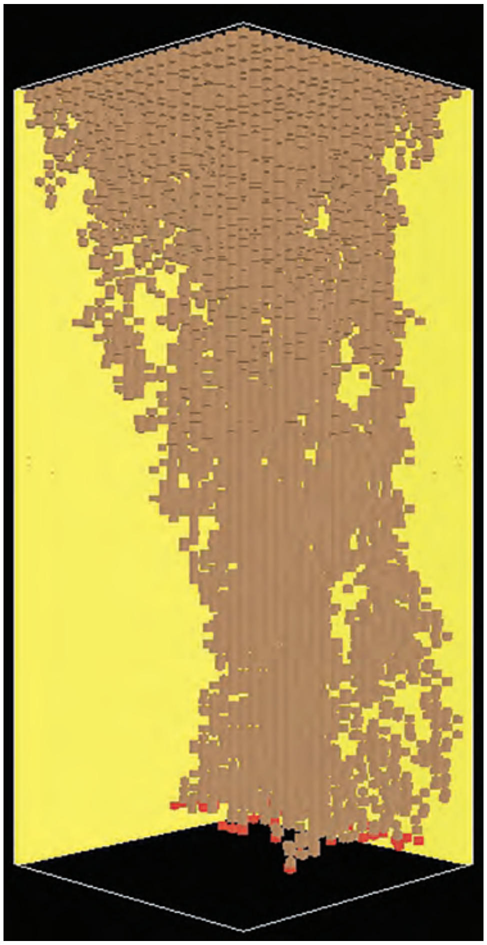


Figure 5.16
Percolation 3D. http://ccl.northwestern.edu/netlogo/models/Percolation3D (*Wilensky & Rand, 2006*).

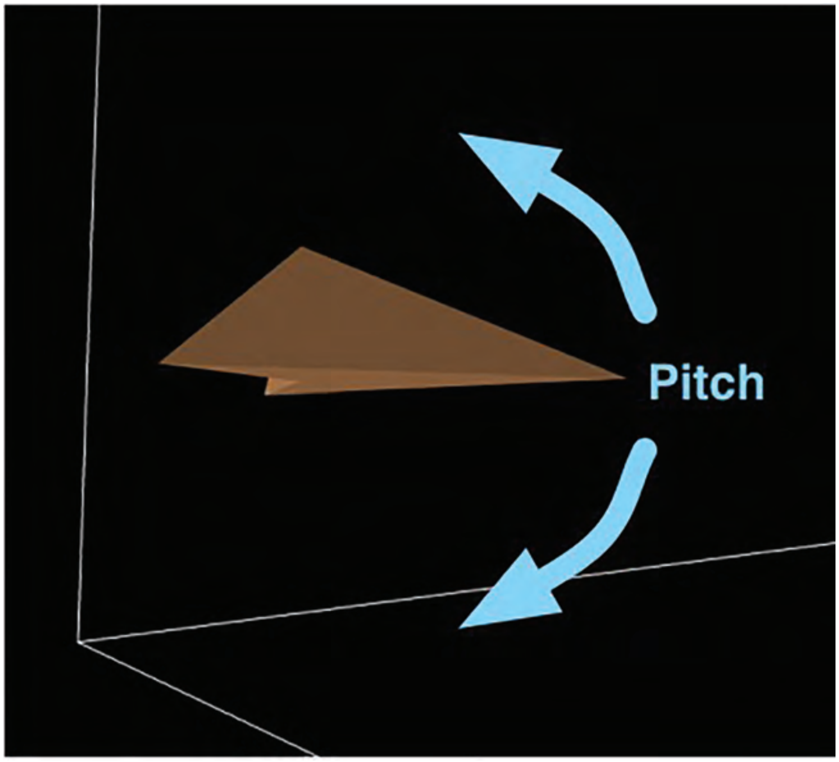


Figure 5.17
Pitch.

heading has a new name, which is *yaw*, but in NetLogo we still use HEADING regardless of whether you are in 2D or 3D NetLogo to keep things consistent.

To see the difference between writing 3D models and 2D models, let us examine the code in each model. The code for the PERCOLATE procedure in 2D Percolation looks like this:

```
to percolate
    ask current-row with [pcolor = red] [
       ;; oil percolates to the two patches southwest and southeast
       ask patches at-points [[-1 -1] [1 -1]]
           [ if (pcolor = brown) and (random-float 100 < porosity)
              [ set pcolor red ] ]
       set pcolor black
       set total-oil total-oil + 1
    ]
    ;; advance to the next row
    set current-row patch-set [patch-at 0 -1] of current-row
end
```

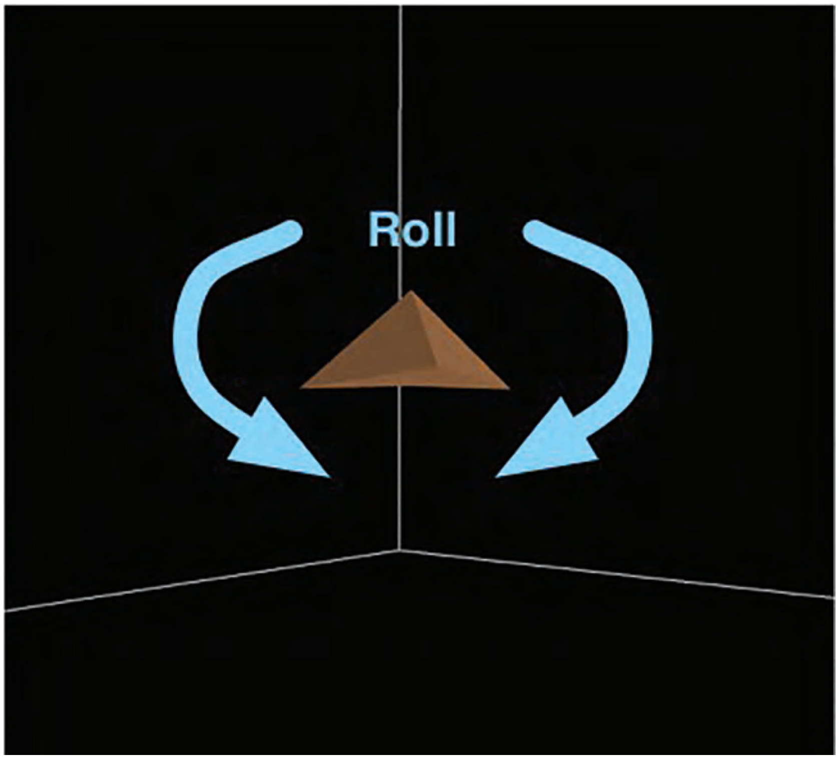


Figure 5.18
Roll.

Notice that in the 2D version, the code has each patch look at patches that are below, left, and right of the patch. This will change in the 3D version. In 3D Percolation (Wilensky & Rand, 2006), PERCOLATE looks like this:

```
to percolate
    ask current-row with [pcolor = red] [
       ;; oil percolates to the four patches one row down in the z-coordinate and
    ;; southwest, southeast, northeast, and northwest
       ask patches at-points [[-1 -1 -1] [1 -1 -1] [1 1 -1] [-1 1 -1]]
           [ if (pcolor = black) and (random-float 100 < porosity)
              [ set pcolor red ] ]
       set pcolor brown
       set total-oil total-oil + 1
    ]
    ;; advance to the next row
    set current-row patch-set [patch-at 0 0 -1] current-row
end
```

The only difference between the 3D percolate and its 2D counterpart is that instead of percolating a line of patches at a time, we percolate a square of patches. As a result, each patch must ask four patches below it (in a cross shape) to percolate, not just two as it does in the 2D version.

As in the preceding case, it can sometimes be very easy to translate a model from two dimensions into three dimensions. In many cases this adds a level of realism to the model. Since many phenomena that exhibit percolation, percolate in all three dimensions, like oil through rock, it is reasonable to use three dimensions to model percolation. However, in some situations a two-dimensional representation of the complex system may be an adequate representation (or even better than a 3D representation). In the end, the decision to include a third dimension or not is driven primarily by the question that the model developers are trying to answer. If a third dimension is necessary to properly address the question at hand, then it should be included. However, if a third dimension is not necessary then using a two-dimensional representation can expedite and facilitate model development. This question might not always be easy to answer. For instance, if a researcher is interested in examining land use and land change on a mountainside, it might at first be assumed that a 3D representation is necessary, since mountains change greatly in elevation and thus have a height as well as a width and length. But as long as the only aspect that is of concern is the surface coverage of the mountain, then there is no data of concern that is either above or below the current area of interest. We can instead model the change in elevation as a patch variable. In fact, this is exactly what is done in the Grand Canyon model in the Earth Science section of the models library.

In NetLogo the 3D view is a separate window, and provides controls to manipulate the point of view such as “orbit,” “zoom,” and “move,” which allow you to better visualize the model (see figure 5.19).

GIS-Based Geographic Information Systems (GIS) are environments that record large amounts of data that are related to physical locations in the world.[6] GISs are widely used by environmental scientists, urban planners, park managers, transportation engineers, and many others, and help to organize data and make decisions about any large area of land. Using GIS, we can index all the information about a particular subject or phenomenon by its location in the physical world. Moreover, GIS researchers have developed analysis tools that enable them to quickly examine the patterns of this data and its spatial distribution. As a result, GIS tools and techniques allow for a more in-depth exploration of the pattern of a complex system.

Agents moving on a GIS terrain may be constrained to interact on that terrain. Thus, GIS can serve as an interaction topology. However, GIS systems need not be solely

6. It is important to note that a GIS environment is, in fact, a spatial environment, and it can also contain within it both continuous and discrete spaces, but since the way GIS data is handled is very different from traditional mathematical spaces, we have decided to address it separately in this textbook.

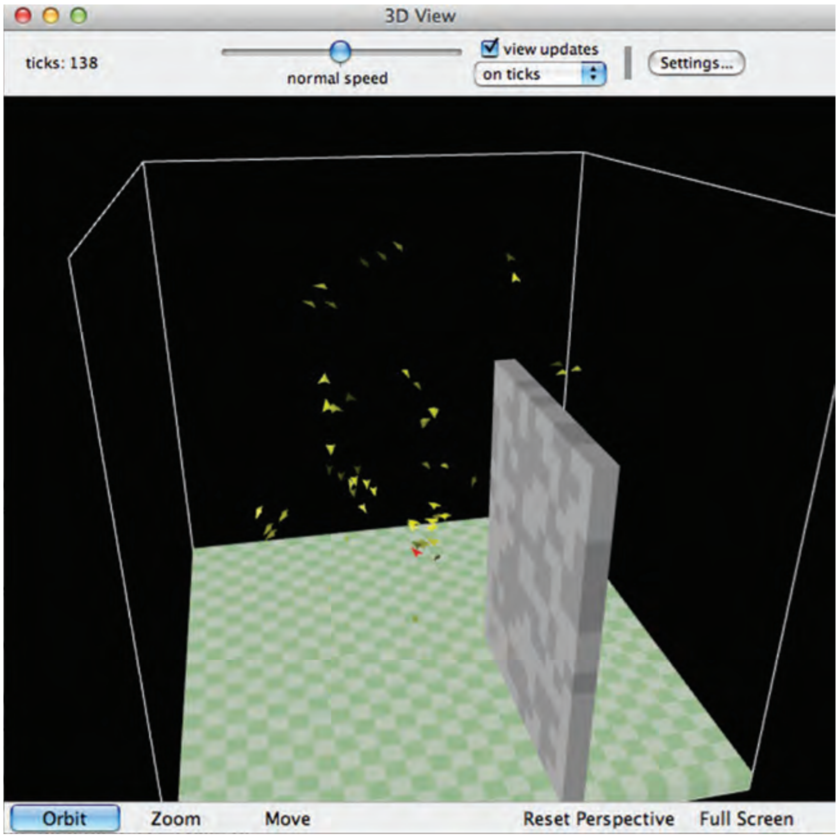


Figure 5.19
The 3D model, Flocking 3D Alternate, in which a flock of birds move around a wall. http://ccl.northwestern.edu/netlogo/models/Flocking3DAlternate (*Wilensky, 2005*).

employed for strictly geographical data such as elevation, land use, and surface cover. Whenever we have large amounts of data about a particular subject or phenomena, we can take advantage of the GIS to encode that data. For instance, we can embed the socioeconomic status of neighborhoods in a location related to that neighborhood.

Where does ABM fit in to this picture? GIS can provide an environment for an ABM to operate in. Since ABM encompasses a rich model of process of a complex system, it is a natural match for GIS, which has a rich model of pattern. By allowing ABM to examine and manipulate GIS data, we can build models that have a richer description of the complex system we are examining. GIS thus enables modelers to construct more realistic and elaborate models of complex phenomena.

To illustrate this, examine the Grand Canyon model in the NetLogo models library (see figure 5.20). This model shows how water drains in the Grand Canyon. We use a digital

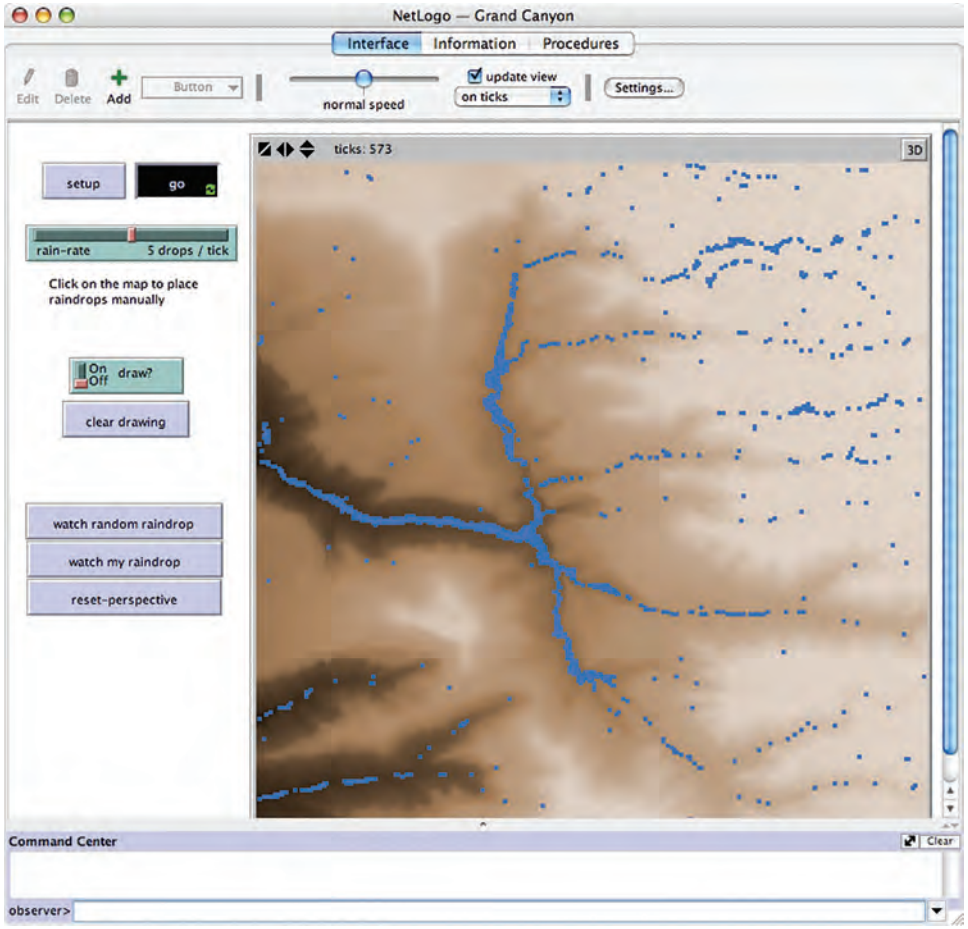


Figure 5.20
Grand Canyon model. http://ccl.northwestern.edu/netlogo/models/GrandCanyon (*Wilensky, 2006*).

elevation map of the Grand Canyon (created from a GIS topographical elevation dataset) to enable the user to visualize a simple model of a complex process (water drainage) in a real environment (the Grand Canyon).

How was the GIS data incorporated into this model? The first step was to examine the data in a GIS toolkit. This data was gathered from the National Elevation Dataset (http://seamless.usgs.gov). It was then converted from the original ESRI grid file format to an ASCII grid file. ESRI is a large producer of GIS software, and many datasets are found in this format. You can use one of many ESRI products, e.g., ArcGIS, to examine ESRI data. After changing the file format, the data file was rescaled to lie in the range 0 to 999, while the headers were stripped out of the file, allowing the data to be more easily managed by NetLogo. The end result is a file stored as one large list with 90,601 entries. This list represents 301 rows, with each row containing 301 elevation values. The first row looks like this:

```
819 820 822 828 830 832 834 835 836 837 839 841 842 844 845 846 847 848 849 848 847
844 840 833 829 826 825 825 826 827 828 830 832 835 838 841 841 842 842 843 844 845
847 849 850 850 852 855 858 862 864 866 865 864 864 870 873 876 878 880 880 880 880
879 878 877 877 879 879 881 884 887 888 888 887 883 881 880 879 873 870 866 864 860
860 861 860 856 853 852 853 852 851 850 848 845 843 841 840 836 834 833 833 834 835
835 836 838 838 839 840 843 845 847 848 853 856 859 863 871 874 876 879 884 887 890
894 902 905 908 910 912 912 912 912 911 908 906 906 901 897 897 900 904 909 913 918
919 919 916 914 915 915 914 912 913 911 907 904 899 899 903 906 911 913 915 917 922
923 921 918 907 903 904 908 910 911 911 910 909 912 915 918 918 918 920 921 919 918
918 919 921 922 924 924 926 929 931 932 935 938 940 942 944 945 946 947 947 945 942
943 949 950 952 953 956 957 958 959 960 960 960 959 957 956 955 955 956 956 957 957
959 960 961 962 963 964 964 963 963 963 962 960 954 951 949 948 950 954 959 963 965
966 966 967 968 969 970 971 972 973 974 974 975 975 975 975 975 975 974 975 976 976
977 978 979 980 981 981 981 981 981 981 981 982 982 983 985 987 988 989 991 992 993
993 990 990 992 993 994 993
```

This data file was saved as "Grand Canyon data.txt." The NetLogo model was created with a world of 301 x 301 patches. The data file is read into the NetLogo model using the following code:[7]

```
    file-open "Grand Canyon data.txt"
    let patch-elevations file-read
    file-close
    set color-max max patch-elevations + 200
;; put a little padding on the upper bound so we don't get too much
                  ;; white, and higher elevations have a little more variation.
    let min-elevation min patch-elevations
                  ;; adjust the color-min a little so patches don't end up black
    set color-min min-elevation - ((color-max - min-elevation) / 10)
                  ;; transfer the date from the file into the sorted patches
    ( foreach sort patches patch-elevations
       [ ask ?1 [ set elevation ?2 ] ] )
```

The first line opens the file. Then, a temporary variable, PATCH-ELEVATIONS, is created to store the data from the file as one big list, after which the file is closed. We then determine how to color the world by determining the minimum and maximum values of the elevations. Finally, we assign each patch of the world an elevation equal to the corresponding data point that was read in from the file. Later on in the code, this information is used to color the patches:

7. As a side note, this code is contained in the STARTUP procedure. STARTUP procedures are special NetLogo procedures that are run when the model is opened before any button is pressed.

Box 5.7
The Special Local Variable: ?

The Grand Canyon model code makes use of variables of the form ?1 and ?2. ?, ?1, ?2, and so on are special local variables in NetLogo that hold the current inputs to a reporter or command block for certain primitives. For example, in FOREACH, these variables are set to different values every time the FOREACH primitive goes through its loop, as shown below:

```
( foreach sort patches patch-elevations
       [ ask ?1 [ set elevation ?2 ] ] )
```

For each iteration, ?1 will be set to the next patch as sorted. The default patch sort sorts the patches such that the first item is in the upper left corner and then lists them left-to-right and top-to-bottom, with ?2 set to the next item in the list (PATCH-ELEVATIONS). Another example of these local variables is evidenced in the SORT-BY primitive, which takes a comparator reporter and a list and returns a sorted list. Consider two examples:

```
sort-by [?1 < ?2] [8 5 4 7 2 1]

sort-by [?1 < ?2] [8 5 4 7 2 1]
```

In both cases, ?1 is always the first item being compared;?2, the second. The result of the first sort-by is [1 2 4 5 7 8], whereas the second sort-by reports [8 7 5 4 2 1].

```
ask patches
       [ set pcolor scale-color brown elevation color-min color-max ]
```

SCALE-COLOR tells an agent to choose an appropriate shade of a color on the basis of a numerical value. In this case, we asked the patches to set their color to some shade of brown, on the basis of their elevation. This means that patches with lower elevations will be colored darker shades of brown, while patches with higher elevations will be colored lighter shades of brown.

This demonstrates one way to include a minimal amount of GIS data in to an agent-based model that will work with almost any agent-based modeling toolkit. One can also modify this data and export the results back to a GIS (see exploration 4). We will also discuss more advanced ways to integrate GIS and ABM in chapter 8, including the NetLogo GIS extension that is a standard part of the NetLogo package. Using the NetLogo

GIS extension is the preferred method of importing large amounts of GIS data into a NetLogo model.

Interactions

Now that we have discussed both agents and the environments in which they exist, we will look at how agents and environments interact. There are five basic classes of interactions that exist in ABMs: *agent-self*, *environment-self*, *agent-agent*, *environment-environment*, and finally, *agent-environment*. We will discuss each in turn along with some examples of these common interactions.

Agent-Self Interactions Agents do not always need to interact with other agents or the environment. In fact, a lot of agent interaction is done within the agent. For instance, most of the examples of advanced cognition that we discussed in the Agent section involve the agent interacting with itself. The agent considers its current state and decides what to do. One classic type of self-agent interaction that we used in chapter 4, have yet to discuss, is birth. *Birth* events are a typical event in ABMs where one agent creates another agent. Though we tend to discuss birth within the biological paradigm of physically giving birth, similar types of interactions exist in other domains, from social science (e.g., an organization can create another organization) to chemistry (e.g., the combination of two atoms can create a new molecule). Below is the birth routine that we used in chapter 4:

```
;; check to see if this agent has enough energy to reproduce
to reproduce
    if energy > 200 [
       set energy energy - 100  ;; reproduction costs energy to the parent
       hatch 1 [ set energy 100 ] ;; which is transferred to the offspring
   ]
end
```

1. As you can see, the agent considers its own state and, based on this state, decides whether or not to give birth to a new agent. It then manipulates its state, lowering its energy and creating the new agent. This is the typical way of having agents reproduce: they consider whether enough of a resource exists to "give birth" to an offspring and, if so, "hatch" one. Though our example above was one of a turtle creating another turtle, it is also possible to have a patch create a turtle (in NetLogo, this is accomplished with the command SPROUT). Notice that the environment (represented by patches) can create a new turtle, but not new patches for a new environmental area. This is because,

as the old saying goes, "The reason why land is valuable is because they're not making any more of it."

Of course, the opposite of birth is death, and we also had a death procedure in chapter 4. This is also a self-agent interaction in NetLogo. There is no "kill" command that directly causes another agent to die, but rather, another agent may ask a turtle to kill itself (die). (Note: there is also the CLEAR-TURTLES command, which will cause all turtles to die.) Following is the code that we used in chapter 4:

```
;; asks those agents with no energy to die
to check-if-dead
if energy < 0 [
       die
    ]
end
```

This is a fairly typical way of having agents die: If an agent does not have enough of a resource to go on living, they then remove themselves from the simulation.

In Traffic Basic, we have another kind of agent-self interaction where the agents decide what speed they should be traveling at:

```
ask turtles [
    let car-ahead one-of turtles-on patch-ahead 1
    ifelse car-ahead != nobody
        [ slow-down-car car-ahead ]
        ;; otherwise, speed up
        [ speed-up-car ]
;; don't slow down below speed minimum or speed up beyond speed limit
    if speed < speed-min [ set speed speed-min ]
    if speed > speed-limit [ set speed speed-limit ]
    fd speed ]
```

If we ignore the beginning section of this code (where the car senses the cars ahead) and the end of this code (where the car actually moves), all of the actions in between (where the car changes its speed) are agent-self interactions, since the car looks at its current speed and then changes that speed. This is another typical type of self-agent interaction, where an agent considers the resources it has at its disposal and then decides how to spend them.

Environment-Self Interactions Environment-self interactions are when areas of the environment alter or change themselves. For instance, they could change their internal state variables as a result of calculations. In chapter 4, the classic example of an environment-self interaction is when the grass regrows:

```
;; regrow the grass
to regrow-grass
    ask patches [
       set grass-amount grass-amount + grass-regrowth-rate
       if grass > 10 [
           set grass 10
                         ]
recolor-grass
    ]
end
```

Each patch is asked to examine its own state and increment the amount of grass it has, but if it has too much grass then it is set back to the maximum value it can contain. Finally the patch colors itself based on the amount of grass it contains.

Agent-Agent Interactions Interactions between two or more agents are usually the most important type of action within agent-based models. We saw a canonical example of agent-agent interactions in the Wolf Sheep Predation model when the wolves consume the sheep:

```
;; wolves eat sheep
to eat-sheep
    if any? sheep-here [ ;; if there are sheep here then eat one
       let target one-of sheep-here
       ask target [
           die
       ]
       ;; increase the energy by the parameter setting
       set energy energy + energy-gain-from-sheep
    ]
end
```

In this case, one agent is consuming another agent and taking its resources, whereby the wolf always eats the sheep. However, it is also possible to add competition or flight to this model, where the wolf gets a chance of eating the sheep and the sheep gets a chance to flee. Competition is another example of agent-agent interaction.

Traffic Basic features another typical kind of agent-agent interaction: the sensing of other agents by an agent. At the beginning of the GO loop

```
     ask turtles [
         let car-ahead one-of turtles-on patch-ahead 1
         ifelse car-ahead != nobody
             [ slow-down car-ahead]
…
```

We can see that the current car is sensing whether there are cars ahead of it. If there are, it then changes its speed to reflect that of the cars ahead of it. When developing or interacting with agent-based models, it can be tempting to anthropomorphize our agents. That is, we may assume that they will have the knowledge, properties, or behaviors that come naturally to the thing being modeled, but are not automatically present in the modeled agents. It is important to remind yourself that computational agents are very simple, in that the rules about how agents sense the world around them, and how to act in response to that information, must be spelled out completely. In addition to sensing other agents, agents can also sense the environment, a phenomenon we will discuss in a later section.

A final example of agent-agent interaction is communication. Agents can share information about their own state as well as that of the world around them. This type of interaction allows agents to gain information to which they might not have direct access. For instance, in the Traffic Basic model, the current car asks for a car ahead of it to communicate its speed. A more classic example of agent-agent communication is the Communication T-T Example model in the Code Examples section of the NetLogo models library. In this model, one turtle starts with a message and then communicates it to other turtles using this procedure:

```
;; the core procedure
to communicate ;; turtle procedure
    if any? other turtles-here with [message?]
         [ set message? true ]
end
```

The turtles communicate by choosing another local turtle with which to communicate. If the other turtle has a message, the current turtle copies that message. If the turtles are linked together over a network, we can then change the procedure so that the turtles communicate with others with which they are linked instead of just local turtles. This code is also in the Communication-T-T Network example model in the chapter 5 subfolder of the IABM Textbook folder in the NetLogo models library:

```
;; the core procedure
to communicate ;; turtle procedure
    if any? link-neighbors with [message?]
         [ set message? true ]
end
```

This allows turtle-turtle interaction over links. In this instance, the links serve more as part of the environment than as full agents. Communication is not the only type of interactions that can occur over links. Links can be used for many kinds of agent interactions.

Environment-Environment Interactions Interactions between different parts of the environment are probably the least commonly used type of interaction in agent-based models. However, there are some common uses of environment-environment interactions: one of these is diffusion. In the Ants model discussed in chapter 1, the ants place a pheromone in the environment, which is then diffused throughout the world via an environment-environment interaction. This interaction is contained in the following piece of code in the main GO procedure:

```
diffuse chemical (diffusion-rate / 100)
ask patches
    [ set chemical chemical * (100 - evaporation-rate) / 100
;; slowly evaporate chemical
       recolor-patch ]
```

The first part of this code is the only environment-environment interaction—the DIFFUSE command automatically spreads the chemical from each patch to the patches that immediately surround it. The second part of this code is actually an example of an environment-self interaction. Each of the patches loses some of its chemical over time to evaporation, changing its color to reflect how the chemical has changed over that period.

Agent-Environment Interactions Agent-environment interactions occur when the agent manipulates or examines part of the world in which it exists, or when the environment in some way alters or observes the agent. A common type of agent-environment interaction involves agents observing the environment. The Ants model demonstrates this kind of interaction when the ants examine the environment to look for food and sense pheromone:

```
to look-for-food ;; turtle procedure
    if food > 0
    [ set color orange + 1      ;; pick up food
        set food food - 1                 ;; and reduce the food source
        rt 180                                                   ;; and turn around
    stop ]
    ;; face in the direction where the chemical smell is strongest
    if (chemical >= 0.05) and (chemical < 2)
    [ uphill-chemical ]
end
```

In the Ants model, the patches contain food and chemical, so the first part of this code checks to see if there is any food in the current patch. If there is food, the ant then picks up the food and turns around back to the nest, and the procedure stops. Otherwise, the ant checks to see if there is chemical, wherein it follows the chemical in that direction.

Another common type of agent-environment interaction is agent movement. In some ways, movement is simply an agent-self interaction, since it only alters the current agent's state. But since a major property of any given area in the environment is the agents contained in it, movement is also a form of agent-environment interaction. Depending on the topology of the world, agent movement will have differing effects on the environment.

Consider the following two types of movement.

In the Ants model the ants move around by "wiggling":

```
to wiggle ;; turtle procedure
    rt random 40
    lt random 40
    if not can-move? 1 [ rt 180 ]
end
```

Note that in the last line of this procedure, the ant checks to see if it has reached the edge of the world. If it has, it turns back around and heads in the other direction.

You can also have environment interactions where an agent goes off one of the edges of the world but comes back on the other edge. In the Traffic Basic model, the topology "wraps" horizontally. As a result, the cars continuously move in a straight line (or, viewed as a torus, on a circular track).

We have reviewed here the five basic different types of interactions: agent-self, environment-self, agent-agent, environment-environment, and agent-environment. While there are many other different examples of these types of interactions, we have covered here some examples of their most common applications. More of these will be covered in the explorations at the end of this chapter.

Observer/User Interface

Now that we have talked about the agents, the environments, and the interactions that occur between agents and environmental attributes, we may discuss who controls the running of the model. The *observer* is a high-level agent that is responsible for ensuring that the model runs and proceeds according to the steps developed by the model author.[8] The observer issues commands to agents and the environment, telling them to manipulate their data or to take certain actions. Most of the control that model developers have with an ABM is mediated through the observer. However, the observer is a special agent. It does not have many properties, though it can access global properties like any agent or patch can. The only properties that one could consider to be specific to the *observer* are those relating to

8. In NetLogo, this agent is called the Observer; in other ABM toolkits, it goes by other names, such as *Modeler* or *Controller*.

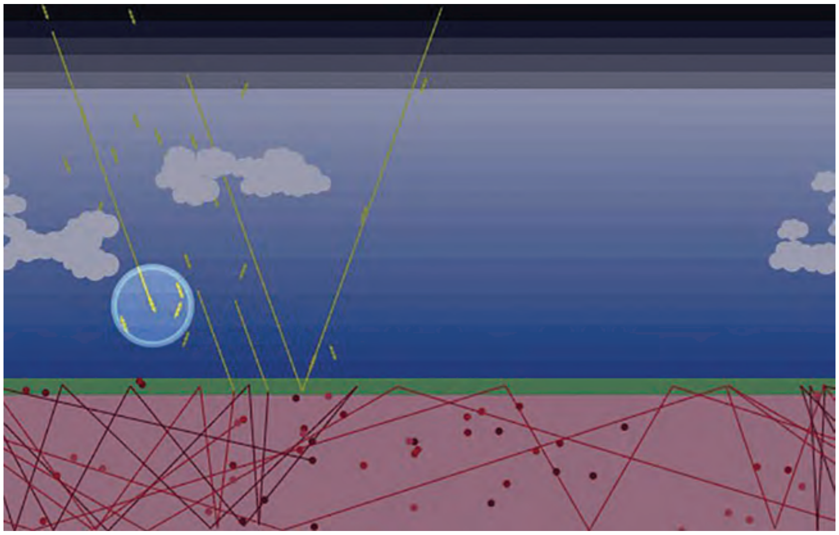


Figure 5.21
Climate Change model (with a sun ray turtle being watched, which is shown by the transparent "halo" around it). http://ccl.northwestern.edu/netlogo/models/ClimateChange (*Tinker & Wilensky, 2007*).

the perspective from which the modeled world is viewed. For instance, in NetLogo the view may be centered on a specific agent, or focusing a highlight on a certain agent, using the FOLLOW, WATCH, or RIDE commands (See figure 5.21).

The 2D NetLogo world can also be viewed from a 3D perspective by clicking the 3D button in the top right corner of the view control strip. In NetLogo 3D models, the observer's vantage point can also be manipulated (using the commands FACE, FACEXYZ, and SETXYZ, or with the 3D controls), and the world can be seen from a particular turtle's eye view, rather than the usual bird's-eye view. (Other than that, the observer's basic actions are asking agents to do things, and manipulating data and properties.)

In NetLogo there are observer buttons and agent buttons. Observer buttons tell the observer to do something. For instance, we can create a SETUP button and place the following code in it:

```
create-turtles100 [ setxy random-xcor random-ycor ]
```

Only the observer can run this code. We cannot ask a turtle to create turtles using the CREATE-TURTLES primitive (though we could use HATCH to achieve the same effect). If we then wanted the turtles to do something, we could create a turtle button (by selecting turtle in the drop-down box in the edit button dialog), with the following code:

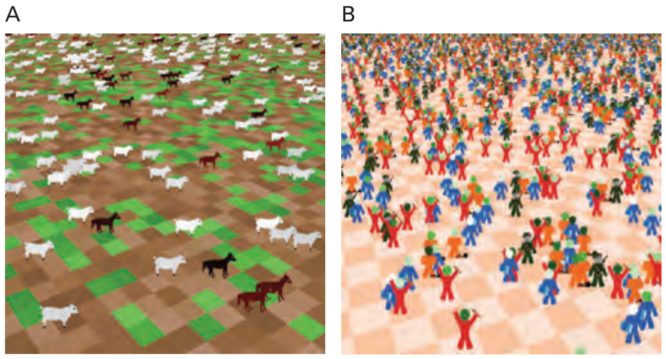


Figure 5.22
3D views of 2D models. (A) Wolf Sheep Predation model. (B) Rebellion model.

```
fd random 5 rt random 90
```

We can also create an observer button that does the same thing, but if we put the exact same code in we will get an error that only turtles can execute FD. So we have to tell the observer to ask the turtles to do something:

```
ask turtles [
fd random 5 rt random 90
]
```

Thus the observer plays the role of a general supervisor for the model. Usually when building models we interact with the model through the observer.

User Input and Model Output We have already made use of many of the standard ways to interface with an ABM when we extended models in chapter 3 and when we built our first model in chapter 4, but it is worth recapping some of them herein. ABMs require a control interface or a parameter group that allows the user to setup different parameters and settings for the ABM. The most common control mechanism is a *button*, which executes one or more commands within the model; if it is a *forever button* it will continue to execute those commands until the user presses the button again. A second way that the user can request that actions be performed within the ABM is via the command center (along with the mini–command centers within agent monitors). The command center is a very useful feature of NetLogo, as it allows the user to interactively test out commands, and manipulate agents and the environment.

The other interface controls that are usually provided to the model user are data-driven as opposed to action-driven. Among data-driven interface controls, one can differentiate input controls and output controls. The input controls include sliders, switches, choosers, and input boxes. The output controls consist of monitors, plots, an output area, and notes. Though these names are specific to NetLogo similar controls exist in most ABM toolkits. In NetLogo's interface, buttons are colored blue-gray, input controls are green, and output controls are khaki-colored.

Sliders enable the model user to select a particular value from a range of numerical values. For instance, a slider could range from 0 to 50 (by increments of 0,1) or 1 to 1,000 by increments of 1. In the Code tab the value of a slider is accessed as if it were a global variable. *Switches* enable the user to turn various elements of a model off or on. In the Code tab, they are also accessed as global variables, but they are Boolean variables. *Choosers* enable a model user to select a choice from a predefined drop-down menu that the modeler has created. Again these are accessed as global variables in the Code tab, but they are variables that have strings as their values, these strings being the various choices in the chooser. Finally, *input boxes* are more free-form, allowing the user to input text that the model can use.

As for output controls, *monitors* display the value of a global variable or calculation updated several times a second. They have no history but show the user the current state of the system. *Plots* provide traditional 2D graphs enabling the user to observe the change of an output variable over time. *Output boxes* enable the modeler to create free-form text-based output to send to the user. Finally, *notes* enable the modeler to place text information on the Interface tab (for example, to give a model user directions on how to use the model). Unlike in monitors, the text in notes is unchanging (unless you manually edit them).

These methods among many other ways of creating output will be discussed in more depth in chapter 6 where we will discuss how to analyze ABMs. However, one last method should be mentioned before we move on. Besides these direct manipulation methods of interfacing with an ABM, there are also file-based methods. For instance, you can write code to read data in from a file. This allows the user to modify that file to change the input to the model. Similarly, besides the traditional output methods within NetLogo, the modeler can also output data to a file. This is often useful because it creates a historical trace of the model run that does not disappear even after NetLogo has been closed. Moreover, using tools or analysis packages like Excel and R, it is possible to aggregate this file data into summary statistics. This will also be discussed in more depth in chapter 6.

Visualization Visualization is the part of model design concerned with how to present the data contained in the model in a visual way. Creating cognitively efficient and aesthetic visualizations can make it much easier for model authors and users to understand the model. Though there is a long history of work on how to present data in static images (Bertin, 1967; Tufte, 1983, 1996), there is much less work on how to represent data in real-time dynamic situations. However, attempts have been made to take current static guidelines and apply them to dynamic visualizations (Kornhauser, Rand & Wilensky, 2007). In general, there are three guidelines that should be kept in mind whenever designing the visualization of an ABM: simplify, explain, and emphasize.

Simplify the visualization Make the visualization as simple as possible so that anything that does not present additional usable information (or that is irrelevant to the current point being explained) has been eliminated from the visualization. This prevents the model user from being distracted by unnecessary "graph clutter" (Tufte, 1983).

Explain the components If there is an aspect of the visualization that is not immediately obvious then there should be some quick way to determine what that visualization is illustrating, such as a legend or description. Without clear and direct descriptions of what is going on the model user may misinterpret what the model author is attempting to portray. If a model is to be useful it is necessary that anyone viewing the model can easily understand what it is saying.

Emphasize the main point Model visualizations are themselves simplifications of all the possible data that a model could present to the model user. Therefore, a model visualization should emphasize the main points and interactions that the model author wants to explore, and, in turn, to communicate to end users. By exaggerating certain aspects of the visualization they can draw attention to these key results.

Model visualization is often overlooked, but a good visualization can make a model much easier to understand, and can provide a visceral appeal to model users who are thus more likely to enjoy working with the model. So how is good visualization actually accomplished? A full description would be beyond the scope of this textbook but a good place to start is with shapes and colors. Every NetLogo agent has a shape and color, and by selecting appropriate shapes and colors we can highlight some agents while backgrounding others. For instance, to simplify visualization if agents in our model are truly homogenous, like the ants, then we might make all the agents the same color, as in the Ants model. To explain the visualization we might change their shape to indicate something about their properties. For instance, in the Ethnocentrism model based on Hammond and Axelrod's model (2003) agents employing the same strategies have the same shape, even if they are different colors. Finally, to emphasize a point, we can use color as well as other tools. For instance, in the Traffic Basic model we have one car that is both red and haloed, in order to highlight that car. If this car were blue and not haloed, it would be hard to pick out from the rest of the cars, and thus it would be hard to figure out what a typical car was doing. Investing effort and attention to detail will be rewarded when designing a model's user interface.

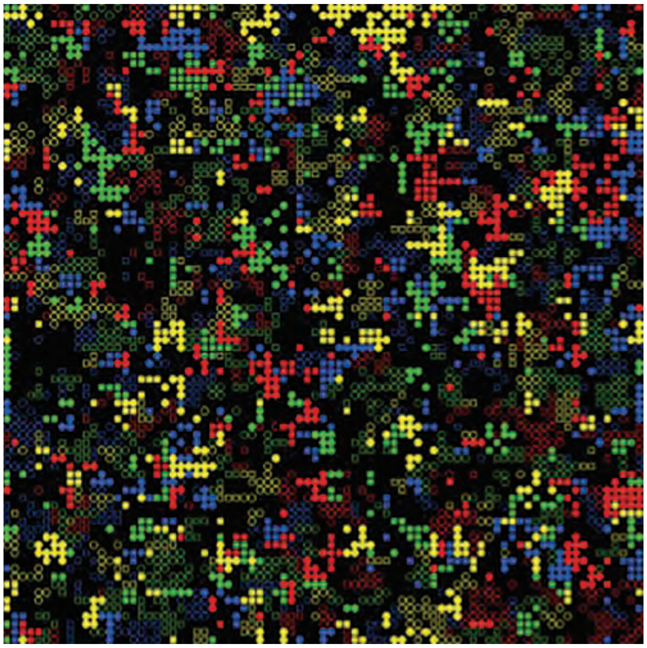


Figure 5.23
NetLogo Ethnocentrism model. This model uses the shape of agents to visualize agent strategy. http://ccl.northwestern.edu/netlogo/models/ethnocentrism (*Wilensky, 2003*).

Batch vs. Interactive When you open a blank NetLogo model, it starts with a command center that says "Observer." This blank window allows you to manipulate NetLogo in an interactive manner. If you open a model, you still have to press SETUP and GO to make the model work. Moreover, with most models, even when they are running, you can manipulate sliders and settings to see how these new parameters affect the performance of a model even during the middle of a run. This kind of spur-of-the-moment control is called *interactive running*, because the user can interact as the model is running. When designing an ABM, it is important to think about how a model user is going to interact with your model. For instance, can they manipulate all of the parameters of the model while it is running? Or do you want to only allow them to manipulate a subset of the parameters? In the latter case, it is useful to try and indicate that to the model user? For example, in NetLogo it is a convention to put any controls the user is going to initialize before the model runs (and leave unchanged during the model run) above the SETUP button. Controls that the user can manipulate while the model is running are placed below the SETUP button. Regardless of the solution used, modelers should be mindful of these considerations when developing their model.

In contrast to interactive running, another type of user running of a model is called *batch running*. With batch running, instead of controlling the model directly, the user writes a script to run the model many times, usually with different seeds for the pseudo-random number generator and with different parameter sets. This allows the user to conduct experiments on the model they are running and to collect results about multiple runs of the same model under different conditions. NetLogo provides two typical ways to conduct batch runs: (1) BehaviorSpace and (2) NetLogo's Controlling API. BehaviorSpace, which we will explore more in chapter 6, is an interactive tool that allows a model user to specify initial conditions and parameter sweeps for a model. BehaviorSpace can run either from the NetLogo user interface or in a *headless* mode without any graphics (e.g., from a command line interface). NetLogo's Controlling API is a method of interacting with NetLogo that involves writing Java code (or another JVM-compatible language) to control the NetLogo model. The Controlling API can either control the NetLogo GUI, or it can run headless as well. More details about headless running and NetLogo's Controlling API are beyond the scope of this book, but additional information can be found in the NetLogo documentation. However, when designing the user interface for your model, it is important to keep in mind that some users are going to want to be able to interact with your model via batch running. Usually this means that you should build your model so that the model clears the world every time it is run and does not rely on any past information.

Schedule

The *schedule* is a description of the order in which the model operates. Different ABM toolkits can have more or less explicit representations of the schedule. In NetLogo, there is no single identifiable object that can be identified as "the schedule." Rather, the schedule is the order of events that occur within the model, which depends on the sequence of buttons that the user pushes, and the code/procedures that those buttons run. We will first discuss the common SETUP/GO idiom which is employed in almost all agent-based models, and then move on to discuss some of the subtler issues concerning scheduling in ABMs.

SETUP and GO First of all, there is usually an initialization procedure that creates the agents, initializes the environment, and readies the user interface. In NetLogo this procedure is usually called SETUP and it executes whenever a user presses the SETUP button on a NetLogo model. The SETUP routine usually starts by clearing away all the agents and data related to the pervious run of the model. Then it examines how the user has manipulated the various variables controlled by the user interface creating new agents and data to reflect the new run of the model. For instance, in Traffic Basic the SETUP procedure looks like this:

```
to setup
    clear-all
    ask patches [ setup-road ]
    setup-cars
    watch sample-car
end
```

As is typical of many ABMs, this procedure calls a bunch of other procedures. It starts by clearing the world (CA), and then asks the patches to SETUP-ROAD, which creates the environment for the model. Afterward, it creates the cars using SETUP-CARS, which checks the value of the NUMBER-OF-CARS slider and uses that to determine how many cars will be created. Back in the SETUP procedure, the WATCH call tells the observer to highlight one particular car.

The other main part of the schedule is what is often called the main loop, or in NetLogo, the GO procedure. The GO procedure describes what happens in one time unit (or tick) of the model. Usually that involves the agents being told what to do, the environment changing if necessary, and the user interface updating to reflect what has happened. In Traffic Basic the GO procedure looks like this:

```
to go
;; if there is a car right ahead of you, slow down to a speed below its speed
    ask turtles [
         let car-ahead one-of turtles-on patch-ahead 1
         ifelse car-ahead != nobody
             [   slow-down-car car-ahead]
             ;; otherwise, speed up
             [ speed-up-car ]
    ;; don't slow down below speed minimum or speed up beyond speed limit
         if speed < speed-min [ set speed speed-min ]
         if speed > speed-limit [ set speed speed-limit ]
         fd speed ]
    tick
end
```

In this procedure, the agents change their speeds and move, and then the tick counter is advanced, which allows all of the components of the model to know that a tick has passed. In this model, the environment remains constant, but it is possible for potholes and repairs to be made to the road, which would cause another call to be included in the GO procedure.

SETUP and GO provide a high level view at the schedule of a NetLogo model. However, to get a full description of the schedule, it would also be necessary to examine the procedures that are called within SETUP and GO.

Two issues must be considered when thinking about the schedule of an ABM. The first is whether the ABM uses synchronous updates (all agents updating at the same time) or asynchronous updates (some agents updating before others). Relatedly, one must determine whether the model operates sequentially (agents take turns acting), in parallel (agents operate at the same time) or in a simulated concurrency (somewhere between sequential and concurrent). We will examine these issues in turn.

Asynchronous vs. Synchronous Updates If a model uses an *Asynchronous* update schedule, this means that when agents change their state, that state is immediately seen by other agents. In a *Synchronous* update schedule changes made to an agent are not seen by other agents until the next clock tick—that is, all agents update simultaneously. Both forms of updating are commonly used in agent-based modeling. Asynchronous updates can be more realistic, since asynchrony is more like the real world where agents all act and update independently of each other rather than waiting for each other. The Traffic Basic, Wolf Sheep Predation, Ants, Segregation, and Virus models all use asynchronous updating. However, synchronous updates can be easier to manage and debug and are therefore commonly used. In the NetLogo models library, Fire, Ethnocentrism, and the Cellular Automata models are examples of models that use synchronous updating. The form of updating can make quite a bit of difference to the behavior of the model. For instance, in a synchronous update, the order in which agents take actions does not matter, because they are only affected by the state of other agents at the end of the last tick, rather than the most current state of the agents. However, in asynchronous updating, it is important to know in what order and how agents take actions, which is what we will discuss in the next paragraph.

Sequential vs. Parallel Actions Within the realm of asynchronous updating, agents can act either sequentially or in parallel. *Sequential* actions involve only one agent acting at a time while *Parallel* actions are those in which all agents act independently. In NetLogo (versions 4.0 and later), sequential action is the standard behavior for agents. In other words, when you ASK agents to do something, one agent completes the full set of actions that you have requested before passing on control to the next agent. In some ways, this is not a very realistic model of actions since in reality agents are constantly acting, thinking, and affecting other agents all at the same time and they do not take turns to wait for each other. Still, sequential actions are easier to design from a model implementer's viewpoint because it is more difficult to understand how parallel actions interact. Moreover, for the agents to act truly in parallel, you would need parallel hardware so that the actions of each agent would be carried out by a separate processor.

However, there is an intermediate solution. *Simulated concurrency* uses one processor to simulate many agents acting in parallel. Very few agent-based modeling toolkits support simulated concurrency, although NetLogo does have limited support for it. One way to take advantage of this is through the use of "turtle forever" buttons. With a turtle forever button, each agent acts completely independently of the others, and the observer is not involved at all, For example, examine the Termites model (*Wilensky, 1997c*) in the Biology section of the NetLogo models library. In this case, all of the turtles are executing this procedure:

```
to go  ;; turtle procedure
    search-for-chip
    find-new-pile
    put-down-chip
end
```

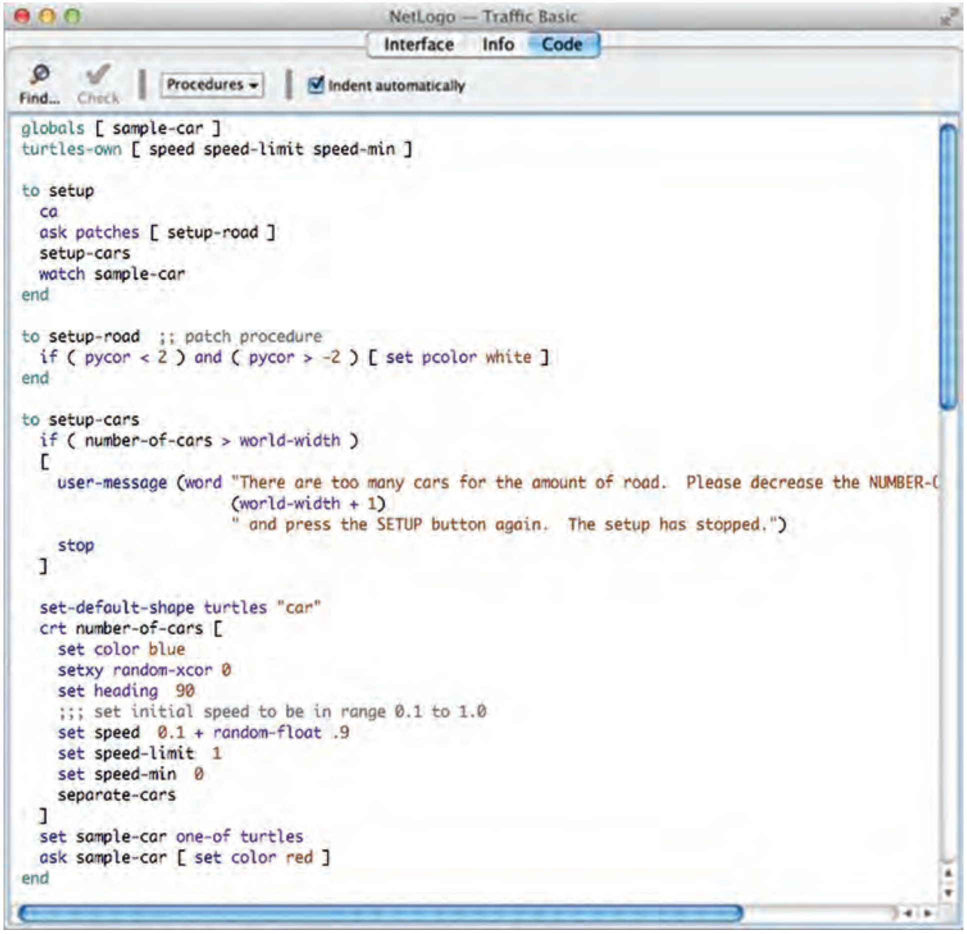
In this example, some termites can be on their second SEARCH-FOR-CHIP, even though some other termites have not finished the PUT-DOWN-CHIP command. While the difference is subtle, it can be important in some cases, providing a way to implement simulated parallel actions.

Wrapping It All Up

Now that we have reviewed all the components of an ABM, you should have a solid foundation for creating an agent-based model. In particular, there are three parts of any ABM that are essential to keep in mind as you are creating your model: the code, the documentation, and the interface. The *code*, which we have discussed and worked with for the last three chapters, tells the agents what to do and tells the model how to execute. The *documentation* describes the code, placing the model in the context of the real world. The *interface* is what allows users to control the model and thus manipulate its results.

In NetLogo, the code is placed in the Code tab (see figure 5.24). We have already discussed the code in some detail in the previous chapters so we will not discuss it much more here, except to highlight the importance of placing documentation in your code as well. In the code examples and models that we have developed, we have placed comments alongside the code (using the semicolon "comment" character). This is done to help anyone who reads your code to understand what is going on. In fact, it can even help you. Often modelers will reuse code or get asked questions about code, years after they have written it. If they have not documented their code well, it can take much longer to figure out what the code does, or why it was written as it was. In the end, documenting your code is a short-term cost for a long-term benefit, which usually outweighs the initial cost.

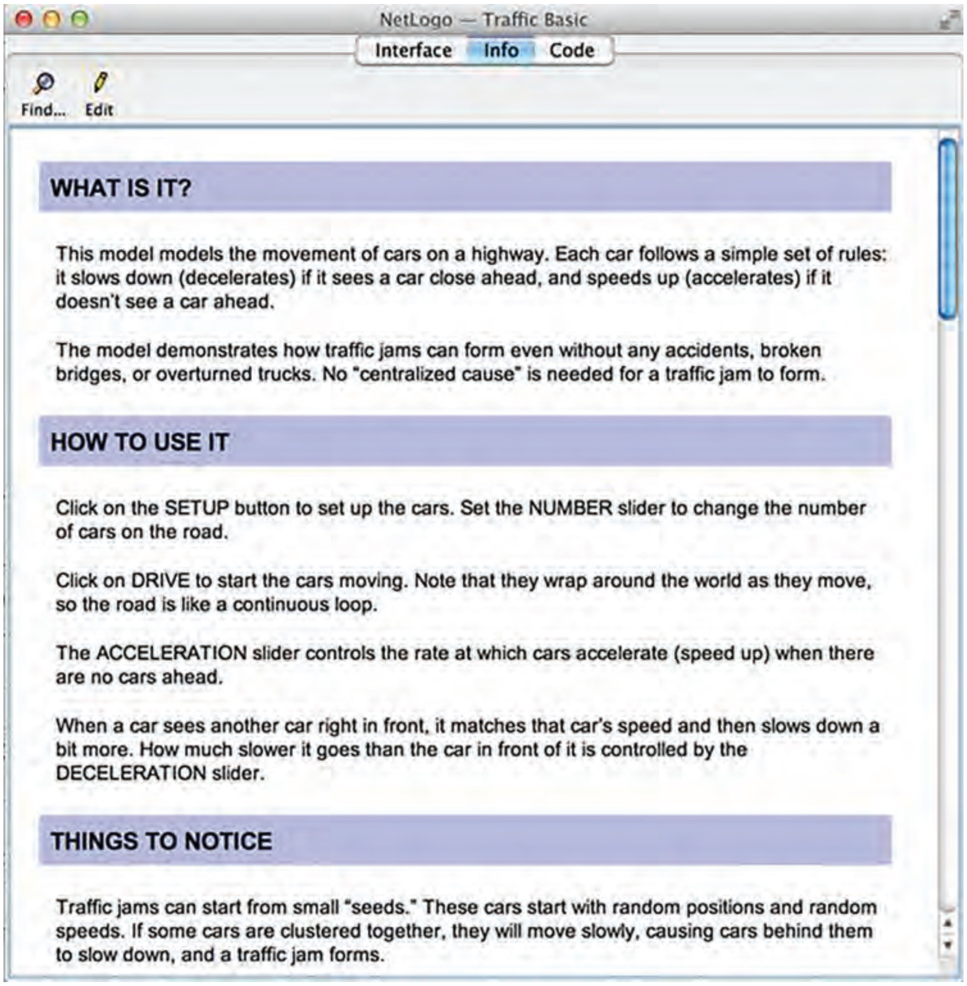
In NetLogo, the documentation is placed in the Info tab (see figure 5.25). This is in addition to the documentation that you have already placed in your code comments. This documentation should give an overall purpose and structure to your model. In



```
globals [ sample-car ]
turtles-own [ speed speed-limit speed-min ]

to setup
  ca
  ask patches [ setup-road ]
  setup-cars
  watch sample-car
end

to setup-road  ;; patch procedure
  if ( pycor < 2 ) and ( pycor > -2 ) [ set pcolor white ]
end

to setup-cars
  if ( number-of-cars > world-width )
  [
    user-message (word "There are too many cars for the amount of road.  Please decrease the NUMBER-C
                 (world-width + 1)
                 " and press the SETUP button again.  The setup has stopped.")
    stop
  ]

  set-default-shape turtles "car"
  crt number-of-cars [
    set color blue
    setxy random-xcor 0
    set heading  90
    ;;; set initial speed to be in range 0.1 to 1.0
    set speed  0.1 + random-float .9
    set speed-limit  1
    set speed-min  0
    separate-cars
  ]
  set sample-car one-of turtles
  ask sample-car [ set color red ]
end
```

Figure 5.24
Code tab.

NetLogo, this documentation is typically broken down into eight sections (See the "Sections of the Info tab" text box), though model authors are free to add their own sections. Though this format of eight sections is specific to NetLogo, in general it makes sense to provide documentation that covers the same basic aspects of the model. A general overview to give the context of the model (WHAT IS IT?), a user's guide for how to take advantage of the model (HOW TO USE IT), interesting results (THINGS TO NOTICE and THINGS TO TRY), future improvements for the model (EXTENDING THE MODEL), special techniques employed in the model (NETLOGO FEATURES), and related work that inspired the model, as well as other places for information on the model (RELATED



WHAT IS IT?

This model models the movement of cars on a highway. Each car follows a simple set of rules: it slows down (decelerates) if it sees a car close ahead, and speeds up (accelerates) if it doesn't see a car ahead.

The model demonstrates how traffic jams can form even without any accidents, broken bridges, or overturned trucks. No "centralized cause" is needed for a traffic jam to form.

HOW TO USE IT

Click on the SETUP button to set up the cars. Set the NUMBER slider to change the number of cars on the road.

Click on DRIVE to start the cars moving. Note that they wrap around the world as they move, so the road is like a continuous loop.

The ACCELERATION slider controls the rate at which cars accelerate (speed up) when there are no cars ahead.

When a car sees another car right in front, it matches that car's speed and then slows down a bit more. How much slower it goes than the car in front of it is controlled by the DECELERATION slider.

THINGS TO NOTICE

Traffic jams can start from small "seeds." These cars start with random positions and random speeds. If some cars are clustered together, they will move slowly, causing cars behind them to slow down, and a traffic jam forms.

Figure 5.25
Info tab.

Box 5.8
Sections of the Info tab

WHAT IS IT? Provides a general description of the phenomenon that is modeled.

HOW TO USE IT Gives instructions on how to run the model and describes the interface elements of the model.

THINGS TO NOTICE Describes interesting phenomena that the model exhibits.

THINGS TO TRY Describes how the user can manipulate the model to produce new results.

EXTENDING THE MODEL Gives suggestions and challenges on how to change the model to examine new features and phenomena. (This is similar to the future work section of a research paper.)

NETLOGO FEATURES Discusses some particularly interesting features of NetLogo that are used in the model.

RELATED MODELS Lists other agent-based models (often from the NetLogo models library) that are related to this model.

CREDITS AND REFERENCES Tells the user who created the model and where the user can go to find more information about the model.

MODELS and CREDITS AND REFERENCES). There are several other formats for documenting a model. One popular format is the ODD format, developed by Grimm and colleagues (2006) that specifically aims to be a standard protocol for describing simulation models. Documentation completes the loop of tying the model to the real world phenomenon it is supposed to be describing. No model is complete without accompanying documentation.

The final part of the model that is necessary for it to be complete is the user interface. In NetLogo, the user interface is in the Interface tab (see figure 5.26). The UI enables the user to set the parameters for a particular model run and to watch the model unfold. Also, as we have discussed in previous sections, a good visualization choice can help the model user to gain insights into the model that they may not have gotten based purely upon numerical outputs. In fact, in many cases ABM modelers learn more about their model from watching runs than they do from the raw data that the model outputs at the end of a run.

The other components that we have described (i.e., agents, environments, interactions, interface/observer, and schedule) are central to code, documentation, and interfaces. They are formally controlled in the code, described in the documentation, and employed by the user interface. Together, these three parts are the necessary wrapping to deliver the full model.

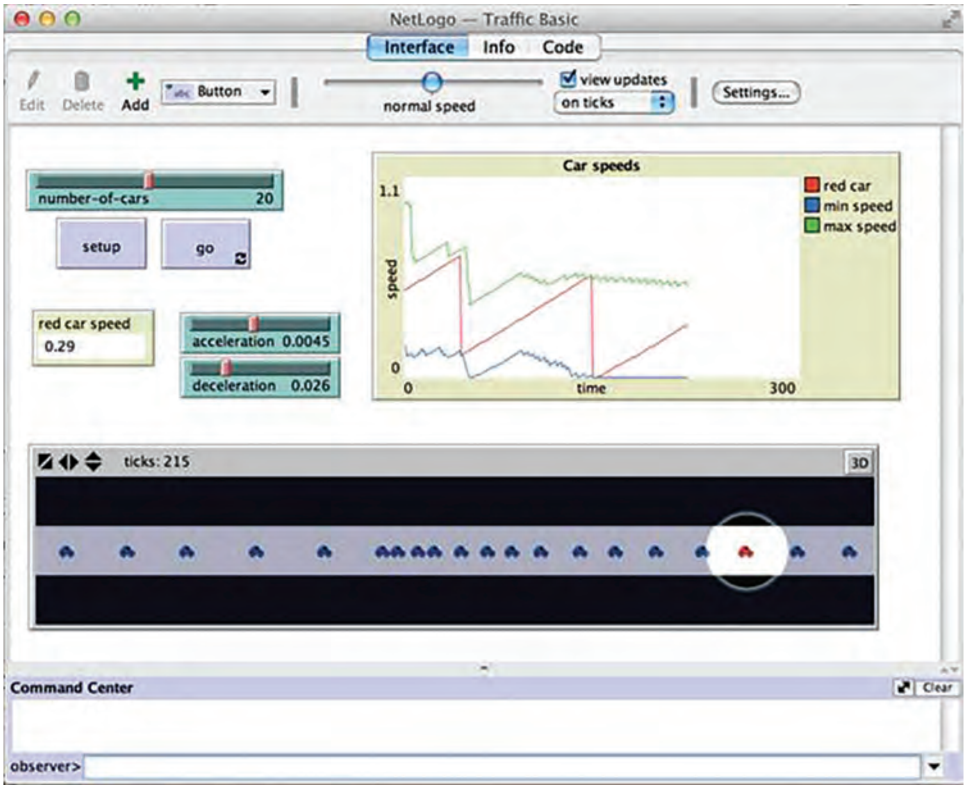


Figure 5.26
Interface tab.

Summary

We have described five classes of ABM components: agents, environments, interactions, observer/user interface, and schedule. These five classes of components of ABMs are described at a conceptual level. When creating your own models, you will find that you will need many instances of components, but they will all fall within these five general categories. Agents can have many different properties and behaviors, and there are many kinds of agents, but they are the basic component of an agent-based model, no agents, no ABM. The environment is where agents reside, and thus adequately describing the environment is important to an agent-based model. Interactions are how the dynamics of the model evolve, and therefore are critical to the operation of an agent-based model. The observer/interface is how the model is controlled and how data is extracted from the model.

Without the observer and interface, the model would not be usable. Finally, the schedule tells the model when to do what when, and there are important details about the schedule that must be considered when building a model.

These five components come together to form an agent-based model. They come together in the three parts of a model, the code, the documentation, and the user interface. The code is the formalistic description of the model, while the documentation ties the model to its real-world system, and the user interface enables the user to control the inputs, outputs, and operation of the model.

These components and parts are also a useful starting point for the design of an ABM. For example, suppose you want to expand on the traffic model to model a full urban transportation system. You might start by asking what the basic agents of the model are. Do you want to model trains, cars, buses, ferries, cyclists, and even pedestrians? After this, you might consider the environment. Trains tend to move on a network of rail systems, but pedestrians and cyclists are less geographically constrained. Your decisions about agents will affect your decisions about the environment and vice versa. After this you have to consider how agents and the environment interact. Can city hall create new transportation systems, or are you considering the current transport network to be exogenously fixed? Can some agents delay other agents? Do you just want to model commuters on their way to work or also agents on leisure trips? How does the observer and user interface interact with all of this? Will the movement of agents be visualized or will the system simply compute the mean time for a given commute? How does the schedule work? Who gets to decide where they are going? Can commuters change their minds based upon their previous workday? By thinking about these basic components, we can begin to frame the discussion that we will use to create our new model.

After completing this chapter, you have learned how to use simple agent-based models, extend an ABM, and build your own agent-based model, while familiarizing yourself with the components of an agent-based model. In the next chapter we will start to learn how to analyze a model so as to create useful results for answering the question that launched your model.

Explorations

1. Choose a model from the sample models in the models library. What are the agents in the model? What is the environment? What are the interactions? What responsibilities does the observer have? How is the schedule set up?
2. NetLogo turtle agents come with a default set of properties (heading, xcor, ycor, color, and others). Look at the complete list, which be found in the NetLogo documentation, then come up with a new property that you think might be a good addition to the default properties of agents. What are the arguments for including this property? What are the reasons that this property should not be included?

3. We have talked about geographic and network-based environments. Is there a model where you would want to use both geographic and network-based environments at the same time? Describe the model. Why would using both environment types simultaneously be needed for this model? Can the two environments interact with each other?
4. Modify the Grand Canyon model so that the water slowly erodes the elevation over time. Run the model several different times and export the resulting elevation maps back to a GIS package. Compare and contrast these different maps.
5. We have presented several basic interactions that agents often exhibit in agent-based models. Many times though the typical set of interactions that agents carry out is based upon the particular domain that you are trying to model. Thus, it is often useful to create a set of interactions for a particular domain. Can you describe a set of interactions that might be typical in a particular domain? For instance, what kinds of interactions might agents routinely carry out in economic models? Biological models? Engineering models?
6. Both the observer and the schedule require the model author to put together components of an agent-based model so that they will work together. How is the observer different from the schedule? Are these two components separate? Or can they be conceived of as one large component, with one part of the componentry controlling time and the other controlling the interaction topology?
7. *The human body as an ABM* In the world of agent-based modeling, sometimes the term *meta-agent* is sometimes used to describe a group of agents that act together in such a way that they can be collectively thought of as an agent themselves. For instance, a company can be thought of as an agent when it is interfacing with other companies, even though that company is itself composed of individual agents (i.e., the company's employees). That company could in turn be part of a larger conglomerate of companies that act together as another meta-agent. Describe the human body in terms of meta-agents. What are the basic agents of the human body? What are its meta-agents?
8. *Preferential attachment* Open the Preferential Attachment Simple model in the IABM chapter 5 folder. This model grows a network in which new nodes being added to the network are more likely to be connected to nodes that already have many links. These types of networks are common in real-world situations that are both engineered and emergent, like airline networks, the Internet, Hollywood actors in the same film, and the power grid. In many of these cases, if you are joining a network you would want to be more connected to a higher degree node. However, the model does not take a node's value into account at all. What if a node has an intrinsic value in addition to its connectedness? Modify this model to give the nodes an intrinsic value. Have new nodes attach to the network taking both degree and intrinsic value into account. Add a slider so that you can control this relationship. What are the results of your model what happens when the new nodes use only degree to make their decision? What happens when they only use intrinsic value to make their decision? What happens in between?

9. Modify the Preferential Attachment model so that as hubs get more and more attachment requests, they decrease their probability of accepting them. How does this change affect the resultant network?

10. *Giant component* Another model in the Networks section of the models library is the Giant Component model. This model starts with a group of nodes and then adds random links to them. The giant component is the largest connected subcomponent of this network. One question you might ask is, is there a point at which the giant component size grows quickly? How many links do you have to add for this to occur? Does this value change as you change the number of nodes? What if you want all of the nodes to be in the giant component? How many links do you have to add for this to occur? Run the model several times and examine how many ticks it takes to have all the nodes be part of the giant component. Is there any regularity to this amount of time?

11. In the Giant Component model, the probability of any two nodes getting connected to each other is the same. Can you think of ways to make some nodes more attractive to connect to than others? How would that affect the formation of the giant component?

12. Another model in the Networks section of the models library is the Small Worlds model (*Wilensky, 2005a*). That model is initialized with a set of nodes connected in a special circular network. Can you find another initial network to start with that can be easily modified to be small world?

13. Another model in the Networks section of the models library is the Virus on a Network model, which demonstrates the spread of a virus though a network of nodes each of which may be in one of three states: susceptible, infected, or resistant. Suppose the virus is spreading by emailing itself out to everyone in the computer's address book. Since being in someone's address book is not a symmetric relationship, change this model to use directed links instead of undirected links.

14. Try making a model similar to Virus on a network model but where the virus has the ability to mutate itself. Such self-modifying viruses are a considerable threat to computer security, since traditional methods of virus signature identification may not work against them. In your model, nodes that become immune may be reinfected if the virus has mutated to become significantly different from the variant that originally infected the node.

15. Another model in the Networks section of the models library is the Team Assembly model, which illustrates how the behavior of individuals in assembling small teams for short-term projects can give rise to a variety of large-scale network structures over time. Collaboration networks can alternatively be thought of as those networks consisting of individuals linked to projects. For example, one can represent a scientific journal with two types of nodes, scientists and publications. Ties between scientists and publications represent authorship. Thus, links between a publication multiple scientists specify coauthorship. More generally, a collaborative project may be represented by one type of node, and participants another type. Can you modify the model to assemble teams using bipartite network?

16. *Different topologies* Build a model of a phenomenon where agents communicate with nearby agents in a Euclidean space. Then build another model of the same phenomenon in which agents communicate across a social network. When would you use each of these models? What advantages does each topology have? What disadvantages? Compare the results of the two models.

17. *Möbius strips* We discussed many different topologies that can be used for agent interactions. One interesting topology is that of the Möbius strip. To create a Möbius strip you take a strip of paper, twist it and connect one end to the other end of the strip. The interesting thing about the Möbius strip is that while a normal sheet of paper has two sides, it only has one. You can prove this to yourself by tracing a line on the Möbius strip. Build an agent-based model with this topology. For instance, can you modify the Flocking model to have the birds fly on a Möbius strip? (Hint: One way to do this would to use a bounded topology in the top-bottom, and have a toroidal wrapping topology in the left-right.)

18. *Smarter agents* Examine one of the Artificial Neural Net models from the Computer Science section of the models library. This model shows how you can use a simple machine learning technique to develop an algorithm that matches inputs to outputs. In this chapter, we discussed how machine-learning algorithms can be used to make agents smarter. Take the Neural Net code and embed it in another model to make the agents smarter. For instance, take the Wolf Sheep Predation model and have the wolves use neural nets to decide in what direction to move to most efficiently predate.

19. *Adaptive Traffic Basic* In the section on adaptive agents, we gave you some of the basic pieces of code necessary to build an adaptive Traffic Basic model. Take this code and make it fully work. To do this you may need to rewrite the SETUP procedure and add some global declarations.

20. *Individual Adaptive Traffic Basic* In the adaptive Traffic Basic model that we discussed and developed in the previous exploration, all of the turtles have the same acceleration rates. Modify this code so that each turtle has its own acceleration and adapts this acceleration based on its own experience.

21. *Info Tab* We have discussed how important documentation is to a model. The Info Tab is a critical part of that documentation, since it describes what the model represents. In fact, some scientists argue that without proper documentation detailing the relationship between an implemented model (i.e., the code) and the real-world phenomenon, a model is not a model; it is simply a piece of software that computes some numbers. Substantiate this argument. What is a sufficient level of documentation that a model author must provide to complete a model? Provide an explanation for your decision.

22. *Interface Tab* The Interface Tab enables a model user to interact with a model that you have designed. It is important that the Interface Tab provide both a clear visualization and an easy-to-use set of interactive tools to control the model. Describe three guidelines for designing an informative visualization. Describe three guidelines for designing an easy-to-use interface.

23. *Discrete event scheduler* We have discussed several different issues to consider when designing the schedule for your agent-based model. There is one other approach to scheduling that we did not discuss in this chapter. Discrete Event Scheduling (DES) is when the agents and the environment of your model schedule events to take place at discrete times. For instance, examine the Mousetraps model in the Mathematics section of the models library. Right now, whenever a ball flies through the air it simply moves until it comes to a resting place and then attempts to trigger a mousetrap. Instead of this method, you could imagine implementing this model without any ball agents; instead, when a mousetrap is tripped, it schedules two other mousetraps to trip within a radius at a certain time in the future. This would be a discrete event scheduler. Reimplement this model using a DES.
24. The DES idea of discrete ticks being used to represent time is very useful. However, this may not be the most computationally efficient mechanism for all systems. Can you speculate about how to move away from a discrete scheduler toward something that would allow the model builder to specify when something was executed rather than how often it was executed? What would such a system look like? How would it differ from the standard tick based system? What are the costs and benefits of this new approach?
25. *Activation* The order in which agents execute can greatly affect how a model runs. Open the Life model in the NetLogo models library under Computer Science > Cellular Automata. This model operates with all of the patches running in synchronous order, that is, each patch waits to update its state until all other patches have updated their state. Modify this model so that it operates in an asynchronous order. How do the two models compare? Describe the patterns that arise from each model. Is it possible to create the glider (discussed in chapter 2) in the asynchronous model?
26. *3D models* It is often possible to create a 3D version of a 2D ABM. Sometimes the behavior of the model is similar, but other times it can have quite different results. Create a 3D version of the Segregation model in the Social Science section of the models library.
27. In the NetLogo 3D Flocking model, can you extend the model so that the birds can fly around obstacles in the middle of the world?
28. In the NetLogo 3D Termites model, can you extend the model to have the termites sort several colors of wood? Create some informative plots that are useful to measure the termites' progress.
29. In the section on goal-directed agents, we described how the goal-directed agents in Traffic Grid were not very smart because they would get caught in infinite loops of going back and forth between the same squares both of which are equally distant from their goal. As a result, they never reach their goal. Modify the algorithm so that the agents always reach their goal.
30. After modifying the Traffic Grid model as in the previous exploration, the traffic still might not flow that smoothly as cars are turning around while on the road. Modify the model so that when a car reaches its goal, it moves off the road into the house or work

and remains there for some time, STAY-TIME. Which values of STAY-TIME result in the most efficient traffic?

31. *Turtles vs. patches* Both turtles and patches can be agents in a model. In fact many models could be implemented as either turtle-based models or patch-based models. Open the AIDS model in the Biology section of the models library. Currently this model is turtle-based. Reimplement this model without any turtles, using only patches. Open the Life model in the computer science of the models library, which is patch-based. Reimplement this model without using patches, but instead have turtles that are hatched and killed. Describe your experience in reimplementing these models. At what times is it more beneficial to use turtles as agents? At what times is it more beneficial to use patches as agents?

32. Open the Robby the Robot model in the Computer Science section of the NetLogo models library. Robby is a virtual robot that moves around a room and picks up cans. This model demonstrates the use of a genetic algorithm (GA) to evolve control strategies for Robby. The GA starts with randomly generated strategies and then uses evolution to improve them. Vary the settings on the POPULATION-SIZE and MUTATION-RATE sliders. How do these affect the best fitness in the population as well as the speed of evolution? Try different rules for selecting the parents of the next generation. What leads to the fastest evolution? Is there ever a trade-off between fast evolution at the beginning and how effective the winning strategies are at the end?

33. Many times in both nature and society, distributed elements can synchronize their behavior. This occurs with physical systems such as coupled oscillators, with biological systems such as synchronized firefly flashing and with human systems such as audiences clapping.

Open the Fireflies model from the Biology section of the NetLogo models library. It presents two strategies for fireflies to synchronize their flashes: “phase advance” and “phase delay.”

(a) Change the strategy chooser between “delay” and “advance” while keeping the other settings steady (in particular, keep FLASHES-TO-RESTART at 2). Which strategy seems more effective? Why?

(b) Try adjusting FLASHES-TO-RESTART between 0, 1 and 2 using both phase delay and phase advance settings. Notice that each setting will give a characteristically different plot, and some of them do not allow for synchronization at all (for example, with the delay strategy, contrast FLASHES-TO-RESTART set to 1 as opposed to 2). Why does this control make such a difference in the outcome of the simulation?

(c) This model explores only two general strategies for attaining synchrony in such cycle-governed fireflies. Can you find any others? Can you improve the existing strategies?

(d) There are many other possible situations in which distributed agents must synchronize their behavior through the use of simple rules. What if, instead of perceiving only other discrete flashes, an insect could sense where another insect was in its cycle (perhaps by hearing an increasingly loud hum)? What kinds of strategies for synchronization might be useful in such a situation?

(e) If all fireflies had adjustable cycle-lengths (initially set to random intervals) would it then be possible to coordinate both their cycle-lengths and their flashing?

34. *List Manipulation* Write a reporter that takes two lists as input and reports a list of the pairwise sums.

35. Write a procedure that takes a list of turtles and a list of numbers and asks each turtle to move the corresponding number forward.

36. Write a reporter that takes a list as input and reports the list in reverse order.