4 Creating Agent-Based Models

If you didn't grow it, you didn't explain it.

—Josh Epstein (1999)

In a minute there is time for decisions and revisions which a minute will reverse.

—T. S. Eliot (from "The Love Song of J. Alfred Prufrock," 1920)

What I cannot create, I do not understand.

—Richard Feynman (as seen on his blackboard and attributed by Hawking, 2001)

In the previous two chapters, we had our first taste of working with agent-based modeling code, writing simple models, examining model code, and extending it. One can accomplish a lot by working with publicly available models and modifying and extending them. Even in the most advanced ABM models one can often find code snippets borrowed from other models. However, eventually you will want to design and build your own model from the ground up. This chapter is intended to take you from the first step of devising a question or area you want to explore, all the way through designing, and building your model, to refining your question and revising your model, to analyzing your results and answering your question. This sequence is presented here in linear order, but in reality these steps fold back on each other and are part of an iterative exploration and refinement of the model and motivating question.

We will explore all of this within the context of a particular model, but at the same time we will discuss general issues related to model authoring and model design. To facilitate this process, this chapter is broken into three main sections: (1) *Designing your model* will take you through the process of determining what elements to include in your model, (2) *Building your model* will demonstrate how to take a conceptual model and create a computational object, and (3) *Examining your model* will address how to run your model, create some results, and analyze those results to provide a useful answer to your motivating question.

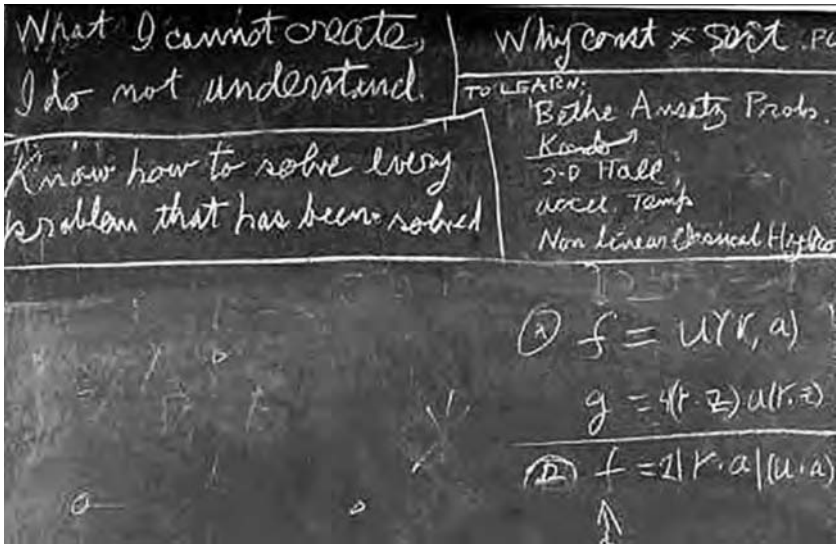

Figure 4.1
From the physicist Richard Feynman's blackboard at the time of his death.

Throughout this chapter we will be designing, building, and examining a simple model of an ecological system. The basic question that we will be addressing is: "How do the population levels of two habitat-sharing animal species change over time?" For our purposes in this chapter, we will call this model the Wolf Sheep Simple model. Though we will discuss this model in the context of two biological species, the model could be generalized to other situations such as companies competing for consumers, electoral parties competing for votes, or viruses evolving in a computer system. More important, the components that we will be developing in this model are basic components utilized in most ABMs.

Designing Your Model

There are many ways of designing an agent-based model.[1] Which you choose will depend on many factors including the type of phenomenon to be modeled, your level of knowledge of the content domain, your comfort with NetLogo coding and your personal modeling style.

1. As mentioned in chapter 1, a model can be either the conceptual/textual description of a process or the implemented software-based description of the model. In this chapter we will use the word *model* to describe either concept, but when it is necessary to distinguish between them we will describe the textual description as a conceptual model.

We consider two major categories of modeling: *phenomena-based modeling* and *exploratory modeling*. In *phenomena-based modeling*, you begin with a known target phenomenon. Typically, that phenomenon has a characteristic pattern, known as a *reference pattern*. Examples of reference patterns might include common housing segregation patterns in cities, spiral-shaped galaxies in space, leaf arrangement patterns on plants, or oscillating population levels in interacting species. The goal of phenomena-based modeling is to create a model that will somehow capture the reference pattern. In ABM, this translates to finding a set of agents, and rules for those agents, that will generate the known reference pattern. Once you have generated the reference pattern you have a candidate explanatory mechanism for that pattern and may also vary the model parameters to see if other patterns emerge, and perhaps try to find those patterns in data sets or by conducting experiments. You can also use phenomena-based modeling with other forms of modeling, such as equation-based modeling. In equation-based modeling, this would mean writing equations that will give rise to the reference pattern.

The second core modeling form is *exploratory modeling.* This form is perhaps less common in equational contexts than it is in ABM. In exploratory modeling with ABM, you create a set of agents, define their behavior, and explore the patterns that emerge. One might explore them solely as abstract forms, much like Conway and Wolfram did with cellular automata as we read in chapter 2. But to count as modeling, we must note similarities between the behavior of our model and some phenomena in the world (just as patterns we saw generated by cellular automata in chapter 2 resembled patterns on shells). We then refine our model in the direction of perceived similarities with these phenomena and converge toward an explanatory model of some phenomenon.

Another distinction in modeling methodology is to what degree we specify a question to be answered by a model. At one end of the spectrum, we formulate a specific research question (or set of questions) such as "How does a colony of ants forage for food?" or "How does a flock of geese fly in a V-shape?" At the other end, we may only begin with a sense of wanting to model ants or bird behavior, but without a clear question to be answered. As we explore the model design space, we will gradually refine our question to one that can be addressed by a specific model.

Yet a third dimension is the degree to which the process of designing the conceptual model is combined with coding your model. In some cases, it is advisable to work out the entire conceptual model design in advance of any coding of the model. This is referred to as *top-down* design. In a top-down design, the model designer will have worked out the types of agents in the model, the environment they reside in, and their rules of interaction before writing a single line of code. In other cases, the conceptual model design and the coding of the model will coevolve, each influencing the evolution of the other. This is often referred to as *bottom-up* design. In bottom-up design, you choose a domain or phenomenon of interest with or without specifying a formal question. Using this approach, you would then start writing code relevant to that domain, building the conceptual model

from the bottom up, accumulating the necessary mechanisms, properties and entities, and perhaps formulating some formal research questions along the way. For example, in a bottom-up design, you might start with a question about how an economic market would evolve, code some behaviors of buyers and sellers and, in so doing, realize you'll need to add brokers as agents in the model.

These model design dimensions can be combined in various arrangements. You could start with a very specific research question and design all the agents and rules before coding, or you can start with some agents, play with various rules for them and only get to your modeling question near the end of the process.

In practice, model authors rarely use exclusively one style when building their models, but use some combination of the styles, and often switch back and forth between the forms and styles as their research needs and interests change. In cases where a scientist is collaborating with a programmer who will code the model, the top-down design style is usually the one employed as it separates the roles of the two team members. NetLogo was designed to make it easier for scientists to code their own models. Often, as modelers become more comfortable with coding, they use the NetLogo code as a tool to build their conceptual model. In this chapter, we will present our model building using a mixture of the approaches, but for clarity of the exposition, we will emphasize the top-down approach.

The top-down design process starts by choosing a phenomenon or situation that you want to model or coming up with a question that you want to answer, and then designing agents and rules of behavior that model the elements of the situation. You then refine that *conceptual model* and continue to revise it until it is at a fine enough level of detail that you can see how to write the code for the model.

Throughout the design process there is one major principle that we will use. We call this the *ABM design principle*: Start simple and build toward the question you want to answer.[2] There are two main components of this principle. The first is to begin with the simplest set of agents and rules of behavior that can be used to explore the system you want to model. This part of the principle is illustrated by a quote from Albert Einstein, "The supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience" (1933). Or in another phrase he is reputed to have said: "Everything should be made as simple as possible, but not simpler." In the case of ABM, this means making your model as simple as possible given that it must provide you with a

2. This design principle is stated from a top-down perspective of model building. The bottom-up variant is not that different: Start simple and be alert to possibly interesting questions, increasing the complexity of the model to pursue these questions. In a bottom-up process, you start with a domain or phenomenon of interest and build a very simple model related to that domain or several components that might be useful for investigating the phenomenon. You then explore the simple model or model components, looking for promising directions. The bottom-up perspective does not require a driving question in advance; the question and the model coevolve, changes in one driving changes in the other.

stepping-stone toward your final destination. Second, always have your question in mind, which means not adding anything to your model that does not help you in answering your question. The statistician George Box provides a quote that illustrates this point, "All models are wrong, but some models are useful" (1979). What Box meant was that all models are by necessity incomplete because they simplify aspects of the world. However, some of them are useful because they are designed to answer particular questions and the simplifications in the model do not interfere with obtaining that answer.

This core ABM design principle is useful in several ways. First, it reminds us to examine every candidate model agent and agent-rule and eliminate it if progress can be made without it. It is not uncommon for novice modelers to build a model in which certain components have no effect whatsoever. By starting small and slowly adding elements to your model, you can make sure that these extraneous components never get developed. By examining each additional component as to whether it is needed to answer the research question you are pursuing, you reduce the temptation to, paraphrasing William of Occam, "multiply entities unnecessarily." In so doing, you reduce the chance of introducing ambiguities, redundancies, and inconsistencies into your model. Another virtue of the ABM design principle is that, by keeping the model simple, you make it both more understandable and easier to verify. *Verification* is the process of ensuring that a computational model faithfully implements its target conceptual model. A simpler conceptual model leads to a simpler model implementation, which makes it easier to verify the model. Starting simple and building up also facilitates the process of just-in-time results. At every point of the model development process, the model should be able to provide you with some answers to your research question. Not only does this help you make productive use of your model early on, but it also enables you to start questioning your model assumptions and examining its results early on in the modeling process. This can prevent you from going too far down an unproductive path. Fewer components also mean fewer combinations to test in order to develop a causal account of your results.

To apply our principle to the context of the Wolf Sheep Simple model, we need to start our design by reflecting on two habitat-sharing animal species and identifying simple agents and behaviors for our model. We will start by identifying a question we want to explore, which is required by the ABM design principle (top-down version). After that we will discuss what the agents in our models are and how they can act. Then we move on to the environment and its characteristics. As part of this process, we need to discuss what happens in an individual time step of the model. Finally, we discuss what measures we will be using to answer our question.

Choosing Your Questions

Choosing a question may seem to be a separate issue from model design. After all, the natural progression seems to be: *first* choose a question, and *second* build a model to answer that question. Sometimes, that may indeed be the procedure we follow, but in many

instances we will need to refine our questions when we start to think about it in an agent-based way. Our original question for the Wolf Sheep Simple model was: "How do the population levels of two species change over time when they coexist in a shared habitat?" We will now evaluate whether this question is one that is amenable to ABM and refine our question within the ABM paradigm.

Agent-based modeling is particularly useful for making sense of systems that have a number of interacting entities, and therefore have unpredictable results. As we discussed in chapter 1, there are certain problems and questions that are more amenable to ABM solutions. If our primary question of interest violates our guidelines, it may be an indication that we should consider a different modeling method. For example, we might be interested in examining the dynamics of two very large populations under the assumption that the species are homogenous and well-mixed (no spatial component or heterogeneous properties) and that the population level of each species is simply dependent on the population level of the other species. If that is the case then we could have used an equation-based model instead of an agent-based model since EBM's work well for large homogeneous groups, and as we mentioned in chapter 0 there is a classic EBM for this situation known as the Lotka-Volterra differential equations (Lotka, 1925; Volterra, 1926). (Near the end of this chapter there will be additional discussion of the relative merits of using EBM versus ABM for ecological predation models.) ABM will be more useful to us if we are thinking of the agents as heterogeneous with spatial locations. This affects how we conceptualize the agents. One aspect of the animals that is likely to be relevant to our question is how they make use of their resources. Animals make use of food resources by converting them to energy hence we will want to make sure that our agents have different amounts of energy and different locations in the world. A third guideline is to consider whether the aggregate results are dependent on the interactions of the agents and on the interaction of the agents with their environment. For example, if one species is consuming another then the results will be dependent on agent interaction. Predator-prey interactions are usually set in rich environments. Keeping the ABM design principle in mind, we start with the simplest environment—we enrich the environment a little by going beyond just predators and prey and including resources from the environment that the lowest level prey species consume. Yet another guideline is that agent-based modeling is most useful for modeling time dependent processes. In the Wolf Sheep Simple model, our core interest lies in examining how population levels change over time. We might therefore refine our question to focus on conditions that lead to the two species coexisting together for some time. In this way our guidelines help us evaluate whether our question is well suited to ABM and, if so, to focus our question and conceptual model.

Having evaluated our question's suitability to ABM, we are now in position to state it more formally. "Can we find model parameters for two species that will sustain positive population levels in a limited geographic area when one species is a predator of the other

and the second species consumes resources from the environment?" Now, keeping this question in mind, we can proceed to design the conceptual model.

A Concrete Example

Now that we have identified our research question in detail it can be useful to consider a particular context for this research question. Earlier, we discussed reference patterns as a source of phenomena-based agent-based models. Sometimes that reference pattern is the original inspiration for the model. Other times, as now, we have refined our research question enough that we seek out a reference pattern that will help us test whether our model is a valid answer to the question. In the case of the predator-prey relations, there is a famous case of cohabiting small predator-prey populations in a small geographic area. This is the case of fluctuating wolf and moose populations in Isle Royale, Michigan.

> The wolves and moose of Isle Royale have been studied for more than five decades. This research represents the longest continuous study of any predator-prey system in the world. . . . Isle Royale is a remote wilderness island, isolated by the frigid waters of Lake Superior, and home to populations of wolves and moose. As predator and prey, their lives and deaths are linked in a drama that is timeless and historic. Their lives are historic because we have been documenting their lives for more than five decades. This research project is the longest continuous study of any predator-prey system in the world. (From the Wolves & Moose of Isle Royale Project Website, http://isleroyalewolf.org/)

Figure 4.2 shows the wolf and moose populations in Isle Royale from 1959 through 2009. This graph can serve as a reference pattern for our model. Our completed model

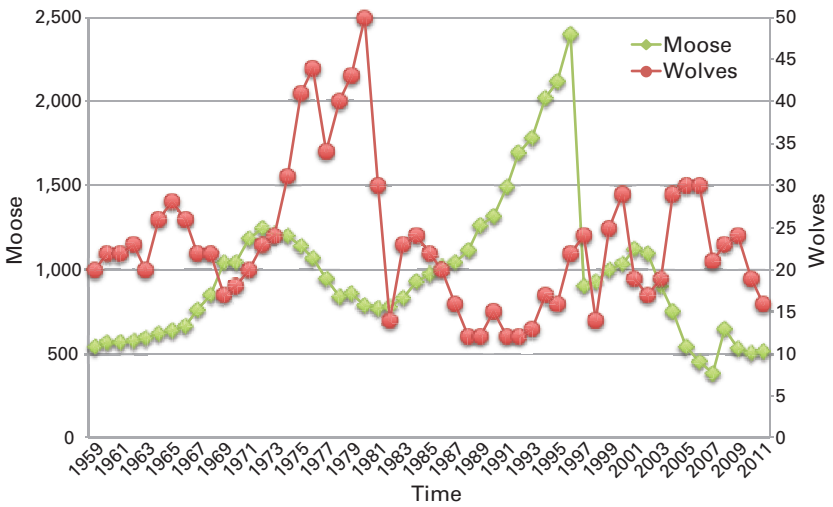

Figure 4.2
Five decades of fluctuating wolf and moose populations at Isle Royale. Note that when the wolf population peaks, the moose population is at a low point and, similarly, when the moose population peaks, the wolf population is at a low point.

will have to be able to generate a graph "similar" to this one to be a possible explanatory model of these phenomena. In general, this process is called *validation* and will be discussed in detail in chapters 7 and 8.

As we can see in the data from Isle Royale, the wolf and moose populations in Isle Royale have been sustaining themselves for more than fifty years without either species going extinct. The populations also exhibit a rough oscillation, with moose at a low when wolves peak and vice versa. This data can serve as a reference pattern for our phenomena-based modeling. It allows us to further refine our research question to this: "Can we find model parameters for two species that will sustain *oscillating* positive population levels in a limited geographic area when one species is a predator of the other and the second species consumes resources from the environment?"

In the models in this chapter, instead of modeling wolf and moose, we will model wolf and sheep. The wolf and moose data set is well established, but our goal in this chapter is not to match this particular data, but to introduce you to classic examples of predator-prey modeling and to try to reproduce the oscillating sustained pattern of population levels.

Choosing Your Agents

Now that we have identified and contextualized our driving research question, we can begin to design the components that will help us answer it. The first question we should ask ourselves is: What are the agents in the model? When designing our agents, we want to choose those components of our model that are autonomous and have properties, states, and behaviors that could possibly have bearing on our question. But we must be careful to avoid agent overload. Depending on the perspective one takes, almost any model component could be considered an agent. However, a model that is designed with an excess of agent types can quickly become unmanageable. When choosing what are to be the agents in a model, it is important to concentrate on those autonomous entities which are most relevant to our research question.

A related issue is the "granularity" of the agent. Every entity is composed of multiple smaller entities. What is the right level of entity to choose? Should our agent be molecules or atoms? Body organs or cells? Some agents can be treated as mass properties. If we want to model a field of grass, we might not want to model every blade of grass, but instead choose "clumps" of grass as our agents. It is important that the granularity of each agent be at roughly the same level. For instance in the temporal scale, if you are modeling the sheep actions at the level of days of activity, but the grass minute by minute, that can be difficult to reconcile. And in the physical scale, if there is more grass than the sheep can consume then you will not see many interesting behaviors since grass will not serve as a limiting condition.

If we suspect that, in the future, some model entities might need to become full agents, we can choose to design them as *proto-agents*. By the term proto-agent, we mean agents that do not have individual properties, states, or behaviors but instead inherit some or all of their characteristics from a global agent type. For example, in the Wolf Sheep Simple model we might desire to have a human hunter who interacts with the other two species. This hunter might simply eliminate a part of the populations every now and then. There is no need to build the hunter as a full agent at the start; instead we can create a simpler proto-agent that has the ability to kill off a random percentage of the population. Eventually, if needed, this hunter could become a full agent and have full properties and behavior just like any other agent. We discuss proto-agents further in chapter 5.

Given the preceding discussion, we start our Wolf Sheep Simple model design by choosing three agent types. We model the predators, which we will call wolves, and the prey, which we will call sheep, and the resources the sheep consume, grass. We could have added many other agents to this model. For example, we could model the hunter described above or the precipitation levels or soil nutrition. However, by choosing just wolves, sheep, and grass, we stick to the ABM design principle. We have the two simple mobile agent types, and one stationary agent type to model the environment, and those are the minimal set of agents necessary to answer our question of what parameters will allow two populations to coexist in a limited geographic area.

Choosing Agent Properties

Agents have properties that distinguish them from other agents. It is important to determine these properties in advance so we can conceptualize the agent and design the agents' interaction with each other and with the environment.

In the Wolf Sheep Simple model we give the sheep and wolves three properties each: (1) an energy level, which tracks the energy level of the agent, (2) a location, which is where in the geographic area the agent is, and (3) a heading, which indicates the direction the agent is currently moving or would be moving.[3] The energy property is not merely describing temporary energy (such as whether an animal is fresh or fatigued). Rather, "energy" incorporates some notion of the amount of "vitality" in a creature, abstracting away the messy details of metabolism, calorie storage, or starvation, and condensing it all into a single measure. We could add additional properties and some of them might be useful for future extensions. For instance, we could add a movement speed and allow different agents to move at different speeds, or an offense/defense capability that affects the ability of the individual to predate or resist predation. However, these additional properties

3. We also give the wolf and sheep agents shapes, and to all three agent types we give colors, which are not core to the behavioral rules for the ABM but are important for effective visualizations.

do not seem necessary to answer our simplest question, and thus we resist the temptation to include them unnecessarily.

If the sheep and the wolves have exactly the same properties, then what makes them different from each other? We will discuss this in the next section, where we talk about the behavior that each of these two agent types exhibit.

Choosing Environmental Characteristics and Stationary Agents Now that we have the mobile agents and their behaviors defined, we can decide on the nature of the environment in which these mobile agents will live and how they can interact with that environment.

In the Wolf Sheep Simple model, the first obvious environmental attribute is the presence or absence of grass, since that is what the sheep consume. We could model many other attributes such as elevation, water, woodlands, and other features that might affect the movement of the animals or affect sheep predation. However, in keeping with our design principle, we start with an environment consisting of a large grassy field. We use the stationary patch agent types to model the grass. As mentioned, it would not make sense to model every blade of grass, so we model the grass by giving the patches a "grass amount" property that will have a numerical value. This is effectively using the patches to model clumps of grass, which is our stationary agent type. The numerical value of this property should be in proportion to the feeding behavior of the sheep, since that is how it will be used in the model. In other words, the granularity of this variable should be set appropriately as discussed above.

In order to avoid dealing with boundary conditions (such as wolves stepping beyond the bounds of the modeled world), the world will "wrap" horizontally and vertically, so a wolf stepping off the right edge of the world will appear on the left. This "torus-shaped" world topology is often convenient for ABMs, and is thus the default for new models in NetLogo. (Other topologies will be discussed in chapter 5.)

It is also worth noting that in some ABMs the environment also controls the birth and death processes of the agents. In this model birth and death will be modeled endogenously within the actions of the agents, but it is possible to simply have birth and death of agents controlled by the environment instead. This is a less "emergent" way of modeling life-cycles, but sometimes is a useful simplification.

Choosing Agent Behavior

In addition to designing the structure of the agents, it is important to determine what kind of behavior the agents can exhibit. These behaviors are necessary to describe how agents interact with each other and the environment.

In the Wolf Sheep Simple model, sheep and wolves share many common behaviors. They both have the ability to turn randomly, move forward, reproduce and die. However, sheep and wolves differ in that sheep have the ability to consume grass and wolves have the ability to consume sheep. This differentiates the two species/agent types from

each other. Of course, once again there are many other behaviors we could prescribe for these agents. For instance, we could give sheep the ability to huddle in herds to defend against wolf attacks, or the ability to fight back. The wolves could have the ability to move at different speeds from a walk to a run or to chase sheep. Wolves and sheep also engage in a number of behaviors, such as sleeping, digesting food, and seeking shelter during a thunderstorm. However, again the behaviors we have described (moving, reproducing, eating, and dying) are reasonable choices for a simple model that can address our research question. For the grass clump agents, we give one simple behavior, the ability to grow.

Designing a Time Step Now that we have established the basic components of the model, we can design the typical time step in the model. To do this we need to think through all of the behaviors that will be exhibited by the agents of our model and decide how they will perform these behaviors and in what order they should be performed. In the real world, animals behave concurrently, and time appears continuous. To build our ABM, we simplify by dividing time into discrete steps, and we further divide each step into serialized, ordered phases. By doing it this way, we are making an implicit assumption that having the agents use some order to perform their actions will not substantially affect our results. This is a working assumption and may need to be reexamined later. In general, determining the order in which agents exhibit behaviors can be tricky. We will discuss agent "scheduling" further in chapter 5.

In the Wolf Sheep Simple model, there are four basic animal behaviors (move, die, eat, reproduce) and one grass behavior (grow). Another working assumption we may make is that, given that we need an order, deciding which order the animals perform their behaviors can be arbitrary. Any order for the behaviors would be reasonable, so we arbitrarily choose an order, because it's much easier to work with (and debug) a fixed order of behaviors. We choose to order the behaviors as in the first sentence of this paragraph. We can check to make sure the order makes sense. Movement is the act of turning and then stepping forward. Since the *move* action changes the location of the agents and thus changes the local environment of each agent, it makes sense to move first. In the Wolf Sheep Simple model, movement costs energy and thus we schedule *death* next, because we should check to see if any of the agents expended so much energy while moving that they have no energy left. After that we schedule the agents to attempt to gain new energy by *eat*ing if there is something in their local environment that they can eat. Since they now have new energy the agents may be able to *reproduce* (which also requires energy). Thus, each agent checks to see if they have enough energy to create a new agent.[4] Finally,

4. This is a drastic simplification of biological reproduction! We have made the choice to simplify reproduction to asexual reproduction based on a working hypothesis that for the purposes of answering our question, the details of reproduction are not relevant.

since the model has done everything else, the grass agents *grow* before we cycle to the next movement step.

There are many alternative ways to set up this time step. The order that behaviors occur could be altered, and in some ABMs the order can significantly change results (Wilensky & Rand, 2007). However, in this case there is no obvious reason why changing the order would have a significant effect on the model, and the order is a logical one. We could also add additional steps to this large picture framework, for instance we could separate out the wolves and sheep and allow all the wolves to move before the sheep move. However, the order and steps presented above are logical and simple and provide a good starting point. We take note of the many working assumptions we make along the way, and, if necessary, this time step structure can be reexamined and revised in the future.

Choosing Parameters of the Model

We could decide to write one set of completely specified rules to control the behavior of all of these agents and their environmental interactions during a time step, but it makes more sense to create some parameters that enable us to control the model, so that we can easily examine different conditions. A next step is to define what attributes of the model we will be able to control through parameters.

There are several possible parameters of interest in the Wolf Sheep Simple model. For instance, we will want to be able to control the number of initial sheep and wolves. This will enable us to see how different values of the initial population levels affect the final population levels. Another factor to control is how much energy it costs an agent to move. Using this parameter, we can make the landscape more or less difficult to traverse, and thus simulate different types of terrain. Related to the cost of movement is the energy that each species gains from food. Thus, we choose to have parameters for controlling the energy gained from grass and the energy gained from sheep. Finally, since the sheep consume grass, in order to sustain the population over time we will want the grass to regrow. So we will need a parameter for the grass regrowth rate.

There are many other parameters that we could have included in this model. For instance, the parameters we chose are homogenous across the model. In other words, one sheep will gain the same from grass as any other sheep. However, we could make this model more heterogeneous by drawing the energy gain for each sheep from a normal distribution, and have two model parameters that control the mean and variance of the energy gain. We could also add parameters to control aspects that we are currently planning to specify as constant values in the code. For example, we did not create a parameter to control the speed of the agents. Having the ability to modify those speeds and (particularly the ratio between movement rates for wolves and sheep) might dramatically affect

the model. However, guided by our ABM design principle, this complication does not seem necessary at this stage of the modeling process. Allowing different movement speeds for wolves and sheep is an expansion on this model that is left for the reader to explore. See the explorations listed at the end of this chapter.

Choosing Your Measures If we had implemented all of the preceding components, then we would have a working model. However, we still would have no process for answering our question. For that purpose we need to decide what measures we will collect from the model. Creating measures can be very simple at times, but often some of the most interesting results of a model are not recognized until after the measures have been properly designed. When considering what measures to incorporate into your model it is useful to review the research question. It is advisable to include only the most relevant measures, because extraneous measures can overwhelm you with data and may also slow down model execution.

In the Wolf Sheep Simple model, the measures that are most relevant are the population counts of the wolf and sheep over time, since what we are interested in is what sets of parameters will enable us to sustain positive levels of both populations over time.

We could construct measures of much other data in this model, such as the amount of energy possessed by sheep or wolves on average. This might bear on our question, since it could indicate how likely the current populations are to persist, but it does not directly address the question so we do not include it here. Sometimes it is useful to include measures like this for debugging purposes. For example, if we saw that the energy levels of sheep were increasing even though there was no grass regrowing, then we would wonder if there was a bug in the section of the code where we converted grass to energy for the sheep.

Summary of the Wolf Sheep Simple Model Design

Now that we have gone through the major design steps, we can create a summary document that describes our model design. The Wolf Sheep Simple model can be described in the following way:

Driving Question Under what conditions do two species sustain oscillating positive population levels in a limited geographic area when one species is a predator of the other and the second species consumes limited but regenerating resources from the environment?

Agent Types Sheep, Wolves, Grass

Agent Properties Energy, Location, Heading (wolf and sheep), Grass-amount (grass)

Agent Behaviors Move, Die, Reproduce (wolf and sheep), Eat-sheep (wolf only), Eat-grass (sheep only), Regrow (grass)

Parameters Number of Sheep, Number of Wolves, Move Cost, Energy Gain From Grass, Energy Gain From Sheep, Grass Regrowth Rate
Time Step:

1. Sheep and Wolves Move
2. Sheep and Wolves Die
3. Sheep and Wolves Eat
4. Sheep and Wolves Reproduce
5. Grass Regrows

Measures Sheep Population versus Time, Wolf Population versus Time

It is quite useful while designing a model to write notes to yourself, as we have done in this section. You will find these notes invaluable after you have left the model for a while, since you will be able to go back and recall why you made certain decisions and alternative choices that you considered. Also, for the purposes of explaining your model to others, it is very helpful to have such documentation about the model. Finally, we recommend that you date your notes as you create them so that you can track your model design process.

For instance, if you are using the top-down design process that we have just discussed, then you might look over the following set of questions and write down answers to them as a provisional guide for how to build your model:

1. What part of your phenomenon would you like to build a model of?
2. What are the principal types of agents involved in this phenomenon?
3. In what kind of environment do these agents operate? Are there environmental agents?
4. What properties do these agents have (describe by agent type)?
5. What actions (or behaviors) can these agents take (describe by agent type)?
6. How do these agents interact with this environment or each other?
7. If you had to define the phenomenon as discrete time steps, what events would occur in each time step, and in what order?
8. What do you hope to observe from this model?

A more bottom-up approach would not start with these questions. Instead, you might know only that you want to build some kind of ecological model and could begin by creating some sheep and have them spread out in the world. Next, you might think of and start implementing some behaviors for the sheep such as moving, taking steps, eating, reproducing and dying. In order for the sheep to eat, you might decide to add grass to the model. Having sheep eat grass provokes the question, "What eats sheep?" Therefore, you modify the model to include wolves, as predators for the sheep. This process allows the wolf sheep model to be designed and even implemented without first considering the final goal of the model.

Now that we have completed the initial design of the Wolf Sheep Simple model, the next step is to implement the model.

Building a Model Having designed our conceptual model, we can begin the implementation process. In the model building process as well, we will continue to apply the ABM design principle. Even though our model as described is fairly simple, we will break this model down into a series of submodels that we will implement over five iterations. These submodels will all be capable of running on their own and will enable us to build the complete model in steps, checking our progress along the way and making sure that the model is working as we hoped it would work.

Many times, in agent-based models, the end results are not what we expect. This can be due to an error in model implementation. But often it is neither our implementation nor our conceptual model that is wrong, but rather our surprise stems from a core property of complex systems—emergent behavior, which, as we have seen, is notoriously difficult to predict. By building up our model gradually we can observe unusual behavior and more easily determine its cause than if we had built the model all at once. Thus, the ABM design principle still applies throughout the model implementation process as well.

First Version What is the simplest form of our model that we could create that would exhibit some sort of behavior? One simplification that we can make from the total model is to only look at one species, and ignore the environment. Given these two simplifications, it seems that the simplest model would have some sheep wandering around on a landscape. To do this we create two procedures, a SETUP procedure, which creates the sheep, and a GO procedure that has them move.

The first thing we need to do is create a sheep *breed* in the NetLogo Code tab:

```
breed [ sheep a-sheep ]
```

This just says that a class of mobile agents (in NetLogo, turtles) called SHEEP exists. The plural form "sheep" is given first, followed by the singular form "a-sheep," which is a little awkward in the case of sheep. It will feel more natural when we add the "wolves"/"wolf" breed later. In your code, you will mostly need to use the plural form (SHEEP), but it is helpful to provide the optional singular form in the breed declaration as well, so that NetLogo can give you more meaningful error messages, among other things. One last thing to note is that even though we are creating SHEEP at this point, the agentset of TURTLES still exists. All mobile agents in a NetLogo model are TURTLES regardless of their breed. So if you want to ask all the moving agents to do something, i.e., both SHEEP and WOLVES, then you can ASK TURTLES. If you want to just ask the SHEEP, then you can ASK SHEEP, and if you want to just ask the WOLVES, then you can ASK WOLVES (once we create them).

After we have created the SHEEP breed, we can create the SETUP procedure:

```
;; this procedure sets up the model
to setup
    clear-all
    ask patches [ ;; color the world green
       set pcolor green
    ]
    create-sheep 100 [ ;; create the initial sheep
       setxy random-xcor random-ycor
       set color white
       set shape “sheep”
    ]
    reset-ticks
end
```

This SETUP procedure will end up being the longest procedure of the finished model, but its behavior is fairly straightforward. First, it clears the world. The *world* of the model is the representation of all the agents, including the mobile agents (e.g., turtles, sheep, wolves) as well as the stationary agents (e.g., patches, grass). As we saw in the previous chapters, the command CLEAR-ALL resets any variables in the model and readies it so that a new run can be executed. After this, all the patches are asked to set their patch color (PCOLOR) to green to represent grass. Even though our model does not yet include any grass properties nor any rules for sheep to interact with the grass, changing the color helps the visualization. Finally, we create one hundred sheep. When we create the sheep, we also give them some initial properties. We assign them a random x-coordinate and a random y-coordinate to spread them over the world. We then set their color to white and their shape to the “sheep” shape[5] so they look a little more like real sheep. The final line, RESET-TICKS, starts the NetLogo clock so the model is ready to run.

Documenting the procedures within your model (by using the semicolon to comment the code) is very useful. Any text written after a semicolon is ignored when the model is run; adding text in this way is called “commenting” your code. Without these comments, not only is it quite difficult for someone else to read your code, but it will also become more and more difficult, as time passes, for you to understand your own code. A model without comments (and other documentation) is not very useful, since it will be difficult for others to figure out what the model is trying to do.

After we have created the sheep, we go on to write a GO procedure to tell them how to behave. Looking back at our design document, one of the main behaviors the sheep exhibit is moving around, so we will have them do that. We break up the sheep movement

5. NetLogo has a default set of turtle shapes that is available to all models, and “sheep” is included in this. There is also an extensive library of additional shapes, which can be imported for the models use, or you can design your own custom shapes. See the “Turtle Shapes Editor” under the Tools menu.

into two parts, turning and moving forward. We create the procedures WIGGLE and MOVE, which we define later.

```
to go
    ask sheep [
       wiggle ;; first turn a little bit in each direction
       move   ;; then step forward
    ]
    tick
end
```

This asks all of the sheep to perform a series of actions: WIGGLE, then MOVE. The sheep will take their turns in random order, and each sheep will complete both actions before the next sheep takes its turn. After all the sheep have finished, the TICK command increments the model clock, indicating that one unit of time has passed. Of course, to make the code work, we must define WIGGLE and MOVE. The sheep will execute both of these procedures. As such, we document them as "sheep procedures," that is, procedures that do not explicitly ask any agents but are written with the implicit assumption that the calling procedure will ask the right set of agents to perform them.

```
;; sheep procedure, the sheep changes its heading
to wiggle
       ;; turn right then left, so the average direction is straight ahead
       right random 90
       left random 90
end

to move
       forward 1
end
```

The first procedure simply turns to the right a random amount between 0 and 90, and then back to the left a random amount between 0 and 90. The idea behind this turning behavior is to have the sheep change the direction they are heading, without bias toward turning either left or right.[6] This type of randomized turning is very common in ABMs, and could be called an ABM "idiom." In NetLogo, it is common to refer to this turning behavior as "WIGGLE." The second procedure simply moves the sheep forward one unit.[7]

6. Technically, RIGHT 90 LEFT 90 causes the new heading of the turtle to be randomly drawn from a binomial distribution, centered on the previous heading of the turtle. Qualitatively, this means that smaller turns (in either direction) are more likely than large turns. A binomial distribution is similar to a normal distribution in this respect.

7. Since the move procedure is just one command, we did not have to break it out into a separate procedure, but because in our design we had included other effects of the movement, we anticipate the more complex move procedure ahead by breaking it out at the outset.

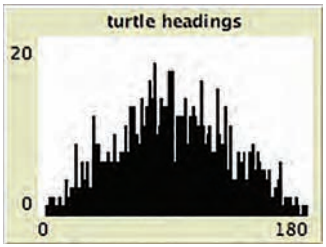

Figure 4.3
A plot of the headings of 1,000 turtles that have "wiggled." All turtles start at a heading of 90, and their headings after wiggling are distributed binomially around 90

As mentioned in the design section, the distance the sheep moves could later be controlled by a global parameter, but for now we will keep it to a constant single unit (the width of one patch). (See figure 4.3.)

You can go ahead and run this model right now. You can just type "setup" and then "go" in the command center.[8] To make this model easier to use, it helps to create GO and SETUP buttons in the model's Interface tab. In the GO button, we check the "forever" check box so that the GO procedure will indefinitely repeat. After this, your model should look like figure 4.4.

Second Version Now that we have our sheep moving around, we have something we can see and we have a first verification that our model is working as we intended it to work. Next we need to consider what is the simplest extension that we can develop that follows our design. We have the sheep moving around but, in our first version, movement does not cost them anything. In the real world, movement does require energy. Therefore, the next step in our model development is to include a movement cost. Recall that we designed the sheep to have three properties: heading, location, and energy. The sheep in the first version of this model already have headings and locations—these properties are automatically provided to any NetLogo "turtle" agents. However, we have to define a new property (variable) for energy, which we can do by adding this code near the top of our model's procedures:

```
sheep-own [ energy ]    ;; sheep have an energy variable
```

8. As was explained in the NetLogo tutorials, the command center is where you can type single command lines to test their effects.

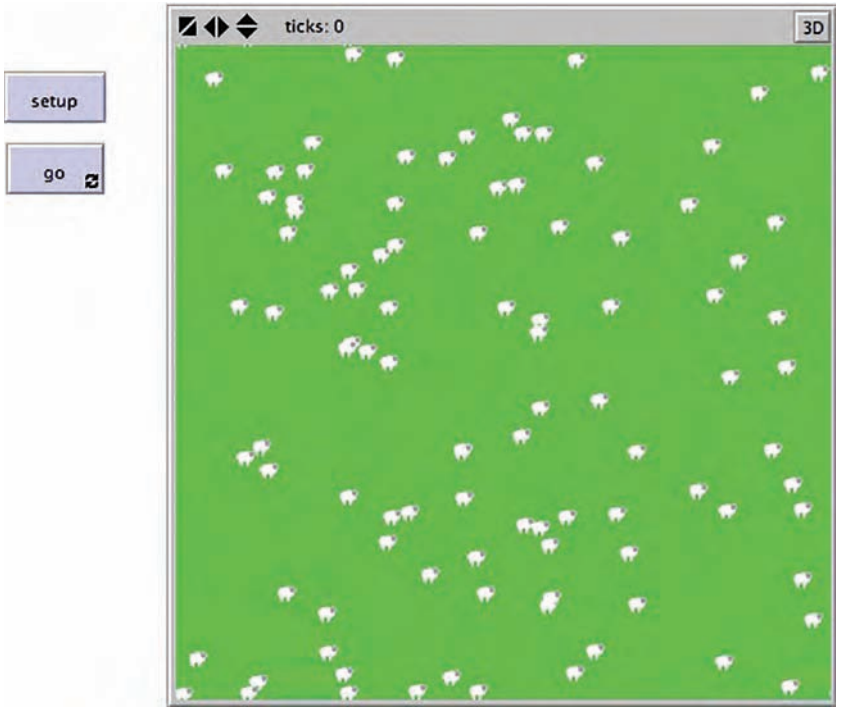

Figure 4.4
The first version of the Wolf Sheep Simple model. (See the supplementary materials.)

Just declaring that sheep have energy is not enough. We also need to initialize the energy variable and make it change when sheep move. While we are editing the code, we will also make it so that the number of sheep initially created is based on a NUMBER-OF-SHEEP slider on the model's Interface tab, rather than hard-coded to be 100. This will allow us to change the number of sheep in the model easily from the interface, because, as mentioned in the Design section earlier, the number of sheep is a model parameter that we would like to be able to manipulate. In creating the slider widget, we need to give it some properties, a minimum value for the NUMBER-OF-SHEEP initially created, a maximum number, and an increment, which is the amount the slider will change when you click on it. In this case, we can set the minimum number to 1 (since less than one sheep makes no sense), the maximum number to start with to 1,000 and the increment to 1, since it makes no sense to have e.g., 2.1 sheep. (See figure 4.5.)

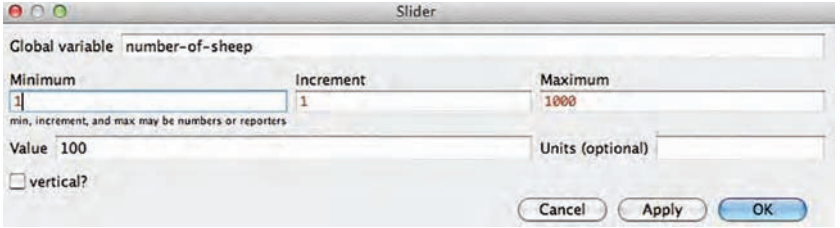

Figure 4.5
Setting up the NUMBER-OF-SHEEP slider.

The modified SETUP procedure is thus:

```
;; this procedure sets up the model
to setup
    clear-all
    ask patches [ ;; color the whole world green
        set pcolor green
    ]
 ;; create the initial sheep and set their initial properties
 create-sheep number-of-sheep [
       setxy random-xcor random-ycor
       set color white
       set shape "sheep"
       set energy 100
    ]
    reset-ticks
end
```

We also need to add one line to the MOVE procedure to give a cost to movement:

```
;; sheep procedure, the sheep moves which costs it energy
to setup
  forward 1
  set energy energy - 1   ;; take away a unit of energy
end
```

We can start by setting the cost to one unit, knowing that as we extend the model, we will want to make the movement cost a parameter of the model. Adding a cost to movement does not mean anything if there is no penalty for expending energy. We want the sheep to die if they have too little energy. Therefore, we also need to check to see if the sheep have expended all of their energy. We can modify the GO procedure to call a subprocedure that checks whether a sheep should die.

We write this procedure:

```
to go
ask sheep [
       wiggle ;; first turn a little bit
       move   ;; then step forward
       check-if-dead ;; checks to see if sheep dies
    ]
tick
end
```

We also need to write the check-if-dead procedure itself:

```
;; sheep procedure, if my energy is low, I die
to check-if-dead
    if energy < 0 [
       die
    ]
end
```

Now if we press SETUP and GO, the model will run for a while and then all the sheep will disappear at the same time. Unfortunately the model will keep running (you can tell because the GO button remains depressed). It would be nice if the model would stop when there were no more sheep. We can add a clause to the GO procedure to do that:

```
to go
    if not any? sheep [stop]
    ask sheep [
    wiggle
       move
       check-if-dead ;; checks to see if sheep dies
    ]
    tick
end
```

Now, if you rerun the model, when all the sheep disappear the model stops running. It might be useful to know how many sheep there are, so we can add a plot that indicates what the population of the sheep is at any point in time. If you remember, we also created a plot in chapter 2, but in that chapter we used code within the plot widget to control the updating of the plot. In NetLogo, there are two ways to handle plotting. The chapter 2 method is referred to as the *widget-based* method, since it makes use of a graphic widget. The other method is the *programmatic* or *code-based* method. In both methods, code is

written to update the plots, but in the *code-based* method the code is actually located in the NetLogo Code tab. In the widget-based method that we saw in the previous chapters, the code is located inside the plot widget itself. In general, each method of plotting can do any plotting task the other can do, so it is up to the modeler to decide which method to use. They both have advantages and disadvantages. The advantage of the widget-based method is that you do not need to clutter up the Code tab with extra code for plotting, and, that for simple plots, it is quicker to set up. However, for complex plots with many pens, it can be more difficult to set up a plot with the widget-method. Furthermore, if there is buggy code in the widget plot, it can be hard to notice, since it won't show up as an error in the Code tab. The Code tab plotting method has the reverse advantages and disadvantages. We regard this choice as a stylistic choice for the modeler. Our general recommendation is to use the widget for relatively simple plots and the Code tab for complex plots with intricate setup conditions and/or many pens.

In this chapter, we will introduce the programmatic method of updating the plot using plotting code written in the Code tab. Everything we do here in the Code tab can also be done using the widget-based approach, but it is useful to know both methods so you can decide which method best suits your model. To plot using the programmatic method, we first create a plot on the interface and set its properties, then we add a call to a plotting routine in the main GO procedure:

```
to go
    if not any? sheep [
       stop
    ]
    ask sheep [
       wiggle
       move
       check-if-dead ;; checks to see if sheep dies
    ]
    tick
    my-update-plots ;; plot the population counts
end
```

Then we need to define the procedure MY-UPDATE-PLOTS (the "update-plots" primitive does something similar for widget-based plotting):

```
;; update the plots in the interface tab
to my-update-plots
    plot count sheep
end
```

We could have used "plot count sheep" directly at the end of the GO procedure. But we know it is likely that we will have to plot other population counts, so we can look

ahead and create the "my-update-plots" procedure. We can then use this procedure later on for other plots as well. Now, if we run the model, the plot shows us that there was a full population of sheep up until the end of the run, when they all died out. The death of all of the sheep in the 101st time step is a result of the initial energy and the movement cost that we have assigned to the sheep (set to 1 energy unit per movement step). Recall that we wanted the movement cost to be a parameter of the model. We need to add another slider to control the movement cost and then modify the MOVE procedure to take this in to account:

```
to move
    forward 1
    ;; reduce the energy by the cost of movement
    set energy energy - movement-cost
end
```

At the end of the run, your model should look like figure 4.6.

Third Version At present, the model exhibits a very predictable behavior. Every time the model runs for 100/MOVEMENT-COST time steps (ticks), then all of the sheep disappear and the model stops. The reason is because the sheep currently expend energy

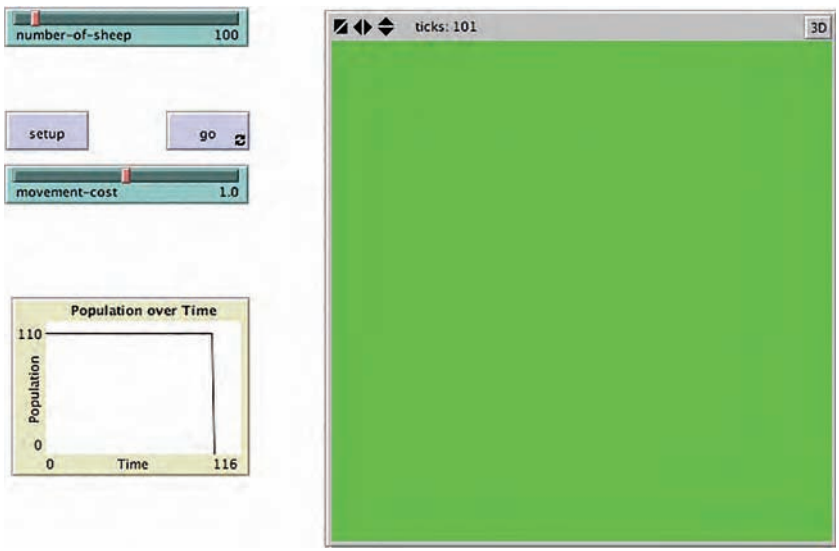

Figure 4.6
The second version of the Wolf Sheep Simple model at the end of a run. (See Wolf Sheep Simple 2 model in the supplementary materials.)

(by moving) but have no way of gaining energy. Therefore, we need to give the sheep the ability to eat grass and gain energy. However, first we must create the grass. To do this, we tell NetLogo that the patches (which serve as the grass clumps for this model) have a GRASS-AMOUNT property, which measures how much grass is currently available on that patch by adding the following line after the SHEEP-OWN line we already have:

```
patches-own [ grass-amount ]   ;; patches have an amount of grass
```

Then we need to set up this grass, and, while we are at it, we will modify the color of the patches so that they indicate how much grass is available. We do this by setting the initial amount of grass to a random (floating-point) number between 0.0 and 10.0.[9] We use a floating-point number for grass, since unlike sheep, which are individuals, each patch contains a "clump" of grass, not an individual blade. This ensures some variability in the amount of grass and creates some agent heterogeneity. Then we set the color of the grass to a shade of green such that if there is no grass at all the patch will be black and if there is a lot of grass it will be bright green:[10]

```
to setup
    clear-all
    ask patches [
       ;; patches get a random amount of grass
       set grass-amount random-float 10.0
       ;; color it shades of green
       set pcolor scale-color green grass-amount 0 20
    ]
    create-sheep number-of-sheep [
       setxy random-xcor random-ycor
       set color white
       set shape "sheep"
       set energy 100
    ]
    reset-ticks
end
```

9. "Floating-point" technically refers to the representation used by the computer to store a number in memory, with a floating decimal (binary) point. However, for most practical purposes, what you need to know is that RANDOM N reports a random nonnegative integer less than N, whereas RANDOM-FLOAT X reports a random real number less than X, such as 0.9997 or 3.14159.

10. As we saw in the El Farol model extensions, SCALE-COLOR takes four inputs, a color that we are scaling, the variable that determines how bright or dark to make the color, a lower value of the variable and an upper value of the variable. Here we set the upper value of the variable to 20 even though the most it can be is 10. This means that SCALE-COLOR will only use half the range of GREEN. If we allowed it to use the full range then patches with lots of grass would be white and not bright green.

Now we need to modify the GO procedure so that the sheep can eat the grass. As we mentioned in designing a time step, we put this procedure after the check for death:

```
to go
    if not any? sheep [
       stop
    ]
    ask sheep [
       wiggle
       move
    check-if-dead
    eat
    ]
tick
    my-update-plots
end
```

Then we write the EAT procedure:

```
;; sheep procedure, sheep eat grass
to eat
    ;; check to make sure there is grass here
    if ( grass-amount >= 1 ) [
       ;; increment the sheep's energy
       set energy energy + 1
       ;; decrement the grass
       set grass-amount grass-amount - 1
       set pcolor scale-color green grass-amount 0 20
    ]
end
```

This procedure just checks to see if there is enough grass in the patch below the sheep. If there is enough there to eat, then the sheep converts the grass into added energy, and we decrement the amount of grass in the patch. At the same time we recolor the patch to reflect the new amount of grass in the location.

The model behavior is still not very interesting. The sheep wander around, eat as much grass as they can, and eventually all die out. The only variation in the model is the level of the grass in the patches. Due to the random distribution of grass originally and due to the fact that the sheep move randomly around the landscape, there will be some areas of grass that are completely consumed by the sheep and other areas that will be only partially consumed.

To make the model a little more interesting we add in a procedure to make the grass agents regrow. By allowing the grass to regrow it should be possible to maintain the sheep

population over time since there will be a renewable source of energy for them. We begin by modifying the GO procedure:

```
to go
    if not any? sheep [
       stop
    ]
    ask sheep [
       wiggle
       move
       check-if-dead
       eat
    ]
    regrow-grass ;; the grass grows back
    tick
    my-update-plots ;; plot the population counts
end
```

Then we define the REGROW-GRASS procedure:

```
;; regrow the grass
to regrow-grass
    ask patches [
       set grass-amount grass-amount + 0.1
       if grass-amount > 10 [
           set grass-amount 10
       ]
       set pcolor scale-color green grass 0 20
    ]
end
```

This procedure simply tells all of the grass clump agents to increase the amount of grass that they have by one tenth of one unit. We also make sure that the grass never exceeds 10, which represents the fact that there is a maximum amount of grass that can exist in any one clump. It then changes the color of the grass to match the new value. With this small change the sheep persist throughout a run of the model. We now have the grass recolor code at three different places in the model, so it would also be nice to place that in its own procedure. Often when we start to duplicate code, it is worth placing it in a separate procedure; that way we have to modify the code in only one location if we need to change it later (for instance, if we want grass to be colored yellow instead of green). Keeping the code more concise, and placing useful pieces of code in appropriately named subprocedures, will also help make your code more readable for others. So we define a RECOLOR-GRASS procedure:

```
;; recolor the grass to indicate how much has been eaten
to recolor-grass
    set pcolor scale-color green grass-amount 0 20
end
```

Now we just replace the coloring code in SETUP, REGROW-GRASS, and EAT with RECOLOR-GRASS, and the model works the same as before.

Running the model several times with one hundred initial sheep, it becomes clear that one hundred sheep cannot consume all the grass, and thus eventually the whole world becomes a solid shade of green. However, if you increase the number of initial sheep to a larger number, say seven hundred, and then run the model, the sheep will consume almost all the grass in the model, and then many of them will die off. However, a few of them that had a large amount of energy before all the grass disappeared will survive, and eventually the grass will regrow permitting them to persist since they are no longer competing with as many sheep for the grass.

Another parameter that we want to introduce, which may affect the dynamics of the model as much as the initial number of sheep, is the rate at which grass regrows. To do this we add a slider called GRASS-REGROWTH-RATE, give it boundary values of 0 and 2 and an increment of 0.1, and then we modify the REGROW-GRASS procedure to reflect the use of this new parameter:

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

Now if we set the GRASS-REGROWTH-RATE to a high enough value (try 2.0), then even with seven hundred sheep in the model, the full sheep population can be sustained. This is because the sheep are able to gain a full unit of energy from the grass, which regrows that energy in one time step. The sheep then expend that energy in the next time step moving, but that energy is immediately replaced. However, if you change the MOVEMENT-COST slider to be greater than 1.0, then the sheep will eventually die off. This is because they are expending energy faster than they can gather it from the environment, even if there is no shortage of grass. In order to make our model more flexible, we can add yet another parameter, ENERGY-GAIN-FROM-GRASS, which will control the amount of energy the sheep can gain from eating the grass. As in the previous

sliders, we will need to set reasonable bounds and an increment for this slider as well. To use this new slider, we need to modify the EAT procedure:

```
;; sheep procedure, sheep eat grass
to eat
    ;; check to make sure there is grass here
    if ( grass-amount >= energy-gain-from-grass ) [
       ;; increment the sheep's energy
       set energy energy + energy-gain-from-grass
       ;; decrement the grass
       set grass-amount grass-amount - energy-gain-from-grass
       recolor-grass
    ]
end
```

Note that we used the energy-gain-from-grass parameter both to increment the sheep's energy gain from eating as well as to decrement the grass's value. We could have used two different parameters for these two functions, but we can think of the sheep/grass system as energy conversion, so that the energy in the grass flows to the sheep. Now we can get some interesting dynamics. For instance, in figure 4.7, you can see an instance of the run where we started out with seven hundred sheep, and they lasted for around three hundred time steps. But then there was a mass starvation, which became more gradual, until after around

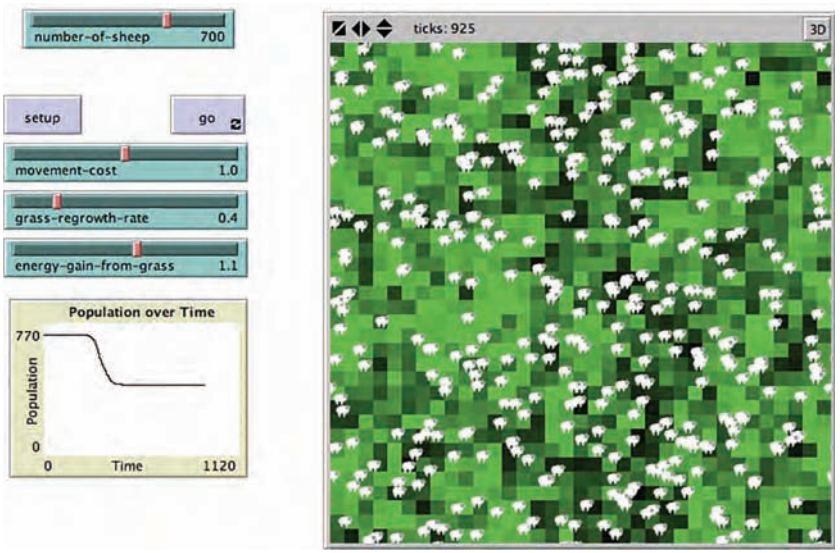

Figure 4.7
Third version of the model, having reached an equilibrium. (See Wolf Sheep Simple 3 model in the supplementary materials.)

five hundred ticks, the population held steady with a little over four hundred sheep. After enough sheep have died out, the grass can continually regenerate and support the living sheep and the system has reached a state of equilibrium. Since sheep movement is random, it is possible that a large number of sheep might happen to cluster together on the same few patches for a long time, and thus starve, but this is not likely. Depending on your choices for the model parameters, many other outcomes are also possible. Feel free to experiment and explore before moving on to the next version of the model.

Fourth Version So now the model has sheep moving around on a landscape, consuming resources, and dying. However, there is no way for the sheep population to go back up; currently it can only go down. Thus to get it to go back up, we will add reproduction to the model.

To build a full reproductive model with sexual pairings and to have a pregnant sheep would take a long time, and it is not clear that it would be worth the effort to answer the question we posed at the outset of making our model. Instead, we will make two simplifying assumptions. First, single sheep can produce new sheep. You can view this as either asexual reproduction or you can think of each sheep as representing a life-bonded pair of sheep. This assumption may seem strange at first and is certainly obviously contrary to reality. This is a good time to recall George Box's words: "all models are wrong, but some are useful." It is OK to make our model wrong about such a basic fact of reproduction if the simplified model is still useful. If later we see that this simplification lost us some usefulness of the model, we can always add sexual reproduction later. The second simplifying assumption is this: Rather than worry about gestation, we will assume that sheep immediately give birth to a new lamb when they reach a certain energy level. This energy level can be seen as a proxy for having the ability to gather enough resources to make it all the way through the gestation period.

To implement this we begin by adding code to the GO procedure:

```
;; make the model run
to go
    if not any? sheep [
       stop
    ]
    ask sheep [
       wiggle ;; first turn a little bit
       move   ;; then step forward
       check-if-dead   ;; check to see if sheep dies
       eat                          ;; sheep eat grass
       reproduce    ;;sheep reproduce
]
    regrow-grass ;; the grass grows back
    tick
    my-update-plots ;; plot the population counts
end
```

Then we need to write the REPRODUCE procedure:

```
;; check to see if this sheep has enough energy to reproduce
to reproduce
    if energy > 200 [
       set energy energy - 100   ;; reproduction transfers energy
       hatch 1 [ set energy 100 ] ;; to the new sheep
    ]
end
```

This code checks to see if the current sheep has enough energy to reproduce (twice the original amount of energy). If the sheep does then it decrements its energy by 100, and creates a new child sheep (HATCH makes a clone of the parent agent on the same patch) and sets its energy to 100.

Now if we run the model with a low energy movement cost compared to the rate of energy gain from grass, starting from 700 sheep, the population increases over time, and eventually levels off near 1,300 sheep, as shown in figure 4.8.

Fifth Version Now we essentially have the sheep working the way we described in our conceptual model but our original goal was to have two species. So now we need to add

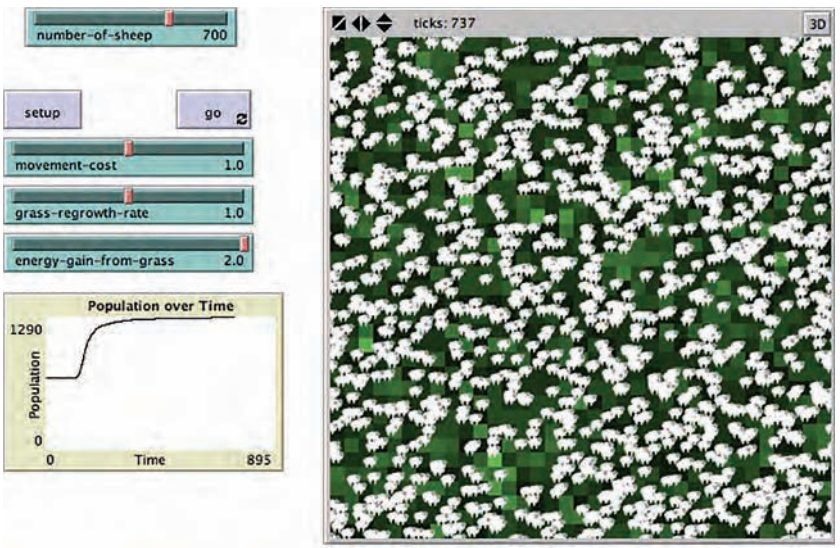

Figure 4.8
The fourth version of Wolf Sheep Simple model (includes reproduction). (See Wolf Sheep Simple 4 model in the supplementary materials.)

in the wolves. The first thing we need to do is tell NetLogo that there is now a second breed of turtles that we are calling wolves. At the same time we need to give wolves ENERGY as well. We could do this by adding a WOLVES-OWN like our SHEEP-OWN statement, but since the only turtle agents in the model are sheep and wolves, we can make ENERGY a generic property of all turtles. We do this by changing our SHEEP-OWN statement to a TURTLES-OWN statement:

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

turtles-own [ energy ]    ;; agents own energy
```

After that we need to create the wolves, just like we did the sheep. First, we add a NUMBER-OF-WOLVES slider, and then we modify the setup procedure:

```
;; this procedures sets up the model
to setup
    clear-all
    ask patches [
       set grass random-float 10.0 ;; give grass to the patches
       recolor-grass ;; change the world to green
    ]
    create-sheep number-of-sheep [ ;; create the initial sheep
       setxy random-xcor random-ycor
       set color white
       set shape "sheep"
       set energy 100 ;; set the initial energy to 100
    ]
    create-wolves number-of-wolves [ ;; create the initial wolves
       setxy random-xcor random-ycor
       set color brown
       set shape "wolf"
       set size 1.5  ;; increase their size so they are a little easier to see
       set energy 100   ;; set the initial energy to 100
    ]
    reset-ticks
end
```

Now that we have added wolves to the model, we need to add in their behaviors as well. We note that all of the behaviors are common to both the wolves and sheep, even if the exact details differ (e.g., wolves eat sheep, while sheep eat grass, but both "eat"). So we replace "sheep" in our GO procedure with "turtles," since all of our moving agents will execute these behaviors.

```
;; make the model run
to go
    if not any? turtles [ ;; this time check for any turtles
       stop
    ]
    ask turtles [
       wiggle ;; first turn a little bit
       move  ;; then step forward
       check-if-dead    ;; check to see if agent should die
       eat      ;; wolves eat sheep, sheep eat grass
       reproduce  ;; wolves and sheep reproduce
    ]
    regrow-grass ;; regrow the grass
    tick
    my-update-plots ;; plot the population counts
end
```

We note that all of the behaviors that we gave to sheep apply equally well to wolves, so the model will run as is. However, the eating behavior for the wolves is different from the eating behavior for the sheep, so we will need to modify our "eat" procedure.

```
;; sheep eat grass, wolves eat sheep
to eat
    ifelse breed = sheep [
       eat-grass
    ]
    [
       eat-sheep
    ]
end
```

Now our eat behavior will be different for sheep and wolves. The sheep will eat grass and the wolves will eat sheep. We rename our old "eat" procedure to "eat-grass" as that is the behavior we defined. We now must define the "eat-sheep" behavior. We start doing this by adding a slider for ENERGY-GAIN-FROM-SHEEP, just like the ENERGY-GAIN-FROM-GRASS slider we added earlier, and we write the EAT-SHEEP procedure:

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

In this procedure the wolf first checks to see if there are any sheep available to eat on the patch it is on. If there are, then it kills one of them (chooses one randomly from the sheep on the patch) and gets an energy increase according to the energy gain parameter.

Now our model has all of the agents, behaviors, and interactions that we had set out to create. However, our graph does not contain all the information yet. It would be helpful if it also displayed the wolf population, and at the same time we can add a display of the amount of grass in the world. To do this we first add two additional pens to the population plot, WOLVES and GRASS. We also rename the default plot pen to SHEEP. Then we modify the MY-UPDATE-PLOTS procedure:

```
to my-update-plots
    set-current-plot-pen "sheep"
    plot count sheep

    set-current-plot-pen "wolves"
    plot count wolves * 10 ;; scaling factor so plot looks nice

    set-current-plot-pen "grass"
    plot sum [grass] of patches / 50  ;; scaling factor so plot looks nice
end
```

This code is fairly straightforward. The "* 10" and "/ 50" are just scaling factors so that the plot is readable when all of the data is plotted on the same axis. (But keep in mind, when reading the number of wolves off of the plot, the actual population count is ten times smaller.)[11] It is often useful to add monitors for these variables as well to be able to read off exact values. We can now experiment with a variety of parameter settings for the Wolf Sheep Simple model. Many parameter settings will result in extinction of one or both species. But we can find parameters that result in a self-sustaining ecosystem where the species' population levels vary in a cyclic fashion. One such set of parameters is shown in figure 4.9. With those parameters, the wolf and sheep populations are sustained and oscillate.

Examining a Model

We have found one set of parameters that exhibits the behavior of our reference pattern. We note that these particular values for the parameters do not correspond to any real predator and prey populations. We did not calibrate our model from real-world data, so the parameter values themselves are not important. But, discovering that there exist model

11. Note that all of this plotting could also be done within the widget-based method.

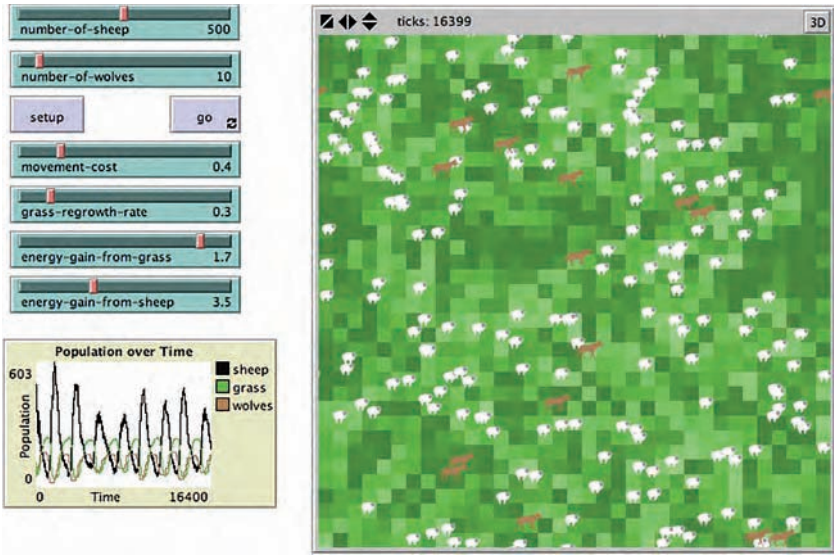

Figure 4.9
The Wolf Sheep Simple model, now including wolves. (See Wolf Sheep Simple model 5 in the supplementary materials.)

parameters that exhibit our reference pattern gives us insight into the natural phenomenon we were trying to model. Now that we have built our model, and we have found a set of parameters that allow the wolf and sheep populations to coexist and to oscillate, we have partially answered our research question: What parameters of two species will sustain oscillating positive population levels in a limited geographic area when one species is a predator of the other and the second species consumes limited but regrowing resources from the environment? We now know that it is possible for the rules we set up to produce the target reference pattern and therefore they could be a possible generative explanation for that pattern. However, just observing the behavior of this model once does not provide us with a robust answer. First, because many components of our model are stochastic in nature, there is no guarantee that the model when run again with the exact same parameters will exhibit the same behavior. Second, we have found one set of parameters that allow wolves and sheep to exist, but are there other parameter settings? A more general answer would allow us to make statements about ranges of parameters, and relationships between parameters, that allow both the wolves and the sheep to survive. However, if we run the model many times with a variety of parameter settings, this will create a lot of data. Thus, to give us even a simple answer to our question, we really need to examine multiple runs across multiple different parameter settings and summarize this data in a useful way. We will explore data analysis in greater detail in chapter 6, but before we leave the Wolf Sheep

Simple model, we will perform a basic analysis, in order to develop a preliminary answer to our question. With this analysis, we will have taken a model from the very beginning stages of its design to a first set of actual results. Chapter 6 will revisit many of these topics in greater detail.

Multiple Runs

Whenever you have a model that has stochastic components, it is important to run the model several times so that you can be certain you have correctly characterized the behavior of the model. If the model is run only once, you might happen to see anomalous behavior that is not what the model usually produces. For instance, in the Wolf Sheep Simple model it might be possible to run the model and arrive at a state in which there is simply one wolf and one sheep, and the sheep produces a second sheep often enough to keep the wolf fed, but not often enough to produce three sheep. However, due to the way the wolves and the sheep wander around the landscape, such an outcome is extremely unlikely, and it would not be typical of the model. Thus, it is important to run the model multiple times so that you can characterize the normal/average behavior of the model and not the aberrant behavior of the model.[12] On the other hand, there are times when the anomalous/aberrant model behavior *is* what you are interested in investigating, in which case you will need to run the model many times in order to find it and characterize it.

Most ABM platforms provide a way to do this. NetLogo provides you with the BehaviorSpace tool (*Wilensky & Shargel, 2002*) that enables you to run a model for several iterations with the same (or different) parameter settings and collect the results. We will learn to use the BehaviorSpace tool in chapter 6.

One additional consideration when performing multiple runs is how many time steps to run the model for. Since we want to be able to compare the results across different random number seeds, it is useful to hold the number of time steps we run the model constant. How long to run the model, how many times to run the model, and how to average the model results are nontrivial questions when you are attempting to describe the behavior of a model. These questions will be explored in chapter 6, but for now let us take the Wolf Sheep Simple model and run it with the parameter settings mentioned before for a thousand time steps ten times, and let us output the wolf, sheep, and grass population at the end of the model run.

Parameter Sweeping and Collection of Results As we mentioned in the introduction, just because you have found one set of parameters that seem to answer your question does not

12. This is true not just of ABM but also of stochastic modeling in general. To learn more about the history and methods of stochastic simulation (sometimes known as Monte Carlo methods), see Hammersley & Handscomb, 1964; Kalos & Whitlock, 1986; Metropolis & Ulam, 1949.

mean that you are done. Often there are other sets of parameters that will also result in a similar behavior. On the other hand, maybe these particular parameter settings are unique and other parameter settings, even ones that are close to the current settings, will result in very different behavior. Thus, it is important to examine critical parameters in the model to explore the robustness of the model behavior and to understand how sensitive the model is to changes in parameters.

Robustness and sensitivity analysis will be explored more in chapter 6, but for now we can concentrate on one important factor. In the Wolf Sheep Simple model, one parameter that seems to affect the behavior at times is the initial number of wolves. If there are too many wolves, then they will eat all the sheep, and then they will die for lack of a food source. If there are too few wolves, then the population may not survive until the sheep have increased their population enough to sustain them. To examine this effect let us run the model ten times for each of the eleven different values of the initial number of wolves parameter from 5 to 15.

Analysis of Data Summarizing data can be done in a variety of ways. Not only does data analysis enable us to describe complex data results in a much more compact form, but it also gives us a uniform way of looking at data so we can compare and contrast data sets.

Having lots of data is nice, but it is difficult to make claims about three (3) different variables with ten (10) different random number seeds and eleven (11) different initial parameter settings. Altogether that combination produces 330 different values. Thus, we need to summarize and analyze this data in some way, so that it is comprehensible. One typical way to summarize the data is to average it across the runs. This turns ten values into two values if we express the results in terms of the average and standard deviation. Another method for summarizing data is to plot it on a graph. Thus, we plot the initial parameter values on the x-axis and the values themselves on the y-axes. Given these two simplifications we have reduced our 330 different values into three plot lines. This data is now much easier to understand, and is observable in figure 4.10 (the results have been scaled by the same scaling factors we used in the Wolf Sheep Simple model plot). Of course, it is possible to examine all of the data that the model produces, and we will discuss ways of doing so later in the book.

Now we have designed a model, implemented it, and conducted a basic analysis in order to answer a question of interest. Since in many cases there are oscillatory patterns in the wolves, sheep, and grass counts, the final numbers are not always relevant, but they can be useful as a starting point for investigation. For instance, though all that is plotted in what follows is the average of the final numbers, if you were to examine the variance of those numbers you might have more insight into how oscillatory the patterns were at the end of a run. In chapter 6, we will go into much more detail about how to do model analysis.

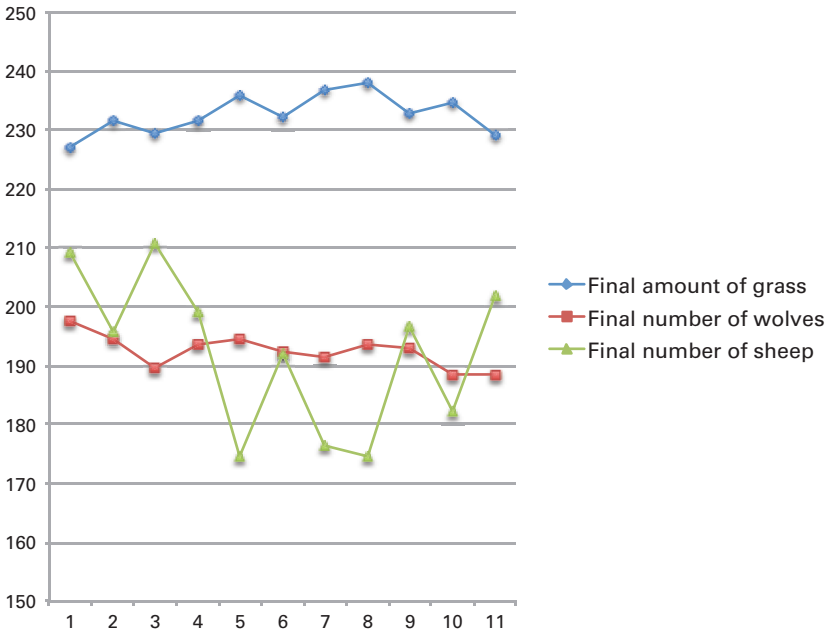

Figure 4.10
Results of Wolf Sheep Simple model analysis.

Predator-Prey Models: Additional Context

One of the first uses of agent-based modeling was for modeling of ecosystems, where it was often called individual-based modeling (see the appendix for a brief discussion of the historical role of individual-based modeling). Modeling of ecological systems has been of interest to biologists and environmentalists for quite some time. If we can better understand how ecologies operate and what the effects of successful ecosystems on the global environment are, then we may be able to better intervene to assist in the sustenance of these systems. For more information on this topic, refer to Grimm and Railsback (2005).

As discussed briefly in chapter 0, one of the first attempts to study ecologies and population dynamics in a concrete way was the work of Lotka (1925) and Volterra (1926). They developed a system of equations to describe a two species predator-prey interaction. These simple equations showed that you could meaningfully model ecological systems with just a few parameters. And once you have a model of a system you can start to explore options for perturbing the system into favorable states. The general result of the Lotka-Volterra equations is that predators and prey populations move in cycles. This is seen in figure 4.11.

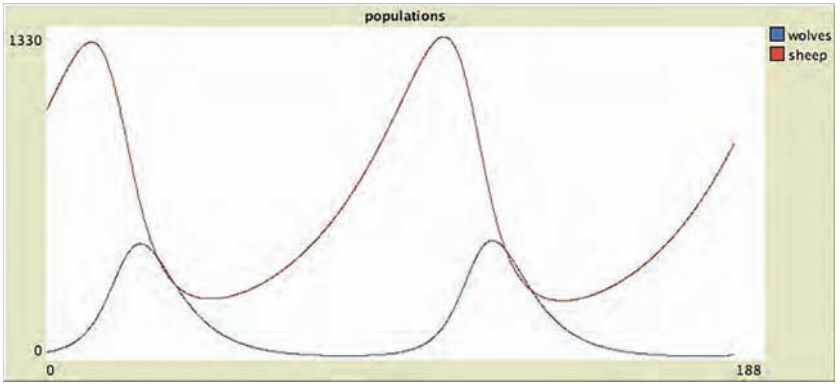

Figure 4.11
Lotka-Volterra relationship (Wolf Sheep Predation System Dynamics model) (*Wilensky, 2005*).

Depending on the parameter settings, if the two populations are equal at the start, then the predators will begin consuming the prey; this will result in a decrease of the prey population. Eventually, the prey will die off to the point where predators cannot find any more prey to eat. This will in turn cause predators to die off because they have no prey to feed on. At this point the prey will be able to reproduce before the predators eat them, and the prey population will increase. As the prey population increases it will become easier for the predator population to find them and the predator population will increase as well. This, in turn, results in the prey population's decreasing trend, and the whole process begins again.

The Lotka-Volterra equations have been a standard way of describing the fluctuations in predator and prey populations. Because they are differential equations (see figure 4.12), they represent a continuous model of population change. But populations are not continuous; they are discrete. Sometimes a continuous approximation of a discrete process is fine, but in this case it may be an oversimplification. The general problem is that as the population of a species becomes very small, standard differential equation models do not allow the population to actually go to 0, which means that there is always a positive probability that the population will rebound. However, this is not what happens in the real world. There is no rebounding from 0.1 prey. If a species goes extinct, it goes extinct, and there is no probability that it will rebound. This phenomenon is so common in differential equation-based modeling that it is sometimes referred to as the "nano-sheep problem" in specific reference to the wolf-sheep/predator-prey model (Wilson, 1998). The problem is that nano-sheep (i.e., a millionth of a sheep) do not exist: Sheep either exist or do not exist, and therefore modeling them with a continuous distribution can be problematic.

To rectify the nano-sheep problem, biologists have created agent-based (or to use the term biologists favor, individual-based) models of predator-prey relationships, similar to

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \delta xy - \gamma y$$

Figure 4.12
Classic Lotka Volterra differential equations. x is the number of prey, y is the number of predators, and α , β, γ, and δ are parameters describing the interactions of the two species.

the model that was built in this chapter. Under certain conditions these models have been shown to replicate the Lotka-Volterra results, but without the nano-sheep problem. They also predict that this system is in fact highly susceptible to extinctions, something that the Lotka-Volterra equations fail to capture (Wilson et al., 1993; Wilson, 1998). In 1934, Gause showed that indeed for isolated predator and prey (with no other competing species) the agent-based model was more accurate in its predictions—indeed, extinctions happen much more frequently than the Lotka-Volterra equations predict. The simple agent-based model we have created in this chapter shows similar results. For many parameter values, either wolves or both sheep and wolves go extinct, but for some sets of parameter values, the population levels are sustained and oscillate.

Advanced Modeling Applications

More broadly, the modeling of environmental and ecological systems has a long and rich history, representing some of the first applications of an agent-based modeling paradigm (DeAngelis & Gross, 1992). It also continues to be an exciting area for current research. One use of agent-based models in ecological systems is the modeling of food webs (Yoon et al., 2004). This combines ABM with another new complex systems methodology, network analysis (Schmitz & Booth, 1997). By understanding both the structure of ecological interactions via network analysis and the process of animal interactions via ABM, researchers gain a deeper insight into overall ecological systems (See, e.g., figure 4.13). Another particularly interesting application of ABM and ecological modeling has been in doing prescriptive design of engineered systems to ameliorate human interventions in the environment. For instance, Weber and his colleagues have done significant modeling of fish populations, and then examined the effect of various fish ladders near dams on salmon populations (Weber et al., 2006).

Ecological models can also be underpinnings of models of evolution. Agent-based modeling has been frequently used to model evolution of organisms (Aktipis, 2004; Gluckmann & Bryson, 2011; Hillis, 1991; Wilensky & Novak, 2010). Evolution lends itself well to the ABM approach as natural selection and other evolutionary mechanisms can be thought of as computational algorithms. Models can be used to try to understand adaptation and speciation in the historical record. Figure 4.14 shows two examples of such

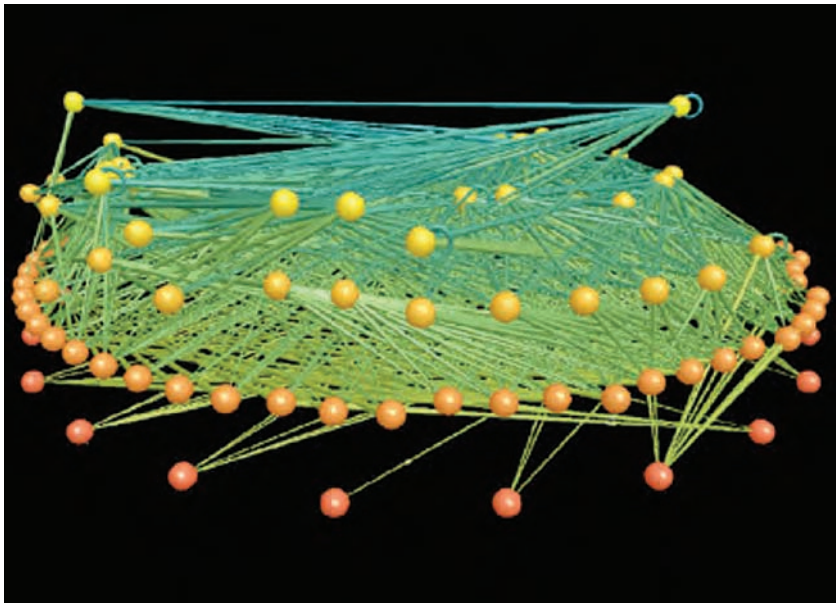

Figure 4.13
Little Rock Lake food web (foodwebs.org, 2006).

models of evolution from the NetLogo models library. In the field of Artificial Life, scientists sometimes evolve artificial organisms *in silico*. Moreover, as we will see in chapter 8, mechanisms inspired by evolution can be employed as methods of machine learning.

Conclusion

The Wolf Sheep Simple model was designed using the ABM design principle: Start simple and build toward the question you want to answer. We showed how this principle guided us not only in the design of the model, but also in its implementation and analysis. To design ABMs requires a new way of thinking about modeling, but it is more natural in some ways because it simply asks us to think like an agent (Wilensky & Reisman, 2006). Thus, we do not need to guess at the causal relationship between our model and the real world. Instead, by thinking as an agent and encoding the decisions that those agents make in the real world, we construct our model from the bottom up. This is true both when designing our ABM but also when implementing it. Finally, once our ABM has been constructed, we observe its behavior and analyze the results. This analysis may cause us to rethink some of our decisions about the ABM design and revisit our original design.

The Wolf Sheep Simple model is not only meant to be a basis for understanding the concepts and principles of constructing your own agent-based model, but it is also meant

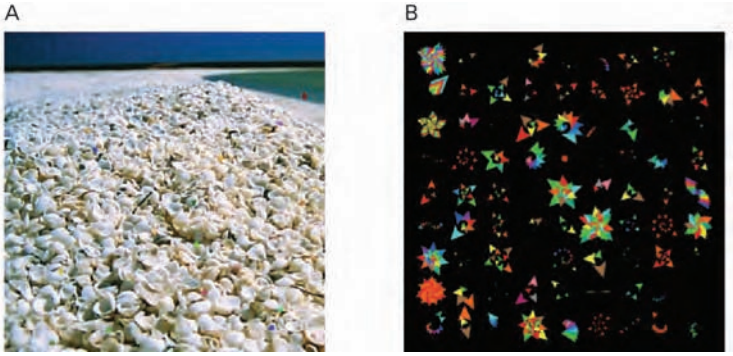

Figure 4.14
Agent-based models of evolution. (A) Evolution of camouflage of insects on a landscape. Different landscapes lead to the bugs evolving colorations that camouflage them in that landscape. (http://ccl.northwestern.edu/netlogo/models/BugHuntCamouflage.) (B) Evolution of computational biomorphs. Artificial flowers evolve through mating and blending their characteristics. (http://ccl.northwestern.edu/netlogo/models/SunflowerBiomorphs.)

to be generalizable enough to be used as the basis for other models that you are interested in developing on your own. For instance, for someone interested in exploring economics, this model could resemble a model of companies (the wolves) competing for consumers (the sheep) constrained by their budgets (grass). If a model author were interested in politics, then the model might be reminiscent of politicians (the wolves) competing for voters (the sheep), and the location of the agents could represent their feelings on particular issues. Of course, this model, in its present form, should not be directly used in these contexts—some of the mechanisms would need to be changed and the parameters or outputs of interest might be very different. These examples illustrate how the ABM methodology can be applied to a broad set of phenomena, and moreover that seemingly unrelated phenomena can be viewed as similar from an agent-based perspective as, even though the agents themselves might be quite different, they follow a similar set of rules.

Explorations

1. *Adding new parameters* When we added reproduction to the Wolf Sheep Simple model, we set two constants in the code. First, reproduction can occur only if the animal has more than 100 units of energy, and second, reproduction costs 100 units of energy. Create sliders for these parameters. How does varying these parameters affect the behavior of the model?
2. The Wolf Sheep Simple model only plots the counts of the animals, but the energy of the animals is almost as important as how many animals there are. Add a new plot to the model that plots the energy of the animals over time.
3. Right now all of the turtle agents in the Wolf Sheep Simple model move at the same speed. How would you expand the model so that they could move at different speeds?

How would you change the model so that the sheep and wolves could move at different speeds from each other? What effect would this have on the model? In particular, how does the ratio of the wolf speed to sheep speed affect the model dynamics?

4. The ability to move faster might come at the cost of a higher rate of energy expenditure. How would you change the model so that the speed of an agent affected its movement cost?

5. In this chapter we used the programmatic, as opposed to widget-based, method of updating the plots. However, what we did could be done using widget-based plots instead. Rewrite the model so plots are widget-based instead of programmatic/code-based.

6. In the Wolf Sheep Simple model, the wolves and sheep move randomly. In the real world, predators chase prey and prey try to escape. Can you modify the movement mechanism so the wolves chase the sheep? Again, how does the ratio of the wolf speed to sheep speed affect the model dynamics?

7. The Wolf Sheep Simple model explores two animal species interacting, but most ecologies have many more animal species. Create a third species in the model. One interesting way to do this would be to create a species that competes with the wolves for sheep, and can also eat wolves and be eaten by wolves. Can you get this three species ecosystem to stabilize?

8. Extend the Wolf Sheep Simple model by making the grass grow probabilistically, instead of at a constant rate.

9. The Wolf Sheep Simple model as written has the animals reproduce asexually. Modify the model so that it more realistically models the reproductive process. Possible avenues include: requiring two agents of the same breed to be on the same patch for reproduction to occur; giving the agents gender; and implementing a gestation period between when two agents are on the same patch and when the new agent is created. Does it change the dynamics of the model in any significant way?

10. We have talked a lot about designing models, but we have assumed so far that you know what the level of complexity of your model will be. Is there always an appropriate level of complexity of an agent-based model? Discuss your thoughts about the comparative benefits of making a simple model with the benefits of making a more elaborated and realistic model.

11. *Deterministic and random behavior* It is human nature to assume that there is a causal force behind everything that we observe, but some times this is not the case. However, even when we have a perfect understanding of a phenomenon, that is we know how all the underlying processes work, we might still want to use nondeterminism in our model. Can you explain some reasons why you would want to include random behavior in a model that could be completely deterministic? Is it possible to have deterministic macrobehavior even though the microbehavior is nondeterministic? Build a model of agents moving such that they always result in the same pattern even though they take random steps and random turns to get there.

12. *Designing a model* Assume that you have been asked by the city council to build an agent-based model of transportation patterns in the city. In particular they are interested in seeing where they should spend money with the goal of minimizing the time it takes for the average inhabitant to get to work. Design two models that would help them answer this question. In the first model, concentrate on the scale of a single neighborhood or political ward in the city. In the second model, concentrate on the level of the whole city. What are the agents in your model? Are they different in the two different scenarios? What data would you need if you were to actually build these models? How would you simulate the different policy choices, namely, the allocation of funds to minimize commuter time? What measures would you collect to answer the council's question?
13. *Evolution* In the Wolf Sheep model that we have built during this chapter, the wolf and sheep are exactly the same in every generation, but real wolves and sheep evolve over time. For instance, wolves might get faster or have better eyesight (though both of these traits might also incur a higher metabolism cost). Sheep might also get faster and might learn to avoid wolves (though again this might have a cost). Add an evolutionary mechanism to the wolf-sheep model. Does this make it more or less difficult to maintain a population?
14. *Modeling food webs* We mentioned how ABM can be used to describe food webs. However, these models are often written as an aggregate description. Imagine a model where instead of individual wolf and sheep there is simply a description of how wolf populations and sheep populations increase and decrease. In addition, imagine that there are many more species described, like the grass, insects, and birds. The description of how all these animals predate on each other is called a food web. Is this still an agent-based model? Please explain your answer.
15. *Crossing disciplines* At the end of the chapter, we speculated on how the Wolf Sheep Simple model might be construed as, or converted into, a model of companies moving around in a marketplace searching for resources. Develop this parallel and describe how you would use the Wolf Sheep Simple model as the basis for a model of economic competition.
16. *Building one model from another* Related to exploration 15, more generally, another way to build a model of a phenomenon you wish to model is to take a model that you already have that uses a similar mechanism to the model you want to build, and repurpose that model to transform it into a model of what you wish to model. Can you repurpose the Wolf Sheep Simple model along any of the lines suggested in the conclusion of the chapter?
17. Suppose a professional sports team approaches you. They are interested in understanding how their workers collect trash after a game. They want you to build a model where individuals pick up trash and then deposit it in garbage cans that are located around the stadium. This reminds you of the Termites model in the NetLogo models library. Modify

the Termites model to reflect this scenario, so that the termites are humans and they deposit the garbage (wood chips) in garbage cans.

18. *Building a simple model from a textual description* Now that you know how to build a simple model, can you build one that someone else has described? For instance, imagine you read the following in a scientific paper: "Agents are placed randomly throughout the world. Each agent has a status of either being healthy or sick. At the start of the model all agents are healthy except for one. Each time step agents move locally to a new random location. If, after they move, there are any agents that are sick nearby then they become sick." Build this model. Is it specified with sufficient detail that you believe your model is the same as the model built by the authors of the scientific paper?

19. *Types of agents* Look at the Termites model in the models library. In this model, the termites gather wood in piles. This model only has one type of termite, a wood-piler termite. Now imagine that there is a second type, or breed, of termite, a wood-unpiler termite. Add this second type to the model. How does this second type affect the results? Compare and contrast the patterns generated by the two models.

20. *Cyclic cellular automata* Create a cyclic cellular automaton. In a 2D cyclic CA, each cell can take on one of k states. But unlike a traditional CA, if a cell is in state i, it can only advance to state $i + 1$, and if state $i = k$, then it advances to state 0. A cell changes its state if some threshold of its neighbors is currently in the state that the cell is considering advancing to. Build this model. Change the number of states and the threshold how do these two parameters affect the resulting patterns of the model? How does changing the size of the world affect the model? Can you think of any real-world phenomena that might have similar behavior to a cyclic CA?

21. *Adding humans to the model* Take the Wolf Sheep Simple model that we have built in this chapter. Add humans to the model. Humans kill the wolves, but they do not gain energy from the wolves because they do not eat them. Humans will also kill the sheep, but they do gain energy from the sheep. How does the addition of humans affect the behavior of the model?

22. *Spatial location and grass* In the Wolf Sheep Simple model, the grass is randomly distributed across the space. However it is more realistic that areas of grass are spatially collocated, that is, areas with more grass are likely to be near each other, and areas low in grass are near each other. Change the SETUP and/or GO procedure for the grass to reflect this. How does this change affect the model? How does it affect the sheep?

23. *Changing mechanisms* In the Wolf Sheep Simple model, the grass grows linearly every time step an increment of grass is added to the patch. This is not realistic. Real grass will grow quicker if there is more of it around since there are more plants to produce seeds. Change the way grass grows so that it is dependent on the density of grass already present in the local area. Eventually the grass will hit a limit of physical space and be adversely affected by overcrowding and competition for water, nutrients, and sunlight. How would

you model that? How does this change in the grass mechanism affect the behavior of the model?

24. *Different types of distributions* In this model we used a mean and discussed how to use variance to characterize the results of multiple runs. This characterization assumes that the distribution of data can be described using these statistics. However, for some data, these statistics can be unhelpful or misleading. Describe a scenario where model output data are not well described by mean and variances. Why is it not possible to describe such a data set using these descriptive statistics? How would you describe this data set?

25. *ABM and OOP* Agent-based modeling shares many features with object-oriented programming (OOP). In some ways ABM and OOP are very different from each other. OOP describes a class of potential programming languages, whereas ABM also describes a perspective on thinking about the world. With that in mind, compare and contrast OOP and ABM. In what ways are agents like objects in OOP? How are agents different from objects?